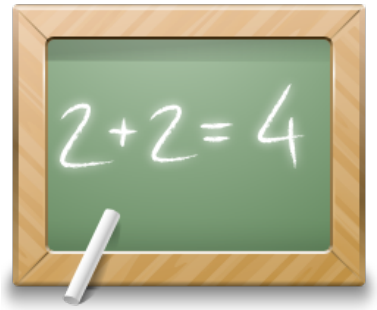
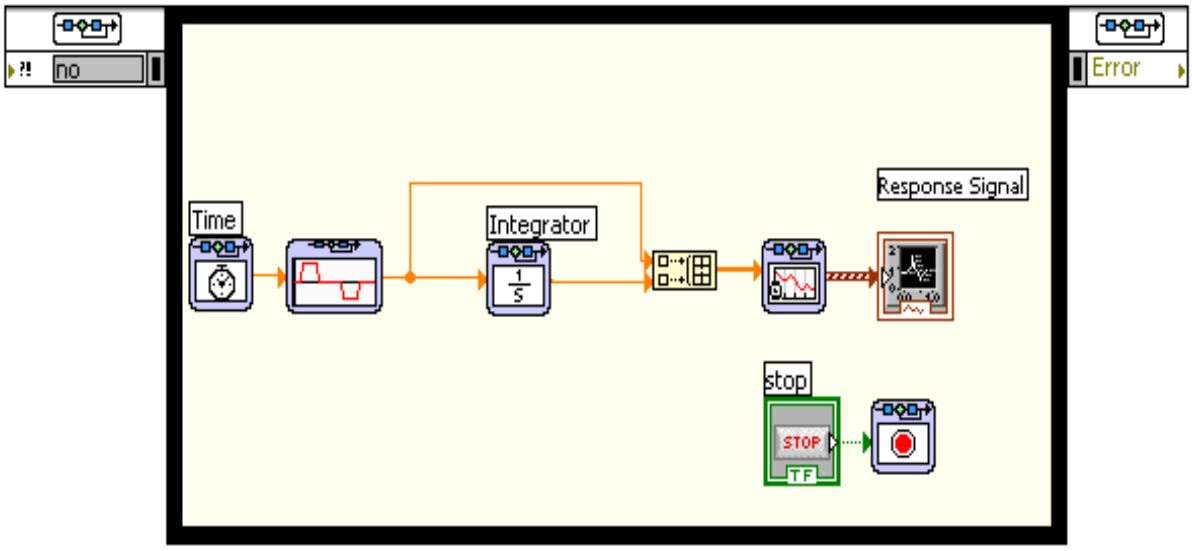
https://www.halvorsen.blog



Control and Simulation in LabVIEW

Hans-Petter Halvorsen



Control and Simulation in LabVIEW

Hans-Petter Halvorsen

E-Mail: hans.p.halvorsen@usn.no

Web: https://www.halvorsen.blog

https://www.halvorsen.blog

Preface

This document explains the basic concepts of using LabVIEW for **Control and Simulation** purposes.

For more information about LabVIEW, visit my Blog: https://www.halvorsen.blog.

You need the following software:

- LabVIEW
- LabVIEW Control Design and Simulation Module
- LabVIEW MathScript RT Module
- NI-DAQmx
- NI Measurement & Automation Explorer

Table of Contents

1 Introduction to LabVIEW

LabVIEW (short for **Lab**oratory **V**irtual **I**nstrumentation **E**ngineering **W**orkbench) is a platform and development environment for a visual programming language from National Instruments. The graphical language is named "G". Originally released for the Apple Macintosh in 1986, LabVIEW is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms including Microsoft Windows, various flavors of Linux, and Mac OS X. Visit National Instruments at www.ni.com.

The code files have the extension ".vi", which is an abbreviation for "Virtual Instrument". LabVIEW offers lots of additional Add-Ons and Toolkits.

1.1 Dataflow programming

The programming language used in LabVIEW, also referred to as G, is a dataflow programming language. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

1.2 Graphical programming

LabVIEW ties the creation of user interfaces (called front panels) into the development cycle. LabVIEW programs/subroutines are called virtual instruments (VIs). Each VI has three components: a block diagram, a front panel, and a connector panel. The last is used to represent the VI in the block diagrams of other, calling VIs. Controls and indicators on the front panel allow an operator to input data into or extract data from a running virtual instrument. However, the front panel can also serve as a programmatic interface. Thus a virtual instrument can either be run as a program, with the front panel serving as a user interface, or, when dropped as a node onto the block diagram, the front panel defines the inputs and outputs for the given node through the connector pane. This implies each VI can be easily tested before being embedded as a subroutine into a larger program.

The graphical approach also allows non-programmers to build programs simply by dragging and dropping virtual representations of lab equipment with which they are already familiar. The LabVIEW programming environment, with the included examples and the documentation, makes it simple to create small applications. This is a benefit on one side, but there is also a certain danger of underestimating the expertise needed for good quality "G" programming. For complex algorithms or large-scale code, it is important that the programmer possess an extensive knowledge of the special LabVIEW syntax and the topology of its memory management. The most advanced LabVIEW development systems offer the possibility of building stand-alone applications. Furthermore, it is possible to create distributed applications, which communicate by a client/server scheme, and are therefore easier to implement due to the inherently parallel nature of G-code.

1.3 Benefits

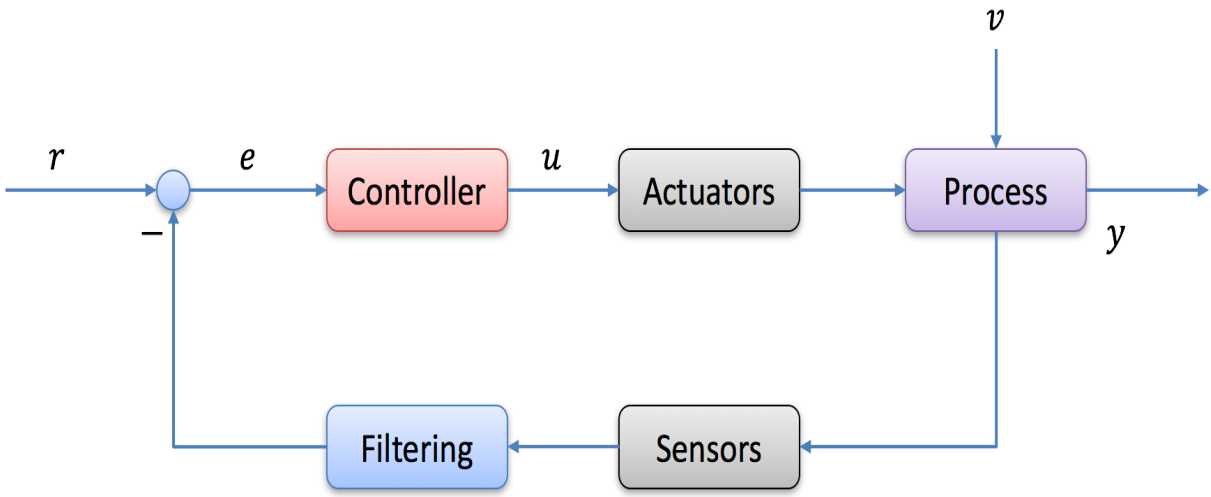
One benefit of LabVIEW over other development environments is the extensive support for accessing instrumentation hardware. Drivers and abstraction layers for many different types of instruments and buses are included or are available for inclusion. These present themselves as graphical nodes. The abstraction layers offer standard software interfaces to communicate with hardware devices. The provided driver interfaces save program development time. The sales pitch of National Instruments is, therefore, that even people with limited coding experience can write programs and deploy test solutions in a reduced time frame when compared to more conventional or competing systems. A new hardware driver topology (DAQmxBase), which consists mainly of G-coded components with only a few register calls through NI Measurement Hardware DDK (Driver Development Kit) functions, provides platform independent hardware access to numerous data acquisition and instrumentation devices. The DAQmxBase driver is available for LabVIEW on Windows, Mac OS X and Linux platforms.

2 Introduction to Control and Simulation

Control design is a process that involves developing mathematical models that describe a physical system, analyzing the models to learn about their dynamic characteristics, and creating a controller to achieve certain dynamic characteristics.

Simulation is a process that involves using software to recreate and analyze the behavior of dynamic systems. You use the simulation process to lower product development costs by accelerating product development. You also use the simulation process to provide insight into the behavior of dynamic systems you cannot replicate conveniently in the laboratory.

Below we see a closed-loop feedback control system:



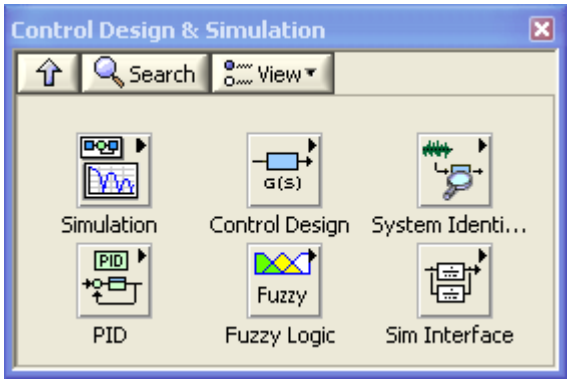
3 Control and Simulation in LabVIEW

LabVIEW has several additional modules and Toolkits for Control and Simulation purposes, e.g., "**LabVIEW Control Design and Simulation Module**", "**LabVIEW PID and Fuzzy Logic Toolkit**", "**LabVIEW System Identification Toolkit**" and "LabVIEW Simulation Interface Toolkit". LabVIEW MathScript is also useful for Control Design and Simulation.

- LabVIEW Control Design and Simulation Module
- LabVIEW PID and Fuzzy Logic Toolkit
- LabVIEW System Identification Toolkit
- LabVIEW Simulation Interface Toolkit

This tutorial will focus on the main aspects in these modules and toolkits.

All VIs related to these modules and toolkits are placed in the Control Design and Simulation Toolkit:



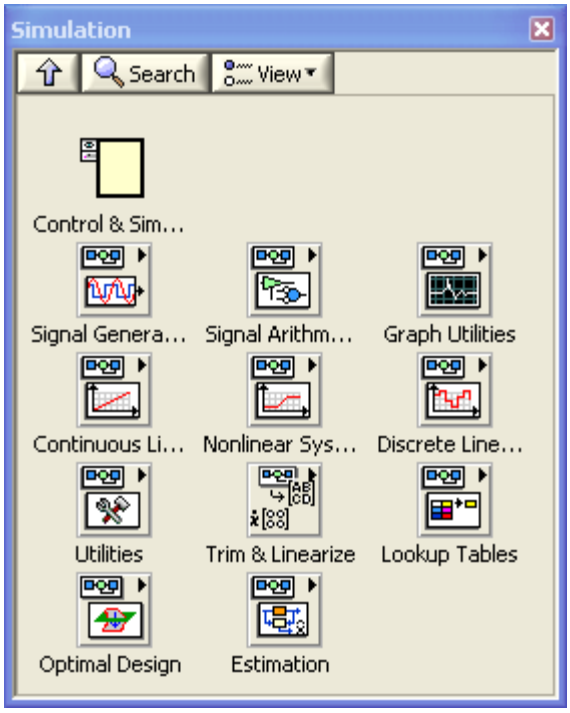
3.1 LabVIEW Control Design and Simulation Module

With LabVIEW Control Design and Simulation Module you can construct plant and control models using transfer function, state-space, or zero-pole-gain. Analyze system performance with tools such as step response, pole-zero maps, and Bode plots. Simulate linear, nonlinear, and discrete systems with a wide option of solvers. With the NI LabVIEW Control Design and

Simulation Module, you can analyze open-loop model behavior, design closed-loop controllers, simulate online and offline systems, and conduct physical implementations.

3.1.1 Simulation

The **Simulation** palette in LabVIEW:

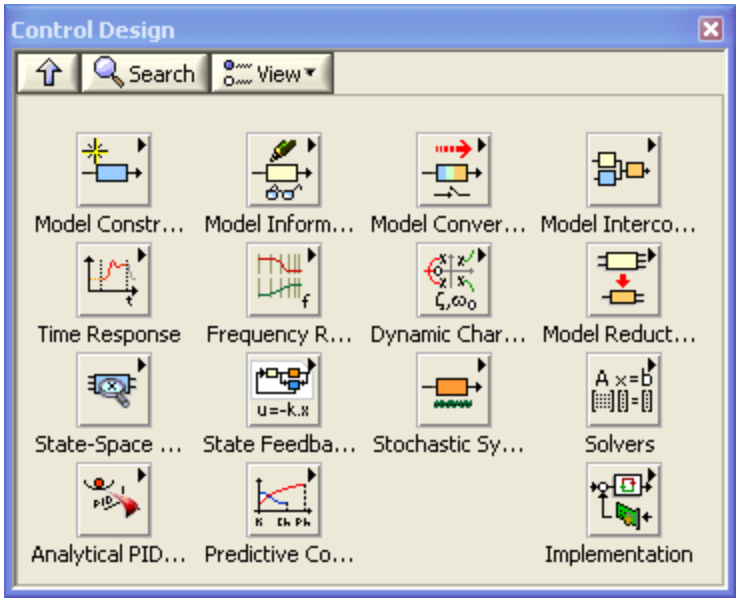


The main features in the Simulation palette are:

- **Control and Simulation Loop** - You must place all Simulation functions within a Control & Simulation Loop or in a simulation subsystem.
- **Continuous Linear Systems Functions** - Use the Continuous Linear Systems functions to represent continuous linear systems of differential equations on the simulation diagram.
- **Signal Arithmetic Functions** - Use the Signal Arithmetic functions to perform basic arithmetic operations on signals in a simulation system.

3.1.2 Control Design

The **Control Design** palette in LabVIEW**:**

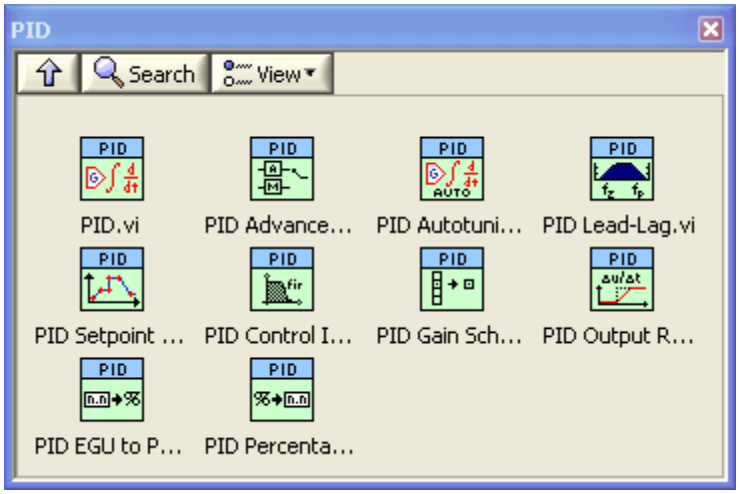


3.2 LabVIEW PID and Fuzzy Logic Toolkit

The NI LabVIEW PID and Fuzzy Logic Toolkit add control algorithms to LabVIEW. By combining the PID and fuzzy logic control functions in this toolkit with the math and logic functions in LabVIEW software, you can quickly develop programs for automated control. You may integrate these control tools with the power of data acquisition.

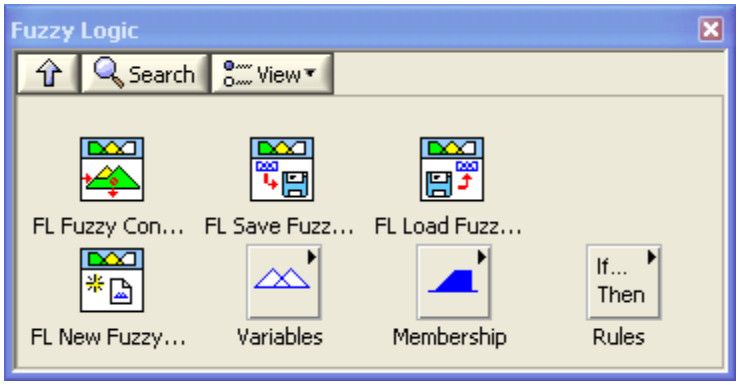
3.2.1 PID Control

The **PID** palette in LabVIEW**:**



3.2.2 Fuzzy Logic

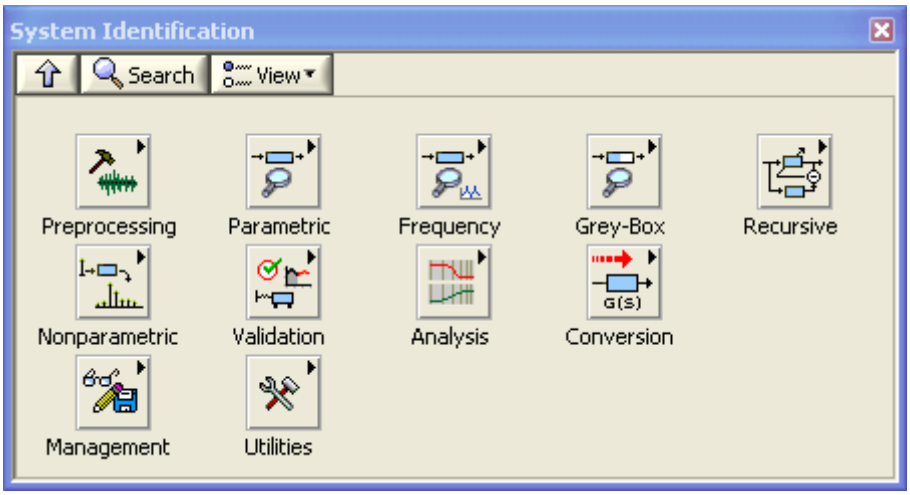
The **Fuzzy Logic** palette in LabVIEW**:**



3.3 LabVIEW System Identification Toolkit

The “LabVIEW System Identification Toolkit” combines data acquisition tools with system identification algorithms for plant modeling. You can use the LabVIEW System Identification Toolkit to find empirical models from real plant stimulus-response information.

