iSim ESL

ESL Simulation Software

User Guide and Tutorial

Document Information
Version: 1.9.3
Date Published: March 2023

This document relates to ESL version 8.3.0

ISIM welcomes any suggestions to improve
the ESL Simulation Software and documentation.

If you have any suggestions, or would like to point out
any errors or omissions, please contact us:

ISIM International Simulation Limited
161 Claremont Road
Salford
M6 8PA
UK

Tel: +44 (0) 161-736-5283

Email: info@isimsimulation.com
Web: https://www.isimsimulation.com

Table of Contents

CHAPTER 1

Introduction

ESL is a powerful and flexible software tool used to simulate complex dynamic systems. It comprises the simulation language itself (ESL) and its interactive development environment (ESL-Studio).

This guide is not intended to be an exhaustive reference manual for ESL. Rather it aims to introduce the main features of the software through a series of exercises and provide enough information to get you started using both ESL-Studio and the ESL language. Detailed information on all topics introduced will be found in the on-line ESL-Studio and ESL Help Pages and on-line Documents, which you are encouraged to consult at each stage.

1.1 The ESL Language

ESL was originally written to meet the simulation requirements of the European Space Agency. It is a general-purpose Continuous System Simulation Language (CSSL) with discrete event capabilities and may be applied in any field where dynamic systems are to be studied.

The main characteristics of ESL are:

- Provision of an Interpreter for fast program development, and a Translator (providing C++ or Fortran code) for efficient production runs.
- A well-defined lexical structure.
- Separate program units may be used to describe the system and the experiment to be performed on it.
- Modular model concepts in the form of submodels to define independent parts of the system within a hierarchical structure.
- Parallel processor segmentation concepts to enable models to be partitioned into segments and executed concurrently in a multiple-processor environment.
- Techniques for the accurate description and detection of discontinuities.
- Steady state finding and linearization facilities.
- Full matrix/vector operations.
- Derivative notation, integral notation and transfer function notation for describing differential equations.
- Comprehensive run-time and post-run graphical display of results.
- Automatic ordering of the model definition equations.
- Eight numerical integration algorithms including three stiff methods.
- Extensive diagnostic checks during compilation to determine model "correctness".
- C++, C or FORTRAN routines may be incorporated into a simulation that has been created through the translator route.
- ESL segments may be run embedded in a non-ESL C++ or FORTRTAN main program.
- Facilities to dynamically communicate with other program modules via FORTRAN common blocks or C++ structures.
- Full range of standard procedural facilities including file and character handling.
- Extensive library of ESL submodels which may be incorporated into user programs.

1.2 ESL-Studio

ESL-Studio is an integrated development environment for creating ESL simulations using block diagrams and ESL source code. It is an alternative to, and replacement for, ESL's older Integrated Simulation Environment (ISE). It may be used with either ESL-Pro or ESL-Lite.

Using ESL-Studio's graphical user interface you can manage each stage of the simulation activity.

ESL-Studio provides the following facilities:

- Multi-window graphical block diagram editor for model construction.
- Inclusion of ESL coded submodels where appropriate.
- Interactive control of simulation execution (via the ESL-SEC program) with run-time graph plotting.
- Display manager with post-run graph plotting (via the ESL-Displays program).
- Sophisticated profile features allow themes for diagram appearance and for standard and library simulation entities.

ESL-Studio includes a graphical editor for block diagram style model descriptions, while allowing textual ESL code to be used where appropriate (for example, to describe highly non-linear elements). You select standard simulation elements and interconnect them on a block diagram to build up the simulation description. ESL submodels can be created and included in a diagram through a special submodel element.

Note: ESL-Studio can allow you to import legacy ESL ISE applications into ESL-Studio. To support this, you must include the ESL ISE component when you install ESL.

Once you have created a simulation program (graphically, textually or a combination of both), compilation is initiated from ESL-Studio. You may then execute the compiled program immediately through an interpreter, or, for ESL-Pro, you have the option to further translate it to C++ or FORTRAN. The resulting executable program may then be run from ESL-Studio. In either case, execution is managed by the ESL-SEC (Simulation Execution and Control) program which provides run-time control of the simulation. You have access to all program variables and parameters from the ESL-SEC program. This includes simulation parameters such as the communication interval, final simulation time, choice of integration algorithm and error tolerances. All variables and parameters can be set and changed dynamically. You can specify graphical and tabulated output on your block diagram using special simulation display elements or alternatively from a versatile display manager window. You can log all run time commands and output specifications to a driver file that can be used later to repeat simulation scenarios.

CHAPTER 2

A Simple Example

You will create a simple single-input, single-output feedback control system application. This will introduce the following features:

- layout of the ESL-Studio Integrated Development Environment (IDE)
- simulation elements
- use of the graphical editor to create a top-level block diagram
- setting simulation element properties
- use of the display elements
- running and interacting with the simulation
- post-run plotting

The example to be considered is the feedback control system shown in block diagram form in Figure 1:

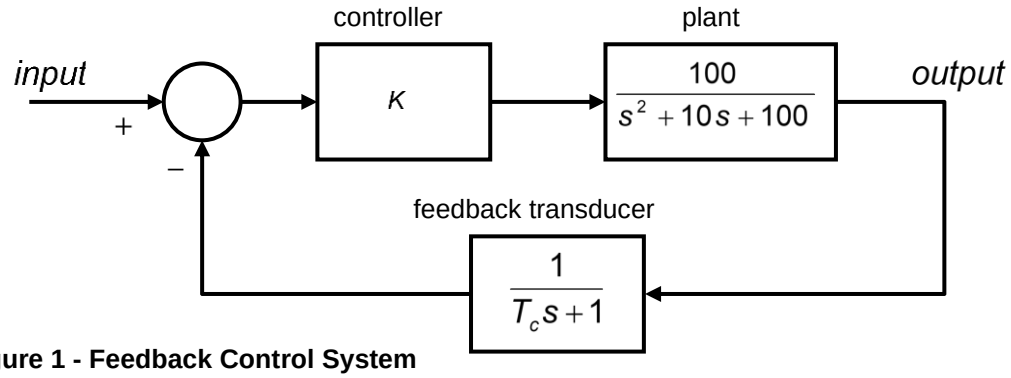

Figure 1 - Feedback Control System

We are interested in the response of the system for different types of input (step, sinusoidal, etc) while varying the values of the gain (K) and feedback time constant (T_c). Initially, K = 2.0 and T_c = 0.1 seconds.

2.1 Creating the Block Diagram

Start ESL-Studio - usually from the Start Menu. The appearance of the ESL-Studio IDE will be similar to Figure 2.

The central main view area can contain different views. They can be selected with tabs at the top of the main view area. These views are primarily used to represent or model the simulation, for example as block diagrams. By default, ESL-Studio creates a diagram view in the main view area to allow you to create a block diagram for your simulation model.

There are also secondary views or "panes" primarily used for information or editing. They may be docked in various locations round the main view area or floated free (as secondary windows). The main panes are:

- Toolbar – provides short cuts to most common menu selections.
- Application – displays the structure of the current application.
- Elements – lists the available simulation elements in a tree structure.
- Properties – displays the properties of a selected simulation element. If no element is selected, the top-level properties of the module are displayed. If the Help box is checked, the bottom section of the properties pane displays explanations and help for selected properties.
- Messages – where build information and error diagnostics are displayed.

Visibility of the above panes may be changed from the View menu.

ESL-Studio runs in one of two modes: Edit mode and Browsing mode. On start-up it is in Edit mode in which you create simulation models. When you run your simulation it is in Browsing mode in which you may inspect your simulation model but not edit it.

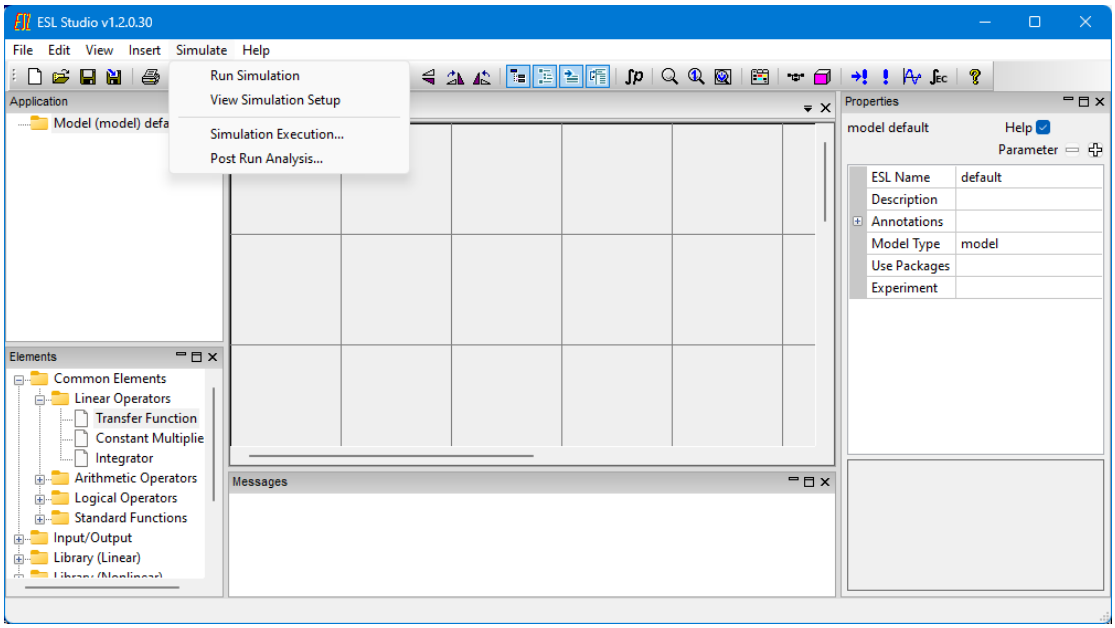

Figure 2 – ESL-Studio Main Window

By expanding Input/Output (and below) in the Elements pane, find and select a Step Input and drag it on to the diagram in the main view area. Similarly, under Common Elements, select Summer, Constant Multiplier and two Transfer Function elements and drag them to the diagram in the arrangement shown in Figure 3. (Alternatively, double clicking an element will place it at the centre of the main view area, from which it can be moved to the required position.)

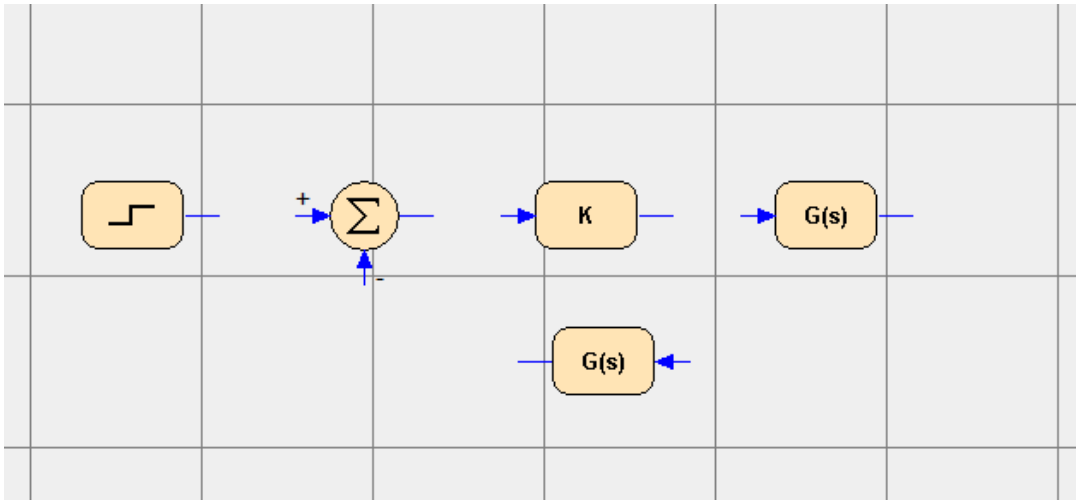

Figure 3 - Positioning Simulation Elements

Note that the orientation of the lower Transfer Function has been reversed by selecting a Left/Right flip from its context menu, which is displayed when you do a right mouse click on the element.

The simulation elements can now be interconnected as shown in Figure 4. To do this, select a simulation element connection port (at the end of its stem), with a left mouse click, extend the

signal line to the next port, and complete the connection with a further left mouse click. You will see a slight flash of the connection port when the signal line is properly connected to it, and the line will stay with the simulation element if you move it. In general, changes in direction of the signal lines are introduced automatically. However, intervening nodes may be created by additional left mouse clicks.

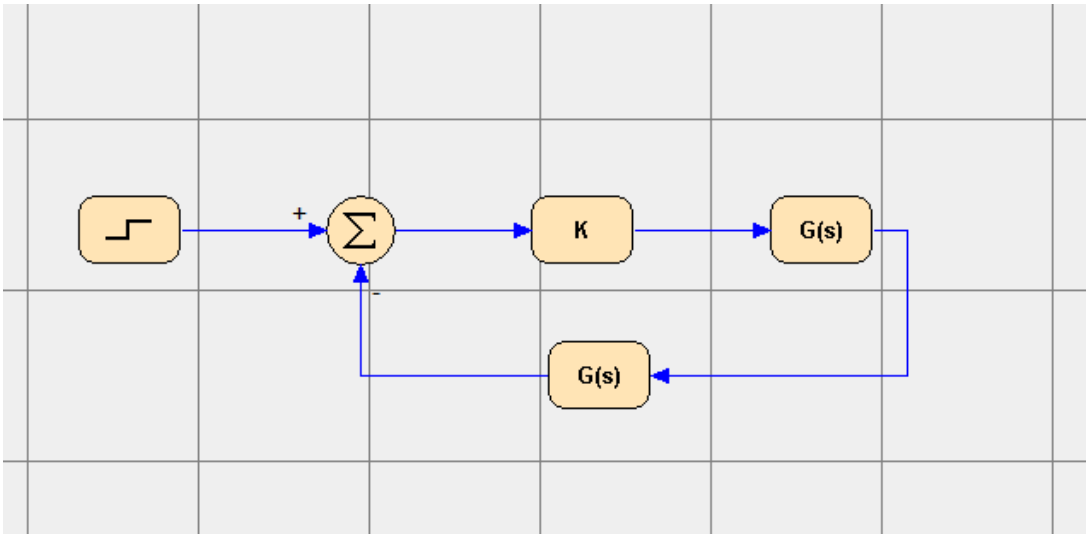

Figure 4 - Interconnecting Simulation Elements

Signal lines started in error can be abandoned by a right mouse click. Existing signal lines and nodes can be removed through their appropriate context menu option.

In general, if you make any mistakes editing in ESL-Studio, the most straightforward way to get back is to *Undo* those changes in the undo/redo stack. This can be done via the Edit>Undo menu, or by clicking the equivalent toolbar icon, or, in a diagram, the key combination Ctrl-Z.

2.2 Setting Properties

First, set a couple of global properties of the module. Click on a blank area of the diagram and examine the Properties pane (on the right-hand side of the screen in Figure 2). We can set the *ESL Name* and a *Description* e.g. *tutorial1* and *Feedback Control System*. Note that when a property is selected, a brief explanation of its function and use appears in the bottom section of the Properties pane. *ESL Name* will be used for the generated ESL code and *Description* is just that. Both these properties may be annotated by expanding *Annotation* and checking *ESL Name* and *Description*. This information will now appear in the top left of the diagram. Both pieces of text can be moved together by selecting and dragging (with the left mouse button down), or individually by double clicking one and dragging. Also, a double-clicked piece of text can have its size and colour changed from the Properties pane. Note that other properties displayed for the module are *Model Type*, which is initially set as *model*, *Use Packages* and *Experiment*. There will be more about these properties later.

We are also going to declare two Model Parameters for the gain (K) and time-constant Tc. Model Parameters are variables whose values can be interactively changed while the simulation is running. To create a new parameter, click the plus sign next to *Parameter* at the top of the Properties pane. Expand the parameter *Par*, change its *ESL Name* to *Gain* and its *Value* to *2*. In a similar manner, create a second Parameter, *Tc* with a value of *0.1*. The Properties pane should now look like Figure 5 (with items expanded, you may need to scroll in the pane to see all of it).

Note: *At this point, it is a good idea to save the application through the File>Save menu. Navigate to a suitable working directory and give the application a suitable name e.g., tutorial1 (here, we use the same name as given to the model's ESL Name) then it will be saved as tutorial1.eslstudio. You can save the application from time to time during its development and, anyway, will be prompted to save it if changes have been made when you exit ESL-Studio.*

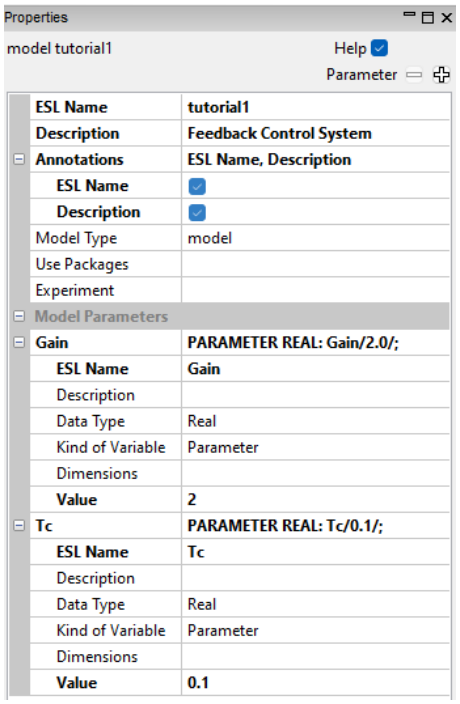

Figure 5 - Model Properties

The next stage is to set the simulation element properties. You can select any of the simulation elements in the diagram by a left click. Its properties (which generally include settable attributes) are displayed in the Properties pane.

