# About *SchematicSolver* Version 2.0 ©2003-2004 by Lutovac & Tosic

Authors: Miroslav D. Lutovac and Dejan V. Tosic

Welcome to *SchematicSolver*, a powerful and easy-to-use schematic capture, symbolic analysis, processing and implementation tool in *Mathematica*. Using *SchematicSolver*'s unique capabilities and mixed symbolic-numeric processing, you can perform fast and accurate simulations of discrete-time (digital) and continuous-time (analog) systems.

*SchematicSolver* is a convenient and comprehensive environment in which to draw, analyze, solve, design, and implement systems in *Mathematica*. It is the first mouse-driven, interactive drawing tool based entirely on *Mathematica*'s built-in functions and palettes.

With even a minimum understanding of basic system theory, you can successfully use *SchematicSolver* to design and simulate various systems: dynamic feedback and control systems, digital filters, nonlinear discrete-time systems, and much more. For beginners, *SchematicSolver* is perfect for learning and experimenting with system analysis, implementation and design. For advanced and experienced users, *SchematicSolver*'s symbolic analyses and processing provide a sophisticated environment for testing and trying all the "what if" scenarios for system design. Best of all, you can accomplish more in less time with *SchematicSolver* than with traditional prototyping methods.

The *SchematicSolver* application package requires *Mathematica* 4.2.1 or later.

We are dedicated to producing only the finest quality software and supporting customers after the initial purchase. If you encounter problems while using *SchematicSolver* or just need general help, contact us via electronic mail or postal mail and we’ll provide prompt and courteous support.

**Email:**

lutovac@kondor.etf.bg.ac.yu

**Postal mail:**

Miroslav Lutovac

Bulevar Arsenija Carnojevica 219

11000 Belgrade, Serbia, Europe

**Web:**

http://www.wolfram.com/products/applications/schematicsolver/

http://www.SchematicSolver.com

http://kondor.etf.bg.ac.yu/~lutovac

# Contents

# 1. Introduction

## 1.1. What is *SchematicSolver*?

Welcome to *SchematicSolver*, a powerful and easy-to-use schematic capture, symbolic analysis, processing, and implementation tool in *Mathematica*. Using *SchematicSolver*'s unique capabilities and mixed symbolic-numeric processing, you can perform fast and accurate simulations of discrete-time (digital) and continuous-time (analog) systems.

*SchematicSolver* is a convenient and comprehensive environment in which to draw, analyze, solve, design, and implement systems in *Mathematica*. It is the first mouse-driven, interactive drawing tool based entirely on *Mathematica*'s built-in functions and palettes.

You can find many practical solutions in the rich *SchematicSolver*'s documentation, such as velocity servo system, adaptive LMS system, automatic gain control (AGC) system, quadrature amplitude modulation (QAM) system, square-law envelope detector, thermodynamics of a house, high-speed recursive filters, Hilbert transformer, and efficient multirate systems.

*SchematicSolver* has many unique features not available in other software:

- The graphical representation of a system is not a frozen picture (it is not a bitmap image); it changes automatically as you change system parameters or element values.
- A large schematic can be made of replicas of simpler schematics; you can write a code to automate drawing for an arbitrary number of repeated parts.
- Functions exist for generating schematics for arbitrary symbolic system parameters.
- Symbolic signal processing brings you computation of transfer function matrices as closed-form expressions in terms of system parameters kept as symbols, and much more: for a symbolic input sequence you can compute the symbolic output sequence with system parameters and states specified by symbols.
- Automated generation of software implementation of linear and nonlinear discrete systems. The generated implementation function can symbolically process symbolic samples.
- Symbolically derives important closed-form relations between parameters of a system, such as power-complementary property of high-speed filters.
- Find the closed-form symbolic response from the schematic of a linear system keeping system parameters and the state as symbols; all system parameters and the initial conditions are given by symbols and the derived result is the most general.
- Symbolically optimize a selected parameter for the specified response.
- Symbolic design: For known transfer function, impulse, or step response, generates the schematic of the system and computes the system parameters.
- Design of optimal multirate implementations by working in the symbolic domain.
- Model a system that works with symbolic complex signals, such as the Hilbert transformer.
- Find closed-form expressions of output signals for known stimuli given by closed-form expressions for certain classes of nonlinear systems.
- Solve systems with unconnected elements: signals at unconnected element inputs are automatically generated as unique symbols.

## 1.2. Required User Background

With even a minimum understanding of basic system theory, you can successfully use *SchematicSolver* to design, implement, and simulate various systems: dynamic feedback and control systems, digital filters, nonlinear discrete-time systems, and much more. For beginners, *SchematicSolver* is perfect for learning and experimenting with system analysis, implementation and design. For advanced and experienced users, *SchematicSolver*'s symbolic analyses and processing provide a sophisticated environment for testing and trying all the "what if" scenarios for system design. Best of all, you can accomplish more in less time with *SchematicSolver* than with traditional prototyping methods.

## 1.3. Technical Support

We are dedicated to producing only the finest quality software and supporting customers after the initial purchase. If you encounter problems while using *SchematicSolver* or just need general help, contact us via electronic mail or postal mail and we'll provide prompt and courteous support.

NOTE: Please be prepared to provide your name and license number (found on the Registration Card) when contacting us.

**Email:** lutovac@kondor.etf.bg.ac.yu

**Postal mail:** Miroslav Lutovac

Bulevar Arsenija Carnojevica 219

11000 Belgrade, Serbia, Europe

Future versions of *SchematicSolver* are planned so please feel free to write and let us know what features or additions you would like to see. Our goal is to provide a product that will meet your needs and expectations, so feedback from the end user is essential!

For more information:

http://www.wolfram.com/products/applications/schematicsolver/

http://www.SchematicSolver.com

http://kondor.etf.bg.ac.yu/~lutovac

## 1.4. About this Manual

This *User's Guide* has been designed to guide you through *SchematicSolver*'s many features and simplify the retrieval of specific information once you have a working knowledge of the product.

The manual assumes that you are familiar with the operating system and its use of icons, menus, windows and the mouse. It also assumes a basic understanding about how the operating system manages applications (programs and utilities) and documents (data files) to perform routine tasks such as starting applications, opening documents and saving your work.

## 1.5. Manual Conventions

The following conventions are used to identify information needed to perform *SchematicSolver* tasks.

Step-by-step instructions for performing an operation are generally numbered as in the following examples:

1. Select the Adder Element on the Palette.

Menu names, menu commands, and Palette items usually appear in bold type as are text strings to be typed:

2. Type the Value: **3400**.

This manual also includes some special terminology—words that are either unique to schematic capture and system simulation or have some specific meaning within *SchematicSolver*. Such terms are italicized when first introduced.

## 1.6. Teams Up with Other *Mathematica* Applications

*SchematicSolver* complements *Control System Professional* with tools for drawing and solving systems described by block diagrams.

*Control System Professional* is a Wolfram Research application. It uses analytical solutions to study relationships between design elements and gain added insight into complex composite systems, and use numerical solutions for plotting and testing. It handles linear MIMO and SISO systems in both time and frequency domains and provides linearization techniques for non-linear systems. For more information: www.wolfram.com/products/applications/.

*SchematicSolver* provides objects such as transfer functions for further analysis with *Signals and Systems*.

*Signals and Systems* is a Wolfram Research application. It greatly simplifies tasks that involve linear transforms, standard signal representations, and visualization with numerous built-in tools. With a focus on symbolic techniques, these tools bring you capabilities not traditionally available in signal processing software, yet increasingly in demand for high-quality signal analysis. In addition, *Signals and Systems* lets educators easily create interactive lessons and have students derive, explain, and submit their solutions in the same notebook.

For more information:

www.wolfram.com/products/applications/.

*SchematicSolver* has access to all *Mathematica* capabilities to perform further manipulations on results returned by the *SchematicSolver*'s functions.

## 1.7. Acknowledgments

We are thankful to Theodore Gray, Chris Carlson, Louis D'Andria, Igor Bakshee, Jeff Bryant, and Ljiljana Milic for making useful suggestions.

# 2. Quick Tour of *SchematicSolver*

With even a minimum understanding of basic system theory, you can successfully use *SchematicSolver* to design, simulate, and implement various systems: dynamic feedback and control systems, digital filters, nonlinear discrete-time systems, and much more.

Symbolic signal processing is a *SchematicSolver*'s unique feature that brings you computation of transfer function matrices as closed-form expressions in terms of system parameters kept as symbols, and much more: for a symbolic input sequence you can compute the symbolic output sequence with system parameters specified by symbols.

You can find many practical solutions in the rich *SchematicSolver*'s documentation, such as velocity servo system, adaptive LMS system, automatic gain control (AGC) system, quadrature amplitude modulation (QAM) system, square-law envelope detector, thermodynamics of a house, high-speed recursive filters, Hilbert transformer, and efficient multirate systems.

For beginners, *SchematicSolver* is perfect for learning and experimenting with system analysis, implementation and design. For advanced and experienced users, *SchematicSolver*'s symbolic analyses and processing provide a sophisticated environment for testing and trying all the "what if" scenarios for system design. Best of all, you can accomplish more in less time with *SchematicSolver* than with traditional prototyping methods.

The graphical representation of a system is essential for supporting a designer's view of the implementation, which often comes in the form of block diagrams. *SchematicSolver* provides an easy graphical user interface for building models as block diagrams, using point-and-click mouse operations for performing the most common drawing tasks. You can draw the models just as you would with pencil and paper.

*SchematicSolver* describes a system as a list of elements. This list specifies what elements are in the system and how they are interconnected. A list describing a system will be referred to as the *schematic specification*. Each element in the system is also described as a list that states what the element is, to which other elements it is connected, and what its value is. A list describing an element will be referred to as the *element specification*.

When you draw a new element, *SchematicSolver* automatically adds a new element specification in the schematic specification. The schematic specification contains all details for drawing, solving, simulating, and implementing the system. In addition, it is not necessary to insert manually all elements. A large schematic can be made of replicas of other schematics. You can draw smaller parts that constitute the large system and combine them into a desired schematic. Once when you have a set of basic schematics, and when you find out that they can be used to build large schematics with repeated parts, you can write a code to automate drawing for an arbitrary number of repeated parts. This is a unique feature of *SchematicSolver* not available in other software for system modeling and analysis. The graphical representation of a system is not a frozen picture (it is not a bitmap image); it changes automatically as you change system parameters or element values.

Chapter Examples of Discrete System Implementation describes solutions to common modeling problems. You can easily build models from automatically generated schematics and clearly visualize sophisticated algorithms. You can change system parameters on the fly and immediately see what happens with the results because the *SchematicSolver*'s simulations are interactive.

Adaptive LMS system example illustrates (a) useful *modeling* of system identification, (b) *simulation* of the system that performs the least mean squares adaptive algorithm, and (c) automated *code generation* for the implementation of the LMS system. Two systems, the unknown linear system and the adaptive nonlinear system, are represented by two schematic specifications. Usually, the impulse response of the unknown system has a finite duration and it can be modeled as an FIR system with symbolic parameters. The numeric parameter values are determined using the adaptive nonlinear system for known input and output sequences of the unknown system. The schematics of the FIR system and the adaptive system are automatically generated for specified number of the unknown parameters. *SchematicSolver* symbolically processes data samples keeping the system parameters as symbols. *SchematicSolver* proves that adaptive system tries to solve a system of linear equations. Consequently, you can identify the parameters of the unknown system with a small number of samples. Furthermore, *SchematicSolver* can process samples in a traditional numerical way.

Automated procedure for generating software implementation of a nonlinear discrete system is illustrated by the AGC system. Nonlinear function value can be any algebraic function of one argument: an algebraic *Mathematica* built-in function or algebraic user-defined function with symbolic parameters. The implementation procedure embeds the code of the nonlinear function. *SchematicSolver* returns the output sequence with symbolic sample values in terms of symbolic parameters. This enables symbolic optimization and presenting results in a more convenient form. For example, if the input samples of the Modulator system are expressions of the form $\sin(2\pi f_1)$ and $\sin(2\pi f_2)$, the output sample contains $\sin(2\pi f_1)\sin(2\pi f_2)$, that can be simplified to the more convenient form $\frac{1}{2}(\cos(2\pi(f_1 - f_2)) - \cos(2\pi(f_1 + f_2)))$. This example demonstrates a *SchematicSolver*'s unique feature, symbolic processing, not available in other software.

QAM system example illustrates modeling of the system by implementing and simulating the subsystems individually, assuming that there are no feedback paths between the subsystems. The output signal from one subsystem is the input signal to another subsystem. The subsystems may have feedback paths and each subsystem can be analyzed individually; for example, you can find the transfer function of a linear subsystem and plot the frequency response or the impulse response.

Square-law envelope detector is another example of the nonlinear system that demodulates the amplitude-modulated signal. It shows how to start modeling with signals and systems represented by mathematical formulas and arrive to actual processing that act on real data.

Simple model of the thermodynamics of a house provides a brief introduction to the efficient modeling concept; you begin with a symbolic description of an algorithm and then try to manipulate it into other symbolic descriptions having a more desirable form such as schematic specification. The MIMO linear discrete-time heating model can be used to find the frequency response or the step response, to simulate data processing, to implement the system, and to process data samples with the automatically generated implementation function. You can simply upgrade the linear model to a nonlinear model of the heating system by inserting a nonlinear element. Various nonlinear models can be implemented and simulated, such as the model with the parametric on-off function or with the user-defined hysteresis function.

Example of high-speed recursive filters presents automatic generation of schematic from known symbolic values of the filter coefficients. The filter is a single-input two-output system. *SchematicSolver* symbolically derives important closed-form relations between parameters of this system, such as power-complementary property. This is a unique feature of *SchematicSolver* not available in purely numeric simulation software.

Velocity servo system example demonstrates another unique feature of *SchematicSolver* not available in numeric software. First, it finds the closed-form symbolic response from the schematic of a continuous-time system keeping system parameters as symbols; all system parameters are given by symbols and the derived result is the most general. Next, it finds the optimal symbolic value of a selected parameter for the specified response; no numeric value appears in the calculation. Numeric optimum value is computed for a particular set of numeric parameters. Substituting the numeric values into the symbolic expression, you can plot the response.

The rational transfer function, impulse, or step response are sufficient to describe the input/output characteristics of the system. They have enough information to describe the internal workings of a general continuous system implementation or the discrete-time Transposed Direct Form 2 IIR implementation. *SchematicSolver* shows that a linear system can be designed in a straightforward manner if its step response is known as a closed-form expression. It finds the corresponding transfer function as closed-form expressions in terms of system parameters. *SchematicSolver* demonstrates how to manipulate the symbolic expressions into a form that is suitable for automatic code generation. You can generate the schematic of the general block-diagram of the system with symbolic parameters. For the known numeric values of the transfer function coefficients, that are computed from the step response, and for the symbolic coefficients, that are computed from the general schematic of the system, you can compute the system parameters and draw a high-quality schematic of the system.

The graphical representation of a system is essential for supporting a designer's view of the implementation. *Schematic-Solver* can help you to simplify your model graphically. You can modify a schematic specification by inspection and try to find a simpler realization of the system. When you draw a schematic, you can solve the system, that is, you can find the symbolic expressions of the transfer functions, by clicking a single button on the palette. Regardless how complex the expressions are, there is a simple procedure for comparing the symbolic expressions. Although the design space is unbounded, you can try to find more efficient and effective schematics with the same transfer function. In order to evaluate and compare relative cost of different implementations, a figure of merit can be used to quantify the implementation complexity.

Chapter Multirate Systems describes the ability of finding optimal multirate implementation by working in the symbolic domain. Various multirate structures have been analyzed in order to find an efficient implementation within a class of possible solutions. In some cases, you can identify that two structures are equivalent comparing their transfer functions. However, in some cases you should analyze symbolically processed symbolic sequences. *SchematicSolver* works with symbolic input, symbolic parameters, and symbolic states. It processes symbolic sequences and returns the output sequences with symbolic sample values. Symbolic multirate system simulation is the *SchematicSolver*'s unique feature not available in other simulation software.

*SchematicSolver* allows you to model a system that works with complex signals. Chapter Hilbert Transformer illustrates how to generate a complex signal from a real discrete signal by passing the real signal through a linear discrete system referred to as the Hilbert transformer. *SchematicSolver*'s functions compute the spectrum of the complex signals and illustrate that the spectrum of the analytic complex signal has zero-valued spectrum for negative digital frequencies. Schematic of the Hilbert transformer clearly visualizes the processing and it can be automatically generated by the corresponding *SchematicSolver*'s function. QAM system, in which the Hilbert transformer is used, is designed and analyzed as an example of a real system that processes complex signals.

Using *SchematicSolver*'s schematic capabilities, symbolic system analysis and signal processing, you can perform fast and accurate simulations of nonlinear discrete-time systems. *SchematicSolver* can solve some classes of nonlinear systems. The term solve means that *SchematicSolver* can find the closed-form expression of the output signal for a known stimulus given by a closed-form expression. *SchematicSolver* illustrates step-by-step procedures for analyzing nonlinear systems; for the given block-diagram of a system, the required equations are formulated as a system of equations, and then the set of equations is solved to find the system response as a discrete function.

Sometimes, it happens that inputs of some system elements are left unconnected. Traditionally, systems with unconnected element inputs are not solvable. *SchematicSolver* successfully solves these systems: signals at unconnected element inputs are automatically generated as unique symbols. Thus, if you by mistake left unconnected an element input, it is easy to identify the mistake. If you intentionally leave some element inputs unconnected, you can assign values to the corresponding input signals after the analysis.

This makes *SchematicSolver* available:

```
In[1]:= Needs["SchematicSolver`"]
```

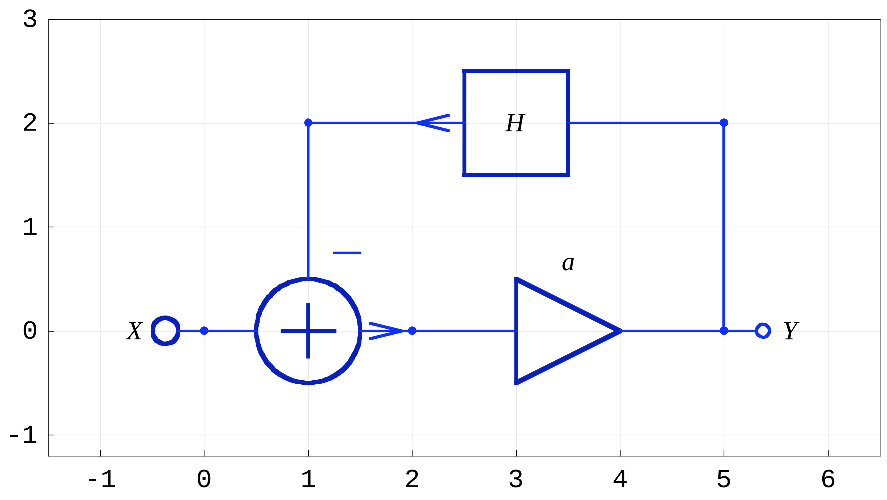
*SchematicSolver* describes a system as a list of elements referred to as the *schematic specification*. A list describing an element will be referred to as the *element specification*. The junction points between elements are referred to as *nodes*.

Here is an example system specification:

```
In[2]:= mySystem = {
          {"Input", {0, 0}, X},
          {"Adder", {{0, 0}, {1, -2}, {2, 0}, {1, 2}}, {1, 0, 2, -1}},
          {"Multiplier", {{2, 0}, {5, 0}}, a},
          {"Block", {{5, 2}, {1, 2}}, H},
          {"Line", {{5, 0}, {5, 2}}},
          {"Output", {5, 0}, Y}
        };
```

`ShowSchematic` shows the system schematic:

```
In[3]:= ShowSchematic[mySystem, PlotRange → {{-1.5, 6.5}, {-1.2, 3}}]
```



The system consists of one adder, one multiplier with gain *a*, and one block of transfer function *H*.

*Linear time-invariant* (LTI) systems, characterized by linear equations, are efficiently analyzed by using the *Laplace* or *z* transforms. The transforms map the equations into new algebraic equations which are easier to manipulate.

`DiscreteSystemTransferFunction` finds the transfer function of the discrete system:

```
In[4]:= {tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[mySystem]
```

Out[4]= $\left\{\left\{\left\{\frac{a}{1+a\,H}\right\}\right\}, \{Y[\{0, 0\}]\}, \{Y[\{5, 0\}]\}\right\}$

The system input is at the coordinate {0,0} and the system output is at the coordinate {5,0}.

The example system is a SISO (single-input single-output) system; therefore, its transfer function matrix is a 1-by-1 matrix.

```
In[5]:= myTF = tfMatrix[[1, 1]]
```

Out[5]= $\frac{a}{1+a\,H}$

Consider a special case in which we assign some numeric and symbolic values to the system parameters:

```
In[6]:= myValues = {a → 1 / 2, H → 1 / z}
```

Out[6]= $\left\{a \to \frac{1}{2}, H \to \frac{1}{z}\right\}$

Here is the transfer function for the special case:

```
In[7]:=  myTFspecial = myTF /. myValues
```

Out[7]= $\frac{1}{2\left(1+\frac{1}{2z}\right)}$

`DiscreteSystemDisplayForm` displays the transfer function in a more convenient way:

```
In[8]:=  myTFspecial // DiscreteSystemDisplayForm
```

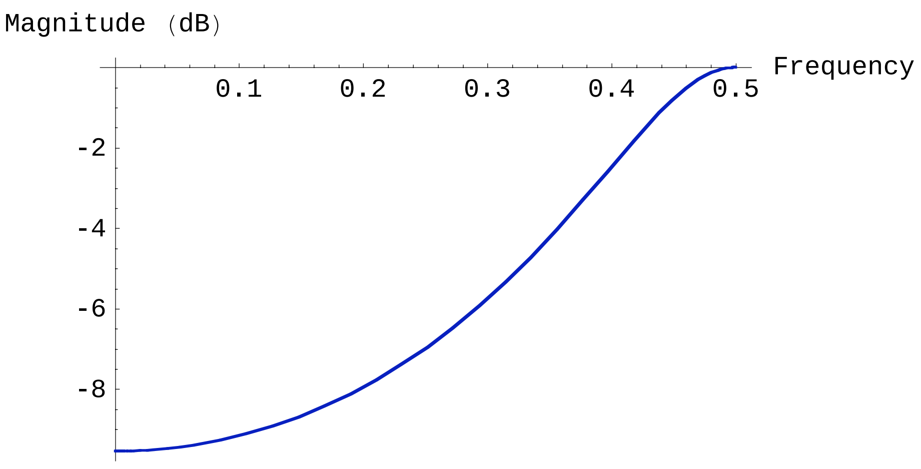
Out[8]//DisplayForm=

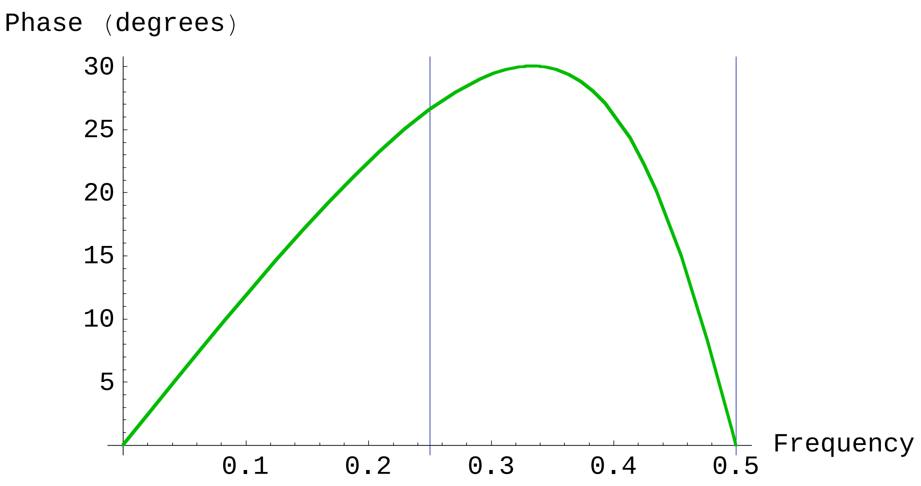
$$\frac{1}{2+z^{-1}}$$

By default, *SchematicSolver* denotes the complex variable with `z`, and the transforms of signals with `Y[{i,j}]` where pairs `{i,j}` designate coordinates on the schematic.

`DiscreteSystemFrequencyResponse` plots the frequency response of the system:

```
In[9]:=  DiscreteSystemFrequencyResponse[myTFspecial];
```



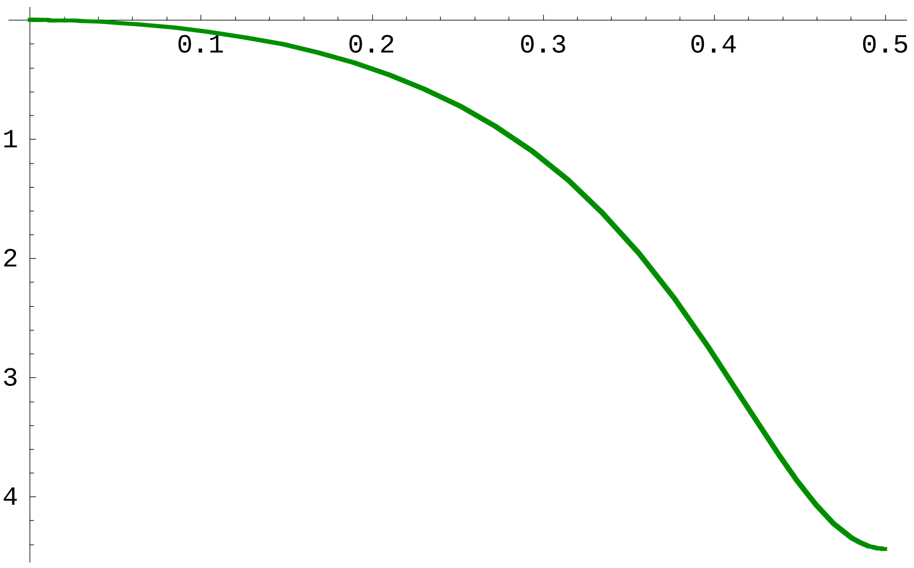


*SchematicSolver* can keep all system parameters as symbols. You can assign various expressions to the parameters:

```
In[10]:=  myTFnested = myTF /. {a → 3 / 2, H → myTFspecial};
          DiscreteSystemDisplayForm[myTFnested]
          DiscreteSystemMagnitudeResponsePlot[myTFnested /. a → 3 / 2];
```

Out[11]//DisplayForm=

$$\frac{6+3\,z^{-1}}{7+2\,z^{-1}}$$

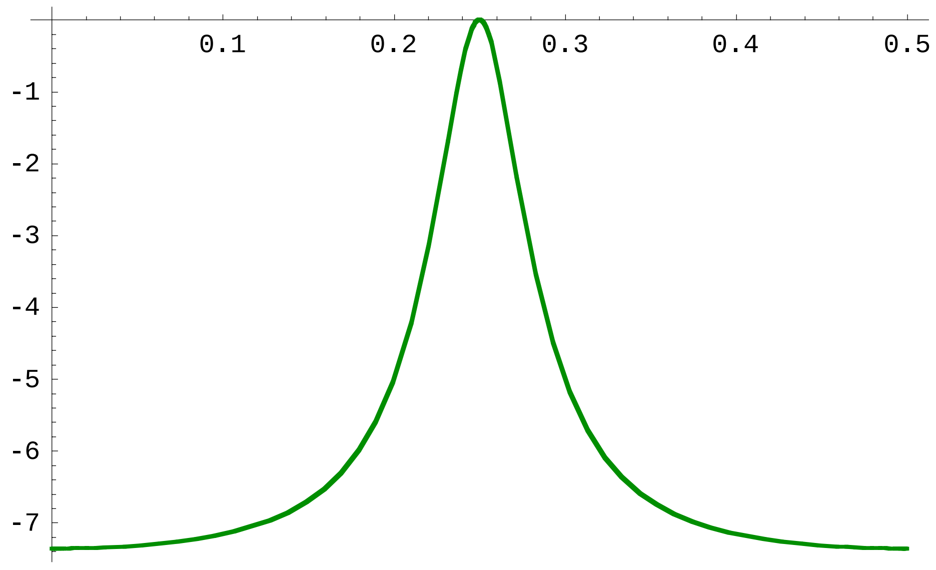


Here is another example of a specific system:

```
In[13]:=  myTFhb = myTF /. {a → 1, H → (1 + z^2) / (1 / 2 + z^2)};
          DiscreteSystemDisplayForm[myTFhb]
          DiscreteSystemMagnitudeResponsePlot[myTFhb];
```

Out[14]//DisplayForm=

$$\frac{2+z^{-2}}{4+3\,z^{-2}}$$



`DiscreteSystemSignals` computes transforms of signals at all nodes:

```
In[16]:=  DiscreteSystemSignals[mySystem] // TableForm
```

*Out[16]//TableForm=*

| $\frac{a X}{1+a H}$ | $\frac{X}{1+a H}$ | $\frac{a H X}{1+a H}$ | X |
|---|---|---|---|
| Y[{5, 0}] | Y[{2, 0}] | Y[{1, 2}] | Y[{0, 0}] |

`DiscreteSystemEquations` sets up the equations that describe the system:

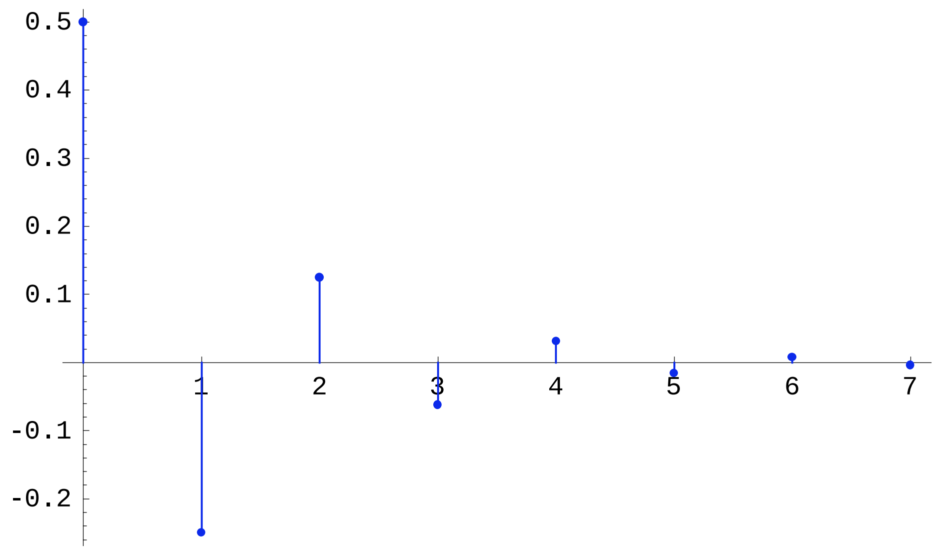
```
In[17]:=  DiscreteSystemEquations[mySystem] // First // ColumnForm

Out[17]=  Y[{0, 0}] == X
          Y[{2, 0}] == Y[{0, 0}] - Y[{1, 2}]
          Y[{5, 0}] == a Y[{2, 0}]
          Y[{1, 2}] == H Y[{5, 0}]
```

`DiscreteSystemProcessingSISO` processes a data list inputted to the system for the transfer function found from the schematic:

```
In[18]:=  myInputData = {1, 0, 0, 0, 0, 0, 0, 0}

Out[18]=  {1, 0, 0, 0, 0, 0, 0, 0}

In[19]:=  myOutput = DiscreteSystemProcessingSISO[myInputData, myTFspecial] // First
```

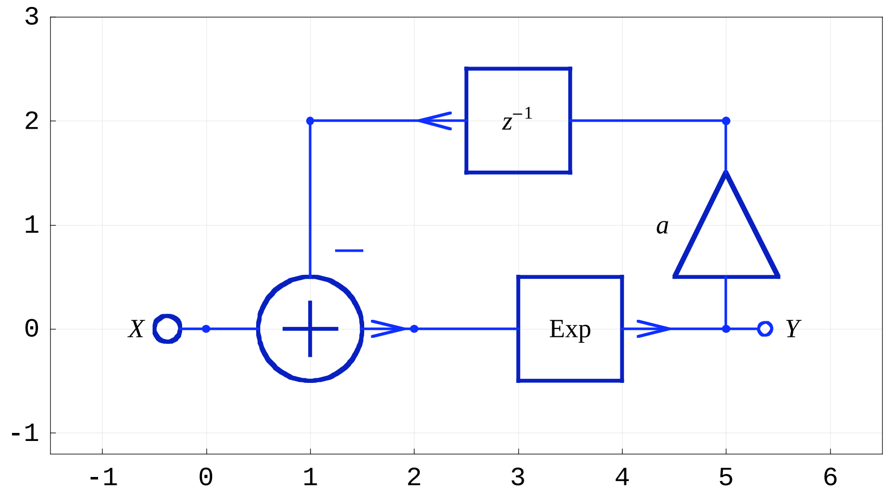
*Out[19]=* $\left\{\frac{1}{2}, -\frac{1}{4}, \frac{1}{8}, -\frac{1}{16}, \frac{1}{32}, -\frac{1}{64}, \frac{1}{128}, -\frac{1}{256}\right\}$

```
In[20]:=  SequencePlot[ListToSequence[myOutput]];
```



Consider a nonlinear system

```
In[21]:= mySystem = {
            {"Input", {0, 0}, X},
            {"Adder", {{0, 0}, {1, -2}, {2, 0}, {1, 2}}, {1, 0, 2, -1}},
            {"Function", {{2, 0}, {5, 0}}, Exp},
            {"Multiplier", {{5, 0}, {5, 2}}, a},
            {"Delay", {{5, 2}, {1, 2}}, 1},
            {"Output", {5, 0}, Y}};
         ShowSchematic[%, PlotRange → {{-1.5, 6.5}, {-1.2, 3}}]
```



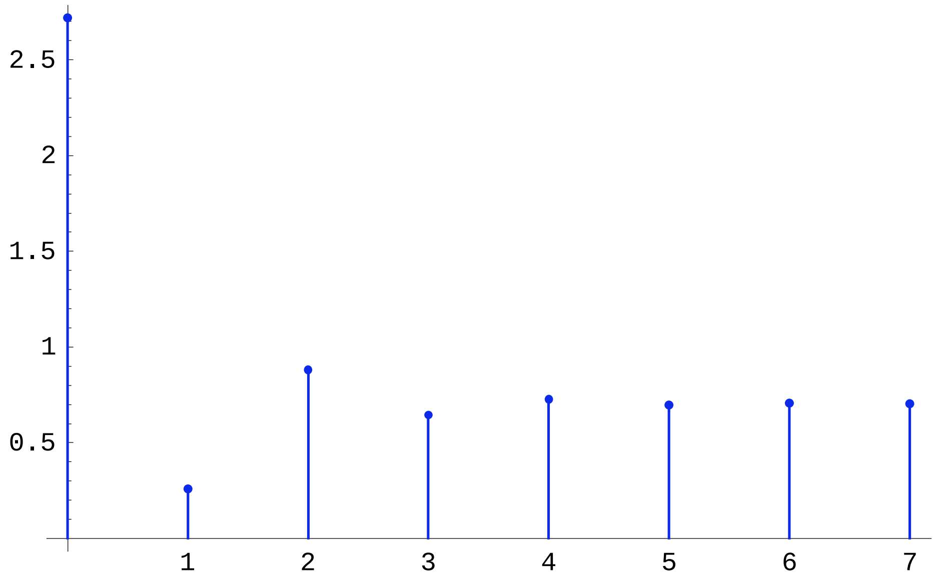
Here is the impulse response of the system:

```
In[23]:= myOutSeq = DiscreteSystemSimulation[mySystem]
```

Out[23]=
$$\left\{\{e\}, \{e^{-a\,e}\}, \{e^{-a\,e^{-a\,e}}\}, \left\{e^{-a\,e^{-a\,e^{-a\,e}}}\right\}, \left\{e^{-a\,e^{-a\,e^{-a\,e^{-a\,e}}}}\right\}, \left\{e^{-a\,e^{-a\,e^{-a\,e^{-a\,e^{-a\,e}}}}}\right\}, \left\{e^{-a\,e^{-a\,e^{-a\,e^{-a\,e^{-a\,e^{-a\,e}}}}}}\right\}, \left\{e^{-a\,e^{-a\,e^{-a\,e^{-a\,e^{-a\,e^{-a\,e^{-a\,e}}}}}}}\right\}\right\}$$

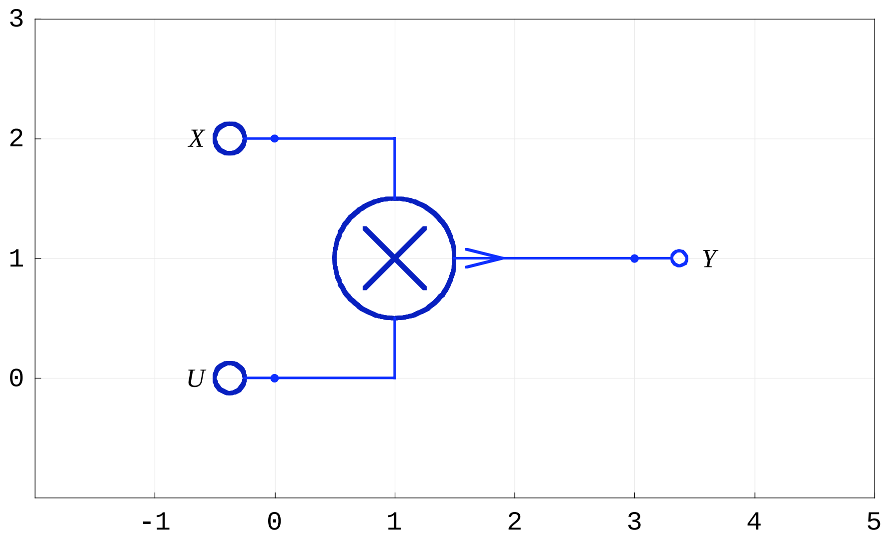
Note that the system parameters are symbols and that the response is a list of symbolic expressions. You can always assign numeric values to the parameters, for example to plot the response, as follows:

```
In[24]:= SequencePlot[myOutSeq /. a → 1 / 2];
```



*SchematicSolver* can solve nonlinear discrete-time systems. Here is an example two-input modulator system:

```
In[25]:=  modulatorSystem = {{"Input", {0, 2}, X},
             {"Input", {0, 0}, U},
             {"Output", {3, 1}, Y},
             {"Modulator", {{0, 1}, {0, 0}, {3, 1}, {0, 2}}, {0, 1, 2, 1}}};
          ShowSchematic[%, PlotRange → {{-2, 5}, {-1, 3}}];
```



Output *Y* is the product of two sinusoidal signals *X* and *U*.

```
In[27]:=  x = UnitSineSequence[8, Fx];
          u = UnitSineSequence[8, Fu];
          inpSeq = MultiplexSequence[x, u];
```

`DiscreteSystemSimulation` simulates the system:

```
In[30]:=  outSeq = DiscreteSystemSimulation[modulatorSystem, inpSeq];
```

*SchematicSolver* works with symbolic signals:

```
In[31]:=  dataSeq = MultiplexSequence[inpSeq, outSeq];
          % // TableForm
```

```
  Out[32]//TableForm=
        0                0                0
        Sin[2 Fx π]      Sin[2 Fu π]      Sin[2 Fu π] Sin[2 Fx π]
        Sin[4 Fx π]      Sin[4 Fu π]      Sin[4 Fu π] Sin[4 Fx π]
        Sin[6 Fx π]      Sin[6 Fu π]      Sin[6 Fu π] Sin[6 Fx π]
        Sin[8 Fx π]      Sin[8 Fu π]      Sin[8 Fu π] Sin[8 Fx π]
        Sin[10 Fx π]     Sin[10 Fu π]     Sin[10 Fu π] Sin[10 Fx π]
        Sin[12 Fx π]     Sin[12 Fu π]     Sin[12 Fu π] Sin[12 Fx π]
        Sin[14 Fx π]     Sin[14 Fu π]     Sin[14 Fu π] Sin[14 Fx π]
```

The output signal can be presented in a more convenient form that reveals output as a sum of two sinusoidal signals of frequencies `(Fu-Fx)` and `(Fu+Fx)`:

```
In[33]:= (outSeq // Flatten // TrigReduce) //. f_[e_] :→ f[Factor[e]];
         % // MatrixForm
```

Out[34]//MatrixForm=

$$\begin{pmatrix} 0 \\ \frac{1}{2}\,(\mathrm{Cos}[2\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[2\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \\ \frac{1}{2}\,(\mathrm{Cos}[4\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[4\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \\ \frac{1}{2}\,(\mathrm{Cos}[6\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[6\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \\ \frac{1}{2}\,(\mathrm{Cos}[8\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[8\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \\ \frac{1}{2}\,(\mathrm{Cos}[10\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[10\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \\ \frac{1}{2}\,(\mathrm{Cos}[12\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[12\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \\ \frac{1}{2}\,(\mathrm{Cos}[14\,(\mathrm{Fu}-\mathrm{Fx})\,\pi]-\mathrm{Cos}[14\,(\mathrm{Fu}+\mathrm{Fx})\,\pi]) \end{pmatrix}$$

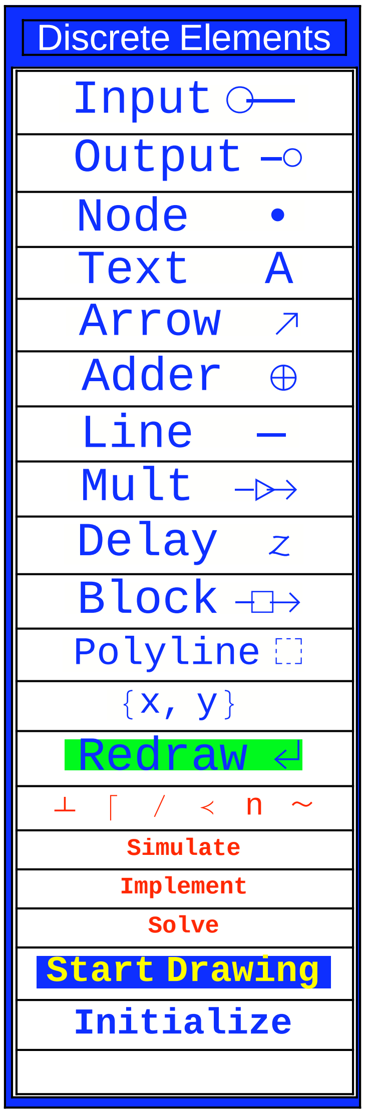
With a focus on symbolic techniques, *SchematicSolver* brings you capabilities not traditionally available in signal processing software.

Palettes provide a simple way to access the full range of *SchematicSolver*'s drawing and solving capabilities.

The *SchematicSolver*'s palettes provide an easy point-and-click interface for performing the most common drawing tasks. However, advanced users might prefer to type and evaluate functions directly. But for users who only want to perform the basic operations, these palettes provide the simplest alternative.

If a palette is not open, choose, e.g., the DiscreteElements palette with

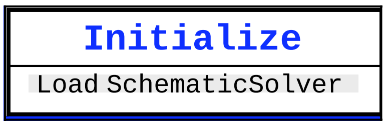
File ▹ Palettes ▹ DiscreteElements



**To Start Drawing a New Schematic**

**1.** Place the insertion point in a new empty cell in your notebook.

**2.** Click the button `Initialize` on the palette to load *SchematicSolver*:



An input cell will be opened with pasted text, as shown below, and then the whole cell will be evaluated:

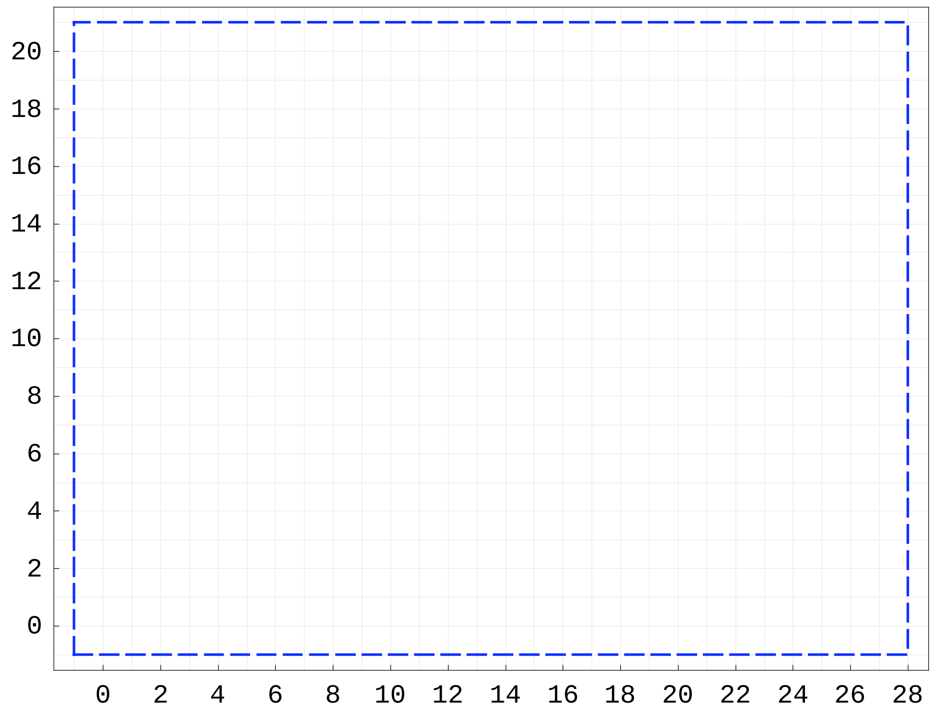
```
In[35]:= Needs["SchematicSolver`"];
             SetOptions[InputNotebook[], ImageSize → {350, 300}, WindowSize → {500, 600}];
```

Palette footer, below the button `Initialize`, indicates the function of this button.

**3.** Click the button `Start Drawing`

A new input cell will be opened with pasted text. Then the whole cell will be evaluated producing a new graphic output cell below the input cell:

```
In[37]:= mySchematic = {

           {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
           ShowSchematic[%];
```



Cy clicking the button `Start Drawing` a new schematic (typically, a system specification) is generated with only one annotation element — Polyline. The `ShowSchematic` function shows the *drawing workspace* with grid lines. By default, the list of elements that describe the schematic is named `mySchematic`. We call this list the *schematic specification*.

**4.** Place the insertion point in the empty line in your schematic specification, above the drawing workspace.

**5.** To draw an input element, click the button `Input`

Move the mouse over the drawing workspace. Click once, say when the mouse position is over the coordinate {5, 10}. The coordinate {5,10} is selected, and it appears in the Input element specification.

The Input element specification is pasted at the current insertion point:

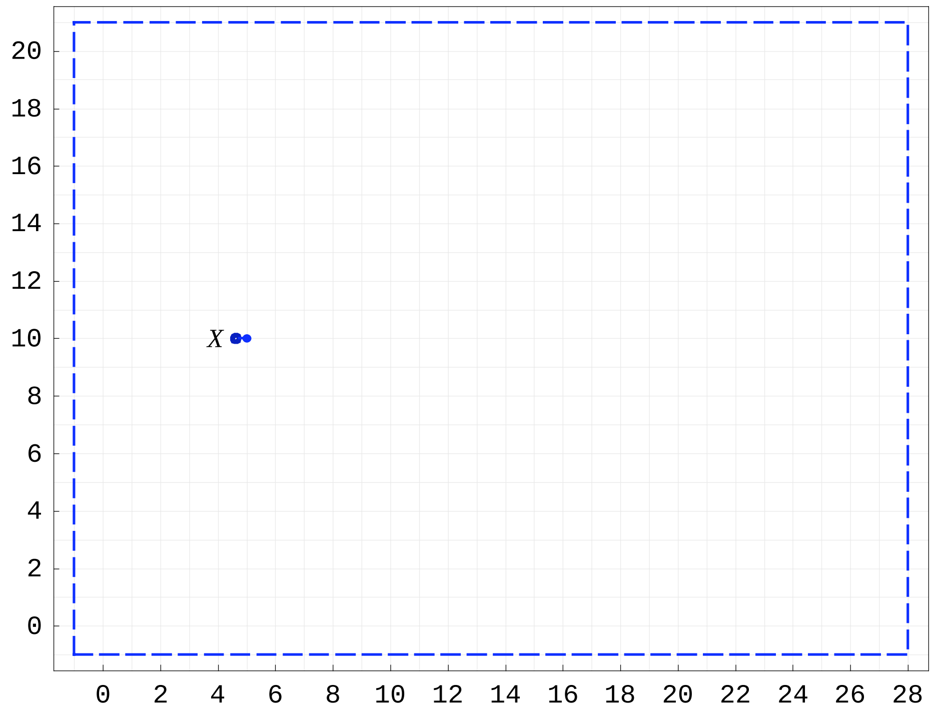
```
{"Input", {5, 10}, X, "", TextOffset → {1, 0}},
```

The schematic specification changes and it has a new element above the empty line.

The insertion point remains in the empty line. The drawing workspace does not change until you evaluate the cell with the schematic specification.

**6.** Click the button `Redraw` to redraw the schematic:

```
In[39]:= mySchematic = {
          {"Input", {5, 10}, X, "", TextOffset → {1, 0}},

          {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
          ShowSchematic[%];
```



The cell insertion bar appears below the drawing workspace.

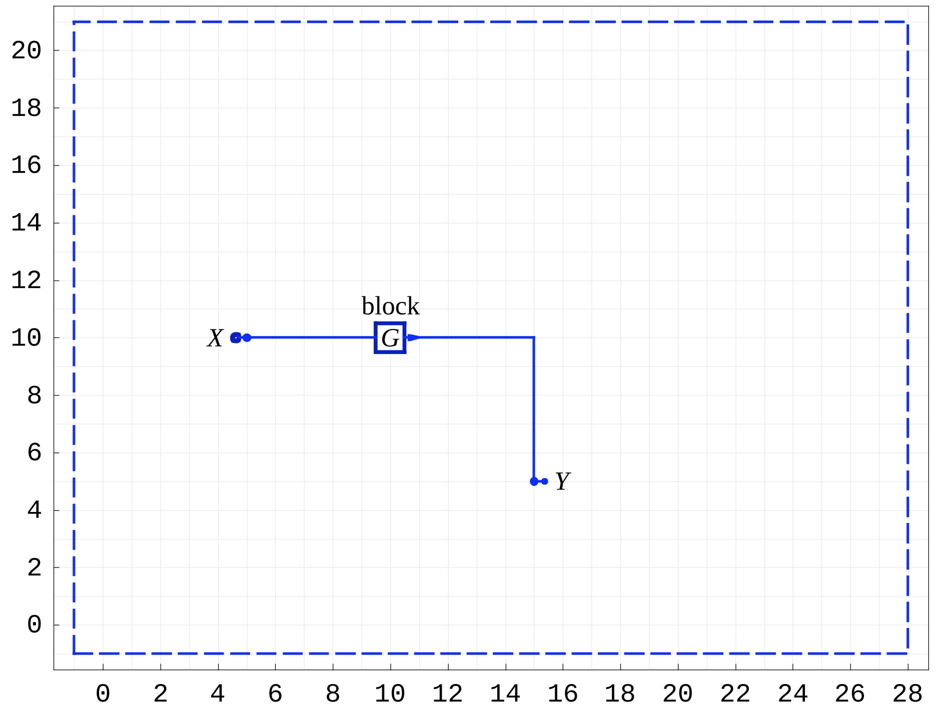
**7.** Place the insertion point in the empty line in your schematic specification, above the drawing workspace.

**8.** You can continue filling in your schematic specification with other elements. For example, to add the Block element, click the button `Block`. Move the mouse over the drawing workspace. Press and hold the mouse button, say when the mouse position is over the coordinate {5, 10}. Drag the mouse to specify the second coordinate. Release the mouse button, say at {15, 5}. The schematic specification changes and it has a new element above the empty line.

In a similar way, you can add the Output element at {15, 5}.

Here is the corresponding schematic specification and the block diagram:

```
In[41]:= mySchematic = {
          {"Input", {5, 10}, X, "", TextOffset → {1, 0}},
         {"Block", {{5, 10}, {15, 5}}, G, "block"},
         {"Output", {15, 5}, Y, "", TextOffset → {-1, 0}},

          {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
          ShowSchematic[%];
```



Typically, we want to solve the system: to find the system response, or to compute the transfer function. The palette button **Solve** pastes and evaluates a template for general solving a system. The button **Solve** assumes that the name of the schematic specification is `mySchematic`:

```
In[7]:= Print["Equations of the System:"];
        {myEquations, myVars} = DiscreteSystemEquations[mySchematic];
        myEquations // ColumnForm
        Print["Response of the System:"];
        {myResponse, myVars} = DiscreteSystemResponse[mySchematic];
        myResponse // ColumnForm
        Print["Signals of the System:"];
        {mySignals, myVars} = DiscreteSystemSignals[mySchematic];
        % // Transpose // TableForm
        Print["Transfer Function Matrix:"];
        {myTF, myInputs, myOutputs} = DiscreteSystemTransferFunction[mySchematic];
        myTF // MatrixForm
        Print["Inputs of the System:"];
        myInputs
        Print["Outputs of the System:"];
        myOutputs
        Print["End of SchematicSolver Solving"];

          Equations of the System:

Out[9]= Y[{5, 10}] == X
        Y[{15, 5}] == G Y[{5, 10}]

          Response of the System:

Out[12]= Y[{15, 5}] → G X
         Y[{5, 10}] → X

          Signals of the System:

  Out[15]//TableForm=
       G X      Y[{15, 5}]
       X        Y[{5, 10}]

          Transfer Function Matrix:

  Out[18]//MatrixForm=
       ( G )

          Inputs of the System:

Out[20]= {Y[{5, 10}]}

          Outputs of the System:

Out[22]= {Y[{15, 5}]}

   End of SchematicSolver Solving
```

**Further reading:**

Chapter 4 Solving Systems

Chapter 5 Examples of Solving Systems

Chapter 6 Solving Large Systems

Chapter 9 Examples of Discrete System Implementation

Chapter 10 Hilbert Transformer

Chapter 11 Multirate Systems

Chapter 12 Hierarchical Systems

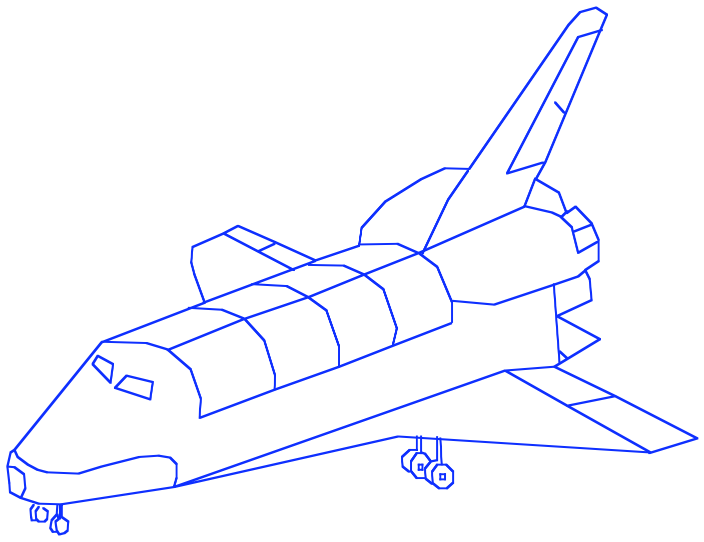
Chapter 13 Palettes for Drawing and Solving Systems

Chapter 15 Processing with *SchematicSolver*

Running Demo in GettingStarted.nb

*SchematicSolver* has many other distinguished features: for example, you can use *SchematicSolver* to create linearts, such as a lineart of the Space Shuttle.

```
In[43]:=  ShowSchematic[SchematicSolverFigureShuttle, GridLines → None, Frame → False];
```



# 3. System Representation

## 3.1. Basic Definitions

*System* is usually defined as a group of related parts, called *elements*, working together. A system takes one or more *signals* as *input*, performs operations on the signals, and produces one or more signals as *output*. Therefore, the input is the *stimulus* or excitation applied to a system from an external source, usually in order to produce a specified *response*. The output is the actual response obtained from a system.

From an implementation point-of-view, a system is an arrangement of physical components connected or related in such a manner as to form and/or act as an entire unit. From a signal processing perspective, a system can be viewed as any process that results in the transformation of signals, in which systems act on signals in prescribed ways.

A system is said to be a *SISO* (*single-input single-output*) *system* if it has only one input and only one output. A system is said to be a *MIMO* (*multiple-input multiple-output*) *system* if it has more than one input or more than one output.

An equation that describes the relation between the input and the output of a system is called the *input-output relationship*, also known as the external description or the input-output description, of the system. In developing this relationship, we assume that the knowledge of the internal structure of a system is unavailable to us. Instead, the only access to the system is by means of the input ports and the output ports. Under this assumption, a system may be considered as a "*black box*."

In a *continuous-time system*, the input and output signals are continuous-time. A *discrete-time system* has discrete-time input and output signals.

A discrete-time system is *digital* if it operates on discrete-time signals whose amplitudes are quantized. Quantization maps each continuous amplitude level into a binary number.

*Analysis* of a system is investigation of the properties and the behavior (response) of an existing system. *Design* of a system is the choice and arrangement of systems components to perform a specific task.

In order to analyze, design and implement a system, the description of its components and their interconnections must be put into a suitable form. A mathematical or graphical representation of a system is called the *model*.

A *mathematical model* is a set of mathematical relations representing the system. The solution of these equations represents the system's behavior.

A more detailed introduction to signals and systems can be found in the book

M. D. Lutovac, D. V. Tosic and B. L. Evans, *Filter Design for Signal Processing Using MATLAB and Mathematica*, Upper Saddle River, NJ: Prentice Hall, 2001.

## 3.2. Loading *SchematicSolver*

*SchematicSolver* is one of many available *Mathematica* applications and is normally installed in a separate directory, SchematicSolver, in parallel to other applications. If this has been done at the installation stage, the application package should be visible to *Mathematica* without further effort on your part. Then, to make all the functionality of the application package available at once, you simply load the package with the `Get` or `Needs` command.

This makes *SchematicSolver* available:

```
In[1]:=  Needs["SchematicSolver`"];
```

## 3.3. Block Diagrams

A *block diagram* is a shorthand pictorial representation of the cause and effect relationship between the input and output of a system. It provides a convenient and useful method for characterizing the functional relationships among the various components of a system.

Block diagrams are representations of either the schematic diagram of a physical system or the set of mathematical equations characterizing its parts.

Firstly, we specify some options to better present the examples of this section:

```
In[2]:=  Needs["SchematicSolver`"];
         SetOptions[ShowSchematic, Frame → False,
           GridLines → None, PlotRange → {{-3, 5}, {-1.2, 1.2}}];
```

Each system has at least one input that *SchematicSolver* represents as a list

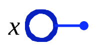
```
In[4]:=  myInput = {"Input", {0, 0}, x}

Out[4]=  {Input, {0, 0}, x}
```

A simple system containing only this element *SchematicSolver* represents as a list

```
In[5]:=  mySystem = {myInput}

Out[5]=  {{Input, {0, 0}, x}}
```

*SchematicSolver* graphically shows a system with the `ShowSchematic` function

```
In[6]:=  mySystem // ShowSchematic
```



Any system has at least one output that is represented by a list

```
In[7]:=  myOutput = {"Output", {2, 0}, y}

Out[7]=  {Output, {2, 0}, y}
```

A system that contains one input and one output, *SchematicSolver* represents as a list o two items

```
In[8]:=  mySystem = {myInput, myOutput}

Out[8]=  {{Input, {0, 0}, x}, {Output, {2, 0}, y}}
```

The schematic of this system looks like

```
In[9]:=  mySystem // ShowSchematic
```

The simplest form of the block diagram is the single *block*, with one input and one output. *SchematicSolver* represents a block as a list

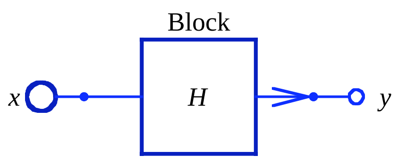
```
In[10]:=  myBlock = {"Block", {{0, 0}, {2, 0}}, H, "Block"}

Out[10]=  {Block, {{0, 0}, {2, 0}}, H, Block}
```

A three-element system, with one input, one output and one block, is represented by

```
In[11]:=  mySystem = {myInput, myOutput, myBlock}

Out[11]=  {{Input, {0, 0}, x}, {Output, {2, 0}, y}, {Block, {{0, 0}, {2, 0}}, H, Block}}
```

The corresponding schematic is

```
In[12]:= mySystem // ShowSchematic
```



The interior of the rectangle representing the block usually contains

(a) the name of the element,

(b) a description of the element, or

(c) the symbol for the mathematical operation to be performed on the input to yield the output.

The *arrows* represent the direction of unilateral information or signal flow.

