**The lecture 5**

**Simulating Dynamic Systems**

Simulating a dynamic system refers to the process of computing a system’s states and outputs over a span of time, using information provided by the system’s model. Simulink simulates a system when you choose **Start** from the model editor’s **Simulation** menu, with the system’s model open.

A Simulink component called the Simulink Engine responds to a Start command, performing the following steps.

**Model Compilation**

First, the Simulink engine invokes the model compiler. The model compiler converts the model to an executable form, a process called compilation. In particular, the compiler

**•**Evaluates the model’s block parameter expressions to determine their values.

**•**Determines signal attributes, e.g., name, data type, numeric type, and dimensionality, not explicitly specified by the model and checks that each block can accept the signals connected to its inputs.

**•**Simulink uses a process called attribute propagation to determine unspecified attributes. This process entails propagating the attributes of a source signal to the inputs of the blocks that it drives.

**•**Performs block reduction optimizations.

**•**Flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain.

**•**Sorts the blocks into the order in which they need to be executed during the execution phase.

**•**Determines the sample times of all blocks in the model whose sample times you did not explicitly specify.

**Determining Block Update Order**

During a simulation, Simulink updates the states and outputs of a model’s blocks once per time step. The order in which the blocks are updated is therefore critical to the validity of the results. In particular, if a block’s outputs are a function of its inputs at the current time step, the block must be updated after the blocks that drive its inputs. Otherwise, the block’s outputs will be invalid. Simulink sorts the blocks into the correct order during the model initialization phase.

**Direct-Feedthrough Ports.** In order to create a valid update ordering, Simulink categorizes a block’s input ports according to the relationship of outputs to inputs. An input port whose current value determines the current value of one of the block’s outputs is called a *direct-feedthrough* port. Examples of blocks that have direct-feedthrough ports include the Gain, Product, and Sum blocks. Examples of blocks that have non-direct-feedthrough inputs include the Integrator block (its output is a function purely of its state), the Constant block

(it does not have an input), and the Memory block (its output is dependent on its input in the previous time step).

**Block Sorting Rules.** Simulink uses the following basic update rules to sort the blocks:

**•**Each block must be updated before any of the blocks whose direct-feedthrough ports it drives.

This rule ensures that the direct-feedthrough inputs to blocks will be valid when the blocks are updated.

**•**Blocks that do not have direct feedthrough inputs can be updated in any order as long as they are updated before any blocks whose direct-feedthrough inputs they drive.

Putting all blocks that do not have direct-feedthrough ports at the head of the update list in any order satisfies this rule. It thus allows Simulink to ignore these blocks during the sorting process.

The result of applying these rules is an update list in which blocks without direct feedthrough ports appear at the head of the list in no particular order followed by blocks with direct-feedthrough ports in the order required to supply valid inputs to the blocks they drive.

During the sorting process, Simulink checks for and flags the occurrence of algebraic loops, that is, signal loops in which a direct-feedthrough output of a block is connected directly or indirectly to the corresponding direct-feedthrough input of the block. Such loops seemingly create a deadlock condition, because Simulink needs the value of the direct-feedthrough input to compute the output. However, an algebraic loop can represent a set of simultaneous algebraic equations (hence the name) where the block’s input and output are the unknowns. Further, these equations can have valid solutions at each time step. Accordingly, Simulink assumes that loops involving direct-feedthrough ports do, in fact, represent a solvable set of

algebraic equations and attempts to solve them each time the block is updated during a simulation.

**Simulation Loop Phase**

The simulation now enters the simulation loop phase. In this phase, the Simulink engine successively computes the states and outputs of the system at intervals from the simulation start time to the finish time, using information provided by the model. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called the step size. The step size depends on the type of solver used to compute the system’s continuous states, the system’s fundamental sample time, and whether the system’s continuous states have discontinuities. The Simulation Loop phase has two subphrases: the Loop Initialization phase and the Loop Iteration phase. The initialization phase occurs once, at the start of the loop. The iteration phase is repeated once per time step from the simulation start time to the simulation stop time.