The **System Identification** palette in LabVIEW:



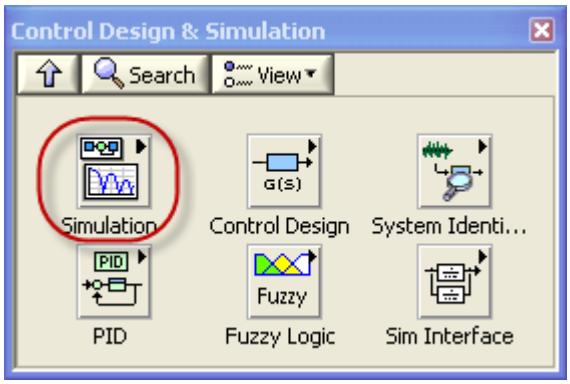
4 Simulation

Simulation is a process that involves using software to recreate and analyze the behavior of dynamic systems. You use the simulation process to lower product development costs by accelerating product development. You also use the simulation process to provide insight into the behavior of dynamic systems you cannot replicate conveniently in the laboratory. For example, simulating a jet engine saves time, labor, and money compared to building, testing, and rebuilding an actual jet engine. You can use the LabVIEW Control Design and Simulation Module to simulate a dynamic system or a component of a dynamic system. For example, you can simulate only the plant while using hardware for the controller, actuators, and sensors (Hardware-in-the-loop Simulation).

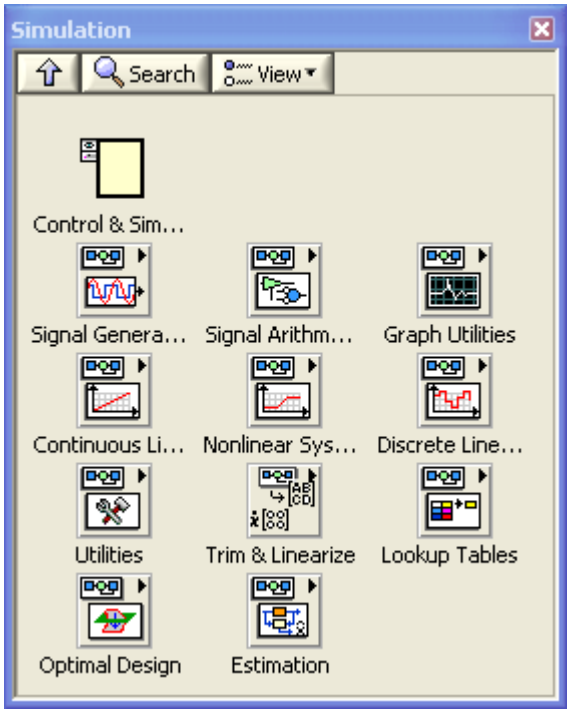
A dynamic system model is a differential or difference equation that describes the behavior of the dynamic system.

4.1 Simulation in LabVIEW

Use the Simulation VIs and functions to create simulation applications in LabVIEW. In the Control Design & Simulation palette we have the **Simulation** Sub palette:



Below we see the Simulation Sub palette:

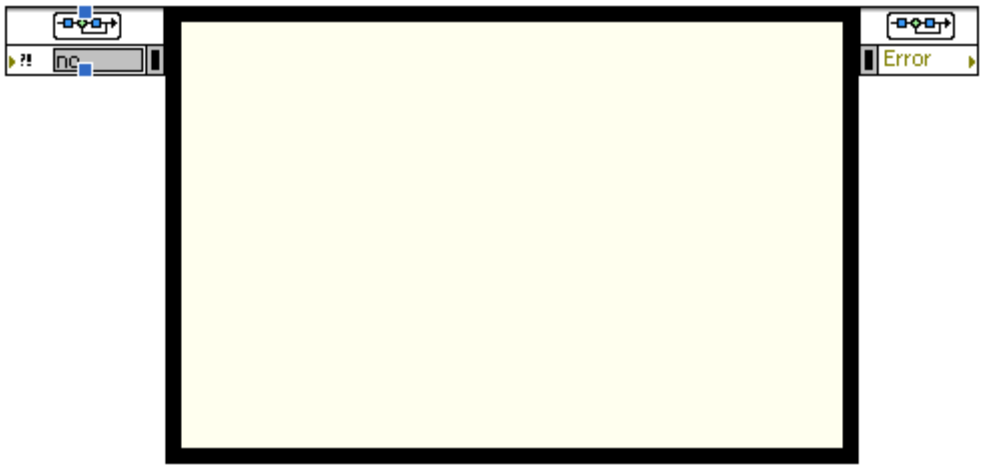


Note! All the "Blocks" in the Simulation palette are not SubVIs, i.e., we cannot double-click on them and open the Block Diagram because they have none. All the Blocks in the Simulation palette must be used inside the Control and Simulation Loop (explained below).

Control and Simulation Loop:

In the "**Simulation**" Sub palette we have the "Control and Simulation Loop" which is very useful in simulations:

You must place all Simulation functions within a Control & Simulation Loop or in a simulation subsystem. You also can place simulation subsystems within a Control & Simulation Loop or another simulation subsystem, or you can place simulation subsystems on a block diagram outside a Control & Simulation Loop or run the simulation subsystems as stand-alone VIs.



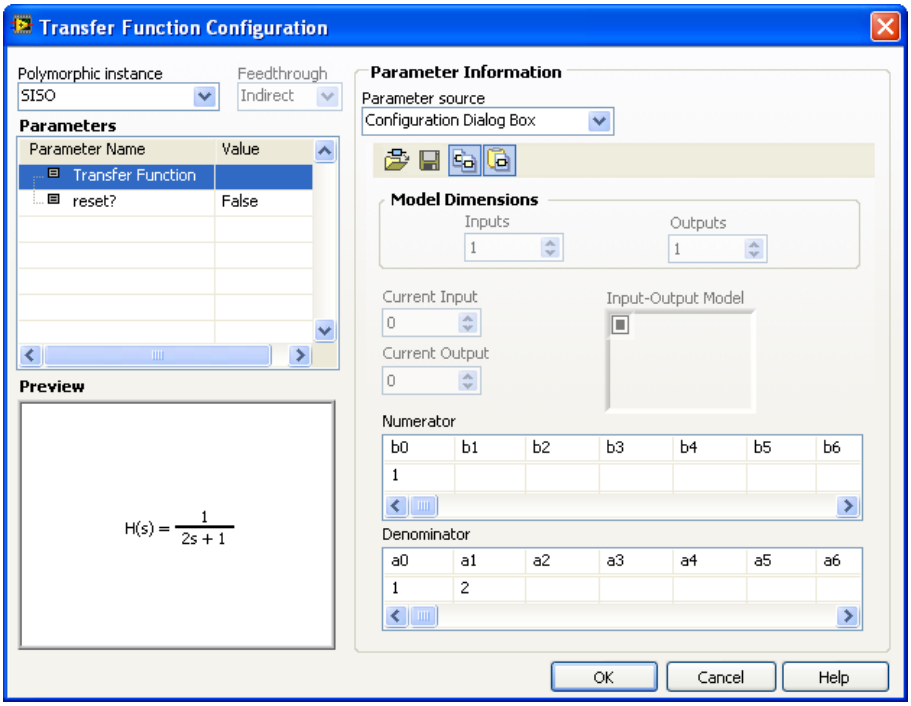
The Control & Simulation Loop has an Input Node (upper left corner) and an Output Node (upper right corner). Use the Input Node to configure simulation parameters programmatically. You also can configure these parameters interactively using the Configure Simulation Parameters dialog box. Access this dialog box by double-clicking the Input Node or by right-clicking the border and selecting Configure Simulation Parameters from the shortcut menu.

Configuration:

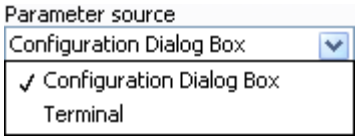
When you place these blocks on the diagram you may double-click or right-click and then select “Configuration…”

Example: Configuration Dialog box

For the “Transfer Function” (Simulation → Continuous Linear Systems) block we have the following Configuration window:



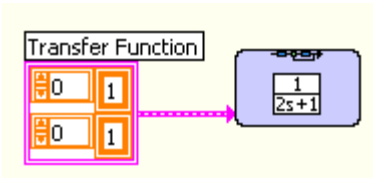
All the different blocks have their own different Configuration window.



In the Parameter source you may select between:

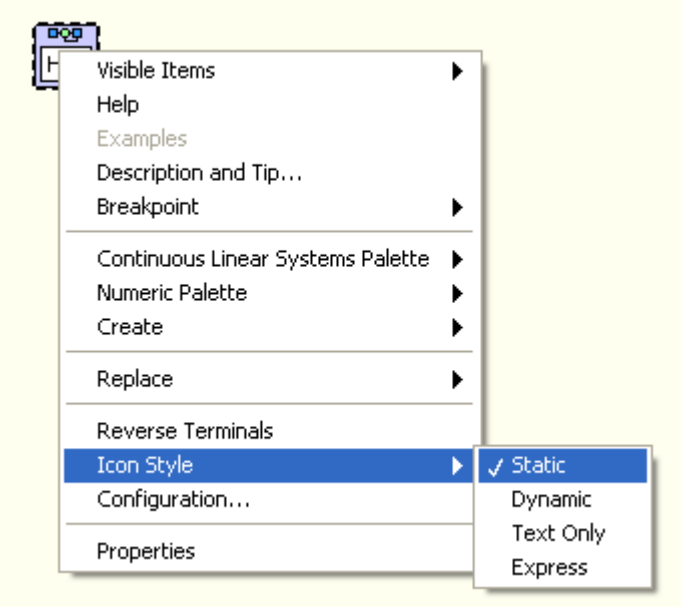
- Configuration Dialog Box
- Terminal

If you select “Configuration Dialog Box” you enter the configuration in the Configuration window like we see above, while if you select “Terminal” that specific configuration is set from the Block Diagram like this:



Icon Style:

When you place the block on the block diagram you may select how that should appear. Right-click on the block/icon and select “Icon Style”:



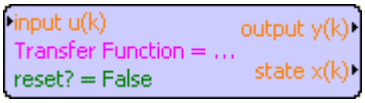
Example: Icon Style

For the “Transfer Function” (Simulation → Continuous Linear Systems) block we have the following different icon styles:

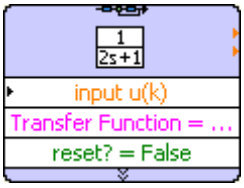
Static:

Dynamic:

Text Only:



Express:

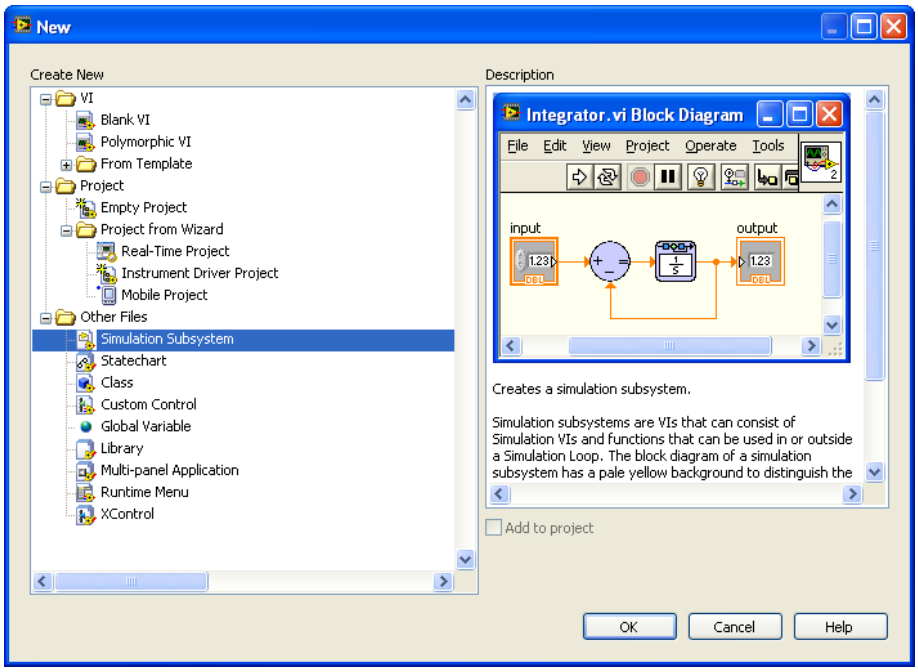


We see for the Dynamic and Express styles that the appearance changes according to configuration parameters we set.

I personally prefer the “static” icon style because it does not require lots of space on the diagram.

4.2 Simulation Subsystem

You may create a **Simulation Subsystem** (File → New…):

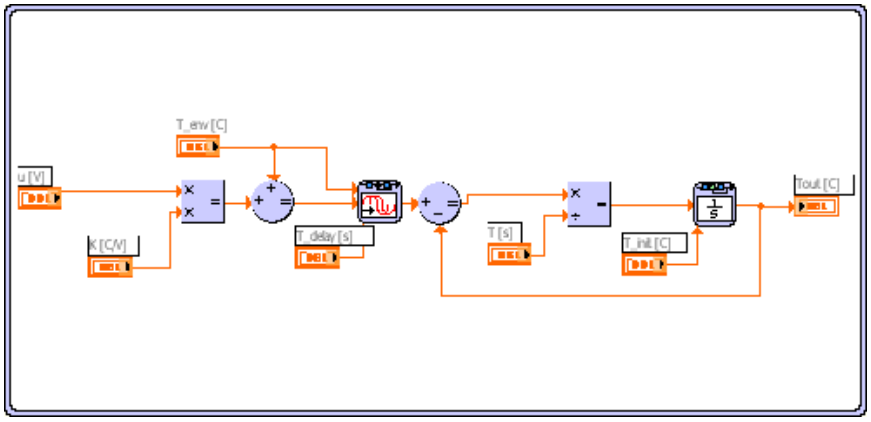


The Simulation Subsystem is very useful when dealing with larger simulation systems in order to create a more structured code. I recommend that you (always) use this feature.

The Simulation Subsystem is almost equal to a normal LabVIEW Block Diagram but notice the background color is slightly darker.

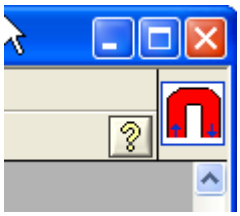
Note! In order to open the Simulation Subsystem, right-click and select “Open Subsystem”.

The Simulation Subsystem may also be represented by different icons. If you select “dynamic” icon style, you will see a “miniature” version of the subsystem like this:



 You may drag in the corner in order to increase or decrease the dynamic icon.

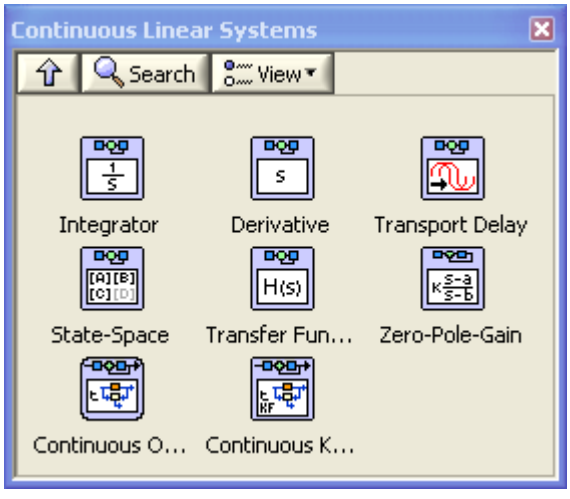
If you select “static” icon style you see the icon you created with the Icon Editor.



Like this: 

4.3 Continuous Linear Systems

In the “**Continuous Linear Systems**” Sub palette we want to create a simulation model:

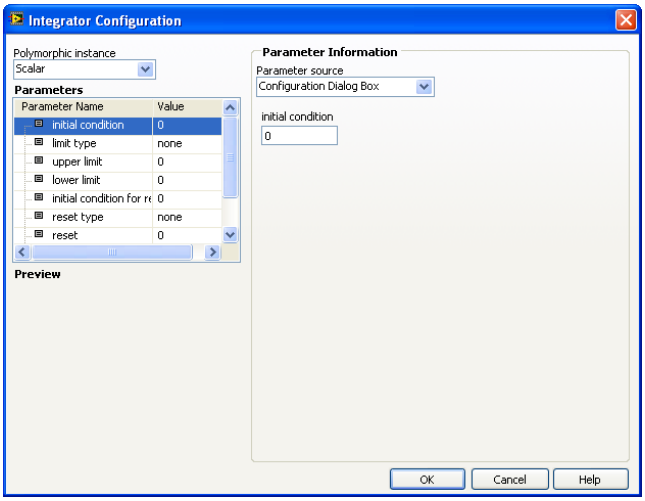


The most used blocks probably are Integrator, Transport Delay, State-Space and Transfer Function.