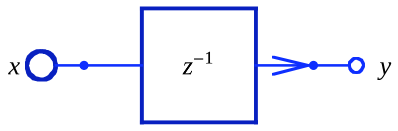
The standard symbols used to represent various types of blocks are

a) *Delay* of a discrete system, $y(n) = x(n-1)$,

```
In[13]:= myDelay = {"Delay", {{0, 0}, {2, 0}}}

Out[13]= {Delay, {{0, 0}, {2, 0}}}
```

A system with one input, one output, and one delay is represented by

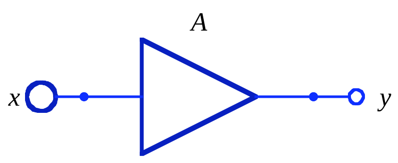
```
In[14]:= mySystem = {myInput, myOutput, myDelay}

Out[14]= {{Input, {0, 0}, x}, {Output, {2, 0}, y}, {Delay, {{0, 0}, {2, 0}}}}

In[15]:= mySystem // ShowSchematic
```



b) *Multiplier* by constant, *Gain*, or *Amplifier*, $y = A\,x$,

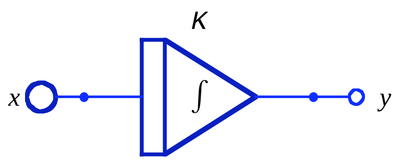
```
In[16]:= myMultiplier = {"Multiplier", {{0, 0}, {2, 0}}, A}

Out[16]= {Multiplier, {{0, 0}, {2, 0}}, A}

In[17]:= mySystem = {myInput, myOutput, myMultiplier}

Out[17]= {{Input, {0, 0}, x}, {Output, {2, 0}, y}, {Multiplier, {{0, 0}, {2, 0}}, A}}
```

```
In[18]:=  mySystem // ShowSchematic
```



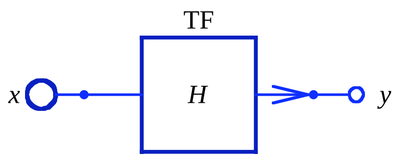
c) *Integrator* with respect to time, $y(t) = K \int x(t)\, dt$.

```
In[19]:=  myIntegrator = {"Integrator", {{0, 0}, {2, 0}}, K}

Out[19]=  {Integrator, {{0, 0}, {2, 0}}, K}

In[20]:=  mySystem = {myInput, myOutput, myIntegrator}

Out[20]=  {{Input, {0, 0}, x}, {Output, {2, 0}, y}, {Integrator, {{0, 0}, {2, 0}}, K}}

In[21]:=  mySystem // ShowSchematic
```



d) Transfer function *Block* element, $Y = H\ X$.

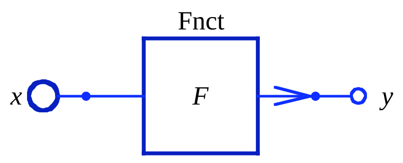
```
In[22]:=  myTF = {"Block", {{0, 0}, {2, 0}}, H, "TF"}

Out[22]=  {Block, {{0, 0}, {2, 0}}, H, TF}

In[23]:=  mySystem = {myInput, myOutput, myTF}

Out[23]=  {{Input, {0, 0}, x}, {Output, {2, 0}, y}, {Block, {{0, 0}, {2, 0}}, H, TF}}

In[24]:=  mySystem // ShowSchematic
```



e) *Function* element, $y = F(x)$.

```
In[25]:=  myFunction = {"Function", {{0, 0}, {2, 0}}, F, "Fnct"}

Out[25]=  {Function, {{0, 0}, {2, 0}}, F, Fnct}
```

```
In[26]:= mySystem = {myInput, myOutput, myFunction}

Out[26]= {{Input, {0, 0}, x}, {Output, {2, 0}, y}, {Function, {{0, 0}, {2, 0}}, F, Fnct}}

In[27]:= mySystem // ShowSchematic
```

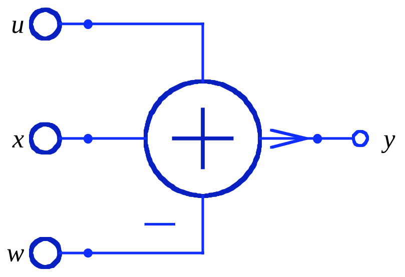


The operations of addition and subtraction are represented by a circle, referred to as *Adder*, also called a *summing point*, with the appropriate minus sign associated with the lines entering the circle.

```
In[28]:= myAdder = {"Adder", {{0, 0}, {0, -1}, {2, 0}, {0, 1}}, {1, -1, 2, 1}}

Out[28]= {Adder, {{0, 0}, {0, -1}, {2, 0}, {0, 1}}, {1, -1, 2, 1}}
```

Let us form a system with one adder, three inputs, and one output:

```
In[29]:= mySystem = {myInput, myOutput, myAdder, {"Input", {0, 1}, u}, {"Input", {0, -1}, w}}

Out[29]= {{Input, {0, 0}, x}, {Output, {2, 0}, y},
          {Adder, {{0, 0}, {0, -1}, {2, 0}, {0, 1}}, {1, -1, 2, 1}},
          {Input, {0, 1}, u}, {Input, {0, -1}, w}}

In[30]:= mySystem // ShowSchematic
```



The output is the algebraic sum of the inputs. In the above example, $y = u + x - w$.

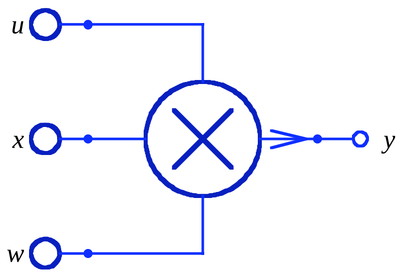
The operation of multiplication is also represented by a circle, referred to as *Modulator*,

```
In[31]:= myModulator = {"Modulator", {{0, 0}, {0, -1}, {2, 0}, {0, 1}}, {1, 1, 2, 1}}

Out[31]= {Modulator, {{0, 0}, {0, -1}, {2, 0}, {0, 1}}, {1, 1, 2, 1}}
```

Let us form a system with one modulator, three inputs, and one output:

```
In[32]:= mySystem =
          {myInput, myOutput, myModulator, {"Input", {0, 1}, u}, {"Input", {0, -1}, w}}

Out[32]= {{Input, {0, 0}, x}, {Output, {2, 0}, y},
          {Modulator, {{0, 0}, {0, -1}, {2, 0}, {0, 1}}, {1, 1, 2, 1}},
          {Input, {0, 1}, u}, {Input, {0, -1}, w}}
```

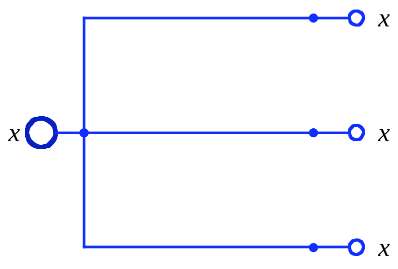
```
In[33]:=  mySystem // ShowSchematic
```



The output is the product of the inputs. In the above example, $y = u\,x\,w$.

In order to employ the same signal or variable as an input to more than one block or summing point, a *takeoff point* is used. *SchematicSolver* uses the *Line* element to represent the takeoff point.

```
In[34]:=  mySystem = {myInput,
            {"Line", {{0, 0}, {0, 1}, {2, 1}}},
            {"Line", {{0, 0}, {2, 0}}},
            {"Line", {{0, 0}, {0, -1}, {2, -1}}},
            {"Output", {2, 1}, x}, {"Output", {2, 0}, x}, {"Output", {2, -1}, x}}

Out[34]=  {{Input, {0, 0}, x}, {Line, {{0, 0}, {0, 1}, {2, 1}}},
           {Line, {{0, 0}, {2, 0}}}, {Line, {{0, 0}, {0, -1}, {2, -1}}},
           {Output, {2, 1}, x}, {Output, {2, 0}, x}, {Output, {2, -1}, x}}

In[35]:=  mySystem // ShowSchematic
```



Takeoff point permits the signal to proceed unaltered along several different paths to several destinations.

The blocks representing the various components of a system are connected in a fashion which characterizes their functional relationship within the system. The arrows connecting one block with another represent the direction of flow of signals or information.

In general, a block diagram consists of a specific configuration of five types of elements: 1) blocks, 2) summing points, 3) modulators, 4) takeoff points, and 5) arrows representing unidirectional signal flow.

*SchematicSolver* represents a system as a list of elements, and each element is specified by a list of items that state what elements are in the system and how they are interconnected.

## 3.4. Discrete Elements

### Introduction

*SchematicSolver* describes a system as a list of elements; this list specifies what elements are in the system and how they are interconnected. A list describing a system will be referred to as the *system specification* or *schematic specification*.

Each element in the system is also described as a list that states what the element is, to which other elements it is connected, and what its value is. A list describing an element will be referred to as the *element specification*.

The junction points between elements are referred to as *nodes*.

*SchematicSolver* supports various discrete elements that can be used to describe a discrete-time system or a digital system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[36]:=  Needs["SchematicSolver`"];
```

## Input Element

*Input* is the stimulus or excitation applied to a system from an external source. It is described by a list of the form

{"**Input**", {*x*,*y*}, *value*, "*label*"}

{"**Input**", {*x*,*y*}, *value*, "*label*", *elementOpts*}

"**Input**" is the element name. Note that the word **Input** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value. It is a know stimulus (excitation).

"*label*" is a label associated to the element. Usually, the element label is a text string.

*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Input-element specification:

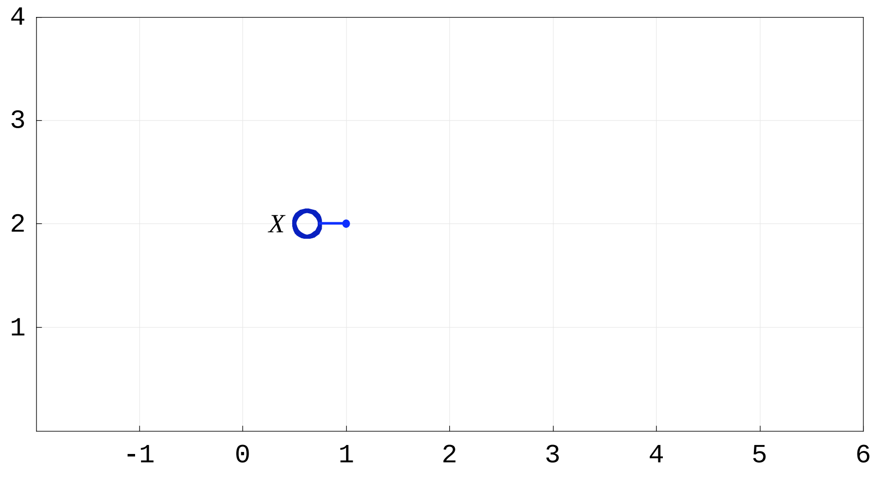
```
In[37]:=  myInputElement = {"Input", {1, 2}, X, "myInput"}

Out[37]=  {Input, {1, 2}, X, myInput}
```

We specify some options to show grid lines and frame for the examples of this section:

```
In[38]:=  SetOptions[ShowSchematic, Frame → True,
            GridLines → Automatic, PlotRange → {{-2, 6}, {0, 4}}];
```

*SchematicSolver* represents a system with single Input element as follows:

```
In[39]:=  {myInputElement} // ShowSchematic
```



In this example **`{1,2}`** are the element coordinates (see Figure above). **`X`** is the element value, and **`"myInput"`** is the element label that is not shown in the schematic.

## Output Element

A system takes one or more signals as input, performs operations on the signals, and produces one or more signals as *output*. The output is the actual response obtained from a system. It is described by a list of the form

{"**Output**", {*x*,*y*}, *value*, "*label*"}

{"**Output**", {*x*,*y*}, *value*, "*label*", *elementOpts*}

"**Output**" is the element name. Note that the word **Output** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value. Typically, it is the name of the output signal.

"*label*" is a label associated to the element. Usually, the element label is a text string.

*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

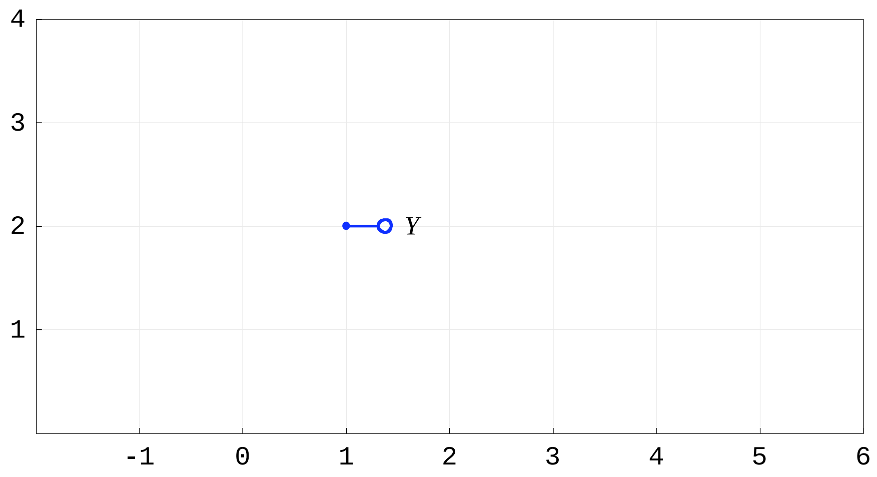
Here is an example of the Output-element specification:

```
In[40]:=  myOutputElement = {"Output", {1, 2}, Y, "myOutput"}
```

```
Out[40]=  {Output, {1, 2}, Y, myOutput}
```

*SchematicSolver* represents a system with single Output element as follows:

```
In[41]:=  {myOutputElement} // ShowSchematic
```



In this example **`{1,2}`** are the element coordinates (see Figure above). **`Y`** is the element value, and **`"myOutput"`** is the element label that is not shown in the schematic.

The circle that graphically represents Output element has a smaller radius than the circle that represents Input element.

## Multiplier Element

*Multiplier* of a discrete-time system is a single-input single-output block defined by the equation $y(n) = A\,x(n)$, where $A$ is the multiplier coefficient, $y(n)$ is the multiplier output, and $x(n)$ is the multiplier input. Multiplier is also referred to as *amplifier* or *gain*. It is described by a list of the form

{"**Multiplier**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*"}

{"**Multiplier**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*", *elementOpts*}

"**Multiplier**" is the element name. Note that the word **Multiplier** is enclosed within double quotation marks.

{{$x$1,$y$1}, {$x$2,$y$2}} are the element coordinates. {$x$1,$y$1} are the input coordinates and {$x$2,$y$2} are the output coordinates.

*value* is the element value. It is the multiplier coefficient, also called the multiplier constant or gain.

"*label*" is a label associated to the element. Usually, the element label is a text string.

*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

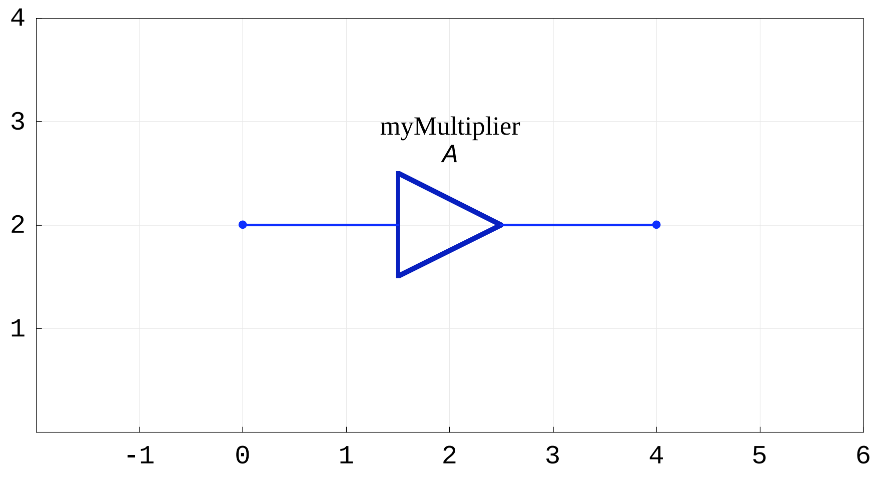
Here is an example of the Multiplier-element specification:

```
In[42]:=  myMultiplierElement = {"Multiplier", {{0, 2}, {4, 2}}, A, "myMultiplier"}
```

```
Out[42]=  {Multiplier, {{0, 2}, {4, 2}}, A, myMultiplier}
```

*SchematicSolver* represents a system with single Multiplier element as follows:

```
In[43]:=  {myMultiplierElement} // ShowSchematic
```



In this example **`{{0,2},{4,2}}`** are the element coordinates (see Figure above). **`A`** is the element value, and **`"myMultiplier"`** is the element label. The element input is at **`{0,2}`**, and the element output is at **`{4,2}`**.

## Delay Element

*Delay* of a discrete-time system is a single-input single-output block defined by the equation $y(n) = x(n - k)$, where $k$ is the number of delayed samples, $y(n)$ is the delay output, and $x(n)$ is the delay input. *Delay* with $k = 1$ is also referred to as the *unit delay*. It is described by a list of the form

{"**Delay**", {{$x$1,$y$1}, {$x$2,$y$2}}, *value*, "*label*"}

{"**Delay**", {{$x$1,$y$1}, {$x$2,$y$2}}, *value*, "*label*", *elementOpts*}

"**Delay**" is the element name. Note that the word **Delay** is enclosed within double quotation marks.

{{$x$1,$y$1}, {$x$2,$y$2}} are the element coordinates. {$x$1,$y$1} are the input coordinates and {$x$2,$y$2} are the output coordinates.

*value* is the element value. It is the number of delayed samples.

"*label*" is a label associated to the element. Usually, the element label is a text string.

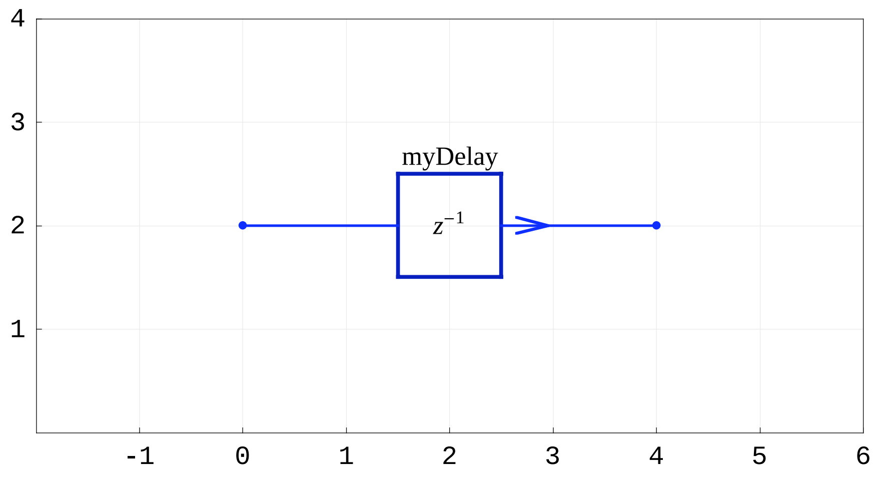
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Delay-element specification:

```
In[44]:=  myDelayElement = {"Delay", {{0, 2}, {4, 2}}, 1, "myDelay"}

Out[44]=  {Delay, {{0, 2}, {4, 2}}, 1, myDelay}
```

*SchematicSolver* represents a system with single Delay element as follows:

```
In[45]:=  {myDelayElement} // ShowSchematic
```



In this example **`{{0,2},{4,2}}`** are the element coordinates (see Figure above). **`1`** is the element value, and **`"myDelay"`** is the element label. The element input is at **`{0,2}`**, and the element output is at **`{4,2}`**. This example illustrates a unit delay.

## Adder Element

*Adder* performs the operations of addition and subtraction of signals. It is represented by a circle, with the appropriate minus sign associated with the lines entering the circle. *SchematicSolver*'s adder of a discrete-time system is a three-input single-output block defined by the equation $y(n) = P_1\, u_1(n) + P_2\, u_2(n) + P_3\, u_3(n)$, where $P$ is the sign parameter, $y(n)$ is the adder output, and $u(n)$ is the adder input. It is described by a list of the form

{"**Adder**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}}, {*p*1,*p*2,*p*3,*p*4}, "*label*"}

{"**Adder**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}}, {*p*1,*p*2,*p*3,*p*4}, "*label*", *elementOpts*}

"**Adder**" is the element name. Note that the word **Adder** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}} are the element coordinates. {*x*1,*y*1} are the coordinates of the left-hand node, {*x*3,*y*3} refer to the right-hand node, {*x*2,*y*2} correspond to the lower node, and {*x*4,*y*4} are the coordinates of the upper node.

{*p*1, *p*2, *p*3, *p*4} is the element value. It is the sign pattern of the element. The sign parameters *p*1, *p*2, *p*3, *p*4 can have an integer value of –1, 0, 1, or 2, and are interpreted as follows: 1 denotes the positive input (addition), –1 designates the negative input (subtraction), 2 designates the output, and 0 denotes the unused port. *p*1 corresponds to {*x*1,*y*1}, *p*2 corresponds to {*x*2,*y*2}, and so on.

"*label*" is a label associated to the element. Usually, the element label is a text string.

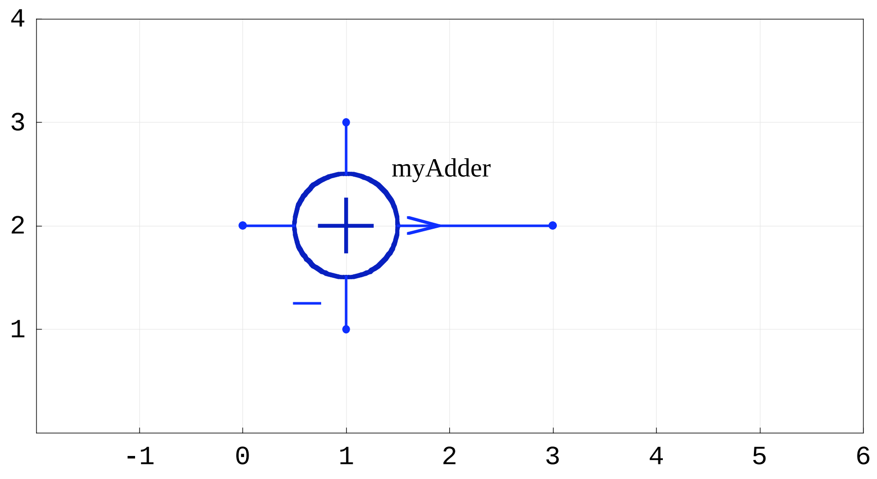
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Adder-element specification:

```
In[46]:=  myAdderElement = {"Adder", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 1}, "myAdder"}

Out[46]=  {Adder, {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 1}, myAdder}
```

*SchematicSolver* represents a system with single Adder element as follows:

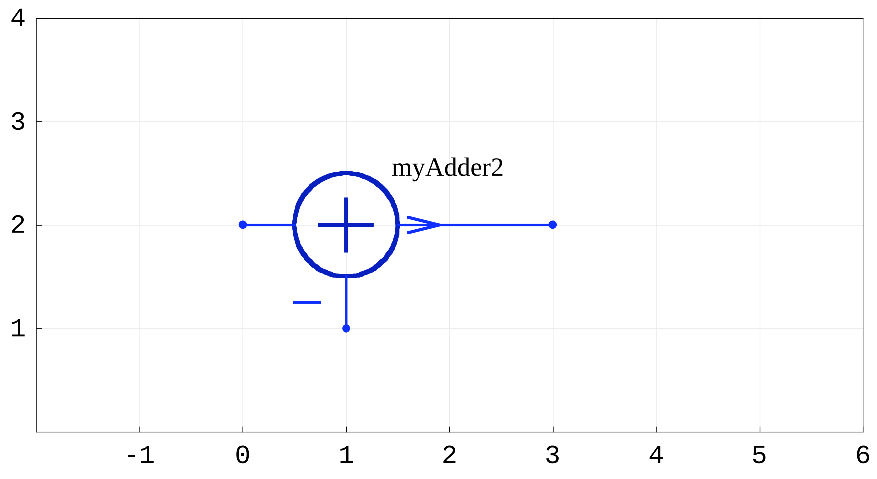
```
In[47]:=  {myAdderElement} // ShowSchematic
```



In this example **{{0,2},{1,1},{3,2},{1,3}}** are the element coordinates (see Figure above). **{1,-1,2,1}** is the element value, and **"myAdder"** is the element label. The element positive inputs are at **{0,2}** and **{1,3}**, the negative input is at **{1,1}**, and the element output is at **{3,2}**. This example illustrates a three-input adder.

An example of a two-input adder follows:

```
In[48]:=  myTwoportAdderElement =
           {"Adder", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 0}, "myAdder2"}

Out[48]=  {Adder, {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 0}, myAdder2}

In[49]:=  {myTwoportAdderElement} // ShowSchematic
```



Note that the unused port at **{1,3}** is not drawn.

## Block Element

*Block* of a discrete-time system is a single-input single-output block defined by the equation $Y(z) = H(z)\,X(z)$, where $H(z)$ is the block transfer function, $Y(z)$ is the block output in the $z$-transform domain, and $X(z)$ is the block input in the $z$-transform domain. Block is also referred to as *black box*. It is described by a list of the form

{"**Block**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*"}

{"**Block**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*", *elementOpts*}

"**Block**" is the element name. Note that the word **Block** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates. {*x*1,*y*1} are the input coordinates and {*x*2,*y*2} are the output coordinates.

*value* is the element value. It is the transfer function of the block.

"*label*" is a label associated to the element. Usually, the element label is a text string.

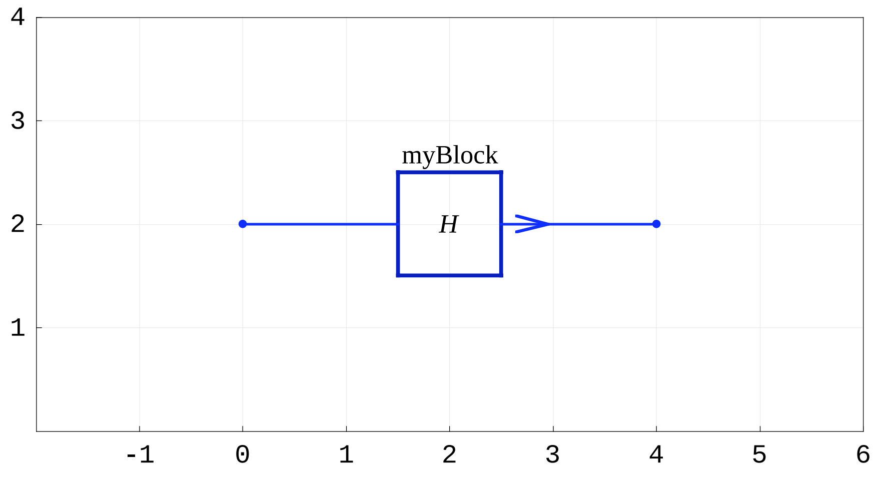
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Block-element specification:

```
In[50]:=  myBlockElement = {"Block", {{0, 2}, {4, 2}}, H, "myBlock"}

Out[50]=  {Block, {{0, 2}, {4, 2}}, H, myBlock}
```

*SchematicSolver* represents a system with single Block element as follows:

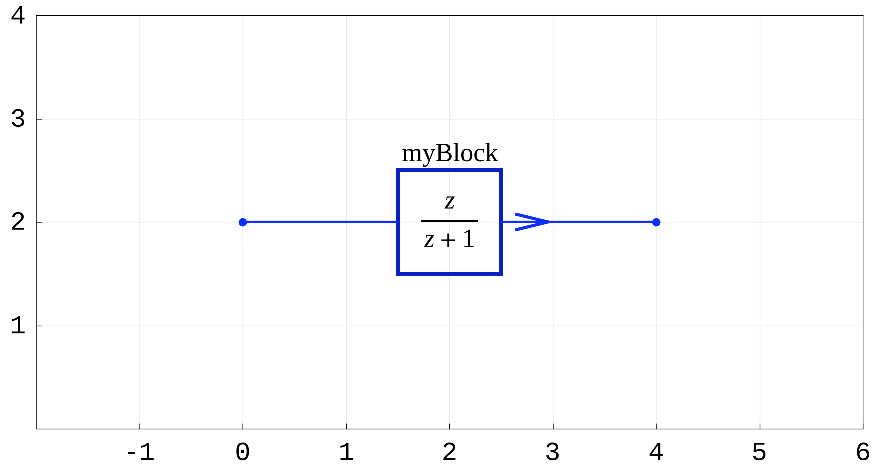
```
In[51]:=  {myBlockElement} // ShowSchematic
```



In this example **`{{0,2},{4,2}}`** are the element coordinates (see Figure above). **`H`** is the element value, and **`"my-Block"`** is the element label. The element input is at **`{0,2}`**, and the element output is at **`{4,2}`**.

The value can be a rational function in terms of the complex variable:

```
In[52]:=  {myBlockElement /. H → z / (z + 1)} // ShowSchematic
```



## Line Element

*Line* serves to connect nodes or element ports. In addition, line can implement takeoff points, and it permits the signal to proceed unaltered along the path specified by the line coordinates. It is described by a list of the form

{"**Line**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }}

{"**Line**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }, *elementOpts*}

"**Line**" is the element name. Note that the word **Line** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... } are the element coordinates. Line can have two or more coordinates. The first and the last coordinate pair represent the line nodes that connect to other nodes.

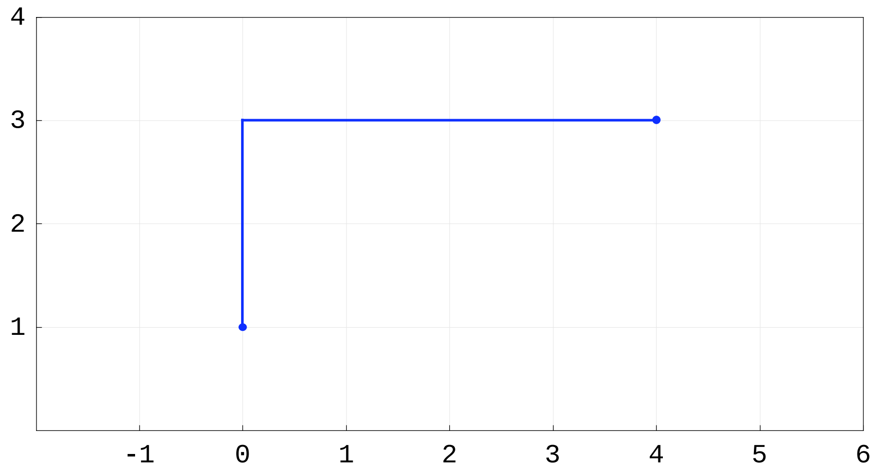
*elementOpts* are element options: `PlotStyle` and `ShowNodes`.

Here is an example of the Line-element specification:

```
In[53]:=  myLineElement = {"Line", {{0, 1}, {0, 3}, {4, 3}}}

Out[53]=  {Line, {{0, 1}, {0, 3}, {4, 3}}}
```

*SchematicSolver* represents a system with single Line element as follows:

```
In[54]:=  {myLineElement} // ShowSchematic
```



## Polyline Element

*Polyline* serves to annotate a schematic. It is described by a list of the form

{"**Polyline**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }}

{"**Polyline**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }, *elementOpts*}

"**Polyline**" is the element name. Note that the word **Polyline** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... } are the element coordinates. Polyline can have two or more coordinates.

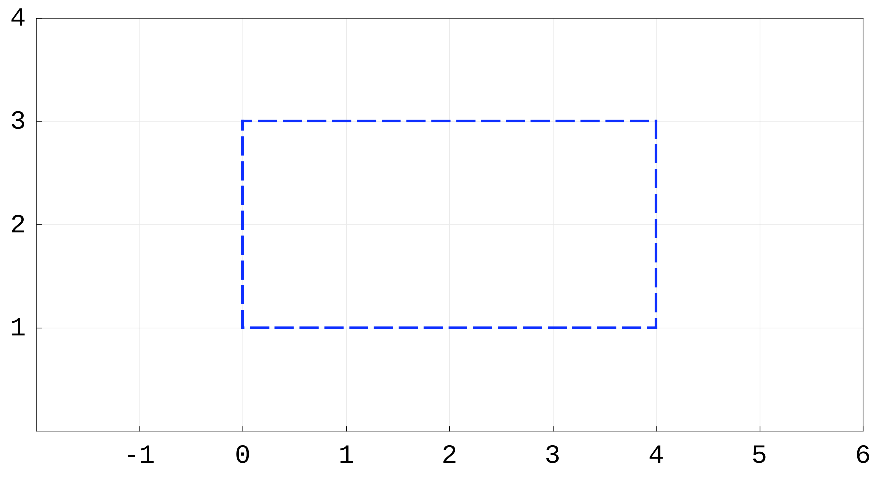
*elementOpts* are element options: `PlotStyle` and `PolylineDashing`.

Here is an example of the Polyline-element specification:

```
In[55]:=  myPolylineElement = {"Polyline", {{0, 1}, {0, 3}, {4, 3}, {4, 1}, {0, 1}}}

Out[55]=  {Polyline, {{0, 1}, {0, 3}, {4, 3}, {4, 1}, {0, 1}}}
```

*SchematicSolver* represents a system with single Polyline element as follows:

```
In[56]:=  {myPolylineElement} // ShowSchematic
```



By default, *SchematicSolver* draws polyline as a dashed line (see Figure above). Typically, polyline can be used to indicate a group of related elements.

## Node Element

*Node* serves to annotate a schematic. It is described by a list of the form

{"**Node**", {*x*,*y*}, *value*, "*label*"}

{"**Node**", {*x*,*y*}, *value*, "*label*", *elementOpts*}

"**Node**" is the element name. Note that the word **Node** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value.

"*label*" is a label associated to the element. Usually, the element label is a text string.

*elementOpts* are element options: `PlotStyle`, `TextOffset`, and `TextStyle`.

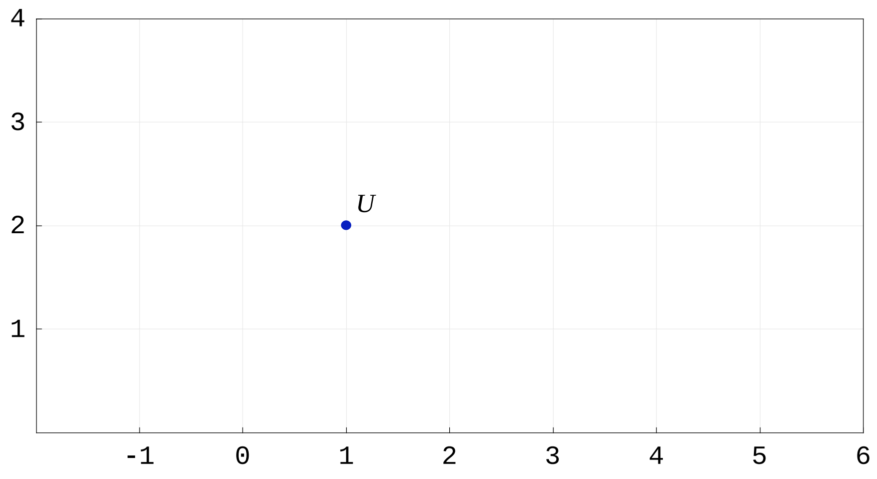
Here is an example of the Node-element specification:

```
In[57]:=  myNodeElement = {"Node", {1, 2}, U, "myNode"}
```

```
Out[57]=  {Node, {1, 2}, U, myNode}
```

*SchematicSolver* represents a system with single Node element as follows:

```
In[58]:=  {myNodeElement} // ShowSchematic
```



In this example **`{1,2}`** are the element coordinates (see Figure above). **`U`** is the element value, and **`"myNode"`** is the element label that is not shown in the schematic.

Node can be used to indicate signals at the schematic nodes. In addition, nodes are used to emphasize the points at which two or more element nodes are connected.

## Text Element

*Text* serves to annotate a schematic. It is described by a list of the form

{"**Text**", {*x*,*y*}, *value*}

{"**Text**", {*x*,*y*}, *value*, *elementOpts*}

"**Text**" is the element name. Note that the word **Text** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value. Usually, the element value is a text string.

*elementOpts* are element options: `TextDirection`, `TextOffset`, and `TextStyle`.

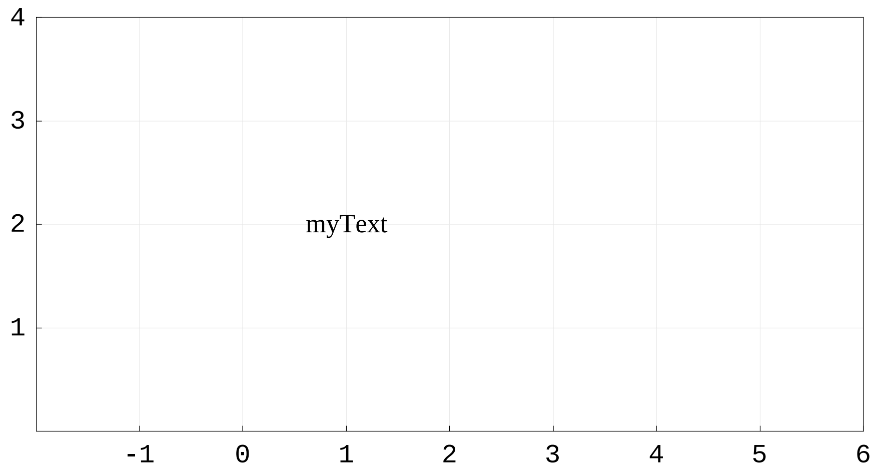
Here is an example of the Text-element specification:

```
In[59]:=  myTextElement = {"Text", {1, 2}, "myText"}
```

```
Out[59]=  {Text, {1, 2}, myText}
```

*SchematicSolver* represents a system with single Text element as follows:

```
In[60]:=  {myTextElement} // ShowSchematic
```



In this example **`{1,2}`** are the element coordinates (see Figure above). **`"myText"`** is the element value.

Note that, by default, the text value is centered around the coordinates.

## Arrow Element

*Arrow* serves to annotate direction of signal paths along lines. It is described by a list of the form

{"**Arrow**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*}

{"**Arrow**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, *elementOpts*}

"**Arrow**" is the element name. Note that the word **Arrow** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates.

*value* is the element value. Usually, the value is a text string.

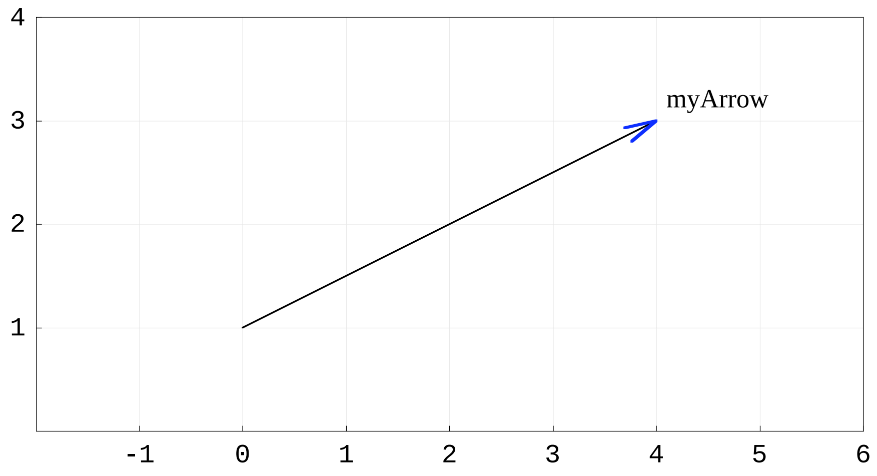
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowArrowTail`, `TextOffset`, and `Text-Style`.

Here is an example of the Arrow-element specification:

```
In[61]:=  myArrowElement = {"Arrow", {{4, 3}, {0, 1}}, "myArrow"}

Out[61]=  {Arrow, {{4, 3}, {0, 1}}, myArrow}
```

*SchematicSolver* represents a system with single Arrow element as follows:

```
In[62]:= {myArrowElement} // ShowSchematic
```



Note that the arrowhead is drawn at the first coordinate pair specified, in this example at **{4,3}**.

## 3.5. Nonlinear Discrete Elements

### Introduction

*SchematicSolver* supports two nonlinear discrete elements, in addition to the previously described discrete elements.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[63]:= Needs["SchematicSolver`"];
```

We specify some options to show grid lines and frame for the examples of this section:

```
In[64]:= SetOptions[ShowSchematic, Frame → True,
           GridLines → Automatic, PlotRange → {{-2, 6}, {0, 4}}];
```

### Function Element

*Function* of a discrete-time system is a single-input single-output block defined by the equation $y = F(x)$, where $F$ is the block function, $y$ is the block output, and $x$ is the block input. It is described by a list of the form

{"**Function**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*"}

{"**Function**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*", *elementOpts*}

"**Function**" is the element name. Note that the word **Function** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates. {*x*1,*y*1} are the input coordinates and {*x*2,*y*2} are the output coordinates.

*value* is the element value. It is a symbol that represents the name of a built-in or user-defined algebraic function of one argument.

"*label*" is a label associated to the element. Usually, the label is a text string.

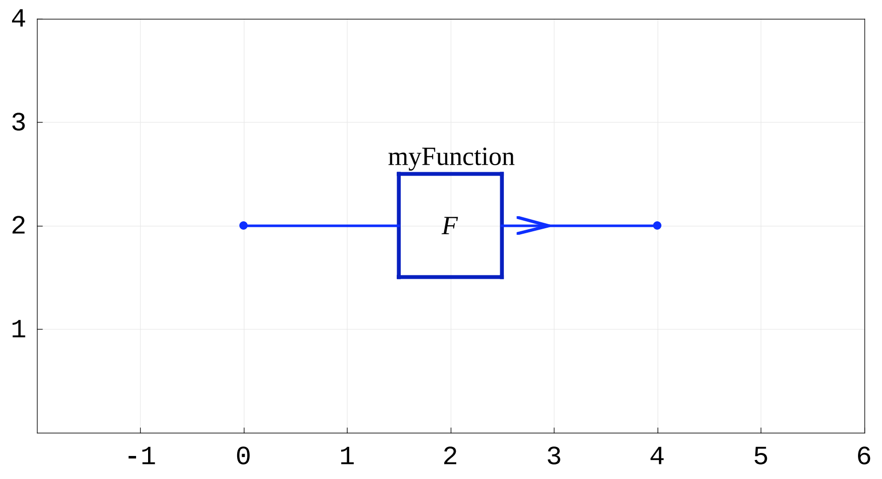
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Function-element specification:

```
In[65]:= myFunctionElement = {"Function", {{0, 2}, {4, 2}}, F, "myFunction"}

Out[65]= {Function, {{0, 2}, {4, 2}}, F, myFunction}
```

*SchematicSolver* represents a system with single Function element as follows:

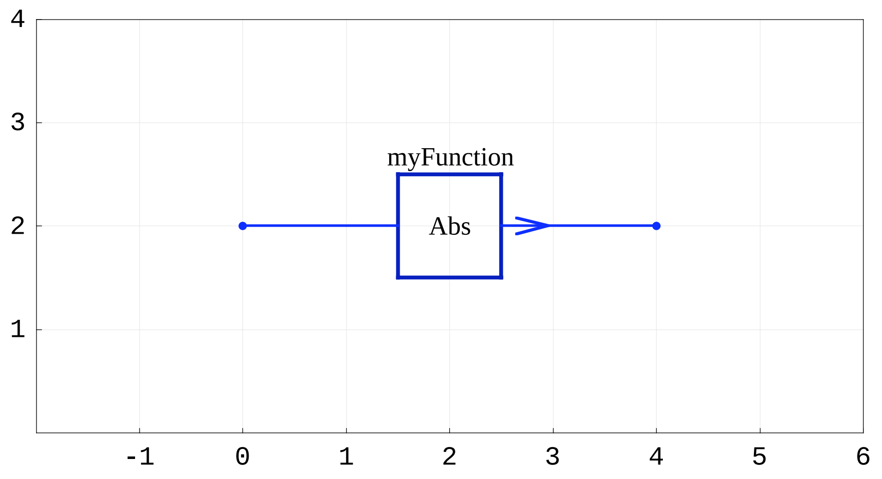
```
In[66]:= {myFunctionElement} // ShowSchematic
```



In this example **`{{0,2},{4,2}}`** are the element coordinates (see Figure above). **`F`** is the element value, and **`"myFunction"`** is the element label. The element input is at **`{0,2}`** and the element output is at **`{4,2}`**.

The Function-element value can be an arbitrary built-in algebraic function of one argument:

```
In[67]:= {myFunctionElement /. F → Abs} // ShowSchematic
```



## Modulator Element

*Modulator* performs the operation of multiplication of signals. It is represented by a circle. *SchematicSolver*'s modulator of a discrete-time system is a three-input single-output block defined by the equation $y(n) = u_1(n)\,u_2(n)\,u_3(n)$, where $y(n)$ is the modulator output and $u(n)$ is the modulator input. It is described by a list of the form

{"**Modulator**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}}, {*p*1,*p*2,*p*3,*p*4}, "*label*"}

{"**Modulator**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}}, {*p*1,*p*2,*p*3,*p*4}, "*label*", *elementOpts*}

"**Modulator**" is the element name. Note that the word **Modulator** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}} are the element coordinates. {*x*1,*y*1} are the coordinates of the left-hand node, {*x*3,*y*3} refer to the right-hand node, {*x*2,*y*2} correspond to the lower node, and {*x*4,*y*4} are the coordinates of the upper node.

{*p*1,*p*2,*p*3,*p*4} is the element value. The parameters *p*1, *p*2, *p*3, *p*4 can have an integer value of 0, 1, or 2, and are interpreted as follows: 1 denotes the input, 2 designates the output, and 0 denotes the unused port. *p*1 corresponds to {*x*1,*y*1}, *p*2 corresponds to {*x*2,*y*2}, and so on.

"*label*" is a label associated to the element. Usually, the label is a text string.

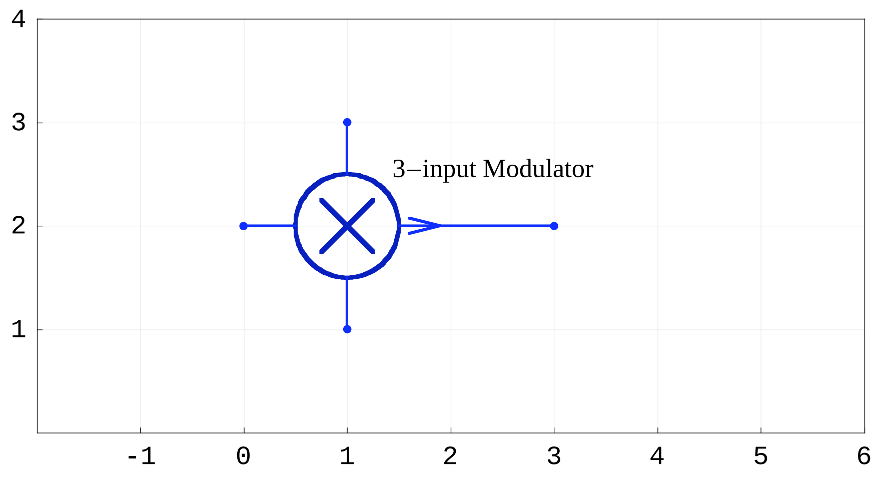
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Modulator-element specification:

```
In[68]:= myModulatorElement =
          {"Modulator", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, 1, 2, 1}, "3-input Modulator"}

Out[68]= {Modulator, {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, 1, 2, 1}, 3-input Modulator}
```

*SchematicSolver* represents a system with single Modulator element as follows:

```
In[69]:= {myModulatorElement} // ShowSchematic
```

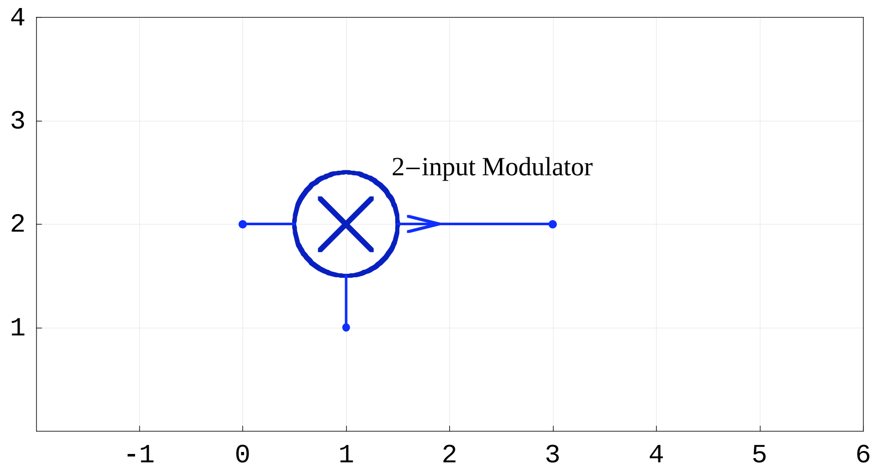


In this example **`{{0,2},{1,1},{3,2},{1,3}}`** are the element coordinates (see Figure above). **`{1,1,2,1}`** is the element value, and **`"3-input Modulator"`** is the element label. The element inputs are at **`{0,2}`**, **`{1,1}`**, and **`{1,3}`**. The element output is at **`{3,2}`**. This example illustrates a three-input modulator. The output is the product of the three inputs.

An example of a two-input modulator follows:

```
In[70]:= myTwoInputModulatorElement =
          {"Modulator", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, 1, 2, 0}, "2-input Modulator"}

Out[70]= {Modulator, {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, 1, 2, 0}, 2-input Modulator}
```

```
In[71]:=  {myTwoInputModulatorElement} // ShowSchematic
```



Note that the unused port at **`{1,3}`** is not drawn. In this case, the output is the product of the two inputs.

## 3.6. Continuous-Time Elements

### Introduction

*SchematicSolver* describes a system as a list of elements; this list specifies what elements are in the system and how they are interconnected. A list describing a system will be referred to as the *system specification*.

Each element in the system is also described as a list that states what the element is, to which other elements it is connected, and what its value is. A list describing an element will be referred to as the *element specification*.

The junction points between elements are referred to as *nodes*.

*SchematicSolver* supports various continuous-time elements that can be used to describe a continuous-time system or an analog system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[72]:=  Needs["SchematicSolver`"];
```

We specify some options to show grid lines and frame for the examples of this section:

```
In[73]:=  SetOptions[ShowSchematic, Frame → True,
            GridLines → Automatic, PlotRange → {{-2, 6}, {0, 4}}];
```

### Input Element

*Input* is the stimulus or excitation applied to a system from an external source. It is described by a list of the form

{"**Input**", {*x*,*y*}, *value*, "*label*"}

{"**Input**", {*x*,*y*}, *value*, "*label*", *elementOpts*}

"**Input**" is the element name. Note that the word **Input** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value. It is a know stimulus (excitation).

"*label*" is a label associated to the element. Usually, the label is a text string.

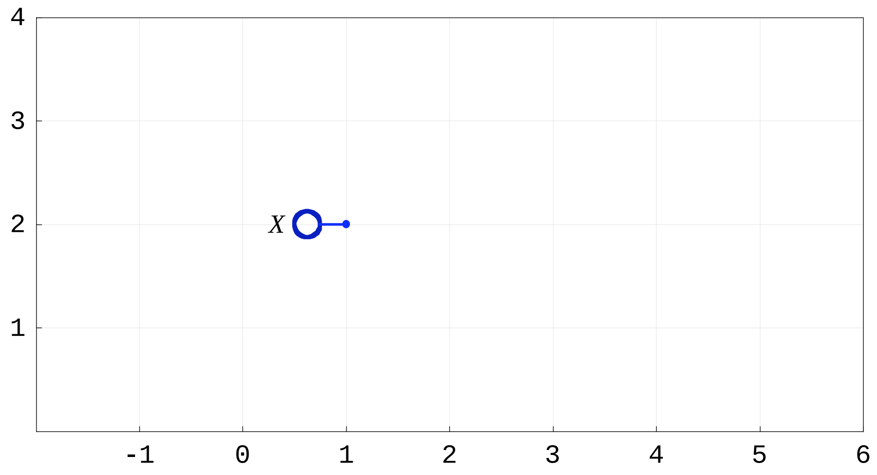
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Input-element specification:

```
In[74]:=  myInputElement = {"Input", {1, 2}, X, "myInput"}

Out[74]=  {Input, {1, 2}, X, myInput}
```

*SchematicSolver* represents a system with single Input element as follows:

```
In[75]:=  {myInputElement} // ShowSchematic
```



In this example **`{1,2}`** are the element coordinates (see Figure above). **`X`** is the element value, and **`"myInput"`** is the element label that is not shown in the schematic.

## Output Element

A system takes one or more signals as input, performs operations on the signals, and produces one or more signals as *output*. The output is the actual response obtained from a system. It is described by a list of the form

{"**Output**", {*x*,*y*}, *value*, "*label*"}

{"**Output**", {*x*,*y*}, *value*, "*label*", *elementOpts*}

"**Output**" is the element name. Note that the word **Output** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value. Typically, it is the name of the output signal.

"*label*" is a label associated to the element. Usually, the label is a text string.

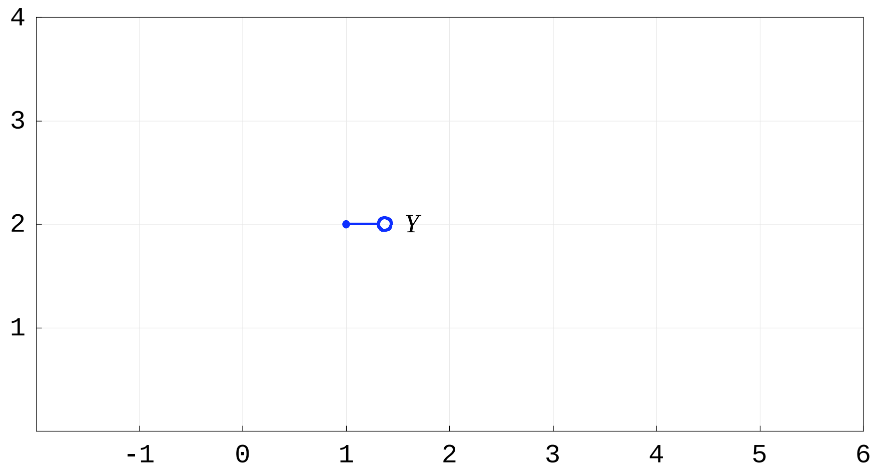
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Output-element specification:

```
In[76]:=  myOutputElement = {"Output", {1, 2}, Y, "myOutput"}

Out[76]=  {Output, {1, 2}, Y, myOutput}
```

*SchematicSolver* represents a system with single Output element as follows:

```
In[77]:=  {myOutputElement} // ShowSchematic
```



In this example **`{1,2}`** are the element coordinates (see Figure above). **`Y`** is the element value, and **`"myOutput"`** is the element label that is not shown in the schematic.

The circle that graphically represents Output element has a smaller radius than the circle that represents Input element.

## Amplifier Element

*Amplifier* of a continuous-time system is a single-input single-output block defined by the equation $y(t) = A\,x(t)$, where $A$ is the amplifier gain, $y(t)$ is the amplifier output, and $x(t)$ is the amplifier input. It is described by a list of the form

{"**Amplifier**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*"}

{"**Amplifier**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*", *elementOpts*}

"**Amplifier**" is the element name. Note that the word **Amplifier** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates. {*x*1,*y*1} are the input coordinates and {*x*2,*y*2} are the output coordinates.

*value* is the element value. It is the amplifier gain.

"*label*" is a label associated to the element. Usually, the label is a text string.

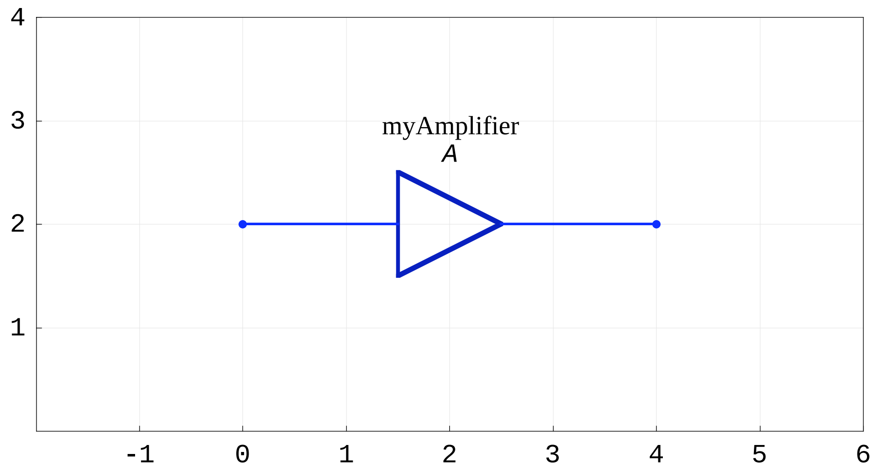
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Amplifier-element specification:

```
In[78]:=  myAmplifierElement = {"Amplifier", {{0, 2}, {4, 2}}, A, "myAmplifier"}

Out[78]=  {Amplifier, {{0, 2}, {4, 2}}, A, myAmplifier}
```

*SchematicSolver* represents a system with single Amplifier element as follows:

```
In[79]:=  {myAmplifierElement} // ShowSchematic
```



In this example **{{0,2},{4,2}}** are the element coordinates (see Figure above). **A** is the element value, and **"myAmplifier"** is the element label. The element input is at **{0,2}**, and the element output is at **{4,2}**.

## Integrator Element

*Integrator* of a continuous-time system is a single-input single-output block defined by the equation $y(t) = y(0) + K \int_0^t x(t)\, dt$, where $K$ is the integrator gain, $y(t)$ is the integrator output, $y(0)$ is the initial condition, and $x(t)$ is the integrator input. It is described by a list of the form

{"**Integrator**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*"}

{"**Integrator**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*", *elementOpts*}

"**Integrator**" is the element name. Note that the word **Integrator** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates. {*x*1,*y*1} are the input coordinates and {*x*2,*y*2} are the output coordinates.

*value* is the element value. *value* can be a pair of the form {*gain*, *initialCondition*}, or it can be an expression representing the gain (assuming zero initial condition).

"*label*" is a label associated to the element. Usually, the label is a text string.

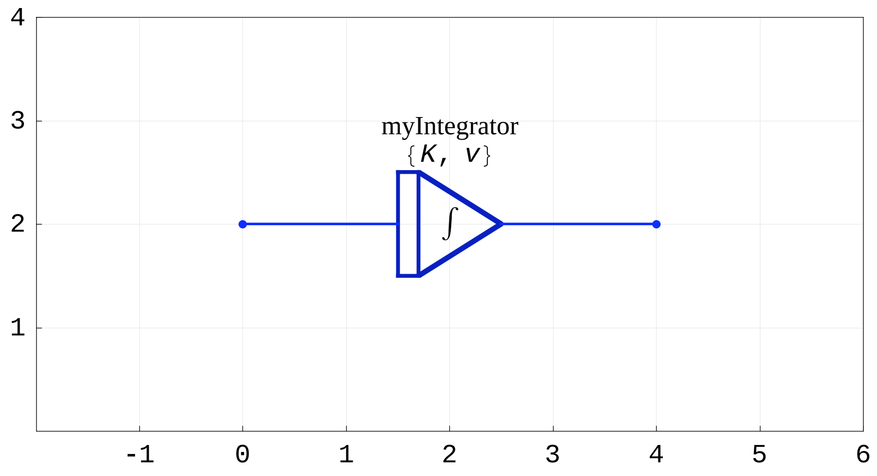
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Integrator-element specification:

```
In[80]:=  myIntegratorElement = {"Integrator", {{0, 2}, {4, 2}}, {K, v}, "myIntegrator"}

Out[80]=  {Integrator, {{0, 2}, {4, 2}}, {K, v}, myIntegrator}
```

*SchematicSolver* represents a system with single Integrator element as follows:

```
In[81]:=  {myIntegratorElement} // ShowSchematic
```



In this example **`{{0,2},{4,2}}`** are the element coordinates (see Figure above). **`{K,v}`** is the element value, and **`"myIntegrator"`** is the element label. The element input is at **`{0,2}`** and the element output is at **`{4,2}`**. **`K`** is the gain and **`v`** is the initial condition.

## Adder Element

*Adder* performs the operations of addition and subtraction of signals. It is represented by a circle, with the appropriate minus sign associated with the lines entering the circle. *SchematicSolver*'s adder of a continuous-time system is a three-input single-output block defined by the equation $y(t) = P_1\, u_1(t) + P_2\, u_2(t) + P_3\, u_3(t)$, where $P$ is the sign parameter, $y(t)$ is the adder output, and $u(t)$ is the adder input. It is described by a list of the form

{"**Adder**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}}, {*p*1,*p*2,*p*3,*p*4}, "*label*"}

{"**Adder**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}}, {*p*1,*p*2,*p*3,*p*4}, "*label*", *elementOpts*}

"**Adder**" is the element name. Note that the word **Adder** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, {*x*4,*y*4}} are the element coordinates. {*x*1,*y*1} are the coordinates of the left-hand node, {*x*3,*y*3} refer to the right-hand node, {*x*2,*y*2} correspond to the lower node, and {*x*4,*y*4} are the coordinates of the upper node.

{*p*1, *p*2, *p*3, *p*4} is the element value. It is the sign pattern of the element. The sign parameters *p*1, *p*2, *p*3, *p*4 can have an integer value of –1, 0, 1, or 2, and are interpreted as follows: 1 denotes the positive input (addition), –1 designates the negative input (subtraction), 2 designates the output, and 0 denotes the unused port. *p*1 corresponds to {*x*1,*y*1}, *p*2 corresponds to {*x*2,*y*2}, and so on.

"*label*" is a label associated to the element. Usually, the label is a text string.

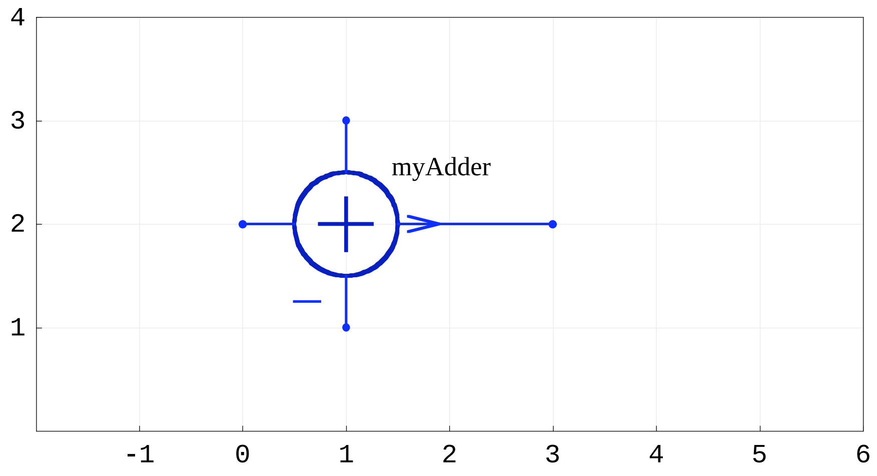
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Adder-element specification:

```
In[82]:=  myAdderElement = {"Adder", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 1}, "myAdder"}

Out[82]=  {Adder, {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 1}, myAdder}
```

*SchematicSolver* represents a system with single Adder element as follows:

```
In[83]:= {myAdderElement} // ShowSchematic
```

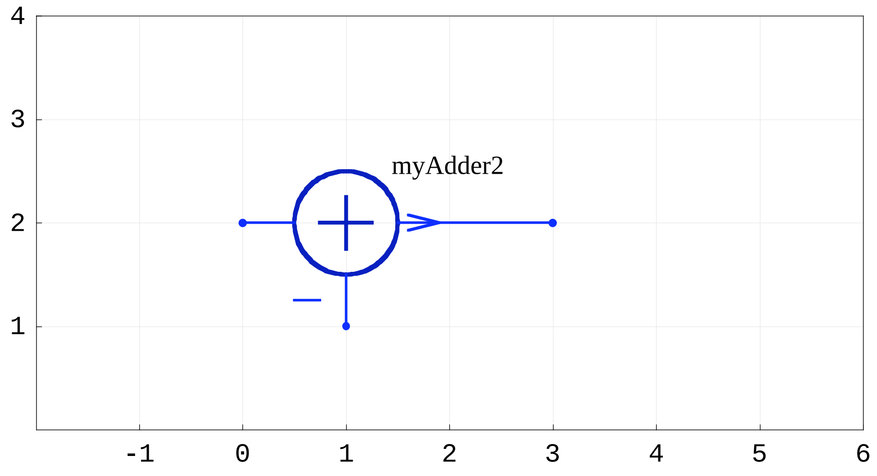


In this example **{{0,2},{1,1},{3,2},{1,3}}** are the element coordinates (see Figure above). **{1,-1,2,1}** is the element value, and **"myAdder"** is the element label. The element positive inputs are at **{0,2}** and **{1,3}**, the negative input is at **{1,1}**, and the element output is at **{3,2}**. This example illustrates a three-input adder.

An example of a two-input adder follows:

```
In[84]:= myTwoportAdderElement =
           {"Adder", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 0}, "myAdder2"}

Out[84]= {Adder, {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 0}, myAdder2}
```

```
In[85]:= {myTwoportAdderElement} // ShowSchematic
```



Note that the unused port at **{1,3}** is not drawn.

## Block Element

*Block* of a continuous-time system is a single-input single-output block defined by the equation $Y(s) = H(s)\, X(s)$, where $H(s)$ is the block transfer function, $Y(s)$ is the block output in the *Laplace*-transform domain, and $X(s)$ is the block input in the *Laplace*-transform domain. Block is also referred to as *black box*. It is described by a list of the form

{"**Block**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*"}

{"**Block**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, "*label*", *elementOpts*}

"**Block**" is the element name. Note that the word **Block** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates. {*x*1,*y*1} are the input coordinates and {*x*2,*y*2} are the output coordinates.

*value* is the element value. It is the transfer function of the block.

"*label*" is a label associated to the element. Usually, the label is a text string.

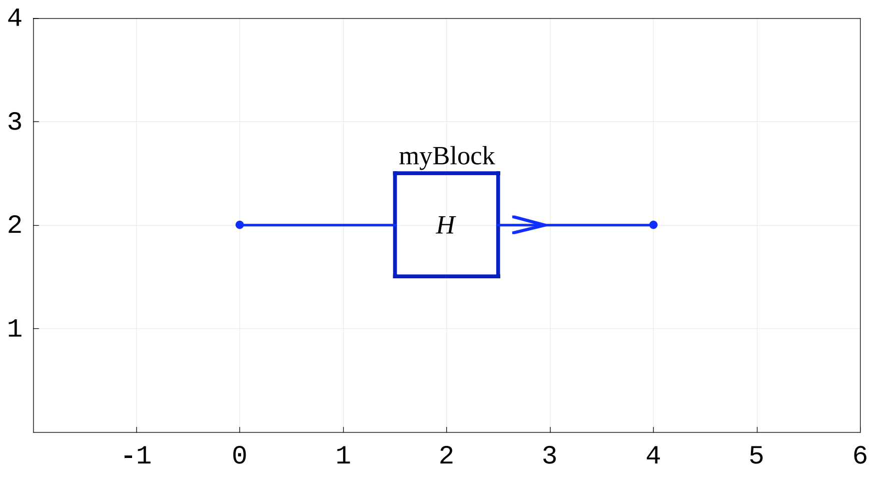
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowNodes`, `TextOffset`, and `TextStyle`.

Here is an example of the Block-element specification:

```
In[86]:=  myBlockElement = {"Block", {{0, 2}, {4, 2}}, H, "myBlock"}

Out[86]=  {Block, {{0, 2}, {4, 2}}, H, myBlock}
```

*SchematicSolver* represents a system with single Block element as follows:

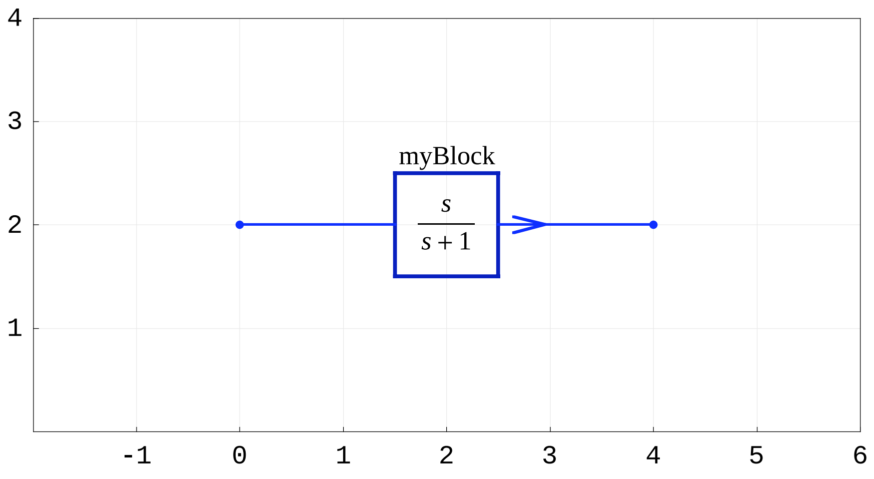
```
In[87]:=  {myBlockElement} // ShowSchematic
```



In this example **`{{0,2},{4,2}}`** are the element coordinates (see Figure above). **`H`** is the element value, and **`"myBlock"`** is the element label. The element input is at **`{0,2}`** and the element output is at **`{4,2}`**.

The value can be a rational function in terms of the complex variable:

```
In[88]:=  {myBlockElement /. H → s / (s + 1)} // ShowSchematic
```



## Line Element

*Line* serves to connect nodes or element ports. In addition, line can implement takeoff points, and it permits the signal to proceed unaltered along the path specified by the line coordinates. It is described by a list of the form

{"**Line**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }}

{"**Line**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }, *elementOpts*}

"**Line**" is the element name. Note that the word **Line** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... } are the element coordinates. Line can have two or more coordinates. The first and the last coordinate pair represent the line nodes that connect to other nodes.

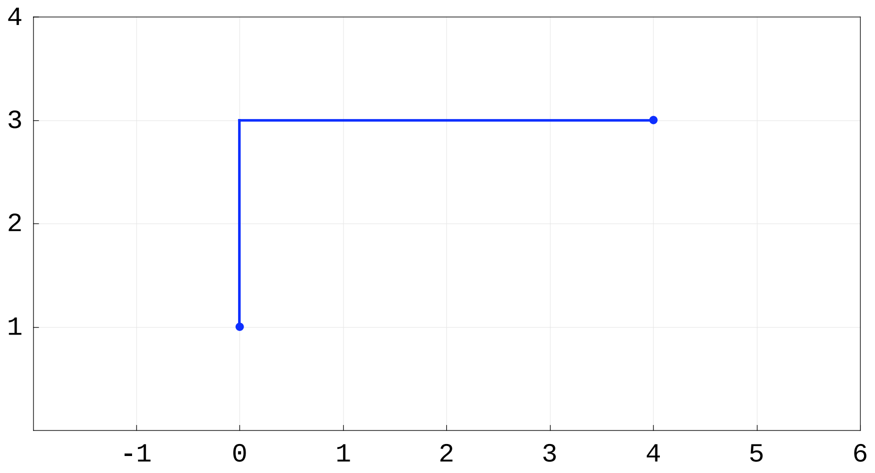
*elementOpts* are element options: `PlotStyle` and `ShowNodes`.

Here is an example of the Line-element specification:

```
In[89]:=  myLineElement = {"Line", {{0, 1}, {0, 3}, {4, 3}}}

Out[89]=  {Line, {{0, 1}, {0, 3}, {4, 3}}}
```

*SchematicSolver* represents a system with single Line element as follows:

```
In[90]:=  {myLineElement} // ShowSchematic
```



## Polyline Element

*Polyline* serves to annotate a schematic. It is described by a list of the form

{"**Polyline**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }}

{"**Polyline**", {{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... }, *elementOpts*}

"**Polyline**" is the element name. Note that the word **Polyline** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}, {*x*3,*y*3}, ... } are the element coordinates. Polyline can have two or more coordinates.

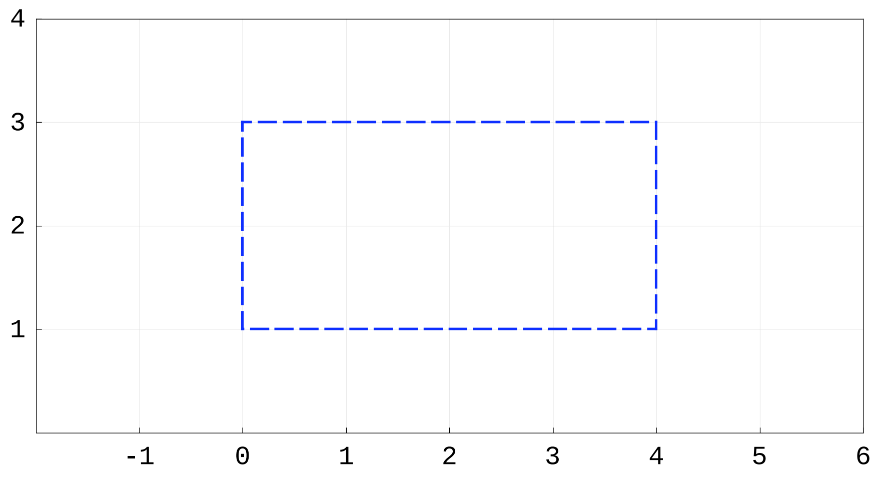
*elementOpts* are element options: `PlotStyle` and `PolylineDashing`.

Here is an example of the Polyline-element specification:

```
In[91]:=  myPolylineElement = {"Polyline", {{0, 1}, {0, 3}, {4, 3}, {4, 1}, {0, 1}}}

Out[91]=  {Polyline, {{0, 1}, {0, 3}, {4, 3}, {4, 1}, {0, 1}}}
```

*SchematicSolver* represents a system with single Polyline element as follows:

```
In[92]:=  {myPolylineElement} // ShowSchematic
```



By default, *SchematicSolver* draws polyline as a dashed line (see Figure above). Typically, polyline can be used to indicate a group of related elements.

## Node Element

*Node* serves to annotate a schematic. It is described by a list of the form

{"**Node**", {*x*,*y*}, *value*, "*label*"}

{"**Node**", {*x*,*y*}, *value*, "*label*", *elementOpts*}

"**Node**" is the element name. Note that the word **Node** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value.

"*label*" is a label associated to the element. Usually, the label is a text string.

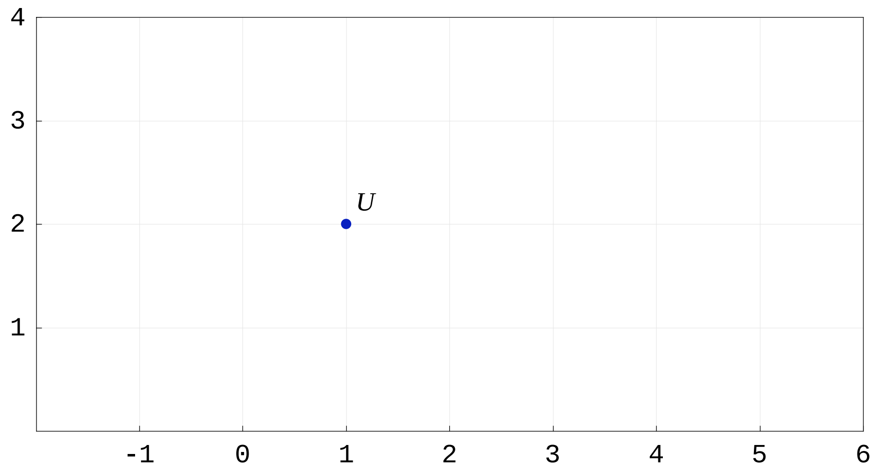
*elementOpts* are element options: `PlotStyle`, `TextOffset`, and `TextStyle`.

Here is an example of the Node-element specification:

```
In[93]:=  myNodeElement = {"Node", {1, 2}, U, "myNode"}

Out[93]=  {Node, {1, 2}, U, myNode}
```

*SchematicSolver* represents a system with single Node element as follows:

```
In[94]:= {myNodeElement} // ShowSchematic
```



In this example **{1,2}** are the element coordinates (see Figure above). **U** is the element value, and **"myNode"** is the element label that is not shown in the schematic.

Node can be used to indicate signals at the schematic nodes. In addition, nodes are used to emphasize the points at which two or more element nodes are connected.

## Text Element

*Text* serves to annotate a schematic. It is described by a list of the form

{"**Text**", {*x*,*y*}, *value*}

{"**Text**", {*x*,*y*}, *value*, *elementOpts*}

"**Text**" is the element name. Note that the word **Text** is enclosed within double quotation marks.

{*x*,*y*} are the element coordinates.

*value* is the element value. Usually, the value is a text string.

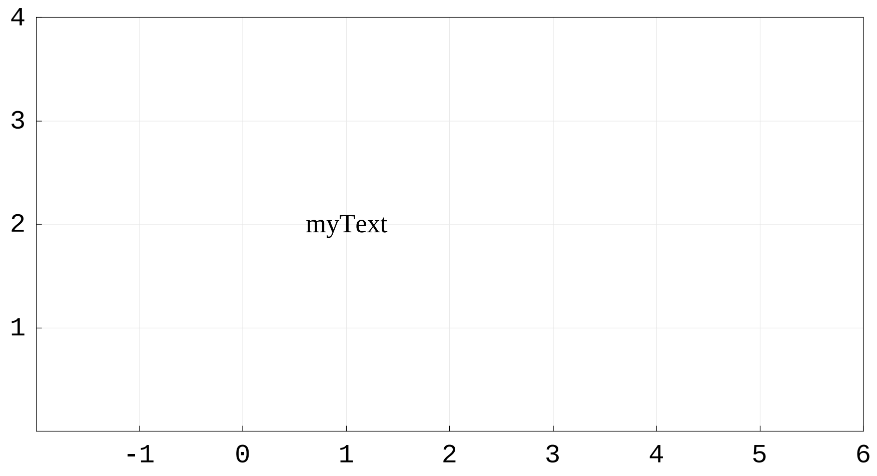
*elementOpts* are element options: `TextDirection`, `TextOffset`, and `TextStyle`.

Here is an example of the Text-element specification:

```
In[95]:= myTextElement = {"Text", {1, 2}, "myText"}

Out[95]= {Text, {1, 2}, myText}
```

*SchematicSolver* represents a system with single Text element as follows:

```
In[96]:=  {myTextElement} // ShowSchematic
```



In this example `{1,2}` are the element coordinates (see Figure above). `"myText"` is the element value.

Note that, by default, the text value is centered around the coordinates.

## Arrow Element

*Arrow* serves to annotate direction of signal paths along lines. It is described by a list of the form

{"**Arrow**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*}

{"**Arrow**", {{*x*1,*y*1}, {*x*2,*y*2}}, *value*, *elementOpts*}

"**Arrow**" is the element name. Note that the word **Arrow** is enclosed within double quotation marks.

{{*x*1,*y*1}, {*x*2,*y*2}} are the element coordinates.

*value* is the element value. Usually, the value is a text string.

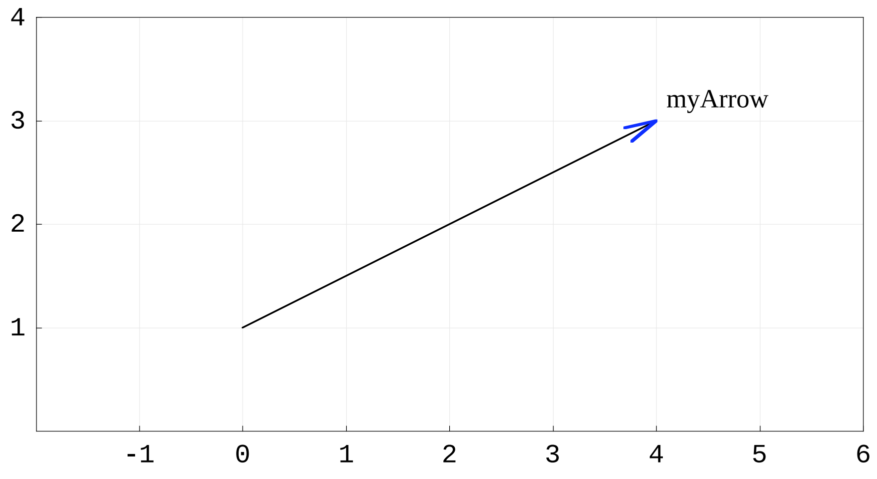
*elementOpts* are element options: `ElementSize`, `PlotStyle`, `ShowArrowTail`, `TextOffset`, and `Text-Style`.

Here is an example of the Arrow-element specification:

```
In[97]:=  myArrowElement = {"Arrow", {{4, 3}, {0, 1}}, "myArrow"}

Out[97]=  {Arrow, {{4, 3}, {0, 1}}, myArrow}
```

*SchematicSolver* represents a system with single Arrow element as follows:

```
In[98]:=  {myArrowElement} // ShowSchematic
```



Note that the arrowhead is drawn at the first coordinate pair specified, in this example at **`{4,3}`**.

## 3.7. Drawing Options for Elements

### Introduction

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[99]:=  Needs["SchematicSolver`"];
```

*SchematicSolver* can draw elements in different colors and sizes by means of *element options*. The following options are available:

```
In[100]:=
     Options[DrawElement]

Out[100]=
     {ElementSize → {1, 1},
      PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
        {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
      ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
      TextStyle → {FontFamily → Times, FontSize → 10}}
```

A detailed description and examples for each option follow.

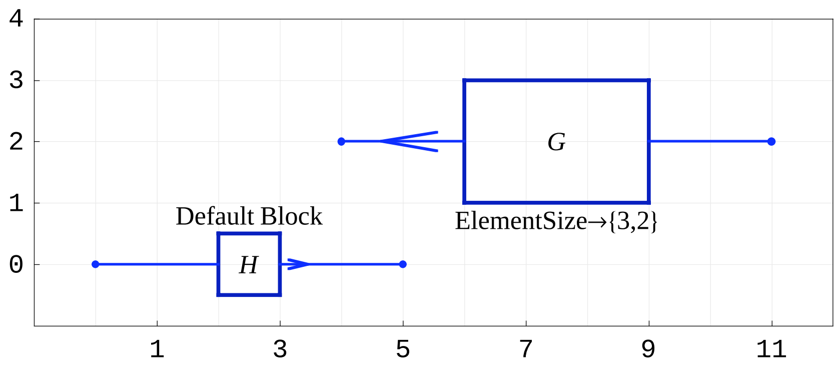
We specify some options to show grid lines and frame for the examples of this section:

```
In[101]:=
     SetOptions[ShowSchematic, Frame → True,
       GridLines → Automatic, PlotRange → {{-1, 12}, {-1, 4}}];
```

### `ElementSize`

`ElementSize` is an option that specifies the size and aspect ratio of a schematic element.

```
In[102]:=
     myDefaultBlock = {"Block", {{0, 0}, {5, 0}}, H, "Default Block"};
     myLargeBlock = {"Block", {{11, 2}, {4, 2}}, G, "ElementSize→{3,2}",
        ElementSize → {3, 2}};
     {myDefaultBlock, myLargeBlock} // ShowSchematic
```

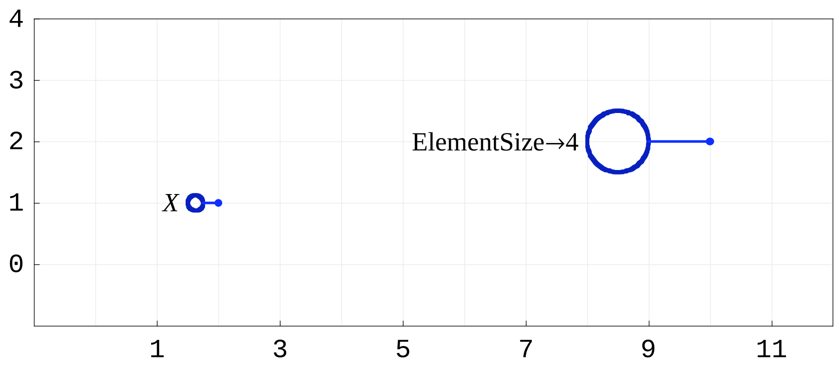


In above Figure, **ElementSize→{3,2}** specifies a block whose width is three times larger than the default width, and whose height is two times larger that the default height.

Default value for `ElementSize` is **ElementSize→{1,1}**.

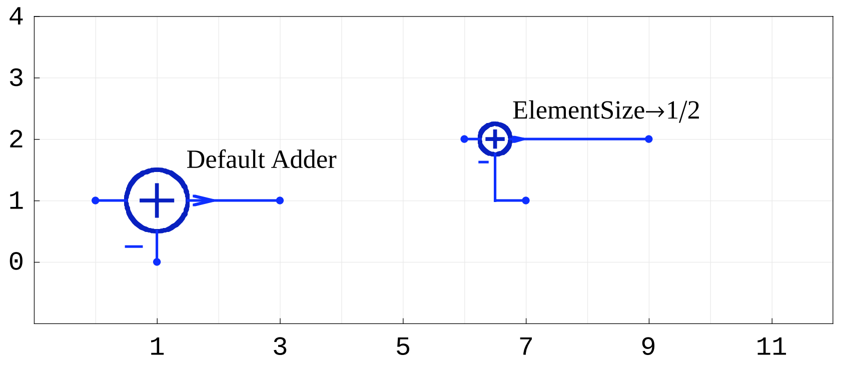
The following example changes the radius of the circle that represents the Input element. The scaled element is four times larger than the default one: **ElementSize→4**.

```
In[105]:=
     myDefaultInput = {"Input", {2, 1}, X};
     myLargeInput = {"Input", {10, 2}, "ElementSize→4", " ",
        ElementSize → 4};
     {myDefaultInput, myLargeInput} // ShowSchematic
```



Here is an example of an adder of reduced size; its radius is two times smaller than the radius of the default element: **ElementSize→1/2**.

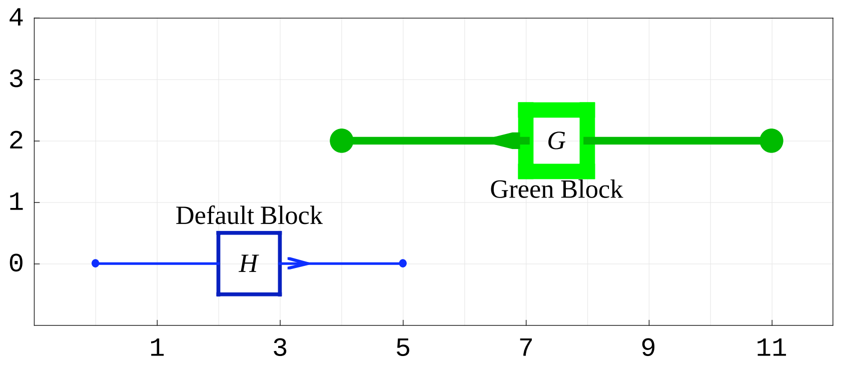
```
In[108]:=
      myDefaultAdder =
        {"Adder", {{0, 1}, {1, 0}, {3, 1}, {1, 2}}, {1, -1, 2, 0}, "Default Adder"};
      mySmallAdder = {"Adder", {{6, 2}, {7, 1}, {9, 2}, {7, 3}},
         {1, -1, 2, 0}, "ElementSize→1/2",
         ElementSize → 1 / 2};
      {myDefaultAdder, mySmallAdder} // ShowSchematic
```



## PlotStyle

`PlotStyle` specifies color, line thickness and point size of elements. Two specifications are given: one for the element shape (graphic symbol), and one for the element ports (lines connecting the graphic symbol and nodes).

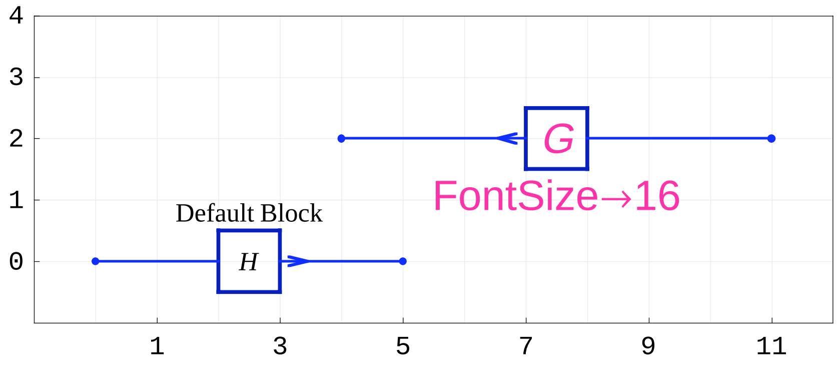
```
In[111]:=
      myDefaultBlock = {"Block", {{0, 0}, {5, 0}}, H, "Default Block"};
      myGreenBlock = {"Block", {{11, 2}, {4, 2}}, G, "Green Block",
         PlotStyle → {{RGBColor[0, 1, 0], Thickness[0.02]},
           {RGBColor[0, 0.7, 0], Thickness[0.01], PointSize[0.03]}}};
      {myDefaultBlock, myGreenBlock} // ShowSchematic
```



## TextStyle

`TextStyle` specifies font properties of labels and values. You can specify font family, font size, font color, etc. See *Mathematica* help for details.

```
In[114]:=
      myDefaultBlock = {"Block", {{0, 0}, {5, 0}}, H, "Default Block"};
      myFontBlock = {"Block", {{11, 2}, {4, 2}}, G, "FontSize→16",
         TextStyle → {FontFamily → Helvetica, FontSize → 16, FontColor → Hue[0.9]}};
      {myDefaultBlock, myFontBlock} // ShowSchematic
```



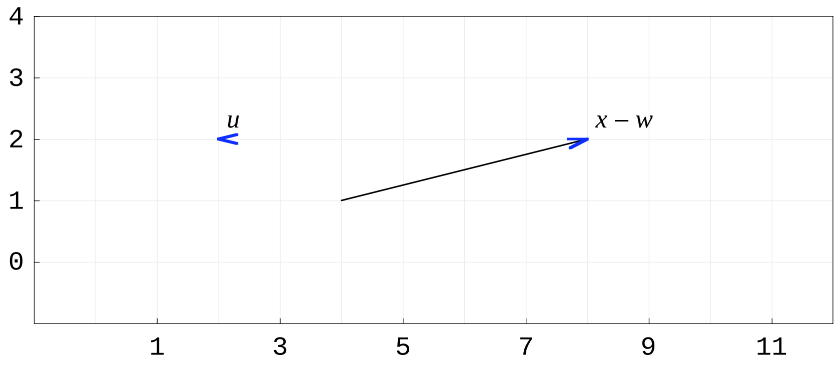
### ShowArrowTail

ShowArrowTail specifies the appearance of Arrow element.

**ShowArrowTail→True** draws both the arrow head and the arrow tail.

**ShowArrowTail→False** draws only the arrow head.

```
In[117]:=
      myDefaultArrow = {"Arrow", {{8, 2}, {4, 1}}, x - w};
      myNoTailArrow = {"Arrow", {{2, 2}, {6, 2}}, u,
         ShowArrowTail → False};
      {myDefaultArrow, myNoTailArrow} // ShowSchematic
```



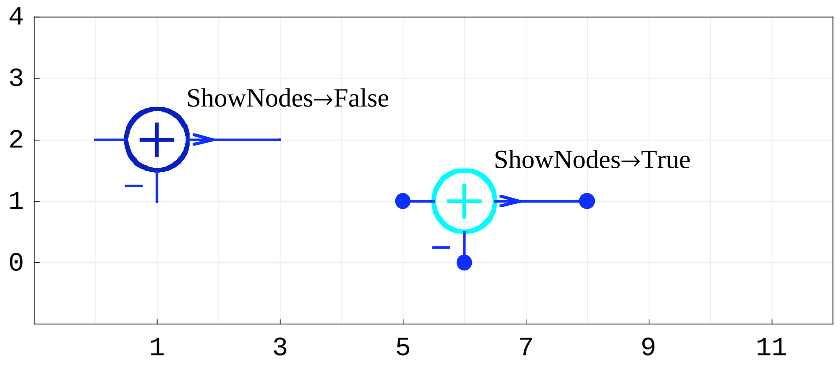
### ShowNodes

ShowNodes specifies the appearance of circles that represent nodes (junctions of element ports).

**ShowNodes→True** draws circles that represent nodes.

**ShowNodes→False** does not draw circles that represent nodes.

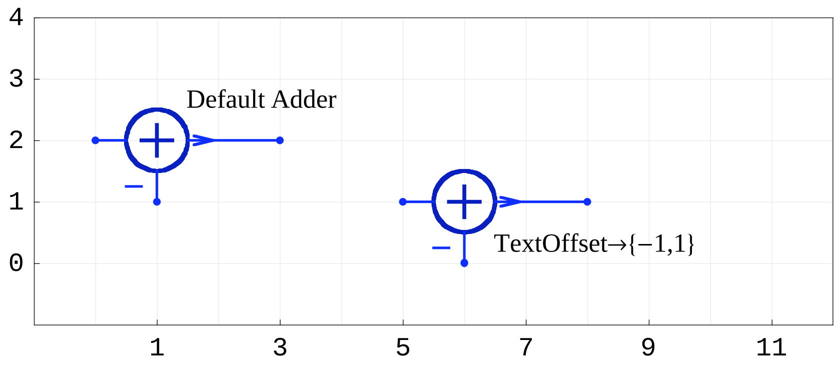
```
In[120]:=
      myAdder1 = {"Adder", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 0}, "ShowNodes→False",
         ShowNodes → False};
      myAdder2 =
        {"Adder", {{5, 1}, {6, 0}, {8, 1}, {6, 2}}, {1, -1, 2, 0}, "ShowNodes→True",
         PlotStyle → {{Hue[0.5]}, {PointSize[0.02]}},
         ShowNodes → True};
      {myAdder1, myAdder2} // ShowSchematic
```



## TextOffset

`TextOffset` specifies position of the element value and label.

```
In[123]:=
      myAdder1 = {"Adder", {{0, 2}, {1, 1}, {3, 2}, {1, 3}}, {1, -1, 2, 0}, "Default Adder"};
      myAdder2 =
        {"Adder", {{5, 1}, {6, 0}, {8, 1}, {6, 2}}, {1, -1, 2, 0}, "TextOffset→{-1,1}",
         TextOffset → {-1, 1}};
      {myAdder1, myAdder2} // ShowSchematic
```



See the *Mathematica* **Text** function for details about choosing the text offset.

## Default Options

Obtain the default options for drawing elements with the *Mathematica* function

```
In[126]:=
      Options[DrawElement]

Out[126]=
      {ElementSize → {1, 1},
       PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
         {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
       ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
       TextStyle → {FontFamily → Times, FontSize → 10}}
```

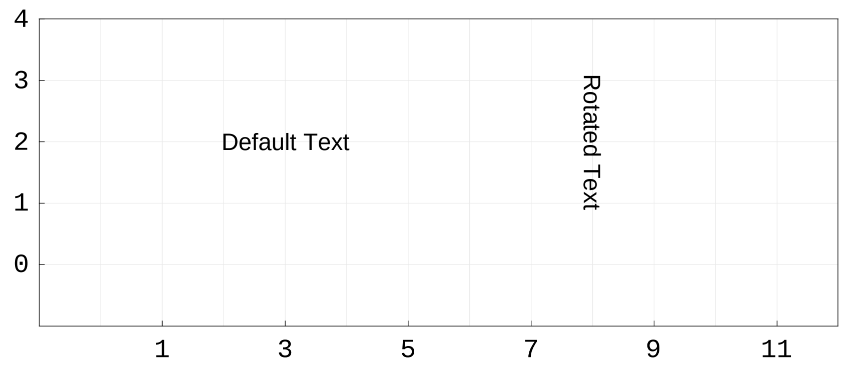
and change the options defaults with the *Mathematica* **SetOptions** function

```
In[127]:=
      SetOptions[DrawElement, TextStyle → {FontFamily → Arial, FontSize → 9}]

Out[127]=
      {ElementSize → {1, 1},
       PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
          {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
       ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
       TextStyle → {FontFamily → Arial, FontSize → 9}}
```

## Text Direction

Text element often annotates a schematic with a rotated text. The option `TextDirection` controls the angle of that rotation.

```
In[128]:=
      myDefaultText = {"Text", {3, 2}, "Default Text"};
      myRotatedText = {"Text", {8, 2}, "Rotated Text",
         TextDirection → {0, -1}};
      {myDefaultText, myRotatedText} // ShowSchematic
```

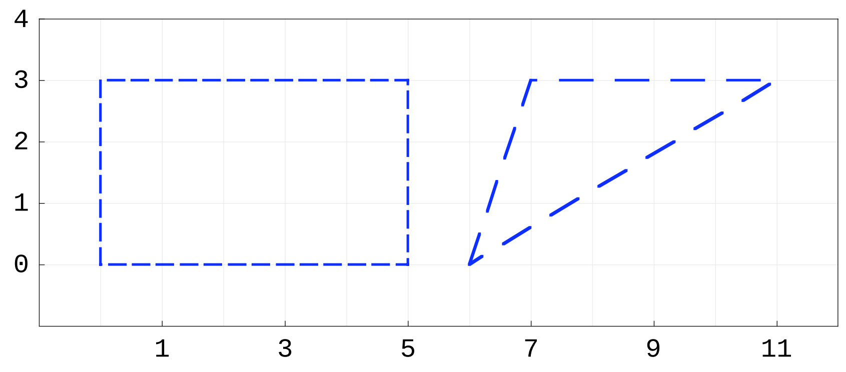


See the *Mathematica* **`Text`** function for details about choosing the text direction.

## Polyline Dashing

Special option is provided for controlling the dashing of the Polyline element.

```
In[131]:=
      myDefaultPolyline = {"Polyline", {{0, 0}, {0, 3}, {5, 3}, {5, 0}, {0, 0}}};
      myDashPolyline = {"Polyline", {{6, 0}, {7, 3}, {11, 3}, {6, 0}},
         PolylineDashing → Dashing[{0.04, 0.03}]};
      {myDefaultPolyline, myDashPolyline} // ShowSchematic
```



See the *Mathematica* **`Dashing`** function for details about choosing the dashing parameters.

## 3.8. Showing Schematic of Systems

### Graphical Representation of Systems

*SchematicSolver* describes a system as a list of elements; this list specifies what elements are in the system and how they are interconnected. A list describing a system will be referred to as the *system specification*.

Each element in the system is also described as a list that states what the element is, to which other elements it is connected, and what its value is. A list describing an element will be referred to as the *element specification*.

*SchematicSolver* draws the schematic of a system with the `ShowSchematic` function that takes the system specification as input. `ShowSchematic` is called as follows:

`ShowSchematic[`*systemSpecification*`]`

`ShowSchematic[`*systemSpecification*, *options*`]`

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[134]:=
      Needs["SchematicSolver`"];
```

Here is an example of a system specification. Firstly, let us specify elements that constitute the system:

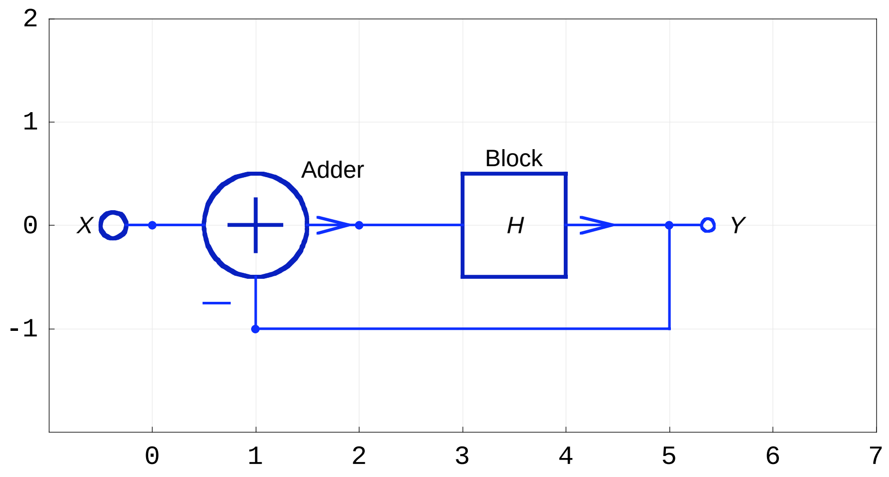
```
In[135]:=
      myInput = {"Input", {0, 0}, X, "Input"}
Out[135]=
      {Input, {0, 0}, X, Input}

In[136]:=
      myAdder = {"Adder", {{0, 0}, {1, -1}, {2, 0}, {1, 1}}, {1, -1, 2, 0}, "Adder"}
Out[136]=
      {Adder, {{0, 0}, {1, -1}, {2, 0}, {1, 1}}, {1, -1, 2, 0}, Adder}

In[137]:=
      myBlock = {"Block", {{2, 0}, {5, 0}}, H, "Block"}
Out[137]=
      {Block, {{2, 0}, {5, 0}}, H, Block}

In[138]:=
      myOutput = {"Output", {5, 0}, Y, "Output"}
Out[138]=
      {Output, {5, 0}, Y, Output}

In[139]:=
      myLine = {"Line", {{5, 0}, {5, -1}, {1, -1}}}
Out[139]=
      {Line, {{5, 0}, {5, -1}, {1, -1}}}
```

Next, let us form the system specification as a list of the element specifications:

```
In[140]:=
      mySchematic = {myInput, myAdder, myBlock, myOutput, myLine}

Out[140]=
      {{Input, {0, 0}, X, Input},
       {Adder, {{0, 0}, {1, -1}, {2, 0}, {1, 1}}, {1, -1, 2, 0}, Adder},
       {Block, {{2, 0}, {5, 0}}, H, Block},
       {Output, {5, 0}, Y, Output}, {Line, {{5, 0}, {5, -1}, {1, -1}}}}
```

Finally, we show the graphic representation of the system with `ShowSchematic`:

```
In[141]:=
      ShowSchematic[mySchematic, PlotRange → {{-1, 7}, {-2, 2}}]
```



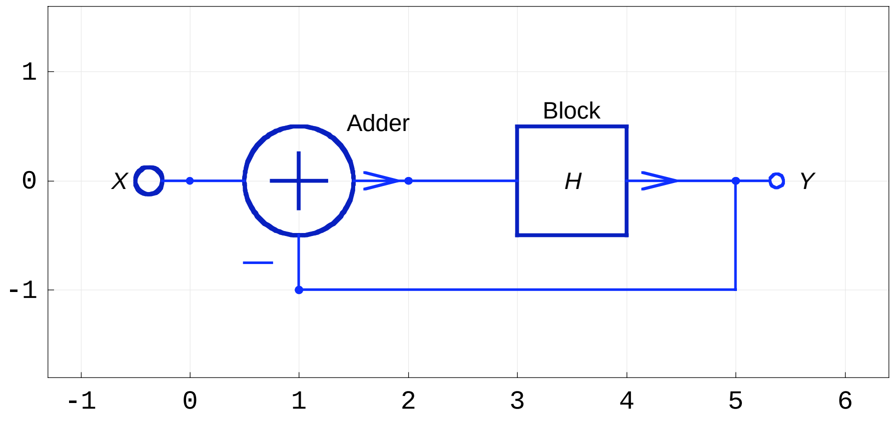
## Drawing Options for ShowSchematic

Graphics created by `ShowSchematic` can be fine-tuned by various options:

```
In[142]:=
      Options[ShowSchematic] // ColumnForm

Out[142]=
      ElementScale → 1
      FontSize → Automatic
      Frame → True
      GridLines → Automatic
      PlotRange → {{-1, 12}, {-1, 4}}
```

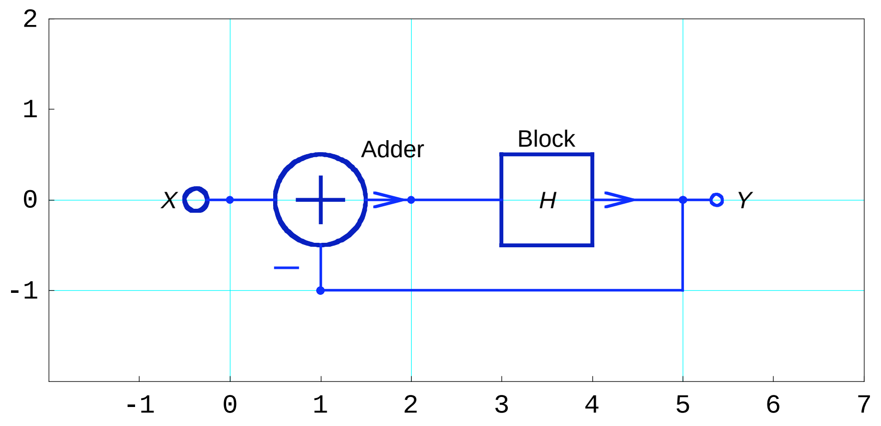
`PlotRange` is an option that specifies what points to include in a plot. (See the *Mathematica* help for details.)

```
In[143]:=
        ShowSchematic[mySchematic,
         PlotRange → {{-1.3, 6.4}, {-1.8, 1.6}}]
```



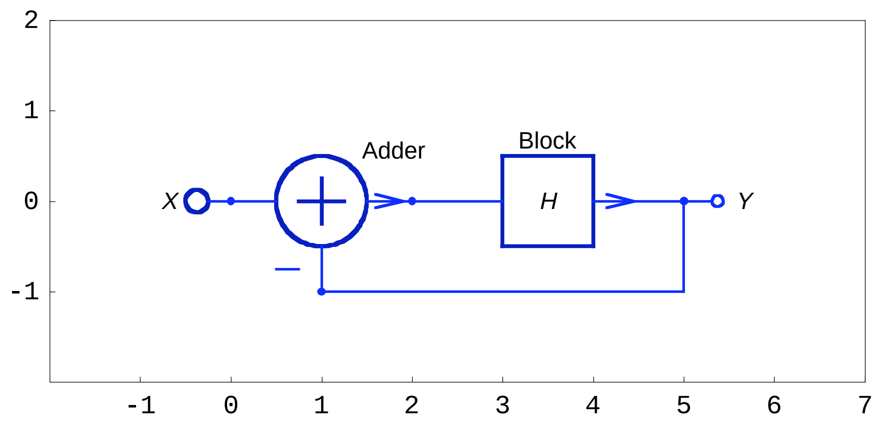
`GridLines` is an option that specifies grid lines. (See the *Mathematica* help for details.)

```
In[144]:=
        myXhorGrid = {{0, {Hue[0.5]}}, {2, {Hue[0.5]}}, {5, {Hue[0.5]}}};
        myYverGrid = {{-1, {Hue[0.5]}}, {0, {Hue[0.5]}}};
        myGridLines = GridLines → {myXhorGrid, myYverGrid}

Out[146]=
        GridLines → {{{0, {Hue[0.5]}}, {2, {Hue[0.5]}}, {5, {Hue[0.5]}}},
          {{-1, {Hue[0.5]}}, {0, {Hue[0.5]}}}}

In[147]:=
        ShowSchematic[mySchematic,
         PlotRange → {{-2, 7}, {-2, 2}}, myGridLines]
```



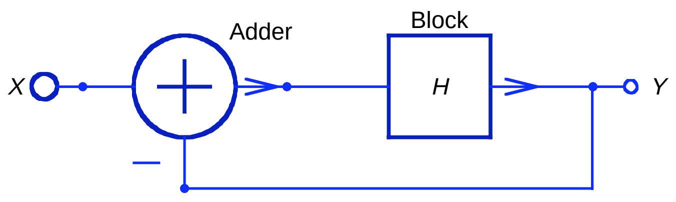
No grid lines are drawn with **`GridLines→None`**:

```
In[148]:=
        ShowSchematic[mySchematic, PlotRange → {{-2, 7}, {-2, 2}},
         GridLines → None]
```



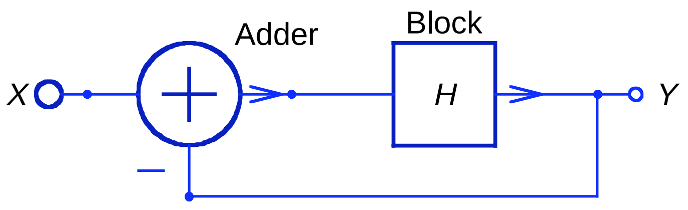
`Frame` is an option that specifies whether a frame should be drawn around the plot. **`Frame → True`** by default draws a frame with tick marks. **`Frame → False`** draws no frame:

```
In[149]:=
        ShowSchematic[mySchematic, PlotRange → {{-2, 7}, {-2, 2}}, GridLines → None,
         Frame → False]
```



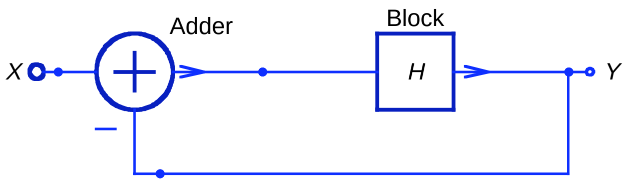
`FontSize` is an option that specifies the size in points of the font in which to render values and labels. Let us choose the font size of 12 points:

```
In[150]:=
        ShowSchematic[mySchematic,
         PlotRange → {{-2, 7}, {-2, 2}}, GridLines → None, Frame → False,
         FontSize → 12]
```



`ElementScale` is an option that specifies the reduction or magnification of element dimensions. Let us reduce the elements by 25%:

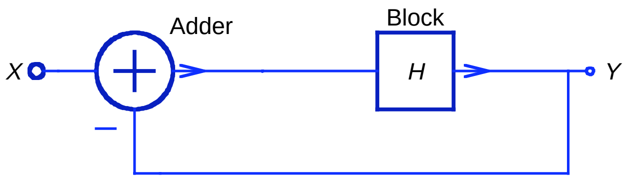
```
In[151]:=
      ShowSchematic[mySchematic,
       PlotRange → {{-2, 7}, {-2, 2}}, GridLines → None, Frame → False,
       ElementScale → (1 - 25 / 100)]
```



You can change the default options for drawing elements. For example, to suppress drawing element nodes, execute

```
In[152]:=
      SetOptions[DrawElement, ShowNodes → False];
```

```
In[153]:=
      ShowSchematic[mySchematic, PlotRange → {{-2, 7}, {-2, 2}},
       GridLines → None, Frame → False, ElementScale → 0.75]
```



If necessary, you can restore the initial option by

```
In[154]:=
      SetOptions[DrawElement, ShowNodes → True];
```

Feel free to experiment with different setting for the `ShowSchematic` options.

## 3.9. Check Syntax of Schematic Specification

### Introduction

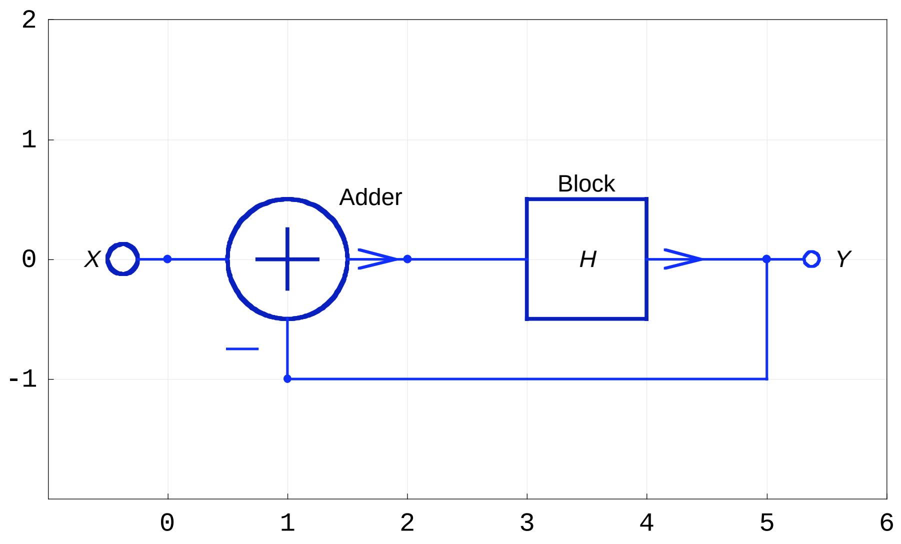
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[155]:=
      Needs["SchematicSolver`"];
```

## Check Schematic Syntax

`CheckSchematicSyntax` checks the syntax of a schematic specification.

```
In[156]:=
      mySystem = {{"Input", {0, 0}, X, "Input"},
         {"Adder", {{0, 0}, {1, -1}, {2, 0}, {1, 1}}, {1, -1, 2, 0}, "Adder"},
         {"Block", {{2, 0}, {5, 0}}, H, "Block"},
         {"Output", {5, 0}, Y, "Output"},
         {"Line", {{5, 0}, {5, -1}, {1, -1}}}};
      ShowSchematic[%, PlotRange → {{-1, 6}, {-2, 2}}];
```



`CheckSchematicSyntax` returns `False` if the schematic specification is not a list specifying a schematic.

```
In[158]:=
      CheckSchematicSyntax[mySystem]

Out[158]=
      True
```

## Check Element Syntax

`CheckElementSyntax` checks the syntax of an element specification. We can individually check any element in the schematic specification:

```
In[159]:=
      mySystem[[3]]
      CheckElementSyntax[%]

Out[159]=
      {Block, {{2, 0}, {5, 0}}, H, Block}

Out[160]=
      True
```

`CheckElementSyntax` returns `False` if the element specification is not a list specifying an element:

```
In[161]:=
      myBadElement = {"Gain", {{0, 0}, {4, 0}}, 3 / 4, "g"}
      CheckElementSyntax[%]
```

```
Out[161]=
```

$\{\text{Gain}, \{\{0, 0\}, \{4, 0\}\}, \frac{3}{4}, \text{g}\}$

```
      CheckElementSyntax::unsup : Unsupported element "Gain" in the specification
         {"Gain", {{0, 0}, {4, 0}}, 3/4, "g"}. Supported elements are Adder, Amplifier,
         Arrow, Block, Delay, Input, Integrator, Line, Multiplier, Node, Output,
         Polyline, and Text. Element name is enclosed within double quotation marks.

Out[162]=
      False
```

## 3.10. Discrete Signals

### Introduction

*SchematicSolver* can draw, solve, and implement discrete systems with several inputs and several outputs. These systems are known as multiple-input multiple-output systems, or MIMO systems for short.

Samples that are inputted to, or outputted from, MIMO systems are represented in *SchematicSolver* as matrices that contain several signals, and are of the form

$\{\{a_0, b_0, c_0, \ldots, w_0\}, \{a_1, b_1, c_1, \ldots, w_1\}, \{a_2, b_2, c_2, \ldots, w_2\}, \ldots, \{a_{N-1}, b_{N-1}, c_{N-1}, \ldots, w_{N-1}\}\}$

where

$a_0, a_1, a_2, \ldots, a_{N-1}$ represent $N$ samples of the first discrete signal,

$b_0, b_1, b_2, \ldots, b_{N-1}$ represent $N$ samples of the second discrete signal, and so on.

Each column of the above matrix represents a discrete signal; each row of this matrix is a data set that is processed at a time as a unit. *SchematicSolver* refers to this matrix as *data sequence*, or *sequence* for short.

Consider a single-input single-output system, also referred to as the SISO system. The input sequence to the system is of the form

$\{\{a_0\}, \{a_1\}, \{a_2\}, \ldots, \{a_{N-1}\}\}$

and it is an $N$-by-1 matrix. In other words, this sequence is a single-column matrix.

Here are example sequences:

```
In[163]:=
      dataSeqMIMO = {{a0, b0, c0}, {a1, b1, c1}, {a2, b2, c2}, {a3, b3, c3}}

Out[163]=
      {{a0, b0, c0}, {a1, b1, c1}, {a2, b2, c2}, {a3, b3, c3}}

In[164]:=
      dataSeqMIMO // TraditionalForm

Out[164]//TraditionalForm=
```

$$\begin{pmatrix} a_0 & b_0 & c_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix}$$

```
In[165]:=
      dataSeqSISO = {{a₀}, {a₁}, {a₂}, {a₃}}
```

```
Out[165]=
      {{a₀}, {a₁}, {a₂}, {a₃}}
```

```
In[166]:=
      dataSeqSISO // TraditionalForm
```

```
Out[166]//TraditionalForm=
```

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

## Creating Input Sequences

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[167]:=
      Needs["SchematicSolver`"];
```

You can create the most typical input sequences with the following *SchematicSolver*'s functions:

```
In[168]:=
      myImpSeq = UnitImpulseSequence[]
```

```
Out[168]=
      {{1}, {0}, {0}, {0}, {0}, {0}, {0}, {0}}
```

```
In[169]:=
      myStepSeq = UnitStepSequence[]
```

```
Out[169]=
      {{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}}
```

```
In[170]:=
      myRampSeq = UnitRampSequence[]
```

```
Out[170]=
      {{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}}
```

```
In[171]:=
      mySineSeq = UnitSineSequence[]
```

```
Out[171]=
```

$$\left\{\{0\}, \left\{\frac{1}{\sqrt{2}}\right\}, \{1\}, \left\{\frac{1}{\sqrt{2}}\right\}, \{0\}, \left\{-\frac{1}{\sqrt{2}}\right\}, \{-1\}, \left\{-\frac{1}{\sqrt{2}}\right\}\right\}$$

```
In[172]:=
      myExpSeq = UnitExponentialSequence[]
```

```
Out[172]=
```

$$\left\{\{1\}, \left\{\frac{1}{2}\right\}, \left\{\frac{1}{4}\right\}, \left\{\frac{1}{8}\right\}, \left\{\frac{1}{16}\right\}, \left\{\frac{1}{32}\right\}, \left\{\frac{1}{64}\right\}, \left\{\frac{1}{128}\right\}\right\}$$

```
In[173]:=
      myRandSeq = UnitNoiseSequence[]
```

```
Out[173]=
      {{0.856559}, {-0.224317}, {-0.516228}, {-0.316171},
       {-0.361907}, {-0.0408154}, {0.624452}, {0.597351}}
```

The above functions can be combined to generate compound sequences:

```
In[174]:=
      myCombSeq = UnitExponentialSequence[8, -0.1, E] * Abs[UnitSineSequence[8, 1 / 8]]
Out[174]=
      {{0}, {0.639817}, {0.818731}, {0.523838}, {0}, {0.428882}, {0.548812}, {0.351139}}
```

In addition, you can create your own sequences by typing the particular sample values:

```
In[175]:=
      mySeq = {{1}, {0.9}, {-0.7}, {0.5}, {0.1}, {0}, {-1 / 6}, {-0.3}}
Out[175]=
```

$\left\{\{1\}, \{0.9\}, \{-0.7\}, \{0.5\}, \{0.1\}, \{0\}, \left\{-\frac{1}{6}\right\}, \{-0.3\}\right\}$

The above sequence can be converted to a list with

```
In[176]:=
      myList = SequenceToList[mySeq]
Out[176]=
```

$\left\{1, 0.9, -0.7, 0.5, 0.1, 0, -\frac{1}{6}, -0.3\right\}$

A list of values can be converted to a sequence with

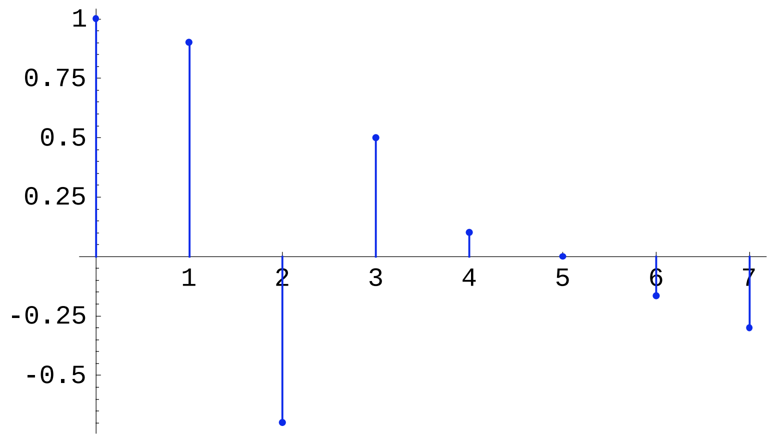
```
In[177]:=
      mySequence = ListToSequence[myList]
Out[177]=
```

$\left\{\{1\}, \{0.9\}, \{-0.7\}, \{0.5\}, \{0.1\}, \{0\}, \left\{-\frac{1}{6}\right\}, \{-0.3\}\right\}$

## Plotting Sequences

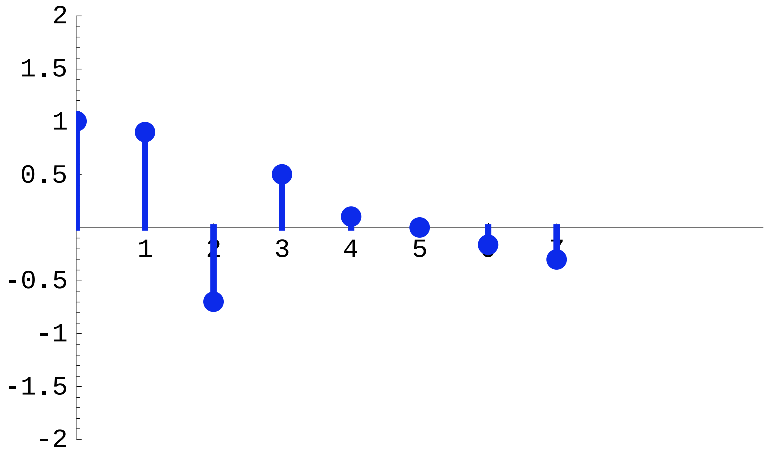
*SchematicSolver*'s function `SequencePlot` plots sequences in a traditional way as a stem plot:

```
In[178]:=
      SequencePlot[mySequence];
```



Add options to fine-tune the plot:

```
In[179]:=
      SequencePlot[mySequence,
        PlotRange → {{0, 10}, {-2, 2}},
        SequencePointSize → 0.03,
        SequenceLineThickness → 0.01];
```



## Combining Sequences

You can combine several sequences into one sequence with the *SchematicSolver*'s function `MultiplexSequence`:

```
In[180]:=
      myMuxSeq = MultiplexSequence[myRandSeq, myStepSeq / Sqrt[2], 2 * myExpSeq]
```

```
Out[180]=
```

$$\left\{\left\{0.856559, \frac{1}{\sqrt{2}}, 2\right\}, \left\{-0.224317, \frac{1}{\sqrt{2}}, 1\right\}, \left\{-0.516228, \frac{1}{\sqrt{2}}, \frac{1}{2}\right\}, \left\{-0.316171, \frac{1}{\sqrt{2}}, \frac{1}{4}\right\}, \left\{-0.361907, \frac{1}{\sqrt{2}}, \frac{1}{8}\right\}, \left\{-0.0408154, \frac{1}{\sqrt{2}}, \frac{1}{16}\right\}, \left\{0.624452, \frac{1}{\sqrt{2}}, \frac{1}{32}\right\}, \left\{0.597351, \frac{1}{\sqrt{2}}, \frac{1}{64}\right\}\right\}$$

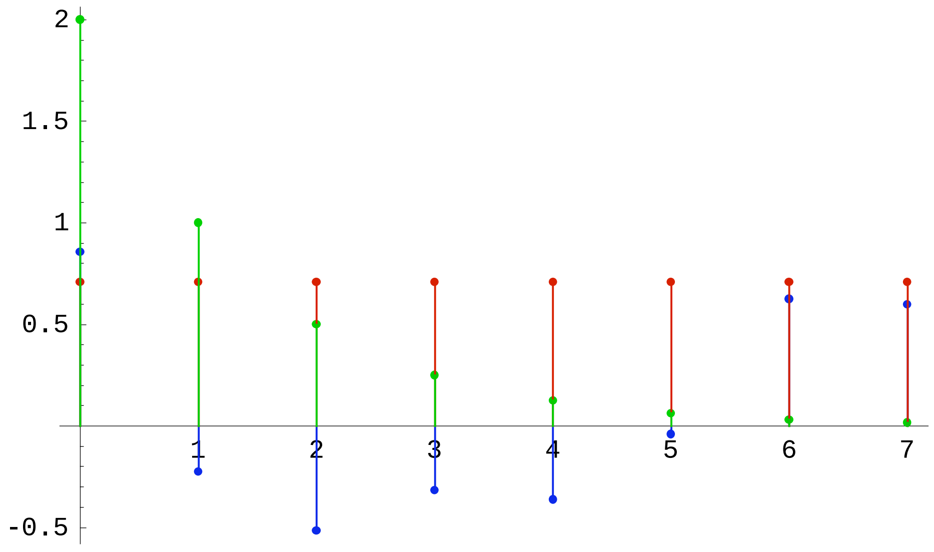
```
In[181]:=
      myMuxSeq // TraditionalForm
```

```
Out[181]//TraditionalForm=
```

$$\begin{pmatrix} 0.856559 & \frac{1}{\sqrt{2}} & 2 \\ -0.224317 & \frac{1}{\sqrt{2}} & 1 \\ -0.516228 & \frac{1}{\sqrt{2}} & \frac{1}{2} \\ -0.316171 & \frac{1}{\sqrt{2}} & \frac{1}{4} \\ -0.361907 & \frac{1}{\sqrt{2}} & \frac{1}{8} \\ -0.0408154 & \frac{1}{\sqrt{2}} & \frac{1}{16} \\ 0.624452 & \frac{1}{\sqrt{2}} & \frac{1}{32} \\ 0.597351 & \frac{1}{\sqrt{2}} & \frac{1}{64} \end{pmatrix}$$

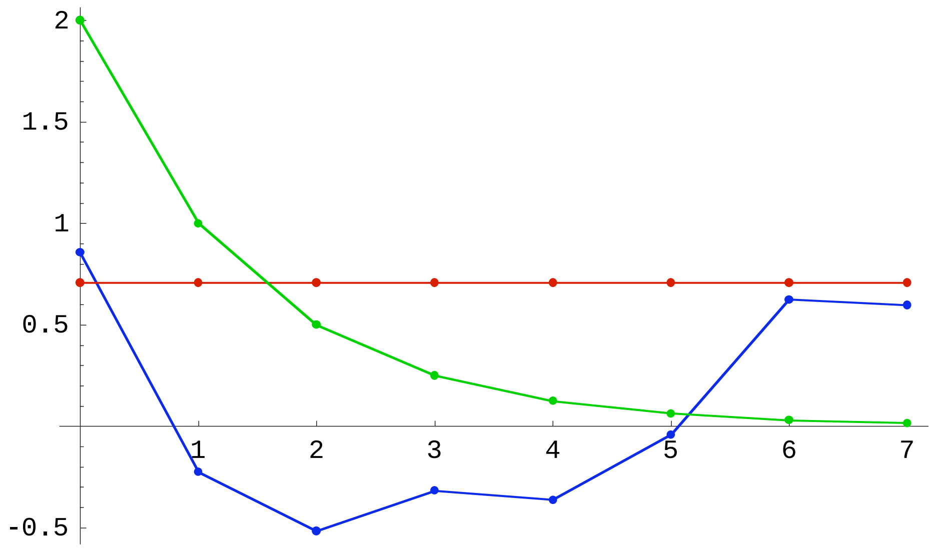
The above sequence is referred to as a *multiplexed sequence*. The plot of the multiplexed sequences follows:

```
In[182]:=
      SequencePlot[myMuxSeq];
```



You can plot the discrete signals more clearly by setting the `SequencePlot` options to `StemPlot→False` and `PlotJoined→True`.