2.2.1 Step Input Properties

The Step Input Properties are shown in Figure 6 (left figure). Note that some properties are fixed – for information only and some can be set or changed. In this case the top-level properties are:

- Type the type of simulation element
- Summary a summary of the functionality of the element
- Help a link to ESL-Studio Help Pages for the element - double click in the row to open the link in your browser
- View where applicable double click in the row to see the underlying ESL submodel code in an ESL view in the main view area
- Description allows a description of the *particular* element to be added
- Annotations annotation text(s) to be added to the element on the diagram
- Attributes attributes specific to the particular type of simulation element
- Ports information about the element's input and output ports

Note that when any of these properties are selected, a summary is displayed in the help area of the Properties pane (this facility can be disabled by unchecking *Help* at the top of the pane).

In our case, set *Description* to *step input* and check *Description* under *Annotations*. The attributes of the Step Input simulation element are *Amplitude* and *Time Delay*. We will leave these with their default values of 1.0 and 0.0.

Under *Ports* there is just one port – *Port 1 (output)* and this is something we want to modify. Expand *Port 1(output)* to reveal:

- Tag port identifier
- Data Type data-type for the value associated with the port

- ESL Name a unique (within the scope of the module) name – this will be the variable name in the generated ESL code
- Description a description of the particular simulation element port
- Annotations specifies which port properties to be annotated on the diagram

Only *ESL Name*, *Description* and *Annotations* can be set/changed. Set the *ESL Name* to *in* and select *ESL Name* for annotation. The appearance of the Step Input Properties pane should now be as in Figure 6 (right figure).

Note that the Step Input annotations now appear on the block diagram.

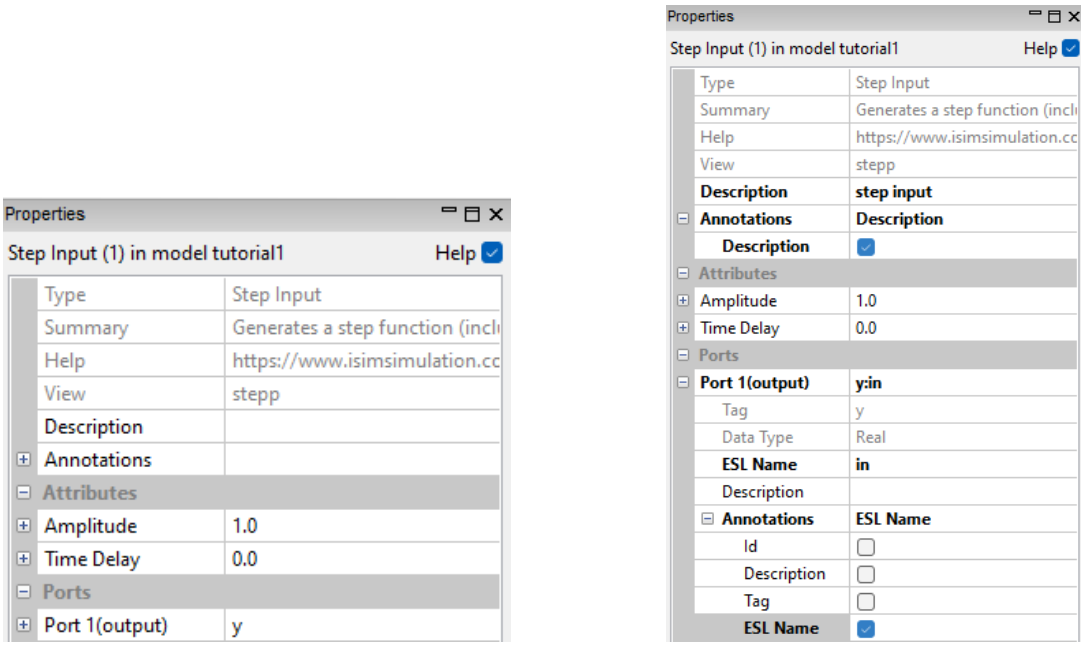

Figure 6 - Step Input Properties

2.2.2 Summer Properties

Change the ESL Name of the output port (port 3) to *error* and set the annotation as shown in Figure 7

Note: *It is not necessary to change the ESL Names of simulation element outputs – the original generated name can be left unaltered. However, adopting meaningful names will make both the block diagram and generated code more easily readable.*

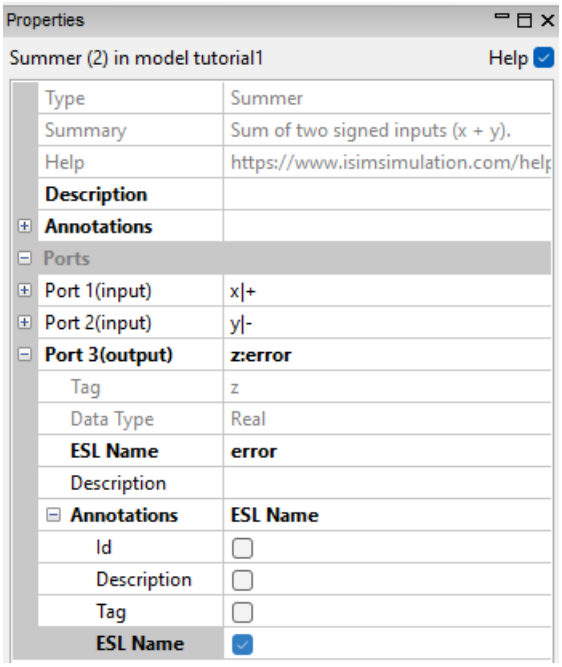

Figure 7 - Summer

2.2.3 Constant Multiplier Properties

Select the Constant Multiplier, set its description to *controller* and check Description under Annotations. We want to associate the Constant Multiplier *Coefficient* attribute with the model parameter *Gain*. Expand Coefficient and set its ESL Name to *K*. Select Source and in the drop-down box select *Parameter*. This will set the Value attribute to *Gain*, the first compatible model parameter.

Note: *The drop-down box next to Gain would allow a different parameter to be selected.*

Under Annotations check *ESL Name* and Value. Note that this will show the numerical value of the parameter *Gain* on the block diagram.

We can also set the ESL Name of output port (Port 2) to *u* and annotate this. The appearance of the Constant Multiplier properties should now be as in Figure 8.

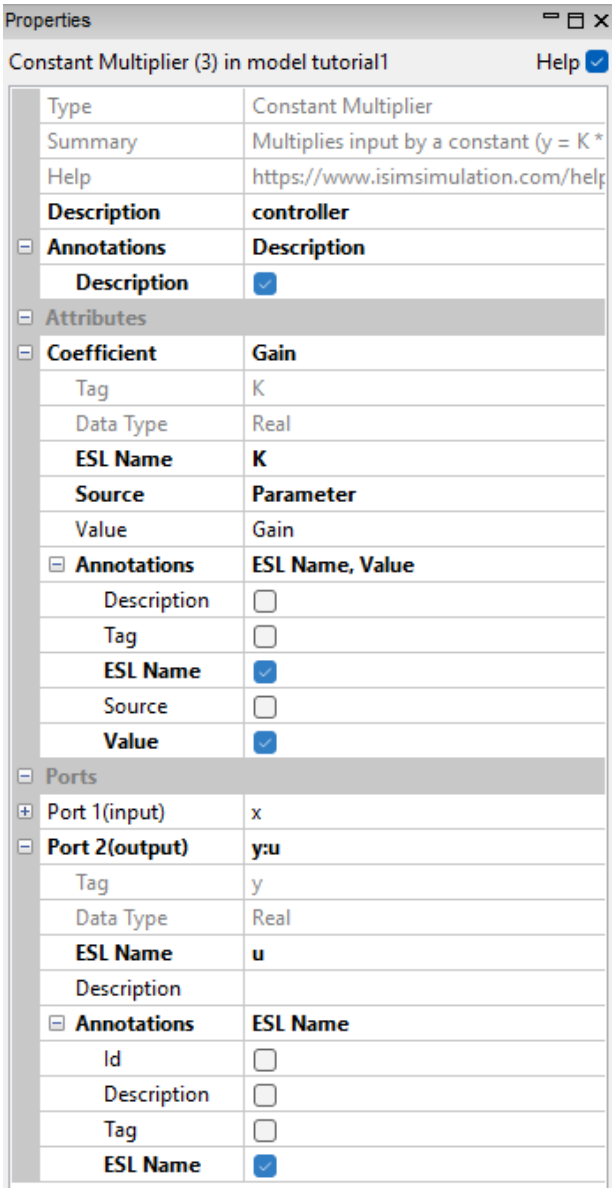

Figure 8 - Constant Multiplier Properties

2.2.4 Transfer Function Properties

Select the Transfer Function in the forward path (the one not flipped in Figure 3). This represents the plant being controlled therefore set its description *plant* and annotate. Expand the *Transfer Function* Attribute and type in its *Value* as *100/(s**2+10*s+100)* and annotate the *Value*.

Note: *Note the use of ****** for exponentiation and ***** for multiplication. See section 5.1.4 of the Development Guide for a comprehensive definition of transfer function syntax. The syntax of the transfer function syntax is checked as it is entered – errors are indicated by an audio chime and error message in the Messages pane.*

We can also set the *ESL Name* of the output port to *out (and annotate it)*.

Select the feedback path Transfer Function; give it the *Description sensor*; enter its value as *1/(Tc*s+1)* and set its output port *ESL Name feedback*.

See Figure 9 and Figure 10 for the Transfer Function properties and the current appearance of the block diagram.

You may adjust the position of an annotation on the diagram by double clicking to select it and then dragging.

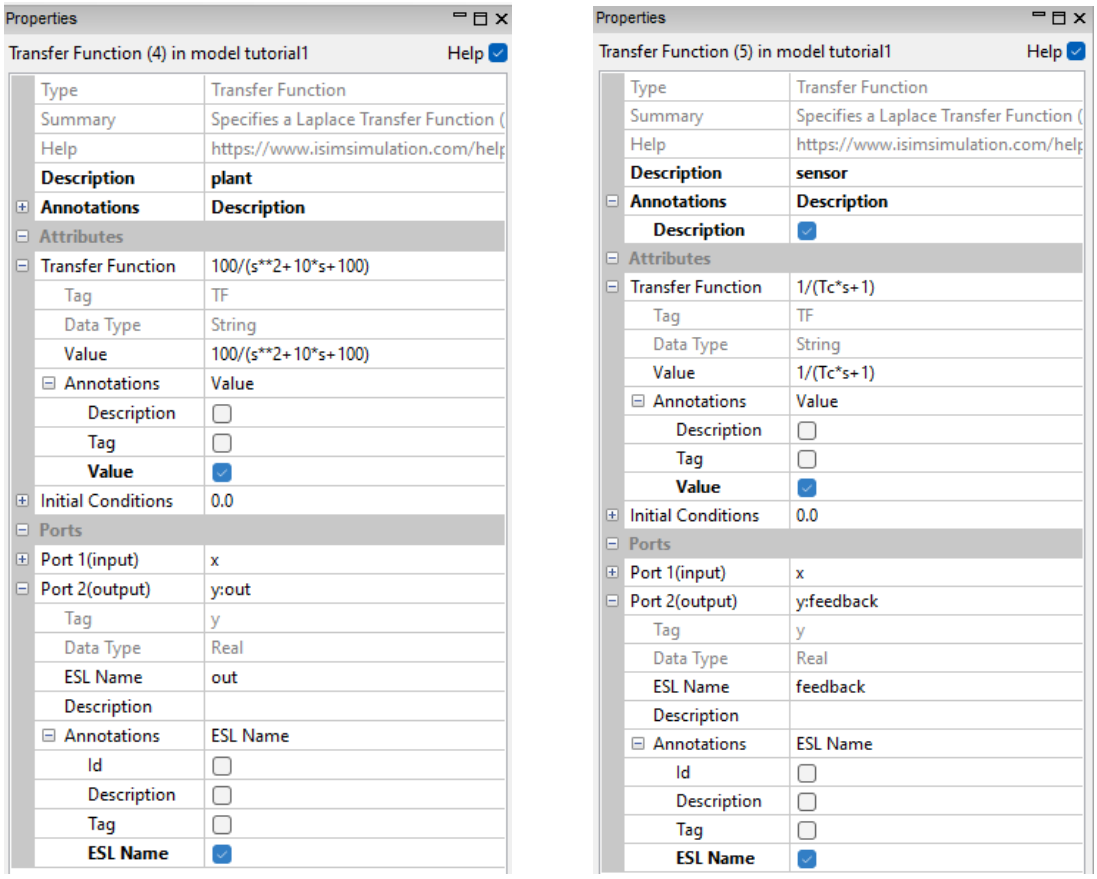

Figure 9 - Transfer Function Properties

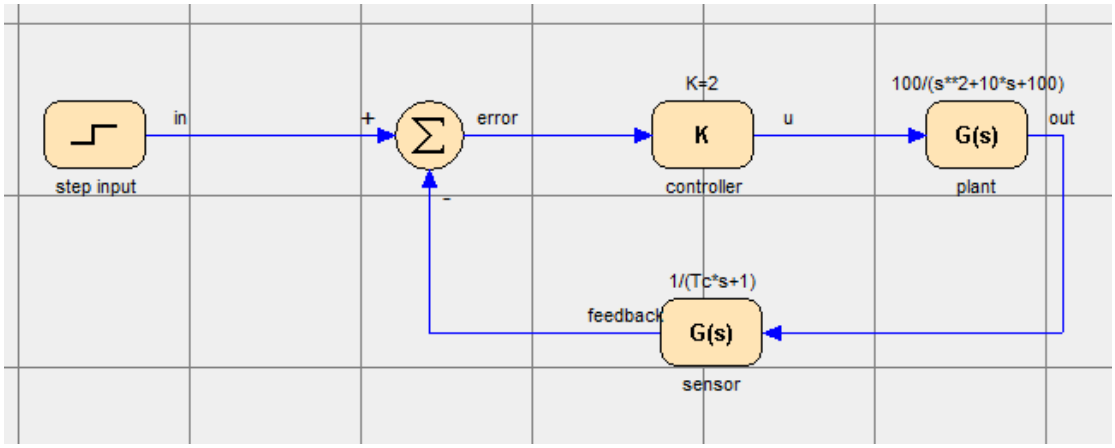

Figure 10 - Block Diagram so far

2.3 Specifying Output

There are two main ways of specifying output from an ESL-Studio application's simulation:

- by placing *Display Icons* on the diagram and connecting them to the outputs to be displayed
- using the *Runtime Displays* from the *Simulation Execution Control* window (see section 2.6)

The main difference is that Display Icons are placed and connected during the editing phase and therefore become part of the block diagram, whereas the Runtime Displays lets you specify *and change* output specifications at run-time. Note that the Runtime Displays is the only way of specifying output interactively when running an external ESL program i.e., an ESL program created textually, running from ESL-Studio – see Chapter 4. We will use Display Icons.

Expand *Display Icons* in the *Elements* pane and drag a *Plot* element and a *Table* element onto the main view area. Connect both the *Plot* and *Table* icons to the outputs of the Step Input and the Plant Transfer Function simulation elements. To start an instrumentation line, left click near the centre of the display icon. Click on an output connection port to make the join. You will see the flash of the connection port when the instrumentation line attaches properly. The diagram should now have the appearance of Figure 11.

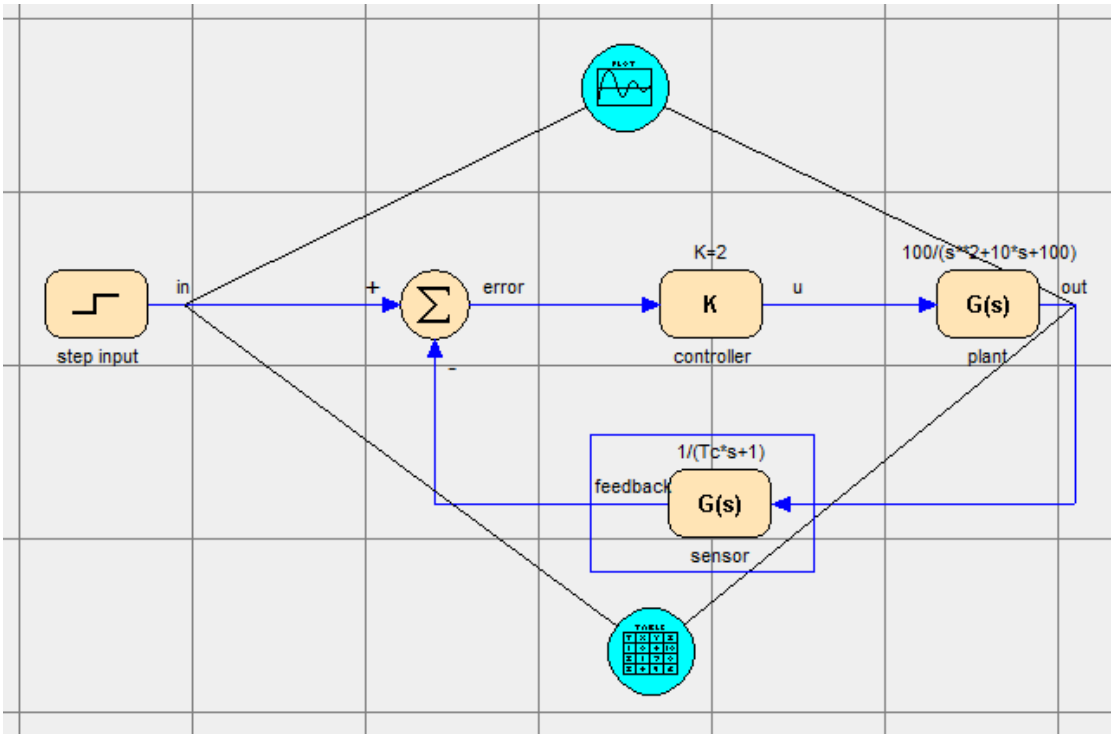

Figure 11 - Display Icons connected

Display Icon Properties allow the specification of *Title*, *Subtitle* and *Update* (Communication Points, Communication Points and Discontinuities or Step Points). In the case of a Plot icon, the *Plot Style* and *X and Y-Axis* details can be set. In the case of a Table icon, the *Output* (Window, Tab file or CSV file) and *Table Style* (Trend or Monitor) can be set.