When you place these blocks on the diagram you may double-click or right-click and then select “Configuration…”

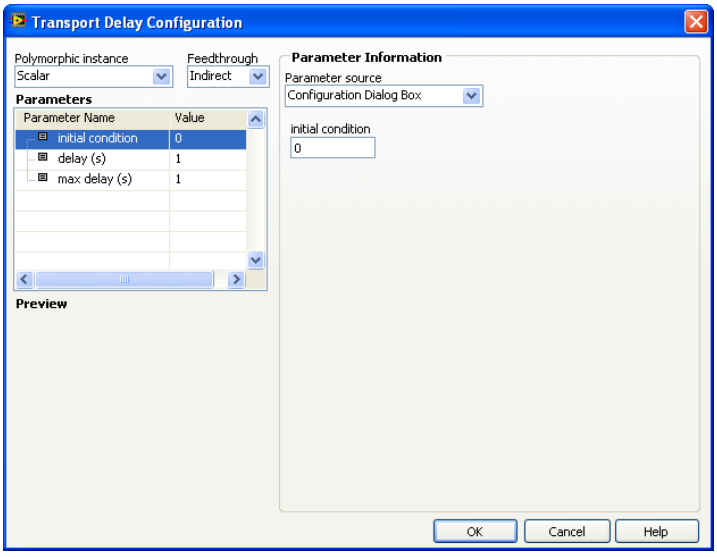
 **Integrator** - Integrates a continuous input signal using the ordinary differential equation (ODE) solver you specify for the simulation.

The Configuration window for the Integrator block looks like this:



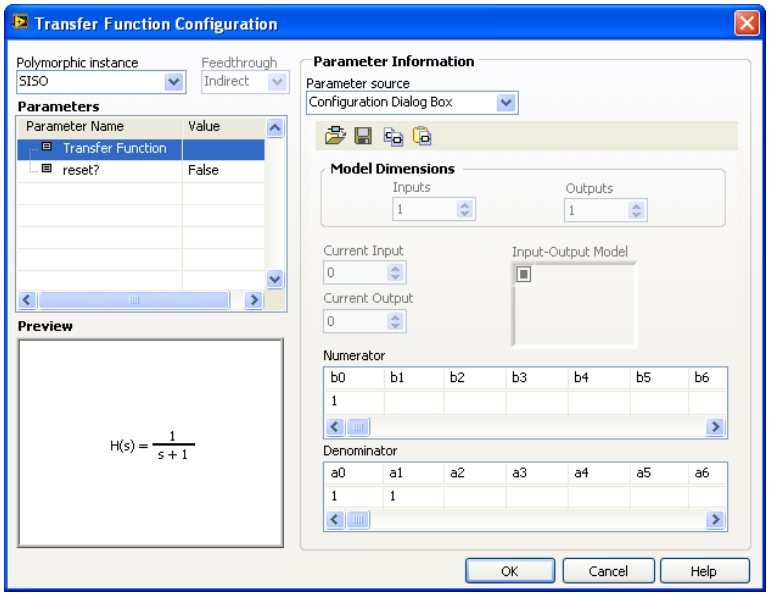
 **Transport Delay** - Delays the input signal by the amount of time you specify.

The Configuration window for the Transport Delay block looks like this:



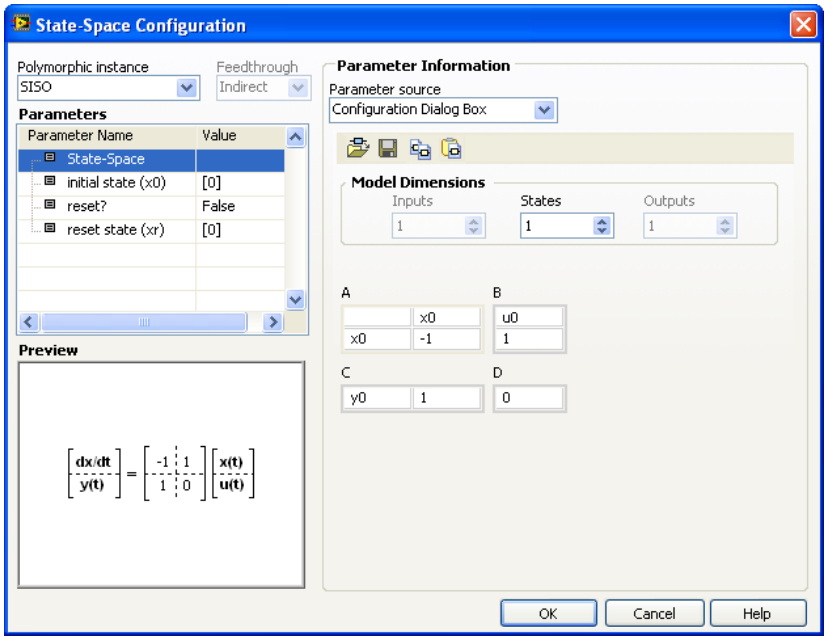
 **Transfer Function** - Implements a system model in transfer function form. You define the system model by specifying the Numerator and Denominator of the transfer function equation.

The Configuration window for the Transfer Function block looks like this:



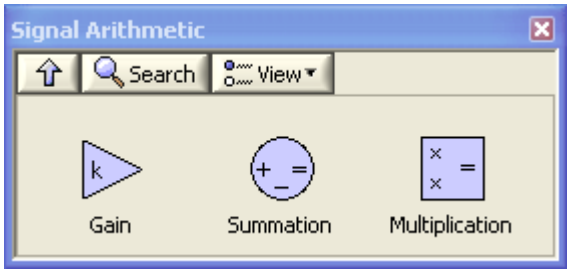
State-Space - Implements a system model in state-space form. You define the system model by specifying the input, output, state, and direct transmission matrices.

The Configuration window for the State-Space block looks like this:



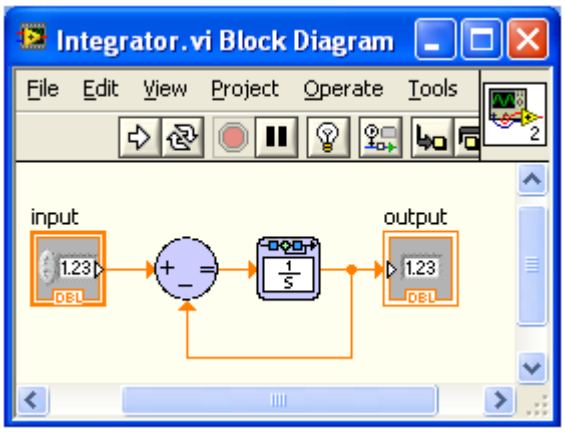
Signal Arithmetic:

The "**Signal Arithmetic**" Sub palette is also useful when creating a simulation model:



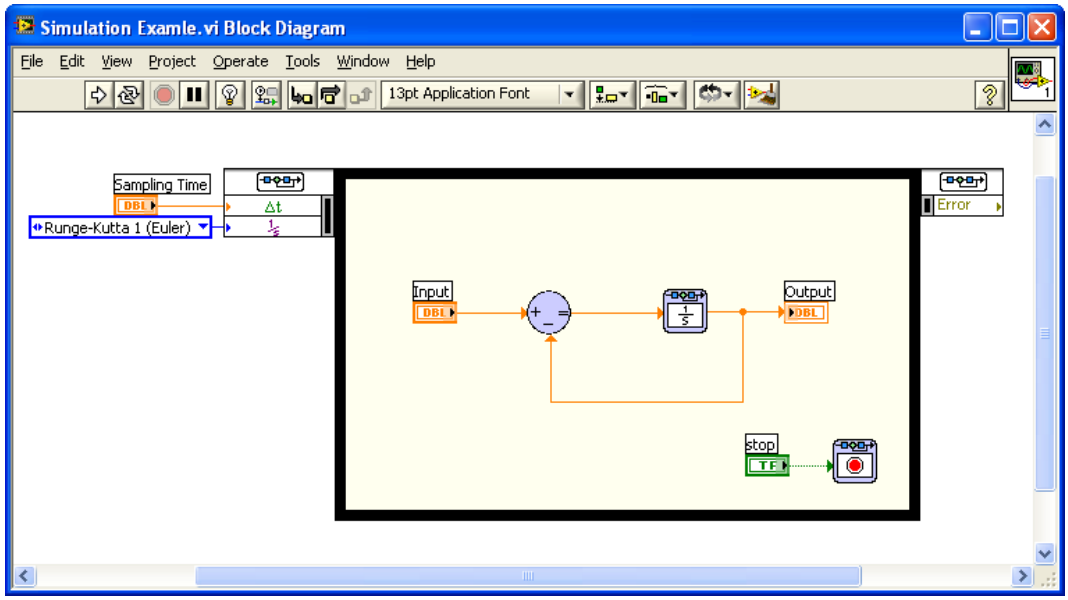
Example: Simulation Model

Below we see an example of a simulation model created in LabVIEW.



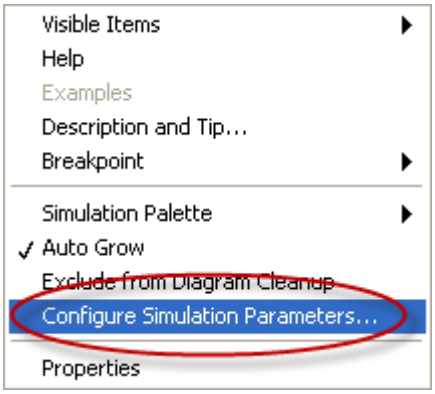
Example: Simulation

Below we see an example of a simulation model using the Control and Simulation Loop.

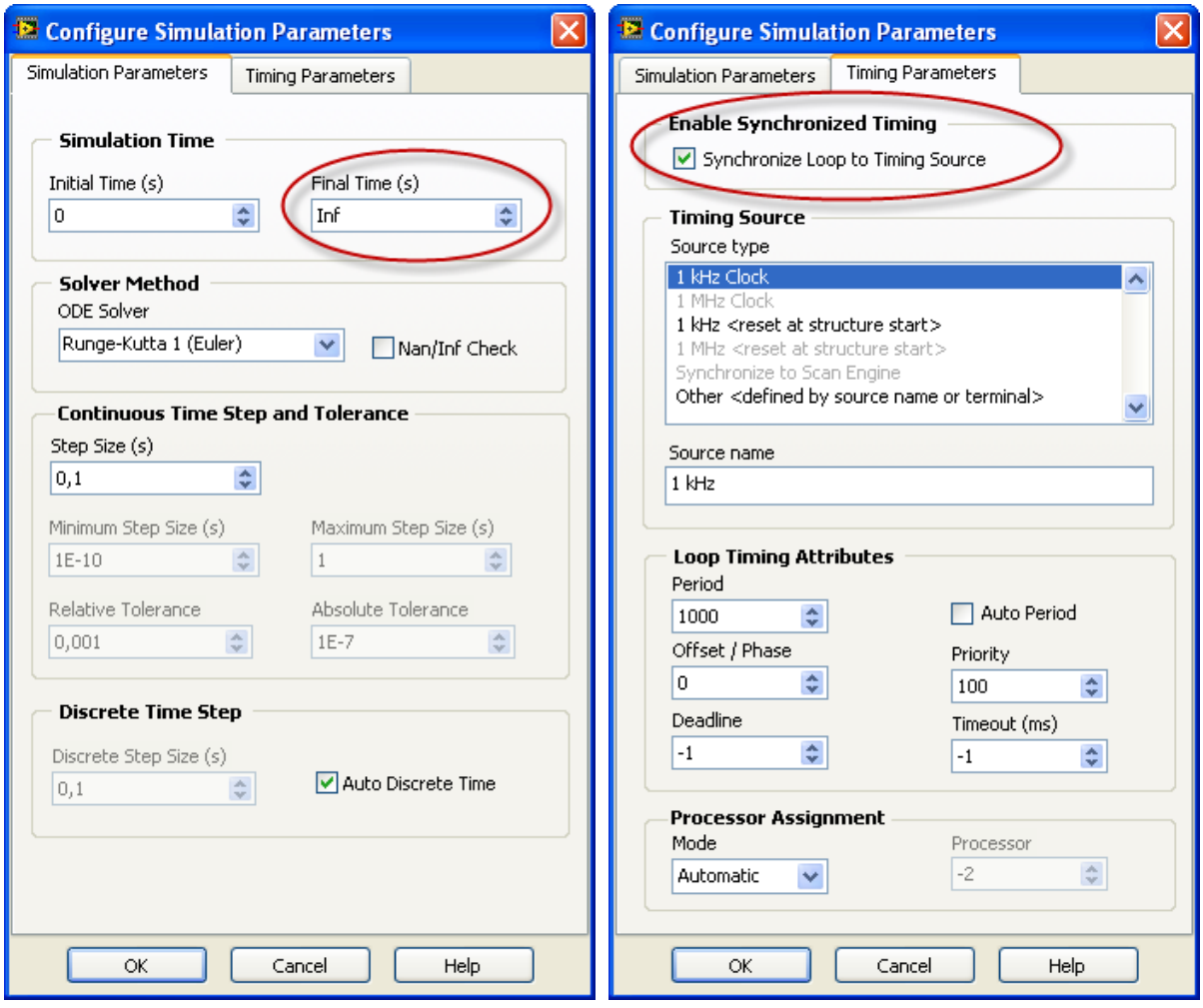


Notice the following:

Click on the border of the simulation loop and select "**Configure Simulation Parameters…**"



The following window appears (Configure Simulation Parameters):



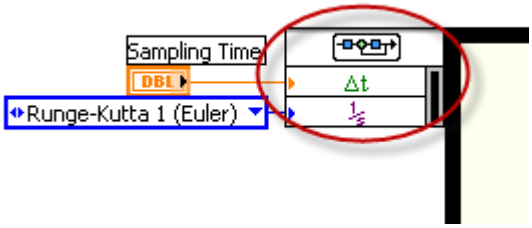
In this window you set some Parameters regarding the simulation, some important are:

- **Final Time (s)** – set how long the simulation should last. For an infinite time set "**Inf**".
- **Enable Synchronized Timing** - Specifies that you want to synchronize the timing of the Control & Simulation Loop to a timing source. To enable synchronization, place a

checkmark in this checkbox and then choose a timing source from the Source type list box.

Click the Help button for more details.

You may also set some of these Parameters in the Block Diagram:



You may use the mouse to increase the numbers of Parameters and right-click and select "Select Input".

Exercises

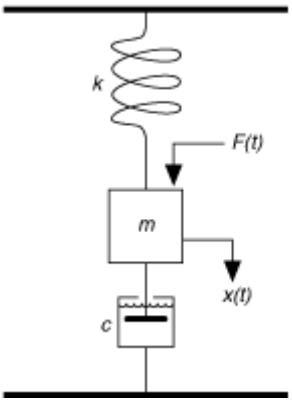
Exercise: Simulation of a spring-mass damper system

In this exercise you will construct a simulation diagram that represents the behavior of a dynamic system. You will simulate a spring-mass damper system.

$$F(t) - c\dot{x}(t) - kx(t) = m\ddot{x}(t)$$

where t is the simulation time, F(t) is an external force applied to the system, c is the damping constant of the spring, k is the stiffness of the spring, m is a mass, and x(t) is the position of the mass. $\dot{x}$ is the first derivative of the position, which equals the velocity of the mass. $\ddot{x}$ is the second derivative of the position, which equals the acceleration of the mass.

The following figure shows this dynamic system.



The goal is to view the position x(t) of the mass m with respect to time t. You can calculate the position by integrating the velocity of the mass. You can calculate the velocity by integrating the acceleration of the mass. If you know the force and mass, you can calculate this acceleration by using Newton's Second Law of Motion, given by the following equation:

Force = Mass × Acceleration

Therefore,

Acceleration = Force / Mass

Substituting terms from the differential equation above yields the following equation:

$$\ddot{x} = \frac{1}{m}(F - c\dot{x} - kx)$$

You will construct a simulation diagram that iterates the following steps over a period of time.

Creating the Simulation Diagram

You create a simulation diagram by placing a Control & Simulation Loop on the LabVIEW block diagram.

1. Launch LabVIEW and select File»New VI to create a new, blank VI.
2. Select Window»Show Block Diagram to view the block diagram. You also can press the <Ctrl-E> keys to view the block diagram.
3. If you are not already viewing the Functions palette, select View»Functions Palette to display this palette.
4. Select Control Design & Simulation»Simulation to view the Simulation palette.
5. Click the Control & Simulation Loop icon.
6. Move the cursor over the block diagram. Click to place the top left corner of the loop, drag the cursor diagonally to establish the size of the loop, and click again to place the loop on the block diagram.

The simulation diagram is the area enclosed by the Control & Simulation Loop. Notice the simulation diagram has a pale yellow background to distinguish it from the rest of the block diagram. You can resize the Control & Simulation Loop by dragging its borders.