```
In[183]:=
      SequencePlot[myMuxSeq, StemPlot → False, PlotJoined → True];
```



Each multiplexed sequence is plotted in a different color. The first sequence is plotted in blue, the second sequence is plotted in red, and so on.

You can extract individual sequences from a multiplexed sequence with

```
In[184]:=
      {seq1, seq2, seq3} = DemultiplexSequence[myMuxSeq]
```

```
Out[184]=
```

$$\{\{\{0.856559\},\{-0.224317\},\{-0.516228\},\{-0.316171\},\{-0.361907\},\{-0.0408154\},\{0.624452\},\{0.597351\}\},$$
$$\{\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\},\{\tfrac{1}{\sqrt{2}}\}\},$$
$$\{\{2\},\{1\},\{\tfrac{1}{2}\},\{\tfrac{1}{4}\},\{\tfrac{1}{8}\},\{\tfrac{1}{16}\},\{\tfrac{1}{32}\},\{\tfrac{1}{64}\}\}\}$$

Obviously, the sequences `seq1` and `myRandSeq` must be the same:

```
In[185]:=
      SameQ[seq1, myRandSeq]

Out[185]=
      True
```

## 3.11. Fourier Transform of Discrete Signals

### Introduction

The theory of transforms has an important aspect in examining the nature of signals or sequences. The term *transform* refers to a mathematical operation that takes a given function, called the *original* and returns a new function, referred to as the *image*. The transformation is often done by means of summation formula. Commonly used transforms are named after Fourier.

The *Fourier Transform* (FT) occupies a central place in analysis of signals and systems. A major reason for this is that the most convenient way of measuring and specifying the performance of a system or signal is based on frequency. The *Discrete-Time Fourier Transform* (DTFT) and the *Discrete Fourier Transform* (DFT) are very important tools for signal processing techniques.

*SchematicSolver* has functions for computing and plotting FT:

`SequenceDiscreteFourierTransform`,

`SequenceDiscreteFourierTransformMagnitudePlot`,

`SequenceFourierTransform`,

`SequenceFourierTransformMagnitudePlot`.

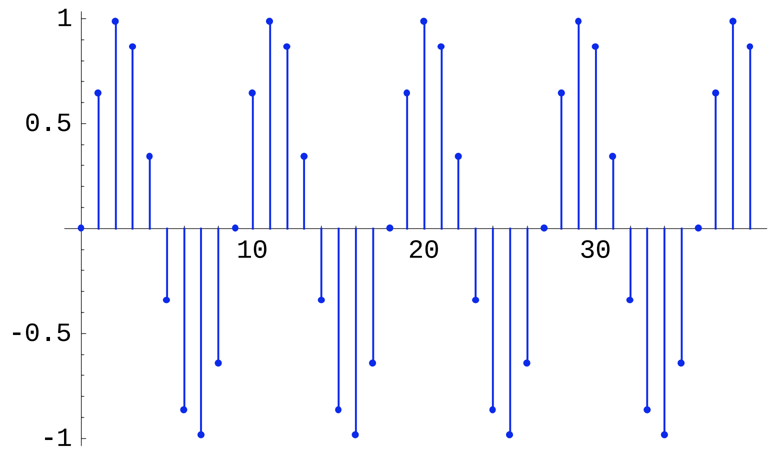
The functions will be illustrated by typical examples often met in practice.

### Discrete-Time Fourier Transform (DTFT)

Consider a sinusoidal sequence, of amplitude 1, of digital frequency $\frac{1}{9}$, and of zero initial phase, represented by 40 samples:

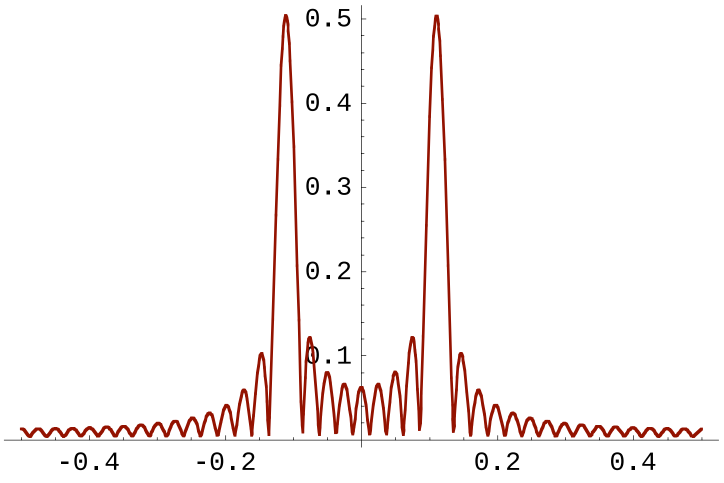
```
In[186]:=
      Needs["SchematicSolver`"];

In[187]:=
      amplitude = 1;
      sineDigitalFreq = 1 / 9;
      initialPhase = 0;
      numberOfSamples = 40;

In[191]:=
      sineSeq = amplitude * UnitSineSequence[numberOfSamples, sineDigitalFreq, initialPhase];
```

*In[192]:=*
```
SequencePlot[sineSeq];
```



`SequenceFourierTransformMagnitudePlot` plots the Discrete-Time Fourier Transform magnitude of the sequence `sineSeq`.

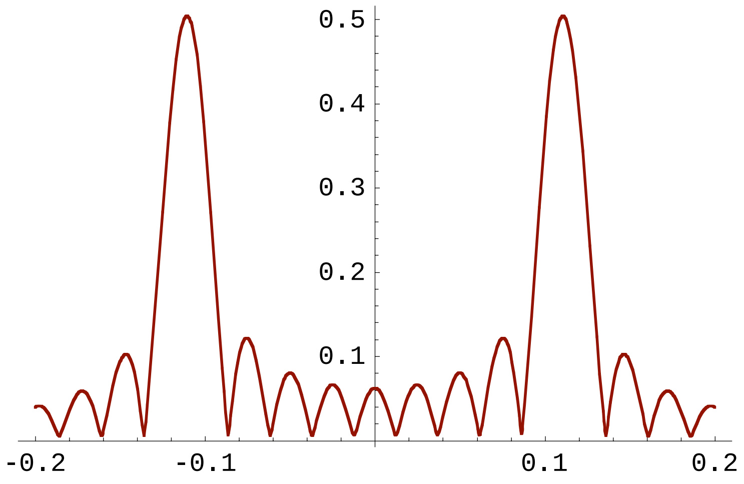
*In[193]:=*
```
SequenceFourierTransformMagnitudePlot[sineSeq];
```



The magnitude plot shows peaks that correspond to `sineDigitalFreq`. By default, `SequenceFourierTransformMagnitudePlot` plots magnitude over the frequency range from $\frac{-1}{2}$ to $\frac{1}{2}$. You can change the frequency range to zoom the peaks:

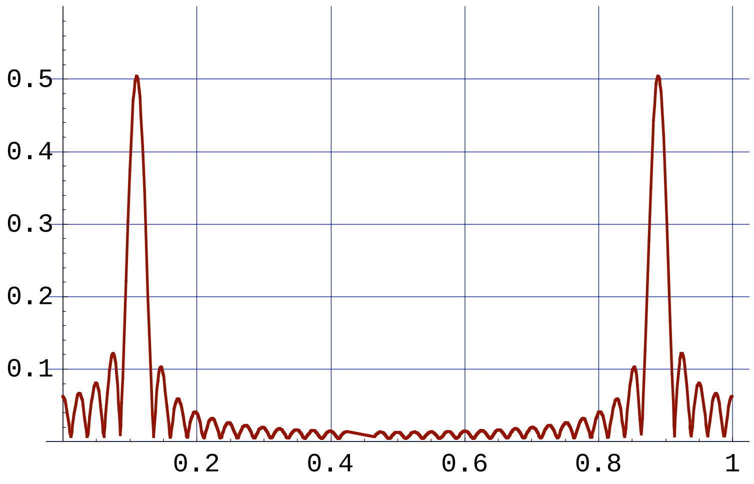
*In[194]:=*
```
freqRange = {-0.2, 0.2};
```

```
In[195]:=
        SequenceFourierTransformMagnitudePlot[sineSeq, freqRange];
```



In addition, you can change the plot range and add grid lines to refine the graphic:

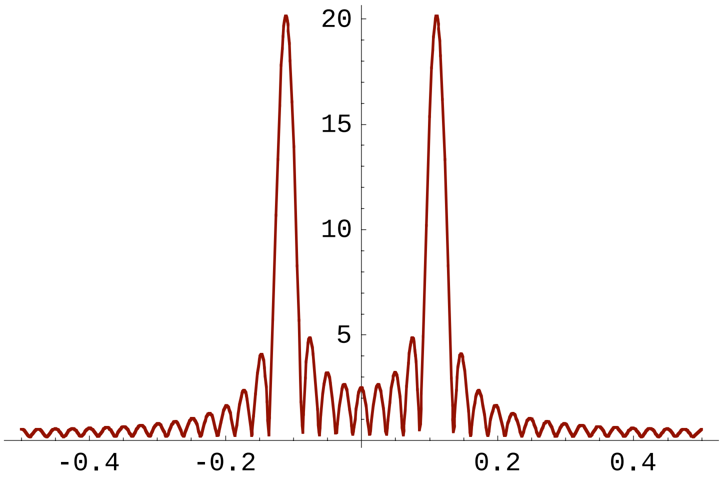
```
In[196]:=
        SequenceFourierTransformMagnitudePlot[sineSeq, {0, 1},
          PlotRange → {Automatic, {0, 0.6}},
          GridLines → Automatic];
```



By default, `SequenceFourierTransformMagnitudePlot` plots normalized magnitude, so the peak values correspond to `amplitude/2`. The *normalized magnitude* is the DTFT magnitude divided by the number of samples.

Here is the magnitude plot without normalization:

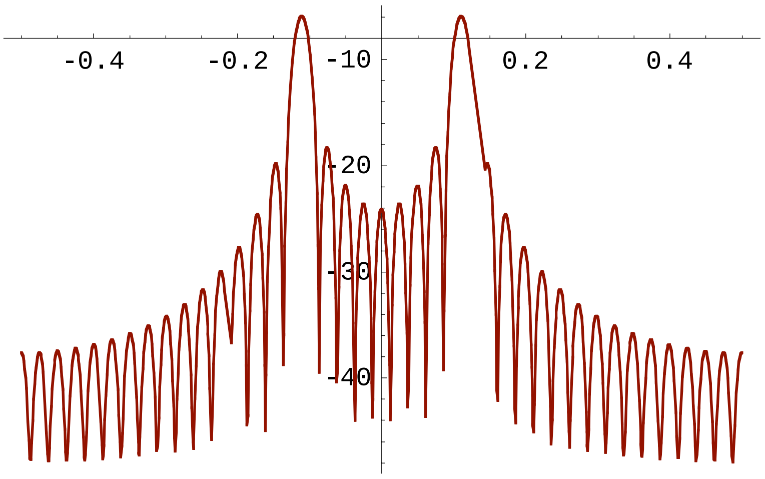
```
In[197]:=
      SequenceFourierTransformMagnitudePlot[sineSeq,
        NormalizedSpectrum → False];
```



Note that, for `NormalizedSpectrum→False`, the peak values correspond to `numberOf-Samples*amplitude/2`.

`SequenceFourierTransformMagnitudePlot` plots the DTFT magnitude in linear scale. You can plot the magnitude in decibels (dB) with:

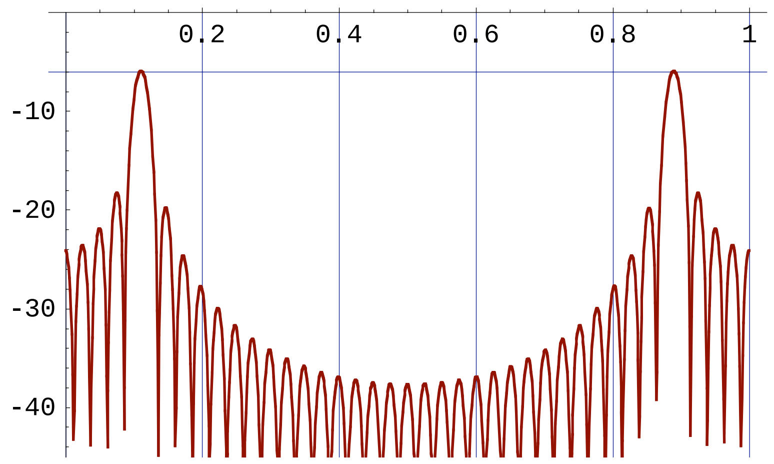
```
In[198]:=
      SequenceFourierTransformMagnitudePlot[sineSeq,
        dBMagnitudePlot → True];
```



The peak value in dB corresponds to

```
In[199]:=
      dBamplitude2 = 20 * Log[10, amplitude / 2];
      % // N

Out[200]=
      -6.0206
```

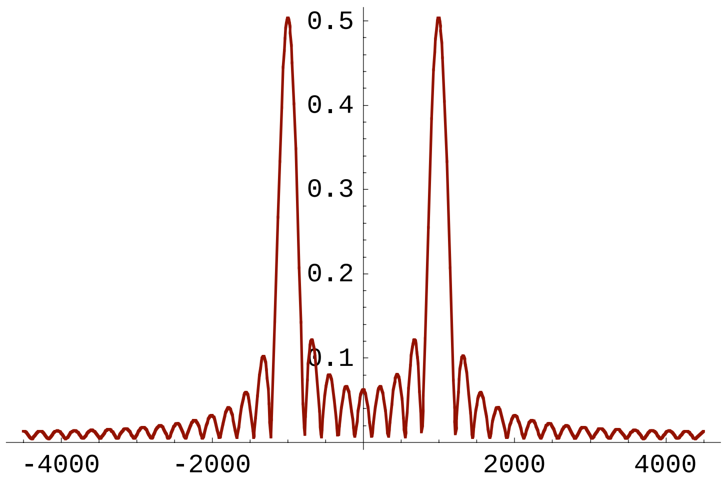
```
In[201]:=
      SequenceFourierTransformMagnitudePlot[
        sineSeq, {0, 1}, dBMagnitudePlot → True, PlotRange → {-45, 0},
        GridLines → {Automatic, {dBamplitude2}}];
```



Assume that the sinusoidal sequence was obtained by sampling a continuous-time sine signal, of frequency 1000 Hz, at the rate $F_{\text{samp}}$ = 9000 Hz.

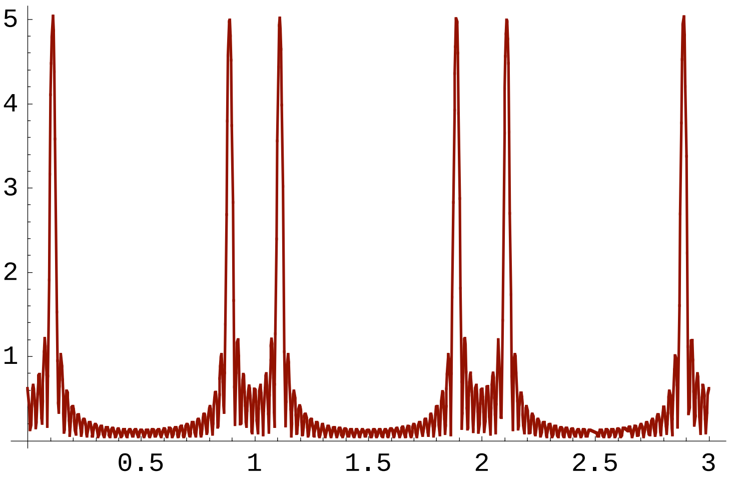
`SequenceFourierTransformMagnitudePlot` can plot the magnitude versus the continuous-time frequency with

```
In[202]:=
      SequenceFourierTransformMagnitudePlot[sineSeq,
        SequenceSamplingFrequency → 9000];
```



`SequenceFourierTransformMagnitudePlot` can plot the magnitude over an arbitrary range of digital frequencies:

```
In[203]:=
      SequenceFourierTransformMagnitudePlot[sineSeq, {0, 3}];
```

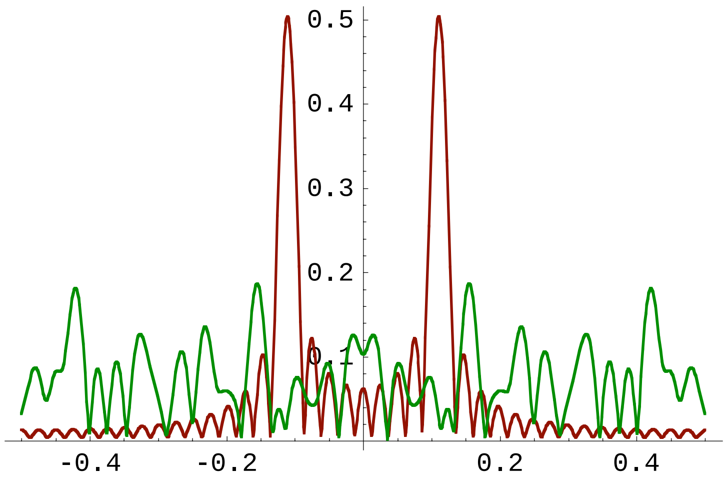


You can plot DTFT magnitudes of several sequences:

```
In[204]:=
      noiseSeq = UnitNoiseSequence[numberOfSamples];
```

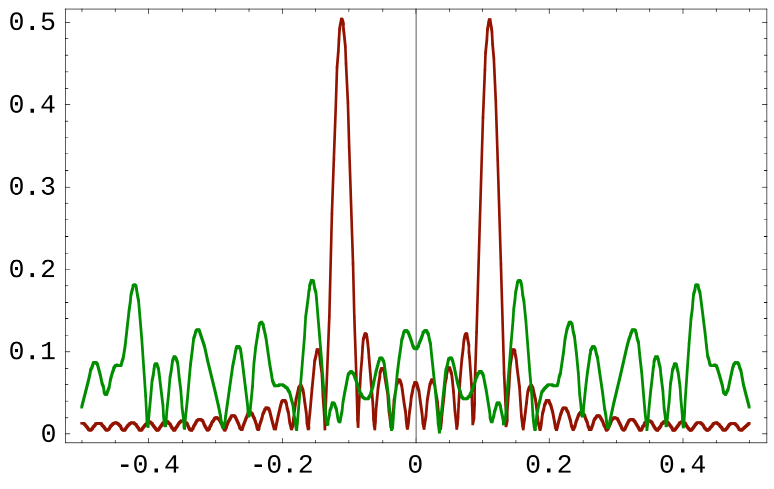
```
In[205]:=
      sineNoiseSeq = MultiplexSequence[sineSeq, noiseSeq];
```

```
In[206]:=
      SequenceFourierTransformMagnitudePlot[sineNoiseSeq];
```

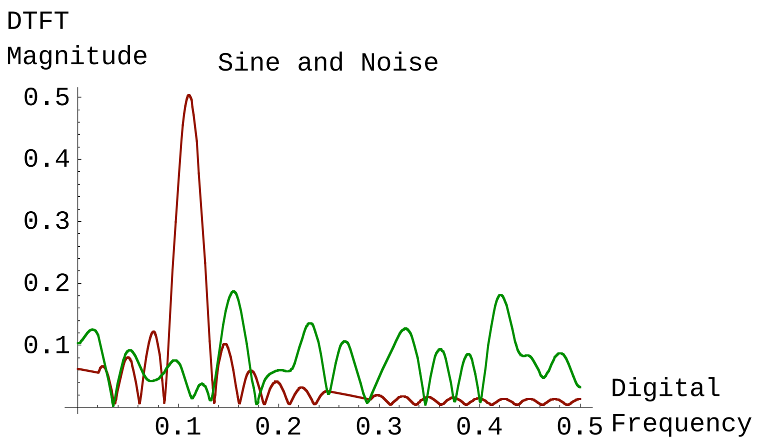


Pass any `Plot` option to `SequenceFourierTransformMagnitudePlot` to refine the graphic:

```
In[207]:=
      SequenceFourierTransformMagnitudePlot[sineNoiseSeq,
        Frame → True];
```



```
In[208]:=
      SequenceFourierTransformMagnitudePlot[sineNoiseSeq, {0, 1 / 2},
        PlotLabel → "Sine and Noise",
        AxesLabel → {"Digital\nFrequency", "DTFT\nMagnitude"}];
```



SequenceFourierTransform computes the Discrete-Time Fourier Transform (DTFT) of a sequence. It returns exact closed-form expressions for DTFT in terms of the digital frequency and sample values:

```
In[209]:=
      sineSeq10 = UnitSineSequence[10, sineDigitalFreq]
Out[209]=
```

$$\left\{\{0\}, \left\{\text{Sin}\left[\frac{2\pi}{9}\right]\right\}, \left\{\text{Sin}\left[\frac{4\pi}{9}\right]\right\}, \left\{\frac{\sqrt{3}}{2}\right\}, \left\{\text{Sin}\left[\frac{8\pi}{9}\right]\right\}, \left\{\text{Sin}\left[\frac{10\pi}{9}\right]\right\}, \left\{-\frac{\sqrt{3}}{2}\right\}, \left\{\text{Sin}\left[\frac{14\pi}{9}\right]\right\}, \left\{\text{Sin}\left[\frac{16\pi}{9}\right]\right\}, \{0\}\right\}$$

```
In[210]:=
      {sineSeq10DTFT} = SequenceFourierTransform[sineSeq10]
Out[210]=
```

$$\left\{\frac{1}{2}\sqrt{3}\, e^{-6 i f \pi} - \frac{1}{2}\sqrt{3}\, e^{-12 i f \pi} + e^{-2 i f \pi}\,\text{Sin}\left[\frac{2\pi}{9}\right] + e^{-4 i f \pi}\,\text{Sin}\left[\frac{4\pi}{9}\right] + e^{-8 i f \pi}\,\text{Sin}\left[\frac{8\pi}{9}\right] + e^{-10 i f \pi}\,\text{Sin}\left[\frac{10\pi}{9}\right] + e^{-14 i f \pi}\,\text{Sin}\left[\frac{14\pi}{9}\right] + e^{-16 i f \pi}\,\text{Sin}\left[\frac{16\pi}{9}\right]\right\}$$

By default, `SequenceFourierTransform` denotes the digital frequency with *f*. You can specify an alternative symbol for the digital frequency with

```
In[211]:=
      SequenceFourierTransform[sineSeq10, f1]
```

Out[211]=

$$\left\{\frac{1}{2}\sqrt{3}\,e^{-6\,i\,f1\,\pi}-\frac{1}{2}\sqrt{3}\,e^{-12\,i\,f1\,\pi}+e^{-2\,i\,f1\,\pi}\,\mathrm{Sin}\left[\frac{2\pi}{9}\right]+e^{-4\,i\,f1\,\pi}\,\mathrm{Sin}\left[\frac{4\pi}{9}\right]+e^{-8\,i\,f1\,\pi}\,\mathrm{Sin}\left[\frac{8\pi}{9}\right]+e^{-10\,i\,f1\,\pi}\,\mathrm{Sin}\left[\frac{10\pi}{9}\right]+e^{-14\,i\,f1\,\pi}\,\mathrm{Sin}\left[\frac{14\pi}{9}\right]+e^{-16\,i\,f1\,\pi}\,\mathrm{Sin}\left[\frac{16\pi}{9}\right]\right\}$$

If you work with sampled continuous-time signals, you can specify the sampling frequency as an option:

```
In[212]:=
      SequenceFourierTransform[sineSeq10,
       SequenceSamplingFrequency → Fsamp]
```

Out[212]=

$$\left\{\frac{1}{2}\sqrt{3}\,e^{-\frac{6\,i\,f\,\pi}{Fsamp}}-\frac{1}{2}\sqrt{3}\,e^{-\frac{12\,i\,f\,\pi}{Fsamp}}+e^{-\frac{2\,i\,f\,\pi}{Fsamp}}\,\mathrm{Sin}\left[\frac{2\pi}{9}\right]+e^{-\frac{4\,i\,f\,\pi}{Fsamp}}\,\mathrm{Sin}\left[\frac{4\pi}{9}\right]+e^{-\frac{8\,i\,f\,\pi}{Fsamp}}\,\mathrm{Sin}\left[\frac{8\pi}{9}\right]+e^{-\frac{10\,i\,f\,\pi}{Fsamp}}\,\mathrm{Sin}\left[\frac{10\pi}{9}\right]+e^{-\frac{14\,i\,f\,\pi}{Fsamp}}\,\mathrm{Sin}\left[\frac{14\pi}{9}\right]+e^{-\frac{16\,i\,f\,\pi}{Fsamp}}\,\mathrm{Sin}\left[\frac{16\pi}{9}\right]\right\}$$

In that case, `f` denotes the continuous-time frequency in Hertz (Hz).

`SequenceFourierTransform` works with symbolic values of samples. Here is a symbolic sequence and its DTFT:

```
In[213]:=
      symbSeq = UnitSymbolicSequence[6, a, 0]

Out[213]=
      {{a0}, {a1}, {a2}, {a3}, {a4}, {a5}}

In[214]:=
      SequenceFourierTransform[symbSeq]
```

Out[214]=

$$\{a0+a1\,e^{-2\,i\,f\,\pi}+a2\,e^{-4\,i\,f\,\pi}+a3\,e^{-6\,i\,f\,\pi}+a4\,e^{-8\,i\,f\,\pi}+a5\,e^{-10\,i\,f\,\pi}\}$$

Note that `SequenceFourierTransform` returns the result as a list.

`SequenceFourierTransform` can find DTFT of a multiplex sequence:

```
In[215]:=
      symbSeqTwo = UnitSymbolicSequence[6, b, 0]

Out[215]=
      {{b0}, {b1}, {b2}, {b3}, {b4}, {b5}}

In[216]:=
      mxSymbSeq = MultiplexSequence[symbSeq, symbSeqTwo]
      % // MatrixForm

Out[216]=
      {{a0, b0}, {a1, b1}, {a2, b2}, {a3, b3}, {a4, b4}, {a5, b5}}

Out[217]//MatrixForm=
```

$$\begin{pmatrix} a0 & b0 \\ a1 & b1 \\ a2 & b2 \\ a3 & b3 \\ a4 & b4 \\ a5 & b5 \end{pmatrix}$$

```
In[218]:=
      SequenceFourierTransform[mxSymbSeq]
      % // MatrixForm
Out[218]=
```

$\{a0 + a1\,e^{-2 i f \pi} + a2\,e^{-4 i f \pi} + a3\,e^{-6 i f \pi} + a4\,e^{-8 i f \pi} + a5\,e^{-10 i f \pi},\ b0 + b1\,e^{-2 i f \pi} + b2\,e^{-4 i f \pi} + b3\,e^{-6 i f \pi} + b4\,e^{-8 i f \pi} + b5\,e^{-10 i f \pi}\}$

```
Out[219]//MatrixForm=
```

$$\begin{pmatrix} a0 + a1\,e^{-2 i f \pi} + a2\,e^{-4 i f \pi} + a3\,e^{-6 i f \pi} + a4\,e^{-8 i f \pi} + a5\,e^{-10 i f \pi} \\ b0 + b1\,e^{-2 i f \pi} + b2\,e^{-4 i f \pi} + b3\,e^{-6 i f \pi} + b4\,e^{-8 i f \pi} + b5\,e^{-10 i f \pi} \end{pmatrix}$$

`SequenceFourierTransform` computes DTFT of samples

$\{x_0, x_1, \ldots, x_n, \ldots, x_{N-1}\}$

according to the formula

$X(f) = \sum_{n=0}^{N-1} x_n\, w^{-n}$, where $w = e^{i 2 \pi f}$.

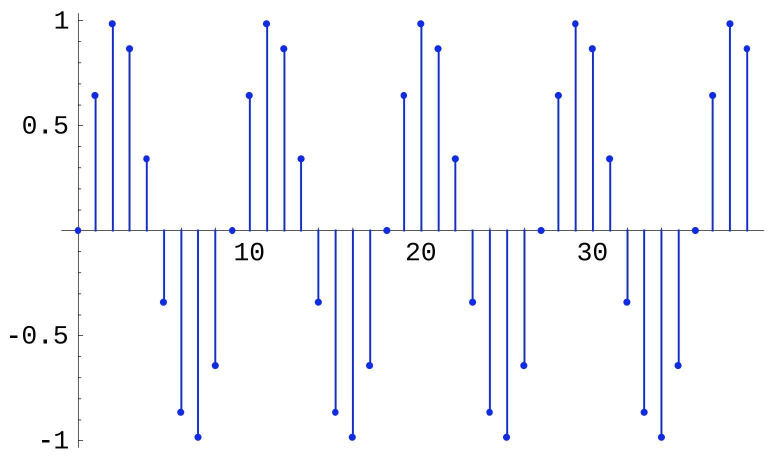
It is assumed that the first sample index is equal to zero.

`SequenceFourierTransformMagnitudePlot` computes the normalized DTFT as $X_{\text{norm}}(f) = \frac{1}{N} X(f)$.

## Discrete Fourier Transform (DFT)

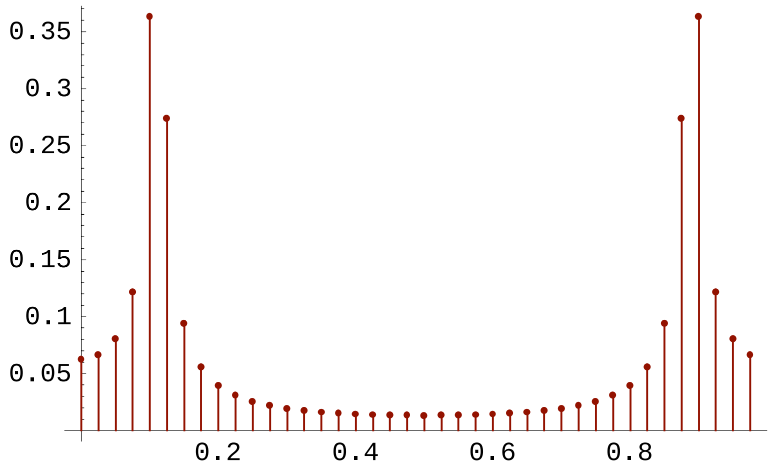
Consider a sinusoidal sequence, of amplitude 1, of digital frequency $\frac{1}{9}$, and of zero initial phase, represented by 40 samples:

```
In[220]:=
      Needs["SchematicSolver`"];

In[221]:=
      amplitude = 1;
      sineDigitalFreq = 1 / 9;
      initialPhase = 0;
      numberOfSamples = 40;

In[225]:=
      sineSeq = amplitude * UnitSineSequence[numberOfSamples, sineDigitalFreq, initialPhase];

In[226]:=
      SequencePlot[sineSeq];
```



`SequenceDiscreteFourierTransformMagnitudePlot` plots the Discrete Fourier Transform magnitude of the sequence `sineSeq`.

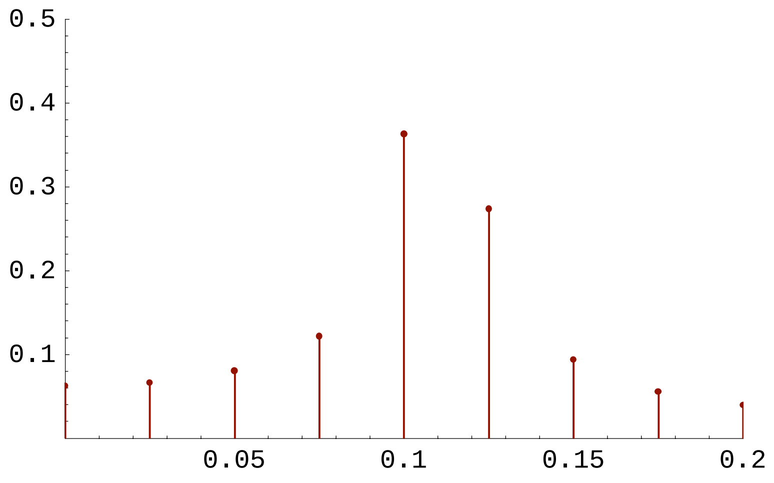
```
In[227]:=
        SequenceDiscreteFourierTransformMagnitudePlot[sineSeq];
```



The magnitude plot shows peaks that correspond to `sineDigitalFreq`.

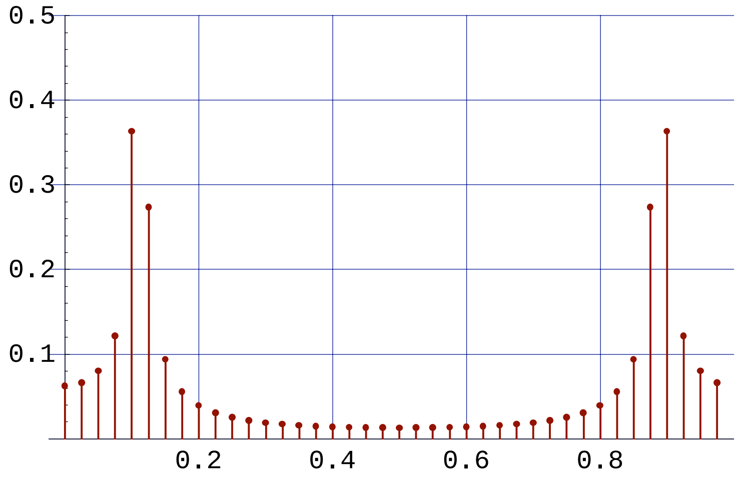
`SequenceDiscreteFourierTransformMagnitudePlot` plots magnitude over the frequency range from 0 to 1. You can change the plot range to zoom the peaks:

```
In[228]:=
        SequenceDiscreteFourierTransformMagnitudePlot[sineSeq,
         PlotRange → {{0, 0.2}, {0, 0.5}}];
```



In addition, you can change the plot range and add grid lines to refine the graphic:

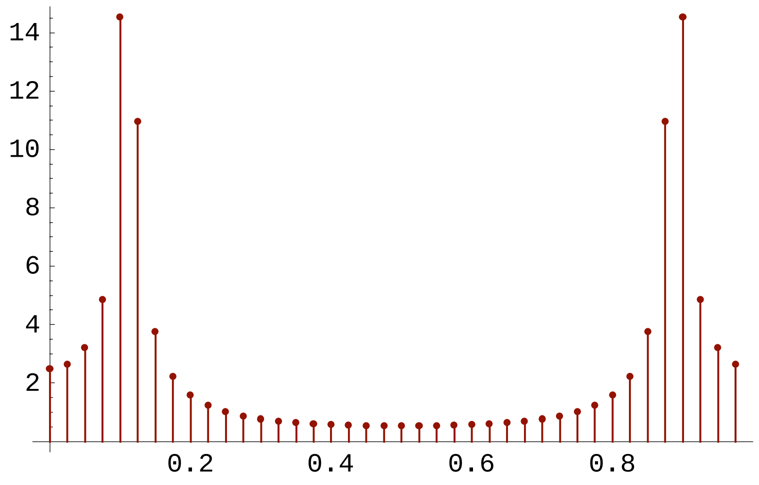
```
In[229]:=
      SequenceDiscreteFourierTransformMagnitudePlot[sineSeq,
        PlotRange → {Automatic, {0, 0.5}},
        GridLines → Automatic];
```



By default, `SequenceDiscreteFourierTransformMagnitudePlot` plots normalized magnitude. The *normalized magnitude* is the DFT magnitude divided by the number of samples.

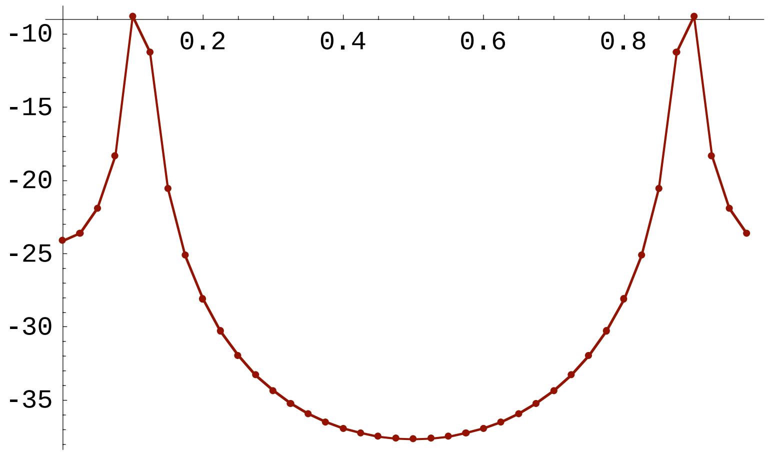
Here is the magnitude plot without normalization:

```
In[230]:=
      SequenceDiscreteFourierTransformMagnitudePlot[sineSeq,
        NormalizedSpectrum → False];
```



`SequenceDiscreteFourierTransformMagnitudePlot` plots the Discrete Fourier Transform (DFT) magnitude in linear scale. You can plot the magnitude in decibels (dB) with

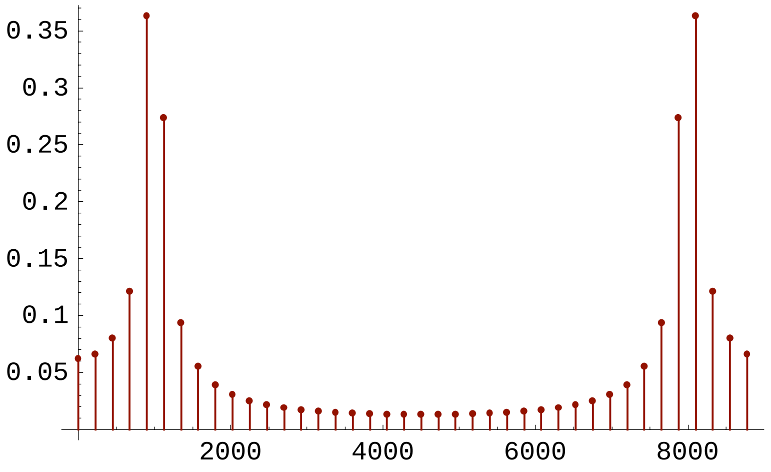
```
In[231]:=
      SequenceDiscreteFourierTransformMagnitudePlot[sineSeq,
       PlotJoined → True, StemPlot → False,
       dBMagnitudePlot → True];
```



Assume that the sinusoidal sequence was obtained by sampling a continuous-time sine signal, of the frequency 1000 Hz, at the rate $F_{\text{samp}} = 9000$ Hz.

`SequenceDiscreteFourierTransformMagnitudePlot` can plot the magnitude versus the continuous-time frequency with

```
In[232]:=
      SequenceDiscreteFourierTransformMagnitudePlot[sineSeq,
       SequenceSamplingFrequency → 9000];
```

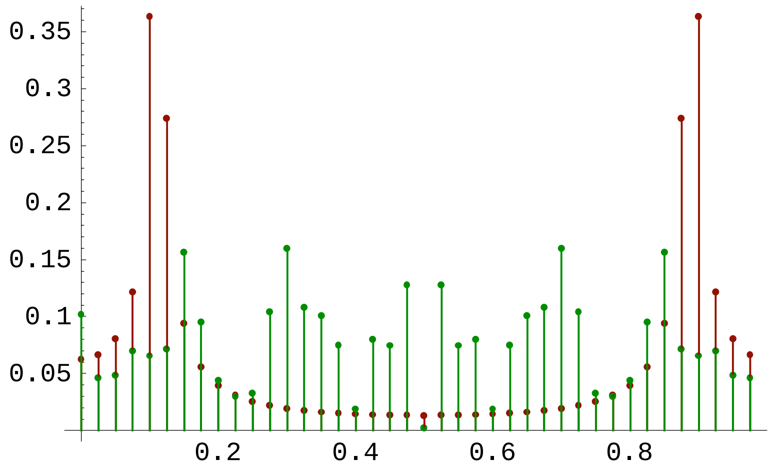


You can plot DFT magnitudes of several sequences:

```
In[233]:=
      noiseSeq = UnitNoiseSequence[numberOfSamples];
```

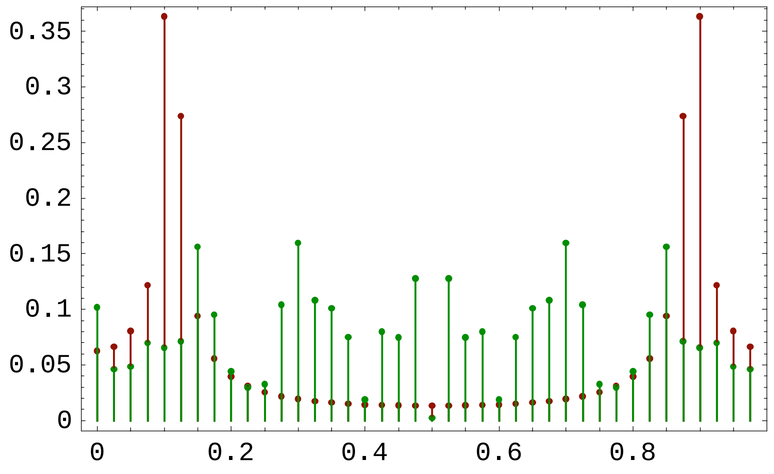
```
In[234]:=
      sineNoiseSeq = MultiplexSequence[sineSeq, noiseSeq];
```

```
In[235]:=
        SequenceDiscreteFourierTransformMagnitudePlot[sineNoiseSeq];
```

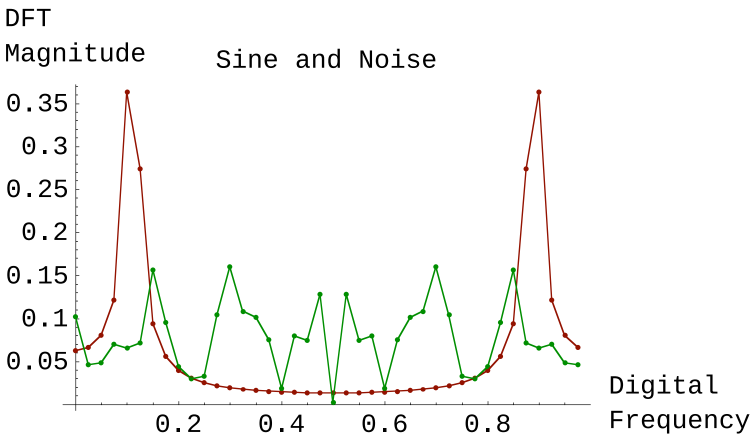


Pass any `Graphics` option to refine the graphic:

```
In[236]:=
        SequenceDiscreteFourierTransformMagnitudePlot[sineNoiseSeq,
          Frame → True];
```



```
In[237]:=
      SequenceDiscreteFourierTransformMagnitudePlot[sineNoiseSeq,
        PlotLabel → "Sine and Noise",
        AxesLabel → {"Digital\nFrequency", "DFT\nMagnitude"},
        PlotJoined → True, StemPlot → False];
```



`SequenceDiscreteFourierTransform` computes the Discrete Fourier Transform (DFT) of a numeric sequence. Here is a sinusoidal sequence of 10 samples:

```
In[238]:=
      sineSeq10 = UnitSineSequence[10, sineDigitalFreq]
```

Out[238]=

$$\{\{0\}, \{\mathrm{Sin}[\tfrac{2\pi}{9}]\}, \{\mathrm{Sin}[\tfrac{4\pi}{9}]\}, \{\tfrac{\sqrt{3}}{2}\}, \{\mathrm{Sin}[\tfrac{8\pi}{9}]\}, \{\mathrm{Sin}[\tfrac{10\pi}{9}]\}, \{-\tfrac{\sqrt{3}}{2}\}, \{\mathrm{Sin}[\tfrac{14\pi}{9}]\}, \{\mathrm{Sin}[\tfrac{16\pi}{9}]\}, \{0\}\}$$

```
In[239]:=
      SequenceDiscreteFourierTransform[sineSeq10]
```

Out[239]=

$$\{\{-2.22045\times10^{-16}+0.\,i\}, \{1.42837-4.39607\,i\}, \{-0.485918+0.668808\,i\}, \{-0.391336+0.284322\,i\}, \{-0.369133+0.119939\,i\}, \{-0.36397+0.\,i\}, \{-0.369133-0.119939\,i\}, \{-0.391336-0.284322\,i\}, \{-0.485918-0.668808\,i\}, \{1.42837+4.39607\,i\}\}$$

`SequenceDiscreteFourierTransform` finds DFT of a multiplex sequence as a matrix of spectral components. Here is a noise sequence of 10 samples:

```
In[240]:=
      noiseSeq10 = UnitNoiseSequence[10]

Out[240]=
      {{-0.144899}, {-0.424729}, {0.605028}, {0.151323}, {0.713948},
       {-0.093818}, {-0.728581}, {-0.433926}, {0.345475}, {-0.376499}}
```

You can form a multiplex sequence from `sineSeq10` and `noiseSeq10`:

*In[241]:=*
```
mxSeq10 = MultiplexSequence[sineSeq10, noiseSeq10]
```
*Out[241]=*

$$\left\{\{0, -0.144899\}, \left\{\mathrm{Sin}\left[\frac{2\pi}{9}\right], -0.424729\right\}, \left\{\mathrm{Sin}\left[\frac{4\pi}{9}\right], 0.605028\right\}, \left\{\frac{\sqrt{3}}{2}, 0.151323\right\}, \left\{\mathrm{Sin}\left[\frac{8\pi}{9}\right], 0.713948\right\}, \left\{\mathrm{Sin}\left[\frac{10\pi}{9}\right], -0.093818\right\}, \left\{-\frac{\sqrt{3}}{2}, -0.728581\right\}, \left\{\mathrm{Sin}\left[\frac{14\pi}{9}\right], -0.433926\right\}, \left\{\mathrm{Sin}\left[\frac{16\pi}{9}\right], 0.345475\right\}, \{0, -0.376499\}\right\}$$

The multiplex sequence is a matrix of samples. Each column of `mxSeq10` corresponds to a discrete signal:

*In[242]:=*
```
mxSeq10 // TraditionalForm
```
*Out[242]//TraditionalForm=*

$$\begin{pmatrix} 0 & -0.144899 \\ \sin(\frac{2\pi}{9}) & -0.424729 \\ \sin(\frac{4\pi}{9}) & 0.605028 \\ \frac{\sqrt{3}}{2} & 0.151323 \\ \sin(\frac{8\pi}{9}) & 0.713948 \\ \sin(\frac{10\pi}{9}) & -0.093818 \\ -\frac{\sqrt{3}}{2} & -0.728581 \\ \sin(\frac{14\pi}{9}) & -0.433926 \\ \sin(\frac{16\pi}{9}) & 0.345475 \\ 0 & -0.376499 \end{pmatrix}$$

Here is the DFT of the multiplex sequence:

*In[243]:=*
```
mxSeq10DFT = SequenceDiscreteFourierTransform[mxSeq10]
```
*Out[243]=*
```
{{-2.22045×10^-16 + 0. ⅈ, -0.386679 + 0. ⅈ},
 {1.42837 - 4.39607 ⅈ, -0.3064 - 1.623 ⅈ}, {-0.485918 + 0.668808 ⅈ, -1.03117 + 1.60924 ⅈ},
 {-0.391336 + 0.284322 ⅈ, -0.805614 - 0.829495 ⅈ},
 {-0.369133 + 0.119939 ⅈ, 0.627721 + 0.566491 ⅈ},
 {-0.36397 + 0. ⅈ, 1.96862 + 0. ⅈ}, {-0.369133 - 0.119939 ⅈ, 0.627721 - 0.566491 ⅈ},
 {-0.391336 - 0.284322 ⅈ, -0.805614 + 0.829495 ⅈ},
 {-0.485918 - 0.668808 ⅈ, -1.03117 - 1.60924 ⅈ}, {1.42837 + 4.39607 ⅈ, -0.3064 + 1.623 ⅈ}}
```

DFT of the multiplex sequence is a matrix of spectral components. Each column of `mxSeq10DFT` corresponds to a discrete signal:

```
In[244]:=
        mxSeq10DFT // TraditionalForm

Out[244]//TraditionalForm=
```

$$\begin{pmatrix} -2.22045\times 10^{-16}+0.\,i & -0.386679+0.\,i \\ 1.42837-4.39607\,i & -0.3064-1.623\,i \\ -0.485918+0.668808\,i & -1.03117+1.60924\,i \\ -0.391336+0.284322\,i & -0.805614-0.829495\,i \\ -0.369133+0.119939\,i & 0.627721+0.566491\,i \\ -0.36397+0.\,i & 1.96862+0.\,i \\ -0.369133-0.119939\,i & 0.627721-0.566491\,i \\ -0.391336-0.284322\,i & -0.805614+0.829495\,i \\ -0.485918-0.668808\,i & -1.03117-1.60924\,i \\ 1.42837+4.39607\,i & -0.3064+1.623\,i \end{pmatrix}$$

`SequenceDiscreteFourierTransform` computes DFT of samples

$\{x_0, x_1, \ldots, x_n, \ldots, x_{N-1}\}$

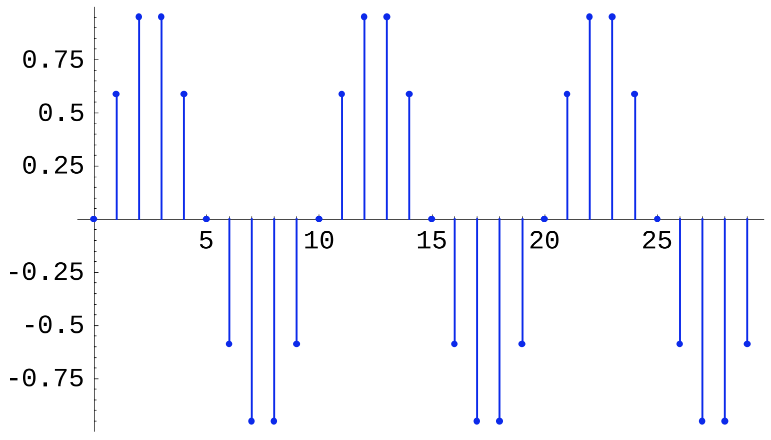
as spectral components $\{X_0, X_1, \ldots, X_k, \ldots, X_{N-1}\}$,

where $X_k = \sum_{n=0}^{N-1} x_n\, w^{-k\,n}$, $w = e^{i\,\frac{2\pi}{N}}$.

`SequenceDiscreteFourierTransformMagnitudePlot` computes the normalized DFT as $X_{k,\text{norm}} = \frac{1}{N}\, X_k$.

You can use `SequenceDiscreteFourierTransformMagnitudePlot` to demonstrate a well know feature of sinusoidal sequences: if the number of samples is an integer multiple of the digital period, then DFT has only two nonzero components.

```
In[245]:=
        digitalPeriod = 10;
        numberOfSamples = 3 * digitalPeriod;
        sineDigitalFreq = 1 / digitalPeriod;

In[248]:=
        sineSeq30 = UnitSineSequence[numberOfSamples, sineDigitalFreq];
        % // SequencePlot;
```



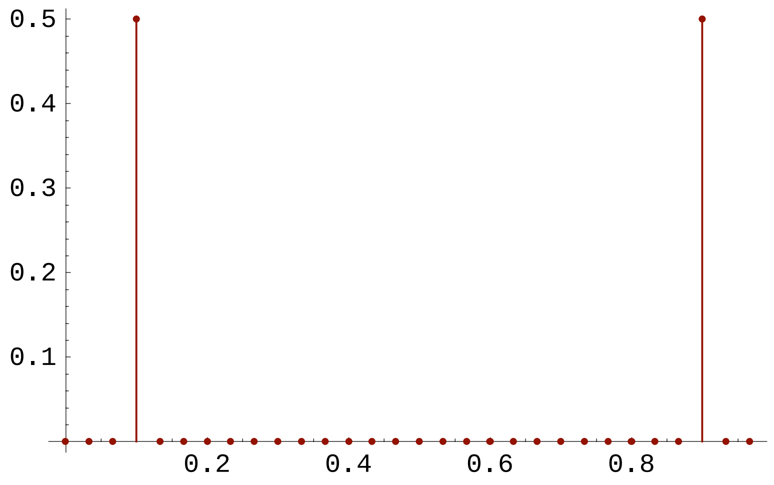
```
In[250]:=
      sineSeq30DFT = SequenceDiscreteFourierTransform[sineSeq30]

Out[250]=
      {{0. + 0. i}, {0. + 0. i}, {0. + 0. i}, {6.93239 × 10^-16 - 15. i},
       {0. + 0. i}, {0. + 0. i}, {8.74563 × 10^-17 - 3.21791 × 10^-16 i}, {0. + 0. i},
       {0. + 0. i}, {-8.74563 × 10^-17 - 3.21791 × 10^-16 i}, {0. + 0. i},
       {0. + 0. i}, {-6.93239 × 10^-16 + 4.44089 × 10^-16 i}, {0. + 0. i}, {0. + 0. i},
       {0. + 0. i}, {0. + 0. i}, {0. + 0. i}, {-6.93239 × 10^-16 - 4.44089 × 10^-16 i},
       {0. + 0. i}, {0. + 0. i}, {-8.74563 × 10^-17 + 3.21791 × 10^-16 i},
       {0. + 0. i}, {0. + 0. i}, {8.74563 × 10^-17 + 3.21791 × 10^-16 i}, {0. + 0. i},
       {0. + 0. i}, {6.93239 × 10^-16 + 15. i}, {0. + 0. i}, {0. + 0. i}}
```

`SequenceDiscreteFourierTransform` may return numbers that are very close to zero, and you may well want to *assume* that the numbers should be exactly zero. The function `Chop` allows you to replace approximate real numbers that are close to zero by the exact integer `0`.

```
In[251]:=
      sineSeq30DFT // Chop

Out[251]=
      {{0}, {0}, {0}, {-15. i}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0},
       {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {15. i}, {0}, {0}}
```

Note that there are exactly two non-zero spectral components.

```
In[252]:=
      SequenceDiscreteFourierTransformMagnitudePlot[sineSeq30];
```



The non-zero spectral components occur at

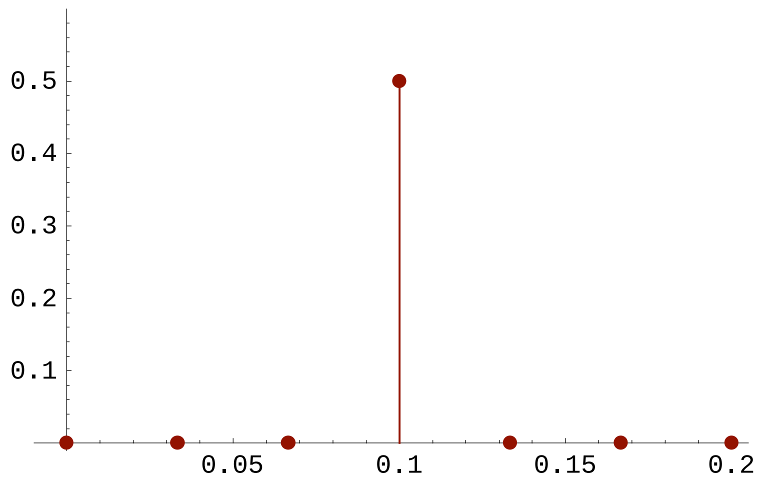
`sineDigitalFreq`

and at

`1-sineDigitalFreq`.

The magnitude of the peaks is half the amplitude of the sinusoidal sequence.