Generally, Step points are best for graphical output and Communication points for tabulated output.

Figure 12 shows suitable property settings for the Plot and Table icons in our case. It is suggested that once you have the simulation running you experiment with different display settings.

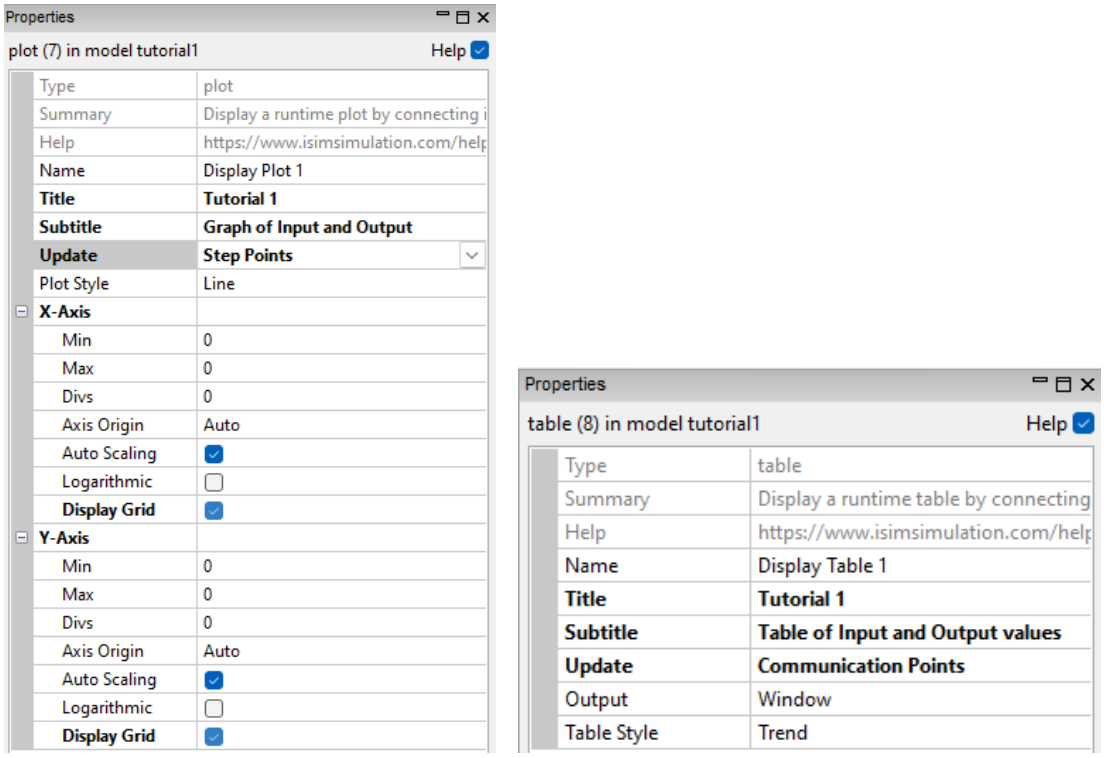

Figure 12 - Display Icon Properties

2.4 Simulation Parameters

Before running a simulation, you should review and, if necessary, change the *Simulation Parameters* i.e., the parameters that specify how the simulation is to be run. Check *View Simulation Parameters* on the *View* menu. Notice that this will open a second view in the main view area.

These parameters, with their default values shown in parenthesis, are listed below:

- TSTART - initial value of T at start of run (0.0)
- TFIN - final value of T at end-of-run (10.0)
- CINT - communication interval (1.0)
- DISERR - discontinuity detection error tolerance (0.0001)
- INTERR - integration error tolerance (0.001)
- ALGO - integration algorithm (RK5).
- NSTEP - number of integration steps in CINT(1)

Accept the default values except *NSTEP* which you should change to 10 (to give smoother graphs) as in Figure 13.

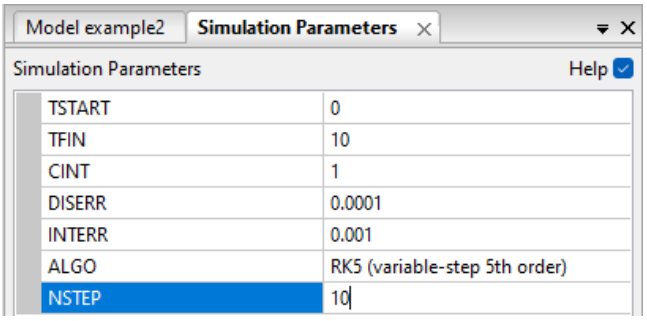

Figure 13 - Simulation Parameters

2.5 Simulation Setup

Select *View Simulation setup* from the *Simulate* menu. This will open a third view in the main view area. Here is where data about the simulation build and execution are specified. Check *View generated ESL* so that you will be able to see the ESL code that will be generated from the block diagram. *Execution Command* lets you specify *interpret* or *translate* options. In this instance leave *Compile and Interpret* selected. This is shown in Figure 14

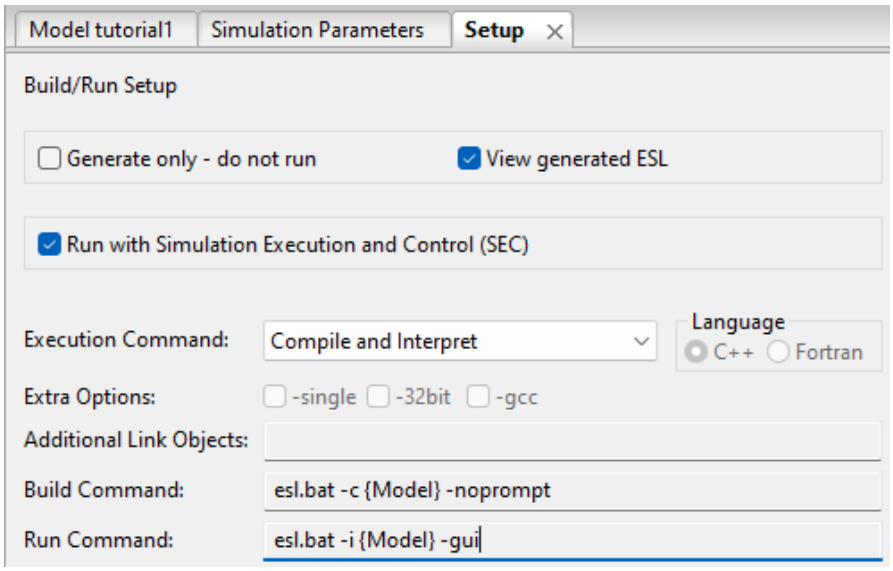

Figure 14 - Simulation Setup

2.6 Running the Simulation

You have now created a block diagram representing the system to be simulated, entered all the relevant data and are in a position to run the simulation. This is initiated from *Run Simulation* on the *Simulate* menu. You will get a dialog saying "*Application has changed – will be saved Do you want to continue*", click Yes. Assuming there are no errors, the *Simulation Execution Control* window of the ESL-SEC program will be opened – Figure 15 and also a *Plot* window and a *Trend* window corresponding to the two display icons. A fourth view will also be opened in the main view area displaying the generated ESL code. A summary of the successful compilation will appear in the *Messages* pane. You may need to move the windows about to achieve a good layout.

Simulation Execution Control is where the running of the simulation is controlled. It also gives you access to *Simulation Parameters, Variables, Runtime Displays* and *Advanced Simulation Options*.

Click *Start* to run the simulation. The appearance of ESL-Studio should now be similar to Figure 16.

ESL-Studio stays in Browsing mode until you exit running the simulation. You may change views by clicking on "tab"s in the main view area, and generally navigate about and inspect the application, without being allowed to change it.

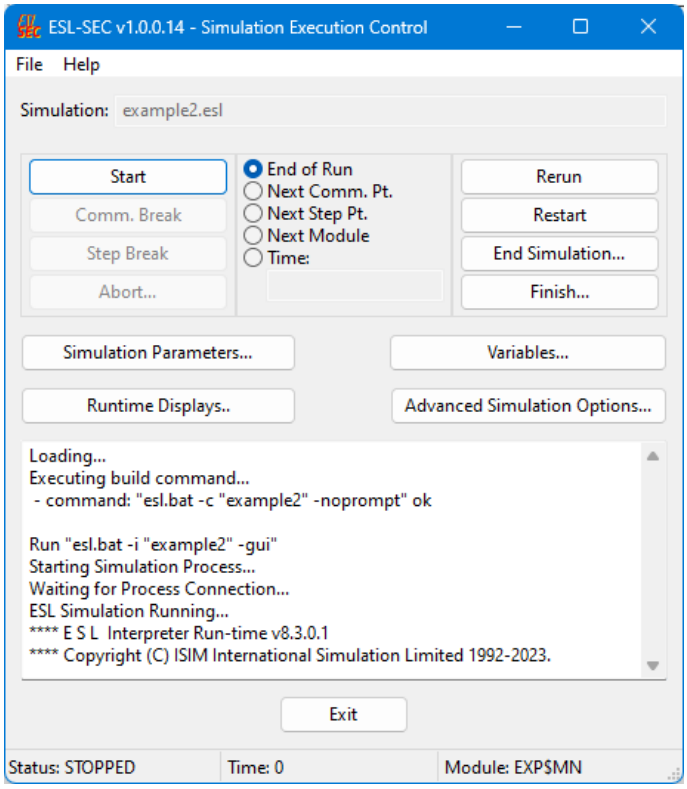

Figure 15 - Simulation Execution Control

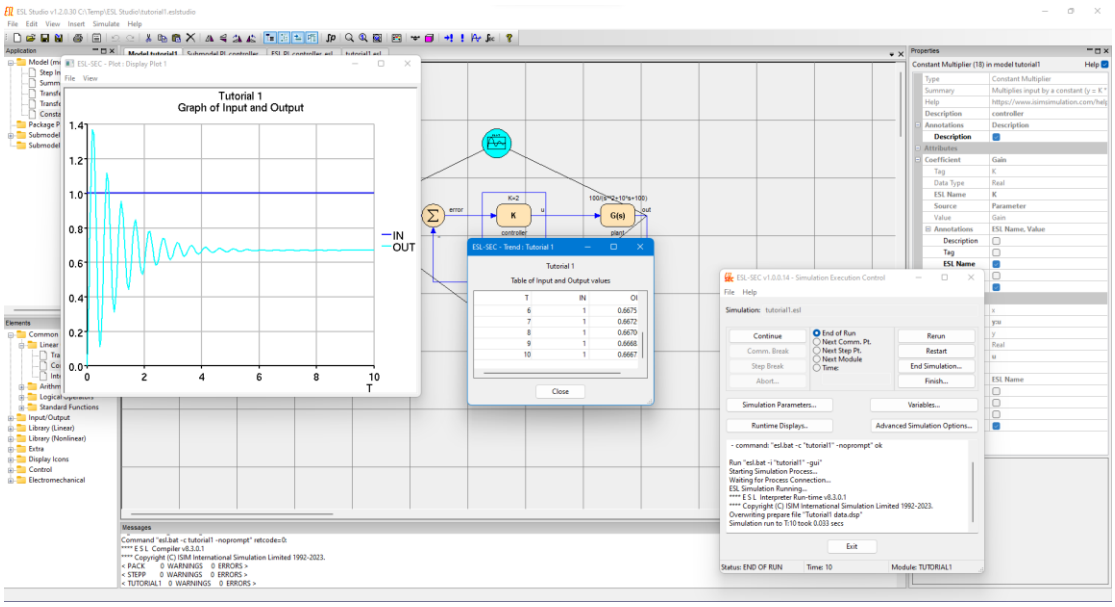

Figure 16 - Results of running the Simulation

2.7 Varying Parameter Values

Having made one run of the simulation, you will typically want to investigate the effect of varying one or more parameter values. In ESL-Studio, any model parameter may have its value changed at run-time. From *Simulation Execution Control*, click the *Variables* button to open the Variables window. This gives you access to all parameters and variables in all simulation modules.

Select *TUTORIAL1* (the name of the model in this case) from the *Modules* list and *GAIN* from the *Variables* list. Note the details of *GAIN* displayed, change the value from 2 to 2.5 (say) and click Apply Figure 17 (left figure). To re-run the model, click *Rerun* followed by *Continue* in *Simulation Execution Control*. A second graph will appear in the plot window as in Figure 17 (right figure). Notice also a second set of values in the *Trend* window Note the reduced steady-state error but increased oscillation caused by the increase in gain.

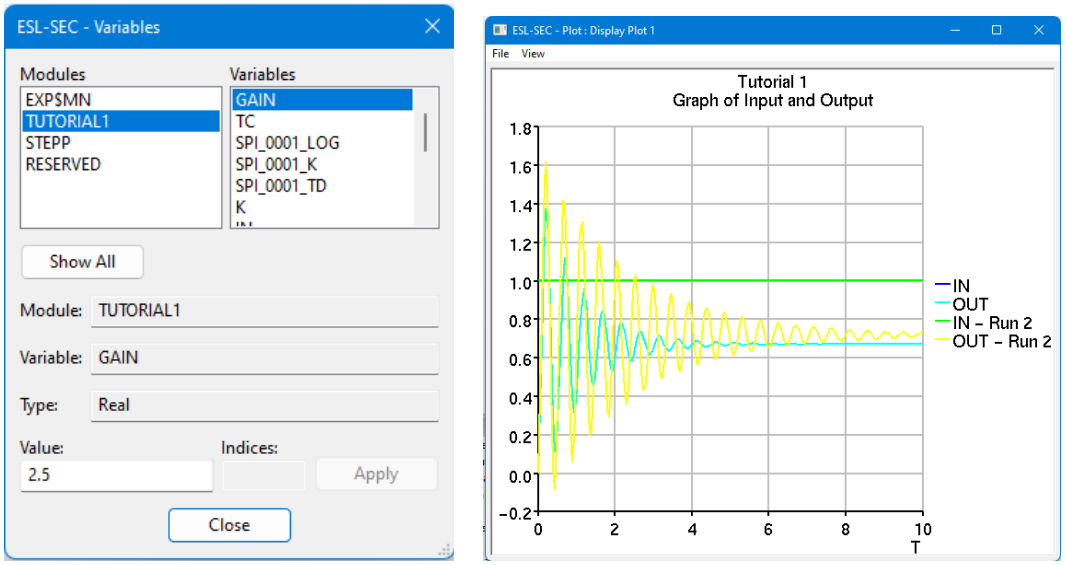

Figure 17 - Variables dialog and new graph

You should try varying the gain further or changing the value of the feedback time constant parameter Tc in a similar manner. Click *Rerun* followed by *Continue* in *Simulation Execution Control* to initiate each new run.

On completion of running the model, exit running the simulation by clicking the Exit button in *Simulation Execution Control*; click *Yes* to the message confirming that the simulation will terminate and click *Save* to the *Specification Changed* message (if you moved the displays).

2.8 Using Runtime Displays

As stated at the start of section 2.3, an alternative way of specifying output is through *Runtime Displays*, which can be opened from *Simulation Execution Control*. Run the previous example (or go to *Simulation Execution Control* if the simulation is still open) and click the *Runtime Displays* button. The appearance should be as Figure 18. Under the *Plot* tab you will see the specification of *Display Plot 1*. Similarly, *Display Table 1* will be seen under the *Table* tab. Notice the text *Icon Display* under the *Modules* panel indicating this output was generated from *Display Icons*. To specify a new runtime plot, go back to the *Plot* tab and click *New*. Accept the default name, *Plot 2*, or set a different name. Select a Module from the left-hand panel (*TUTORIAL1* is the model in this case) and select a *Variable* from the right-hand panel, for example *ERROR*. Click *Add* (or double click the variable) to add it to the *Contents* panel. Note that the dependent variable is pre-set as (*RESERVED*) *T*, but this can be changed using the buttons on the right. Properties of the plot can be set from the *Properties* button (as with the Display Icon settings). For instance, you may set an appropriate title and subtitle and perhaps specify grid lines. Finally click Show Display to open a plot window. In a

similar manner a new table can be specified under the Table tab. Close the Runtime Displays window and start the simulation by clicking *Continue* (or *Restart* followed by *Continue* if *Simulation Execution Control* was still open). The new plot display should be as Figure 19.