Configuring Simulation Parameters

The Control & Simulation Loop contains the parameters that define how the simulation executes. Complete the following steps to view and configure these simulation parameters.

1. Double-click the Input Node, attached to the left side of the Control & Simulation Loop, to display the Configure Simulation Parameters dialog box. You also can right-

click the loop border and select Configure Simulation Parameters from the shortcut menu.

2. Ensure the value of the **Final Time (s)** numeric control is 10, which specifies that this tutorial simulates ten seconds of time.
3. Click the ODE Solver pull-down menu to view the list of ODE solvers the Control Design and Simulation Module includes. If the term (variable) appears next to an ODE solver, that solver has a variable step size. The other ODE solvers have a fixed step size. Ensure a checkmark is beside the default ODE solver **Runge-Kutta 23 (variable)**.
4. Because this ODE solver is a variable step-size solver, you can specify the **Minimum Step Size (s)** and **Maximum Step Size (s)** this ODE solver can take. Enter 0.01 in the Maximum Step Size (s) numeric control to limit the size of the time step this ODE solver can take.
5. Click the Timing Parameters tab to access parameters that control how often the simulation executes.
6. Ensure the Synchronize Loop to Timing Source checkbox does not contain a checkmark. This option specifies that the simulation executes without any timing restrictions. Use this option when you want the simulation to run as fast as possible. If you are running this simulation in real-time, you can place a checkmark in this checkbox and configure how often the simulation executes.
7. Click the OK button to save changes and return to the simulation diagram.

Building the Simulation

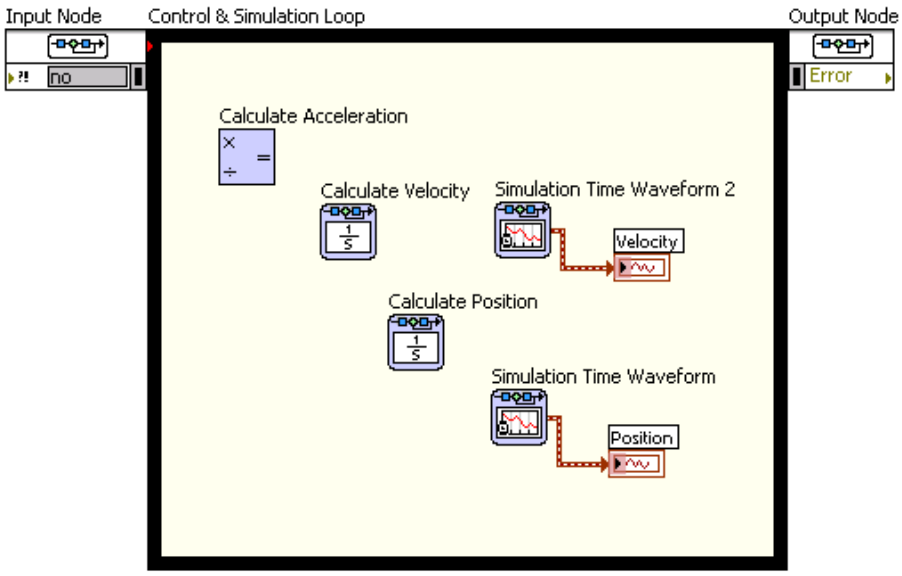
The next step is to build the simulation by placing Simulation functions on the simulation diagram and wiring these functions together. Note that you can place most Simulation functions only on the simulation diagram, that is, you cannot place Simulation functions on a LabVIEW block diagram. Complete the following steps to build the simulation of this dynamic system.

Placing Functions on the Simulation Diagram

1. Open the Simulation palette.
2. Select the **Signal Arithmetic** palette and place a **Multiplication** function on the simulation diagram. You will use this function to divide the force by the mass to calculate the acceleration.
3. Double-click the Multiplication function to display the Multiplication Configuration dialog box. You can double-click most Simulation functions to view and change the parameters of that function.
4. The function currently displays two × symbols on the left side of the dialog box. This setting specifies that both incoming signals are multiplied together. Click the bottom × symbol to change it to a ÷ symbol. This Multiplication function now divides the top signal by the bottom signal.
5. Click the OK button to save changes and return to the simulation diagram.

6. Right-click the Multiplication function and select Visible Items»Label from the shortcut menu. Double-click the Multiplication label and enter Calculate Acceleration as the new label.
7. Return to the Simulation palette and select the **Continuous Linear Systems** palette.
8. Place an Integrator function on the simulation diagram. You will use this function to calculate velocity by integrating acceleration.
9. Label this Integrator function Calculate Velocity.
10. Press the <Ctrl> key and click and drag the Integrator function to another location on the simulation diagram. This action creates a copy of the Integrator function, which you will use to calculate position by integrating velocity. Label this new Integrator function Calculate Position.
11. Select the Graph Utilities palette and place two **SimTime Waveform** functions on the simulation diagram. You will use these functions to view the results of the simulation over time.
12. Each SimTime Waveform function has an associated Waveform Chart. Label the first waveform chart Velocity and the second waveform chart Position.
13. Arrange the functions to look like the following simulation diagram.
14. Save this VI by selecting File»Save. Save this VI to a convenient location as "Spring-Mass Damper Example.vi".

The Block Diagram should now look like this:



Wiring the Simulation Functions Together

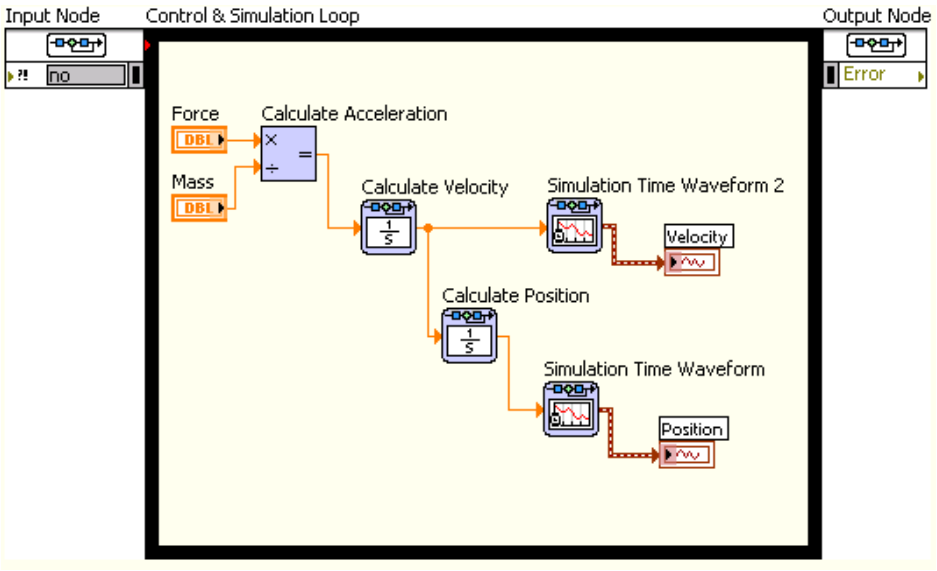
The next step is wiring the functions together to represent the flow of data from one function to another.

Note! Wires on the simulation diagram include arrows that show the direction of the dataflow, whereas wires on a LabVIEW block diagram do not show these arrows.

Complete the following steps to wire these functions together.

1. Right-click the Operand1 input of the Calculate Acceleration function and select Create»Control from the shortcut menu to add a numeric control to the front panel window.
2. Label this control Force.
3. Double-click this control on the simulation diagram. LabVIEW displays the front panel and highlights the Force control.
4. Display the block diagram and create a control for the Operand2 input of the Calculate Acceleration function. Label this new control Mass.
5. Wire the Result output of the Calculate Acceleration function to the input input of the Calculate Velocity function.
6. Wire the output output of the Calculate Velocity function to the input input of the Calculate Position function.
7. Right-click the wire you just created and select Create Wire Branch from the shortcut menu. Wire this branch to the Value input of the SimTime Waveform function that has the Velocity waveform chart.
8. Wire the output output of the Calculate Position function to the Value input of the SimTime Waveform function that has the Position waveform chart.

The Block Diagram should now look like this:

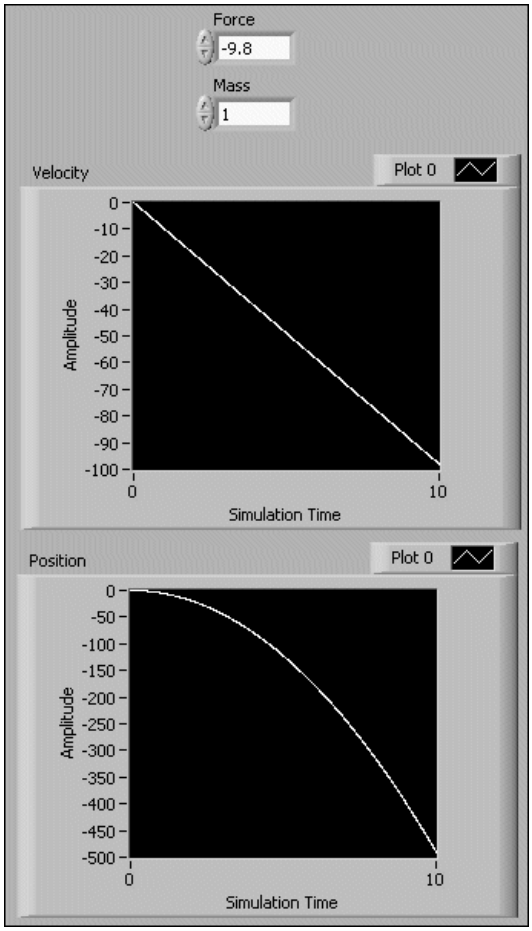


Running the Simulation

You now can run this simulation to test that the data is flowing properly through the Simulation functions. Complete the following steps to run this simulation.

1. Select Window»Show Front Panel, or press <Ctrl-E>, to view the front panel of this simulation. The front panel displays the following objects: a control labeled Force, a control labeled Mass, a waveform chart labeled Velocity, and a waveform chart labeled Position.
2. If necessary, rearrange these controls and indicators so that all objects are visible.
3. Enter -9.8 in the Force numeric control. This value represents the force of gravity, 9.8 meters per second squared, acting on the dynamic system.
4. Enter 1 in the Mass numeric control. This value represents a mass of one kilogram.
5. Click the Run button, or press the <Ctrl-R> keys, to run the VI.

The Front Panel should look like this:



In the Figure above notice that the force of gravity causes the mass position and velocity to constantly decrease. However, in the real world, a mass attached to a spring oscillates up and down. This simulated spring does not oscillate because the simulation diagram does not represent damping or stiffness. You must represent these factors to have a complete simulation of the dynamic system.

Representing Damping and Stiffness

Representing damping and stiffness involves feeding back the velocity and position, each multiplied by a different constant, to the input of the Calculate Acceleration function. Recall the following differential equation this VI simulates.

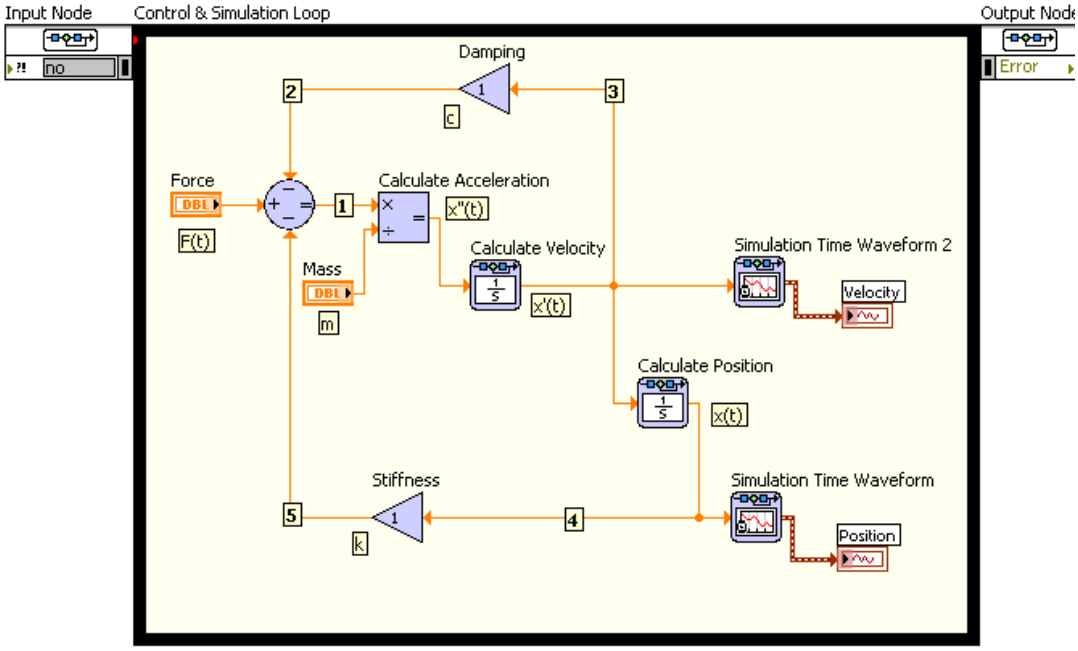
$$F(t) - c\dot{x}(t) - kx(t) = m\ddot{x}(t)$$

In the previous equation, notice you multiply the damping constant c by the velocity of the mass $\dot{x}$. You multiply the stiffness constant k by the mass position x(t). You then subtract these quantities from the external force applied to the mass.

Complete the following steps to represent damping and stiffness in this dynamic system model.

1. View the simulation diagram.
2. Select the Signal Arithmetic palette and place a **Summation** function on the simulation diagram. Move this function to the left of the Force and Mass controls.
3. Double-click the Summation function to configure its operation. By default, the Summation function displays the following three input terminals: a Ø symbol, a + symbol, and a – symbol. This configuration subtracts one input signal from another.
4. Click the Ø symbol twice to change this terminal to the – symbol. This Summation function now subtracts the top and bottom input signals from the left input signal.
5. Click the OK button to save changes and return to the simulation diagram.
6. Select the Signal Arithmetic palette and place a Gain function on the simulation diagram. Move this function above the existing simulation diagram code but still within the Control & Simulation Loop.
7. The input of the Gain function is on the left side of the function, and the output is on the right side. You can reverse the direction of these terminals to indicate feedback better. Right-click the Gain function and select Reverse Terminals from the shortcut menu. The Gain function now points toward the left side of the simulation diagram.
8. Label this Gain function Damping.
9. Press the <Ctrl> key and drag the Gain function to create a separate copy. Move this copy below the existing simulation diagram code but still within the Control & Simulation Loop. Label this function Stiffness.
10. Right-click the wire connecting the Force control to the Calculate Acceleration function and select Delete Wire Branch from the shortcut menu. Move the Force control to the left of the Summation function, and wire this control to the Operand2 input of the Summation function.
11. Create wires 1–5 as indicated in the Figure below. The simulation diagram now fully represents the equation that defines the behavior of the dynamic system.
12. Press <Ctrl-S> to save the VI.

The Block Diagram should now look like this:

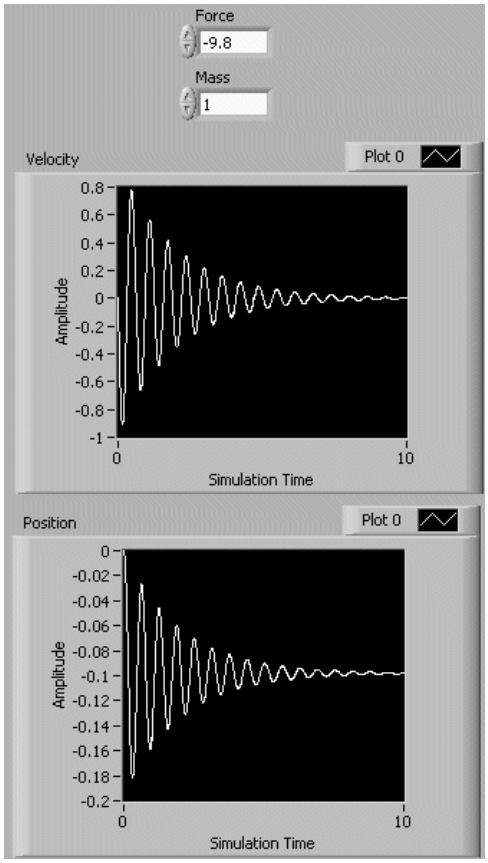


Configuring the Stiffness of the Spring

Before you run the simulation again, you must configure the stiffness of the simulated spring. Complete the following steps to configure this Simulation function.

1. Double-click the Stiffness function to display the Gain Configuration dialog box.
2. Enter 100 in the gain numeric control. This value represents a stiffness of 100 Newtons per meter.
3. Click OK to return to the simulation diagram. Notice that the Stiffness function displays 100.
4. Display the front panel and ensure the Force control is set to -9.8 and the Mass control is set to 1.
5. Run the simulation. The Velocity and Position charts display the behavior of the mass as the spring oscillates. Notice the new behavior compared to the last time you ran the simulation. This time, the velocity and position do not constantly decrease. Both values oscillate, which is how a spring behaves in the real world.
6. Change the value of the Mass control to 10 and run the simulation again. Notice the different behavior in the Velocity and Position charts. The 10 kg mass forces the spring to oscillate less frequently and within a smaller velocity/position range.