```
In[253]:=
      SequenceDiscreteFourierTransformMagnitudePlot[
        sineSeq30, PlotRange → {{-0.01, 0.205}, {-0.01, 0.6}},
        SequencePointSize → 0.02];
```



`SequencePointSize`→*p* sets relative point size to *p*, $0 < p < 1$. Use this option to plot larger dots that correspond to spectral components.

# 4. Solving Systems

## 4.1. Continuous-Time Systems

### Introduction

Using *SchematicSolver*'s schematic capabilities, symbolic system analysis and signal processing, you can perform fast and accurate simulations of continuous-time (analog) systems.

*Linear time-invariant* (LTI) systems, characterized by linear constant-coefficient differential equations, are efficiently analyzed by using the *Laplace* transform. The transform maps the differential equations into algebraic equations which are easier to manipulate. This section illustrates step-by-step procedures for analyzing LTI systems in the transform domain. For the given block-diagram of a system, the required equations are formulated and mapped by the transform into a system of algebraic equations. The set of algebraic equations is solved to find the system response in the transform domain. Next, by the inverse transform, the system response is computed as a continuous-time function.

`ContinuousSystemEquations`, `ContinuousSystemResponse`, `ContinuousSystemSignals`, and `ContinuousSystemTransferFunction` are *SchematicSolver*'s functions that solve systems described by lists of element specifications. The functions can take only one argument, the system specification, and return the corresponding solution.

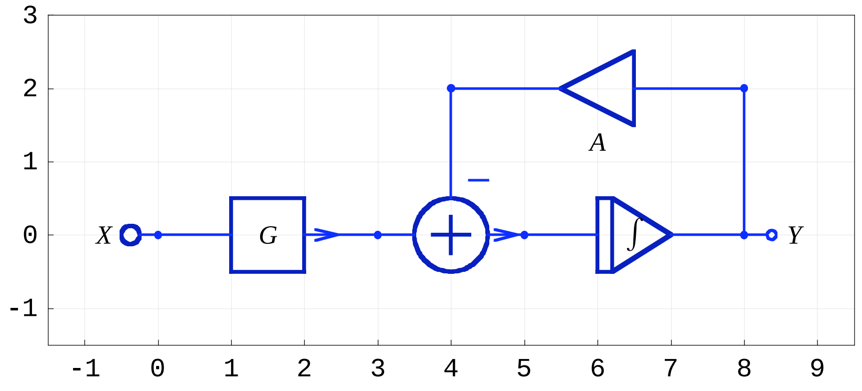
```
In[1]:= Needs["SchematicSolver`"]
```

Consider a simple system specified by the following list:

```
In[2]:= mySystem = {
          {"Input", {0, 0}, X},
          {"Block", {{0, 0}, {3, 0}}, G},
          {"Adder", {{3, 0}, {4, -2}, {5, 0}, {4, 2}}, {1, 0, 2, -1}},
          {"Integrator", {{5, 0}, {8, 0}}, 1},
          {"Amplifier", {{8, 2}, {4, 2}}, A},
          {"Line", {{8, 0}, {8, 2}}},
          {"Output", {8, 0}, Y}
        };
```

`ShowSchematic` shows the block diagram of the system:

```
In[3]:=  ShowSchematic[mySystem, PlotRange → {{-1.5, 9.5}, {-1.5, 3}}];
```



## System Equations

`ContinuousSystemEquations` sets up the equations directly from the schematic:

```
In[4]:=  myEquations = ContinuousSystemEquations[mySystem]
```

Out[4]= $\{\{Y[\{0,0\}] == X,\ Y[\{3,0\}] == G\,Y[\{0,0\}],\ Y[\{5,0\}] == Y[\{3,0\}] - Y[\{4,2\}],\ Y[\{8,0\}] == \frac{Y[\{5,0\}]}{s},\ Y[\{4,2\}] == A\,Y[\{8,0\}]\},\ \{Y[\{8,0\}], Y[\{5,0\}], Y[\{4,2\}], Y[\{3,0\}], Y[\{0,0\}]\}\}$

`ContinuousSystemEquations` returns a list of the form {*systemEquations*, *systemVariables*}. The first item, *systemEquations*, is a list of equations describing the system:

```
In[5]:=  First[myEquations] // ColumnForm
```

Out[5]=
$$\begin{array}{l} Y[\{0,0\}] == X \\ Y[\{3,0\}] == G\,Y[\{0,0\}] \\ Y[\{5,0\}] == Y[\{3,0\}] - Y[\{4,2\}] \\ Y[\{8,0\}] == \frac{Y[\{5,0\}]}{s} \\ Y[\{4,2\}] == A\,Y[\{8,0\}] \end{array}$$

The last item, *systemVariables*, is a list of symbols that represent transforms of signals at nodes:

```
In[6]:=  Last[myEquations]

Out[6]=  {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}
```

The symbol **`Y[{0,0}]`** denotes the signal at the system input and **`Y[{8,0}]`** stands for the signal at the system output.

By default, **`Y[{k,n}]`** designates a signal, in the transform domain, at node with coordinates **`{k,n}`**. The default symbol for the complex frequency is **`s`**.

## System Response

`ContinuousSystemResponse` finds the response of the system directly from the schematic:

```
In[7]:=  myResponse = ContinuousSystemResponse[mySystem]
```

Out[7]= $\{\{Y[\{5,0\}] \to \frac{G\,s\,X}{A+s},\ Y[\{8,0\}] \to \frac{G\,X}{A+s},\ Y[\{4,2\}] \to \frac{A\,G\,X}{A+s},\ Y[\{3,0\}] \to G\,X,\ Y[\{0,0\}] \to X\},\ \{Y[\{8,0\}], Y[\{5,0\}], Y[\{4,2\}], Y[\{3,0\}], Y[\{0,0\}]\}\}$

`ContinuousSystemResponse` returns a list of the form {*systemResponse*, *systemVariables*}. The first item, *system-Response*, is a list of replacement rules describing the system response:

```
In[8]:=  First[myResponse] // ColumnForm
```

Out[8]= $Y[\{5,0\}] \to \frac{G\,s\,X}{A+s}$
$Y[\{8,0\}] \to \frac{G\,X}{A+s}$
$Y[\{4,2\}] \to \frac{A\,G\,X}{A+s}$
$Y[\{3,0\}] \to G\,X$
$Y[\{0,0\}] \to X$

The last item, *systemVariables*, is a list of symbols that represent transforms of signals at nodes:

```
In[9]:=  Last[myResponse]

Out[9]=  {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}
```

## System Signals

`ContinuousSystemSignals` finds the transforms of signals at all nodes of the system directly from the schematic:

```
In[10]:=  mySignals = ContinuousSystemSignals[mySystem]
```

Out[10]= $\left\{\left\{\frac{G\,X}{A+s}, \frac{G\,s\,X}{A+s}, \frac{A\,G\,X}{A+s}, G\,X, X\right\}, \{Y[\{8,0\}], Y[\{5,0\}], Y[\{4,2\}], Y[\{3,0\}], Y[\{0,0\}]\}\right\}$

`ContinuousSystemSignals` returns a list of the form {*systemSignals*, *systemVariables*}. The first item, *systemSig-nals*, is a list of expressions representing the transforms of signals at all nodes of the system:

```
In[11]:=  First[mySignals] // ColumnForm
```

Out[11]= $\frac{G\,X}{A+s}$
$\frac{G\,s\,X}{A+s}$
$\frac{A\,G\,X}{A+s}$
$G\,X$
$X$

The last item, *systemVariables*, is a list of corresponding symbols that represent transforms of signals at nodes:

```
In[12]:=  Last[mySignals]

Out[12]=  {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}
```

The following table shows the signal expressions and the corresponding names:

```
In[13]:=  TableForm[Transpose[mySignals]]
```

Out[13]//TableForm=

| | |
|---|---|
| $\frac{G\,X}{A+s}$ | Y[{8, 0}] |
| $\frac{G\,s\,X}{A+s}$ | Y[{5, 0}] |
| $\frac{A\,G\,X}{A+s}$ | Y[{4, 2}] |
| $G\,X$ | Y[{3, 0}] |
| $X$ | Y[{0, 0}] |

## Transfer Function

`ContinuousSystemTransferFunction` finds the transfer function directly from the schematic:

```
In[14]:= {myTransferFunction, systemInp, systemOut} =
          ContinuousSystemTransferFunction[mySystem]
```

Out[14]= $\left\{\left\{\left\{\frac{G}{A+s}\right\}\right\}, \{Y[\{0, 0\}]\}, \{Y[\{8, 0\}]\}\right\}$

`ContinuousSystemTransferFunction` returns a list of the form {*transferFunction*, *systemInput*, *systemOutput*}. The first item, *transferFunction*, is the transfer function matrix of the system:

```
In[15]:= myTransferFunction
```

Out[15]= $\left\{\left\{\frac{G}{A+s}\right\}\right\}$

The second item, *systemInput*, is the symbol that represents the system input:

```
In[16]:= systemInp

Out[16]= {Y[{0, 0}]}
```

The last item, *systemOutput*, is the symbol that represents the system output:

```
In[17]:= systemOut

Out[17]= {Y[{8, 0}]}
```

## Frequency Response

For specific values of system parameters, you can plot the magnitude response:

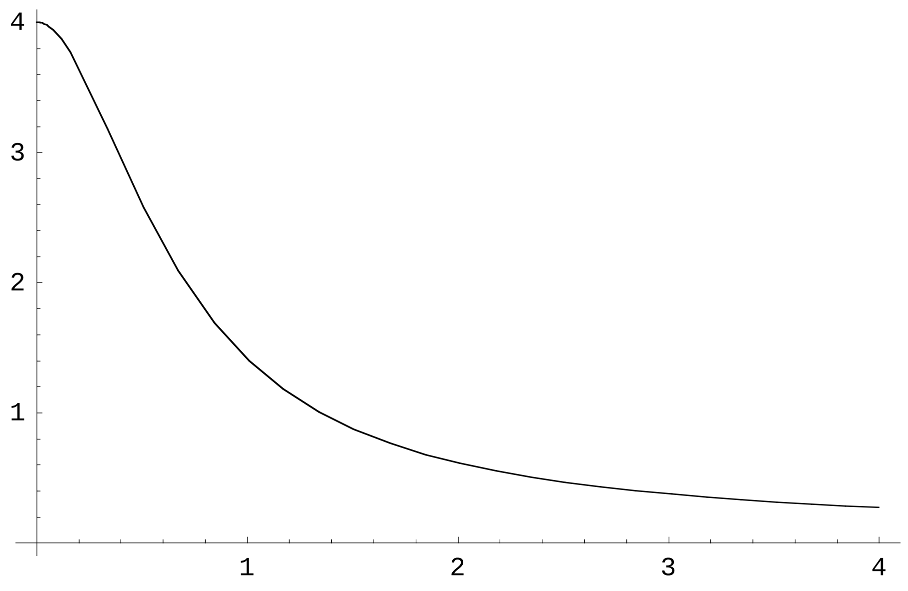
```
In[18]:= myValues = {A → 0.5, G → (s + 2) / (s + 1)}
```

Out[18]= $\left\{A \to 0.5,\ G \to \frac{2+s}{1+s}\right\}$

```
In[19]:= myTF = myTransferFunction[[1, 1]] /. myValues
```

Out[19]= $\frac{2+s}{(0.5+s)\ (1+s)}$

```
In[20]:= Plot[Abs[myTF /. s → I * w], {w, 0, 4}];
```



*SchematicSolver* provides symbolic solution to the system: the response and the transfer function are closed-form expressions in terms of the system parameters, kept as symbols, and the complex frequency.

## Impulse and Step Response

*Impulse response* of a continuous-time system is a continuous-time function. It can be computed as the inverse Laplace transform of the system transfer function returned by *SchematicSolver*:

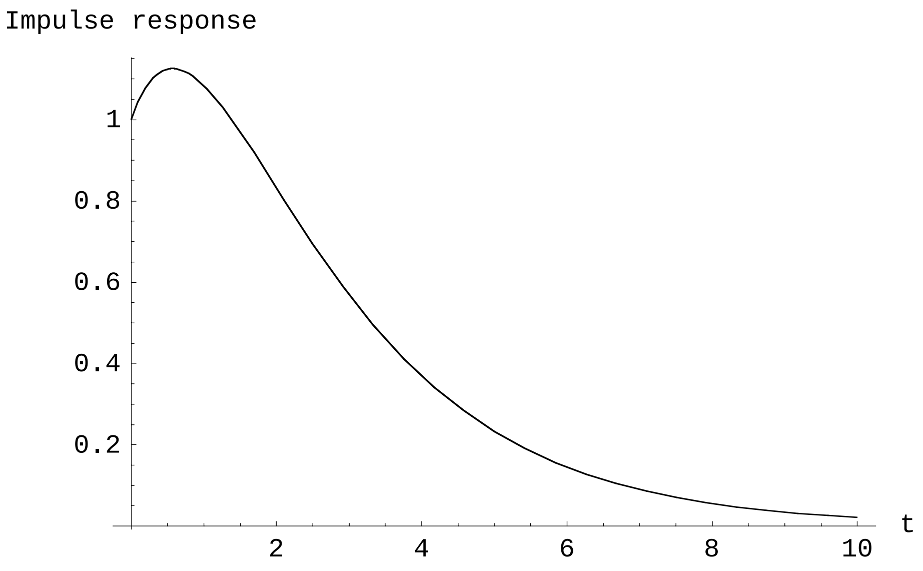
```
In[21]:=  myTF
```

Out[21]= $\frac{2+s}{(0.5+s)\ (1+s)}$

```
In[22]:=  impulseResponse = InverseLaplaceTransform[myTF, s, t]
```

Out[22]= $-2.\ e^{-1.\ t}+3.\ e^{-0.5\ t}$

```
In[23]:=  Plot[impulseResponse, {t, 0, 10},
            PlotRange → All, AxesLabel → {"t", "Impulse response"}];
```



*Step response* of a continuous-time system is a continuous-time function. It can be computed as the inverse Laplace transform of myTF/s:

In[24]:= $\frac{\text{myTF}}{s}$

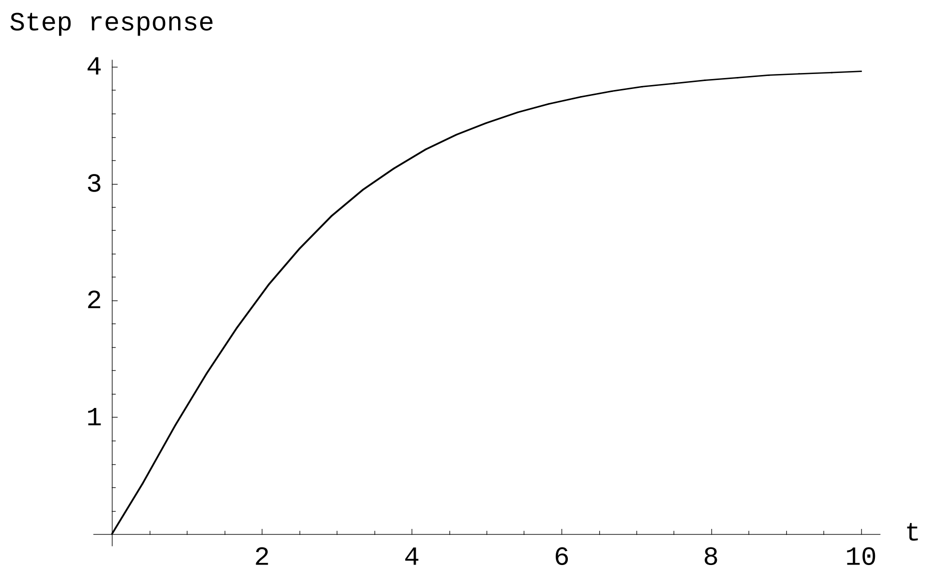
Out[24]= $\frac{2+s}{s\ (0.5+s)\ (1+s)}$

In[25]:= stepResponse = InverseLaplaceTransform$\left[\frac{\text{myTF}}{s}, s, t\right]$

Out[25]= $4.+2.\ e^{-1.\ t}-6.\ e^{-0.5\ t}$

Here is the plot of the step response:

```
In[26]:= Plot[stepResponse, {t, 0, 10}, PlotRange → All, AxesLabel → {"t", "Step response"}];
```



## Time-Domain Response

Time-domain response of linear time-invariant systems can be found by using the *Laplace transform*. Assume a sinusoidal stimulus

```
In[27]:= stimulus = 10 * Sin[2 * t];
```

Find the Laplace transform of the stimulus:

```
In[28]:= stimulusLT = LaplaceTransform[stimulus, t, s]
```

Out[28]= $\frac{20}{4+s^2}$

Next, by the inverse transform of the product `stimulusLT*myTF`, compute the response:

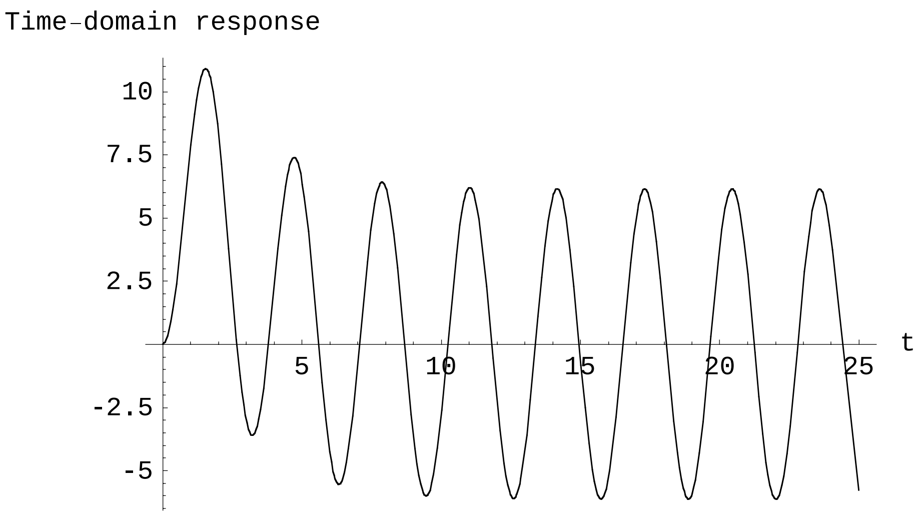
```
In[29]:= responseLT = myTF * stimulusLT
```

Out[29]= $\frac{20\,(2+s)}{(0.5+s)\,(1+s)\,(4+s^2)}$

```
In[30]:= response = InverseLaplaceTransform[responseLT, s, t] // Re // ComplexExpand
```

Out[30]= $-8.\,e^{-1.\,t}+14.1176\,e^{-0.5\,t}-6.11765\,\mathrm{Cos}[2.\,t]-0.470588\,\mathrm{Sin}[2.\,t]$

```
In[31]:= Plot[response, {t, 0, 25},
           PlotRange → All, AxesLabel → {"t", "Time-domain response"}];
```



## 4.2. Discrete-Time Systems

### Introduction

Using *SchematicSolver*'s schematic capabilities, symbolic system analysis and signal processing, you can perform fast and accurate simulations of discrete-time (digital) systems.

*Linear time-invariant* (LTI) systems, characterized by linear constant-coefficient difference equations, are efficiently analyzed by using the *z*-transform. The transform maps the difference equations into algebraic equations which are easier to manipulate. This section illustrates step-by-step procedures for analyzing LTI systems in the transform domain. For the given block-diagram of a system, the required equations are formulated and mapped by the transform into a system of algebraic equations. The set of algebraic equations is solved to find the system response in the transform domain. Next, by the inverse transform, the system response is computed as a discrete function.

`DiscreteSystemEquations`, `DiscreteSystemResponse`, `DiscreteSystemSignals`, and `Discrete-SystemTransferFunction` are *SchematicSolver*'s functions that solve systems described by lists of element specifications. The functions can take only one argument, the system specification, and return the corresponding solution.

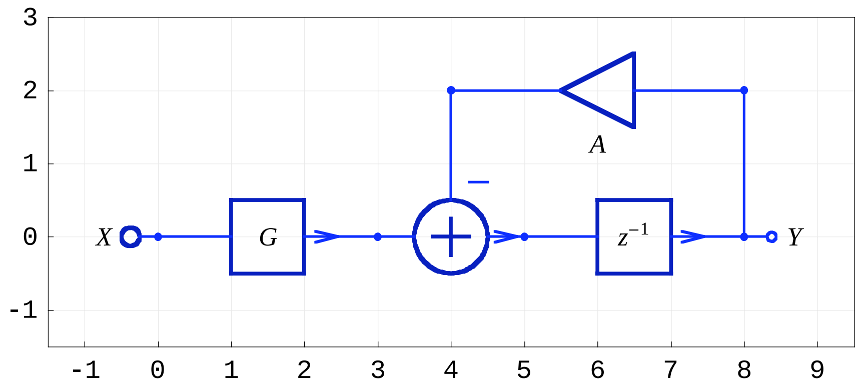
```
In[32]:= Needs["SchematicSolver`"]
```

Consider a simple system specified by the following list:

```
In[33]:= mySystem = {
            {"Input", {0, 0}, X, ""},
            {"Block", {{0, 0}, {3, 0}}, G, ""},
            {"Adder", {{3, 0}, {4, -2}, {5, 0}, {4, 2}}, {1, 0, 2, -1}, ""},
            {"Delay", {{5, 0}, {8, 0}}, 1, ""},
            {"Multiplier", {{8, 2}, {4, 2}}, A, ""},
            {"Line", {{8, 0}, {8, 2}}},
            {"Output", {8, 0}, Y, ""}
          };
```

`ShowSchematic` shows the block diagram of the system:

```
In[34]:=  ShowSchematic[mySystem, PlotRange → {{-1.5, 9.5}, {-1.5, 3}}];
```



## System Equations

`DiscreteSystemEquations` sets up the equations directly from the schematic:

```
In[35]:=  myEquations = DiscreteSystemEquations[mySystem]

Out[35]=  {{Y[{0, 0}] == X, Y[{3, 0}] == G Y[{0, 0}],
            Y[{5, 0}] == Y[{3, 0}] - Y[{4, 2}], Y[{8, 0}] == Y[{5, 0}]/z, Y[{4, 2}] == A Y[{8, 0}]},
           {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}}
```

`DiscreteSystemEquations` returns a list of the form {*systemEquations*, *systemVariables*}. The first item, *systemEquations*, is a list of equations describing the system:

```
In[36]:=  First[myEquations] // ColumnForm

Out[36]=  Y[{0, 0}] == X
          Y[{3, 0}] == G Y[{0, 0}]
          Y[{5, 0}] == Y[{3, 0}] - Y[{4, 2}]
          Y[{8, 0}] == Y[{5,0}]/z
          Y[{4, 2}] == A Y[{8, 0}]
```

The last item, *systemVariables*, is a list of symbols that represent transforms of signals at nodes:

```
In[37]:=  Last[myEquations]

Out[37]=  {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}
```

The symbol **`Y[{0,0}]`** denotes the signal at the system input and **`Y[{8,0}]`** stands for the signal at the system output.

By default, **`Y[{i,j}]`** designates a signal, in the transform domain, at node with coordinates **`{i,j}`**. The default symbol for the complex variable is **`z`**.

## System Response

`DiscreteSystemResponse` finds the response of the system directly from the schematic:

```
In[38]:=  myResponse = DiscreteSystemResponse[mySystem]

Out[38]=  {{Y[{5, 0}] → G X z/(A + z), Y[{8, 0}] → G X/(A + z), Y[{4, 2}] → A G X/(A + z), Y[{3, 0}] → G X, Y[{0, 0}] → X},
           {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}}
```

`DiscreteSystemResponse` returns a list of the form {*systemResponse*, *systemVariables*}. The first item, *systemResponse*, is a list of replacement rules describing the system response:

```
In[39]:= First[myResponse] // ColumnForm
```

```
Out[39]= Y[{5, 0}] → (G X z)/(A+z)
         Y[{8, 0}] → (G X)/(A+z)
         Y[{4, 2}] → (A G X)/(A+z)
         Y[{3, 0}] → G X
         Y[{0, 0}] → X
```

The last item, *systemVariables*, is a list of symbols that represent transforms of signals at nodes:

```
In[40]:= Last[myResponse]
```

```
Out[40]= {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}
```

## System Signals

`DiscreteSystemSignals` finds the transforms of signals at all nodes of the system directly from the schematic:

```
In[41]:= mySignals = DiscreteSystemSignals[mySystem]
```

```
Out[41]= {{(G X)/(A + z), (G X z)/(A + z), (A G X)/(A + z), G X, X}, {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}}
```

`DiscreteSystemSignals` returns a list of the form {*systemSignals*, *systemVariables*}. The first item, *systemSignals*, is a list of expressions representing the transforms of signals at all nodes of the system:

```
In[42]:= First[mySignals] // ColumnForm
```

```
Out[42]= (G X)/(A+z)
         (G X z)/(A+z)
         (A G X)/(A+z)
         G X
         X
```

The last item, *systemVariables*, is a list of corresponding symbols that represent transforms of signals at nodes:

```
In[43]:= Last[mySignals]
```

```
Out[43]= {Y[{8, 0}], Y[{5, 0}], Y[{4, 2}], Y[{3, 0}], Y[{0, 0}]}
```

The following table shows the signal expressions and the corresponding names:

```
In[44]:= TableForm[Transpose[mySignals]]
```

```
   Out[44]//TableForm=
       (G X)/(A+z)        Y[{8, 0}]
       (G X z)/(A+z)      Y[{5, 0}]
       (A G X)/(A+z)      Y[{4, 2}]
       G X                Y[{3, 0}]
       X                  Y[{0, 0}]
```

## Transfer Function

`DiscreteSystemTransferFunction` finds the transfer function directly from the schematic:

```
In[45]:= {myTransferFunction, systemInp, systemOut} =
           DiscreteSystemTransferFunction[mySystem]
```

Out[45]= $\left\{\left\{\left\{\frac{G}{A+z}\right\}\right\}, \{Y[\{0, 0\}]\}, \{Y[\{8, 0\}]\}\right\}$

`DiscreteSystemTransferFunction` returns a list of the form {*transferFunction*, *systemInput*, *systemOutput*}. The first item, *transferFunction*, is the transfer function matrix of the system:

```
In[46]:= myTransferFunction
```

Out[46]= $\left\{\left\{\frac{G}{A+z}\right\}\right\}$

The second item, *systemInput*, is the symbol that represents the system input:

```
In[47]:= systemInp

Out[47]= {Y[{0, 0}]}
```

The last item, *systemOutput*, is the symbol that represents the system output:

```
In[48]:= systemOut

Out[48]= {Y[{8, 0}]}
```

## Frequency Response

For specific values of system parameters, you can plot the magnitude response:

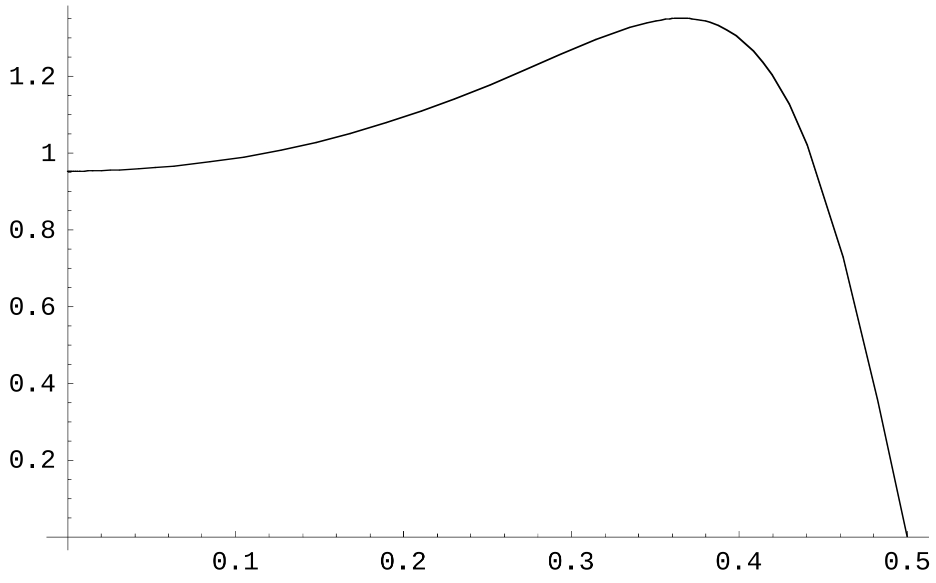
```
In[49]:= myValues = {A → 0.4, G → (1 + z^(-1)) / (1 + 0.5 * z^(-1))}
```

Out[49]= $\left\{A \to 0.4,\ G \to \frac{1+\frac{1}{z}}{1+\frac{0.5}{z}}\right\}$

```
In[50]:= myTF = myTransferFunction[[1, 1]] /. myValues
```

Out[50]= $\frac{1+\frac{1}{z}}{\left(1+\frac{0.5}{z}\right)(0.4+z)}$

```
In[51]:= Plot[Abs[myTF /. z → Exp[I * 2 * Pi * f]], {f, 0, 0.5}];
```



*SchematicSolver* provides symbolic solution to the system: the response and the transfer function are closed-form expressions in terms of the system parameters, kept as symbols, and the complex variable.

## Impulse and Step Response

*Impulse response* of a discrete system is a discrete function. It can be computed by using the inverse *z*-transform of the system transfer function returned by *SchematicSolver*.

```
In[52]:=  myTF
```

Out[52]= $\dfrac{1+\frac{1}{z}}{\left(1+\frac{0.5}{z}\right)(0.4+z)}$

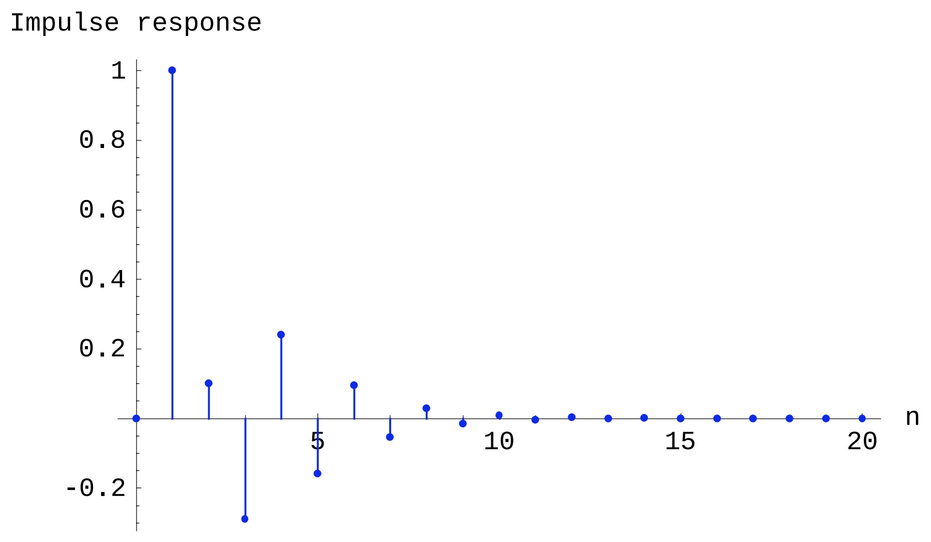
```
In[53]:=  impulseResponse = InverseZTransform[myTF, z, n]
```

Out[53]= $-10.\ (-0.5)^n + 10.\ (-0.4)^n + (20.\ (-0.5)^n - 25.\ (-0.4)^n)\ \text{UnitStep}[-1+n]$

`myTF` is the system transfer function returned by *SchematicSolver*.

Here is the plot of the impulse response:

```
In[54]:=  impulseResponseSeq = impulseResponse /. n → Range[0, 20] // ListToSequence

Out[54]=  {{0.}, {1.}, {0.1}, {-0.29}, {0.241}, {-0.1589}, {0.09481},
           {-0.053549}, {0.0292321}, {-0.0155991}, {0.00819276}, {-0.00425367},
           {0.00218975}, {-0.00112004}, {0.000570086}, {-0.00028907}, {0.000146145},
           {-0.000073717}, {0.0000371162}, {-0.0000186612}, {9.37182×10^-6}}

In[55]:=  SequencePlot[impulseResponseSeq, AxesLabel → {"n", "Impulse response"}];
```



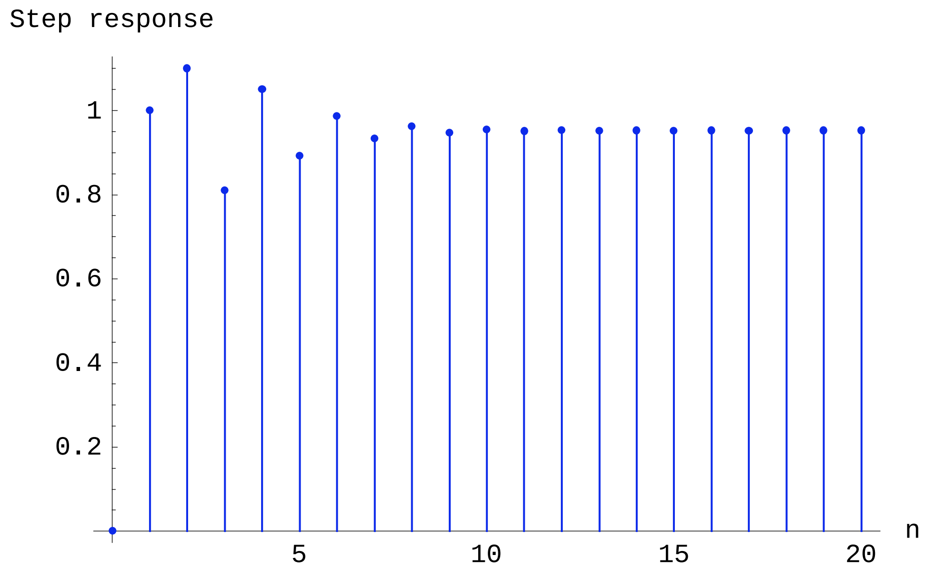
You can use `DiscreteSystemProcessingSISO` to compute the impulse or step response as a sequence of samples for the transfer function, returned by *SchematicSolver*, as follows:

```
In[56]:=  stepData = UnitStepSequence[21] // SequenceToList

Out[56]=  {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

In[57]:=  {stepResponseData, finalStates} = DiscreteSystemProcessingSISO[stepData, myTF];
```

```
In[58]:=  stepResponseSeq = stepResponseData // ListToSequence

Out[58]=  {{0}, {1}, {1.1}, {0.81}, {1.051}, {0.8921}, {0.98691}, {0.933361}, {0.962593},
           {0.946994}, {0.955187}, {0.950933}, {0.953123}, {0.952003}, {0.952573},
           {0.952284}, {0.95243}, {0.952356}, {0.952393}, {0.952375}, {0.952384}}
```

Here is the plot of the step response:

```
In[59]:=  SequencePlot[stepResponseSeq, AxesLabel → {"n", "Step response"}];
```



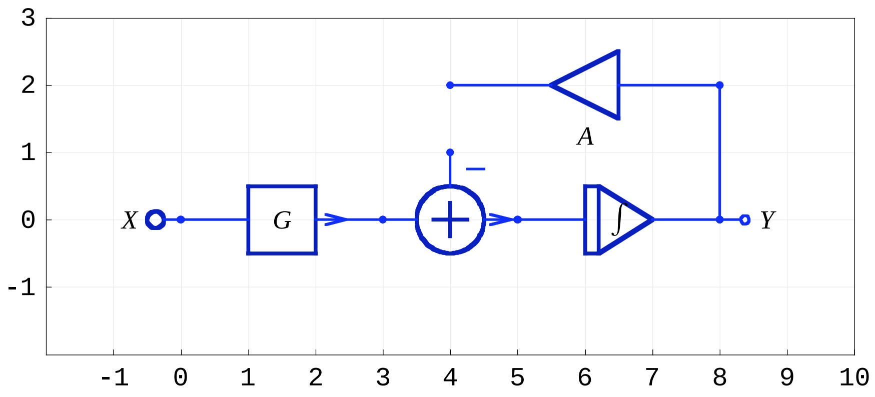
## 4.3. Systems with Unconnected Inputs

Sometimes, it happens that inputs of some system elements are left unconnected. Traditionally, systems with unconnected element inputs are not solvable. *SchematicSolver* successfully solves these systems: signals at unconnected element inputs are automatically generated as unique symbols. Thus, if you by mistake left unconnected an element input, it is easy to identify the mistake. If you intentionally leave some element inputs unconnected, you can assign values to the corresponding input signals after the analysis.

```
In[60]:=  Needs["SchematicSolver`"]
```

Consider the following system:

```
In[61]:= myUnconnectedSystem = {
            {"Input", {0, 0}, X, ""},
            {"Block", {{0, 0}, {3, 0}}, G, ""},
            {"Adder", {{3, 0}, {4, -2}, {5, 0}, {4, 1}}, {1, 0, 2, -1}, ""},
            {"Integrator", {{5, 0}, {8, 0}}, 1, ""},
            {"Amplifier", {{8, 2}, {4, 2}}, A, ""},
            {"Line", {{8, 0}, {8, 2}}},
            {"Output", {8, 0}, Y, ""}
          };
         ShowSchematic[%, PlotRange → {{-2, 10}, {-2, 3}}]
```



Obviously, the negative adder input is left unconnected. Let us find the response of the system:

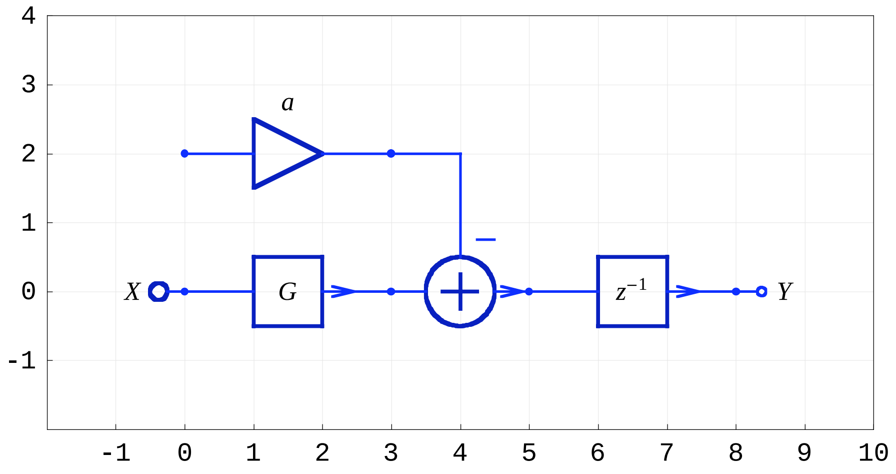
```
In[63]:= ContinuousSystemResponse[myUnconnectedSystem] // First // ColumnForm
```

```
Out[63]= Y[{4, 2}] → A (G X−Y[{4,1}]) / s
         Y[{8, 0}] → − (−G X+Y[{4,1}]) / s
         Y[{5, 0}] → G X − Y[{4, 1}]
         Y[{3, 0}] → G X
         Y[{0, 0}] → X
```

The above solution is correct. *SchematicSolver* finds the output signal **Y[{8,0}]** in terms of two input signals, **X** and **Y[{4,1}]**. *SchematicSolver* assigns the symbol **Y[{4,1}]** to the unconnected adder input. The appearance of the symbol **Y[{4,1}]** in the response indicates that an error might exist in the schematic.

Consider another system:

```
In[64]:=  mySystem = {
             {"Input", {0, 0}, X, ""},
             {"Block", {{0, 0}, {3, 0}}, G, ""},
             {"Adder", {{3, 0}, {4, -2}, {5, 0}, {3, 2}}, {1, 0, 2, -1}, ""},
             {"Delay", {{5, 0}, {8, 0}}, 1, ""},
             {"Multiplier", {{0, 2}, {3, 2}}, a, ""},
             {"Output", {8, 0}, Y, ""}
           };
          ShowSchematic[%, PlotRange → {{-2, 10}, {-2, 4}}]
```



Obviously, the multiplier input is left unconnected. Let us find the response of the system:

```
In[66]:=  myResponse = DiscreteSystemResponse[mySystem] // First
```

Out[66]= $\{Y[\{8,0\}] \to -\frac{-G X + a Y[\{0,2\}]}{z},\ Y[\{5,0\}] \to G X - a Y[\{0,2\}],\ Y[\{3,0\}] \to G X,\ Y[\{3,2\}] \to a Y[\{0,2\}],\ Y[\{0,0\}] \to X\}$

Again, the above solution is correct. *SchematicSolver* finds the output signal **`Y[{8,0}]`** in terms of two input signals, **`X`** and **`Y[{0,2}]`**. *SchematicSolver* assigns the symbol **`Y[{0,2}]`** to the unconnected multiplier input.

```
In[67]:=  Youtput = Y[{8, 0}] /. myResponse
```

Out[67]= $-\frac{-G X + a Y[\{0,2\}]}{z}$

Suppose, now, that the multiplier input should be connected to the same input **`X`**. The system output becomes

```
In[68]:=  Youtput /. Y[{0, 2}] → X
```

Out[68]= $-\frac{a X - G X}{z}$

and it is an expression in terms of system parameters and the known stimulus.

*SchematicSolver* can report on unconnected inputs if you specify the option **`PrintFloatingPorts→True`**:

```
In[69]:=  DiscreteSystemResponse[mySystem, PrintFloatingPorts → True]

            Floating ports = {Y[{0, 2}]}
```

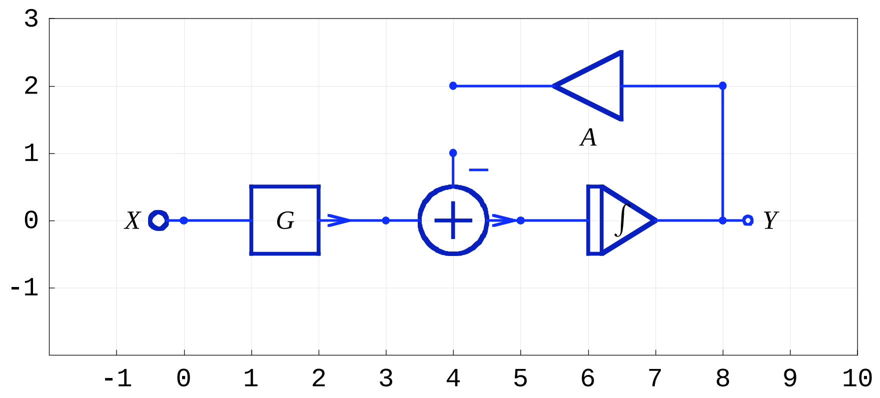
Out[69]= $\{\{Y[\{8,0\}] \to -\frac{-G X + a Y[\{0,2\}]}{z},\ Y[\{5,0\}] \to G X - a Y[\{0,2\}],\ Y[\{3,0\}] \to G X,\ Y[\{3,2\}] \to a Y[\{0,2\}],\ Y[\{0,0\}] \to X\},\ \{Y[\{8,0\}], Y[\{5,0\}], Y[\{3,2\}], Y[\{3,0\}], Y[\{0,0\}]\}\}$

Default value of `PrintFloatingPorts` is `False`.

## 4.4. Combining Unconnected Systems

Unconnected element inputs can be used for combining subsystems into flexible larger systems. For example, let us consider again the system **`myUnconnectedSystem`** and find its response:

```
In[70]:= myUnconnectedSystem = {
            {"Input", {0, 0}, X, ""},
            {"Block", {{0, 0}, {3, 0}}, G, ""},
            {"Adder", {{3, 0}, {4, -2}, {5, 0}, {4, 1}}, {1, 0, 2, -1}, ""},
            {"Integrator", {{5, 0}, {8, 0}}, 1, ""},
            {"Amplifier", {{8, 2}, {4, 2}}, A, ""},
            {"Line", {{8, 0}, {8, 2}}},
            {"Output", {8, 0}, Y, ""}
          };
         ShowSchematic[%, PlotRange → {{-2, 10}, {-2, 3}}]
```



```
In[72]:= myUnconnectedSystemResponse = ContinuousSystemResponse[myUnconnectedSystem] // First
```

Out[72]= $\left\{Y[\{4,2\}] \to \frac{A\,(G\,X - Y[\{4,1\}])}{s},\ Y[\{8,0\}] \to -\frac{-G\,X + Y[\{4,1\}]}{s},\ Y[\{5,0\}] \to G\,X - Y[\{4,1\}],\ Y[\{3,0\}] \to G\,X,\ Y[\{0,0\}] \to X\right\}$

The amplifier output **`Y[{4,2}]`** is left unconnected. We shall denote this signal with **`YoutUnconnected`** and find it as

```
In[73]:= YoutUnconnected = Y[{4, 2}] /. myUnconnectedSystemResponse
```

Out[73]= $\frac{A\,(G\,X - Y[\{4,1\}])}{s}$

Note that **`YoutUnconnected`** is in terms of the unconnected input **`Y[{4,1}]`**. If we connect the two unconnected nodes, **`Y[{4,1}]`** becomes

```
In[74]:= myNewY41 = Solve[{Y[{4, 1}] == YoutUnconnected}, Y[{4, 1}]] // Flatten // Simplify
```

Out[74]= $\left\{Y[\{4,1\}] \to \frac{A\,G\,X}{A+s}\right\}$

The output of the unconnected system is in terms of **`Y[{4,1}]`**

```
In[75]:= Youtput = Y[{8, 0}] /. myUnconnectedSystemResponse
```

Out[75]= $-\frac{-G\,X + Y[\{4,1\}]}{s}$

and after connected the two nodes, it becomes

```
In[76]:=  YoutputNew = Youtput /. myNewY41 // Simplify
```

Out[76]= $\frac{G\,X}{A + s}$

Previous example illustrates the simplest connection of unconnected nodes. Generally, two nodes can be connected through another system of a know transfer function, say

```
In[77]:=  gTF = g → (s + 1) / (s + 2)
```

Out[77]= $g \to \frac{1 + s}{2 + s}$

In this case, **Y[{4,1}]** becomes

```
In[78]:=  myY41g = Solve[{Y[{4, 1}] == g * YoutUnconnected}, Y[{4, 1}]] /. gTF // Flatten // Simplify
```

Out[78]= $\left\{Y[\{4, 1\}] \to \frac{A\,G\,(1 + s)\,X}{A\,(1 + s) + s\,(2 + s)}\right\}$

the system output takes the value

```
In[79]:=  Youtg = Youtput /. myY41g // Together
```

Out[79]= $\frac{2\,G\,X + G\,s\,X}{A + 2\,s + A\,s + s^2}$

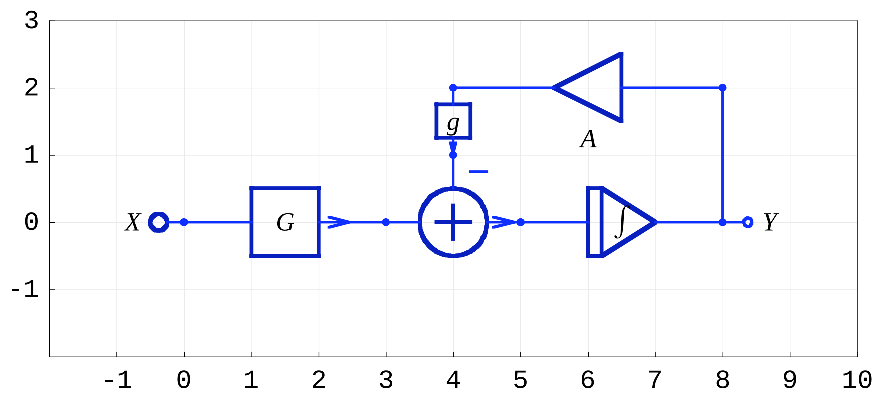
and the corresponding transfer function is

```
In[80]:=  Hg = Youtg / X // Simplify
```

Out[80]= $\frac{G\,(2 + s)}{A\,(1 + s) + s\,(2 + s)}$

This result can be verified by solving the modified system in which the unconnected nodes are connected by the block *g*.

```
In[81]:=  myModifiedSystem = {
             {"Block", {{4, 2}, {4, 1}}, g, ""},
             {"Input", {0, 0}, X, ""},
             {"Block", {{0, 0}, {3, 0}}, G, ""},
             {"Adder", {{3, 0}, {4, -2}, {5, 0}, {4, 1}}, {1, 0, 2, -1}, ""},
             {"Integrator", {{5, 0}, {8, 0}}, 1, ""},
             {"Amplifier", {{8, 2}, {4, 2}}, A, ""},
             {"Line", {{8, 0}, {8, 2}}},
             {"Output", {8, 0}, Y, ""}};
          ShowSchematic[%, PlotRange → {{-2, 10}, {-2, 3}}]
```



```
In[83]:= {myModfiedSystemTF, systemInp, systemOut} =
          ContinuousSystemTransferFunction[myModifiedSystem /. gTF] // Simplify
```

Out[83]= $\left\{\left\{\left\{\frac{G\,(2+s)}{A\,(1+s)+s\,(2+s)}\right\}\right\},\ \{Y[\{0,0\}]\},\ \{Y[\{8,0\}]\}\right\}$

```
In[84]:= SameQ[myModfiedSystemTF[[1, 1]], Hg]

Out[84]= True
```

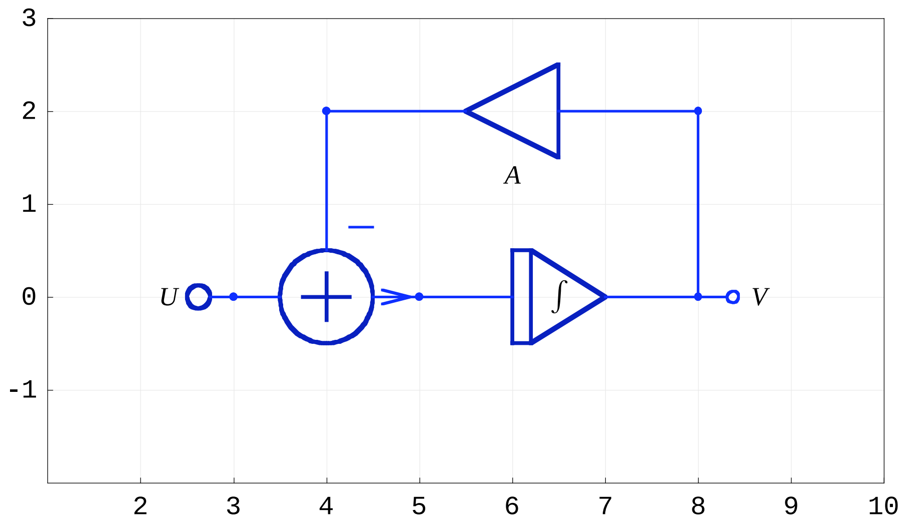
## 4.5. Names for System Variables and Signals

By default, *SchematicSolver* denotes the complex frequency with **s**, the complex variable with **z**, and the transforms of signals with **Y[{i,j}]**. You can change these default names by providing two additional arguments to the function for solving systems.

Consider the following system:

```
In[85]:= mySystem = {
            {"Input", {3, 0}, U},
            {"Adder", {{3, 0}, {4, -2}, {5, 0}, {4, 2}}, {1, 0, 2, -1}},
            {"Integrator", {{5, 0}, {8, 0}}, 1},
            {"Amplifier", {{8, 2}, {4, 2}}, A},
            {"Line", {{8, 0}, {8, 2}}},
            {"Output", {8, 0}, V}
          };
```

```
In[86]:= ShowSchematic[mySystem, PlotRange → {{1, 10}, {-2, 3}}]
```



Analyze the system and denote the signals with **X**, and assume that the complex frequency is designated by **p**:

```
In[87]:= ContinuousSystemResponse[mySystem, X, p] // First // ColumnForm
```

Out[87]= $X[\{4,2\}] \to \frac{A\,U}{A+p}$

$X[\{8,0\}] \to \frac{U}{A+p}$

$X[\{5,0\}] \to \frac{p\,U}{A+p}$

$X[\{3,0\}] \to U$

## 4.6. Solving Nonlinear Discrete-Time Systems

Using *SchematicSolver*'s schematic capabilities, symbolic system analysis and signal processing, you can perform fast and accurate simulations of nonlinear discrete-time systems. *SchematicSolver* can solve some classes of nonlinear systems. The term *solve* means that *SchematicSolver* can find the closed-form expression of the output signal for a known stimulus given by a closed-form expression.

This section illustrates step-by-step procedures for analyzing nonlinear systems. For the given block-diagram of a system, the required equations are formulated as a system of equations. The set of equations is solved to find the system response as a discrete function.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

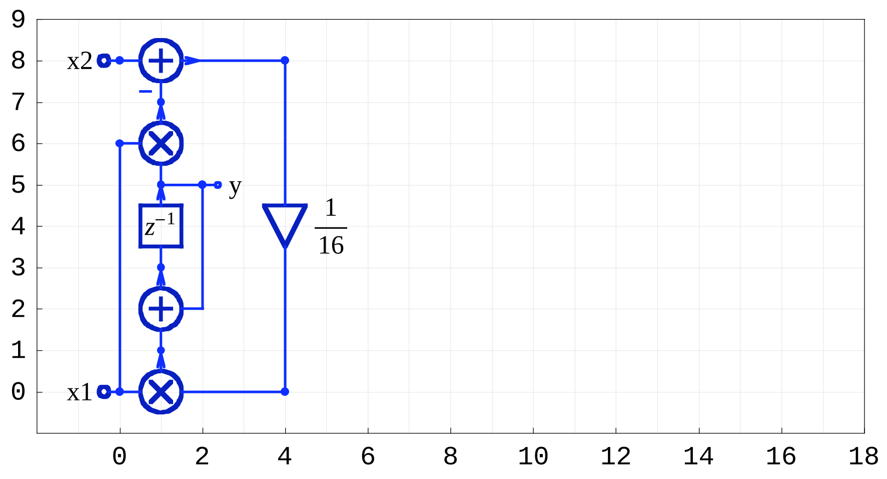
```
In[88]:=  Needs["SchematicSolver`"];
```

Consider a simple system specified by the following list:

```
In[89]:=  nonlinearSystem =  {{"Output", {2, 5}, "y"},
              {"Modulator", {{0, 0}, {4, -1}, {4, 0}, {1, 1}}, {1, 0, 1, 2}},
              {"Modulator", {{0, 6}, {1, 5}, {2, 6}, {1, 7}}, {1, 1, 0, 2}},
              {"Adder", {{0, 8}, {1, 7}, {4, 8}, {1, 9}}, {1, -1, 2, 0}},
              {"Adder", {{0, 2}, {1, 1}, {2, 5}, {1, 3}}, {0, 1, 1, 2}},
              {"Line", {{0, 0}, {0, 6}}}, {"Line", {{1, 5}, {2, 5}}},
              {"Input", {0, 0}, x1}, {"Input", {0, 8}, x2},
              {"Multiplier", {{4, 8}, {4, 0}}, 1/16}, {"Delay", {{1, 3}, {1, 5}}, 1}};
```

`ShowSchematic` shows the block diagram of the nonlinear system:

```
In[90]:=  ShowSchematic[nonlinearSystem, PlotRange → {{-2, 18}, {-1, 9}}];
```



This is a part of an adaptive system based on  the least mean squares (LMS) algorithm.

A unit step sequence is applied to the input `x1`.

```
In[91]:=  numberOfSamples = 50;
```

```
In[92]:=  inpSeq1 = UnitStepSequence[numberOfSamples];
```

Let us apply to the second input the sequence `inpSeq1` multiplied by a constant `gain`.

```
In[93]:=  gain = 10;
```

```
In[94]:=  inpSeq2 = gain * inpSeq1;
```

Here is the multiplex sequence that represents the input to the system:

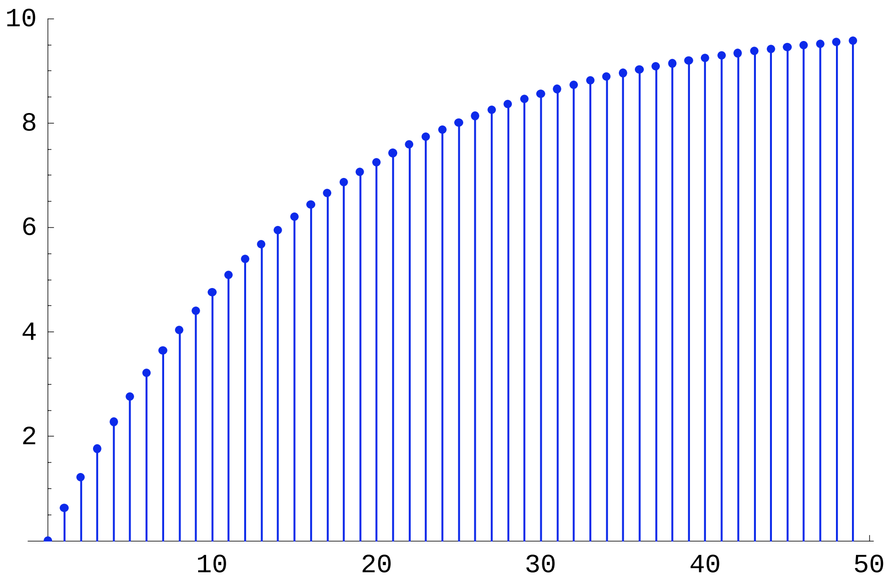
```
In[95]:=  inpSeq = MultiplexSequence[inpSeq1, inpSeq2];
```

`DiscreteSystemSimulation` finds the system output:

```
In[96]:=  outSeq = DiscreteSystemSimulation[nonlinearSystem, inpSeq];
```

`SequencePlot` plots `outSeq`.

```
In[97]:=  SequencePlot[outSeq, PlotRange → {0, gain}];
```



The above figure shows that the output sequence tends to the value of `gain`. After 30 samples, `outSeq` reaches about 90% of `gain`. The system output can be interpreted as an estimation of `gain`.

Can we find `outSeq` as a closed-form expression in terms of the sample index?

First, let us find the system implementation with `DiscreteSystemImplementation`:

```
In[98]:=  DiscreteSystemImplementation[nonlinearSystem, "implementProc"];

            Implementation procedure name: implementProc

            Implementation procedure usage:

             {{Y1p5}, {Y1p3}} = implementProc[{Y0p0, Y0p8},{Y1p5},{}] is the template
              for calling the procedure.  The general template is {outputSamples,
              finalConditions} = procedureName[inputSamples, initialConditions,
              systemParameters]. See also: DiscreteSystemImplementationProcessing
```

Assume that the system was at rest (zero initial conditions) and that the two inputs are excited by the first set of samples (that correspond to the sample index 0).

```
In[99]:=  initialState = {0}

Out[99]=  {0}
```

```
In[100]:=
      firstSampleSet = inpSeq[[1]]

Out[100]=
      {1, 10}
```

The corresponding output sample, `y0`, and the final state, `d0`, are computed with `implementProc`:

```
In[101]:=
      {{y0}, {d0}} = implementProc[firstSampleSet, initialState, {}]

Out[101]=
```
$$\left\{\{0\},\left\{\frac{5}{8}\right\}\right\}$$

Generally, we can compute the symbolic output sample, `y2`, and final state, `d2`, for an arbitrary initial state `d1` and the stimulus `{1,gain}` that correspond to an arbitrary sample index.

```
In[102]:=
      {{y2}, {d2}} = implementProc[{1, gain}, {d1}, {}] // Simplify

Out[102]=
```
$$\left\{\{d1\},\left\{\frac{5}{16}\,(2+3\,d1)\right\}\right\}$$

For the next sample index, the output sample `y3` and final state `d3` follow:

```
In[103]:=
      {{y3}, {d3}} = implementProc[{1, gain}, {d2}, {}] // Simplify

Out[103]=
```
$$\left\{\left\{\frac{5}{16}\,(2+3\,d1)\right\},\left\{\frac{5}{256}\,(62+45\,d1)\right\}\right\}$$

The two output samples are functions of the symbolic initial state `d1`. The function `Eliminate` tries to eliminate the initial state `d1` and tries to find the relation between the two output samples.

```
In[104]:=
      reducedEqns = Eliminate[{y[n - 1] == y2, y[n] == y3}, {d1}]

Out[104]=
      2 (-5 + 8 y[n]) == 15 y[-1 + n]
```

We have to load the package for solving recurrence equations:

```
In[105]:=
     <<DiscreteMath`RSolve`
```

Here is the solution to the recurrence equation:

```
In[106]:=
      Clear[y, n]
      mySol = RSolve[{reducedEqns, y[0] == y0}, y[n], n]

Out[107]=
```
$$\left\{\left\{y[n]\to 10\left(1-\left(\frac{15}{16}\right)^{n}\right)\right\}\right\}$$

We have used the first output sample to specify the initial value of the solution. The solution to the recurrence relation is a replacement rule. To form an expression, we define a new function using the replacement rule.

```
In[108]:=
      Clear[w]
      w[n_] := (y[n] /. mySol[[1]]) // Evaluate
```

```
In[110]:=
     w[k]
```

```
Out[110]=
```

$$10\left(1-\left(\frac{15}{16}\right)^{k}\right)$$

The output sequence `outSeq` and the sequence computed with `w[k]` should be identical:

```
In[111]:=
     wSeq = Table[{w[k]}, {k, 0, 49}];
     SameQ[wSeq, outSeq]

Out[112]=
     True
```

Let us find the number of samples after which the output sequence has the value `gain*b`, say 90% of the `gain`.

```
In[113]:=
     Solve[w[n] == gain * b, n] // Simplify
     % /. b → 0.9

     Solve::ifun :
      Inverse functions are being used by Solve, so some solutions may not be found.

Out[113]=
```

$$\left\{\left\{n\to-\frac{\text{Log}[1-b]}{\text{Log}[\frac{16}{15}]}\right\}\right\}$$

```
Out[114]=
     {{n → 35.6777}}
```

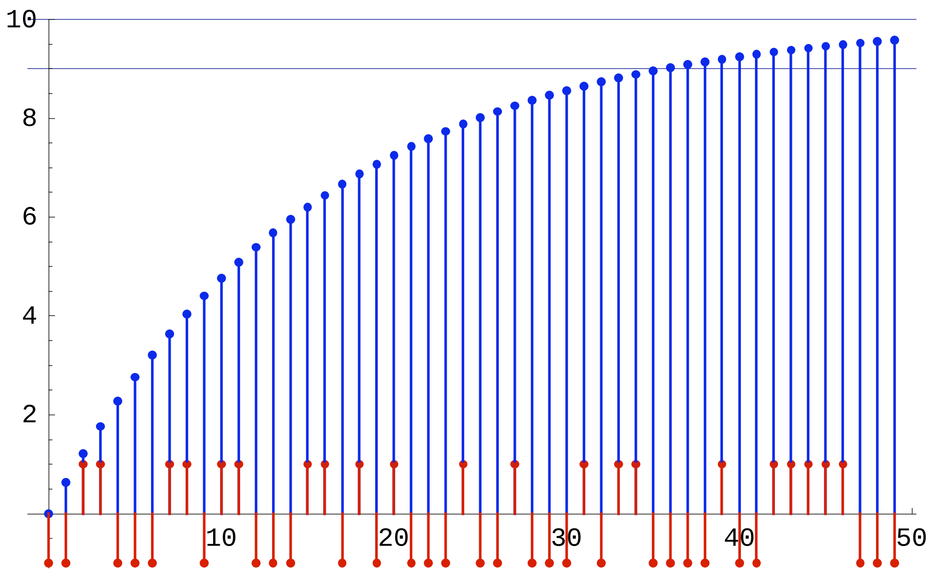
In this example, the function `Solve` finds a closed-form expression in terms of the symbol `b`.

Let us process random samples of amplitude ±1.