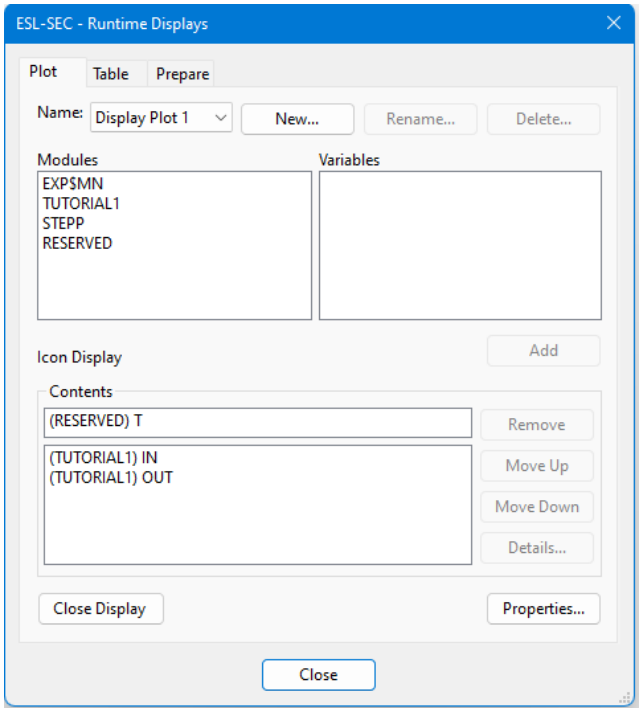

Figure 18 - Runtime Displays

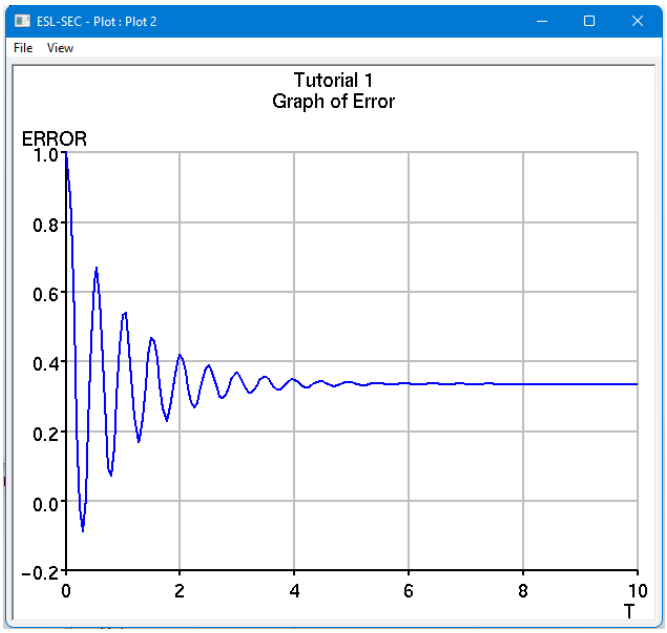

Figure 19 - Runtime Display Plot 2

2.9 Offline Display Analysis

The third tab in the *Runtime Displays* window is *Prepare*. This feature allows you to direct output to a prepare file (.dsp) for post run analysis using the *ESL-Displays* program. You set up a prepare file in a similar manner to Plots and Tables. Select the *Prepare* tab in *Runtime Displays* and choose the variables to be saved (Figure 20). From Properties you can set a title; subtitle; frequency of output and a filename (Figure 21). Click *Create Display*. When the simulation is run, the prepare file will be created. If you rerun the simulation, for example with different model parameter values, the data for multiple runs will be collected in the prepare file.

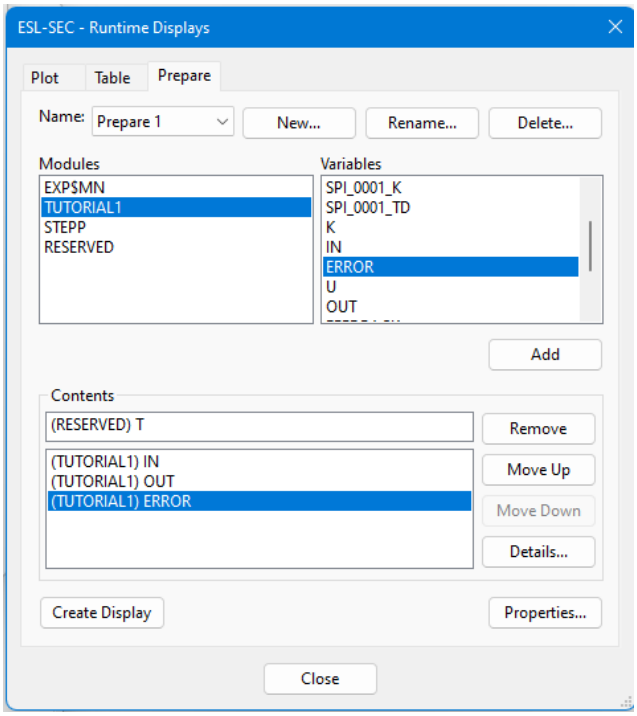

Figure 20 - Prepare file Specification

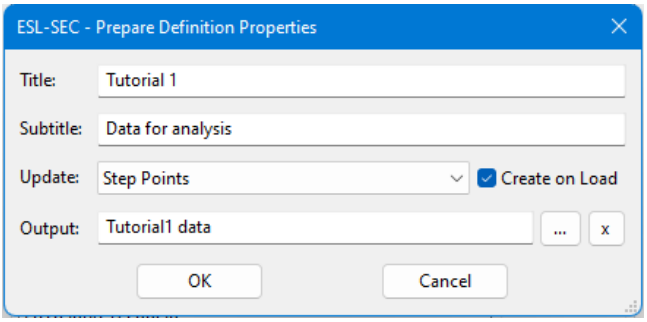

Figure 21 - Prepare Properties

Any prepare files that have been created can be accessed from *Post Run Analysis…* off the *Simulate* menu. This opens *ESL-Displays* (Figure 22). You should exit the simulation first.

Note: *ESL-Displays can also be started from a command prompt (terminal) – esl_displays – useful for analysing data outside the ESL-Studio environment.*

From the *Load* button you can load one or more prepare files. The file of interest is selected from the *Display Files* panel (*Tutorial 1 data.dsp* in this case) and variables for plotting from the *Variables* panel (in a similar manner to Runtime Displays). Variables may be selected from more than one file allowing, for example, comparisons to be made from different runs of the application simulation or other (related) simulations. The *Export* button allows a prepare file to be converted and saved as a readable Tab file. Properties are set as in Runtime Displays and clicking *Plot* generates the display (Figure 23).

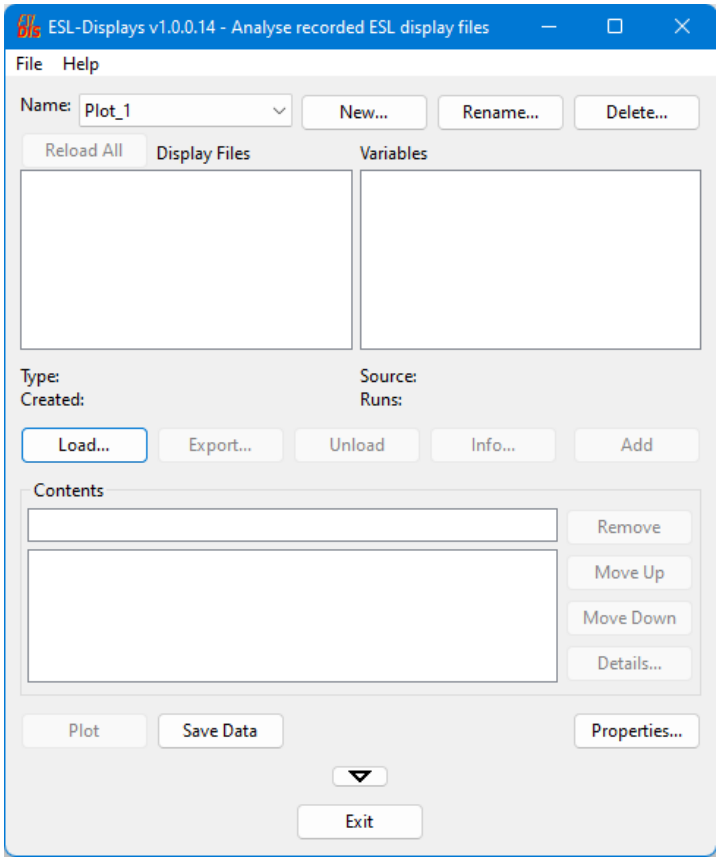

Figure 22 - ESL-Displays

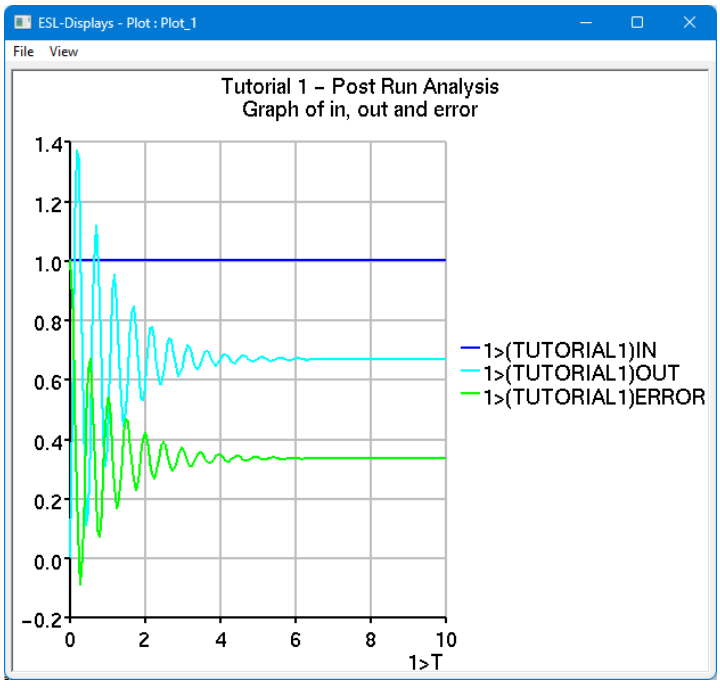

Figure 23 - Prepare file display

2.10 Further Exercises

Try replacing the Step Input simulation element with:

- a Sinusoidal Input
- a Ramp Input or
- a Square-Wave Input

and compare the outputs.

CHAPTER 3

Extending the Example - Graphical Submodels

In this chapter you will be shown how to extend the example that you created in the previous chapter. This will introduce:

- graphically defined submodels

The simple control system example introduced in Chapter 2 used, in effect, a "proportional" control law (represented by the *Constant Multiplier* element in the forward path. Suppose you wanted to use a "proportional plus integral" control law. This could be achieved by replacing the Constant Multiplier with a submodel.

Note: *There is in fact a standard PI control element to be found on the Library (Linear) branch of the Elements pane. Building one from first principles is simply being used to illustrate the general procedure for creating submodels.*

3.1 Defining a Graphical Submodel

First create two new *Parameters: Kp* and *Ki* in a similar manner to *Gain* and *Tc* in section 2.2. Give *Kp* a value of *2.0* and leave *Ki* with a value of zero (Figure 24).

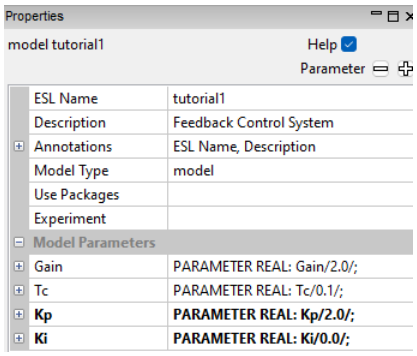

Figure 24 - New Parameters

Select *Insert Submodel Diagram* from the *Insert* menu. This will open a new view – *Submodel Sub* in the main view area. Change the *ESL Name* to *PI_controller* (say) and set the *Description* to *PI Controller*. Annotate these two attributes. Leave *ESL Type* as *submodel* (Figure 25) Note the alternative *ESL Types* are *segment* and *external segment* – more on these later.

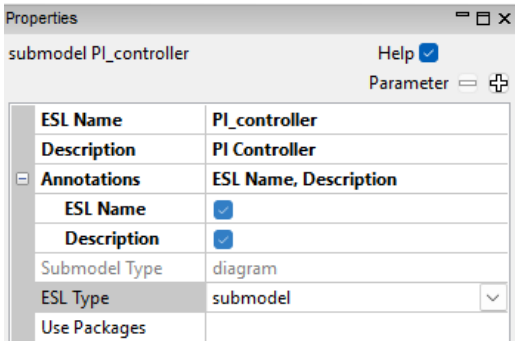

Figure 25 - Submodel Properties

The control law to be represented by the submodel is:

$$y = Kp \times x + Ki \times \int x$$

where *x* is the input error signal, *y* is the output actuation signal to the plant and *Kp* and *Ki* are parameters.

Drag three *Real Input Argument* elements and one *Real Output Argument* element from the Input/Output branch of the Elements pane into the submodel view (in the main view area) for the submodel's inputs and output. Build the rest of the submodel using simulation elements from the *Common Elements* branch as shown in Figure 26. Note the sign change of one of the *Summer* input ports – by double clicking it or changing it in its Properties.

View the properties of the *Input* and *Output Arguments* elements. Set the value of *Tag-name for input* values for the three inputs to *Ki*, *x* and *Kp* respectively. In the case of *Ki* and *Kp*, check the value of *Attribute*. This will make these two inputs into attributes of the submodel rather than wired inputs. Set the *Tag-name for output* value to *y*. See example of Properties for *Input Argument Ki* in Figure 27.

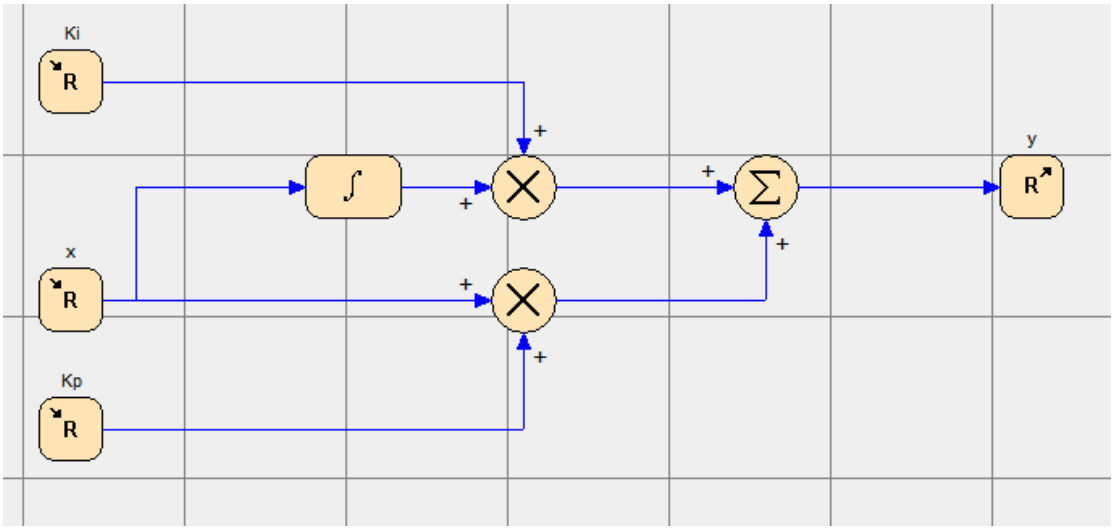

Figure 26 - Graphical PI Controller submodel

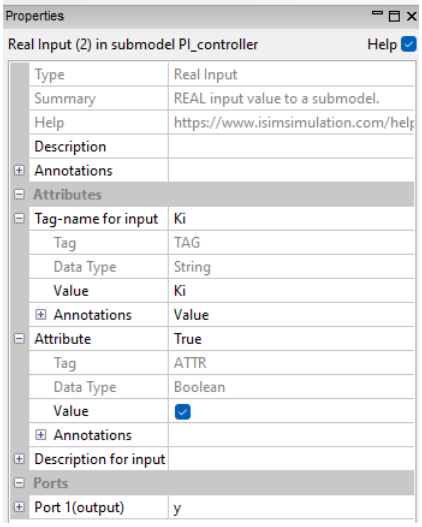

Figure 27 - Attributes of Real Input set as an Attribute

Return to the *Model tutorial1* view and delete the *Constant Multiplier* and drag a *Submodel* element from the *Extra* branch of *Elements* in its place. From the *Submodel* attribute drop-down list select *PI_controller* (that is the one you have just constructed). Annotate *Submodel*. Under the *Ki* attribute, select *Parameter* as *Source* and choose *Ki* as value. Choose to

annotate *Source* and *Value*. Repeat for the *Kp* attribute. The *Submodel Parameters* should now be as Figure 28.

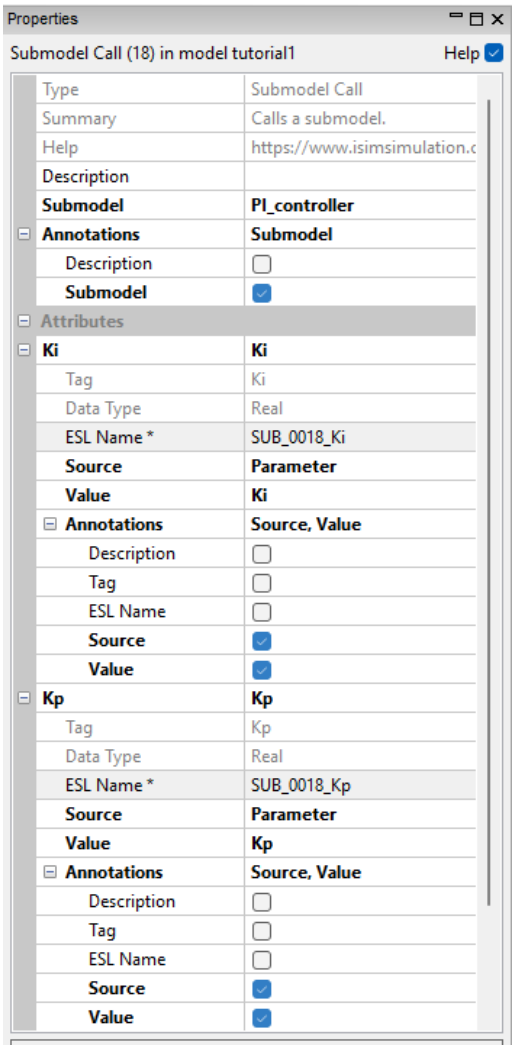

Figure 28 - Submodel Parameters

Connect the *PI_controller* between the *Summer* and *Plant Transfer Function* as in Figure 29. The modified model with a graphical submodel is now complete and you could *Save As* with a different name.