The Front Panel should look like this:



Configuring Simulation Functions Programmatically

The previous section provided information about configuring Simulation functions using the configuration dialog box. Instead of using the configuration dialog box, you can improve the interactivity of a simulation by creating front panel controls that configure a Simulation function programmatically. Complete the following steps to configure the Stiffness function programmatically.

1. If you are not already viewing the Context Help window, press <Ctrl-H> to display this window.
2. Display the block diagram and move the cursor over the Stiffness function. Notice this function has only one input terminal.
3. Display the Gain Configuration dialog box of the Stiffness function.
4. Select Terminal from the Parameter source pull-down menu. This action disables the gain numeric control.
5. Click the OK button to save changes and return to the block diagram.
6. Move the cursor over the Stiffness function. Notice the Context Help window displays the Gain function with the new gain input terminal.
7. Create a control for this input, and label the control gain (k).

8. View the front panel. Notice the new control gain (k). Enter a value of 100 for this control and run the simulation. Notice the behavior is exactly the same as when you used the configuration dialog box to configure the Stiffness function.

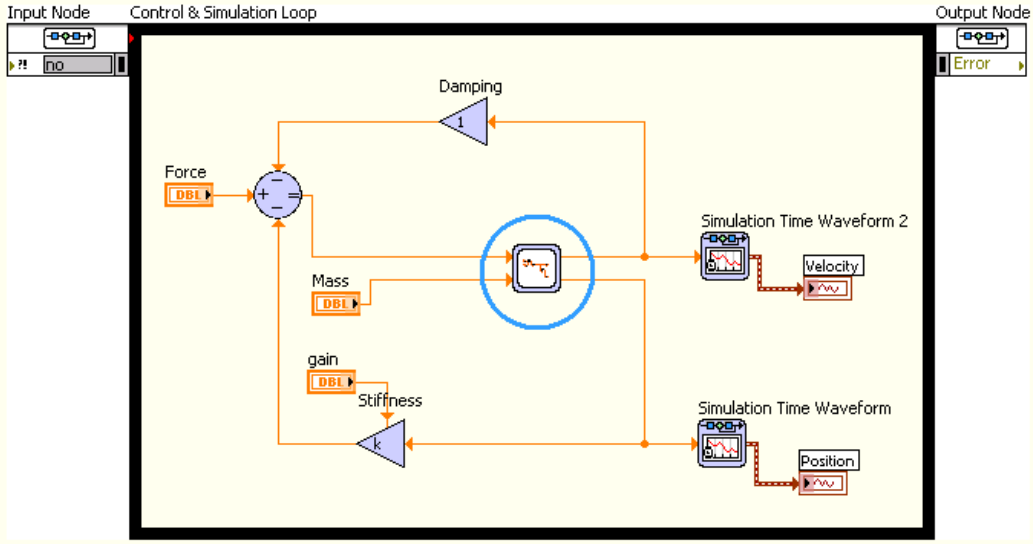
Modularizing Simulation Diagram Code

You can create simulation subsystems to divide simulation diagrams into components that are modular, reusable, and independently verifiable. Complete the following steps to create a simulation subsystem from this simulation diagram.

1. View the simulation diagram.
2. Select the Calculate Acceleration, Calculate Velocity, and Calculate Position functions by pressing the <Shift> key and clicking each function.
3. Select Edit»Create Simulation Subsystem. LabVIEW replaces these three functions with a single function that represents the simulation subsystem, which is circled in the Figure below. The inputs and outputs of the simulation subsystem include the inputs and outputs of all the functions you selected. Also, notice the amount of blank space on the simulation diagram. Because you combined three functions into a subsystem, you can resize the Control & Simulation Loop and reposition the functions to make the simulation diagram easier to view.
4. Press <Ctrl-S> to save the simulation diagram. LabVIEW prompts you to save the simulation subsystem you just created. Click the Yes button and save this simulation subsystem as “Newton.vi”. You now have a simulation subsystem that obtains the position of a mass by using Newton's Second Law of Motion.

Note! You can resize the simulation subsystem to better display its simulation diagram. You also can double-click the simulation subsystem to display the configuration dialog box of that simulation subsystem.

The simulation subsystem should look like this:



Editing the Simulation Subsystem

Edit the simulation subsystem “Newton.vi” by right-clicking this subsystem and selecting Open Subsystem from the shortcut menu. View the simulation diagram.

Notice this simulation subsystem does not contain a Control & Simulation Loop, but the entire background is pale yellow to indicate a simulation diagram. If you place this simulation subsystem in a Control & Simulation Loop, the simulation subsystem inherits all simulation parameters from the Control & Simulation Loop.

If you run this subsystem as a stand-alone VI, you can configure the simulation parameters by selecting Operate»Configure Simulation Parameters. Any parameters you configure using this method do not take effect when the subsystem is within another Control & Simulation Loop. If you place this simulation subsystem on a block diagram outside a Control & Simulation Loop, you can configure the simulation parameters by double-clicking the simulation subsystem to display the configuration dialog box of that simulation subsystem.

Configuring Simulation Parameters Programmatically

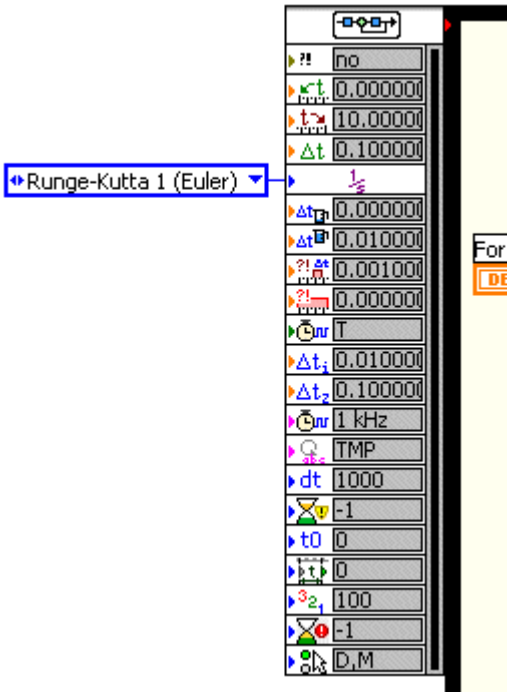
Earlier in this exercise, you used the Configure Simulation Parameters dialog box to configure the parameters of “Spring-Mass Damper Example.vi”. You also can configure simulation parameters programmatically by using the Input Node of the Control & Simulation Loop. Complete the following steps to configure simulation parameters programmatically.

1. View the simulation diagram of “Spring-Mass Damper Example.vi”.
2. Move the cursor over the Input Node to display resizing handles.
3. Drag the bottom handle down to display all available Node inputs. You use these inputs to configure the simulation parameters without displaying the Configure Simulation Parameters dialog box. You also can right-click the Input Node and select Show All Inputs from the shortcut menu.

   Notice the gray boxes next to each input. These boxes display values you configure in the Configure Simulation Parameters dialog box. For example, the third gray box from the top displays 10.0000, which is the value of the Final Time numeric control that you configured. The fifth gray box from the top displays RK 23. This box specifies the current ODE solver, which you configured as Runge-Kutta 23 (variable). Move the cursor over the left edge of each Node input to display the label of that input.
4. Right-click the input terminal of the ODE Solver input and select Create»Constant from the shortcut menu. A block diagram constant appears outside the Control & Simulation Loop. The value of this constant is Runge-Kutta 1 (Euler), which is different than what you configured in the Configure Simulation Parameters dialog box. However, the gray box disappears from the Input Node, indicating that the value

of this parameter does not come from the Configure Simulation Parameters dialog box. Values that you programmatically configure override any settings you made in the Configure Simulation Parameters dialog box.

The Input Node should now look like the following figure:



Summary

This exercise introduced you to the following concepts:

The simulation diagram reflects the dynamic system model you want to simulate. This dynamic system model is a differential or difference equation that represents a dynamic system.

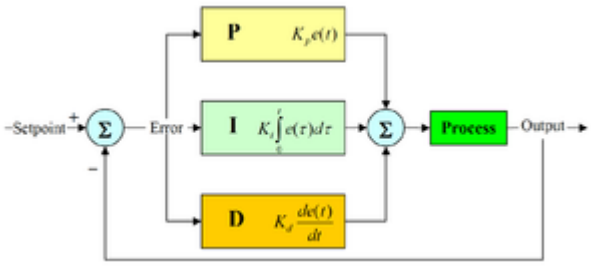
The Control & Simulation Loop contains the parameters that define the behavior of the simulation. The Control & Simulation Loop also defines the visual boundary of the simulation diagram. Double-click the Input Node of the Control & Simulation Loop to access configurable parameters. You also can expand the Input Node to access these parameters.

The Simulation palette contains the VIs and functions you use to build a simulation. You can double-click most Simulation functions to display a dialog box that configures that function. You also can create input terminals for function inputs.

You can create simulation subsystems to modularize, encapsulate, validate, and re-use portions of the simulation diagram.

5 PID Control

Currently, the Proportional-Integral-Derivative (PID) algorithm is the most common control algorithm used in industry. Often, people use PID to control processes that include heating and cooling systems, fluid level monitoring, flow control, and pressure control. In PID control, you must specify a process variable and a setpoint. The process variable is the system parameter you want to control, such as temperature, pressure, or flow rate, and the setpoint is the desired value for the parameter you are controlling. A PID controller determines a controller output value, such as the heater power or valve position. The controller applies the controller output value to the system, which in turn drives the process variable toward the setpoint value.



[Wikipedia]

The PID controller compares the setpoint (SP) to the process variable (PV) to obtain the error (e).

$$e = SP - PV$$

Then the PID controller calculates the controller action, u(t), where Kc is controller gain.

$$u(t) = K_c\left(e + \frac{1}{T_i}\int_0^t e\,dt + T_d\frac{de}{dt}\right)$$

Ti is the integral time in minutes, also called the reset time, and Td is the derivative time in minutes, also called the rate time.

The following formula represents the proportional action.

$$u_p(t) = K_c e$$

The following formula represents the integral action.

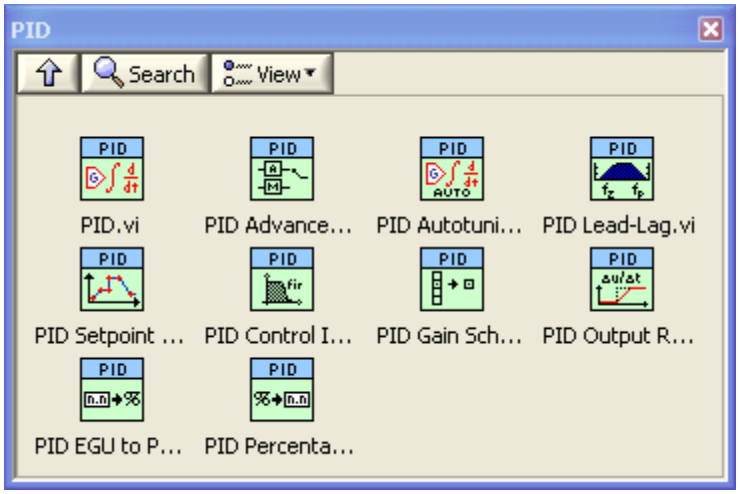
$$u_I(t) = \frac{K_c}{T_i}\int_0^t e dt$$

The following formula represents the derivative action.

$$u_D(t) = K_c T_d \frac{de}{dt}$$

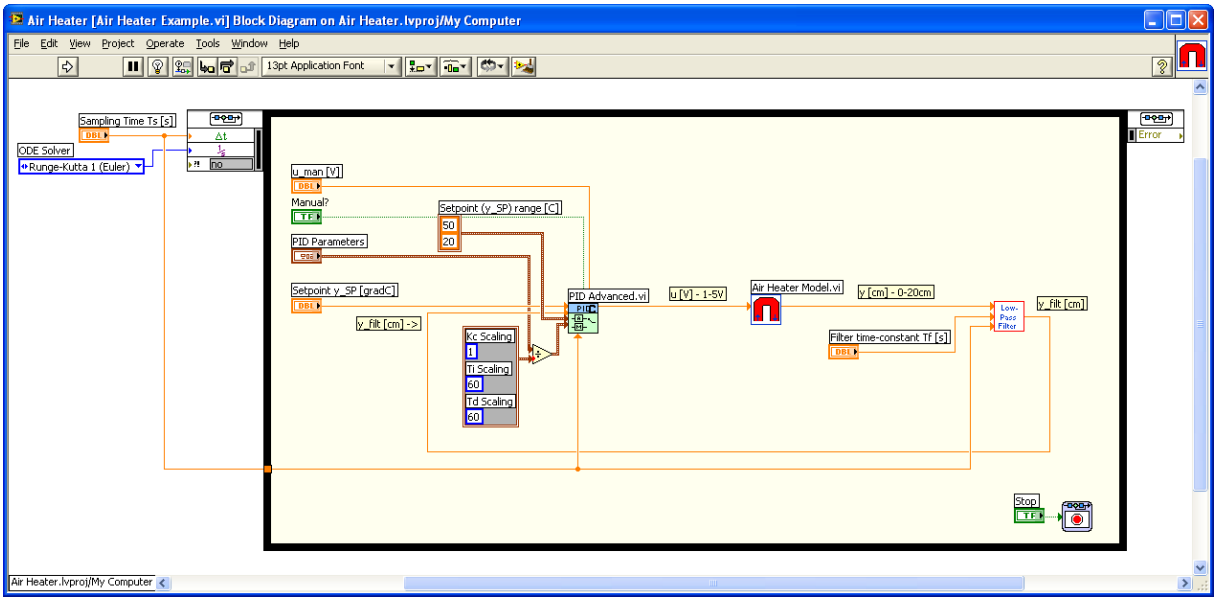
5.1 PID Control in LabVIEW

In the "**PID**" Sub palette we have the functions/SubVIs for PID Control. I recommend that you use the "**PID Advanced.vi**".



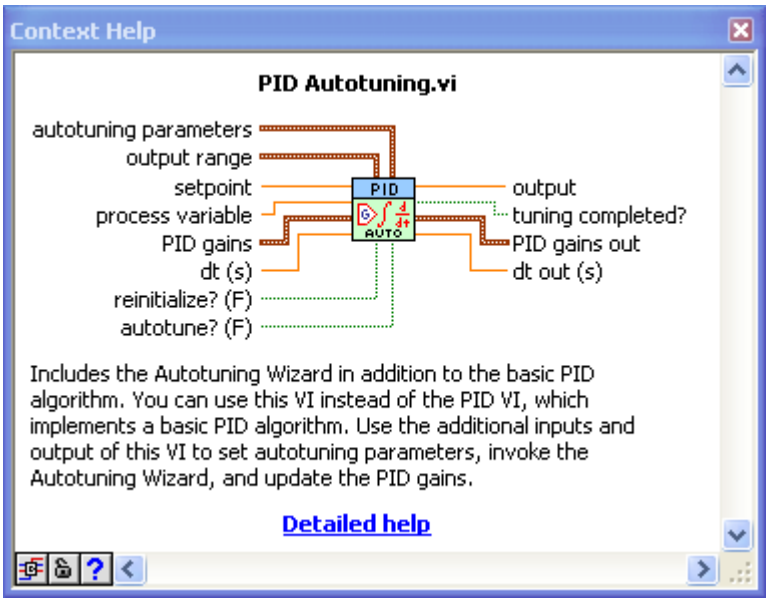
Example: PID Control

Below we see how we can use the PID Advanvanced.vi in order to control a simulated Model.



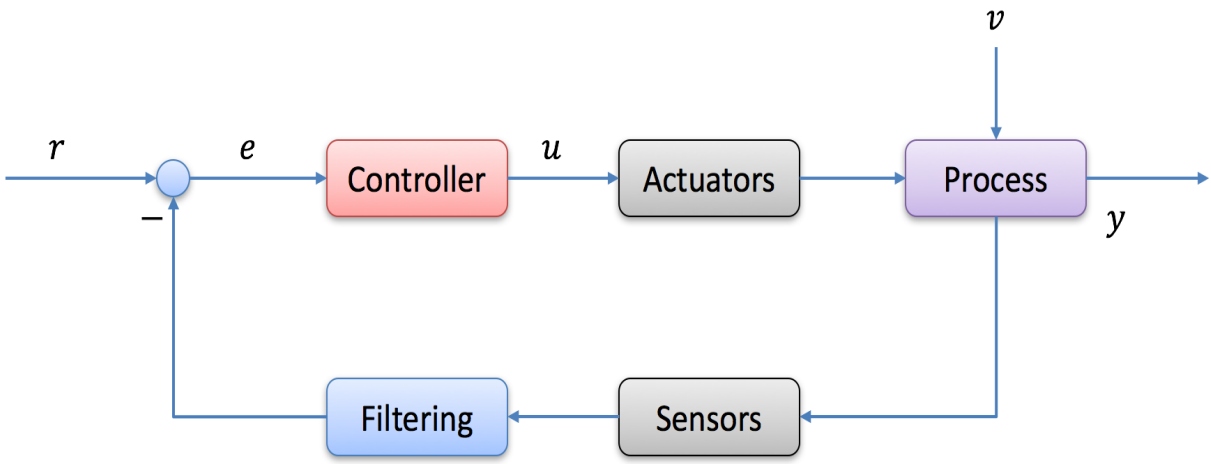
5.2 Auto-tuning

The LabVIEW PID and Fuzzy Logic Toolkit include a VI for auto-tuning.



