**The lecture 6**

**Algebraic Loops**

Some Simulink blocks have input ports with *direct feedthrough*. This means that the output of these blocks cannot be computed without knowing the values of the signals entering the blocks at these input ports. Some examples of blocks with direct feedthrough inputs are as follows:

**•**The Math Function block

**•**The Gain block

**•**The Integrator block’s initial condition ports

**•**The Product block

**•**The State-Space block when there is a nonzero D matrix

**•**The Sum block

**•**The Transfer Fcn block when the numerator and denominator are of the same order

**•**The Zero-Pole block when there are as many zeros as poles

An *algebraic loop* generally occurs when an input port with direct feedthrough is driven by the output of the same block, either directly, or by a feedback path through other blocks with direct feedthrough. An example of an algebraic loop is this simple scalar loop.



Mathematically, this loop implies that the output of the Sum block is an algebraic state *z* constrained to equal the first input *u* minus *z* (i.e. *z = u – z*). The solution of this simple loop is *z = u/2*, but most algebraic loops cannot be solved by inspection. It is easy to create vector algebraic loops with multiple algebraic state variables *z1*, *z2*, etc., as shown in this model.



The Algebraic Constraint block is a convenient way to model algebraic equations and specify initial guesses. The Algebraic Constraint block constrains its input signal *F(z)* to zero and outputs an algebraic state *z*. This block outputs the value necessary to produce a zero at the input. The output must affect the input through some feedback path. You can provide an initial guess of the algebraic state value in the block’s dialog box to improve algebraic loop solver efficiency. A scalar algebraic loop represents a scalar algebraic equation or constraint of the form *F(z) = 0*, where *z* is the output of one of the blocks in the loop and the function F consists of the feedback path through the other blocks in the loop to the input of the block. In the simple one-block example shown on the previous page, *F(z) = z – (u – z)*. In the vector loop example shown above, the equations are

*z2 + z1 – 1 = 0*

*z2 – z1 – 1 = 0*

Algebraic loops arise when a model includes an algebraic constraint *F(z) = 0*. This constraint might arise as a consequence of the physical interconnectivity of the system you are modeling, or it might arise because you are specifically trying to model a differential/algebraic system (DAE).

When a model contains an algebraic loop, Simulink calls a loop solving routine at each time step. The loop solver performs iterations to determine the solution to the problem (if it can). As a result, models with algebraic loops run slower than models without them. To solve *F(z) = 0*, the Simulink loop solver uses Newton's method with weak line search and rank-one updates to a Jacobian matrix of partial derivatives. Although the method is robust, it is possible to create loops for which the loop solver will not converge without a good initial guess for the algebraic states *z*. You can specify an initial guess for a line in an algebraic loop by placing an IC block (which is normally used to specify an initial condition for a signal) on that

line. As shown above, another way to specify an initial guess for a line in an algebraic loop is to use an Algebraic Constraint block.

Whenever possible, use an IC block or an Algebraic Constraint block to specify



In this case, the input at the u2 port of the adder subsystem is equal to the subsystem’s output at the current time step for every time step. The mathematical representation of this system

z = z + 1 reveals that it has no mathematically valid solution.

**Purely Discrete Systems**

Purely discrete systems can be simulated using any of the solvers; there is no difference in the solutions. To generate output points only at the sample hits, choose one of the discrete solvers.

**Multirate Systems**

Multirate systems contain blocks that are sampled at different rates. These systems can be modeled with discrete blocks or with both discrete and continuous blocks. For example, consider this simple multirate discrete model.



For this example the DTF1 Discrete Transfer Fcn block’s **Sample time** is set to [1 0.1], which gives it an offset of 0.1. The DTF2 Discrete Transfer Fcn block’s **Sample time** is set to 0.7, with no offset.

Starting the simulation and plotting the outputs using the stairs function

[t,x,y] = sim('multirate', 3);

stairs(t,y)

produces this plot



For the DTF1 block, which has an offset of 0.1, there is no output until t = 0.1. Because the initial conditions of the transfer functions are zero, the output of DTF1, y(1), is zero before this time.

**Determining Step Size for Discrete Systems**

Simulating a discrete system requires that the simulator take a simulation step at every *sample time hit*, that is, at integer multiples of the system’s shortest sample time. Otherwise, the simulator might miss key transitions in the system’s states. Simulink avoids this by choosing a simulation step size to ensure that steps coincide with sample time hits. The step size that Simulink chooses depends on the system’s fundamental sample time and the type of solver used to simulate the system.