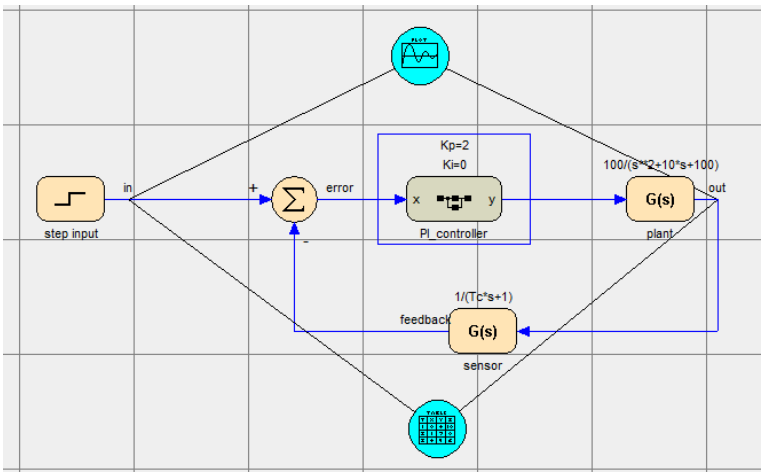

Figure 29 - Simple example with PI Controller

3.2 Running the Modified Model

Run the simulation from the *Simulate>Run Simulation* menu as before and click *Start* in *Simulation Execution Control*. Since the proportional gain, *Kp* has a value of *2.0* and the integral gain, *Ki* has a value of zero, the results should be identical to the previous model. Try changing *Kp* and *Ki* to different values from the *Variables* button e.g., *Kp=1.0* and *Ki=0.5*. You should then see the effect of the integral control action (Figure 30).

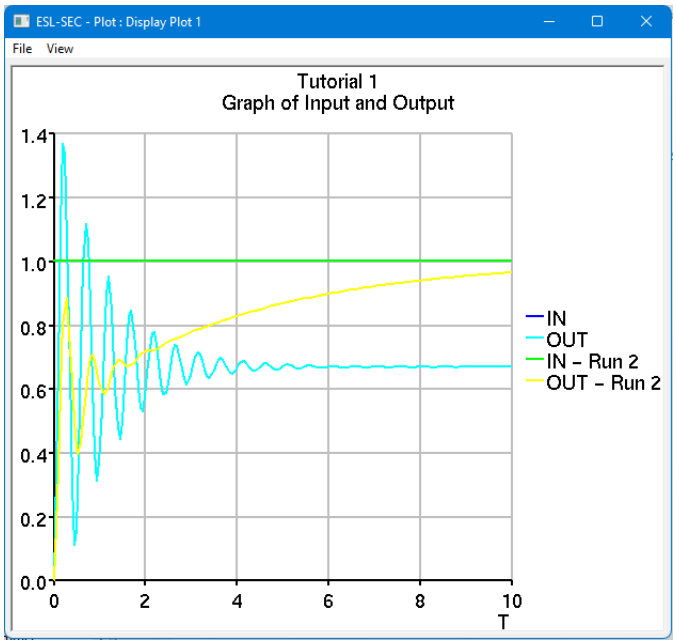

Figure 30 - Output with Kp=1.0 and Ki=0.5.

It is important to understand that the *PI_controller* submodel element on the main diagram represents an "instance", or call of the submodel not the submodel itself. That is, further Submodel elements can be dragged onto the diagram and have their definitions set to *PI_controller*.

CHAPTER 4

Extending the Example - Textual Submodels

ESL-Studio supports both graphical and textual methods of system description. In this section you will replace the graphically defined controller with a textually defined submodel. This will introduce:

- use of the text editor
- an introduction to the ESL language

4.1 Inserting a Textual Submodel

To illustrate the use of textual submodels, you will replace the graphically defined PI Controller created in the previous section with the equivalent written directly in the ESL language.

First delete the instance of PI_controller in the Model view by a right mouse click and choosing Delete from the context menu.

Note: *Note that the submodel itself remains available and can be assigned to a submodel element at any time. To remove the submodel altogether you should close the submodel tab and select the option to remove the submodel from the application.*

Select *Textual Submodel* from the *Insert* menu and select the *ESL option*. You should see:

```
SUBMODEL Txt();
DYNAMIC
END;
```

Note: *The alternative, File option, would allow you to navigate to a pre-existing file which contains the text of a submodel, the text will be displayed for viewing in the window – it cannot be edited. When ESL code is generated from the application for this case, an "Include" statement will be generated which provides a link to the specified file.*

Edit or copy the code, replacing the default text, to create the controller submodel as below.

```
-- Embedded Text Submodel

SUBMODEL PI_controller_esl (REAL:y := CONSTANT REAL:Kp,Ki; REAL:x);
  REAL:int_x;
INITIAL
  int_x := 0.0;
DYNAMIC
  int_x' := x;
  y := Kp*x + Ki*int_x;
END PI_controller_esl ;
```

Note: *The syntax of the SUBMODEL declaration statement is checked when the text is committed into the application. Errors are indicated by an error message in the Messages panel. The submodel name cannot be "PI_controller" if the previous graphical submodel with that name from Chapter 3 has not been deleted from the application."*

The lines of code are explained below:

`-- Embedded Text Submodel`	a comment, all lines commencing with -- are treated as such

`SUBMODEL PI_controller_esl (REAL:y := CONSTANT REAL:Kp,Ki; REAL:x);`	SUBMODEL definition statement defining inputs and outputs; CONSTANT means Kp and Ki are to be treated as attributes of the submodel
`  REAL:int_x;`	declares a variable to represent the integral of x
`INITIAL`	start of INITIAL region
`  int_x := 0.0;`	initialise integral variable
`DYNAMIC`	start of DYNAMIC region
`  int_x' := x;`	differential equation to integrate x
`  y := Kp*x + Ki*int_x;`	control law
`END PI_Controller_esl ;`	submodel END statement

This simple example illustrates the basic structure of an ESL model or submodel. Further detail of the ESL language is given in Chapter 5.

Return to the *Model* view and locate a *Submodel* element on the diagram between the *Summer* and *Plant Transfer Function*, for example from the diagram background context menu. Set the *Submodel* attribute to *PI_controller_esl* and set the remaining attributes as for the graphical submodel. Connect the controller submodel to the *Summer* and *Plant Transfer Function*.

The program can be run in the same manner as before.

The advantage of being able to create textual submodels is that often some parts of a simulation are more easily described in terms of equations rather than by a graphical block diagram. It may be the model description has been provided in equation form and it makes sense to enter it as-is rather than convert it to a diagram. Also, parts of a system that are highly non-linear, particularly if they contain discontinuities, are more naturally described textually. ESL-Studio gives you the ability to combine graphical and textual system descriptions.

CHAPTER 5

The ESL Language

The heart of an ESL simulation is the actual ESL program. The ESL language is extensive and is described in detail in the on-line documentation. In this chapter you will be introduced to the main features of the language through examples.

An extensive set of ESL examples will be found in the ESL installation *…esl\examples* directory, some of which are referred to in this document. It is suggested that you copy these examples to a suitable working directory.

5.1 Program Structure

A standard ESL program is termed a Study and contains several different kinds of program module, as shown below:

```
Study

  <packages>
  <procedures>
  <submodels>
  <model>
  <experiment>

End_study
```

A study normally contains a single model and experiment and, optionally, one or more packages, procedures and submodels. Packages, procedures and submodels can appear in any order, provided each is defined before it is referenced.

For more advanced use, a study may contain more than one model but only one can be executing at any time. Models and submodels may be omitted entirely, in which case the study becomes a purely procedural program.

5.1.1 Packages

Packages provide a way of sharing data (variables, constants and parameters) between program modules. Named packages are referenced from a program module through the *Use* statement. Packages are also used to identify externally accessible data for embedded ESL programs.

5.1.2 Procedures

Procedures are static program modules containing purely procedural code where inputs and outputs are passed through an argument list. An alternative form, which may be included in expressions, is a function version which returns a single data value.

5.1.3 Submodels

Submodels are dynamic program modules containing modelling code (including differential equations). Submodels allow a large system to be modelled in a hierarchical manner. A single generic submodel may be instantiated any number of times to represent specific components of the system. ESL provides a standard library of common submodels. Submodels are called from the *dynamic* region of the model or other submodels.

5.1.4 Model

The model is the top-level program module containing modelling code. A simple system may require a model only; a more complex system will include several layers of submodels below the model.

5.1.5 Experiment

The experiment contains procedural code that defines how the model is to be run. It may be very simple - just calling for a single run of the model, or more complicated, perhaps involving several runs of the model with different parameter values.

5.2 Model and Submodel Structure

An ESL Model is divided into several regions identified by a keyword. The general structure is shown below:

```
Model<name><argument list>;
  <declarations>
  Initial
    <initialisation code>
  Dynamic
    <modelling code>
    Step
      <integration step code, e.g. plotting>
    Communication
      <communication point code, e.g. tabulation>
  Terminal
    <end of run code>
End <name>;
```

Note: *Only the Dynamic region is mandatory (the Initial and Terminal regions and the Step and Communication sub-regions of the Dynamic region are optional).*

The structure of a Submodel is identical to the model except that there is no Terminal region.

The purpose of each of the regions is described in the following sections.

5.2.1 Model Statement

This statement declares the name of the model and may include an optional argument list.

Example:

```
My_model(Real:out1, out2 := Real:in1, In2);
```

The "`:=`" symbol separates the output arguments from the input arguments (output arguments appearing first). The input arguments (if present) are values that are passed to the model from the experiment once only *before* the model is executed; the output arguments (if present) are values passed back to the experiment at the *end,* when the model terminates.

5.2.2 Initial Region

This is where any calculations and assignments are carried out before a simulation run takes place. In particular, it is where state variables are normally initialised.

Example:

```
Par1 := Par2 + Par3;
x  := 0.0;
x' := 1.0;
```

Here *Par1* may be a parameter whose value depends on *Par2* and *Par3*; *x* and *x'* are state variables.

Note: *An alternative way of initializing state variables (or any variables) is in their declaration.*

5.2.3 Dynamic Region

The dynamic region is where the differential and algebraic equations that describe the dynamics of the system go. The main difference between the dynamic region of the model and other regions is that the dynamic region code is *declarative* whereas in other regions the code is *imperative*. Statements in the dynamic region describe dynamic relationships between model variables and are deemed to be executed concurrently or in parallel. Consequently, the order in which such statements are presented in the dynamic region is immaterial – statements can be grouped logically, in the way which best describes the system being simulated. Of course, during execution, the dynamic region statements must be executed in a particular order as the solution is advanced step by step. This is taken care of by the ESL compiler which automatically sorts the statements into an executable order. Because of the nature of the dynamic region, certain rules apply, for example, a model variable may be assigned a value at *one point* only – otherwise you would be trying to assign multiple values to the variable simultaneously. Similarly, all state variables must be correctly initialised. (The ESL compiler ensures that these rules, and others, are obeyed).

Note: *In some rare cases you may want to ensure that the dynamic region statements are executed in precisely the order in which you have presented them, e.g., for reasons of numerical accuracy. In such cases, the automatic sorting function can be overruled by the inclusion of a NOSORT statement in the code following the model statement.*

Examples of dynamic region statements:

```
X'' := -k*x' - x + 1;

Deriv := x1' + x2';

y := INTEG( 0.0, Eps*x + 3.2 );

z := TRANSFER( K(s+1)/(s**2 + 2*s +1) )*w;
```

The first statement is a natural way of writing a differential equation. Here it is a second order equation, but it could be first order or higher order – the limit is that the total length of the variable name plus primes (') must not exceed 28 characters (so you could define a 27th order differential equation in x – if you really wanted!) The second statement is just an algebraic assignment. The third statement is an integral equation using the library submodel INTEG. The fourth statement specifies a transfer function (see the Development Guide for details of this).

5.2.4 Step Region

The step region code is executed at the end of every integration step. Typically Plot or Prepare statements would be placed here to maximize the output and produce smooth graphs. The integration step-size is determined by the reserved variables CINT and NSTEP. CINT specifies the communication interval (see next section); NSTEP specifies the *minimum* number of integration step to be taken in each communication interval. The *maximum* integration step-size is therefore given by CINT/NSTEP.

Note: *If you are using a variable-step integration algorithm such as RK5, the actual step-length will be determined by the algorithm to satisfy the error criteria. However, the step-size will not exceed CINT/NSTEP. For fixed-step integration algorithms such as RK2 and RK4, the step-size will normally be CINT/NSTEP. The exception to this is when the integration must negotiate discontinuities (see Chapter 7).*

5.2.5 Communication Region

The communication region code is executed at regularly spaced communication intervals, as specified by the reserved variable CINT. This is a good place for numeric or tabulated output, as produced by the Tabulate statement.

5.2.6 Terminal Region

The terminal region contains code that is executed when the simulation run terminates, i.e., when *T >= Tfin* or some other terminate condition. It is intended for any calculations that must be carried out at the end of a run. The Terminal region is only allowed in a model.

5.2.7 Simulation Parameters

The simulation parameters, which control a simulation run, are defined in a special *Reserved* package which is always visible in models, submodels and the experiment. If you need access to any simulation parameters a Procedure, simply include a *Use Reserved* statement. The simulation parameters, with their default values are:

TSTART	(0.0)	- initial value of T at start of run
TFIN	(10.0)	- final value of T at end-of-run
CINT	(1.0)	- communication interval
DISERR	(0.0001)	- discontinuity detection error tolerance
INTERR	(0.001)	- integration error tolerance
ALGO	(1 or RK5)	- integration algorithm
NSTEP	(1)	- number of integration steps in CINT

Algo can be specified by assigning one of the following numeric constants:

RK5	(1)	- fifth-order variable-step integration
RK4	(2)	- fourth-order Runge-Kutta integration
RK2	(3)	- second-order Runge-kutta integration
STIFF2	(4)	- second-order stiff integration
GEAR1	(5)	- Gear's variable-step stiff integration
GEAR2	(6)	- Gear's method with diagonal Jacobean
ADAMS	(7)	- Adams predictor-corrector integration
RK1	(8)	- Euler first order integration
LIN1	(21)	- Newton-Raphson Linearization routine
LIN2	(22)	- Simplex Linearization routine.

The last two constants *LIN1* and *LIN2* are used with the steady-state function *Trim*. There are in addition one or two special reserved parameters, described in the Development Guide, providing information about the state of a run.

5.3 Program Example

The following example includes all of the program modules introduced above. It is very similar to the example that was used to illustrate graphical model construction. The code is explained in the following notes.

```
01 Study
02   Include "Integ";
03   Package SystemParameters;
04 -- Parameters of system
05     Real:A, B;
06   End SystemParameters;

07   Procedure ErrorSquared(Real:ActualValue,DemandValue)Return Real;
08 -- Function procedure - calculates square of error
```

```
09     Real: Value;
10     Value := (ActualValue - DemandValue)**2;
11    Return Value;
12   End ErrorSquared;

13   Submodel PIController(Real: y := Real: Gain, Ti, x);
14 -- Proportional plus integral controller submodel
15       Real: Intx;
16     Initial
17       Intx := 0.0;
18     Dynamic
19       Intx' := x;
20       y := Gain*(Intx/Ti + x);
21   End PIController;

22   Submodel System(Real: output := Real: Input);
23 -- Second order system submodel
24     Use SystemParameters;
25     Dynamic
26       output := Transfer(A/(s**2 + B*s + A))*Input;
27   End System;

28   Model ControlSystem(Real: Cost := Real: Gain, Ti);
29 -- Top-level model
30     Real:Demand, Response, Error, ActuationSignal, FeedbackSignal;
31     Initial
32       Demand := 1.0;
33     Dynamic
34       Error := Demand - FeedbackSignal;
35       ActuationSignal := PIController(Gain, Ti, Error);
36       Response := System(ActuationSignal);
37       FeedbackSignal := Transfer(1/(0.1*s + 1))*Response;
38       Cost := Integ(0.0, ErrorSquared(Response, Demand));
39      Step
40        Plot "Control System", t, Demand, [Response], 0,Tfin,0,2;
41      Prepare " ",t,Demand,Response,ActuationSignal,FeedbackSignal;
42   End ControlSystem;

43 -- Experiment
44   Use SystemParameters;
45   Real: Gain, Ti, Cost;
46 -- Set system parameters
47   A := 100.0;
48   B := 10.0;
49 -- Set simulation parameters
50   Tfin := 5.0;
51   Cint := 0.5;
52   Nstep := 5;
53 -- Call model from loop
54   Loop
55     Read Gain, Ti;
56     Terminate Gain = 0.0;
57     ControlSystem(Cost := Gain, Ti);
58     Print "Cost = ", Cost;
59   End_Loop;
60   Clear_Screen;
61 End_Study
```

- line 1 Study statement – start of ESL program
- line 2 include the ESL library submodel Integ

- lines 3-6 defines a package defining system parameters
- lines 7-12 defines a function procedure which returns a Real value
- lines 13-21 defines a submodel for a PI controller
- line 19 - example of a differential equation
- lines 22-27 defines a submodel for the system
- line 24 Use statement giving access to the system parameters
- line 26 Transfer statement describes the system transfer function
- lines 28-42 defines the model
- lines 35, 36 and 38 submodel calls
- line 40 Plot statement to generate a run-time plot
- line 41 Prepare statement to save data for post-run plotting
- lines 44-60 defines the experiment
- line 61 End_study statement – end of ESL program

The procedure ErrorSquared simply calculates the square of the difference between its two arguments and returns the value. This is an example of a function style procedure. The function appears in the expression in line 38.