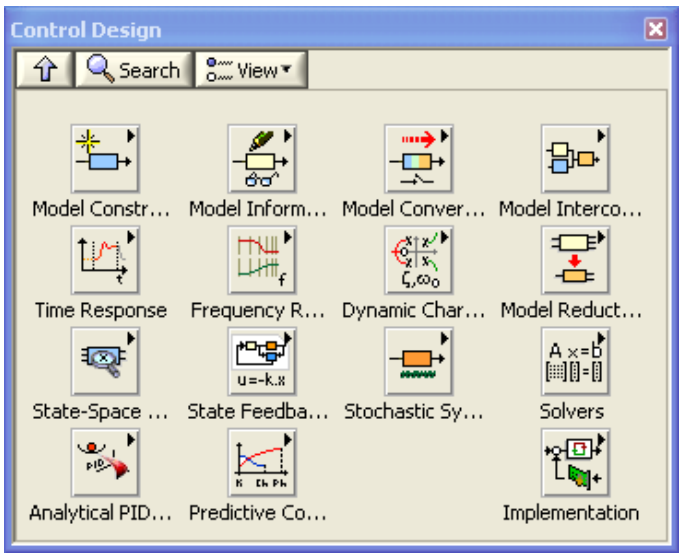
6 Control Design

Control design is a process that involves developing mathematical models that describe a physical system, analyzing the models to learn about their dynamic characteristics, and creating a controller to achieve certain dynamic characteristics.



6.1 Control Design in LabVIEW

Control Design palette**:**

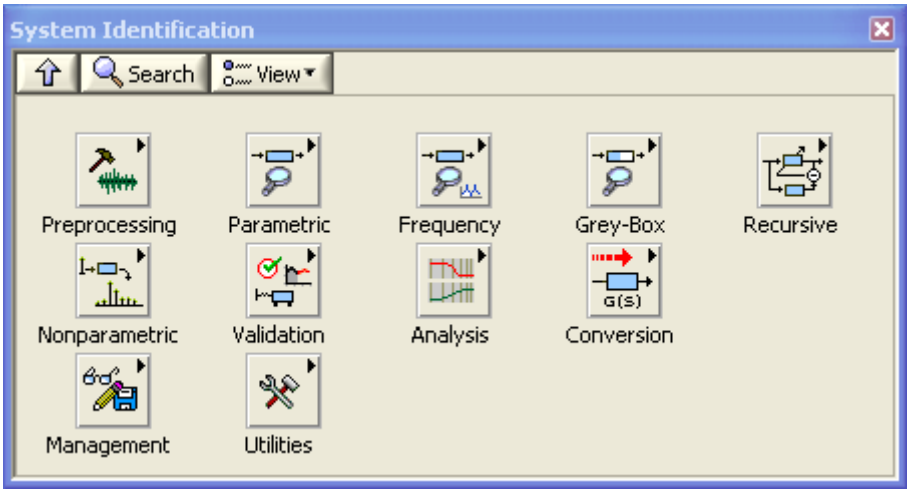


7 System Identification

7.1 System Identification in LabVIEW

The “System Identification Toolkit” combines data acquisition tools with system identification algorithms for accurate plant modeling. You can take advantage of LabVIEW intuitive data acquisition tools such as the DAQ Assistant to stimulate and acquire data from the plant and then automatically identify a dynamic system model. You can convert system identification models to state-space, transfer function, or pole-zero-gain form for control system analysis and design. The toolkit includes built-in functions for common tasks such as data preprocessing, model creation, and system analysis. Using other built-in utilities, you can plot the model with intuitive graphical representation as well as store the model.

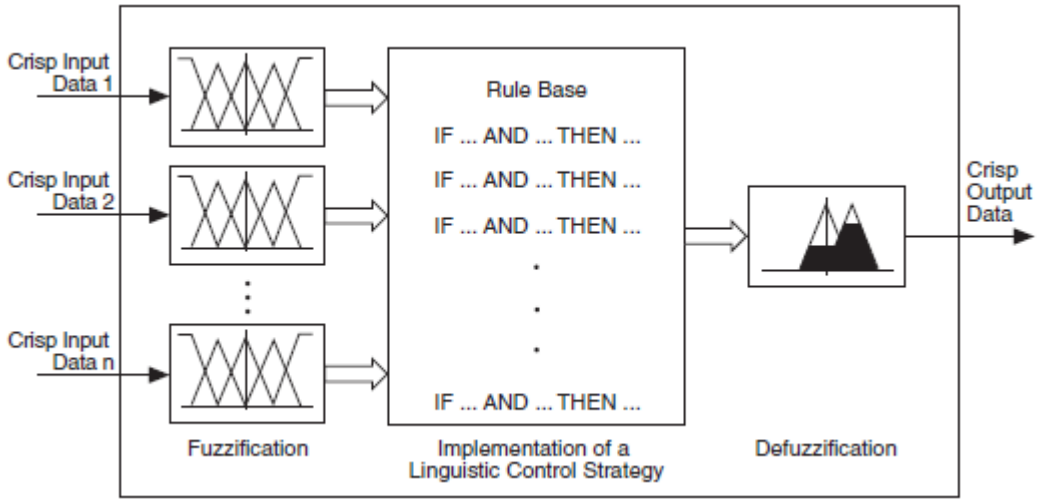
System Identification palette:



8 Fuzzy Logic

Fuzzy logic is a method of rule-based decision making used for expert systems and process control. Fuzzy logic differs from traditional Boolean logic in that fuzzy logic allows for partial membership in a set. You can use fuzzy logic to control processes represented by subjective, linguistic descriptions.

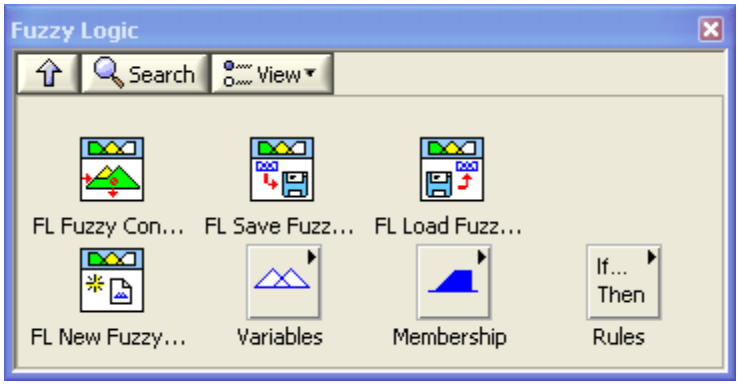
A fuzzy system is a system of variables that are associated using fuzzy logic. A fuzzy controller uses defined rules to control a fuzzy system based on the current values of input variables.



[Wikipedia]

8.1 Fuzzy Logic in LabVIEW

The Fuzzy Logic palette in LabVIEW:


Fuzzy Logic
Search
View
FL Fuzzy Con...
FL Save Fuzz...
FL Load Fuzz...
FL New Fuzzy...
Variables
Membership
If...
Then
Rules

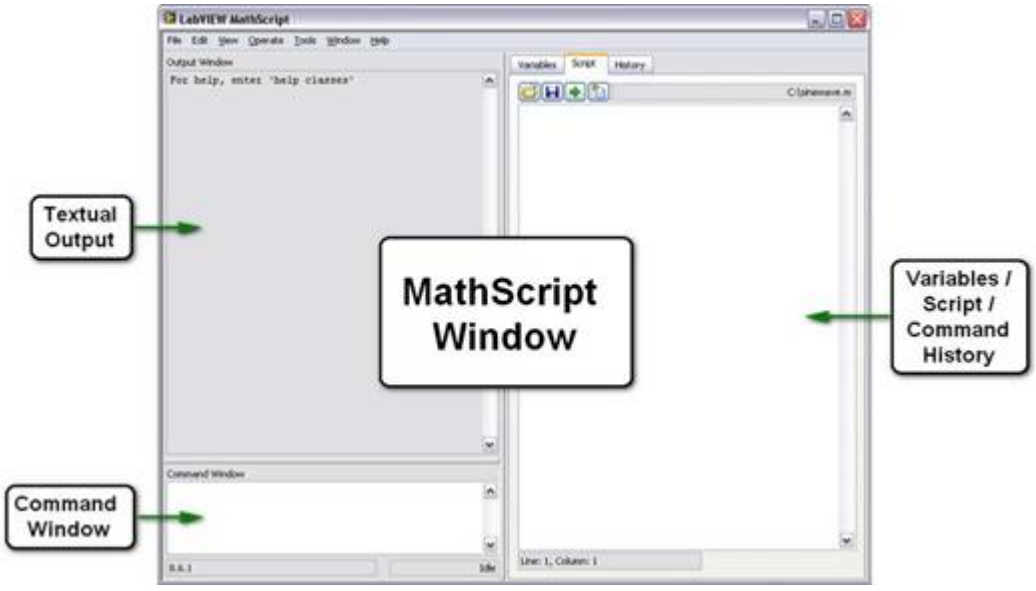
9 LabVIEW MathScript

Requires: **MathScript RT Module**

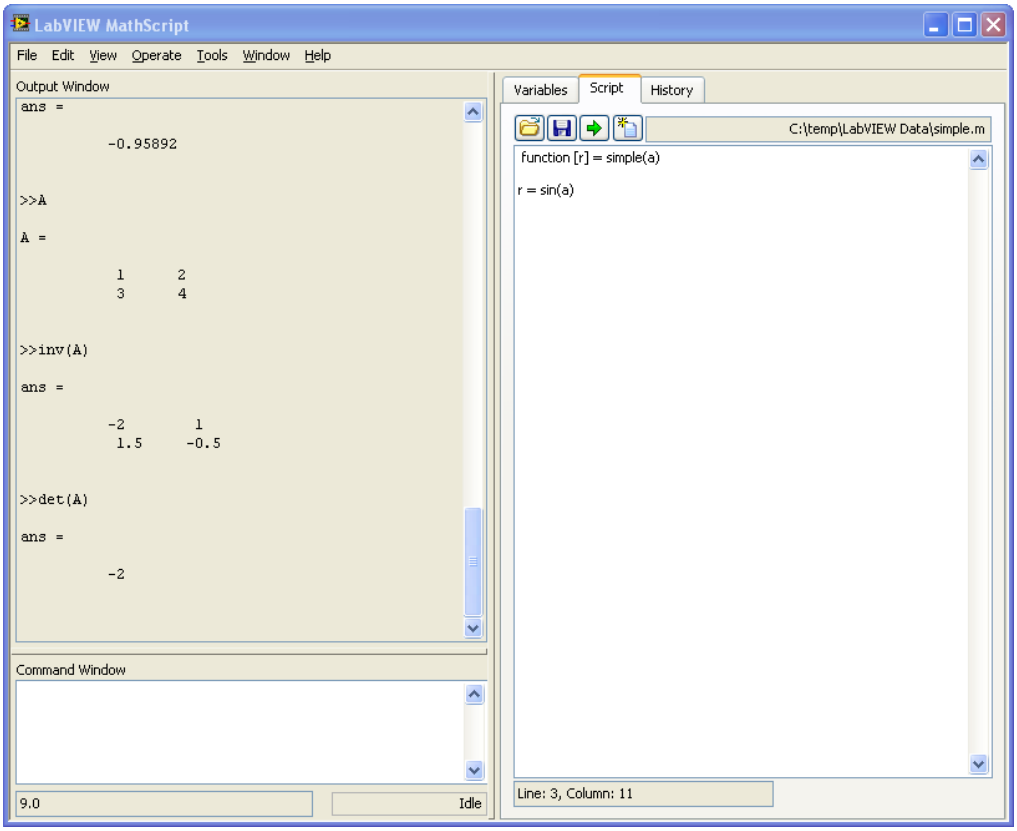
The “LabVIEW MathScript Window” is an interactive interface in which you can enter .m file script commands and see immediate results, variables and commands history. The window includes a command-line interface where you can enter commands one-by-one for quick calculations, script debugging or learning. Alternatively, you can enter and execute groups of commands through a script editor window.

As you work, a variable display updates to show the graphical / textual results and a history window tracks your commands. The history view facilitates algorithm development by allowing you to use the clipboard to reuse your previously executed commands.

You can use the “LabVIEW MathScript Window” to enter commands one at time. You also can enter batch scripts in a simple text editor window, loaded from a text file, or imported from a separate text editor. The “LabVIEW MathScript Window” provides immediate feedback in a variety of forms, such as graphs and text.



Example:



9.1 Help

You may also type help in your command window

```
>>help
```

Or more specific, e.g.,

```
>>help plot
```

9.2 Examples

I advise you to test all the examples in this text in LabVIEW MathScript in order to get familiar with the program and its syntax. All examples in the text are outlined in a frame like this:

```
>>
...
```

This is commands you should write in the **Command Window**.

You type all your commands in the Command Window. I will use the symbol "**>>**" to illustrate that the commands should be written in the Command Window.

Example: Matrices

Defining the following matrix

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}$$

The syntax is as follows:

```
>> A = [1 2;0 3]
```

Or

```
>> A = [1,2;0,3]
```

If you, for an example, want to find the answer to

$$a + b, where\ a = 4, b = 3$$

```
>>a=4
>>b=3
>>a+b
```

MathScript then responds:

```
ans =
      7
```

MathScript provides a simple way to define simple arrays using the syntax: "**init:increment:terminator**". For instance:

```
>> array = 1:2:9
array =
 1 3 5 7 9
```

defines a variable named array (or assigns a new value to an existing variable with the name array) which is an array consisting of the values 1, 3, 5, 7, and 9. That is, the array starts at 1 (the init value), increments with each step from the previous value by 2 (the increment value), and stops once it reaches (or to avoid exceeding) 9 (the terminator value).

The increment value can actually be left out of this syntax (along with one of the colons), to use a default value of 1.

```
>> ari = 1:5
ari =
 1 2 3 4 5
```

assigns to the variable named ari an array with the values 1, 2, 3, 4, and 5, since the default value of 1 is used as the incrementer.

Note that the indexing is one-based, which is the usual convention for matrices in mathematics. This is atypical for programming languages, whose arrays more often start with zero.

Matrices can be defined by separating the elements of a row with blank space or comma and using a semicolon to terminate each row. The list of elements should be surrounded by square brackets: **[]**. Parentheses: **()** are used to access elements and subarrays (they are also used to denote a function argument list).

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
 16  3  2 13
  5 10 11  8
  9  6  7 12
  4 15 14  1
>> A(2,3)
ans =
 11
```

Sets of indices can be specified by expressions such as "2:4", which evaluates to [2, 3, 4]. For example, a submatrix taken from rows 2 through 4 and columns 3 through 4 can be written as:

```
>> A(2:4,3:4)
ans =
 11 8
 7 12
 14 1
```

A square identity matrix of size n can be generated using the function eye, and matrices of any size with zeros or ones can be generated with the functions zeros and ones, respectively.