```
In[115]:=
     randSeq = Sign[UnitNoiseSequence[numberOfSamples]];
     inpSeq = MultiplexSequence[randSeq, gain * randSeq];
     outSeq = DiscreteSystemSimulation[nonlinearSystem, inpSeq];
```

The plot of the input and the output sequences follows:

```
In[118]:=
      plotSeq = MultiplexSequence[outSeq, randSeq];
      SequencePlot[plotSeq,
        PlotRange → {-1.1, gain}, GridLines → {{}, {0.9 * gain, gain}}];
```



The number of samples after which `outSeq` reaches 90% of `gain` is the same as that obtained by solving the system.

# 5. Examples of Solving Systems

## 5.1. Continuous-Time Systems

### Introduction

*SchematicSolver* has many unique features not available in other software: symbolic signal processing brings you

- Computation of transfer functions as closed-form expressions in terms of symbolic system parameters
- Finding the closed-form response from the schematic
- Symbolic optimization of the system response

The derived result is the most general because all system parameters and inputs can be given by symbols.

Other important features include:

- Design of systems for known symbolic transfer function, impulse, or step response; you can generate the schematic of the system and find the system parameters
- Building models from automatically generated schematics; you can change system parameters on the fly and immediately see what happens with the results

*SchematicSolver*'s powerful functions for solving continuous-time (analog) systems are illustrated by the subsequent examples.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[1]:=  Needs["SchematicSolver`"];
```

## Diving Submarine System

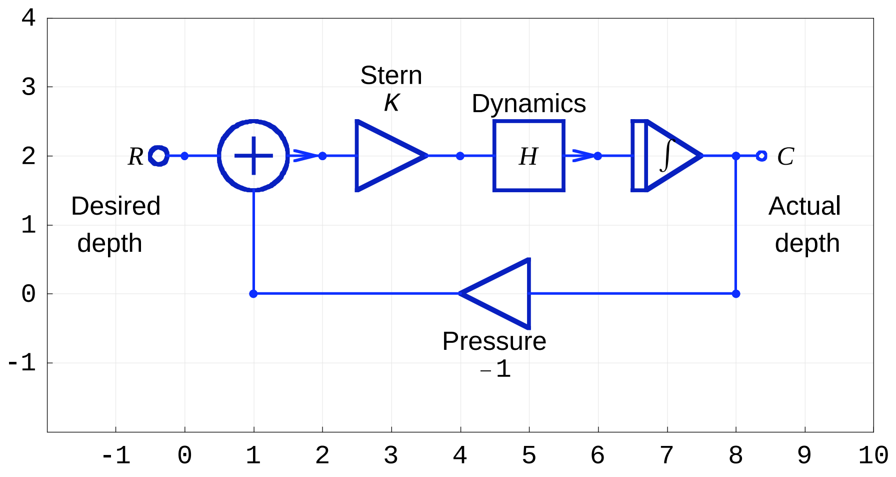
For the simplified block diagram model of a diving submarine find the transfer function $C/R$. *Stern plane actuator* is an amplifier of gain $K$, *submarine dynamics* is represented by $H = \frac{(s+a)^2}{s^2+w^2}$, and *pressure transducer* is an amplifier with a gain of negative one.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[2]:=  Needs["SchematicSolver`"];
```

Here is the system schematic:

```
In[3]:=  divingSubmarineSystem = {
           {"Input", {0, 2}, R},
           {"Output", {8, 2}, C},
           {"Integrator", {{6, 2}, {8, 2}}},
           {"Amplifier", {{2, 2}, {4, 2}}, K, "Stern"},
           {"Amplifier", {{8, 0}, {1, 0}}, -1, "Pressure"},
           {"Block", {{4, 2}, {6, 2}}, H, "Dynamics"},
           {"Text", {-1, 1}, "Desired\n depth"},
           {"Text", {9, 1}, "Actual\n depth"},
           {"Adder", {{0, 2}, {1, 0}, {2, 2}, {1, 3}}, {1, 1, 2, 0}},
           {"Line", {{8, 2}, {8, 0}}}
         };
         ShowSchematic[%, PlotRange → {{-2, 10}, {-2, 4}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
In[5]:=  {tfMatrix, systemInp, systemOut} =
          ContinuousSystemTransferFunction[divingSubmarineSystem]
```

Out[5]= $\left\{\left\{\left\{\frac{H\,K}{H\,K+s}\right\}\right\},\ \{Y[\{0,\ 2\}]\},\ \{Y[\{8,\ 2\}]\}\right\}$

Here is the given submarine dynamics:

```
In[6]:=  submarineDynamics = H → (s + a) ^2 / (s^2 + w^2)
```

Out[6]= $H \to \frac{(a+s)^2}{s^2+w^2}$

The transfer function $C/R$ is the element of the transfer function matrix:

```
In[7]:=  actualDepthTF = tfMatrix[[1, 1]] /. submarineDynamics // Together
```

Out[7]= $$\frac{K\,(a+s)^2}{a^2 K + 2\,a\,K\,s + K\,s^2 + s^3 + s\,w^2}$$

This collects together terms involving the same powers of the complex frequency *s*:

```
In[8]:=  Numerator[actualDepthTF] / Collect[Denominator[actualDepthTF], s] // TraditionalForm
```

Out[8]//TraditionalForm= $$\frac{K\,(a+s)^2}{s^3 + K\,s^2 + (w^2 + 2\,a\,K)\,s + a^2\,K}$$

The derived result is the most general because all system parameters are given by symbols. Here is the transfer function with specific parameter values:

In[9]:= `actualDepthTF1 = actualDepthTF /.` $\left\{a \to 1,\ w \to \frac{1}{\sqrt{10}}\right\}$

Out[9]= $$\frac{K\,(1+s)^2}{K + \frac{s}{10} + 2\,K\,s + K\,s^2 + s^3}$$

```
In[10]:=  Numerator[actualDepthTF1] / Collect[Denominator[actualDepthTF1], s] //
           TraditionalForm
```

Out[10]//TraditionalForm= $$\frac{K\,(s+1)^2}{s^3 + K\,s^2 + \left(2\,K + \frac{1}{10}\right)s + K}$$

## Unstable Plant System

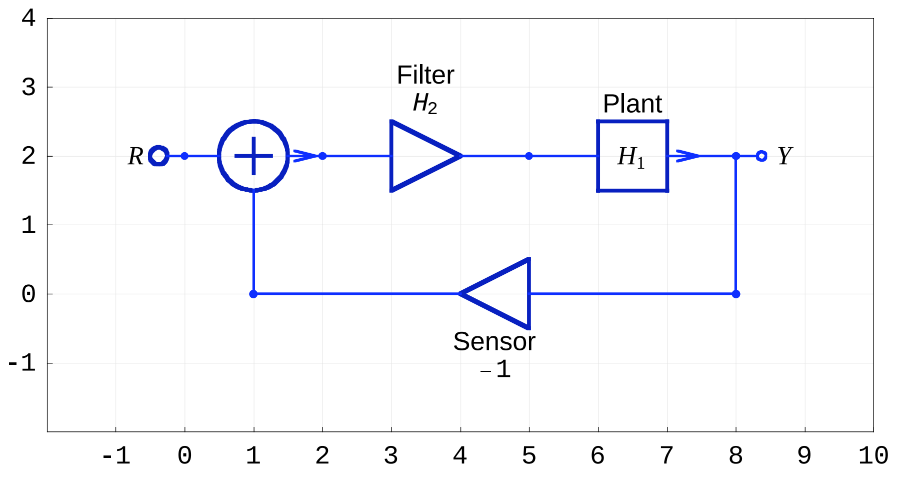
An *unstable plant* described by the transfer function $H_1$ is made part of a new *feedback system* (shown in Figure) that includes a *filter* $H_2$ in the forward path and a *sensor* with a gain of negative one in the feedback path. Find the overall transfer function $H = Y / R$.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[11]:= Needs["SchematicSolver`"];
```

Here is the system schematic:

```
In[12]:= unstablePlantSystem = {
           {"Input", {0, 2}, R},
           {"Output", {8, 2}, Y},
           {"Amplifier", {{2, 2}, {5, 2}}, H2, "Filter"},
           {"Block", {{5, 2}, {8, 2}}, H1, "Plant"},
           {"Amplifier", {{8, 0}, {1, 0}}, -1, "Sensor"},
           {"Adder", {{0, 2}, {1, 0}, {2, 2}, {1, 3}}, {1, 1, 2, 0}}, {"Line", {{8, 2}, {8, 0}}}
          };
         ShowSchematic[% /. {H1 → H₁, H2 → H₂}, PlotRange → {{-2, 10}, {-2, 4}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
In[14]:= {tfMatrix, systemInp, systemOut} =
          ContinuousSystemTransferFunction[unstablePlantSystem]
```

Out[14]= $\left\{\left\{\left\{\frac{\text{H1 H2}}{1+\text{H1 H2}}\right\}\right\}, \{Y[\{0, 2\}]\}, \{Y[\{8, 2\}]\}\right\}$

Assume that the plant and filter are given by

```
In[15]:= H1H2value = {H1 → 1 / (s * (s - 1)), H2 → (k * s + 8) / (s + 10)};
         TraditionalForm[H1H2value /. {H1 → H₁, H2 → H₂}]
```

Out[15]//TraditionalForm=

$$\left\{H_1 \to \frac{1}{(s-1)\, s},\ H_2 \to \frac{k\, s+8}{s+10}\right\}$$

The overall transfer function is the element of the transfer function matrix:

```
In[16]:= unstablePlantTF = tfMatrix[[1, 1]] /. H1H2value // Together
```

Out[16]= $\frac{8+k\,s}{8-10\,s+k\,s+9\,s^2+s^3}$

This collects together terms involving the same powers of the complex frequency *s*:

```
In[17]:= Numerator[unstablePlantTF] / Collect[Denominator[unstablePlantTF], s] //
          TraditionalForm
```

Out[17]//TraditionalForm=

$$\frac{k\,s+8}{s^3+9\,s^2+(k-10)\,s+8}$$

## Supply and Demand System

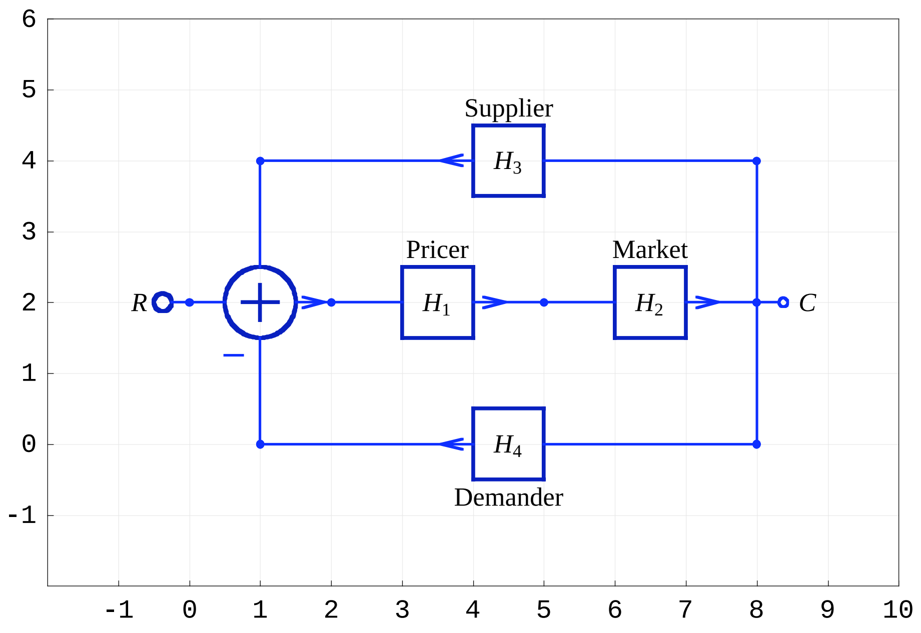
Assume that linear approximations in the form of transfer functions are available for each block of the *supply and demand system*, and that the system can be represented by Figure below. Determine the overall transfer function of the system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[18]:=  Needs["SchematicSolver`"];
```

Here is the system schematic:

```
In[19]:=  supplyDemandSystem = {
             {"Input", {0, 2}, R},
             {"Output", {8, 2}, C},
             {"Block", {{2, 2}, {5, 2}}, H1, "Pricer"},
             {"Block", {{5, 2}, {8, 2}}, H2, "Market"},
             {"Block", {{8, 0}, {1, 0}}, H4, "Demander"},
             {"Block", {{8, 4}, {1, 4}}, H3, "Supplier", TextOffset → {0, 1}},
             {"Adder", {{0, 2}, {1, 0}, {2, 2}, {1, 4}}, {1, -1, 2, 1}},
             {"Line", {{8, 2}, {8, 0}}}, {"Line", {{8, 2}, {8, 4}}}
           };
          ShowSchematic[% /. {H1 → H₁, H2 → H₂, H3 → H₃, H4 → H₄}, PlotRange → {{-2, 10}, {-2, 6}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
In[21]:=  {tfMatrix, systemInp, systemOut} =
           ContinuousSystemTransferFunction[supplyDemandSystem]
```

Out[21]= $\left\{\left\{\left\{-\frac{\mathrm{H1\,H2}}{-1+\mathrm{H1\,H2\,H3}-\mathrm{H1\,H2\,H4}}\right\}\right\}, \{\mathrm{Y}[\{0,2\}]\}, \{\mathrm{Y}[\{8,2\}]\}\right\}$

The transfer function $C/R$ is the element of the transfer function matrix:

```
In[22]:=  supplyDemandTF = tfMatrix[[1, 1]] // Simplify
```

Out[22]= $\frac{\mathrm{H1\,H2}}{1+\mathrm{H1\,H2}\,(-\mathrm{H3}+\mathrm{H4})}$

that is better typeset with

```
In[23]:=  supplyDemandTF /. {H1 → H₁, H2 → H₂, H3 → H₃, H4 → H₄} // TraditionalForm
```

Out[23]//TraditionalForm=

$$\frac{H_1\,H_2}{H_1\,H_2\,(H_4-H_3)+1}$$

## Unity Feedback System

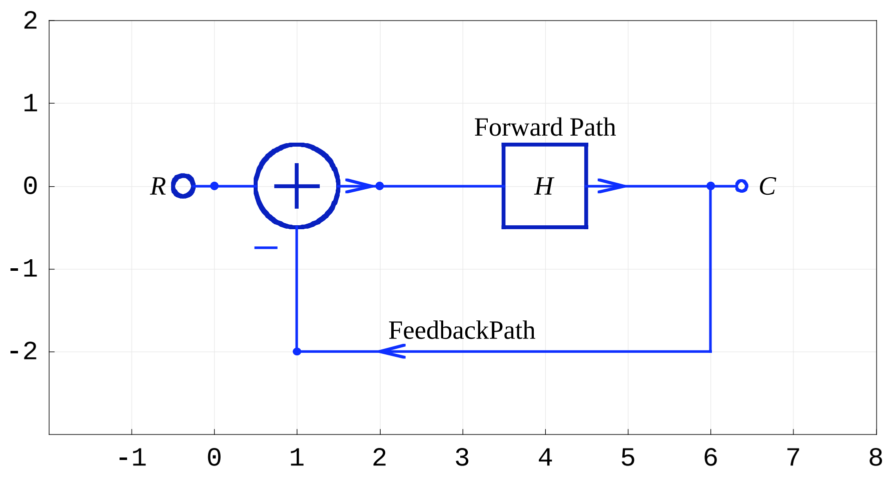
A *unity feedback system* is a feedback system in which the primary feedback is identically equal to the controlled output. Find the response of the system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[24]:=  Needs["SchematicSolver`"];
```

Here is the system schematic:

```
In[25]:=  unityFeedbackSystem = {
              {"Input", {0, 0}, R},
              {"Output", {6, 0}, C},
              {"Block", {{2, 0}, {6, 0}}, H, "Forward Path"},
              {"Arrow", {{2, -2}, {6, -2}}, "FeedbackPath", ShowArrowTail → False},
              {"Adder", {{0, 0}, {1, -2}, {2, 0}, {1, 1}}, {1, -1, 2, 0}},
              {"Line", {{6, 0}, {6, -2}, {1, -2}}}
            };
          ShowSchematic[%, PlotRange → {{-2, 8}, {-3, 2}}];
```



`ContinuousSystemEquations` sets up the equations of the system:

```
In[27]:=  {unityFeedbackEquations, vars} = ContinuousSystemEquations[unityFeedbackSystem];
```

It is better typeset with

```
In[28]:=  typoSubstYkn = {Y[{k_Integer, n_Integer}] :→ Yk,n};
```

```
In[29]:=  unityFeedbackEquations /. typoSubstYkn // ColumnForm // TraditionalForm
```

*Out[29]//TraditionalForm=*

$$Y_{0,0} == R$$
$$Y_{6,0} == H\, Y_{2,0}$$
$$Y_{2,0} == Y_{0,0} - Y_{6,0}$$

`ContinuousSystemResponse` finds the response of the system:

```
In[30]:=  {unityFeedbackResponse, vars} = ContinuousSystemResponse[unityFeedbackSystem];
```

```
In[31]:=  unityFeedbackResponse /. typoSubstYkn // ColumnForm // TraditionalForm
```

*Out[31]//TraditionalForm=*

$$Y_{6,0} \to \frac{H\,R}{H+1}$$
$$Y_{2,0} \to \frac{R}{H+1}$$
$$Y_{0,0} \to R$$

## Satellite Elevation Tracking System

A simplified block diagram of a *satellite elevation tracking system* can be represented by Figure below. The multiloop system uses a combination of unity negative feedback and rate feedback to produce a critically damped response. Obtain the closed-loop transfer function of the system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[32]:=  Needs["SchematicSolver`"];
```

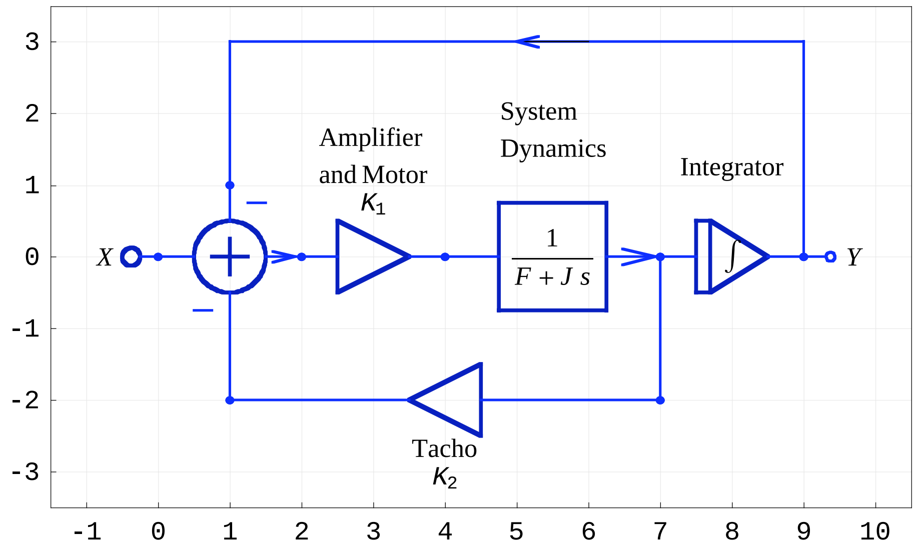
Here is the system schematic:

```
In[33]:=  satelliteElevationTrackingSystem = {
             {"Input", {0, 0}, X, "Elevation demand"},
             {"Adder", {{0, 0}, {1, -2}, {2, 0}, {1, 1}}, {1, -1, 2, -1}},
             {"Amplifier", {{2, 0}, {4, 0}}, K1, "Amplifier\nand Motor"},
             {"Block", {{4, 0}, {7, 0}},
              1 / (J * s + F), "System\nDynamics\n", ElementSize → {3, 2}},
             {"Integrator", {{7, 0}, {9, 0}}, 1, "Integrator\n"},
             {"Line", {{7, 0}, {7, -2}}},
             {"Amplifier", {{7, -2}, {1, -2}}, K2, "Tacho"},
             {"Line", {{9, 0}, {9, 3}, {1, 3}, {1, 1}}},
             {"Arrow", {{5, 3}, {6, 3}}},
             {"Output", {9, 0}, Y, "Pitch"}
           };
```

that is better typeset with

```
In[34]:=  typoSubst = {K1 → K₁, K2 → K₂};
```

```
In[35]:=  ShowSchematic[satelliteElevationTrackingSystem /. typoSubst,
            PlotRange → {{-1.5, 10.5}, {-3.5, 3.5}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
In[36]:=  {tfMatrix, systemInp, systemOut} =
            ContinuousSystemTransferFunction[satelliteElevationTrackingSystem];
```

The transfer function of this single-input single-output system is the element of the transfer function matrix:

```
In[37]:= H = tfMatrix[[1, 1]];
         H /. typoSubst // TraditionalForm
```

*Out[38]//TraditionalForm=*

$$\frac{K_1}{J\, s^2 + F\, s + K_1\, K_2\, s + K_1}$$

## CD-media Controller

Find the transfer function and step response of the *CD-media controller*.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[39]:= Needs["SchematicSolver`"];
```

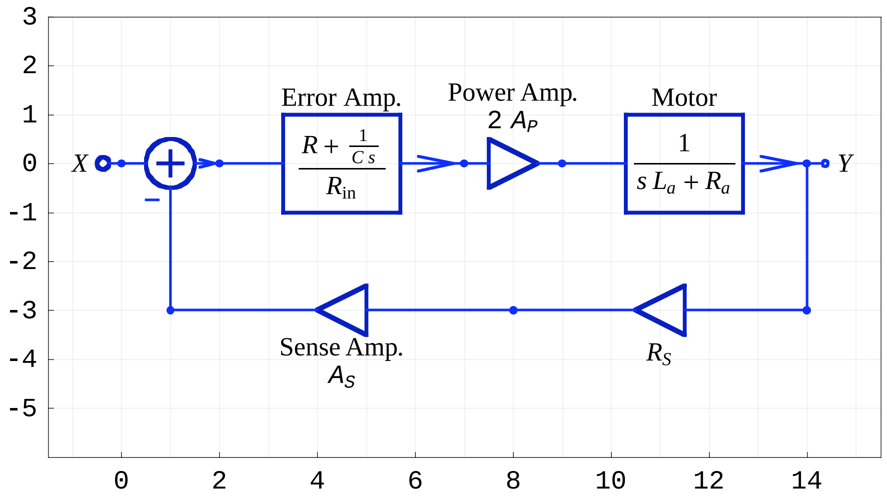
FAN8024D/BD, which is a 4CH motor drive IC suitable for CD-media applications (include CD-ROM, CD-RW, DVDP and DVD-ROM), has 2 current feedback control channels for the Focus and Tracking actuator. The application system, *CD-media controller*, is illustrates by the following block diagram (Application Note 4109, "A guide to the design of current feedback control," Fairchild Semiconductor Corporation, http://www.fairchildsemi.com, AN-4109.pdf, ©2001):

```
In[40]:= CDmediaController = {
            {"Input", {0, 0}, X, "Reference Voltage"},
            {"Adder", {{0, 0}, {1, -3}, {2, 0}, {1, 1}}, {1, -1, 2, 0}},
            {"Block", {{2, 0}, {7, 0}},
             (R + 1 / (C * s)) / Rin, "Error Amp.", ElementSize → {2.4, 2}},
            {"Amplifier", {{7, 0}, {9, 0}}, 2 * AP, "Power Amp."},
            {"Block", {{9, 0}, {14, 0}}, 1 / (Ra + s * La), "Motor", ElementSize → {2.4, 2}},
            {"Line", {{14, 0}, {14, -3}}},
            {"Amplifier", {{8, -3}, {1, -3}}, AS, "Sense Amp."},
            {"Amplifier", {{14, -3}, {8, -3}}, RS},
            {"Output", {14, 0}, Y, "Armature Current"}
           };
```

It is better typeset with

```
In[41]:= typoSubst = {AP → A_P, AS → A_S, La → L_a, Ra → R_a, Rin → R_in, RS → R_S};
```

```
In[42]:= ShowSchematic[CDmediaController /. typoSubst, PlotRange → {{-1.5, 15.5}, {-6, 3}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
In[43]:= {tfMatrix, systemInp, systemOut} =
           ContinuousSystemTransferFunction[CDmediaController];
```

The transfer function of this single-input single-output system is the element of the transfer function matrix:

```
In[44]:=  H = tfMatrix[[1, 1]];
          H /. typoSubst // TraditionalForm
```

Out[45]//TraditionalForm=

$$\frac{2\,(C\,R\,s+1)\,A_P}{C\,L_a\,R_{\mathrm{in}}\,s^2+C\,R_a\,R_{\mathrm{in}}\,s+2\,C\,R\,A_P\,A_S\,R_S\,s+2\,A_P\,A_S\,R_S}$$

This collects together terms involving the same powers of the complex frequency *s*:

```
In[46]:=  Numerator[H] / Collect[Denominator[H], s] /. typoSubst // TraditionalForm
```

Out[46]//TraditionalForm=

$$\frac{2\,(C\,R\,s+1)\,A_P}{C\,L_a\,R_{\mathrm{in}}\,s^2+(C\,R_a\,R_{\mathrm{in}}+2\,C\,R\,A_P\,A_S\,R_S)\,s+2\,A_P\,A_S\,R_S}$$

The load impedance of Maker1 40X focus actuator is 18.5 Ω and 228.5 $\mu$H at 1 kHz, 0.1 V. Consider the required system bandwidth of 60 kHz

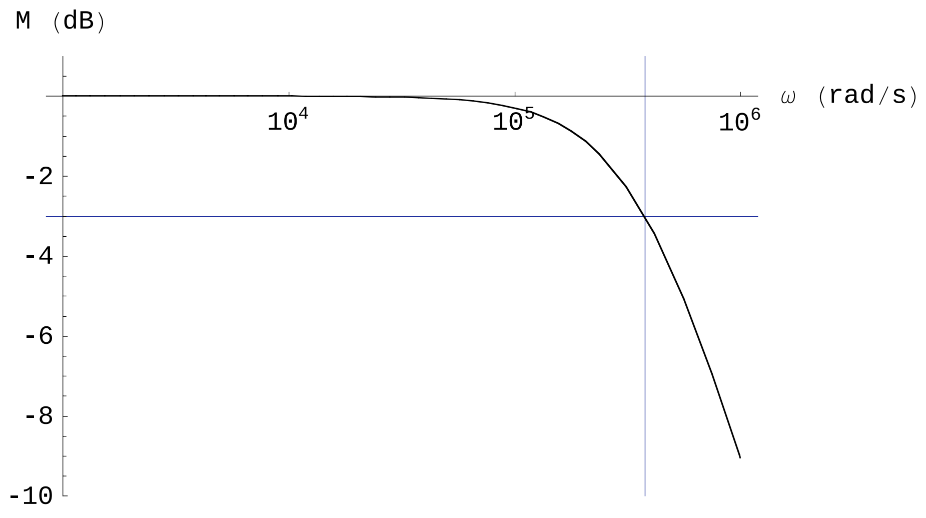
```
In[47]:=  wBandwidth = 2 * π * 60 * 10^3;
```

Assume the sensing resistor of 0.5 Ω and other parameters as follows:

```
In[48]:=  values = {RS → 0.5, AP → 2, AS → 2, C → 76.5 * 10^(-12),
             La → 228.5 * 10^(-6), R → 161.5 * 10^3, Ra → 18.5, Rin → 7.5 * 10^3};
```

Here is the magnitude response in terms of angular frequency:

```
In[49]:=  M = 20 * Log[10, Abs[H /. s → I * w /. values]];

In[50]:=  Plot[M /. w → 10^n, {n, 3, 6},
            AxesLabel → {"ω (rad/s)", "M (dB)"},
            PlotRange → {Automatic, {-10, 1}},
            GridLines → {{Log[10, wBandwidth]}, {-3}},
            Ticks → {{{3, "10^3"}, {4, "10^4"}, {5, "10^5"}, {6, "10^6"}}, Automatic}];
```

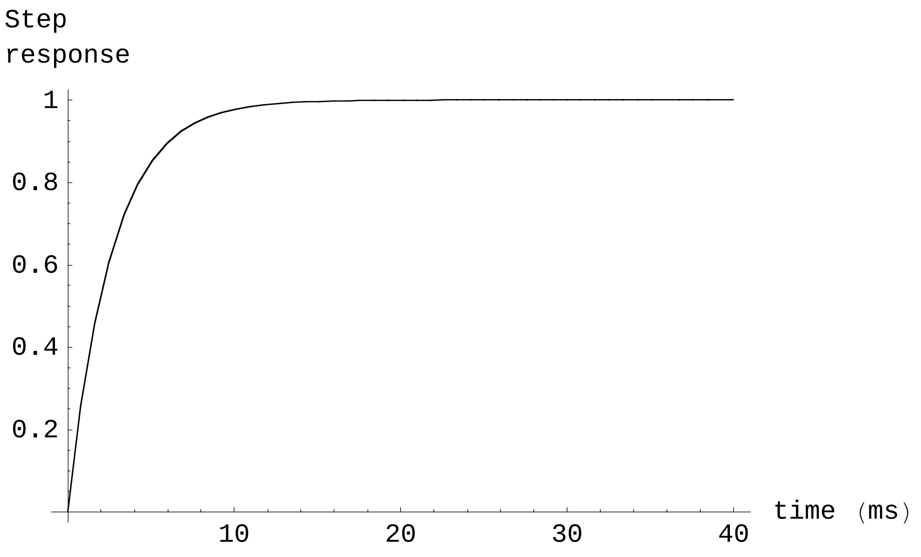


The step response follows:

```
In[51]:=  stepResp = InverseLaplaceTransform[H / s /. values, s, t]
```

Out[51]= $4\,(0.25 - 0.249976\,e^{-376979.\,t} - 0.0000239502\,e^{-80934.4\,t})$

```
In[52]:=  Plot[stepResp /. t → x * 10^(-6), {x, 0, 40},
            PlotRange → All, AxesLabel → {"time (ms)", "Step\nresponse"}];
```



## Shuttle Pitch Control

Find the transfer function matrix of a pitch control MIMO system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[53]:=  Needs["SchematicSolver`"];
```

Here is an example of a *shuttle pitch control* system that incorporates feedback to control the pitch of a vehicle. Measurements are made by the vehicle's inertial unit, gyros and accelerometers. A simplified model of a pitch controller is shown for the space shuttle (Nise, N.S., *Control Systems Engineering*, 3/e, John Wiley and Sons, New York, NY, 2000):

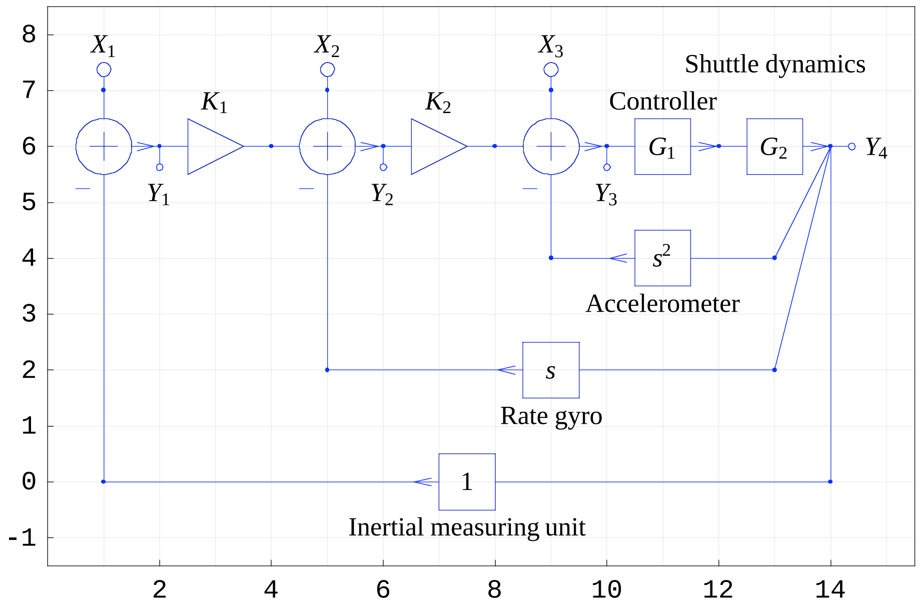
```
In[54]:=  shuttlePitchController = {
             {"Input", {1, 7}, X1, "Commanded pitch", TextOffset → {0, -1}},
             {"Input", {5, 7}, X2, "Commanded pitch rate", TextOffset → {0, -1}},
             {"Input", {9, 7}, X3, "Commanded pitch acceleration", TextOffset → {0, -1}},
             {"Adder", {{0, 6}, {1, 0}, {2, 6}, {1, 7}}, {0, -1, 2, 1}},
             {"Adder", {{4, 6}, {5, 2}, {6, 6}, {5, 7}}, {1, -1, 2, 1}},
             {"Adder", {{8, 6}, {9, 4}, {10, 6}, {9, 7}}, {1, -1, 2, 1}},
             {"Amplifier", {{2, 6}, {4, 6}}, K1},
             {"Amplifier", {{6, 6}, {8, 6}}, K2},
             {"Block", {{10, 6}, {12, 6}}, G1, "Controller"},
             {"Block", {{12, 6}, {14, 6}}, G2, "Shuttle dynamics\n"},
             {"Block", {{14, 0}, {1, 0}}, 1, "Inertial measuring unit"},
             {"Block", {{13, 2}, {5, 2}}, s, "Rate gyro"},
             {"Block", {{13, 4}, {9, 4}}, s^2, "Accelerometer"},
             {"Output", {2, 6}, Y1, "Pitch error", TextOffset → {0, 1}},
             {"Output", {6, 6}, Y2, "Pitch rate error", TextOffset → {0, 1}},
             {"Output", {10, 6}, Y3, "Pitch acceleration error", TextOffset → {0, 1}},
             {"Output", {14, 6}, Y4, "Pitch"},
             {"Line", {{14, 6}, {14, 0}}},
             {"Line", {{14, 6}, {13, 2}}},
             {"Line", {{14, 6}, {13, 4}}}
           };
```

It is better typeset with

```
In[55]:=  SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];
```

```
In[56]:=  typoSubst = {G1 → G_1, G2 → G_2, K1 → K_1, K2 → K_2,
             X1 → X_1, X2 → X_2, X3 → X_3,
             Y1 → Y_1, Y2 → Y_2, Y3 → Y_3, Y4 → Y_4};
```

```
In[57]:=  ShowSchematic[shuttlePitchController /. typoSubst,
            PlotRange → {{0, 15.5}, {-1.5, 8.5}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of this three-input four-output system:

```
In[58]:=  {tfMatrix, systemInp, systemOut} =
            ContinuousSystemTransferFunction[shuttlePitchController];
```

```
In[59]:=  tfMatrix /. typoSubst // Together // TraditionalForm
```

*Out[59]//TraditionalForm=*

$$\begin{pmatrix} \frac{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+1}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & -\frac{G_1\,G_2\,K_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & -\frac{G_1\,G_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} \\ \frac{G_1\,G_2\,K_1\,s^2+K_1}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & \frac{G_1\,G_2\,s^2+1}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & \frac{-s\,G_1\,G_2-G_1\,K_1\,G_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} \\ \frac{K_1\,K_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & \frac{K_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & \frac{1}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} \\ \frac{G_1\,G_2\,K_1\,K_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & \frac{G_1\,G_2\,K_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} & \frac{G_1\,G_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1} \end{pmatrix}$$

System inputs are the commanded pitch $X_1$, commanded pitch rate $X_2$, and commanded pitch acceleration $X_3$. They correspond to the following nodes, respectively:

```
In[60]:=  systemInp

Out[60]=  {Y[{1, 7}], Y[{5, 7}], Y[{9, 7}]}
```

System outputs are the pitch error $Y_1$, pitch rate error $Y_2$, pitch acceleration error $Y_3$, and pitch $Y_4$. They correspond to the following nodes, respectively:

```
In[61]:=  systemOut

Out[61]=  {Y[{2, 6}], Y[{6, 6}], Y[{10, 6}], Y[{14, 6}]}
```

Here is the transfer function from the commanded pitch input $X_1$ to the actual pitch output $Y_4$:

```
In[62]:=  H = tfMatrix[[4, 1]];
          H /. typoSubst // TraditionalForm
```

*Out[63]//TraditionalForm=*

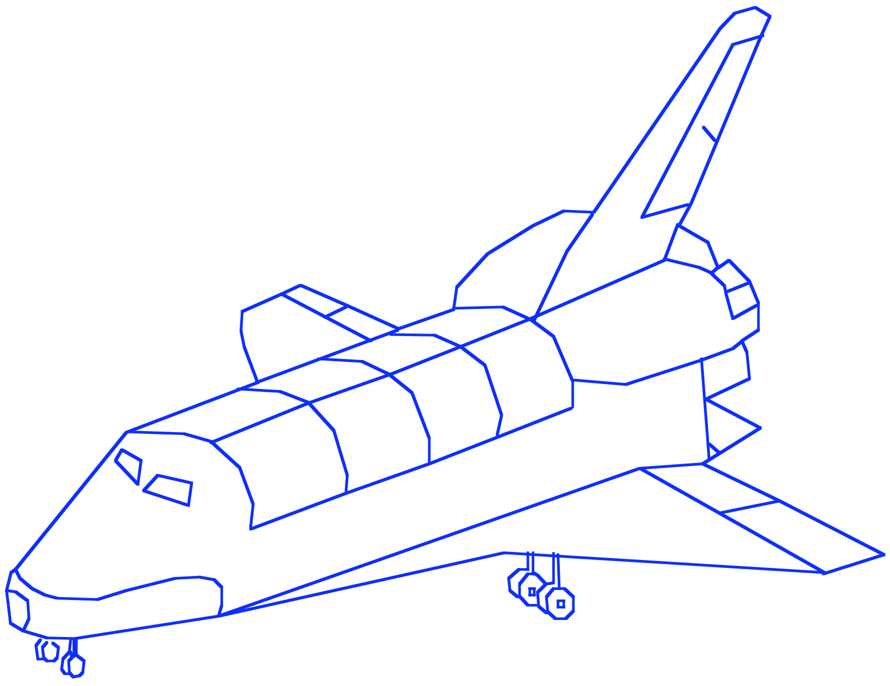
$$\frac{G_1\,G_2\,K_1\,K_2}{G_1\,G_2\,s^2+G_1\,G_2\,K_2\,s+G_1\,G_2\,K_1\,K_2+1}$$

*SchematicSolver* can be used to draw lineart figures. Let us restore the default plot style options:

```
In[64]:=  SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleDefault];
```

Here is a predefined specification to sketch the shuttle:

```
In[65]:=  ShowSchematic[SchematicSolverFigureShuttle, Frame → False, GridLines → None];
```



You can add various annotations to the above lineart figure. For example, you can use the *SchematicSolver*'s Arrow element and Text element to add labels. The corresponding coordinates, you can obtain with the *SchematicSolver*'s palettes.

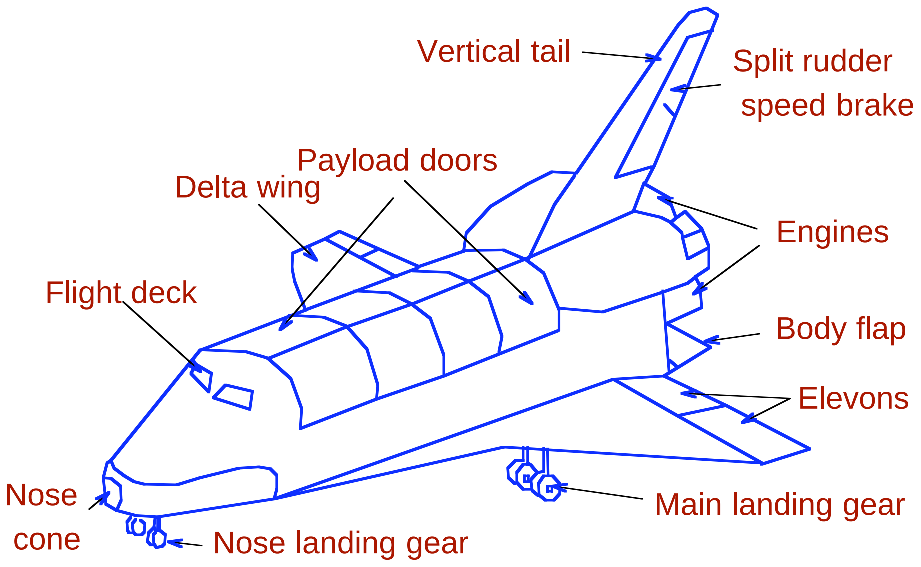
```
In[66]:=  shuttleAnnotations = {
             {"Arrow", {{13, 61}, {-9, 42}}},
             {"Arrow", {{674, 228}, {734, 237}}},
             {"Arrow", {{243, 318}, {179, 380}}},
             {"Arrow", {{715, 138}, {768, 165}}},
             {"Arrow", {{648, 171}, {768, 165}}},
             {"Arrow", {{661, 281}, {734, 335}}},
             {"Arrow", {{622, 388}, {731, 353}}},
             {"Arrow", {{115, 195}, {29, 272}}},
             {"Arrow", {{505, 68}, {604, 54}}},
             {"Arrow", {{79, 6}, {116, 2}}},
             {"Arrow", {{482, 270}, {341, 406}}},
             {"Arrow", {{203, 241}, {328, 387}}},
             {"Arrow", {{637, 507}, {691, 513}}},
             {"Arrow", {{624, 541}, {538, 549}}},
             {"Text", {531, 550}, "Vertical tail", TextOffset → {1, 0}},
             {"Text", {698, 515}, "Split rudder\n speed brake", TextOffset → {-1, 0}},
             {"Text", {771, 166}, "Elevons", TextOffset → {-1, 0}},
             {"Text", {747, 350}, "Engines", TextOffset → {-1, 0}},
             {"Text", {333, 430}, "Payload doors"},
             {"Text", {167, 400}, "Delta wing"},
             {"Text", {27, 283}, "Flight deck"},
             {"Text", {612, 48}, "Main landing gear", TextOffset → {-1, 0}},
             {"Text", {122, 5}, "Nose landing gear", TextOffset → {-1, 0}},
             {"Text", {746, 243}, "Body flap", TextOffset → {-1, 0}},
             {"Text", {-11, 34}, "Nose\n cone", TextOffset → {1, 0}}
           };
```

It is better typeset with

```
In[67]:= SetOptions[DrawElement,
           TextStyle → {FontFamily → "Helvetica", FontColor → RGBColor[0.6, 0, 0]}
          ];
```

Use `Join` to combine the predefined lineart figure and your annotations:

```
In[68]:= ShowSchematic[SchematicSolverFigureShuttle ~ Join ~ shuttleAnnotations,
          Frame → False, GridLines → None, ElementScale → 50]
```



**Body flap** – a flap located at the bottom rear of the shuttle; it provides a thermal shield for the engines during re-entry and also provides pitch control (the movement of the nose up and down) during atmospheric flight (landings).

**Delta wing** – the triangular-shaped wings on either side of the Space Shuttle.

**Elevons** – flaps located on the trailing edge of each wing, used during atmospheric flight; elevons are used to control pitch (the movement of the nose up and down) and roll. The elevons work only in the presence of air (they do not work in space).

**Engines** – engines are located at the rear of the shuttle and are used to maneuver the Space Shuttle into orbit, to make adjustments while in orbit, and to control the Space Shuttle during re-entry into the Earth's atmosphere: controlling the roll, pitch (the movement of the nose up and down), and yaw (the movement of the nose to the left and right) in the absence of air.

**Flight deck** – the part of the Space Shuttle in which the astronauts travel.

**Main landing gear** – the landing gear (wheels used for landing) located at the rear of the Space Shuttle.

**Nose cone** – the front of the Space Shuttle.

**Nose landing gear** – the landing gear (wheels used for landing) located at the front of the Space Shuttle.

**Payload doors** – the doors of the large storage compartment of the Space Shuttle; items like satellites can be carried into space in the cargo (payload) bay.

**Split rudder/speed brake** – a divided flap located on the trailing edge of the tail fin, used during atmospheric flight. This two-part rudder steers the shuttle, controls yaw (the movement of the nose to the left and right), and acts as a brake (when the split rudder is opened like book). The rudder works only in the presence of air.

**Vertical tail** – the fin at the top rear of the shuttle. It provides stability while flying.

This restores default drawing options:

```
In[69]:= SetOptions[DrawElement,
           TextStyle → {FontFamily → "Times", FontColor → RGBColor[0, 0, 0]}
          ];
```

## 5.2. Symbolic Optimization of a Continuous-Time System

Find the optimal value of a selected system parameter for a given value of the steady-state response.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[70]:=  Needs["SchematicSolver`"];
```

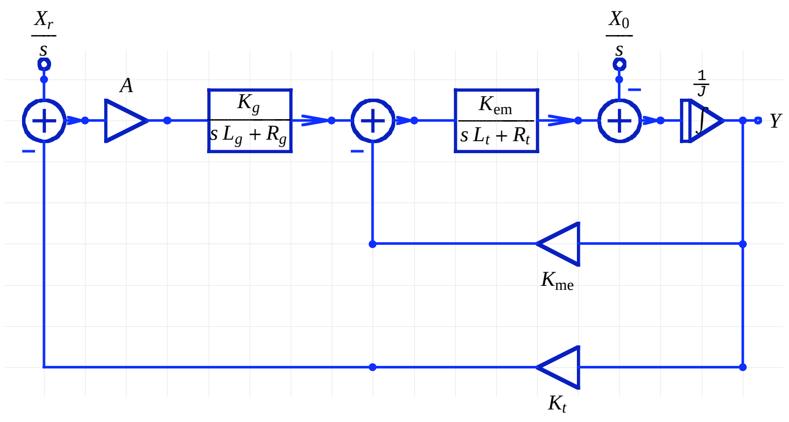
Consider a continuous-time *velocity servo system* represented by the following schematic:

```
In[71]:=  velocityServoSchematic = {
            {"Line", {{18, 3}, {18, 0}}},
            {"Line", {{18, 6}, {18, 3}}},
            {"Adder", {{0, 6}, {9, 0}, {2, 6}, {1, 7}}, {0, -1, 2, 1}},
            {"Adder", {{8, 6}, {9, 3}, {10, 6}, {9, 7}}, {1, -1, 2, 0}},
            {"Adder", {{14, 6}, {15, 5}, {16, 6}, {15, 7}}, {1, 0, 2, -1}},
            {"Amplifier", {{2, 6}, {4, 6}}, A},
            {"Amplifier", {{18, 0}, {9, 0}}, Kt},
            {"Amplifier", {{18, 3}, {9, 3}}, Kme},
            {"Input", {1, 7}, Xr / s, "", TextOffset → {0, -1}},
            {"Integrator", {{16, 6}, {18, 6}}, 1 / J},
            {"Output", {18, 6}, Y},
            {"Block", {{4, 6}, {8, 6}}, Kg / (Rg + s * Lg), "", ElementSize → {2, 1.5}},
            {"Block", {{10, 6}, {14, 6}}, Kem / (Rt + s * Lt), "", ElementSize → {2, 1.5}},
            {"Input", {15, 7}, X0 / s, "", TextOffset → {0, -1}}
          };
```

It is better typeset with

```
In[72]:=  typoSubst = {Kem → K_em, Kg → K_g, Kme → K_me, Kt → K_t,
            Lg → L_g, Lt → L_t, Rg → R_g, Rt → R_t,
            X0 → X_0, Xr → X_r, Yref → Y_ref};
```

```
In[73]:=  ShowSchematic[velocityServoSchematic /. typoSubst, FontSize → 8, Frame → False];
```



The system has two inputs and one output. Both inputs are step stimuli whose transforms are $\frac{X_r}{s}$ and $\frac{X_0}{s}$.

`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
In[74]:=  {tfMatrix, systemInp, systemOut} =
            ContinuousSystemTransferFunction[velocityServoSchematic];
```

The transfer functions of this two-input single-output system are the elements of the transfer function matrix:

```
In[75]:=  H1 = tfMatrix[[1, 1]] // Together;
          % /. typoSubst // TraditionalForm
```

Out[76]//TraditionalForm=

$$(A K_{\text{em}} K_g)/(J L_g L_t s^3 + J L_t R_g s^2 + J L_g R_t s^2 + K_{\text{em}} K_{\text{me}} L_g s + J R_g R_t s + A K_{\text{em}} K_g K_t + K_{\text{em}} K_{\text{me}} R_g)$$

```
In[77]:=  H2 = tfMatrix[[1, 2]] // Together;
          % /. typoSubst // TraditionalForm
```

Out[78]//TraditionalForm=

$$-((s L_g + R_g)(s L_t + R_t))/(J L_g L_t s^3 + J L_t R_g s^2 + J L_g R_t s^2 + K_{\text{em}} K_{\text{me}} L_g s + J R_g R_t s + A K_{\text{em}} K_g K_t + K_{\text{em}} K_{\text{me}} R_g)$$

`ContinuousSystemResponse` finds the system response at all nodes:

```
In[79]:=  {systemResponse, systemVars} = ContinuousSystemResponse[velocityServoSchematic];
```

Here is the transform of the output signal:

```
In[80]:=  Yout = systemOut[[1]] /. systemResponse // Together;
          % /. typoSubst // TraditionalForm
```

Out[81]//TraditionalForm=

$$(-L_g L_t X_0 s^2 - L_t R_g X_0 s - L_g R_t X_0 s - R_g R_t X_0 + A K_{\text{em}} K_g X_r)/$$
$$(s(J L_g L_t s^3 + J L_t R_g s^2 + J L_g R_t s^2 + K_{\text{em}} K_{\text{me}} L_g s + J R_g R_t s + A K_{\text{em}} K_g K_t + K_{\text{em}} K_{\text{me}} R_g))$$

*SchematicSolver* has a unique feature: it finds the symbolic response keeping all system parameters as symbols. The response `Yout` is closed-form and purely symbolic. All system parameters are given by symbols, so the obtained result is the most general.

You can find the optimal value of a selected parameter for the given steady-state value. For instance, the gain of the amplifier `A` can be optimized to provide the output steady state of some value `Yref`. Notice than no numerical value appears in the calculation.

```
In[82]:=  Aopt = A /. First[Solve[Limit[s * Yout, s → 0] == Yref, A]];
          % /. typoSubst // TraditionalForm
```

Out[83]//TraditionalForm=

$$\frac{R_g R_t X_0 + K_{\text{em}} K_{\text{me}} R_g Y_{\text{ref}}}{K_{\text{em}} K_g (X_r - K_t Y_{\text{ref}})}$$

For a particular set of numeric values

```
In[84]:=  parameterValues = {J → 0.005, Kem → 0.0005, Kg → 20, Kme → 0.0005, Kt → 0.1,
             Lg → 10, Lt → 0.020, Rg → 25, Rt → 1.6, X0 → 2.5, Xr → 150, Yref → 150};
```

the optimum gain becomes

```
In[85]:=  Aopt /. parameterValues

Out[85]=  74.0748
```

Substituting the numeric values into the symbolic expression `Yout` yields

```
In[86]:=  numericYout = Yout /. A → Aopt /. parameterValues
```

Out[86]=

$$\frac{11.1122 - 41.25\,s - 0.5\,s^2}{s\,(0.074081 + 0.200003\,s + 0.0825\,s^2 + 0.001\,s^3)}$$

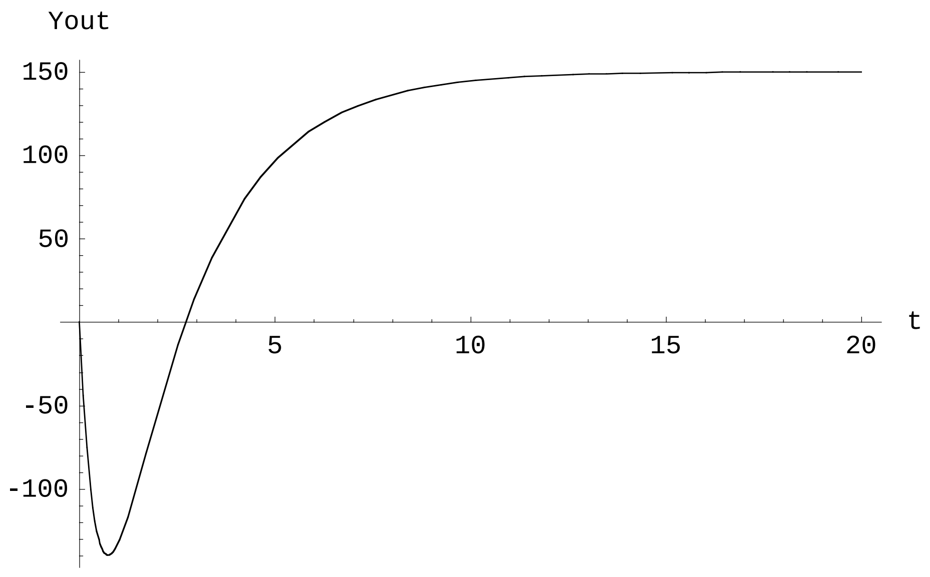
Using the inverse Laplace transform provided by *Mathematica* the time response can be found:

```
In[87]:= timeYout = InverseLaplaceTransform[numericYout, s, t]
```

Out[87]= $150. - 0.222917\, e^{-80.0119\, t} + 371.62\, e^{-2.03257\, t} - 521.397\, e^{-0.45552\, t}$

The corresponding graph of this waveform proves the expected steady state value:

```
In[88]:= Plot[timeYout, {t, 0, 20}, AxesLabel → {"t", "Yout"}];
```



## 5.3. Design of a Continuous-Time System from the Step Response

Linear system can be designed in a straightforward manner if its step response is known as a closed-form expression.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[89]:= Needs["SchematicSolver`"];
```

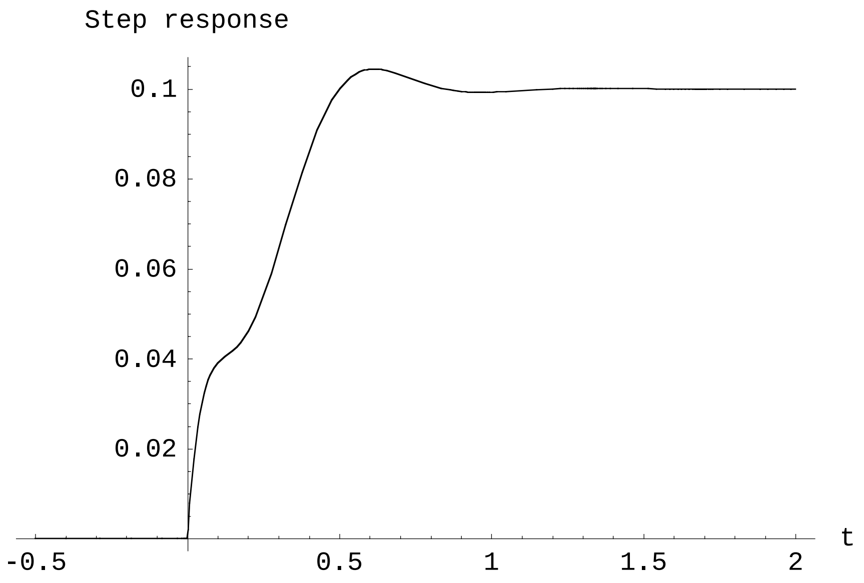
Here is the step response of a linear continuous-time system:

In[90]:= **stepResponse =** $\left(\frac{1}{10} - \frac{1}{5}e^{-10\,t} + \frac{1}{30}e^{-5\,t}\left(3\,\mathrm{Cos}\left[5\sqrt{3}\,t\right] - \sqrt{3}\,\mathrm{Sin}\left[5\sqrt{3}\,t\right]\right)\right)$ **UnitStep[t];**
**% // TraditionalForm**

*Out[91]//TraditionalForm=*

$$\left(\frac{1}{30}e^{-5\,t}\left(3\cos\left(5\sqrt{3}\,t\right) - \sqrt{3}\,\sin\left(5\sqrt{3}\,t\right)\right) - \frac{e^{-10\,t}}{5} + \frac{1}{10}\right)\theta(t)$$

```
In[92]:= Plot[stepResponse, {t, -0.5, 2}, PlotRange → All, AxesLabel → {"t", "Step response"}];
```



First, find the corresponding transfer function from the formula $r(t) = \mathcal{L}^{-1}\,\frac{1}{s}\,H(s)$, or equivalently, $H(s) = s\,\mathcal{L}\,r(t)$, where $\mathcal{L}$ represents the Laplace transform, $r(t)$ denotes the step response, $H(s)$ is the transfer function, and $s$ stands for the complex frequency:

```
In[93]:= stepResponseLT = LaplaceTransform[stepResponse, t, s]
```

*Out[93]=* $$\frac{1}{10\,s} - \frac{1}{5\,(10+s)} - \frac{1}{2\,(100+10\,s+s^2)} + \frac{5+s}{10\,(100+10\,s+s^2)}$$

```
In[94]:= transferFunction = s * stepResponseLT // Together;
         % // TraditionalForm
```

*Out[95]//TraditionalForm=*

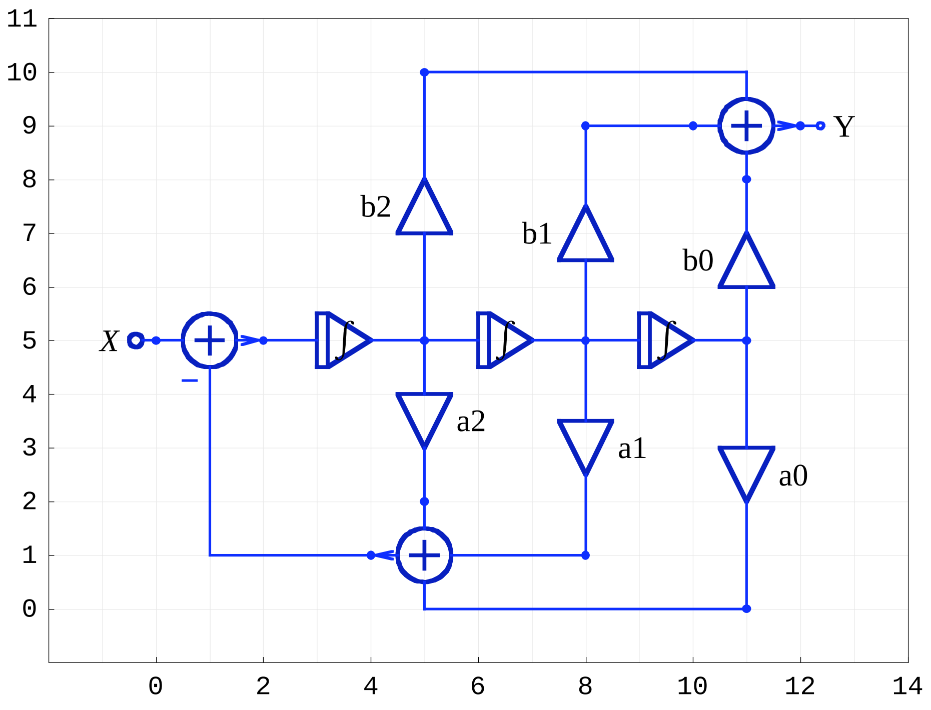
$$\frac{s^2+100}{(s+10)\,(s^2+10\,s+100)}$$

Second, design the system from the transfer function. Minimal number of integrators equals the order of the transfer function

```
In[96]:=  numTF = Numerator[transferFunction]

Out[96]=  100 + s^2

In[97]:=  denTF = Denominator[transferFunction] // Expand

Out[97]=  1000 + 200 s + 20 s^2 + s^3

In[98]:=  numberOfIntegrators = Max[Exponent[numTF, s], Exponent[denTF, s]]

Out[98]=  3
```

The general block-diagram of this system, with symbolic parameters and 3 integrators, can be represented by the following schematic:

```
In[99]:=  generalSchematic3 = {
              {"Input", {0, 5}, X},
              {"Output", {12, 9}, "Y"},
              {"Integrator", {{2, 5}, {5, 5}}, 1},
              {"Integrator", {{5, 5}, {8, 5}}, 1},
              {"Integrator", {{8, 5}, {11, 5}}, 1},
              {"Amplifier", {{5, 5}, {5, 2}}, a2},
              {"Amplifier", {{8, 5}, {8, 1}}, a1},
              {"Amplifier", {{11, 5}, {11, 0}}, a0},
              {"Amplifier", {{5, 5}, {5, 10}}, b2},
              {"Amplifier", {{8, 5}, {8, 9}}, b1},
              {"Amplifier", {{11, 5}, {11, 8}}, b0},
              {"Adder", {{0, 5}, {4, 1}, {2, 5}, {1, 6}}, {1, -1, 2, 0}},
              {"Adder", {{4, 1}, {11, 0}, {8, 1}, {5, 2}}, {2, 1, 1, 1}},
              {"Adder", {{10, 9}, {11, 8}, {12, 9}, {5, 10}}, {1, 1, 2, 1}},
              {"Line", {{8, 9}, {10, 9}}}};
           ShowSchematic[%, PlotRange → {{-2, 14}, {-1, 11}}];
```



`ContinuousSystemTransferFunction` computes the transfer function matrix of the system:

```
  In[101]:=
       {tfMatrix, systemInp, systemOut} =
        ContinuousSystemTransferFunction[generalSchematic3];
```

The transfer function of this single-input single-output system is the element of the transfer function matrix:

```
In[102]:=
      H = tfMatrix[[1, 1]]
```

```
Out[102]=
```
$$\frac{b0 + b1\, s + b2\, s^2}{a0 + a1\, s + a2\, s^2 + s^3}$$

Numeric coefficient values are found as follows.

`Numerator` picks out the numerator of the transfer function:

```
In[103]:=
      numH = Numerator[H]
```

```
Out[103]=
```
$b0 + b1\, s + b2\, s^2$

`Denominator` picks out the denominator of the transfer function:

```
In[104]:=
      denH = Denominator[H]
```

```
Out[104]=
```
$a0 + a1\, s + a2\, s^2 + s^3$

`CoefficientList` finds a list of the coefficients of polynomials:

```
In[105]:=
      CoefficientList[denH, s]

Out[105]=
      {a0, a1, a2, 1}
```

Note that the leading coefficient equals 1.

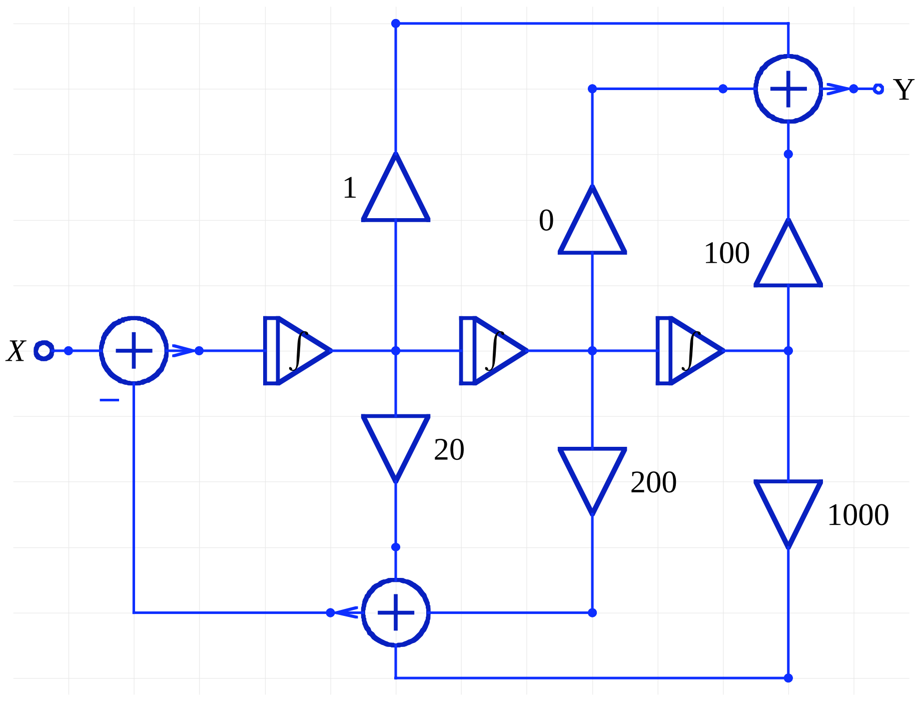
For the known transfer function coefficients, that are computed from the step response, and for the symbolic coefficients, that are computed from the general schematic of the system, we compute the system parameters as follows:

```
In[106]:=
      numParameters =
       Solve[CoefficientList[numTF, s] == CoefficientList[numH, s], {b0, b1, b2}] // Flatten

Out[106]=
      {b0 → 100, b1 → 0, b2 → 1}

In[107]:=
      denParameters =
       Solve[CoefficientList[denTF, s] == CoefficientList[denH, s], {a0, a1, a2}] // Flatten

Out[107]=
      {a0 → 1000, a1 → 200, a2 → 20}

In[108]:=
      systemParameters = Join[numParameters, denParameters]

Out[108]=
      {b0 → 100, b1 → 0, b2 → 1, a0 → 1000, a1 → 200, a2 → 20}
```

The schematic specification of the system with numeric parameters is

```
In[109]:=
      numericSchematic3 = generalSchematic3 /. systemParameters;
```

```
In[110]:=
      ShowSchematic[numericSchematic3, Frame → False];
```



`ContinuousSystemTransferFunction` finds the transfer function of the above system:

```
In[111]:=
      {tfMatrix, systemInp, systemOut} =
        ContinuousSystemTransferFunction[numericSchematic3];
      numericH = tfMatrix[[1, 1]]
```

```
Out[112]=
```

$$\frac{100+s^2}{1000+200\,s+20\,s^2+s^3}$$

The corresponding step response can be computed as the inverse Laplace transform of `numericH/s`:

```
In[113]:=
      numericStepResponse = InverseLaplaceTransform[numericH / s, s, t] // Simplify
```

```
Out[113]=
```

$$\frac{1}{30}\left(3-6\,e^{-10\,t}+e^{-5\,t}\left(3\,\mathrm{Cos}[5\sqrt{3}\,t]-\sqrt{3}\,\mathrm{Sin}[5\sqrt{3}\,t]\right)\right)$$

`FullSimplify` proves that, for positive time, the step response computed from the schematic is the same as the given step response:

```
In[114]:=
      FullSimplify[stepResponse - numericStepResponse, t > 0]
```

```
Out[114]=
      0
```

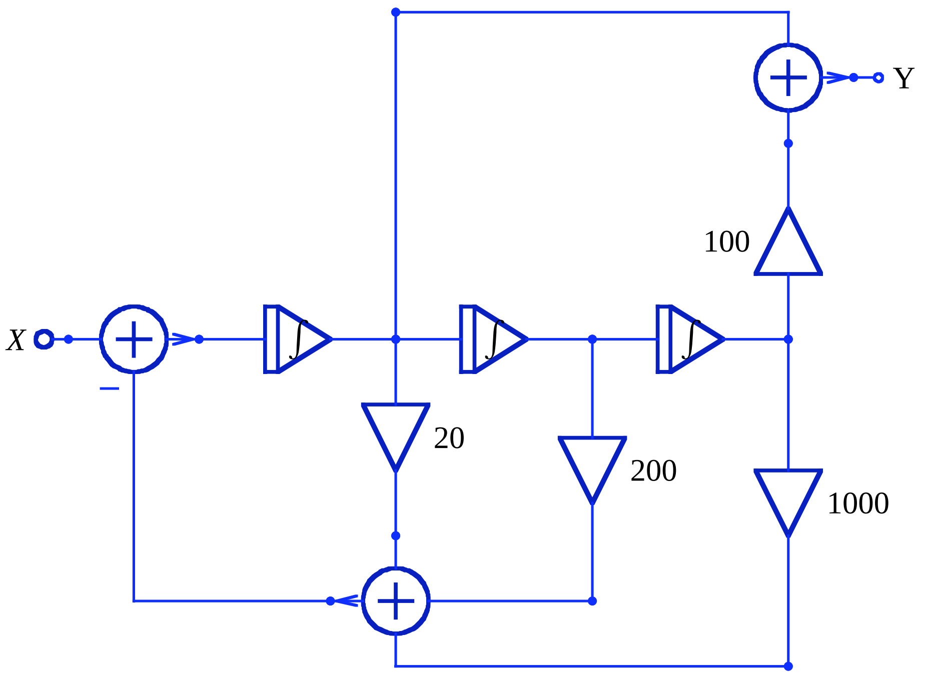
The system representation can be simplified by

- removing the Amplifier elements with zero gain

- replacing the unity-gain Amplifier elements with the Line elements.