At the start of the simulation, the model specifies the initial states and outputs of the system to be simulated. At each step, Simulink computes new values for the system’s inputs, states, and outputs and updates the model to reflect the computed values. At the end of the simulation, the model reflects the final values of the system’s inputs, states, and outputs. Simulink provides data display and logging blocks. You can display and/or log intermediate results by

including these blocks in your model.

**Loop Iteration**

At each time step, the Simulink Engine

**1** Computes the model’s outputs. The Simulink Engine initiates this step by invoking the Simulink model Outputs method. The model Outputs method in turn invokes the model system Outputs method, which invokes the Outputs methods of the blocks that the model contains in the order specified by the Outputs method execution lists generated in the Link phase of the simulation. The system Outputs method passes the following arguments to each block

Outputs method: a pointer to the block’s data structure and to its SimBlock structure. The SimBlock data structures point to information that the Outputs method needs to compute the block’s outputs, including the location of its input buffers and its output buffers.

**2** Computes the model’s states. The Simulink Engine computes a model’s states by invoking a solver. Which solver it invokes depends on whether the model has no states, only discrete

states, only continuous states, or both continuous and discrete states. If the model has only discrete states, the Simulink Engine invokes the discrete solver selected by the user. The solver computes the size of the time step needed to hit the model’s sample times. It then invokes the Update method of the model. The model Update method invokes the Update method of its system, which invokes the Update methods of each of the blocks that the system contains in the order specified by the Update method lists generated in the Link phase.

If the model has only continuous states, the Simulink Engine invokes the continuous solver specified by the model. Depending on the solver, the solver either in turn calls the Derivatives method of the model once or enters a subcycle of minor time steps where the solver repeatedly calls the model’s Outputs methods and Derivatives methods to compute the model’s outputs and derivatives at successive intervals within the major time step. This is done to increase the accuracy of the state computation. The model Outputs method and Derivatives methods in turn invoke their corresponding system methods, which invoke the block Outputs and Derivatives in the order specified by the Outputs and Derivatives methods execution lists generated in the Link phase.

**3** Optionally checks for discontinuities in the continuous states of blocks. Simulink uses a technique called zero-crossing detection to detect discontinuities in continuous states.

**4** Computes the time for the next time step.

**Solvers**

Simulink simulates a dynamic system by computing its states at successive time steps over a specified time span, using information provided by the model. The process of computing the successive states of a system from its model is known as solving the model. No single method of solving a model suffices for all systems. Accordingly, Simulink provides a set of programs, known as *solvers*, that each embody a particular approach to solving a model. The **Configuration** **Parameters** dialog box allows you to choose the solver most suitable for your model.

**Fixed-Step Solvers Versus Variable-Step Solvers**

Simulink solvers fall into two basic categories: fixed-step and variable-step.

***Fixed-step solvers***solve the model at regular time intervals from the beginning to the end of the simulation. The size of the interval is known as the step size. You can specify the step size or let the solver choose the step size. Generally, decreasing the step size increases the accuracy of the results while increasing the time required to simulate the system.

***Variable-step solvers***vary the step size during the simulation, reducing the step size to increase accuracy when a model’s states are changing rapidly and increasing the step size to avoid taking unnecessary steps when the model’s states are changing slowly. Computing the step size adds to the computational overhead at each step but can reduce the total number of steps, and hence simulation time, required to maintain a specified level of accuracy for models with rapidly changing or piecewise continuous states.

**Continuous Versus Discrete Solvers**

Simulink provides both continuous and discrete solvers.

***Continuous solvers***use numerical integration to compute a model’s continuous states at the current time step from the states at previous time steps and the state derivatives. Continuous solvers rely on the model’s blocks to compute the values of the model’s discrete states at each time step.

Mathematicians have developed a wide variety of numerical integration techniques for solving the ordinary differential equations (ODEs) that represent the continuous states of dynamic systems. Simulink provides an extensive set of fixed-step and variable-step continuous solvers, each implementing a specific ODE solution method.

***Discrete solvers***exist primarily to solve purely discrete models. They compute the next simulation time step for a model and nothing else. They do not compute continuous states and they rely on the model’s blocks to update the model’s discrete states.