```
>> eye(3)
ans =
 1 0 0
 0 1 0
 0 0 1
>> zeros(2,3)
ans =
 0 0 0
 0 0 0
>> ones(2,3)
ans =
 1 1 1
 1 1 1
```

9.3 Useful commands

Here are some useful commands:

Command	Description
`eye(x), eye(x,y)`	Identity matrix of order x
`ones(x), ones(x,y)`	A matrix with only ones
`zeros(x), zeros(x,y)`	A matrix with only zeros
`diag([x y z])`	Diagonal matrix
`size(A)`	Dimension of matrix A
`A'`	Inverse of matrix A

9.4 Plotting

This chapter explains the basic concepts of creating plots in MathScript.

Topics:

- Basic Plot commands

Example: Plotting

Function plot can be used to produce a graph from two vectors x and y. The code:

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```

10 Discretization

Often, we need to develop discrete algorithms of our process. In addition, we might need to create our own discrete PI(D) controller. A discrete low-pass filter is also good to have.

There exists lots of different discretization methods like the "Zero Order Hold" (ZOH) method, Tustin's method and Euler's methods (Forward and Backward). We will focus on Eulers methods in this document, because they are very easy to use.

Euler Forward discretization method:

$$\dot{x} \approx \frac{x_{k+1} - x_k}{T_s}$$

Euler Backward discretization method:

$$\dot{x} \approx \frac{x_k - x_{k-1}}{T_s}$$

T_s is the Sampling Time.

10.1 Low-pass Filter

The transfer function for a first-order low-pass filter may be written:

$$H(s) = \frac{y_f(s)}{y(s)} = \frac{1}{T_f s + 1}$$

Where T_f is the time-constant of the filter, $y(s)$ is the filter input and $y_f(s)$ is the filter output.

Discrete version:

It can be shown that a discrete version can be stated as:

$$y_{f,k} = (1 - a)y_{f,k-1} + ay_k$$

Where

$$a = \frac{T_s}{T_f + T_s}$$

Where T_s is the Sampling Time.

It is a golden rule that $T_s \ll T_f$ and in practice we should use the following rule:

$$T_s \leq \frac{T_f}{5}$$

Example:

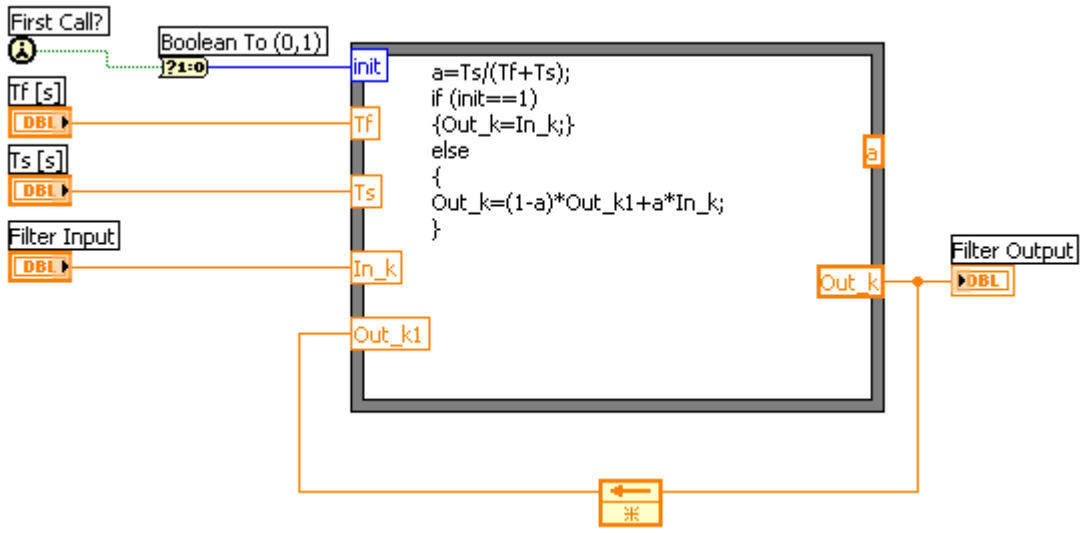
We will implement the discrete low-pass filter algorithm below using a Formula Node in LabVIEW:

$$y_{f,k} = (1 - a)y_{f,k-1} + ay_k$$

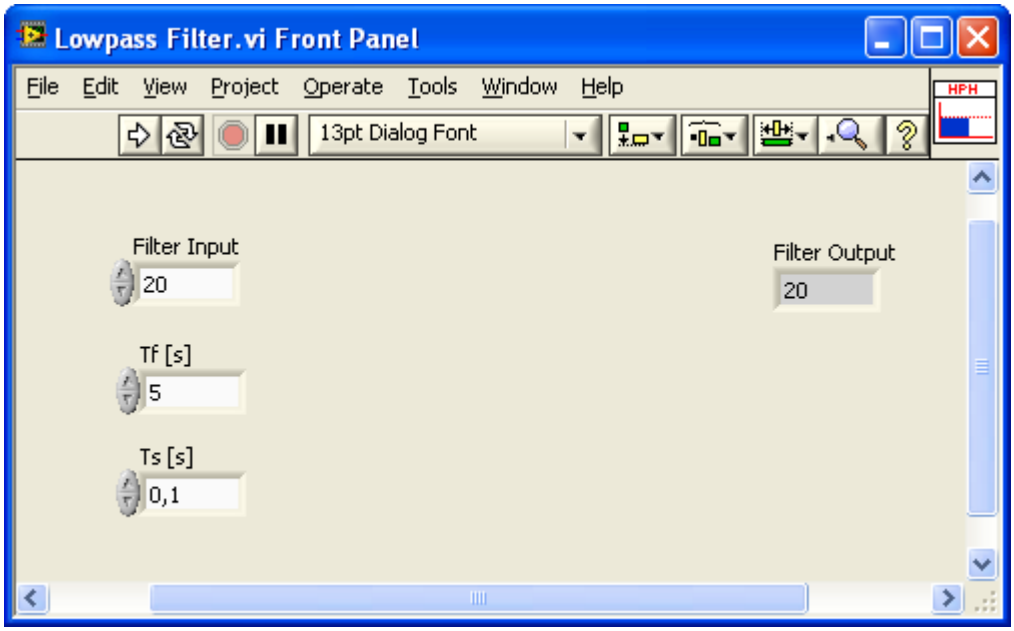
Where

$$a = \frac{T_s}{T_f + T_s}$$

The Block Diagram becomes:



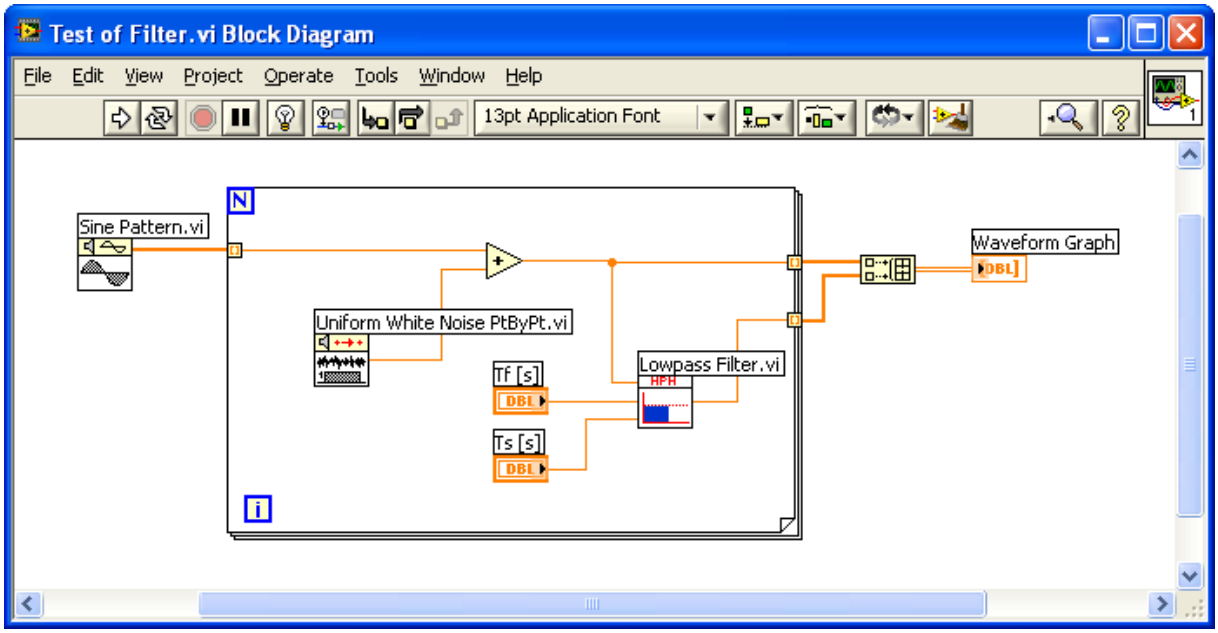
The Front Panel:



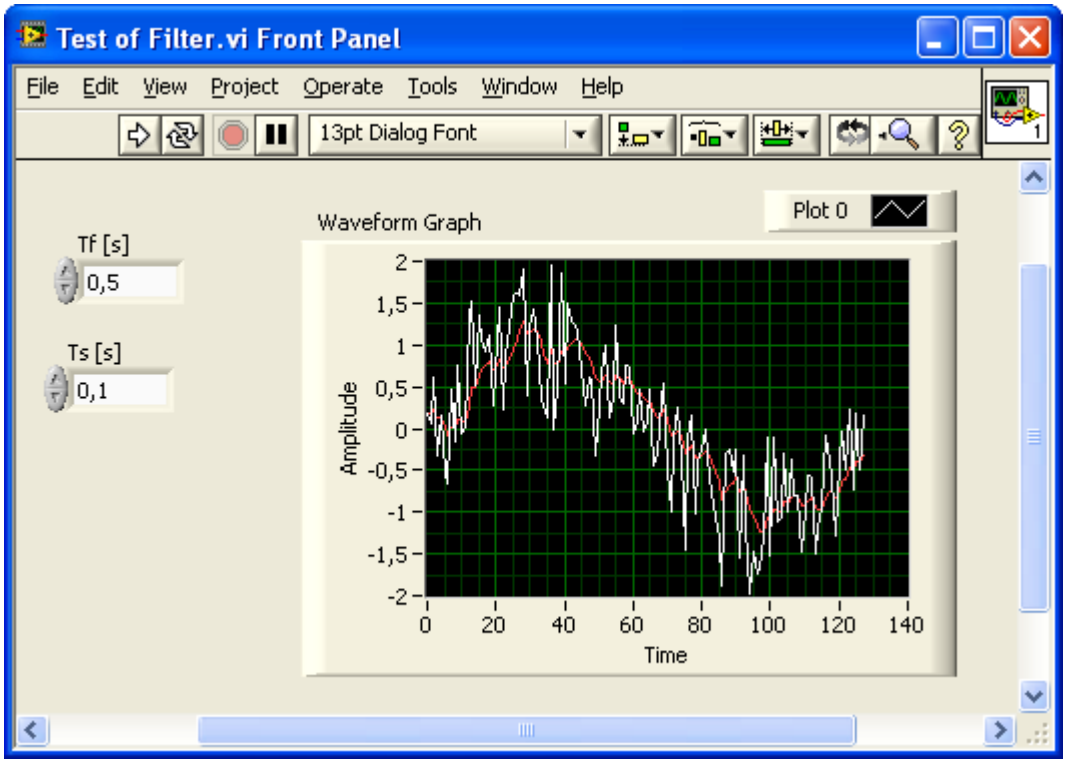
It is a good idea to build this as a SubVIs, and then we can easily reuse the Low-pass filter in all our applications.

We will test the discrete low-pass filter, to make sure it works as expected:

We create a simple test application where we add some random white noise to a sine signal. We will plot the unfiltered and the filtered signal to see if the low-pass filter is able to remove the noise from the sine signal.



We get the following results:



We see that the filter works fine. The red line is the unfiltered sine signal with white noise, while the red line is the filtered results.

[End of Example]

10.2 PI Controller

A PI controller may be written:

$$\boldsymbol{u(t) = u_0 + K_p e(t) + \frac{K_p}{T_i}\int_0^t e d\tau}$$

Where u is the controller output and e is the control error:

$$e(t) = r(t) - y(t)$$

Laplace version:

$$u(s) = K_p e(s) + \frac{K_p}{T_i s} e(s)$$

Discrete version:

We start with:

$$u(t) = u_0 + K_p e(t) + \frac{K_p}{T_i}\int_0^t e d\tau$$

In order to make a discrete version using, e.g., Euler, we can derive both sides of the equation:

$$\dot{u} = \dot{u}_0 + K_p \dot{e} + \frac{K_p}{T_i} e$$

If we use Euler Forward we get:

$$\frac{u_k - u_{k-1}}{T_s} = \frac{u_{0,k} - u_{0,k-1}}{T_s} + K_p \frac{e_k - e_{k-1}}{T_s} + \frac{K_p}{T_i} e_k$$

Then we get:

$$\boldsymbol{u_k = u_{k-1} + u_{0,k} - u_{0,k-1} + K_p(e_k - e_{k-1}) + \frac{K_p}{T_i} T_s e_k}$$

Where

$$e_k = r_k - y_k$$

We can also split the equation above in 2 different pars by setting:

$$\Delta u_k = u_k - u_{k-1}$$

This gives the following PI control algorithm:

$$\boldsymbol{e_k = r_k - y_k}$$

$$\boldsymbol{\Delta u_k = u_{0,k} - u_{0,k-1} + K_p(e_k - e_{k-1}) + \frac{K_p}{T_i} T_s e_k}$$

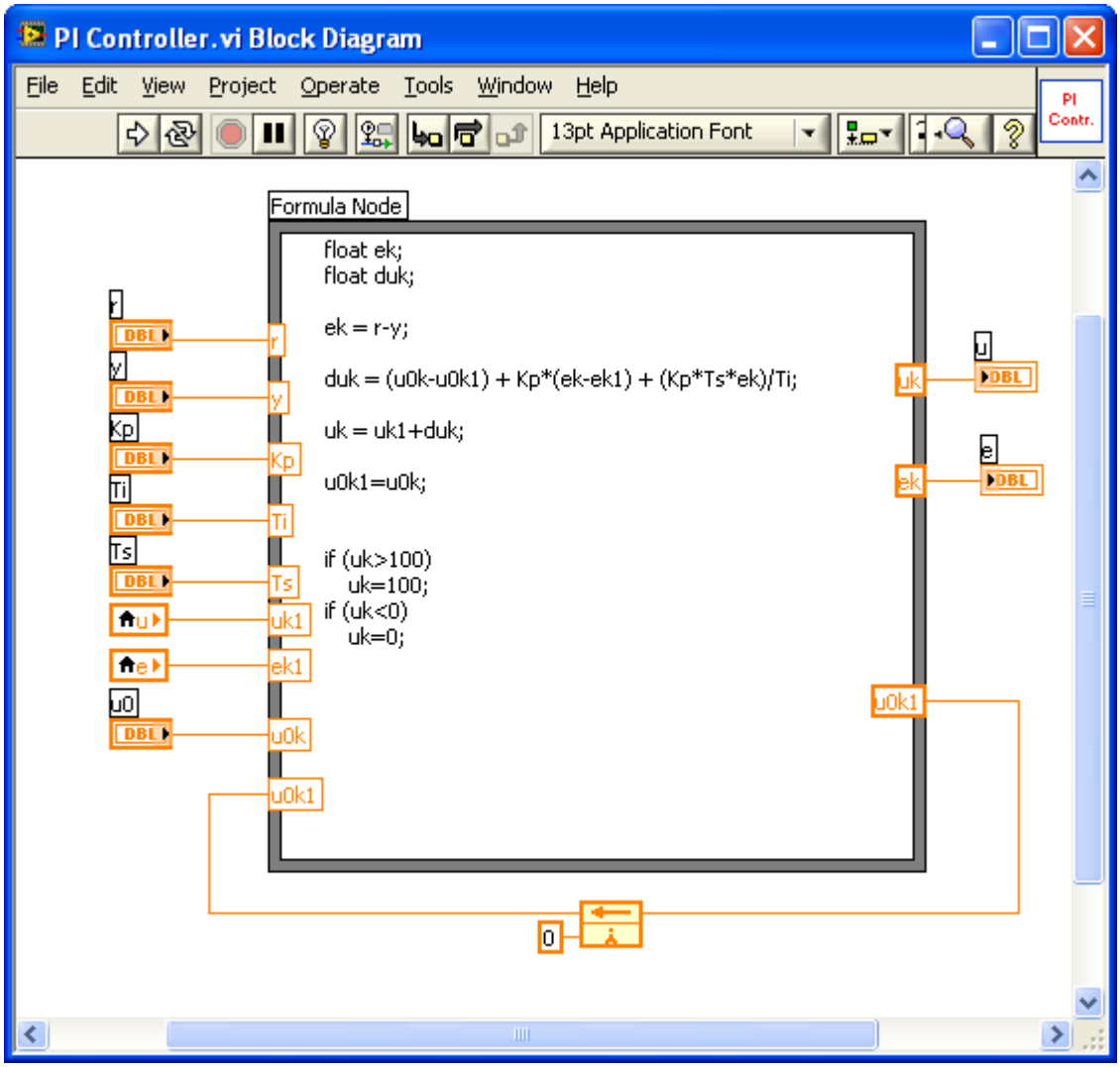
$$\boldsymbol{u_k = u_{k-1} + \Delta u_k}$$

This algorithm can easily be implemented in LabVIEW or other languages such as, e.g., C# or MATLAB.

For more details about how to implement this in C#, see the Tutorial "Data Acquisition in C#", available from https://www.halvorsen.blog.

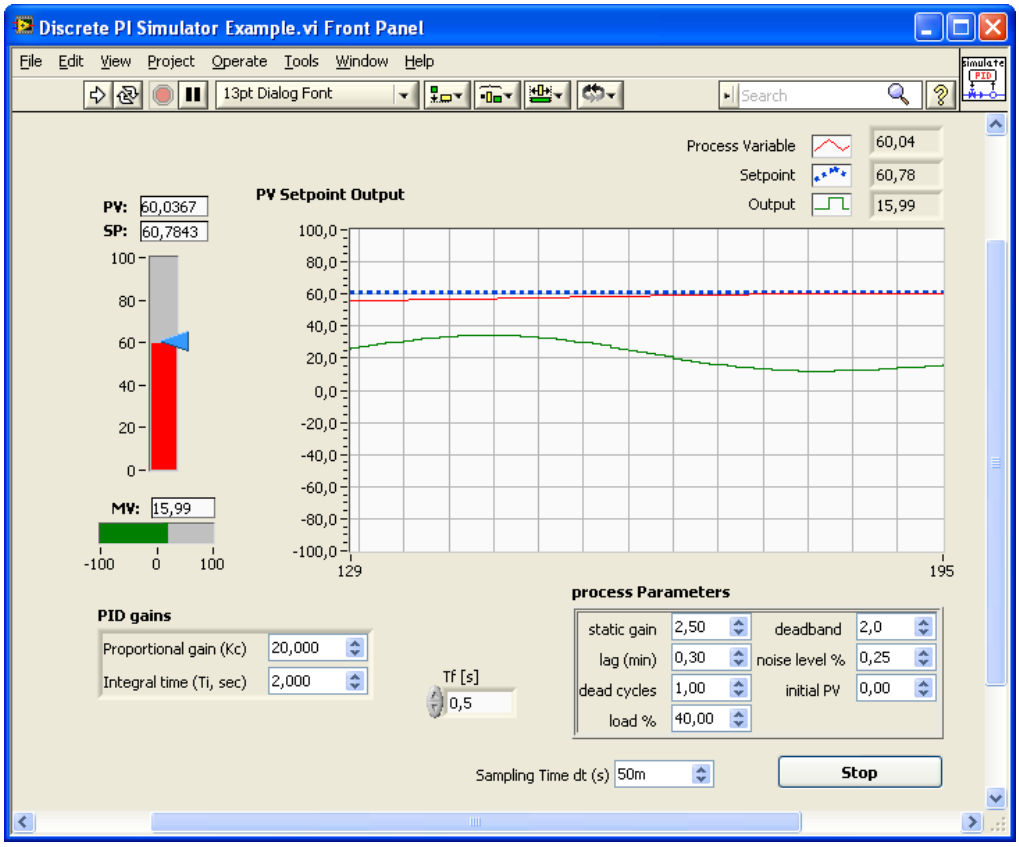
LabVIEW Example:

Below we have implemented the discrete PI controller using a Formula Node in LabVIEW:

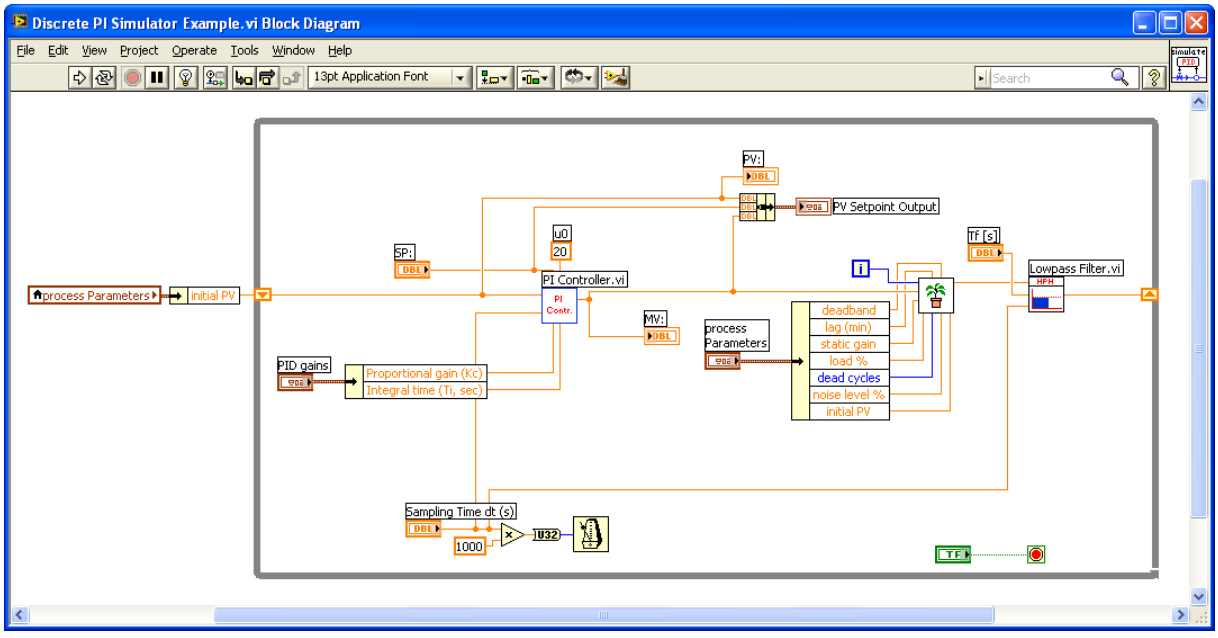


The PI controller is implemented as a SubVI, so it is easy to reuse the algorithm in all our applications.

We test our discrete PI controller with the following application:



Block Diagram:



[End of Example]

10.2.1 PI Controller as a State-space model

We set $z = \frac{1}{s}e \Rightarrow sz = e \Rightarrow \dot{z} = e$

This gives:

$$\dot{z} = e$$

$$u = K_p e + \frac{K_p}{T_i} z$$

Where

$$e = r - y$$

Discrete version:

Using Euler:

$$\dot{z} \approx \frac{z_{k+1} - z_k}{T_s}$$

Where T_s is the Sampling Time.

This gives:

$$\frac{z_{k+1} - z_k}{T_s} = e_k$$

$$u_k = K_p e_k + \frac{K_p}{T_i} z_k$$

Finally:

$$\boldsymbol{e_k = r_k - y_k}$$

$$\boldsymbol{u_k = K_p e_k + \frac{K_p}{T_i} z_k}$$

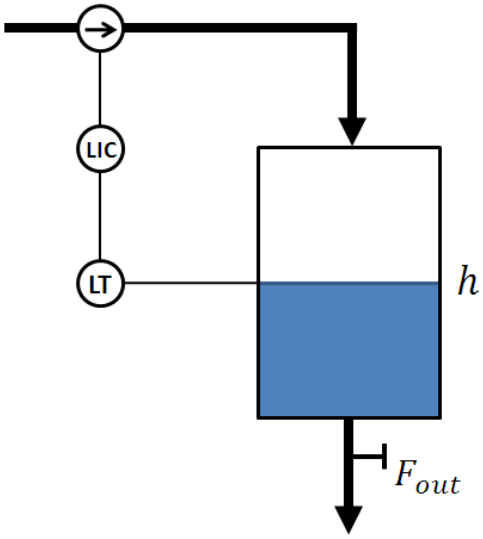
$$\boldsymbol{z_{k+1} = z_k + T_s e_k}$$

This algorithm can easily be implemented in LabVIEW or other languages such as, e.g., C# or MATLAB.

For more details about how to implement this in C#, see the Tutorial "Data Acquisition in C#", available from https://www.halvorsen.blog.

10.3 Process Model

We will use a simple water tank to illustrate how to create a discrete version of a mathematical process model. Below we see an illustration:



A very simple (linear) model of the water tank is as follows:

$$A_t\dot{h} = K_p u - F_{out}$$

or

$$\dot{h} = \frac{1}{A_t}\left[K_p u - F_{out}\right]$$

Where:

- h [cm] is the level in the water tank
- u [V] is the pump control signal to the pump
- A_t [cm2] is the cross-sectional area in the tank
- K_p [(cm3/s)/V] is the pump gain
- F_{out} [cm3/s] is the outflow through the valve (this outflow can be modeled more accurately taking into account the valve characteristic expressing the relation between pressure drop across the valve and the flow through the valve).

We can use the Euler Forward discretization method in order to create a discrete model:

$$\dot{x} \approx \frac{x_{k+1} - x_k}{T_s}$$

Then we get:

$$\frac{h_{k+1} - h_k}{T_s} = \frac{1}{A_t}\left[K_p u_k - F_{out}\right]$$

Finally:

$$\boldsymbol{h_{k+1} = h_k + \frac{T_s}{A_t}\left[K_p u_k - F_{out}\right]}$$

This model can easily be implemented in a computer using, e.g., MATLAB, LabVIEW or C#.

For more details for how to do this in C#, see the Tutorial "Data Acquisition in C#".

In LabVIEW this can, e.g., be implemented in a Formula Node or MathScript Node.

Example:

In this example we will simulate a **Bacteria Population**.

In this example we will use LabVIEW and the LabVIEW Control Design and Simulation Module to simulate a simple model of a bacteria population in a jar.

The **model** is as follows:

$$birth\ rate = bx$$

$$death\ rate = px^2$$

Then the total rate of change of bacteria population is:

$$\dot{x} = bx - px^2$$

We set *b=1*/hour and *p=0.5* bacteria-hour in our example.

We will simulate the number of bacteria in the jar after **1 hour**, assuming that initially there are **100 bacteria** present.

We will simulate the system using a For Loop in LabVIEW and implement the discrete model in a Formula Node.

Step 1: We start by creating the discrete model.

If we use Euler Forward differentiation method:

$$\dot{x} \approx \frac{x_{k+1} - x_k}{T_s}$$

Where T_s is the Sampling Time.

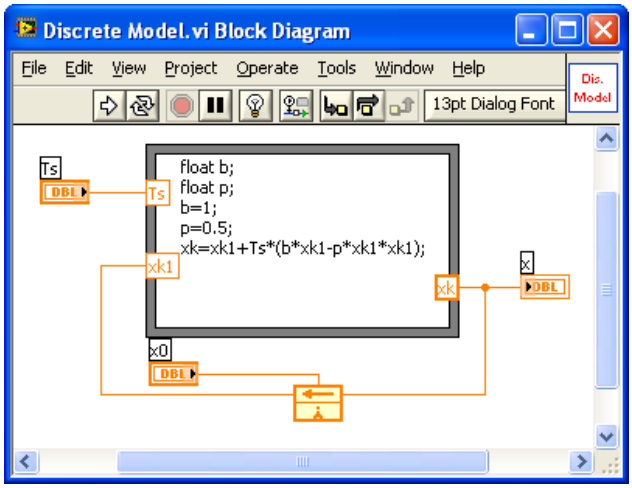
We get:

$$\frac{x_{k+1} - x_k}{T_s} = bx_k - px_k^2$$

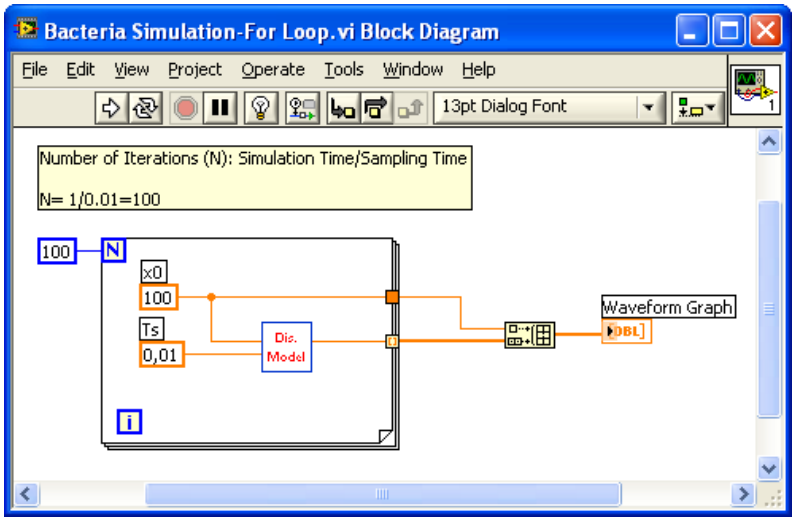
This gives:

$$x_{k+1} = x_k + T_s(bx_k - px_k^2)$$

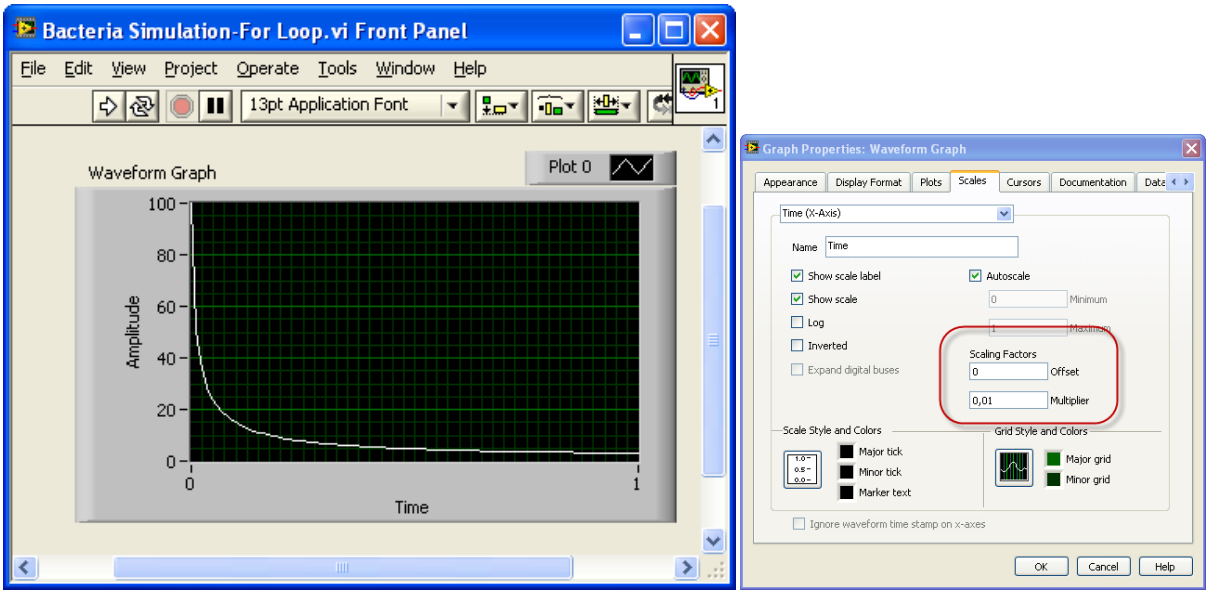
Step 2: We implement the model in the Formula Node and create a Sub VI.



Step 3: We create the simulation program using a For Loop.



We get the following results (note the Scaling Factors set in the Graph Properties):



[End of Example]

Example:

Given the following mathematical model (nonlinear):

$$\dot{x} = -K_1\sqrt{x} + K_2 u$$

We will create a new application in LabVIEW where we simulate this model using a Formula Node to implement the discrete model.

We will use the Euler Forward method (because this is a nonlinear equation):

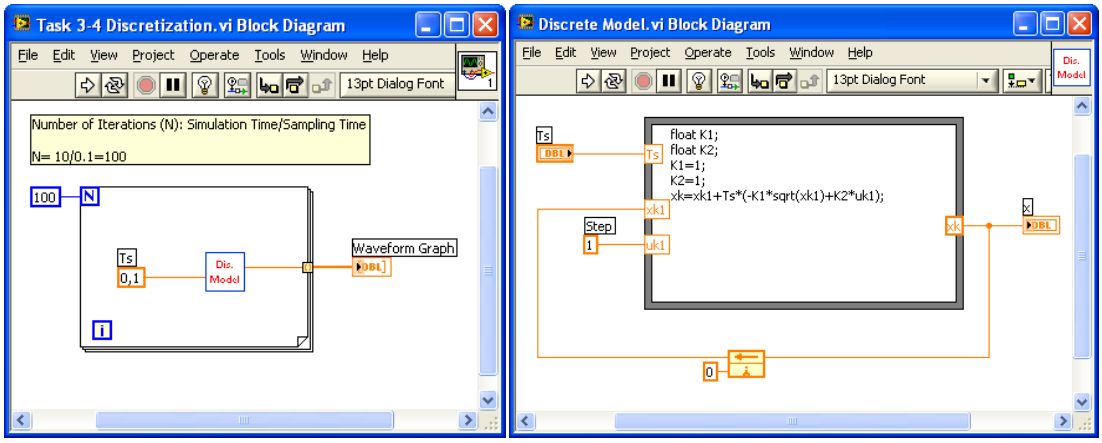
$$\dot{x} \approx \frac{x_{k+1} - x_k}{T_s}$$

This gives:

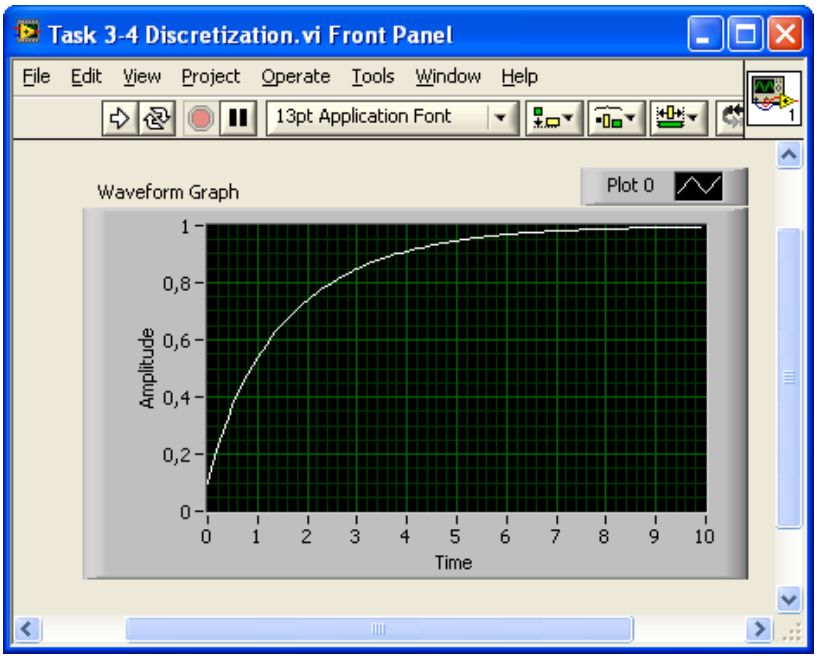
$$\frac{x_{k+1} - x_k}{T_s} = -K_1\sqrt{x} + K_2 u$$

$$\underline{x_k = x_{k-1} + T_s\left[-K_1\sqrt{x_{k-1}} + K_2 u_{k-1}\right]}$$

Block Diagram:



Front Panel:



[End of Example]

Control and Simulation in LabVIEW

Hans-Petter Halvorsen

E-Mail: hans.p.halvorsen@usn.no

Web: https://www.halvorsen.blog

https://www.halvorsen.blog