Submodel PIController implements a simple proportional plus integral controller. Note the state variable *Intx* defined by the differential equation in the Dynamic region (line 19) is initialised in the Initial region (line 17). All state variables must be properly initialised.

Submodel System models the system being controlled. In this case the dynamics of the system are specified as a transfer function in an ESL Transfer statement (line 26). The state variables implied by the transfer function are automatically initialised to zero. (See the Development Guide for how to initialise transfer function variables to non-zero values).

The Model ControlSystem is the high-level program module, which defines the interconnections between the submodels. Line 38 is a call to the standard library submodel Integ (specified by the include statement - line2), used to calculate the cost function. The step region includes statements to plot on-line and save data for post-run plotting. The significance of these statements being in the Step sub-region is that they are executed at every integration step. If they had appeared in the Communication sub-region, output would be generated at regular time intervals as defined by the reserved variable Cint.

The Experiment (which comprises all statements following the program module definitions) includes some local declarations (lines 44 and 45); statements to set the system parameters A and B (lines 47 and 48) and statements to set the simulation parameters Tfin, Cint and Nstep (lines 50 to 52). Tfin is the final time at which the simulation run will terminate. Time will run from Tstart (default value 0.0) to Tfin. Cint specifies that the Model and Submodel Communication sub-regions are executed at regular time intervals of 0.5s and Nstep specifies that there will be a minimum of 5 integration steps in each communication interval. (There may be more steps if an adaptive integration algorithm is used where the step length may be reduced to satisfy the error criteria, or discontinuities occur). The main part of the experiment is a loop in which values are read for the variables Gain and Ti (the controller parameters) and the model is invoked. When the program experiment is run, the user is prompted to enter values for Gain & Ti from the command prompt (terminal) window or from a special input window (depending on how the model is invoked – from the command line or via ESL-Studio/ESL-SEC). Note that the Terminate statement stops the loop if a Gain of zero is entered. The Clear_screen statement closes the run-time plot.

5.3.1 Running the Program

There are three ways to run an ESL program: from a command prompt (terminal); from the ESL-SEC (Simulation Execution Control) program or from ESL-Studio. First of all, type in or copy the code for the example program into a text file named *example.esl* (if you copy the code you need to delete the line numbers).

5.3.1.1 Running from a command prompt (terminal)

Open a command prompt (terminal) in the directory where you have saved *example.esl*.

The simplest way to run the program, using the interpreter option, is to type the command:

```
esl example
```

This will invoke the ESL compiler and, if there are no compilation errors, it will then invoke the ESL interpreter. You should get a response similar to that below:

```
c:\Temp\ESL examples>esl example

c:\Temp\ESL examples>esl example
**** E S L  Compiler v8.3.0.1
**** Copyright (C) ISIM International Simulation Limited 1992-2023.
< INTEG          0  WARNINGS     0  ERRORS >
< SYSTEMPARAMETERS    0  WARNINGS     0  ERRORS >
< ERRORSQUARED    0  WARNINGS     0  ERRORS >
< PICONTROLLER    0  WARNINGS     0  ERRORS >
< SYSTEM         0  WARNINGS     0  ERRORS >
< CONTROLSYSTEM    0  WARNINGS     0  ERRORS >
< EXP$MN         0  WARNINGS     0  ERRORS >
**** E S L  Interpreter Run-time v8.3.0.1
**** Copyright (C) ISIM International Simulation Limited 1992-2023.
Gain, Ti:
```

Enter values for the gain (Gain) and the integral control parameter (Ti), say 1.0 and 1.0. A run of the model will take place, a value should be printed for the cost function and an ESL plot generated (note the Plot is generated from *Plot* statement – line 40) i.e.

```
Gain, Ti: 1.0 1.0
Cost =   0.26006
Gain, Ti:
```

The ESL plot you should see is shown in Figure 31:

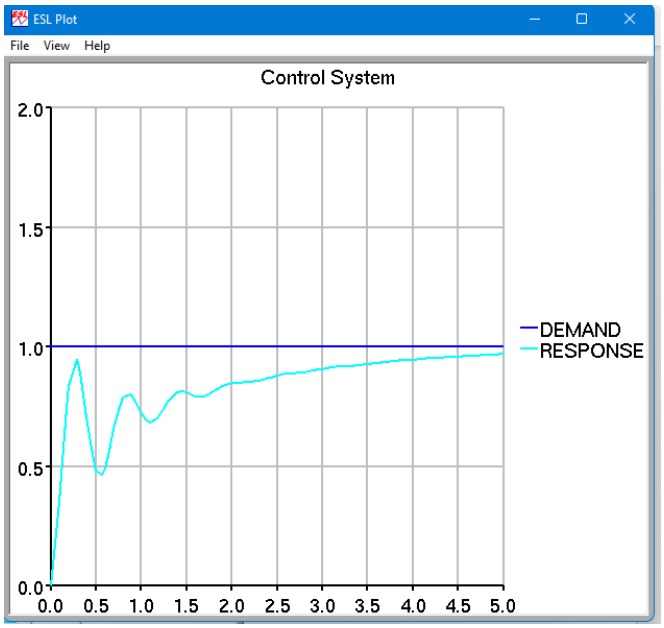

Figure 31 - ESL Plot from example.esl

Further values may now be entered for Gain and Ti giving corresponding cost function values and additional graphs on the same ESL Plot. Entering a value of zero for Gain (and any value for Ti) will terminate the experiment loop and the program.

You will find the full range of `esl` command line options in the Development Guide.

5.3.1.2 Running from ESL-SEC or from ESL-Studio

Chapters 2 and 3 showed you how to run a *graphically* defined simulation from the ESL-SEC Simulation Execution Control window (this was the window that opened when you issued a Run command from ESL-Studio). From ESL-SEC you not only had control over the running of your simulation, you also had access to *Simulation Parameters* and all *Variables* and *Parameters* defined in your model. You were also able to specify *Runtime Displays* and were able to access other *Advanced Simulation Options*. ESL-SEC can be invoked from ESL-Studio or started from a command prompt (terminal) to interactively run any *textually* defined ESL program, giving you all the interactive features available for graphical models. For further information on using ESL-SEC, refer to the ESL-Studio Help Pages.

In the first instance we will simply run the ESL program *example* as it stands.

Either start ESL-SEC from a command prompt (terminal):

```
c:\Temp\ESL examples>esl_sec
```

or from the ESL_Studio Simulate>Simulation Execution… menu selection. Click *Setup* and navigate to example.esl in the *Simulation* text box (the function of the *Specification file* text box is explained later). The dialog should appear as in Figure 32. Note that *Execution Command* allows you to select from several command line options. If you select *Compile, Translate, Link and Execute*, you can select C++ or FORTRAN language options and specify additional link objects (external libraries etc). You will need to have ESL-Pro and the appropriate compilers installed to use these options. Advanced users can also select *Custom Run Command* and set their own Run command. In our case, leave the default Execution Command (Compile and Interpret (.esl)) and simply click the *Load* button. This will return to ESL-SEC with the simulation ready to start. Clicking the *Start* button will open a User Input dialog corresponding to the *Read* statement in line 55 of the program (Figure 33). Enter values for Gain and Ti as before and click OK. This will produce the same ESL Plot you obtained when running the program from a command line. The value of *Cost* (Print statement in line 58) will appear in the message panel of ESL-SEC when you click the Continue button and OK the warning that “Continue may end the simulation” and you will get back to the User Input dialog and be able to enter further values for Gain and Ti (as before, entering a Gain of zero will terminate the program).

Note: *The option to translate and run an ESL program in FORTRAN or C++ requires the appropriate compiler to be installed on your computer. See the Development Guide for details.*

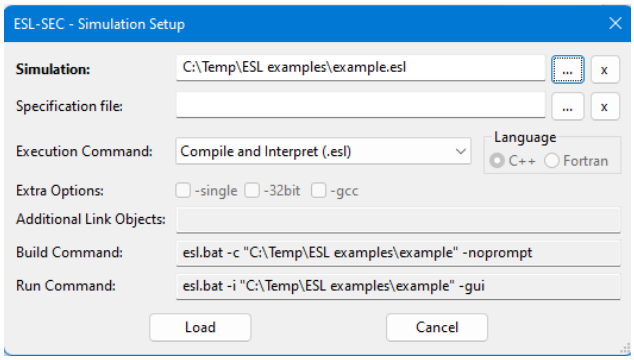

Figure 32 - Simulation Setup dialog

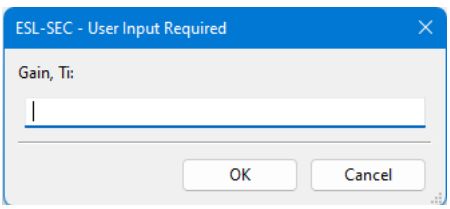

Figure 33 - User Input dialog for example.esl

Although the exercise described above demonstrated that any ESL program that can be run from a command prompt (terminal) can also be run from ESL-SEC (from a command prompt (terminal) or from ESL-Studio), it does not make full use of the interaction offered by ESL-SEC. If you were intending to run your ESL program under ESL-SEC, you would not normally hard-code user input, output and plotting requirements in the program – these can all be specified interactively when running the program. This gives greater flexibility; for example you can easily change the graph plotting specification between runs, and change the values of *any* parameters from ESL-SEC.

To illustrate this, edit the model and experiment of your program example.esl, as shown below, and save as example1.esl. Note that the model argument list has been removed; Gain, Ti and Cost have been re-declared as local parameters and a variable; the Plot and Prepare statements have been commented out; the Gain, Ti and Cost declaration in the experiment has been commented out; the loop has been replaced with a simple model call; and the Clear_Screen statement has been commented out.

```
   ...................
   ...................
   Model ControlSystem;
 -- Top-level model
     Real:Demand, Response, Error, ActuationSignal, FeedbackSignal;
     Parameter Real: Gain/1.0/, Ti/1.0/;
     Real: Cost;
     Initial
       Demand := 1.0;
     Dynamic
       Error := Demand - FeedbackSignal;
       ActuationSignal := PIController(Gain, Ti, Error);
       Response := System(ActuationSignal);
       FeedbackSignal := Transfer(1/(0.1*s + 1))*Response;
       Cost := Integ(0.0, ErrorSquared(Response, Demand));
--       Step
--      Plot "Control System", t, Demand, [Response], 0,Tfin,0,2;
--      Prepare " ",t,Demand,Response,ActuationSignal,FeedbackSignal;
   End ControlSystem;

 -- Experiment
   Use SystemParameters;
 -- Real: Gain, Ti, Cost;
 -- Set system parameters
   A := 100.0;
   B := 10.0;
 -- Set simulation parameters
   Tfin := 5.0;
   Cint := 0.5;
   Nstep := 5;
   ControlSystem;
 --  Clear_Screen;
 End_Study
```

Re-enter the Setup dialog and load example1.esl. Click *Runtime Displays* (Figure 34) to specify Plots, Tables and Prepare output.

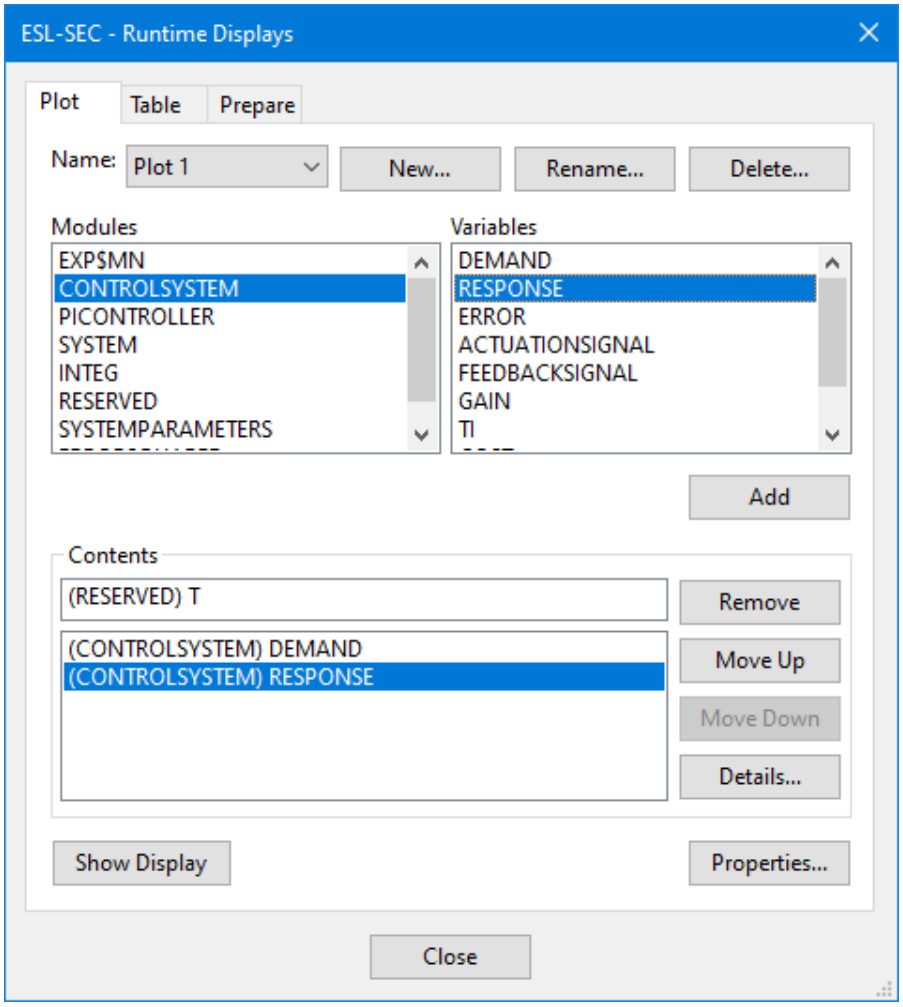

Figure 34 – Runtime Displays dialog

For example, to define a new plot, select the *Plot tab*. The default name for the first plot is *Plot 1*, which you can change if preferred from the *Rename* button. Select the model (CONTROLSYSTEM) from the Modules panel and select the variables to be plotted from the Variables panel (either double click a variable or click the variable followed by the Add button) – *DEMAND and RESPONSE* in this case. The Remove, Move Up and Move Down buttons can be used to rearrange the list of plot variables. The Properties button allows you to refine the appearance of the plot, for example – specify a title and display grid. Finally click *Show Display* to open a plot window. Prepares can be specified in a similar manner from the Prepare tab.

You can set up a table, (from the Table tab) to show the value of the cost function, *Cost*. Under the table Properties, select Monitor from the Style options. This will display just the current value of *T* and *Cost* (rather than a full tabulated list) and will therefore show the final value at the end of each run (as was the case when running from a command line). Do not forget to click *Show Display* to open the table.

If you now click Start on Simulation Execution Control window, you should get one run of the model using the default values of Gain and Ti of 1.0 and 1.0 (specified in their declaration statement). The values of Gain and Ti can now be changed from the Variables dialog (click *Variable* on the Simulation Execution Control window) and further runs made as described under Varying Parameter Values in section 2.7. Don't forget to click Rerun and Continue to obtain each new run.

Figure 35 shows a typical appearance after three runs of the program initiated from Simulation Execution in ESL-Studio.

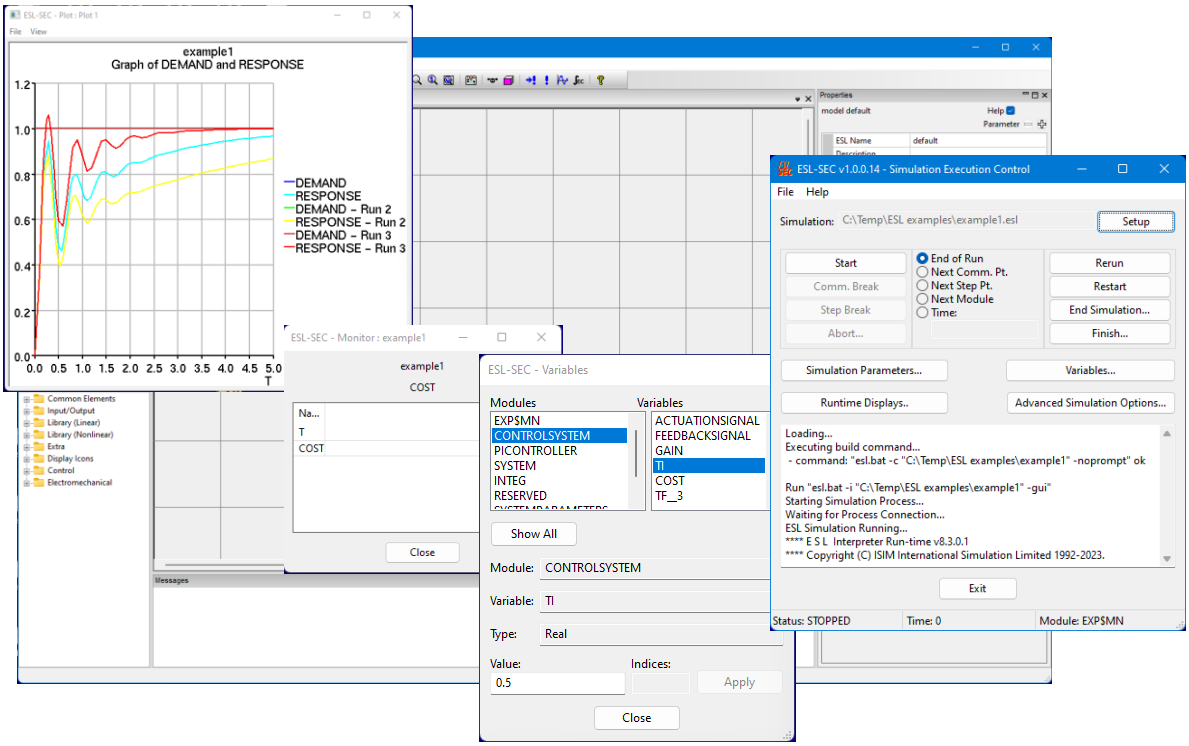

Figure 35 - Running example1.esl from ESL-Studio

Finally click *Exit* on the Simulation Execution Control window to terminate the simulation. You will get a warning – "This will terminate the simulation. Do you wish to continue?" – click *Yes*. This will open a final window - *Specification Changed* (Figure 36). This gives you the option to save a specification file (.sec) which records all the interactive changes made to the program in the current session (Simulation Parameters, Runtime Displays etc). You can view the changes and decide which, if any, to save. You could use the same filename as the ESL program, *example1* and this will create a specification file – *example1.sec*.