```
In[115]:=
      simpleSchematic3 = {
         {"Input", {0, 5}, X},
         {"Output", {12, 9}, "Y"},
         {"Integrator", {{2, 5}, {5, 5}}, 1},
         {"Integrator", {{5, 5}, {8, 5}}, 1},
         {"Integrator", {{8, 5}, {11, 5}}, 1},
         {"Amplifier", {{5, 5}, {5, 2}}, a2},
         {"Amplifier", {{8, 5}, {8, 1}}, a1},
         {"Amplifier", {{11, 5}, {11, 0}}, a0},
         {"Line", {{5, 5}, {5, 10}}},
         {"Amplifier", {{11, 5}, {11, 8}}, b0},
         {"Adder", {{0, 5}, {4, 1}, {2, 5}, {1, 6}}, {1, -1, 2, 0}},
         {"Adder", {{4, 1}, {11, 0}, {8, 1}, {5, 2}}, {2, 1, 1, 1}},
         {"Adder", {{10, 9}, {11, 8}, {12, 9}, {5, 10}}, {0, 1, 2, 1}}};
```

```
In[116]:=
      ShowSchematic[simpleSchematic3 /. systemParameters, Frame → False, GridLines → None];
```



Obviously, the transfer function remains the same:

```
In[117]:=
      {tfMatrix, systemInp, systemOut} =
        ContinuousSystemTransferFunction[simpleSchematic3];
      simpleH = tfMatrix[[1, 1]] /. systemParameters
```

Out[118]=

$$\frac{100 + s^2}{1000 + 200\, s + 20\, s^2 + s^3}$$

```
In[119]:=
      SameQ[numericH, simpleH]
```

```
Out[119]=
      True
```

## 5.4. Automated Drawing and Solving of General Systems

*SchematicSolver* comes with functions that create schematics important for practice. You can easily build new models from these automatically generated schematics.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[120]:=
      Needs["SchematicSolver`"];
```

We shall adjust some options to obtain better appearance of the example schematics:

```
In[121]:=
      SetOptions[ShowSchematic, Frame → False, GridLines → None];
```

Typically, for the specified system order

```
In[122]:=
      systemOrder = 3;
```

the system parameters can be generated automatically with

```
In[123]:=
      paramsNum = UnitSymbolicSequence[systemOrder, a, 0] // Flatten // Reverse
Out[123]=
      {a2, a1, a0}
```

```
In[124]:=
      paramsDen = UnitSymbolicSequence[systemOrder + 1, b, 0] // Flatten // Reverse
Out[124]=
      {b3, b2, b1, b0}
```

Here is an example schematic specification of a discrete system that is generated automatically for the specified system parameters:

```
In[125]:=
      {schematicSpec, inputCoordinates, outputCoordinates} =
       TransposedDirectForm2IIRFilterSchematic[{paramsDen, paramsNum}];
```

You should add input and output to form the system:

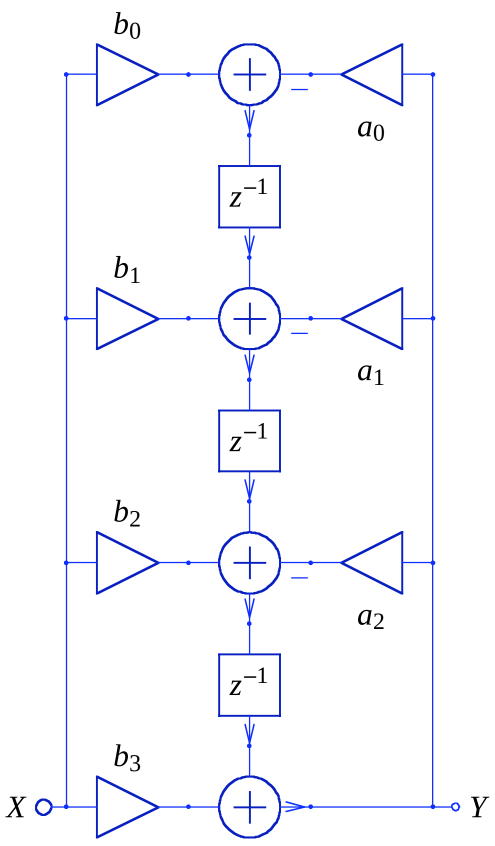
```
In[126]:=
      discreteSchematic = Join[schematicSpec, {
          {"Input", inputCoordinates[[1]], X},
          {"Output", outputCoordinates[[1]], Y}
         }];
```

Note that the coordinates of input and output have been returned by `DirectFormFIRFilterSchematic`.

For better typesetting, you may use

```
In[127]:=
      typoSubst = {a0 → a₀, a1 → a₁, a2 → a₂, b0 → b₀, b1 → b₁, b2 → b₂, b3 → b₃};
```

```
In[128]:=
      ShowSchematic[discreteSchematic /. typoSubst]
```



`DiscreteSystemTransferFunction` finds the transfer function directly from the schematic:

```
In[129]:=
      {tfMatrix, myInputs, myOutputs} =
        DiscreteSystemTransferFunction[discreteSchematic];
      % /. typoSubst // DiscreteSystemDisplayForm
```

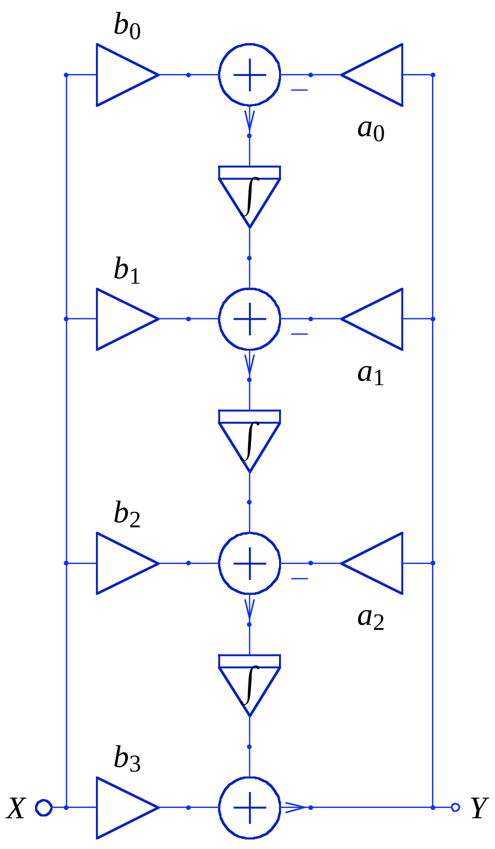
```
Out[130]//DisplayForm=
```

$$\frac{b_3 + b_2\, z^{-1} + b_1\, z^{-2} + b_0\, z^{-3}}{1 + a_2\, z^{-1} + a_1\, z^{-2} + a_0\, z^{-3}}$$

The graphical representation of a system is not a frozen picture. Once when you have a schematic of the discrete-time system, and when you find out that the same structure can be used to build a schematic of a continuous-time system, you can do that by the following simple replacements:

- the Delay element is replaced by the Integrator element
- the Multiplier element is replaced by the Amplifier element

```
In[131]:=
      continuousSchematic =
        discreteSchematic /. {"Delay" → "Integrator", "Multiplier" → "Amplifier"};
      % /. typoSubst // ShowSchematic
```



`ContinuousSystemTransferFunction` finds the transfer function directly from the schematic:

```
In[133]:=
       {tfMatrix, myInputs, myOutputs} =
         ContinuousSystemTransferFunction[continuousSchematic];
       tfMatrix[[1, 1]] /. typoSubst // Together // TraditionalForm

Out[134]//TraditionalForm=
```

$$\frac{b_3\, s^3 + b_2\, s^2 + b_1\, s + b_0}{s^3 + a_2\, s^2 + a_1\, s + a_0}$$

Automated drawing and solving of general continuous-time and discrete-time systems is a unique feature of *Schematic-Solver* not available in other software for system modeling and analysis.

This restores default drawing options:

```
In[135]:=
      SetOptions[ShowSchematic, Frame → True, GridLines → Automatic];
```

## 5.5. Discrete-Time Systems

### Introduction

*SchematicSolver* has many unique features not available in other software: symbolic signal processing brings you

- Computation of transfer functions as closed-form expressions in terms of symbolic system parameters
- Finding the closed-form response from the schematic

The derived result is the most general because all system parameters and inputs can be given by symbols.

Other important features include building models from automatically generated schematics; you can change system parameters on the fly and immediately see what happens with the results.

See other chapters for illustrations of unique features not available in other software:

Chapter 6 Solving Large Systems

Chapter 9 Examples of Discrete System Implementation

Chapter 10 Hilbert Transformer

Chapter 11 Multirate Systems

Chapter 12 Hierarchical Systems

Chapter 15 Processing with *SchematicSolver*

*SchematicSolver*'s powerful functions for solving discrete-time (digital) systems are illustrated by the subsequent examples.

## Direct Form 2 Transposed IIR Filter

Find the transfer function of a digital filter realization known as *direct form 2 transposed IIR*.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[136]:=
     Needs["SchematicSolver`"];
```

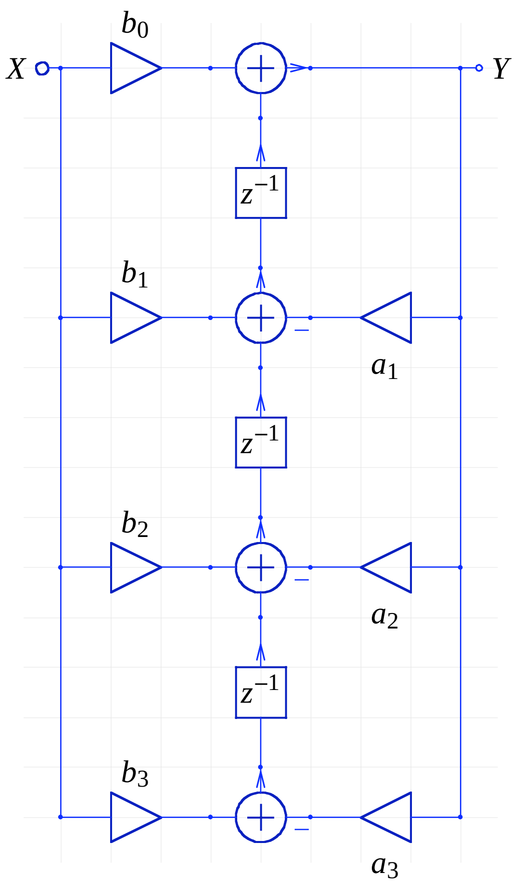
Here is the schematic:

```
In[137]:=
     DirectForm2TransposedIIRSchematic = {
        {"Input", {0, 15}, X},
        {"Output", {8, 15}, Y},
        {"Multiplier", {{8, 10}, {5, 10}}, a1},
        {"Multiplier", {{8, 5}, {5, 5}}, a2},
        {"Multiplier", {{8, 0}, {5, 0}}, a3},
        {"Multiplier", {{0, 15}, {3, 15}}, b0},
        {"Multiplier", {{0, 10}, {3, 10}}, b1},
        {"Multiplier", {{0, 5}, {3, 5}}, b2},
        {"Multiplier", {{0, 0}, {3, 0}}, b3},
        {"Delay", {{4, 1}, {4, 4}}, 1},
        {"Delay", {{4, 6}, {4, 9}}, 1},
        {"Delay", {{4, 11}, {4, 14}}, 1},
        {"Adder", {{3, 0}, {4, -1}, {5, 0}, {4, 1}}, {1, 0, -1, 2}},
        {"Adder", {{3, 5}, {4, 4}, {5, 5}, {4, 6}}, {1, 1, -1, 2}},
        {"Adder", {{3, 10}, {4, 9}, {5, 10}, {4, 11}}, {1, 1, -1, 2}},
        {"Adder", {{3, 15}, {4, 14}, {5, 15}, {4, 16}}, {1, 1, 2, 0}},
        {"Line", {{0, 5}, {0, 0}}}, {"Line", {{0, 10}, {0, 5}}},
        {"Line", {{0, 15}, {0, 10}}}, {"Line", {{8, 15}, {5, 15}}},
        {"Line", {{8, 5}, {8, 0}}}, {"Line", {{8, 10}, {8, 5}}},
        {"Line", {{8, 15}, {8, 10}}}
       };
```

It is better typeset with

```
In[138]:=
      typoSubst = {a1 → a₁, a2 → a₂, a3 → a₃, b0 → b₀, b1 → b₁, b2 → b₂, b3 → b₃};
```

```
In[139]:=
      ShowSchematic[DirectForm2TransposedIIRSchematic /. typoSubst, Frame → False];
```



`DiscreteSystemTransferFunction` computes the transfer function matrix of the system:

```
In[140]:=
      {tfMatrix, systemInp, systemOut} =
        DiscreteSystemTransferFunction[DirectForm2TransposedIIRSchematic];
```

The transfer function of this single-input single-output system is the element of the transfer function matrix:

```
In[141]:=
      df2TF = tfMatrix[[1, 1]];
      df2TF /. typoSubst // Together // TraditionalForm
```

```
Out[142]//TraditionalForm=
```

$$\frac{b_0\, z^3 + b_1\, z^2 + b_2\, z + b_3}{z^3 + a_1\, z^2 + a_2\, z + a_3}$$

*SchematicSolver* can express transfer functions of discrete systems in terms of $z^{-1}$ with its function `DiscreteSystem-DisplayForm`:

```
In[143]:=
      DiscreteSystemDisplayForm[df2TF /. typoSubst]
```

```
Out[143]//DisplayForm=
```

$$\frac{b_0 + b_1\, z^{-1} + b_2\, z^{-2} + b_3\, z^{-3}}{1 + a_1\, z^{-1} + a_2\, z^{-2} + a_3\, z^{-3}}$$

## State-Space Model of Discrete System

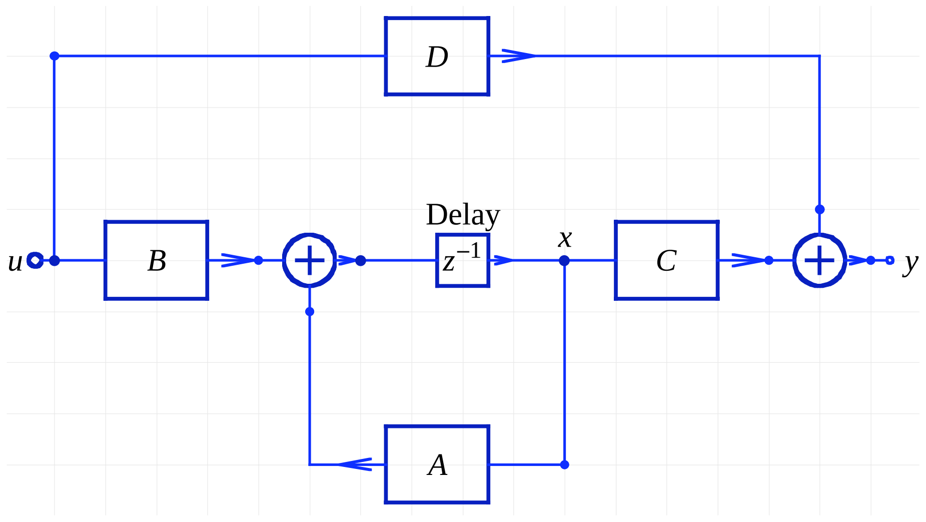
Compute the transfer function for the *state-space model* of the discrete-time system shown below.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[144]:=
      Needs["SchematicSolver`"];
```

Here is the system schematic:

```
In[145]:=
      CSPFigure3p2 = {
         {"Input", {0, 4}, u},
         {"Output", {16, 4}, y},
         {"Block", {{10, 0}, {5, 3}}, A, "", ElementSize → {2, 1.5}},
         {"Block", {{0, 4}, {4, 4}}, B, "", ElementSize → {2, 1.5}},
         {"Block", {{10, 4}, {14, 4}}, C, "", ElementSize → {2, 1.5}},
         {"Block", {{0, 8}, {15, 5}}, D, "", ElementSize → {2, 1.5}},
         {"Delay", {{6, 4}, {10, 4}}, 1, "Delay"},
         {"Adder", {{4, 4}, {5, 3}, {6, 4}, {5, 5}}, {1, 1, 2, 0}},
         {"Adder", {{14, 4}, {15, 3}, {16, 4}, {15, 5}}, {1, 0, 2, 1}},
         {"Line", {{0, 4}, {0, 8}}}, {"Line", {{10, 4}, {10, 0}}},
         {"Node", {0, 4}, ""}, {"Node", {6, 4}, ""},
         {"Node", {10, 4}, x, "", TextOffset → {0, -1}}
        };
       ShowSchematic[%, Frame → False];
```



`DiscreteSystemTransferFunction` computes the transfer function matrix of the system:

```
In[147]:=
      DiscreteSystemTransferFunction[CSPFigure3p2] // DiscreteSystemDisplayForm
```

Out[147]//DisplayForm=

$$\frac{-D + (-B\,C + A\,D)\,z^{-1}}{-1 + A\,z^{-1}}$$

## Unity Feedback System

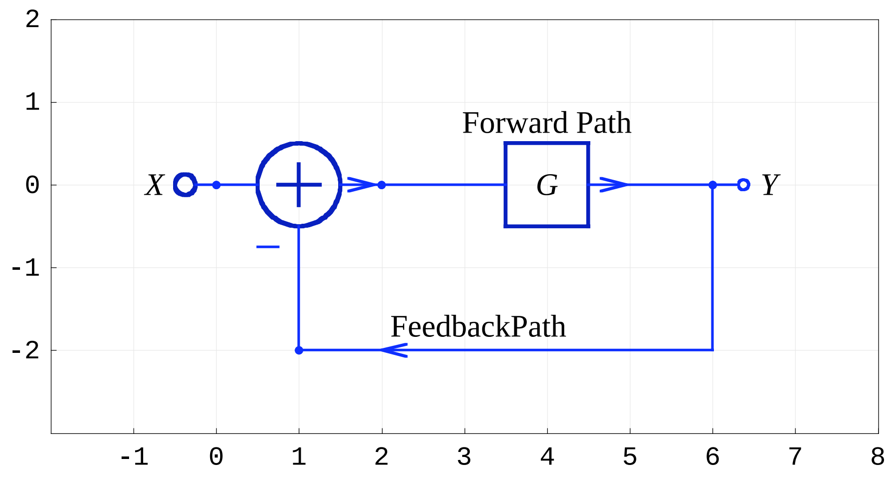
A *unity feedback system* is a feedback system in which the primary feedback is identically equal to the controlled output. Find the response of the system.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[148]:=
      Needs["SchematicSolver`"];
```

Here is the system schematic:

```
In[149]:=
      unityFeedbackSystem = {
         {"Input", {0, 0}, X},
         {"Output", {6, 0}, Y},
         {"Block", {{2, 0}, {6, 0}}, G, "Forward Path"},
         {"Arrow", {{2, -2}, {6, -2}}, "FeedbackPath", ShowArrowTail → False},
         {"Adder", {{0, 0}, {1, -2}, {2, 0}, {1, 1}}, {1, -1, 2, 0}},
         {"Line", {{6, 0}, {6, -2}, {1, -2}}}
        };
      ShowSchematic[%, PlotRange → {{-2, 8}, {-3, 2}}];
```



`DiscreteSystemEquations` sets up the equations of the system:

```
In[151]:=
      {unityFeedbackEquations, vars} = DiscreteSystemEquations[unityFeedbackSystem];
```

It is better typeset with

```
In[152]:=
      typoSubstYkn = {Y[{k_Integer, n_Integer}] :→ Yk,n};
```

```
In[153]:=
      unityFeedbackEquations /. typoSubstYkn // ColumnForm // TraditionalForm
```

*Out[153]//TraditionalForm=*

$Y_{0,0} == X$

$Y_{6,0} == G\,Y_{2,0}$

$Y_{2,0} == Y_{0,0} - Y_{6,0}$

`DiscreteSystemResponse` finds the response of the system:

```
In[154]:=
      {unityFeedbackResponse, vars} = DiscreteSystemResponse[unityFeedbackSystem];
```

```
In[155]:=
      unityFeedbackResponse /. typoSubstYkn // ColumnForm // TraditionalForm
```

*Out[155]//TraditionalForm=*

$Y_{6,0} \to \frac{G X}{G+1}$

$Y_{2,0} \to \frac{X}{G+1}$

$Y_{0,0} \to X$

`DiscreteSystemTransferFunction` computes the transfer function matrix of the system:

```
In[156]:=
      {tfMatrix, systemInp, systemOut} =
        DiscreteSystemTransferFunction[unityFeedbackSystem];
```

The transfer function of this single-input single-output system is the element of the transfer function matrix:

```
In[157]:=
      unityFeedbackTF = tfMatrix[[1, 1]];
      % // TraditionalForm
```

*Out[158]//TraditionalForm=*

$$\frac{G}{G+1}$$

# 6. Solving Large Systems

## 6.1. Combining Schematics

Some large schematics consist of replicas of the subschematics. It is not necessary to manually insert all elements. Instead, you can draw smaller parts that constitute the large system and combine them into a desired schematic.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

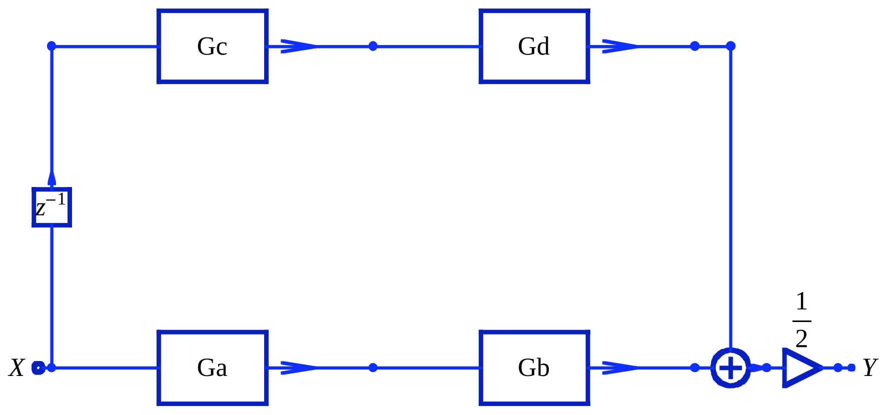
```
In[1]:=  Needs["SchematicSolver`"]
```

We shall adjust some options to obtain better appearance of the example schematics:

```
In[2]:=  SetOptions[InputNotebook[], ImageSize → {360, 450}, ImageMargins → {{0, 0}, {0, 0}}];

In[3]:=  SetOptions[ShowSchematic, PlotRange → {{-2, 24.5}, {-5, 14}}];
```

Consider a schematic that consists of four subschematics and some additional elements.

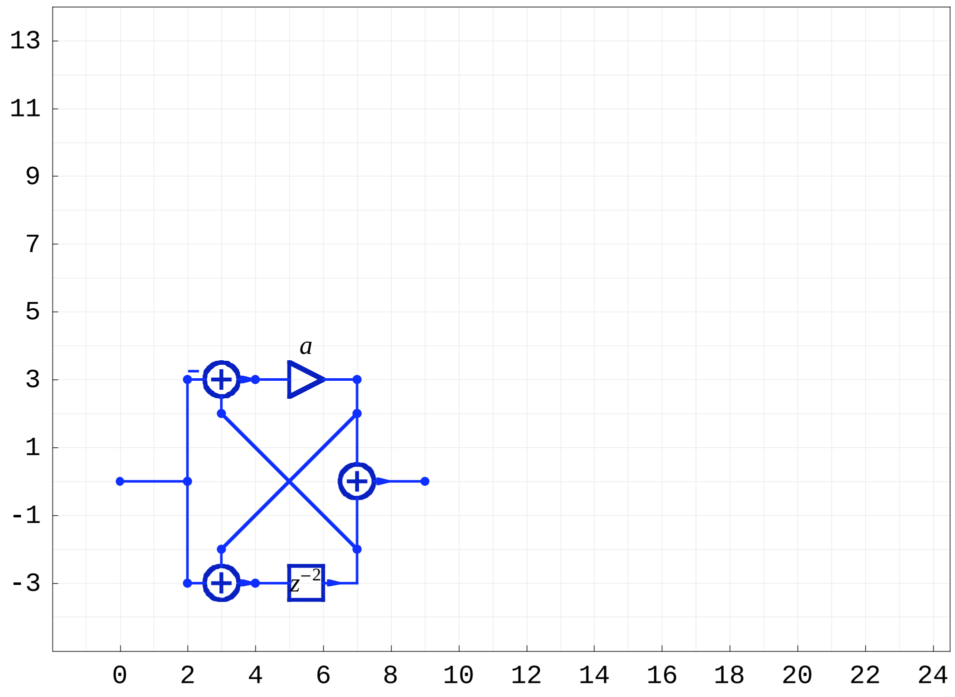
```
In[4]:=  combinedSystemFigure = { {"Input", {0, 0}, X},
            {"Block", {{0, 9}, {9, 9}}, Gc, "", ElementSize → {3, 2}},
            {"Block", {{9, 9}, {18, 9}}, Gd, "", ElementSize → {3, 2}},
            {"Block", {{0, 0}, {9, 0}}, Ga, "", ElementSize → {3, 2}},
            {"Block", {{9, 0}, {18, 0}}, Gb, "", ElementSize → {3, 2}},
            {"Delay", {{0, 0}, {0, 9}}, 1},
            {"Adder", {{18, 0}, {19, -1}, {20, 0}, {19, 9}}, {1, 0, 2, 1}},
            {"Output", {22, 0}, Y},
            {"Multiplier", {{20, 0}, {22, 0}}, 1 / 2},
            {"Line", {{18, 9}, {19, 9}}} };
         ShowSchematic[%, Frame → False, GridLines → None];
```



Large combined schematic can be generated from subschematics by using the *SchematicSolver*'s function `Translate-Schematic` and the *Mathematica* function `Join`.

First, we draw smaller part that constitute the schematic. Here is a simple subschematic:

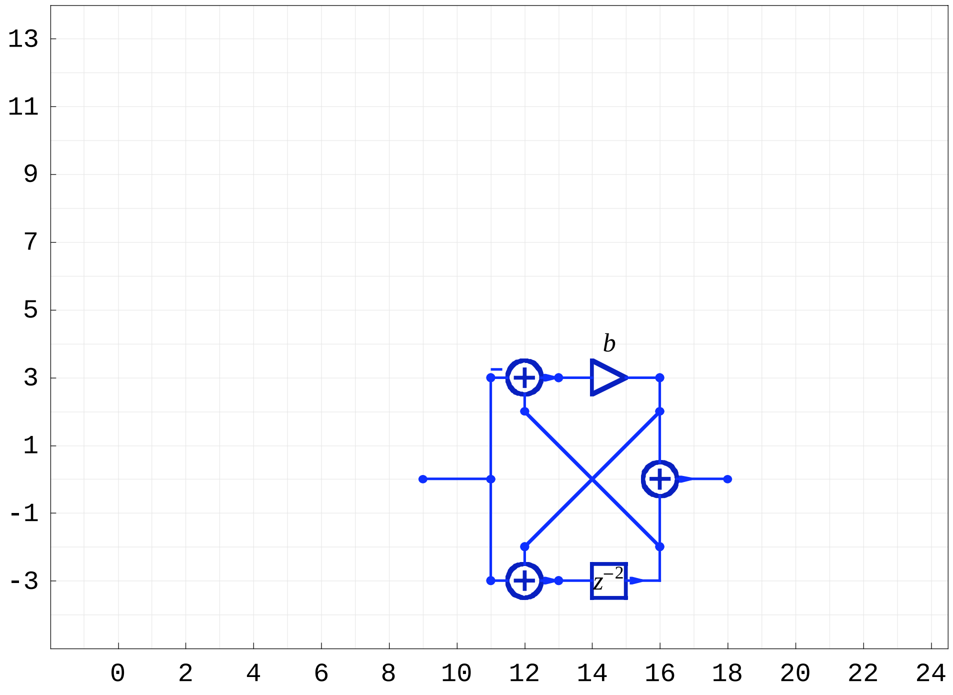
```
In[6]:=  mySubSchematic = {{"Adder", {{2, -3}, {3, -4}, {4, -3}, {3, -2}}, {1, 0, 2, 1}},
            {"Multiplier", {{4, 3}, {7, 3}}, a},
            {"Delay", {{4, -3}, {7, -2}}, 2},
            {"Adder", {{2, 3}, {3, 2}, {4, 3}, {3, 4}}, {-1, 1, 2, 0}},
            {"Adder", {{6, 0}, {7, -2}, {9, 0}, {7, 2}}, {0, 1, 2, 1}},
            {"Line", {{7, -2}, {3, 2}}}, {"Line", {{7, 2}, {3, -2}}},
            {"Line", {{7, 3}, {7, 2}}}, {"Line", {{0, 0}, {2, 0}}},
            {"Line", {{2, 0}, {2, -3}}}, {"Line", {{2, 0}, {2, 3}}}};
         % // ShowSchematic
```



The multiplier coefficient is *a*.

Suppose that we want to add a new subschematic in cascade. The new subschematic is generated by translating `mySubSchematic` by 9 steps to the right by using `TranslateSchematic`:

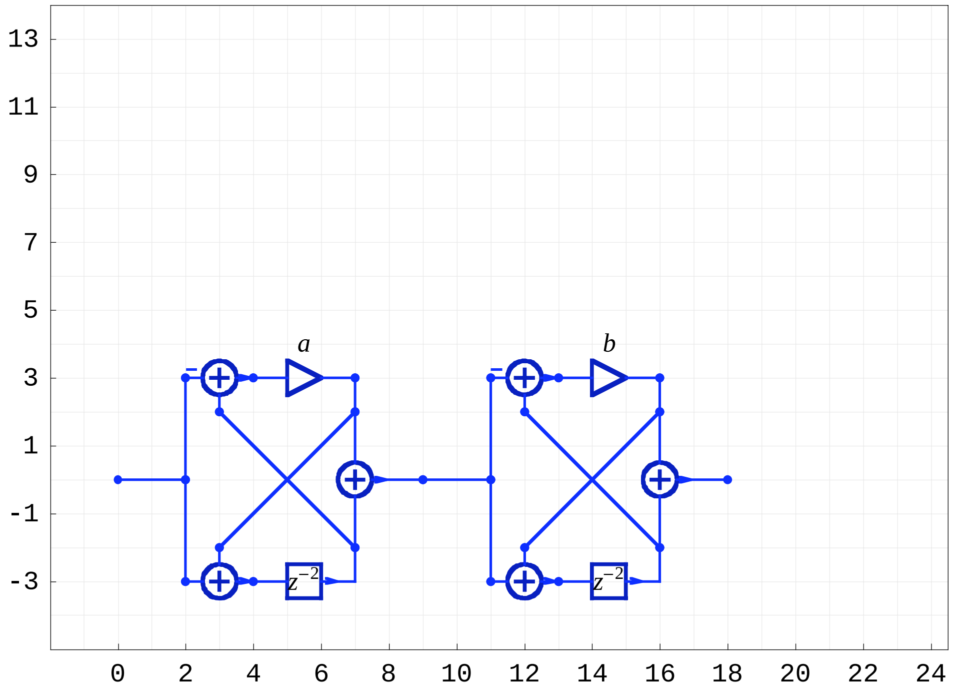
```
In[8]:=  myTranslatedSubSchematic = TranslateSchematic[mySubSchematic, {9, 0}] /. a → b;
         % // ShowSchematic
```



The multiplier coefficient is changed from *a* to *b*.

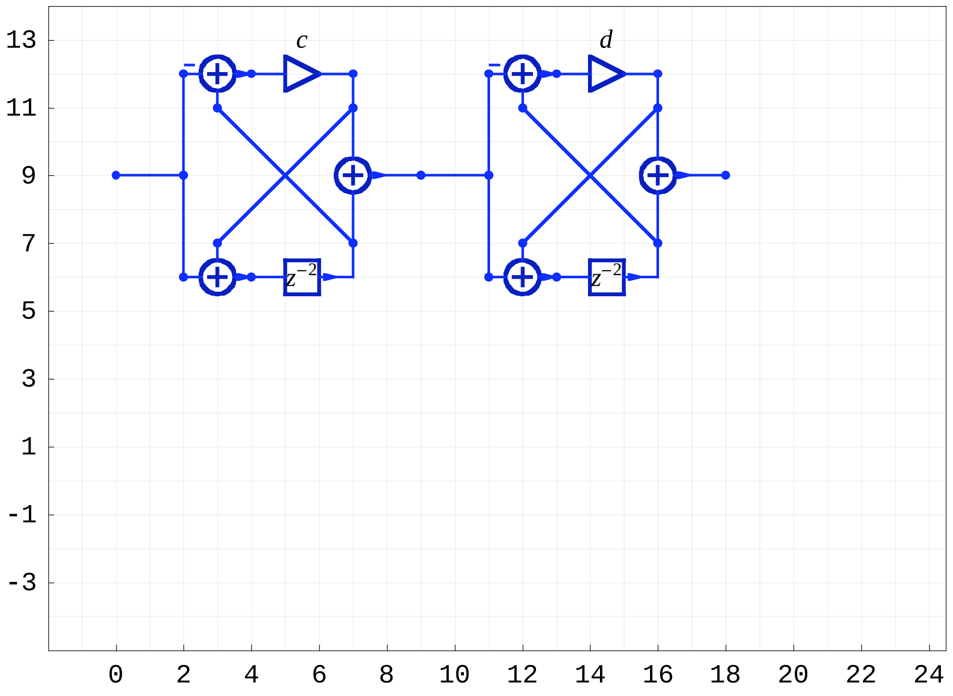
We form the cascade connection with the *Mathematica* built-in function `Join`:

```
In[10]:=  myCascadeConnection = mySubSchematic ~ Join ~ myTranslatedSubSchematic;
          % // ShowSchematic
```



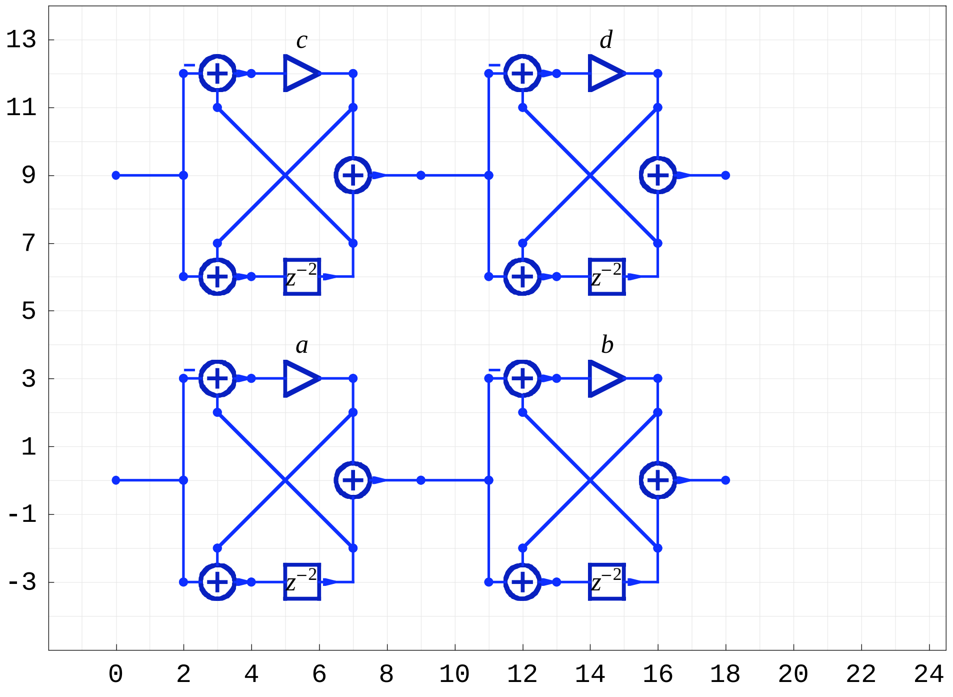
We want to make another cascade subschematic. The new subschematic is generated by translating the subschematic `myTranslatedSubSchematic` by 9 steps upwards:

```
In[12]:= myTranslatedCascadeConnection =
           TranslateSchematic[myCascadeConnection, {0, 9}] /. {a → c, b → d};
         % // ShowSchematic
```



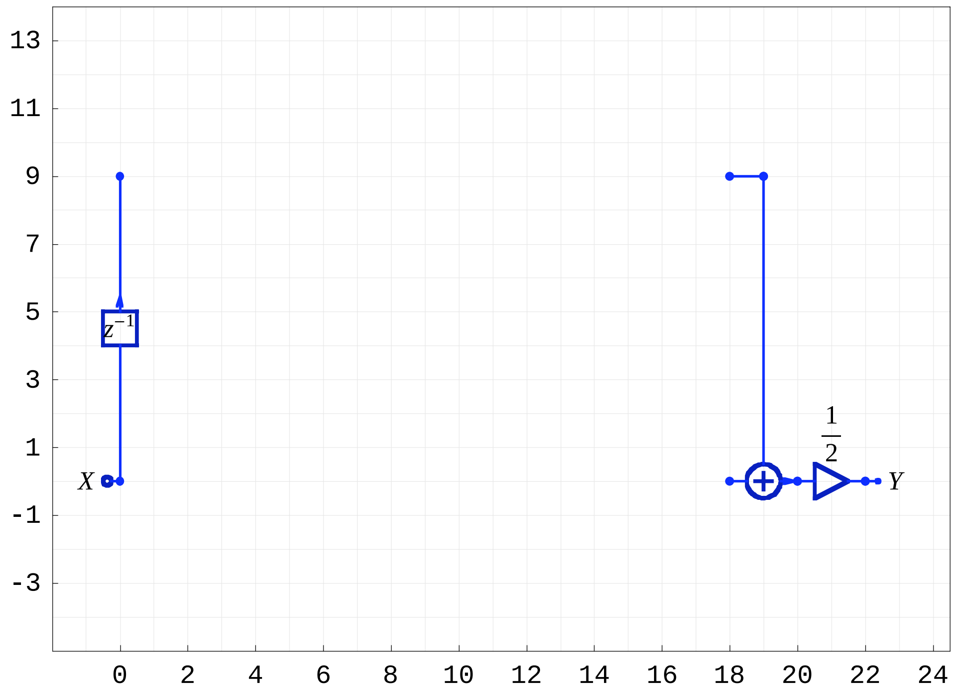
The schematic that consists of two parallel subschematics are generated by `Join`:

```
In[14]:= myParallelConnection = myCascadeConnection ~ Join ~ myTranslatedCascadeConnection;
         % // ShowSchematic
```



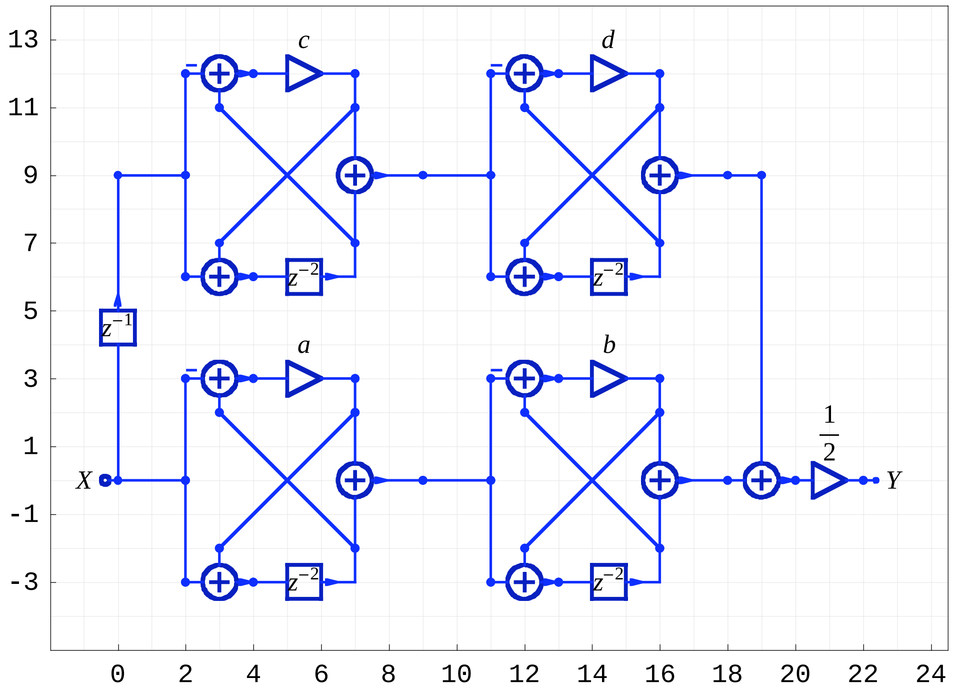
Next, we add some elements to complete the schematic. We can fill a new subschematic `myInOutSubSchematic` with additional elements by selecting the coordinates from the workspace of the previously drawn `myParallelConnection`.

```
In[16]:= myInOutSubSchematic = {{"Input", {0, 0}, X},
            {"Delay", {{0, 0}, {0, 9}}, 1},
            {"Adder", {{18, 0}, {19, -1}, {20, 0}, {19, 9}}, {1, 0, 2, 1}},
            {"Output", {22, 0}, Y},
            {"Multiplier", {{20, 0}, {22, 0}}, 1/2},
            {"Line", {{18, 9}, {19, 9}}}};
         % // ShowSchematic
```



Finally, we build the system as shown below:

```
In[18]:=  mySystem = myInOutSubSchematic ~ Join ~ myParallelConnection;
          % // ShowSchematic
```



## 6.2. Transfer Function

Transfer function of the resulting system is computed by the *SchematicSolver*'s function `DiscreteSystemTransfer-Function`:

```
In[20]:=  {myH, systemInp, systemOut} = DiscreteSystemTransferFunction[mySystem];
          myValues = {a → -0.1091, b → -0.6335, c → -0.3616, d → -0.8774};
          myHspecific = myH /. myValues;
          % // Simplify // TraditionalForm
```

Out[23]//TraditionalForm=

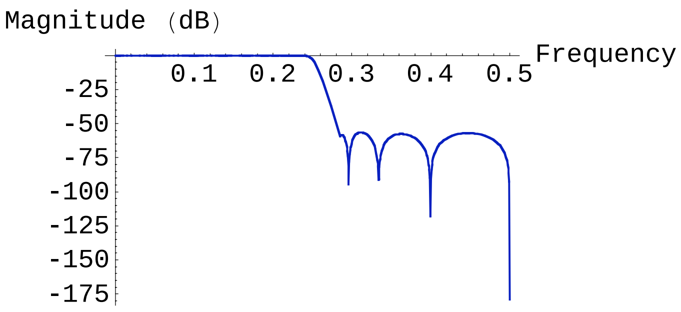
$$\left( \frac{0.0345574\,(z^2+0.399649\,z+1.)\,(z^2+0.569026\,z+1.)\,(z^2+1.00678\,z+1.)\,(z^2+1.61499\,z+1.)\,(z^2+2.\,z+1.)}{z\,(z+1.)\,(z^2+0.1091)\,(z^2+0.3616)\,(z^2+0.6335)\,(z^2+0.8774)} \right)$$

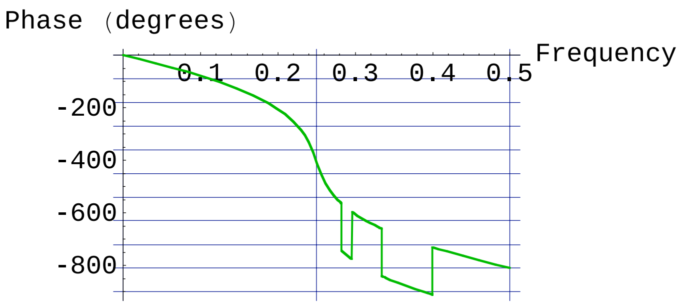
## 6.3. Frequency Response

Frequency response of the system is obtained by the *SchematicSolver*'s function `DiscreteSystemFrequencyResponse`:

```
In[24]:=  SetOptions[InputNotebook[], ImageSize → {260, 190}];
```

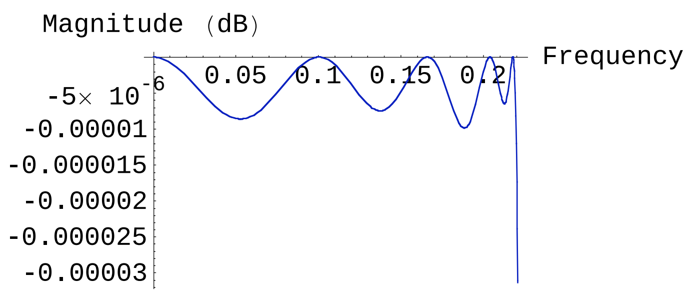
```
In[25]:=  DiscreteSystemFrequencyResponse[myHspecific];
```

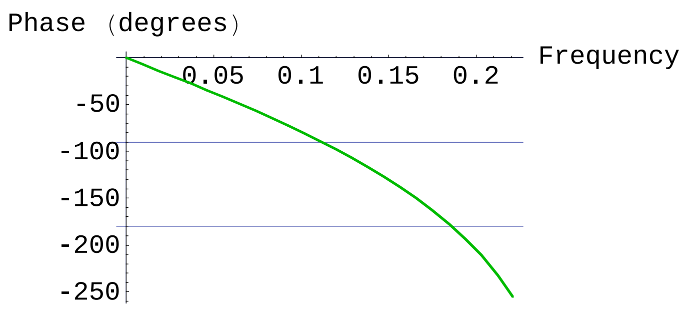




Let us zoom in to get a better insight into the magnitude response:

```
In[26]:=  DiscreteSystemFrequencyResponse[myHspecific, {0, 0.221}];
```





Fully symbolic expression of the transfer function is

```
In[27]:=  DiscreteSystemDisplayForm[myH]
```

*Out[27]//DisplayForm=*

$$\Big(a\,b + c\,d\,z^{-1} + (-a - b - a\,b\,c - a\,b\,d)\,z^{-2} + (-c - d - a\,c\,d - b\,c\,d)\,z^{-3} + (1 + a\,c + b\,c + a\,d + b\,d + a\,b\,c\,d)\,z^{-4} + (1 + a\,c + b\,c + a\,d + b\,d + a\,b\,c\,d)\,z^{-5} + (-c - d - a\,c\,d - b\,c\,d)\,z^{-6} + (-a - b - a\,b\,c - a\,b\,d)\,z^{-7} + c\,d\,z^{-8} + a\,b\,z^{-9}\Big) \Big/ \Big(2 + 2\,(-a - b - c - d)\,z^{-2} + 2\,(a\,b + a\,c + b\,c + a\,d + b\,d + c\,d)\,z^{-4} + 2\,(-a\,b\,c - a\,b\,d - a\,c\,d - b\,c\,d)\,z^{-6} + 2\,a\,b\,c\,d\,z^{-8}\Big)$$

where *a*, *b*, *c*, and *d* are arbitrary system parameters.

# 7. Implementation of Discrete Systems

## 7.1. Introduction

*SchematicSolver* can be used for generating software implementation of discrete systems.

*Software implementation* is a sequence of statements that are executed on a general-purpose computer or on a dedicated hardware.

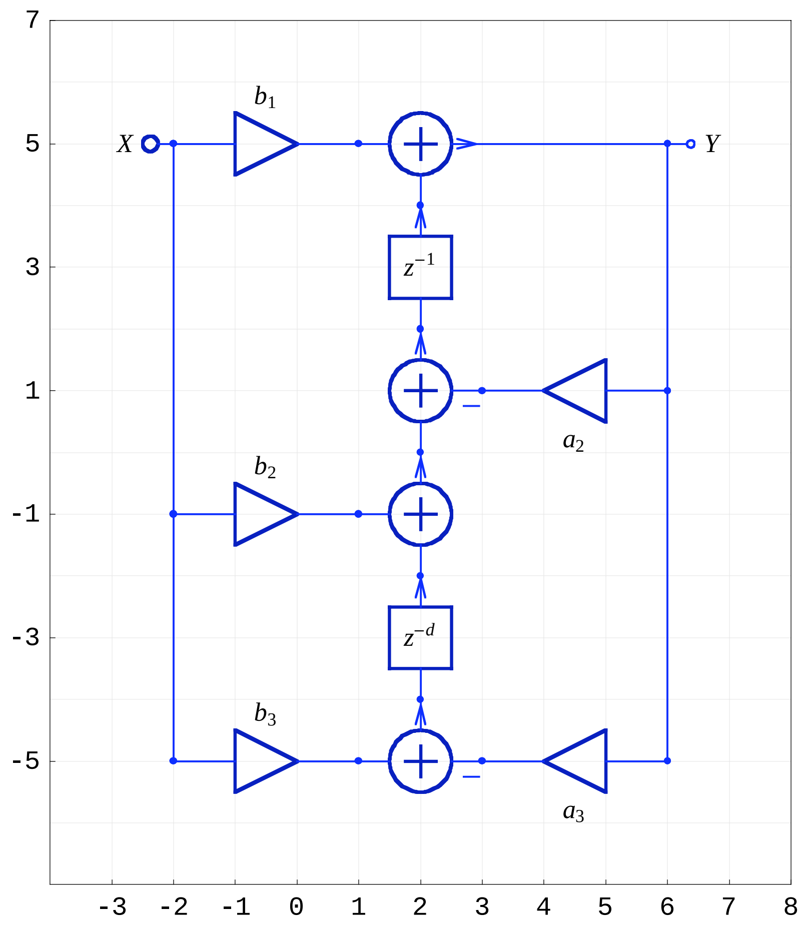
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[1]:= Needs["SchematicSolver`"];
```

## 7.2. Schematic of Discrete System

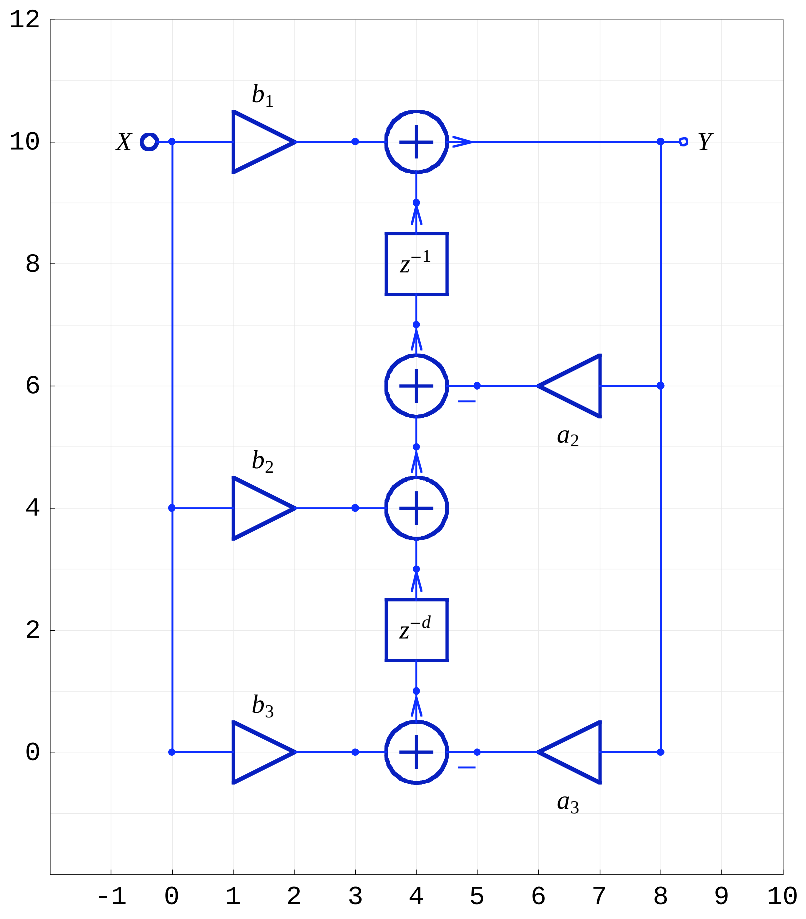
Consider a system

```
In[2]:= discreteSchematic = {
           {"Input", {-2, 5}, X}, {"Output", {6, 5}, Y},
           {"Multiplier", {{-2, -5}, {1, -5}}, b3},
           {"Multiplier", {{-2, -1}, {1, -1}}, b2},
           {"Multiplier", {{-2, 5}, {1, 5}}, b1},
           {"Multiplier", {{6, -5}, {3, -5}}, a3},
           {"Multiplier", {{6, 1}, {3, 1}}, a2},
           {"Adder", {{1, -5}, {2, -6}, {3, -5}, {2, -4}}, {1, 0, -1, 2}},
           {"Adder", {{1, -1}, {2, -2}, {3, -1}, {2, 0}}, {1, 1, 0, 2}},
           {"Adder", {{1, 1}, {2, 0}, {3, 1}, {2, 2}}, {0, 1, -1, 2}},
           {"Adder", {{1, 5}, {2, 4}, {6, 5}, {2, 6}}, {1, 1, 2, 0}},
           {"Delay", {{2, 2}, {2, 4}}, 1}, {"Delay", {{2, -4}, {2, -2}}, d},
           {"Line", {{6, 5}, {6, 1}}}, {"Line", {{-2, 5}, {-2, -1}}},
           {"Line", {{6, 1}, {6, -5}}}, {"Line", {{-2, -1}, {-2, -5}}}};
         ShowSchematic[% /. {a2 → a₂, a3 → a₃, b1 → b₁, b2 → b₂, b3 → b₃},
           PlotRange → {{-4, 8}, {-7, 7}}, Frame → True];
```



It is more convenient to work with positive coordinates, therefore we shall translate the schematic with `AdjustSchematicCoordinates`:

```
In[4]:= discreteSystem = AdjustSchematicCoordinates[discreteSchematic];
        ShowSchematic[% /. {a2 → a₂, a3 → a₃, b1 → b₁, b2 → b₂, b3 → b₃},
          PlotRange → {{-2, 10}, {-2, 12}}, Frame → True];
```



The subsequent sections explain a procedure for generating software implementation of the system described by the schematic specification `discreteSystem`.