The *fundamental sample time* of a discrete system is the greatest integer divisor of the system’s actual sample times. For example, suppose that a system has sample times of 0.25 and 0.5 second. The fundamental sample time in this case is 0.25 second. Suppose, instead, the sample times are 0.5 and 0.75 second. In this case, the fundamental sample time is again 0.25 second. You can direct Simulink to use either a fixed-step or a variable-step discrete solver to solve a discrete system. A fixed-step solver sets the simulation step size equal to the discrete system’s fundamental sample time. A variable-step solver varies the step size to equal the distance between actual sample time hits. The following diagram illustrates the difference between a fixed-step and a variable-size solver.



In the diagram, arrows indicate simulation steps and circles represent sample time hits. As the diagram illustrates, a variable-step solver requires fewer simulation steps to simulate a system, if the fundamental sample time is less than any of the actual sample times of the system being simulated. On the other hand, a fixed-step solver requires less memory to implement and is faster if one of the system’s sample times is fundamental.

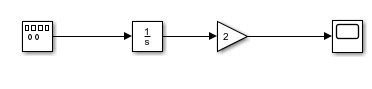
**Sample Time Propagation**

When updating a model’s diagram, for example, at the beginning of a simulation, Simulink uses a process called sample time propagation to determine the sample times of blocks that inherit their sample times. The figure below illustrates a Discrete Filter block with a sample time of Ts driving a Gain block.



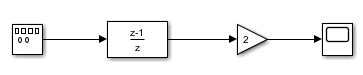
Because the Gain block’s output is simply the input multiplied by a constant, its output changes at the same rate as the filter. In other words, the Gain block has an effective sample rate equal to that of the filter’s sample rate. This is the fundamental mechanism behind sample time propagation in Simulink. Simulink assigns an inherited sample time to a block based on the sample times of the blocks connected to its inputs. If all the inputs have the same sample time, Simulink assigns that sample time to the block. If the inputs have different sample times, if all sample times are integer multiples of the fastest sample time, the block is assigned the sample time of the fastest input. If a variable-step solver is being used, the block is assigned the continuous sample time. If a fixed-step solver is being used and the greatest common divisor of the sample times (the fundamental sample time) can be computed, it is used. Otherwise continuous is used.

Under some circumstances, Simulink also back propagates sample times to source blocks if it can do so without affecting the output of a simulation. For instance, in the model below, Simulink recognizes that the Signal Generator block is driving a Discrete-Time Integrator block, so it assigns the Signal Generator block and the Gain block the same sample time as the Discrete-Time Integrator block.



You can verify this by selecting **Sample Time Colors** from the Simulink **Format** menu and noting that all blocks are colored red. Because the Discrete-Time Integrator block only looks at its input at its sample times, this change does not affect the outcome of the simulation but does result in a performance improvement.

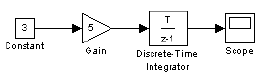
Replacing the Discrete-Time Integrator block with a continuous Integrator block, as shown below, and recoloring the model by choosing **Update diagram** from the **Edit** menu cause the Signal Generator and Gain blocks to change to continuous blocks, as indicated by their being colored black.



**Invariant Constants**

Simulink by default assigns Constant blocks a sample time of infinity, also referred to as a *constant sample time*. This means that the outputs of any blocks that inherit a constant sample time from a Constant block do not change during the simulation unless the parameters are explicitly modified by the model user.

For example, in this model, both the Constant and Gain blocks have constant sample time.



Because Simulink supports the ability to change block parameters during a simulation, all blocks, even blocks having constant sample time, must generate their output at the model’s effective sample time.

Because of this feature, *all* blocks compute their output at each sample time hit, or, in the case of purely continuous systems, at every simulation step. For blocks having constant sample time whose parameters do not change during a simulation, evaluating these blocks during the simulation is inefficient and slows down the simulation.

You can set the inline parameters option to remove all blocks having constant sample times from the simulation “loop.” The effect of this feature is twofold. First, parameters for these blocks cannot be changed during a simulation. Second, simulation speed is improved. The speed improvement depends on model complexity, the number of blocks with constant sample time, and the effective sampling rate of the simulation.