For subsequent runs of the simulation, the specification file may be entered in the Simulation Execution Control, via the File>Setup Simulation menu, in the Setup Simulation dialog *Specification File* box. (Figure 37). This will cause the ESL simulation and all parameter and display specifications that were saved to be loaded.

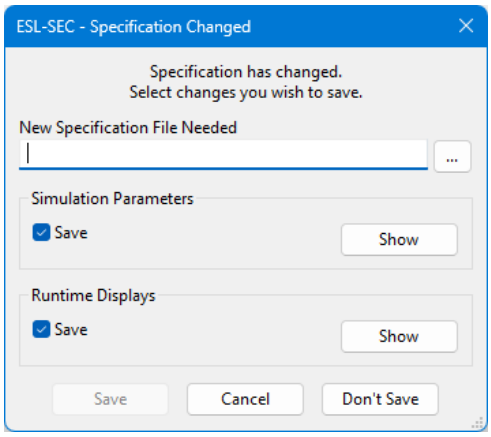

Figure 36 - Specification Changed Dialog

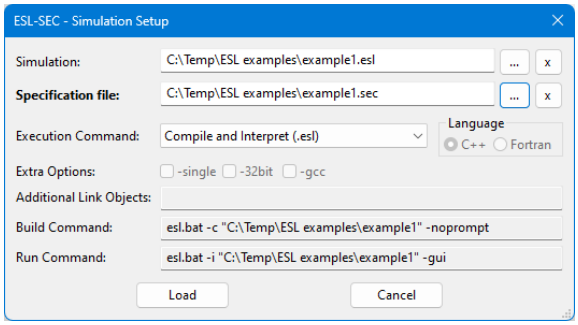

Figure 37 - Specification file in Setup

CHAPTER 6

A Case Study

The aim of the following Case Study is to consolidate and expand what you have learnt in the previous chapters.

6.1 Satellite Roll-Axis Control

6.1.1 Description of system

A three-axis-oriented solar experiment package is to be mounted onto a manned orbiting spacecraft. The package would include, among other things, a 14-inch telescope, together with still-picture and television cameras. A control system is required to point the satellite to the sun, which has a diameter of about 20 arc-minutes. It is also required to point the satellite to a position off the sun but within observation of the sun's corona; consequently, the control system should be capable of positioning a displacement of one degree from the centre of the sun. On account of the limitations of the cameras, the position must be held within 60 arc-seconds with a jitter rate of less than one arc-minute per minute of time. This problem was described by Y Chu in his book 'Digital Simulation of Continuous Systems', McGraw-Hill, 1969.

In this presentation only the control of the roll-axis is considered (Figure 38) which is controlled through a position servo-loop. A high grade rate-integrating gyro is chosen for measuring attitude position, and a dc direct drive torque motor for control actuation. A maximum roll excursion of 7.5 arc-minutes in 15 time-minutes is required. This could be met by using a gyro with a drift of no more than 7.5/15, or 0.5 arc-second per second of time, and by choosing a proper loop gain such that the transient position errors are kept below one arc-minute. Also a maximum roll-rate of one arc-minute per minute of time is required throughout the 15-minute observation period. This could be met by the correct selection of the torque motor and the loop gain.

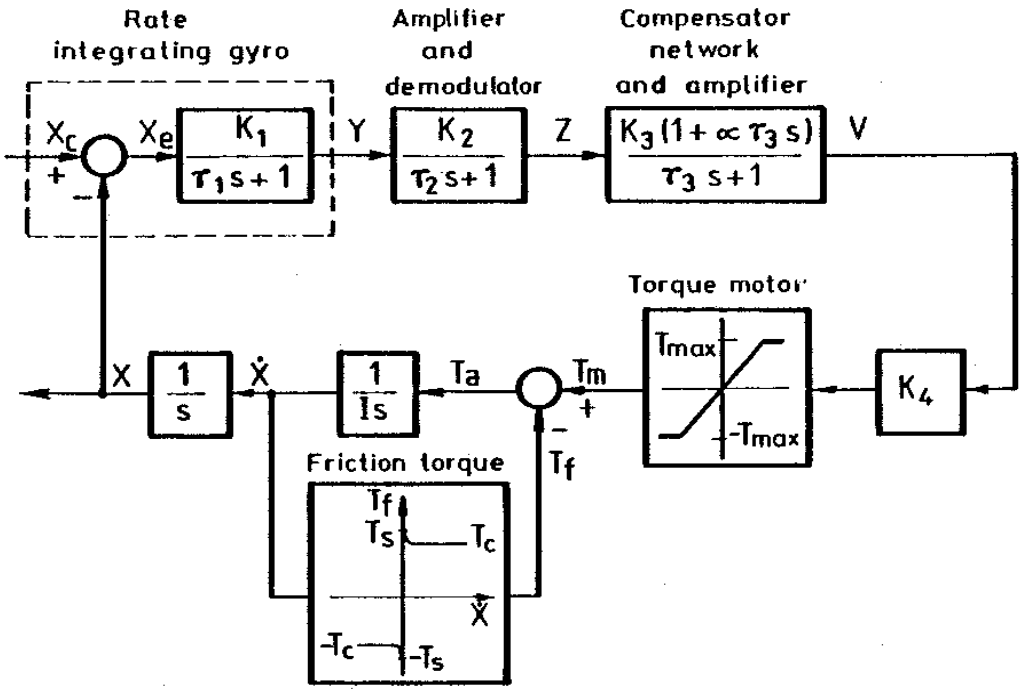

Figure 38 - Satellite Roll-Axis control system

6.1.2 Mathematical Model

The system is adequately described by the block diagram of Figure 38. Alternatively, in terms of differential and algebraic equations, it can be described in the following manner:

As shown in the block diagram, the gyro error Xe is:

Xe = Xc - X

where X is the orientation angle of the roll-axis and Xc the demand angle, and the transfer function of the gyro:

Y' = (K1*Xe - Y)/T1

where Y is the output voltage of the rate-integrating gyro. The output of the gyro is ac-amplified and demodulated. The transfer function of the demodulation is described by:

Z' = (K2*Y - Z)/T2

where Z is the output of the demodulator.

The output of the demodulator is then compensated by a lead-lag network and amplified to drive the torque motor. The transfer function of the combination of the compensator network and the amplifier is given by:

V' = (K3*Z + alpha*T3*K3*Z' - V)/T3

where V is the output from the amplifier. The torque Tm from the torque motor has a saturation characteristic as shown in the block diagram Figure 38

The dynamics of the satellite about the roll-axis is approximated by:

Ta = I*X''

where I is the moment of inertia of the satellite about the roll-axis and Ta is the torque available to overcome the inertia torque. The characteristic of the friction torque Tf is also shown in Figure 38, where Ts is the static friction torque and Tc the Coulomb friction torque. Both Ts and Tc are assumed constant. The effect of the speed and the motor torque Tm on the friction torque Tf can be described in the following manner:

- When X' = 0 and |Tm| <= Ts, Tf = Tm (i.e. the friction torque exactly counters the motor torque, therefore the motor remains stationary)
- When X' = 0 and |Tm| > Ts, Tf = sign(Tm)*Tc (i.e. the motor will begin to move and the friction torque reduces to the Coulomb torque in a direction as to oppose motion)
- When X' <> 0, Tf = sign(X')*Tc (i.e. the friction remains equal to the Coulomb torque)

The available torque is given by the difference between the motor torque and the friction torque

Ta = Tm – Tf.

6.1.3 ESL Simulation

There are several ways in which the system can be programmed in ESL:

- As an ESL-Studio diagram.
- As an ESL program using the *TRANSFER* statement to represent the linear components.
- As an ESL program in terms of the differential equations presented above.
- As a combination of the above, i.e., model part of the system as an ESL submodel.

In the case of the ESL-Studio diagram, you will need to use the *Limiter* and *Friction* simulation elements from the *Library (Nonlinear)>Limiters* and *Misc* Element pane for the torque motor and friction torque blocks.

In the case of an ESL program, you will need to use the standard library *Limit* and *Coulomb* submodels to represent the torque motor and friction torque blocks. Remember to include the flowing statements at the start of the study:

```
INCLUDE "coulomb";
INCLUDE "limit";
```

The values of the various parameters are given in the following table:

Parameter	Value	Units
K1	15.0	V/rad
K2	1000.0	V/V
K3	0.1	V/V
K4	1.356	Nm/V
max_torque	1.356	Nm
I	271.2	kg m^2
T1	0.006	s
T2	0.01	s
T3	0.0555	s
alpha	10.0	
Ts	0.2712	Nm
Tc	0.1356	Nm

6.1.4 Objective

The primary objective is to obtain a plot of the satellite position, X, for a step demand input, Xc, of 1800.0 arc-seconds and numerical values for the final position achieved and the error (expressed in arc-seconds). Other plots of interest are: satellite angular velocity, X', motor torque, Tm, friction torque, Tf and available torque, Ta.

Notes

- In an ESL-Studio Transfer Function element, a coefficient of "s" can only be a single entity (a Number, Constant, Variable or Parameter). Hence, in the third Transfer Function in the forward path the term $\alpha T_3 s$ could be represented as *aT3*s* where *aT3* is a Parameter set to the value of $\alpha \times T_3$.
- The ESL-Studio *Friction* simulation element requires both velocity (X') and applied torque (Tm) as inputs.
- To convert arc-seconds to radians, multiply by π/(180×3600). π may be calculated using: PI := 4.0*ATAN(1.0) in, for example, the INITIAL region of an ESL program.
- You should find values of TFIN = 5.0, CINT = 0.1 and ALGO = RK5 suitable.
- An ESL solution to this problem will be found in the …\esl\examples directory (rollax.esl).

- An ESL-Studio solution to this problem will be found in the …\esl-studio\examples directory (roll_axis.eslstudio).

CHAPTER 7

Advanced Features

In this chapter we introduce some of the more advanced features of ESL. As with the previous material in this User Guide, the aim is to give a broad overview of the topics. A more in-depth treatment will be found in the Development Guide.

7.1 Discontinuities

7.1.1 What are Discontinuities?

A discontinuity is an event which causes the algebraic or differential equations representing the system to suffer a *jump* or *step* change in one or more modelling variables. Such events are very common in real systems, for example limits, dead-space, hysteresis etc. Integration algorithms cannot integrate satisfactorily in the presence of discontinuities. In mathematical terms the function is *piece-wise continuous* with a discontinuity representing an abrupt change in a state variable, or its first or higher derivative. A discontinuity within an integration-step invalidates the Taylor series representation of the step, and consequently any of the integration algorithms used.

Although ESL protects integration from discontinuities, it is helpful to understand the consequences of an *unprotected* discontinuity on the integration process:

- *Fixed-step explicit* - causes erroneous results as the method is attempting to match Taylor series which is invalidated by the discontinuity. Small steps, giving longer execution times minimises this effect.
- *Variable-step explicit* - the method gives inaccurate results which are reflected in the error estimate. This causes the step mechanism to reduce the step which spans the discontinuity to a very small value at which the effect of the discontinuity is minimal. The final result usually has good accuracy but at the expense of excessive computation time.
- *Implicit methods* – are even more sensitive to discontinuities. The result is possibly an abortion, very slow execution and/or erroneous results.

7.1.2 Handling Discontinuities in ESL

ESL incorporates an integration-discontinuity control mechanism which accurately and efficiently detects and locates discontinuities. ESL does not allow a discontinuity to occur within an integration-step. It arranges for it to occur after the end of one step and before the beginning of the next i.e., between steps. This would normally lead to a gross time error, however at the end of each step a check is made to see if a discontinuity has occurred during the step. If this was the case, the step is repeated with a shorter step-length based on an interpolation of the discontinuity function (the relational expression describing the discontinuity). The interpolation process is repeated until the end of step occurs just after the point of discontinuity, but within a specified error bound. The change to a modelling parameter may then be made between steps, before proceeding with the simulation of the new state of the system. As the control mechanism does not allow any change to take effect during an integration-step, the integration routines are protected from the effects of a discontinuity occurring in mid-step.

The method is illustrated in Figure 39. Here a discontinuity occurs when the variable *A* becomes greater than or equal to the variable *B*. A discontinuity function is defined as $\theta = A - B$.

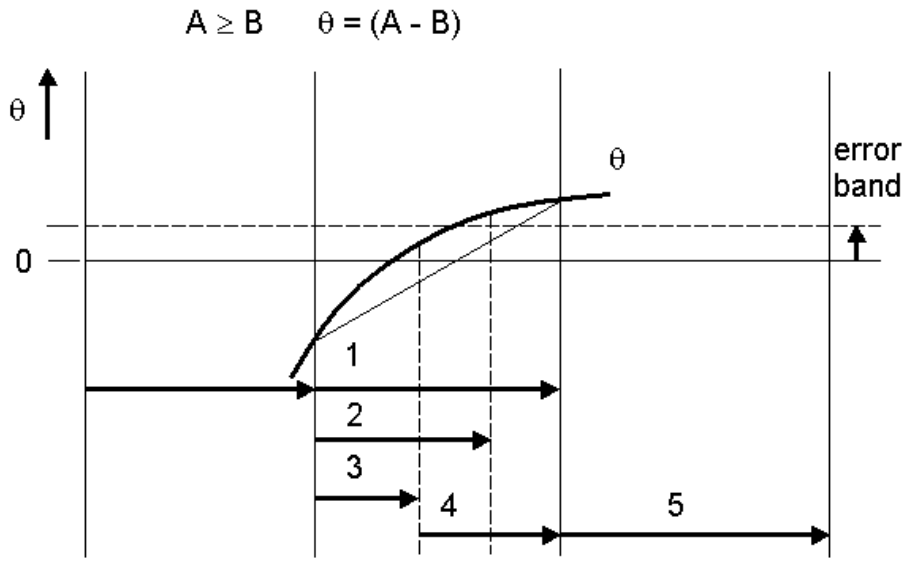

Figure 39 - Discontinuity detection

The sequence is:

- Step (1) has been computed by the integration algorithm, integration accuracy criteria have been satisfied.
- The discontinuity detection control, however, detects a discontinuity as *θ* has changed sign. It uses linear interpolation to suggest a step-length, step (2) that will be close to the point of discontinuity. Note that the linear interpolation *aims* for the centre of the error band.
- Step (2) is undertaken, but again it *overshoots* the discontinuity, and a further interpolation is used to refine the step-length i.e., step (3). This and any subsequent interpolation use quadratic, rather than linear, interpolation based on three values of *θ* which span the discontinuity.
- The result of step (3) is that *θ* now lies within the error-bound, and the discontinuity is regarded as being accurately detected.
- The result of the relational operation, A >= B, is now set to be true; during previous steps 1, 2 and 3, it had been maintained false.
- The recovery step, step (4), is computed using the new result of the relational operation. This step *aims* for the same point in time as the original step, step (1), in which the discontinuity was first encountered.
- Step (5) is a normal step following the discontinuity process.

7.1.3 Representation of Discontinuities in ESL

The ESL library contains submodels for dealing with commonly occurring discontinuities such as limiters, dead-space and hysteresis. However, two language constructs are available for modelling any non-standard discontinuous functions. These are the *If clause* and the *When statement*.

7.1.3.1 If clause

The If-clause is part of a modelling code assignment statement, and it may only appear in the dynamic region of a model or submodel. It acts as a two-way, or multiple-way, switch which assigns a single value to a variable, for example:

```
y:= If a > b Then x1 Else x2;
y:= If a > b Then x1 Else_If x< 0.0 Then x2 Else x3;
y:= If a > b and c >= (2*threshold) Then x1 Else x2;
```

```
      y:= If a > b or c > b Then x1 Else x2;
```

The final *Else* is mandatory because an assignment must always be made to the variable. Additional *Else_If* clauses introduce further branches, or choices.

The value given to the variable *y* corresponds to the first logical expression which is true.