**Zero-Crossing Detection**

When simulating a dynamic system, Simulink checks for discontinuities in the system’s state variables at each time step, using a technique known as zero-crossing detection. If Simulink detects a discontinuity within the current time step, it determines the precise time at which the discontinuity occurs and takes additional time steps before and after the discontinuity. This section explains why zero-crossing detection is important and how it works. Discontinuities in state variables often coincide with significant events in the evolution of a dynamic system. For example, the instant when a bouncing ball hits the floor coincides with a discontinuity in its position. Because discontinuities often indicate a significant change in a dynamic system, it is important to simulate points of discontinuity precisely. Otherwise, a

simulation could lead to false conclusions about the behavior of the system under investigation. Consider, for example, a simulation of a bouncing ball. If the point at which the ball hits the floor occurs between simulation steps, the simulated ball appears to reverse position in midair. This might lead an investigator to false conclusions about the physics of the bouncing ball. To avoid such misleading conclusions, it is important that simulation steps

occur at points of discontinuity. A simulator that relies purely on solvers to determine simulation times cannot efficiently meet this requirement.

Consider, for example, a fixed-step solver. A fixed-step solver computes the values of state variables at integral multiples of a fixed step size. However, there is no guarantee that a point of discontinuity will occur at an integral multiple of the step size. You could reduce the step size to increase the probability of hitting a discontinuity, but this would greatly increase the

execution time.

A variable-step solver appears to offer a solution. A variable-step solver adjusts the step size dynamically, increasing the step size when a variable is changing slowly and decreasing the step size when the variable changes rapidly. Around a discontinuity, a variable changes extremely rapidly. Thus, in theory, a variable-step solver should be able to hit a discontinuity precisely. The problem is that to locate a discontinuity accurately, a variable-step solver must again take many small steps, greatly slowing down the simulation.

**How Zero-Crossing Detection Works**

Simulink uses a technique known as zero-crossing detection to address this problem. With this technique, a block can register a set of zero-crossing variables with Simulink, each of which is a function of a state variable that can have a discontinuity. The zero-crossing function passes through zero from a positive or negative value when the corresponding discontinuity occurs. At the end of each simulation step, Simulink asks each block that has registered

zero-crossing variables to update the variables. Simulink then checks whether any variable has changed sign since the last step. Such a change indicates that a discontinuity occurred in the current time step.

If any zero crossings are detected, Simulink interpolates between the previous and current values of each variable that changed sign to estimate the times of the zero crossings (e.g., discontinuities). Simulink then steps up to and over each zero crossing in turn. In this way, Simulink avoids simulating exactly at the discontinuity, where the value of the state variable might be undefined.

Zero-crossing detection enables Simulink to simulate discontinuities accurately without resorting to excessively small step sizes. Many Simulink blocks support zero-crossing detection. The result is fast and accurate simulation of all systems, including systems with discontinuities.

**Implementation Details**

An example of a Simulink block that uses zero crossings is the Saturation block. Zero crossings detect these state events in the Saturation block:

**•**The input signal reaches the upper limit.

**•**The input signal leaves the upper limit.

**•**The input signal reaches the lower limit.

**•**The input signal leaves the lower limit.

Simulink blocks that define their own state events are considered to have *intrinsic zero crossings*. If you need explicit notification of a zero-crossing event, use the Hit Crossing block.

The detection of a state event depends on the construction of an internal zero-crossing signal. This signal is not accessible by the block diagram. For the Saturation block, the signal that is used to detect zero crossings for the upper limit is zcSignal = UpperLimit – u, where u is the input signal.

Zero-crossing signals have a direction attribute, which can have these values:

**•***rising* – A zero crossing occurs when a signal rises to or through zero, or when a signal leaves zero and becomes positive.

**•***falling* – A zero crossing occurs when a signal falls to or through zero, or when a signal leaves zero and becomes negative.

**•***either* – A zero crossing occurs if either a rising or falling condition occurs. For the Saturation block’s upper limit, the direction of the zero crossing is *either*. This enables the entering and leaving saturation events to be detected using the same zero-crossing signal.

If the error tolerances are too large, it is possible for Simulink to fail to detect a zero crossing. For example, if a zero crossing occurs within a time step, but the values at the beginning and end of the step do not indicate a sign change, the solver steps over the crossing without detecting it.

The following figure shows a signal that crosses zero. In the first instance, the integrator steps over the event. In the second, the solver detects the event.