Note: *It is the logical expressions in the above examples that generate the discontinuity functions, i.e. expressions involving logical comparisons like > < >= etc..*

The following code shows the implementation of a limiter submodel using the If-clause:

```
SUBMODEL LIMIT(REAL:y := CONSTANT REAL:LL,UL; REAL:x);
-----------------------------------------------------------------
-- A limiter sets lower and upper limits on the amplitude
-- of an input variable. The calling sequence is:
--
--    y:= LIMIT(LL,UL,x)
--
-- where:
--    LL is the lower limit;
--    UL is the upper limit;
--    x is the input variable.
--    y is given a value such that:
--          y = x, if LL < x < UL,
--          y = UL, if x >= UL,
--          y = LL, if x <= LL.
--
-- Note the inputs LL, UL must be UL > LL, and are assumed
-- constant throughout a run. The output is an algebraic
-- variable.
-----------------------------------------------------------------
  REAL: range,xnorm;
INITIAL
  if LL >= UL then
    print "**** Error in LIMIT: Limits not consistent";
    STOP;
  end_if;
  range:= UL-LL;
DYNAMIC
  xnorm:= (x-LL)/range;
--
  y:= if xnorm > 1.0 then UL
      else_if xnorm < 0.0 then LL
           else x;
--
END LIMIT;
```

7.1.3.2 When statement

The *When* statement is a modelling code statement which may only appear in the dynamic region of a model or submodel. Its operation is fundamentally different from the If-clause. The If-clause is active on each execution of the dynamic region and causes an assignment to be made. The *When* body, however, is *only* executed at the instant when its logical expression become true. Consider:

```
      When x >= ul Then
        Print "x >= ul has changed from FALSE to TRUE at time= ", T;
        trigger:= true;
      End_When;
```

The body of the *When* statement is procedural, non-modelling code, which is only executed at the instant when the logical expression, *x >= ul*, changes from false to true. The *Print*

statement accurately reflects the situation. Note in this example that if *trigger* is used elsewhere in the dynamic region, then it must have been initialised in the Initial region, or in its declaration. The above, however, will only set *trigger* when *x* becomes greater than or equal to *ul*, and *trigger* is never reset. The following addresses this situation:

```
      When x >= ul Then
        trigger:= true;
      When x < ul Then
        trigger:= false;
      End_When;
```

Note: *Multiple When statements can be concatenated together with a single End_When.*

The following code shows the implementation of a sample and hold submodel using the When statement:

```
SUBMODEL SAMHLD(REAL:y := CONSTANT REAL:per; REAL:x);
-----------------------------------------------------------------
-- Samples and holds the value of an input variable.
-- Samples are taken periodically and the output is the
-- value of the last sample taken. The calling sequence is:
--
--    y:= SAMHLD(per,x)
--
-- where:
--    per is the sampling period;
--    x is the input variable;
--    y is given a value such that:
--          y = x, initially,
--          y = x, at the last sampling period.
--
-- Note per is assumed constant throughout a simulation run.
-- The output is a memory variable.
-----------------------------------------------------------------
  REAL: start;
INITIAL
  y:= x;
  start:= T;
DYNAMIC
  when T - start >= per then
    start:= start+per;
    y:= x;
  end_when;
--
END SAMHLD;
```

7.2 Segments

An important feature of ESL is its *segment* structures. Segments were originally included in ESL as a means of providing a parallel processing capability to improve execution times. The idea is that a large simulation can be broken down into self-contained segments that can be executed in parallel on different processors or networked computers. Communication takes place between segments at pre-determined communication points through a TCP IP protocol. We shall see that segments are useful even when they are not executed in a truly parallel manner in supporting multi-rate simulations and also that segments provide the means of embedding ESL simulations in other programs.

There are three types of segment in ESL:

- *Emulated segments* – these allow parallel operation to be emulated on a single computer – useful for implementing multi-rate simulations and for testing parallel segments before assignment to separate processors.

- *Remote segments* – these can be assigned to different processors for truly parallel operation.
- *Embedded segments* – used where an ESL model is to be integrated with another application.

7.2.1 Emulated Segments

A large simulation will typically include some parts which have fast dynamics (or small time constants) while other parts will have much slower dynamics (or long time constants). Consider, for example, an all-electric ship. The inverters and motor control circuitry will have very small time constants, perhaps sub-microsecond; the propulsion motors will have longer time constants, maybe of the order of milliseconds; while the dynamics of the ship itself would be characterised by time constants of seconds or larger. If the whole simulation is written as a single model-submodel structure, the integration step-length (and hence the time taken for the simulation to run) will be determined by the parts that have the fastest dynamics. Emulated segments allow different parts of the simulation to use the most appropriate step-length and integration algorithm, while still running the simulation on a single computer, and so achieving much shorter simulation times.

Emulated segments are defined within an ESL Study and called from the communication region of the model. The model may include some part of the simulation or may simply be the means of linking the individual segments. The general structure is shown below:

```
Study
  <packages, procedures and submodels>
  .........
  Segment Seg1(Real:out := Real:in);
  .........
  Initial
    CINT  := ...;
    NSTEP := ...;
    ALGO  := ...;
  .........
  Dynamic
    .........
  End seg1;
  <further segments>
  Model Mod1;
  .........
    Dynamic
    .........
      Communication
        Seg1(y := x);
        .........
  End Mod1;
  .........
  Mod1;
End_study
```

The structure of a segment is identical to that of a model. The simulation parameters to be used by the segment (CINT, NSTEP, ALGO) must be set in the segment initial region. CINT will normally be the same as that used by the model, but different values of NSTEP and ALGO may be set allowing a different integration step-length and/or integration algorithm to be used by the segment. An ESL Study may contain multiple segments – all called from the model communication region. The segment in the example has only one input and one output – in general a segment may have multiple inputs and outputs.

An example of a program which uses an emulated segment *seg1.esl* will be found in the ESL installation*…esl\examples* directory and is described in some detail in the Development Guide. It is suggested that as an introduction to ESL segments, you examine and run this example.

7.2.2 Remote Segments

Remote segments provide true parallel or distributed simulation over a network of computers using a client/server arrangement – the main simulation (model and experiment) being the client and the segments the servers.

The main difference between remote segments and emulated segments is that the remote segments are typically converted into executable code (via the FORTRAN or C++ translation route with ESL-Pro) and copied to the computers on which they are to run. The general syntax for a remote segment is:

```
Remote
  <packages, procedures and submodels>
  .........
  Segment Seg1(Real:out := Real:in);
  .........
  Initial
    CINT  := ...;
    NSTEP := ...;
    ALGO  := ...;
  .........
  Dynamic
    .........
  End seg1;
```

Note that the code begins with the keyword *Remote* and contains one and only one segment plus associated packages, procedures and submodels. There is no model, experiment and no final *End_study* statement. The program structure has to be: ESL compiled; translated into FORTRAN or C++; compiled and linked to create an executable. The executable must then copied to the remote computer on which the segment is to be executed.

Note: *Different instances of the same segment may be run on different computers.*

The main simulation (the client), containing the model, must include external segment declaration statements (just the segment declaration statements from the remote structures followed by the keyword *External*), e.g.

```
Study
  <packages, procedures and submodels>
  .........
  Segment Seg1(Real:out := Real:in)External;
  .........
  <further external segment declarations>
  Model Mod1;
  .........
    Dynamic
    .........
      Communication
        Seg1(y := x);
        .........
  End Mod1;
  .........
  Mod1;
End_study
```

Before the distributed simulation can be run, a *segment location file* must be created on the local computer (where the main simulation is located). This file must have the same name as the main simulation program but with extension ".rem" and is used to associate a segment name with a host and executable file. The segment location file has the general form:

```
segment_name<Spaces>remote_host_reference<Spaces>
remote_simulation_command
```

where *segment_name* is the name as given in the main ESL model:

remote_host_reference has the form:

```
[ protocol ':' ] [ remote_user '@' ] remote_host_name
```

The *protocol* may be one of:

- `rsh` - the remote simulation will be launched via the *rsh* protocol. Note that this is the default protocol and may be omitted.
- `ssh` - the remote simulation will be launched via the *ssh* protocol.
- `esl` - the remote simulation will be launched via the custom ESL protocol. Note this requires the ESL Launcher and is the recommended option (see below).

remote_user is for the *rsh* & *ssh* protocols (if required). This will default to the same name as the user on the local host so is generally not required.

remote_host_name may also be an IPv4 address. Also a dash "-" may be used to indicate the localhost. In that case, or if *remote_host_name* is for the current (local) computer, ESL does not use the protocol (if specified) to launch the remote segment as it can launch it locally in another process.

The ESL Launcher is started on the remote computer using:

```
esl_launcher
```

Note: *ESL-Pro must be installed and explicitly authorised on the remote computer to use the esl protocol.*

As an example, consider *seg1m.esl* and *seg1r.esl* from the …*esl\examples* directory.

seg1m.esl is the main simulation (the client):

```
-- File seg1m.esl - basic segment example, specifies model
-- with segment defined as external or remote segment
-- (see file seg1r.esl).
--
STUDY
INCLUDE "realpl";
INCLUDE "integ";
INCLUDE "stepp";
--
SEGMENT SEG(REAL: segout:= REAL: segin,Taus) EXTERNAL;
END SEG;
--
MODEL MODSEG(REAL: y:= REAL: Tau);
  REAL: x,xf,in;
  REAL: Tauf/0.6/;
  LOGICAL: log;
INITIAL
  x:= 0.0;
DYNAMIC
  log:= STEPP(6.0);
  in:= if log then 0.0 else 1.0;
  y:= INTEG(0.0,(in-y)/Tau);
  xf:= REALPL(0.0,Tauf,x);
STEP
  PLOT T,y,0,TFIN,0,1;
  PREPARE "seg1m",T,y,x,xf;
COMMUNICATION
-- Segment invocation
  SEG(x:= y,Tauf);
--
END MODSEG;
-- EXPERIMENT
  REAL: y,Tau/2.0/;
  CINT:= 0.5; NSTEP:= 10; TFIN:= 16.0; ALGO:= RK5;
```

```
-- Model invocation
  MODSEG(y:= Tau);
--
END_STUDY
```

Notice the *EXTERNAL* statement in the segment declaration and the segment invocation in the *COMMUNICATION* region.

seg1r.esl is the remote segment (the server):

```
-- seg1r.esl - remote Segment to be used in connection with
-- model defined in file seg1m.esl.
--
REMOTE
INCLUDE "realpl";
--
SEGMENT SEG(REAL: segout:= REAL: segin,Taus);
INITIAL
  CINT:= 0.5; NSTEP:= 10; TFIN:= 16.0; ALGO:= RK4;
DYNAMIC
  segout:= REALPL(0.0,Taus,segin);
STEP
  PREPARE "seg1r",T,segout,segin;
--  PLOT T,segin [segout],0,tfin,0,1;
COMMUNICATION
  tabulate t,segin,segout;
--
END SEG;
```

Notice the *REMOTE* statement at the start of the code and the simulation parameter specifications in the *INITIAL* region – these will be used for the segment.

Both programs can be compiled, translated to C++ (say) and linked to generate executables on the local computer using:

```
esl -cccl seg1m

esl -cccl seg1r
```

copy *seg1r.exe* to the remote computer (say *PC001*).

If you are going to use the esl protocol (recommended), create a segment location file *seg1m.rem* on the local computer containing:

```
SEG esl:PC001 seg1r
```

Start a command prompt (terminal) on the remote computer in the directory containing *seg1r.exe* and run the ESL Launcher:

```
esl_launcher
```

and start the main simulation on the local computer:

```
seg1m
```

This should generate some tabulated output and the graph shown in Figure 40 on the local computer. These results should be identical to those generated by the *seg1.esl* emulated segment example referred to in the previous section.

See the Development Guide for further details of how to develop and run remote segments.

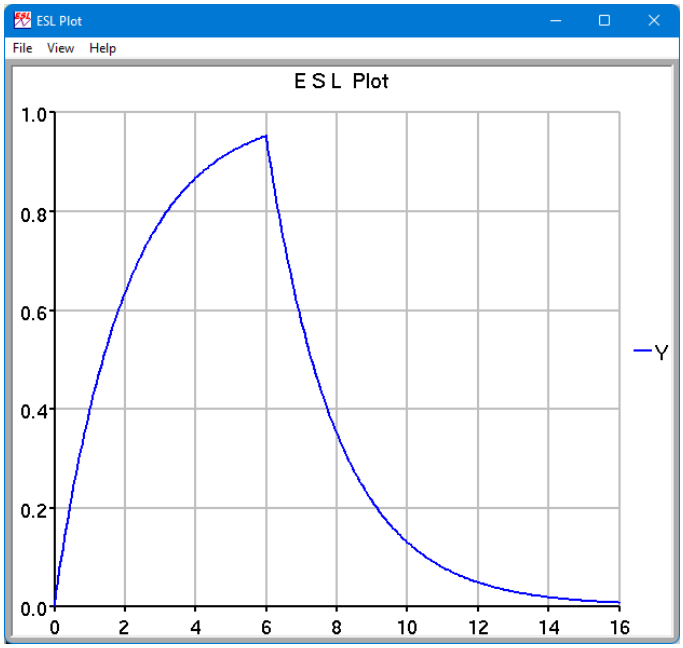

Figure 40 - output from remote segment example seg1m/seg1r

7.2.3 Embedded Segments

The embedded segment provides a means of generating code that can be called from another non-ESL program – thus enabling the segment to be *embedded* in another program.

In an ESL embedded segment, all interface variables appear in ESL *Packages*. The code below is an example of an embedded segment for a simple linear model of a dc motor. Inputs to the model appear in the package Esl_inp; State outputs appear in the package Esl_state and algebraic outputs in package Esl_out. The package Esl_par contains parameters which should be accessible to the user and Esl_view contains viewables, i.e. any variables that may be plotted or are used to drive visualizations. The dynamic model itself is defined in the *Segment* structure. The choice of package name is entirely up to you, however package name beginning ESL_ are automatically exposed by the 'eslgen' command (see *options* below). You could declare all the interface variables in a single package – multiple packages have been used here to help distinguish the use of the variables.

```
EMBEDDED
Package Esl_inp;
  Real: va, tl;
End Esl_inp;
--
Package Esl_state;
  Real: ia, Wa;
End Esl_state;
--
Package Esl_out;
  Real: v_error;
End Esl_out;
--
Package Esl_par;
  Parameter Real:Kt/0.0275/, Kb/0.04/, Ra/9.0/,
    La/4.065e-03/, Ja/1.71e-06/, Ba/1.5e-04/;
End Esl_par;
--
Package Esl_view;
  Real: v_back, t_motor, t_avail;
End Esl_view;
--

Segment dc_motor;
  Use Esl_inp; Use Esl_state;
  Use Esl_out; Use Esl_par;
  Use Esl_view;
  Real: i, w, ve, vb, tm, ta;
  Dynamic
    ve:= va-vb;
    i:=Transfer(1/(La*s + Ra))*ve;
    tm:= Kt*i; ta:= tm-tl;
    w:=  Transfer(1/(Ja*s+Ba))*ta;
    vb:= Kb*w;
  Communication
    ia := i; wa := w;
    v_back  := vb; v_error := ve;
    t_motor := tm; t_avail := ta;
End dc_motor;
```

Using an ESL-Pro utility, *eslgen*, on Windows an embedded segment may be compiled into:

- a dynamic link library (DLL) providing a function interface which can be used in Microsoft Visual Basic or Visual C++ projects;
- a COM object, which can be used in Visual C++ projects (in an object oriented manner) and also other control/ActiveX hosts (such as Web Browsers);
- or a .NET Framework assembly, which can be used in any .NET Framework project such as C#.

The *eslgen* command has the following form:

```
eslgen -dll|-com|-comnr|-clr filename {io_packages}
```

The options are:

```
-dll   - create a DLL from an ESL embedded segment
       eslgen -dll file_no_ext {io_packages}
-com   - create a COM object from an ESL embedded segment and
register it
       eslgen -com file_no_ext {io_packages}
-comnr - create a COM object from an ESL embedded segment (but do not
register it)
       eslgen -comnr file_no_ext {io_packages}
-clr   - create a .NET (2+) assembly from an ESL embedded segment
       eslgen -clr file_no_ext {io_packages}
The {io_packages} are the names of ESL packages that are to be
exposed. If none are specified, any beginning "Esl_" will be exposed.
```

The generated embedded segment code (whether it be DLL, COM or .NET) provides a set of functions or methods for running the code. These are listed in Table 1, below. In addition to these functions, mechanisms are provided for accessing the interface variables (as declared in ESL packages). The detail of how to call the functions and access the variables depends on which type of code has been generated (DLL, COM or .NET) and is described in detail in the Development Guide.

The idea is that, after calling ExStrt to initialise the code, any parameters (including simulation parameters) may be set or changed. ExInit is then called to initialise the segment (the Initial region is executed). ExSim is then called repeatedly in a loop to keep advancing the segment by the communication interval, CINT, on each call. Inputs are passed to the segment before each call of ExSim, and outputs retrieved after each call. For CLR operation, a special function, ExPrestep is provided to update segment outputs that depend directly on the inputs without advancing time. At any time the segment can be re-initialised by calling ExInit. When the simulation is complete the function ExFin is called to properly terminate the code.

Table 1 - Embedded segment functions

Name	***Meaning***
ExStrt	Prepare embedded code for use - must only be used once at program start.
ExInit	Initialise embedded segment for a single simulation run.
ExSim	Advance Simulation by one time-frame (specified by the simulation parameter CINT).
ExPrestep	Evaluate algebraic outputs without advancing the simulation (CLR only).
ExFin	Close down simulation - must only be used once at program termination.

Note: *Please refer to the on-line documents or contact ISIM for further details on the use of embedded segments including directly producing FORTRAN or C++ code that may be used to invoke the simulation in an application.*

Embedded segments are a powerful feature of ESL allowing simulations to be easily incorporated various applications. Examples of the use of embedded segments include training simulators where, the graphical user interface has been provided by other software, a C++ program say, which calls upon an embedded ESL program to provide the underlying dynamic simulation.