## 7.3. Check Schematic Specification

We use `DiscreteSystemImplementationEquations` to test whether the system specification meets the necessary implementation conditions (e.g., unit delays are assumed, the Block element is not supported):

```
In[6]:= DiscreteSystemImplementationEquations[discreteSystem /. d → 1]

Out[6]= {{Y[{0, 10}]}, {Y[{4, 9}], Y[{4, 3}]}, {a2, a3, b1, b2, b3},
         {Y[{0, 10}] == X, Y[{4, 9}] == previousSample[Y[{4, 7}]],
          Y[{4, 3}] == previousSample[Y[{4, 1}]], Y[{3, 0}] == b3 Y[{0, 10}],
          Y[{3, 4}] == b2 Y[{0, 10}], Y[{3, 10}] == b1 Y[{0, 10}],
          Y[{4, 5}] == Y[{3, 4}] + Y[{4, 3}], Y[{8, 10}] == Y[{3, 10}] + Y[{4, 9}],
          Y[{5, 0}] == a3 Y[{8, 10}], Y[{5, 6}] == a2 Y[{8, 10}],
          Y[{4, 1}] == Y[{3, 0}] - Y[{5, 0}], Y[{4, 7}] == Y[{4, 5}] - Y[{5, 6}]},
         {Y[{8, 10}]}, {Y[{4, 7}], Y[{4, 1}]}}
```

Here is an example of a system with one non-unit delay element. Consequently, `DiscreteSystemImplementa-tionEquations` reports an error:

```
In[7]:= DiscreteSystemImplementationEquations[discreteSystem /. d → 2]

        NonlinearDiscreteElementTopology::invdelay :
         {"Delay", {{4, 1}, {4, 3}}, 2, ""} is not a well-formed element
           specification.  The Delay specification is a list of the form
           {"Delay", {{x1,y1},{x2,y2}}, 1, elementLabel}.  The element value
           is 1 because only the unit delay is considered for implementation.

           Unconnected nodes exist: {Y[{4, 3}]}

        DiscreteSystemImplementationEquations::intercon :
         Unexpected interconnection of schematic
           elements. The system cannot be implemented.

Out[7]= {{}, {}, {}, {}, {}, {}}
```

## 7.4. Generate Implementation

*Software implementation* is a sequence of statements that are executed on a general-purpose computer or on a dedicated hardware.

The system summary, generated by `DiscreteSystemImplementationSummary`, points out the system input, initial state, parameter set, output, and final state.

```
In[8]:= DiscreteSystemImplementationSummary[discreteSystem /. d → 1, Verbose → True]

           Input: {Y[{0, 10}]}

           Initial state: {Y[{4, 9}], Y[{4, 3}]}

           Parameter: {a2, a3, b1, b2, b3}

           Equations: Y[{0, 10}] == X
                      Y[{4, 9}] == previousSample[Y[{4, 7}]]
                      Y[{4, 3}] == previousSample[Y[{4, 1}]]
                      Y[{3, 0}] == b3 Y[{0, 10}]
                      Y[{3, 4}] == b2 Y[{0, 10}]
                      Y[{3, 10}] == b1 Y[{0, 10}]
                      Y[{4, 5}] == Y[{3, 4}] + Y[{4, 3}]
                      Y[{8, 10}] == Y[{3, 10}] + Y[{4, 9}]
                      Y[{5, 0}] == a3 Y[{8, 10}]
                      Y[{5, 6}] == a2 Y[{8, 10}]
                      Y[{4, 1}] == Y[{3, 0}] - Y[{5, 0}]
                      Y[{4, 7}] == Y[{4, 5}] - Y[{5, 6}]

           Output: {Y[{8, 10}]}

           Final state: {Y[{4, 7}], Y[{4, 1}]}
```

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system and returns a string that is the *Mathematica* code of that function.

```
In[9]:= codeString =
          DiscreteSystemImplementation[discreteSystem /. d → 1, "implementationProcedure"];

           Implementation procedure name: implementationProcedure

           Implementation procedure usage:

            {{Y8p10}, {Y4p7, Y4p1}} = implementationProcedure[
             {Y0p10},{Y4p9, Y4p3},{a2, a3, b1, b2, b3}] is the template for
             calling the procedure.  The general template is {outputSamples,
             finalConditions} = procedureName[inputSamples, initialConditions,
             systemParameters]. See also: DiscreteSystemImplementationProcessing
```

The name of the implementation function is arbirtrary and it is given as the second argument to `DiscreteSystemImplementation`. In the above example, the name of the implementation function is `implementationProcedure` and it should be enclosed within double quotation marks.

Here is the string that contains the code of the implementation function:

```
In[10]:= codeString

Out[10]= implementationProcedure::usage = " {{Y8p10}, {Y4p7, Y4p1}} =
           implementationProcedure[{Y0p10},{Y4p9, Y4p3},{a2, a3, b1, b2, b3}] is the
           template for calling the procedure.  The general template is {outputSamples,
           finalConditions} = procedureName[inputSamples, initialConditions,
           systemParameters]. See also: DiscreteSystemImplementationProcessing";
           implementationProcedure[] := {1, 2, 5, 12, 1, 2}; implementationProcedure[
           dataSamples_List, initialConditions_List, systemParameters_List] :=
           Module[ {Y0p10, Y4p9, Y4p3, Y3p0, Y3p4, Y3p10, Y4p5, Y8p10, Y5p0,
           Y5p6, Y4p1, Y4p7, a2, a3, b1, b2, b3}, {a2, a3, b1, b2, b3} =
           systemParameters; {Y0p10} = dataSamples; {Y4p9, Y4p3} = initialConditions;
           Y3p0 = b3*Y0p10; Y3p4 = b2*Y0p10; Y3p10 = b1*Y0p10; Y4p5 = Y3p4 +
           Y4p3; Y8p10 = Y3p10 + Y4p9; Y5p0 = a3*Y8p10; Y5p6 = a2*Y8p10;
           Y4p1 = Y3p0 - Y5p0; Y4p7 = Y4p5 - Y5p6; {{Y8p10}, {Y4p7, Y4p1}} ];
```

You can use **??** to get full information about the implementation procedure:

```
In[11]:= ?? implementationProcedure

            {{Y8p10}, {Y4p7, Y4p1}} = implementationProcedure[
             {Y0p10},{Y4p9, Y4p3},{a2, a3, b1, b2, b3}] is the template for
             calling the procedure.  The general template is {outputSamples,
             finalConditions} = procedureName[inputSamples, initialConditions,
             systemParameters]. See also: DiscreteSystemImplementationProcessing

           implementationProcedure[] := {1, 2, 5, 12, 1, 2}

           implementationProcedure[dataSamples_List,
             initialConditions_List, systemParameters_List] :=
            Module[{Y0p10, Y4p9, Y4p3, Y3p0, Y3p4, Y3p10, Y4p5, Y8p10, Y5p0, Y5p6,
              Y4p1, Y4p7, a2, a3, b1, b2, b3}, {a2, a3, b1, b2, b3} = systemParameters;
             {Y0p10} = dataSamples; {Y4p9, Y4p3} = initialConditions;
             Y3p0 = b3 Y0p10; Y3p4 = b2 Y0p10; Y3p10 = b1 Y0p10; Y4p5 = Y3p4 + Y4p3;
             Y8p10 = Y3p10 + Y4p9; Y5p0 = a3 Y8p10; Y5p6 = a2 Y8p10;
             Y4p1 = Y3p0 - Y5p0; Y4p7 = Y4p5 - Y5p6; {{Y8p10}, {Y4p7, Y4p1}}]
```

`DiscreteSystemImplementationEquations` is used to extract the system input, initial state, parameter set, implementation equations, output, and final state:

```
In[12]:= eqns = DiscreteSystemImplementationEquations[discreteSystem /. d → 1];

In[13]:= systemInput = eqns[[1]]

Out[13]= {Y[{0, 10}]}

In[14]:= initialConditions = eqns[[2]]

Out[14]= {Y[{4, 9}], Y[{4, 3}]}

In[15]:= systemParameters = eqns[[3]]

Out[15]= {a2, a3, b1, b2, b3}
```

```
In[16]:= implementationEquations = eqns[[4]];
         % // ColumnForm

Out[17]= Y[{0, 10}] == X
         Y[{4, 9}] == previousSample[Y[{4, 7}]]
         Y[{4, 3}] == previousSample[Y[{4, 1}]]
         Y[{3, 0}] == b3 Y[{0, 10}]
         Y[{3, 4}] == b2 Y[{0, 10}]
         Y[{3, 10}] == b1 Y[{0, 10}]
         Y[{4, 5}] == Y[{3, 4}] + Y[{4, 3}]
         Y[{8, 10}] == Y[{3, 10}] + Y[{4, 9}]
         Y[{5, 0}] == a3 Y[{8, 10}]
         Y[{5, 6}] == a2 Y[{8, 10}]
         Y[{4, 1}] == Y[{3, 0}] - Y[{5, 0}]
         Y[{4, 7}] == Y[{4, 5}] - Y[{5, 6}]

In[18]:= systemOutput = eqns[[5]]

Out[18]= {Y[{8, 10}]}

In[19]:= finalConditions = eqns[[6]]

Out[19]= {Y[{4, 7}], Y[{4, 1}]}
```

## 7.5. Processing Sample by Sample

The function `implementationProcedure` processes one sample at a time. For example, here we process 3 samples:

```
In[20]:= inputSample1 = {1};
         initialConditions1 = 0 * initialConditions;

In[22]:= {outputSample1, finalConditions1} =
           implementationProcedure[inputSample1, initialConditions1, systemParameters];
         % // ColumnForm

Out[23]= {b1}
         {-a2 b1 + b2, -a3 b1 + b3}

In[24]:= inputSample2 = {0};
         initialConditions2 = finalConditions1;

In[26]:= {outputSample2, finalConditions2} =
           implementationProcedure[inputSample2, initialConditions2, systemParameters];
         % // ColumnForm

Out[27]= {-a2 b1 + b2}
         {-a3 b1 - a2 (-a2 b1 + b2) + b3, -a3 (-a2 b1 + b2)}

In[28]:= inputSample3 = {0};
         initialConditions3 = finalConditions2;

In[30]:= {outputSample3, finalConditions3} =
           implementationProcedure[inputSample3, initialConditions3, systemParameters];
         % // ColumnForm

Out[31]= {-a3 b1 - a2 (-a2 b1 + b2) + b3}
         {-a3 (-a2 b1 + b2) - a2 (-a3 b1 - a2 (-a2 b1 + b2) + b3), -a3 (-a3 b1 - a2 (-a2 b1 + b2) + b3)}
```

## 7.6. Processing Sequences

Let us process a unit impulse sequence

```
In[32]:=  exampleInputSequence = UnitImpulseSequence[10]

Out[32]=  {{1}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}}
```

`DiscreteSystemImplementationProcessing` processes `exampleInputSequence` for created `implementationProcedure`.

```
In[33]:=  {outputSequence, finalConditions} =
            DiscreteSystemImplementationProcessing[exampleInputSequence,
             initialConditions1, systemParameters, implementationProcedure];
```

Here are the first 3 output samples:

```
In[34]:=  outputSequence[[1]]
          outputSequence[[2]]
          outputSequence[[3]]

Out[34]=  {b1}

Out[35]=  {-a2 b1 + b2}

Out[36]=  {-a3 b1 - a2 (-a2 b1 + b2) + b3}
```

The same result has been already obtained by processing sample by sample.

## 7.7. Simulation of Discrete System

`DiscreteSystemSimulation` finds the output sequence of a discrete system, given by a schematic, assuming zero initial conditions.

```
In[37]:=  outSeq = DiscreteSystemSimulation[discreteSystem /. d → 1];
```

`DiscreteSystemSimulation`, by default, uses unit impulse sequence as input.

Here are the first 3 output samples:

```
In[38]:=  outSeq[[1]]
          outSeq[[2]]
          outSeq[[3]]

Out[38]=  {b1}

Out[39]=  {-a2 b1 + b2}

Out[40]=  {-a3 b1 - a2 (-a2 b1 + b2) + b3}
```

The same result has been already obtained by processing sample by sample, or by `DiscreteSystemImplementationProcessing`.

The second argument to `DiscreteSystemSimulation` specifies the input sequence to the system. Here we process a ramp input sequence:

```
In[41]:= DiscreteSystemSimulation[discreteSystem /. d → 1, UnitRampSequence[4]];
         % // TraditionalForm
```

Out[42]//TraditionalForm=

$$\begin{pmatrix} 0 \\ b1 \\ -a2\,b1 + 2\,b1 + b2 \\ -a3\,b1 + 3\,b1 + 2\,b2 - a2\,(-a2\,b1 + 2\,b1 + b2) + b3 \end{pmatrix}$$

Symbolic system simulation is the *SchematicSolver*'s unique feature not available in other simulation software. The above example demonstrates that `DiscreteSystemSimulation` returns the output sequence with symbolic sample values.

## 7.8. Test Implementation with `DiscreteSystemProcessingSISO`

*SchematicSolver*'s function `DiscreteSystemProcessingSISO` processes samples with a linear single-input single-output system for a given transfer function.

```
In[43]:= tf = DiscreteSystemTransferFunction[discreteSystem /. d → 1];
         % // DiscreteSystemDisplayForm
```

Out[44]//DisplayForm=

$$\frac{b1 + b2\,z^{-1} + b3\,z^{-2}}{1 + a2\,z^{-1} + a3\,z^{-2}}$$

`DiscreteSystemProcessingSISO` requires a list of data samples, so we use `SequenceToList` to convert `exampleInputSequence` to a list.

```
In[45]:= exampleInputList = exampleInputSequence // SequenceToList

Out[45]= {1, 0, 0, 0, 0, 0, 0, 0, 0, 0}

In[46]:= {outputSISO, finalSISO} = DiscreteSystemProcessingSISO[exampleInputList, tf];
```

The functions `DiscreteSystemProcessingSISO` and `DiscreteSystemImplementationProcessing` with `implementationProcedure` should yield the same result:

```
In[47]:= SameQ[outputSequence, ListToSequence[outputSISO]]
         SameQ[finalConditions, finalSISO]

Out[47]= True

Out[48]= True
```

## 7.9. Implementation Using Palettes

*SchematicSolver*'s function for implementing systems can be called from the palettes.

First, you might assign numeric values to the parameters:

```
In[49]:= systemParameters
         parameterValues = {-0.5, 0.9, 1 , 0.4, 1}
         parameterSubstitution = systemParameters → parameterValues // Thread

Out[49]= {a2, a3, b1, b2, b3}

Out[50]= {-0.5, 0.9, 1, 0.4, 1}

Out[51]= {a2 → -0.5, a3 → 0.9, b1 → 1, b2 → 0.4, b3 → 1}
```

Next, you assign the default name to the schematic that represents the system:

```
In[52]:= mySchematic = discreteSystem /. d → 1 /. parameterSubstitution;
```

Open the DiscreteElements palette, next click the button Implement.

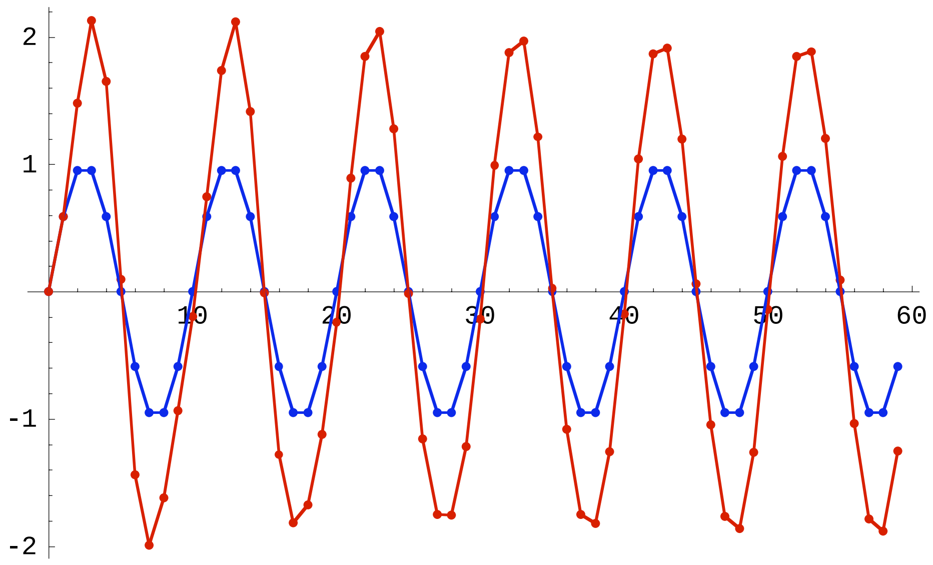
```
In[53]:= procedureName = implementationProcedure;
         DiscreteSystemImplementation[mySchematic, ToString[procedureName]];
         DiscreteSystemImplementationSummary[mySchematic, Verbose -> True]
         Print["--- EXAMPLE: Input Sequence, Initial Conditions, System Parameters"];
         eqns = DiscreteSystemImplementationEquations[mySchematic];
         numberOfInputs = Length[eqns[[1]]];
         inputSequence =
          MultiplexSequence @@ Table[UnitImpulseSequence[], {numberOfInputs}]
         initialConditions = 0 * eqns[[2]]
         systemParameters = eqns[[3]]
         Print["--- PROCESSING: Output Sequence, Final Conditions"];
         {outputSequence, finalConditions} = DiscreteSystemImplementationProcessing[
            inputSequence, initialConditions, systemParameters, procedureName];
         outputSequence
          finalConditions
         Print["--- End of SchematicSolver Implementation ---"];

           Implementation procedure name: implementationProcedure

           Implementation procedure usage:

            {{Y8p10}, {Y4p7, Y4p1}} = implementationProcedure[
             {Y0p10},{Y4p9, Y4p3},{}] is the template for calling
             the procedure.  The general template is {outputSamples,
             finalConditions} = procedureName[inputSamples, initialConditions,
             systemParameters]. See also: DiscreteSystemImplementationProcessing

           Input: {Y[{0, 10}]}

           Initial state: {Y[{4, 9}], Y[{4, 3}]}

           Parameter: {}

           Equations: Y[{0, 10}] == X
                      Y[{4, 9}] == previousSample[Y[{4, 7}]]
                      Y[{4, 3}] == previousSample[Y[{4, 1}]]
                      Y[{3, 0}] == Y[{0, 10}]
                      Y[{3, 4}] == 0.4 Y[{0, 10}]
                      Y[{3, 10}] == Y[{0, 10}]
                      Y[{4, 5}] == Y[{3, 4}] + Y[{4, 3}]
                      Y[{8, 10}] == Y[{3, 10}] + Y[{4, 9}]
                      Y[{5, 0}] == 0.9 Y[{8, 10}]
                      Y[{5, 6}] == -0.5 Y[{8, 10}]
                      Y[{4, 1}] == Y[{3, 0}] - Y[{5, 0}]
                      Y[{4, 7}] == Y[{4, 5}] - Y[{5, 6}]

           Output: {Y[{8, 10}]}

           Final state: {Y[{4, 7}], Y[{4, 1}]}

           --- EXAMPLE: Input Sequence, Initial Conditions, System Parameters

Out[59]= {{1}, {0}, {0}, {0}, {0}, {0}, {0}, {0}}

Out[60]= {0, 0}

Out[61]= {}

           --- PROCESSING: Output Sequence, Final Conditions

Out[64]= {{1}, {0.9}, {0.55}, {-0.535}, {-0.7625}, {0.10025}, {0.736375}, {0.277963}}

Out[65]= {-0.523756, -0.250166}

           --- End of SchematicSolver Implementation ---
```

You can process another input sequence with the same implementation function.

```
In[67]:= inputSequence = UnitSineSequence[60, 0.1];
         {outputSequence, finalConditions} = DiscreteSystemImplementationProcessing[
            inputSequence, initialConditions, systemParameters, procedureName];
```

Here is the plot of the input (blue) and output (red) sequences:

```
In[69]:= MultiplexSequence[inputSequence, outputSequence];
         SequencePlot[%, StemPlot → False, PlotJoined → True];
```



## 7.10. Simulation Using Palettes

Use `DiscreteSystemSimulation` to generate the system impulse response quickly.

Click the button **Simulation** on the DiscreteElements palette.

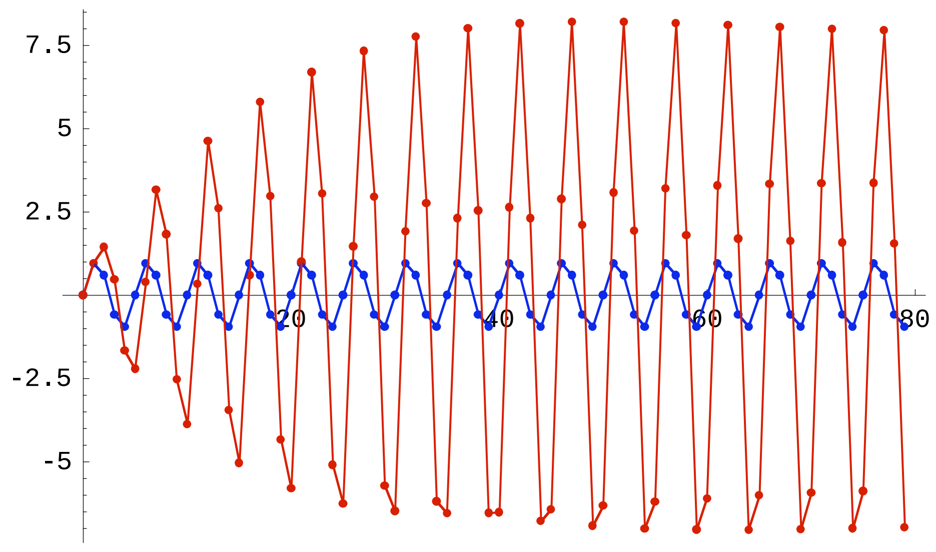
```
In[71]:= DiscreteSystemSimulation[mySchematic]
         Print["--- End of SchematicSolver Simulation ---"];

Out[71]= {{1}, {0.9}, {0.55}, {-0.535}, {-0.7625}, {0.10025}, {0.736375}, {0.277963}}

           --- End of SchematicSolver Simulation ---
```

Find the response to another input sequence as follows:

```
In[73]:= inputSequence = UnitSineSequence[80, 0.2];
         outputSequence = DiscreteSystemSimulation[mySchematic, inputSequence];
```

Here is the plot of the input (blue) and output (red) sequences:

```
In[75]:=  MultiplexSequence[inputSequence, outputSequence];
          SequencePlot[%, StemPlot → False, PlotJoined → True];
```



## 7.11. Benefits of *SchematicSolver*

*SchematicSolver* generates software implementation of a system directly from the system schematic.

*SchematicSolver* symbolically processes data samples keeping the system parameters as symbols.

In addition, *SchematicSolver* can process samples in a traditional numerical way.

# 8. Nonlinear Discrete System Implementation

## 8.1. Introduction

*SchematicSolver* can be used for generating software implementation of nonlinear discrete systems.

*Software implementation* is a sequence of statements that are executed on a general-purpose computer or on a dedicated hardware.

The Function element and the Modulator element are *SchematicSolver*'s nonlinear elements.

## 8.2. Nonlinear Algebraic Function Element

### Generic Function-Element Value

Function-element value can be any algebraic function of one argument. The value can be a symbol, say *F*, without a definition.

First, make sure that *F* has not been used before:

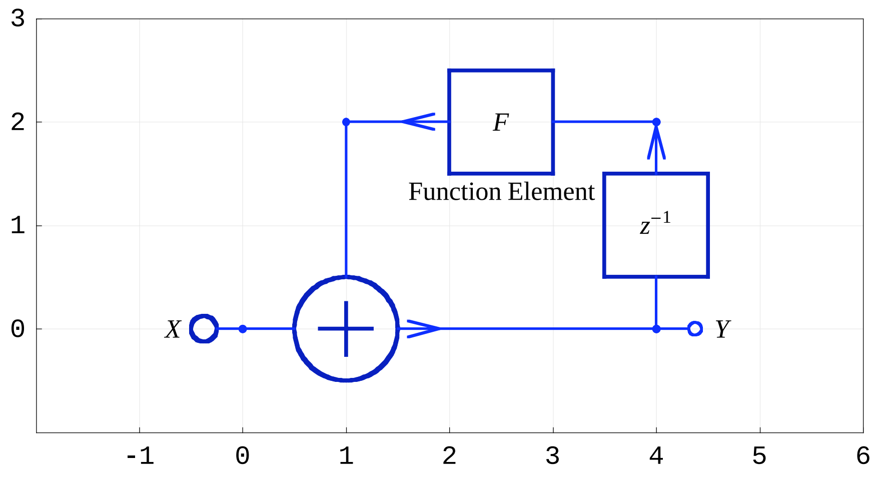
```
In[1]:=  Clear[F]
```

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[2]:=  Needs["SchematicSolver`"];
```

Here is an example system:

```
In[3]:=  discreteSystemGenericF = {{"Input", {0, 0}, X},
            {"Output", {4, 0}, Y},
            {"Adder", {{0, 0}, {1, -1}, {4, 0}, {1, 2}}, {1, 0, 2, 1}},
            {"Delay", {{4, 0}, {4, 2}}},
            {"Function", {{4, 2}, {1, 2}}, F, "Function Element"}};
         ShowSchematic[%, PlotRange → {{-2, 6}, {-1, 3}}];
```



`DiscreteSystemImplementationSummary` points out the system input, initial state, parameter set, output, and final state:

```
In[5]:=  DiscreteSystemImplementationSummary[discreteSystemGenericF]

            Input: {Y[{0, 0}]}

            Initial state: {Y[{4, 2}]}

            Parameter: {F}

            Output: {Y[{4, 0}]}

            Final state: {Y[{4, 0}]}
```

The symbol *F* appears as a parameter of the system `discreteSystemGenericF`.

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system (the default name of this function is `implementationProcedure`):

```
In[6]:=  DiscreteSystemImplementation[discreteSystemGenericF];

            Implementation procedure name: implementationProcedure

            Implementation procedure usage:

             {{Y4p0}, {Y4p0}} = implementationProcedure[{Y0p0},{Y4p2},{F}] is the template
              for calling the procedure.  The general template is {outputSamples,
              finalConditions} = procedureName[inputSamples, initialConditions,
              systemParameters]. See also: DiscreteSystemImplementationProcessing
```

`implementationProcedure` can be used for processing various sequences. Let us find the step response of the system.

```
In[7]:=  inpSeq = UnitStepSequence[];
         initState = {0};
         params = {F};
```

```
In[10]:= {outSeq, finalState} = DiscreteSystemImplementationProcessing[
            inpSeq, initState, params, implementationProcedure];
         outSeq // TraditionalForm
```

Out[11]//TraditionalForm=

$$\begin{pmatrix} F(0)+1 \\ F(F(0)+1)+1 \\ F(F(F(0)+1)+1)+1 \\ F(F(F(F(0)+1)+1)+1)+1 \\ F(F(F(F(F(0)+1)+1)+1)+1)+1 \\ F(F(F(F(F(F(0)+1)+1)+1)+1)+1)+1 \\ F(F(F(F(F(F(F(0)+1)+1)+1)+1)+1)+1)+1 \\ F(F(F(F(F(F(F(F(0)+1)+1)+1)+1)+1)+1)+1)+1 \end{pmatrix}$$

Symbolic processing is the *SchematicSolver*'s unique feature not available in other software. The above example demonstrates that *SchematicSolver* returns the output sequence with symbolic sample values in terms of a symbolic function name F.

Any name of a built-in algebraic single-argument function can be substituted for *F*:

```
In[12]:= outSeq /. F → Abs // TraditionalForm
```

Out[12]//TraditionalForm=

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

You can define an algebraic single-argument function

```
In[13]:= myFunc[x_] := 2 * Abs[x / 3]
```

and substitute for *F*:

```
In[14]:= outSeq /. F → myFunc // TraditionalForm
```

Out[14]//TraditionalForm=

$$\begin{pmatrix} 1 \\ \frac{5}{3} \\ \frac{19}{9} \\ \frac{65}{27} \\ \frac{211}{81} \\ \frac{665}{243} \\ \frac{2059}{729} \\ \frac{6305}{2187} \end{pmatrix}$$

Alternatively, any name of a built-in algebraic single-argument function can be substituted for *F* in the list of the parameters:

```
In[15]:= params2 = {Abs};
         {outSeq2, finalState2} = DiscreteSystemImplementationProcessing[
            inpSeq, initState, params2, implementationProcedure];
         outSeq2 // TraditionalForm
```

Out[17]//TraditionalForm=

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

```
In[18]:= params3 = {myFunc};
         {outSeq3, finalState3} = DiscreteSystemImplementationProcessing[
            inpSeq, initState, params3, implementationProcedure];
         outSeq3 // TraditionalForm
```

Out[20]//TraditionalForm=

$$\begin{pmatrix} 1 \\ \frac{5}{3} \\ \frac{19}{9} \\ \frac{65}{27} \\ \frac{211}{81} \\ \frac{665}{243} \\ \frac{2059}{729} \\ \frac{6305}{2187} \end{pmatrix}$$

## Function-Element Value as Built-in *Mathematica* Function

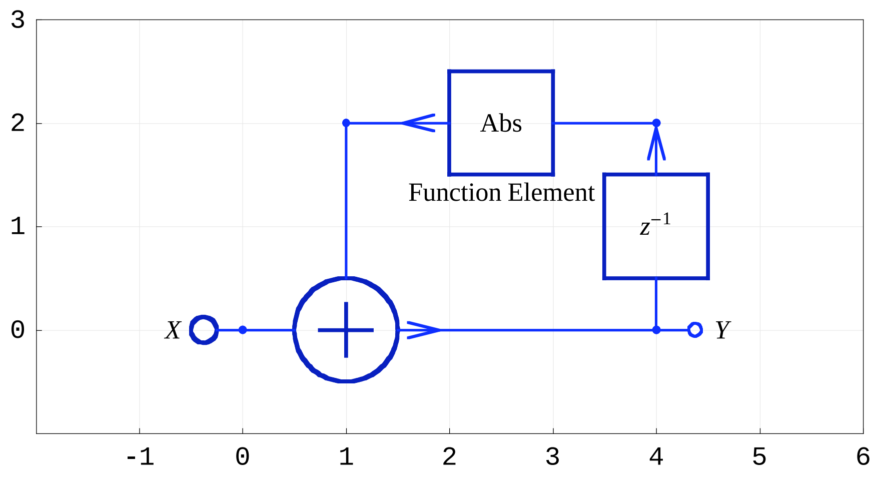
Function-element value can be any algebraic *Mathematica* built-in function of one argument.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[21]:= Needs["SchematicSolver`"];
```

Here is an example system:

```
In[22]:= discreteSystemAbs = {{"Input", {0, 0}, X},
            {"Output", {4, 0}, Y},
            {"Adder", {{0, 0}, {1, -1}, {4, 0}, {1, 2}}, {1, 0, 2, 1}},
            {"Delay", {{4, 0}, {4, 2}}},
            {"Function", {{4, 2}, {1, 2}}, Abs, "Function Element"}};
         ShowSchematic[%, PlotRange → {{-2, 6}, {-1, 3}}];
```



`DiscreteSystemImplementationSummary` points out the system input, initial state, parameter set, output, and final state:

```
In[24]:= DiscreteSystemImplementationSummary[discreteSystemAbs]

           Input: {Y[{0, 0}]}

           Initial state: {Y[{4, 2}]}

           Parameter: {}

           Output: {Y[{4, 0}]}

           Final state: {Y[{4, 0}]}
```

The system `discreteSystemAbs` has no parameters.

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system (the default name of this function is `implementationProcedure`):

```
In[25]:= DiscreteSystemImplementation[discreteSystemAbs];

           Implementation procedure name: implementationProcedure

           Implementation procedure usage:

            {{Y4p0}, {Y4p0}} = implementationProcedure[{Y0p0},{Y4p2},{}] is the template
             for calling the procedure.  The general template is {outputSamples,
             finalConditions} = procedureName[inputSamples, initialConditions,
             systemParameters]. See also: DiscreteSystemImplementationProcessing
```

`implementationProcedure` can be used for processing various sequences. Let us find the step response of the system.

```
In[26]:= inpSeq = UnitStepSequence[];
         initState = {0};
         params = {};
```

```
In[29]:= {outSeq, finalState} = DiscreteSystemImplementationProcessing[
            inpSeq, initState, params, implementationProcedure];
         outSeq // TraditionalForm
```

Out[30]//TraditionalForm=

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

In this example, the Function-element value has been given in the schematic specification. Consequently, we cannot specify a new Function-element value as an argument to `DiscreteSystemImplementationProcessing`.

## Function-Element Value as User-Defined Function

Function-element value can be any algebraic user-defined function of one argument:

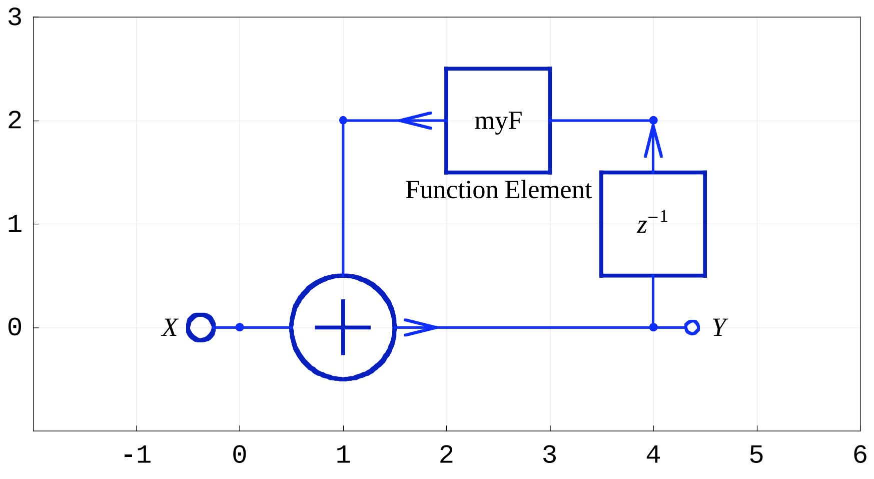
```
In[31]:= myF[x_] := Module[{t}, t = Round[x]; t + 1 / 2 * Sign[x]];
```

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[32]:= Needs["SchematicSolver`"];
```

Here is an example system:

```
In[33]:= discreteSystemMyF = {{"Input", {0, 0}, X},
            {"Output", {4, 0}, Y},
            {"Adder", {{0, 0}, {1, -1}, {4, 0}, {1, 2}}, {1, 0, 2, 1}},
            {"Delay", {{4, 0}, {4, 2}}},
            {"Function", {{4, 2}, {1, 2}}, myF, "Function Element"}};
         ShowSchematic[%, PlotRange → {{-2, 6}, {-1, 3}}];
```



`DiscreteSystemImplementationSummary` points out the system input, initial state, parameter set, output, and final state:

```
In[35]:= DiscreteSystemImplementationSummary[discreteSystemMyF]

           Input: {Y[{0, 0}]}

           Initial state: {Y[{4, 2}]}

           Parameter: {}

           Output: {Y[{4, 0}]}

           Final state: {Y[{4, 0}]}
```

The system `discreteSystemMyF` has no parameters.

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system (the default name of this function is `implementationProcedure`):

```
In[36]:= DiscreteSystemImplementation[discreteSystemMyF];

           Implementation procedure name: implementationProcedure

           Implementation procedure usage:

            {{Y4p0}, {Y4p0}} = implementationProcedure[{Y0p0},{Y4p2},{}] is the template
             for calling the procedure.  The general template is {outputSamples,
             finalConditions} = procedureName[inputSamples, initialConditions,
             systemParameters]. See also: DiscreteSystemImplementationProcessing
```

Here is a processing example.

```
In[37]:= inpSeq = UnitExponentialSequence[];
         initState = {0};
         params = {};

In[40]:= {outSeq, finalState} = DiscreteSystemImplementationProcessing[
            inpSeq, initState, params, implementationProcedure];
         outSeq // TraditionalForm
```

Out[41]//TraditionalForm=

$$\begin{pmatrix} 1 \\ 2 \\ \frac{11}{4} \\ \frac{29}{8} \\ \frac{73}{16} \\ \frac{177}{32} \\ \frac{417}{64} \\ \frac{961}{128} \end{pmatrix}$$

The implementation procedure embeds the code of the user-defined function:

```
In[42]:= ?? implementationProcedure

          {{Y4p0}, {Y4p0}} = implementationProcedure[{Y0p0},{Y4p2},{}] is the template
           for calling the procedure.  The general template is {outputSamples,
           finalConditions} = procedureName[inputSamples, initialConditions,
           systemParameters]. See also: DiscreteSystemImplementationProcessing

         implementationProcedure[] := {1, 1, 0, 4, 1, 1}

         implementationProcedure[dataSamples_List,
           initialConditions_List, systemParameters_List] :=
          Module[{Y0p0, Y4p2, Y1p2, Y4p0}, {Y0p0} = dataSamples; {Y4p2} = initialConditions;
           Y1p2 = Round[Y4p2] + Sign[Y4p2]/2; Y4p0 = Y0p0 + Y1p2; {{Y4p0}, {Y4p0}}]
```

## Function-Element Value as Parameterized Function

Function-element value can be any algebraic user-defined function of one argument, and the function can contain parameters:

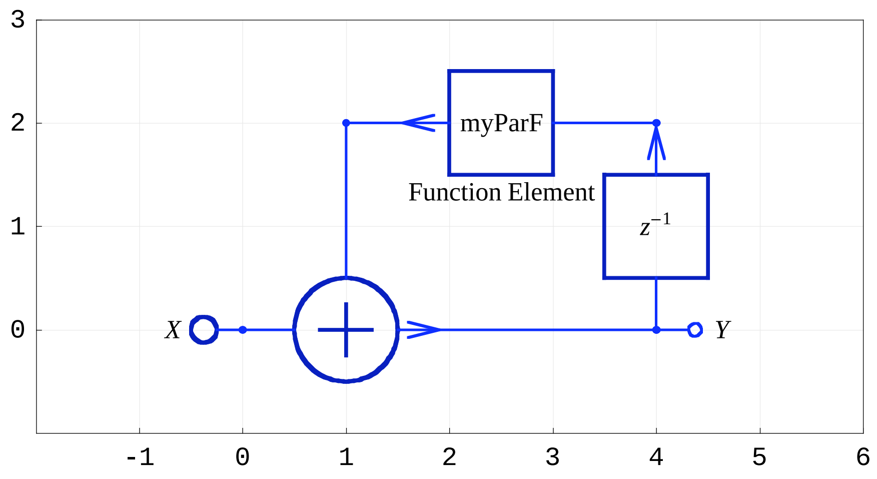
```
In[43]:= Clear[p];
         myParF[x_] := p * Abs[x];
```

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[45]:= Needs["SchematicSolver`"];
```

Here is an example system:

```
In[46]:= discreteSystemParF = {{"Input", {0, 0}, X},
            {"Output", {4, 0}, Y},
            {"Adder", {{0, 0}, {1, -1}, {4, 0}, {1, 2}}, {1, 0, 2, 1}},
            {"Delay", {{4, 0}, {4, 2}}},
            {"Function", {{4, 2}, {1, 2}}, myParF, "Function Element"}};
         ShowSchematic[%, PlotRange → {{-2, 6}, {-1, 3}}];
```



`DiscreteSystemImplementationSummary` points out the system input, initial state, parameter set, output, and final state:

```
In[48]:= DiscreteSystemImplementationSummary[discreteSystemParF]

           Input: {Y[{0, 0}]}

           Initial state: {Y[{4, 2}]}

           Parameter: {}

           Output: {Y[{4, 0}]}

           Final state: {Y[{4, 0}]}
```

The system `discreteSystemParF` has no parameters.

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system (the default name of this function is `implementationProcedure`):

```
In[49]:= DiscreteSystemImplementation[discreteSystemParF];

           Implementation procedure name: implementationProcedure

           Implementation procedure usage:

            {{Y4p0}, {Y4p0}} = implementationProcedure[{Y0p0},{Y4p2},{}] is the template
             for calling the procedure.  The general template is {outputSamples,
             finalConditions} = procedureName[inputSamples, initialConditions,
             systemParameters]. See also: DiscreteSystemImplementationProcessing
```

Here is a processing example.

```
In[50]:= inpSeq = UnitExponentialSequence[];
         initState = {0};
         params = {};

In[53]:= {outSeq, finalState} = DiscreteSystemImplementationProcessing[
            inpSeq, initState, params, implementationProcedure];
         outSeq // TraditionalForm
```

Out[54]//TraditionalForm=

$$\begin{pmatrix} 1 \\ p+\frac{1}{2} \\ p\,|p+\frac{1}{2}|+\frac{1}{4} \\ p\,|p\,|p+\frac{1}{2}|+\frac{1}{4}|+\frac{1}{8} \\ p\,|p\,|p\,|p+\frac{1}{2}|+\frac{1}{4}|+\frac{1}{8}|+\frac{1}{16} \\ p\,|p\,|p\,|p\,|p+\frac{1}{2}|+\frac{1}{4}|+\frac{1}{8}|+\frac{1}{16}|+\frac{1}{32} \\ p\,|p\,|p\,|p\,|p\,|p+\frac{1}{2}|+\frac{1}{4}|+\frac{1}{8}|+\frac{1}{16}|+\frac{1}{32}|+\frac{1}{64} \\ p\,|p\,|p\,|p\,|p\,|p\,|p+\frac{1}{2}|+\frac{1}{4}|+\frac{1}{8}|+\frac{1}{16}|+\frac{1}{32}|+\frac{1}{64}|+\frac{1}{128} \end{pmatrix}$$

```
In[55]:= outSeq /. p → -1 / 2 // TraditionalForm
```

Out[55]//TraditionalForm=

$$\begin{pmatrix} 1 \\ 0 \\ \frac{1}{4} \\ 0 \\ \frac{1}{16} \\ 0 \\ \frac{1}{64} \\ 0 \end{pmatrix}$$

The implementation procedure embeds *p* of the user-defined function:

```
In[56]:= ?? implementationProcedure
```

```
    {{Y4p0}, {Y4p0}} = implementationProcedure[{Y0p0},{Y4p2},{}] is the template
     for calling the procedure.  The general template is {outputSamples,
     finalConditions} = procedureName[inputSamples, initialConditions,
     systemParameters]. See also: DiscreteSystemImplementationProcessing

  implementationProcedure[] := {1, 1, 0, 4, 1, 1}

  implementationProcedure[dataSamples_List,
     initialConditions_List, systemParameters_List] :=
    Module[{Y0p0, Y4p2, Y1p2, Y4p0}, {Y0p0} = dataSamples; {Y4p2} = initialConditions;
     Y1p2 = p Abs[Y4p2]; Y4p0 = Y0p0 + Y1p2; {{Y4p0}, {Y4p0}}]
```

Symbolic processing is the *SchematicSolver*'s unique feature not available in other software. The above example demonstrates that *SchematicSolver* returns the output sequence with symbolic sample values in terms of a symbolic parameter.

## 8.3. Nonlinear Modulator Element

### Symbolic Solving Nonlinear System

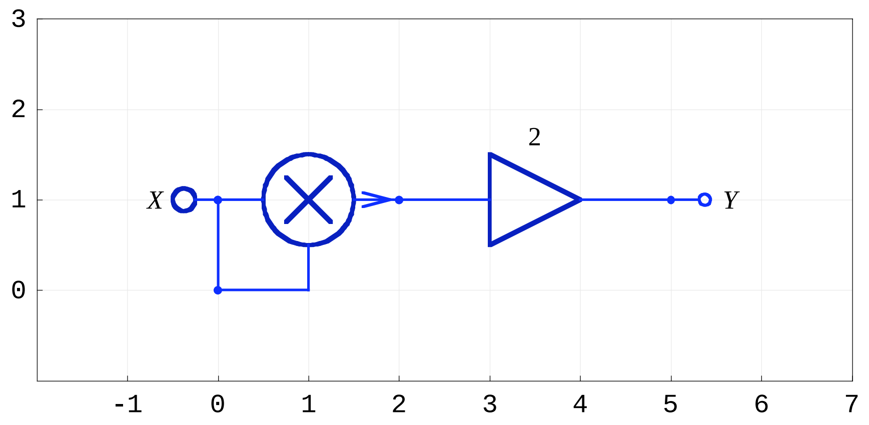
Modulator element can be used for multiplication of two or three signals. If the same signal is applied to two modulator inputs, the output signal is proportional to the signal power.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[57]:= Needs["SchematicSolver`"];
```

Here is an example modulator system:

```
In[58]:= modulatorSystem = {{"Input", {0, 1}, X},
            {"Output", {5, 1}, Y},
            {"Multiplier", {{2, 1}, {5, 1}}, 2, ""},
            {"Line", {{0, 1}, {0, 0}}},
            {"Modulator", {{0, 1}, {0, 0}, {2, 1}, {0, 2}}, {1, 1, 2, 0}}};
         ShowSchematic[%, PlotRange → {{-2, 7}, {-1, 3}}];
```



Assume that the input signal is a unit sine sequence, of 8 samples, of digital frequency Fx:

```
In[60]:= x = UnitSineSequence[8, Fx]

Out[60]= {{0}, {Sin[2 Fx π]}, {Sin[4 Fx π]}, {Sin[6 Fx π]},
          {Sin[8 Fx π]}, {Sin[10 Fx π]}, {Sin[12 Fx π]}, {Sin[14 Fx π]}}
```

`DiscreteSystemSimulation` simulates the system:

```
In[61]:=  y = DiscreteSystemSimulation[modulatorSystem, x]

Out[61]=  {{0}, {2 Sin[2 Fx π]^2}, {2 Sin[4 Fx π]^2}, {2 Sin[6 Fx π]^2},
           {2 Sin[8 Fx π]^2}, {2 Sin[10 Fx π]^2}, {2 Sin[12 Fx π]^2}, {2 Sin[14 Fx π]^2}}
```

*SchematicSolver* works with symbolic signals, so both sequences have symbolic sample values.

We can use `MultiplexSequence` to present the output and input sequence in a more traditional form:

```
In[62]:=  MultiplexSequence[x, y];
          % // TrigReduce // TraditionalForm
```

Out[63]//TraditionalForm=

$$\begin{pmatrix} 0 & 0 \\ \sin(2\,\mathrm{Fx}\,\pi) & 1-\cos(4\,\mathrm{Fx}\,\pi) \\ \sin(4\,\mathrm{Fx}\,\pi) & 1-\cos(8\,\mathrm{Fx}\,\pi) \\ \sin(6\,\mathrm{Fx}\,\pi) & 1-\cos(12\,\mathrm{Fx}\,\pi) \\ \sin(8\,\mathrm{Fx}\,\pi) & 1-\cos(16\,\mathrm{Fx}\,\pi) \\ \sin(10\,\mathrm{Fx}\,\pi) & 1-\cos(20\,\mathrm{Fx}\,\pi) \\ \sin(12\,\mathrm{Fx}\,\pi) & 1-\cos(24\,\mathrm{Fx}\,\pi) \\ \sin(14\,\mathrm{Fx}\,\pi) & 1-\cos(28\,\mathrm{Fx}\,\pi) \end{pmatrix}$$

Note that $y = 2\,x^2 = 2\sin(n\,2\,\pi\,F_x)^2 = 1-\cos(n\,4\,\pi\,F_x)$, for $n = 0, 1, 2, ..., 7$. This formula can be derived by using `DiscreteSystemImplementation`:

```
In[64]:=  DiscreteSystemImplementation[modulatorSystem];

            Implementation procedure name: implementationProcedure

            Implementation procedure usage:

             {{Y5p1}, {}} = implementationProcedure[{Y0p1},{},{}] is the template
              for calling the procedure.  The general template is {outputSamples,
              finalConditions} = procedureName[inputSamples, initialConditions,
              systemParameters]. See also: DiscreteSystemImplementationProcessing
```

Define input sample list symbolically as

```
In[65]:=  Clear[Fx, n];
          inpSamples = {Sin[n * 2 * Pi * Fx]}

Out[66]=  {Sin[2 Fx n π]}
```

Process the sample list with `implementationProcedure`:

```
In[67]:=  {outSamples, finalState} = implementationProcedure[inpSamples, {}, {}]

Out[67]=  {{2 Sin[2 Fx n π]^2}, {}}
```

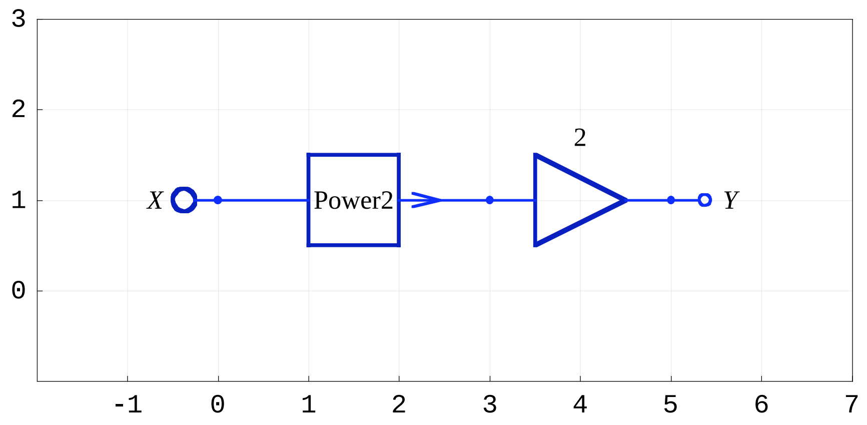
Use *Mathematica* built-in functions to get better insight into the result:

```
In[68]:=  outSamples // TrigReduce

Out[68]=  {1 - Cos[4 Fx n π]}
```

Note that the output sequence has a constant term and a sinusoidal component of digital frequency `2*Fx`.

Alternatively, the same result can be generated with the *SchematicSolver*'s function `Power2`.

```
In[69]:= power2System = {{"Input", {0, 1}, X},
            {"Output", {5, 1}, Y},
            {"Multiplier", {{3, 1}, {5, 1}}, 2, ""},
            {"Function", {{0, 1}, {3, 1}}, Power2}};
         ShowSchematic[%, PlotRange → {{-2, 7}, {-1, 3}}];
```



```
In[71]:= x2 = UnitSineSequence[8, Fx]

Out[71]= {{0}, {Sin[2 Fx π]}, {Sin[4 Fx π]}, {Sin[6 Fx π]},
          {Sin[8 Fx π]}, {Sin[10 Fx π]}, {Sin[12 Fx π]}, {Sin[14 Fx π]}}

In[72]:= y2 = DiscreteSystemSimulation[power2System, x2]

Out[72]= {{0}, {2 Sin[2 Fx π]^2}, {2 Sin[4 Fx π]^2}, {2 Sin[6 Fx π]^2},
          {2 Sin[8 Fx π]^2}, {2 Sin[10 Fx π]^2}, {2 Sin[12 Fx π]^2}, {2 Sin[14 Fx π]^2}}

In[73]:= y2 // TrigReduce

Out[73]= {{0}, {1 - Cos[4 Fx π]}, {1 - Cos[8 Fx π]}, {1 - Cos[12 Fx π]},
          {1 - Cos[16 Fx π]}, {1 - Cos[20 Fx π]}, {1 - Cos[24 Fx π]}, {1 - Cos[28 Fx π]}}
```

Both systems `modulatorSystem` and `power2System` yield the same result:

```
In[74]:= SameQ[y, y2]

Out[74]= True
```

# 9. Examples of Discrete System Implementation

## 9.1. Adaptive LMS System

### System Identification using Adaptive Filters

Adaptive filters can be used to identify an unknown system. If the impulse response of the unknown system has a finite duration, FIR filters can be used to model the unknown system. Least mean squares (LMS) algorithm can be used to determine the coefficients of such FIR filters.

Two sequences should be known for designing an adaptive filter:

1) the input sequence to the unknown system, `inputSignal`, and

2) the output sequence from the unknown system, `desiredSignal`.

LMS algorithm is used to determine the coefficients of the FIR filter that has approximately the same response `filteredSignal` as the unknown system. For the same input `inputSignal` to the unknown system and the adaptive FIR filter, when the difference

```
errorSignal = desiredSignal - filteredSignal
```

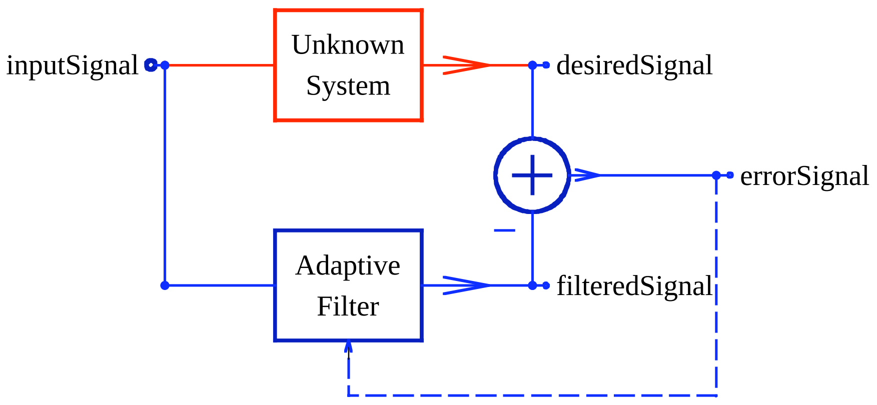
goes to zero and remains there, we achieve a perfect adaptation.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

In the figure below, the unknown system is placed in parallel with the adaptive filter.

```
identificationSystem = {
 {"Input", {2, 10}, "inputSignal"},
{"Block", {{2, 10}, {12, 10}}, "Unknown\n  System",
    "", ElementSize → {4, 3}, PlotStyle → {{Hue[1]}, {Hue[1]}}},
{"Block", {{2, 4}, {12, 4}}, "Adaptive\n   Filter", "", ElementSize → {4, 3}},
{"Line", {{2, 10}, {2, 4}}},
{"Adder", {{10, 7}, {12, 4}, {17, 7}, {12, 10}},
    {0, -1, 2, 1}, "", ElementSize → {2, 2}},
{"Output", {12, 10}, "desiredSignal"},
{"Output", {12, 4}, "filteredSignal"},
{"Output", {17, 7}, "errorSignal"},
{"Polyline", {{17, 7}, {17, 1}, {7, 1}, {7, 2}}},
{"Arrow", {{7, 2.5}, {7, 2}}}};
 ShowSchematic[%, PlotRange → {{-3, 22}, {0, 12}},
   FontSize → 11, GridLines → None, Frame → False];
```



We use the following notation:

`inputSignal` is the signal that is fed to the unknown system and to the adaptive FIR filter.

`desiredSignal` is the signal at the output of the unknown system.

`filteredSignal` is the signal at the output of the adaptive FIR filter.

`errorSignal` is the difference of the desired signal and the filtered signal.

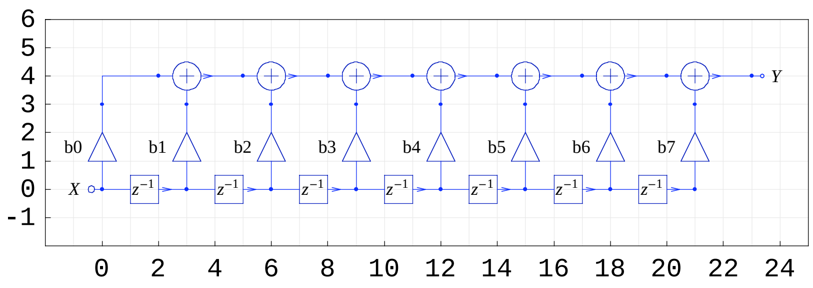
## Example Unknown System

Assume that the unknown system can be represented by the following schematic:

```
parameterSymbols = UnitSymbolicSequence[8, b, 0] // SequenceToList
```

```
{b0, b1, b2, b3, b4, b5, b6, b7}
```

```
{unknownSystemSchematic, inpCoords, outCoords} =
  DirectFormFIRFilterSchematic[parameterSymbols];

SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];

unknownSystem = Join[
   unknownSystemSchematic,
   {{"Input", inpCoords[[1]], X},
    {"Output", outCoords[[1]], Y}}];
ShowSchematic[%, FontSize → 7, PlotRange → {{-2, 25}, {-2, 6}}];
```



The system summary, generated by `DiscreteSystemImplementationSummary`, points out the system input, initial state, parameter set, output, and final state.

```
DiscreteSystemImplementationSummary[unknownSystem]
```

```
    Input: {Y[{0, 0}]}

    Initial state:
     {Y[{3, 0}], Y[{6, 0}], Y[{9, 0}], Y[{12, 0}], Y[{15, 0}], Y[{18, 0}], Y[{21, 0}]}

    Parameter: {b0, b1, b2, b3, b4, b5, b6, b7}

    Output: {Y[{23, 4}]}

    Final state:
     {Y[{0, 0}], Y[{3, 0}], Y[{6, 0}], Y[{9, 0}], Y[{12, 0}], Y[{15, 0}], Y[{18, 0}]}
```

Notice that the assumed unknown system has 8 parameters (8 multiplier coefficients).

## Specify Parameters of the Unknown System and the Input Signal

Let us define the numeric values of the system parameters

```
parameterValues = {-0.005, -0.02, 0.1 , 0.4, 0.5, 0.08, -0.03, -0.002};
parameterSubstitution = parameterSymbols → parameterValues // Thread
```

```
{b0 → -0.005, b1 → -0.02, b2 → 0.1, b3 → 0.4, b4 → 0.5, b5 → 0.08, b6 → -0.03, b7 → -0.002}
```

and the input data

```
inputData = {-0.0026, -0.1111, 0.0751, 0.05, -0.0517,
   -0.0559, -0.0753, 0.0926, -0.0249, -0.015, -0.1258, 0.0313, 0.2690,
   0.0290, -0.1423, 0.0247, -0.1436, 0.0149, -0.1693 , 0.0719, 0};
```

The input signal is a sequence that is obtained from the input data as

```
inputSignal = ListToSequence[inputData]
```

```
{{-0.0026}, {-0.1111}, {0.0751}, {0.05}, {-0.0517}, {-0.0559}, {-0.0753},
 {0.0926}, {-0.0249}, {-0.015}, {-0.1258}, {0.0313}, {0.269}, {0.029},
 {-0.1423}, {0.0247}, {-0.1436}, {0.0149}, {-0.1693}, {0.0719}, {0}}
```

## Symbolic Response of the Unknown System

We can compute the desired signal for symbolic parameters:

```
desiredSignalSymbolic = DiscreteSystemSimulation[unknownSystem, inputSignal]
```

```
{{-0.0026 b0}, {-0.1111 b0 - 0.0026 b1},
 {0.0751 b0 - 0.1111 b1 - 0.0026 b2}, {0.05 b0 + 0.0751 b1 - 0.1111 b2 - 0.0026 b3},
 {-0.0517 b0 + 0.05 b1 + 0.0751 b2 - 0.1111 b3 - 0.0026 b4},
 {-0.0559 b0 - 0.0517 b1 + 0.05 b2 + 0.0751 b3 - 0.1111 b4 - 0.0026 b5},
 {-0.0753 b0 - 0.0559 b1 - 0.0517 b2 + 0.05 b3 + 0.0751 b4 - 0.1111 b5 - 0.0026 b6},
 {0.0926 b0 - 0.0753 b1 - 0.0559 b2 - 0.0517 b3 + 0.05 b4 + 0.0751 b5 - 0.1111 b6 - 0.0026 b7},
 {-0.0249 b0 + 0.0926 b1 - 0.0753 b2 - 0.0559 b3 - 0.0517 b4 + 0.05 b5 + 0.0751 b6 - 0.1111 b7},
 {-0.015 b0 - 0.0249 b1 + 0.0926 b2 - 0.0753 b3 - 0.0559 b4 - 0.0517 b5 + 0.05 b6 + 0.0751 b7},
 {-0.1258 b0 - 0.015 b1 - 0.0249 b2 + 0.0926 b3 - 0.0753 b4 - 0.0559 b5 - 0.0517 b6 + 0.05 b7},
 {0.0313 b0 - 0.1258 b1 - 0.015 b2 - 0.0249 b3 + 0.0926 b4 - 0.0753 b5 - 0.0559 b6 - 0.0517 b7},
 {0.269 b0 + 0.0313 b1 - 0.1258 b2 - 0.015 b3 - 0.0249 b4 + 0.0926 b5 - 0.0753 b6 - 0.0559 b7},
 {0.029 b0 + 0.269 b1 + 0.0313 b2 - 0.1258 b3 - 0.015 b4 - 0.0249 b5 + 0.0926 b6 - 0.0753 b7},
 {-0.1423 b0 + 0.029 b1 + 0.269 b2 + 0.0313 b3 - 0.1258 b4 - 0.015 b5 - 0.0249 b6 + 0.0926 b7},
 {0.0247 b0 - 0.1423 b1 + 0.029 b2 + 0.269 b3 + 0.0313 b4 - 0.1258 b5 - 0.015 b6 - 0.0249 b7},
 {-0.1436 b0 + 0.0247 b1 - 0.1423 b2 + 0.029 b3 + 0.269 b4 + 0.0313 b5 - 0.1258 b6 - 0.015 b7},
 {0.0149 b0 - 0.1436 b1 + 0.0247 b2 - 0.1423 b3 + 0.029 b4 + 0.269 b5 + 0.0313 b6 - 0.1258 b7},
 {-0.1693 b0 + 0.0149 b1 - 0.1436 b2 + 0.0247 b3 - 0.1423 b4 + 0.029 b5 + 0.269 b6 + 0.0313 b7},
 {0.0719 b0 - 0.1693 b1 + 0.0149 b2 - 0.1436 b3 + 0.0247 b4 - 0.1423 b5 + 0.029 b6 + 0.269 b7},
 {0.0719 b1 - 0.1693 b2 + 0.0149 b3 - 0.1436 b4 + 0.0247 b5 - 0.1423 b6 + 0.029 b7}}
```

In addition, we can compute the desired signal for numeric system parameters:

```
desiredSignal = desiredSignalSymbolic /. parameterSubstitution
```

```
{{0.000013}, {0.0006075}, {0.0015865}, {-0.013902}, {-0.0389715},
 {-0.0194045}, {0.0450645}, {0.0091192}, {-0.0554983}, {-0.0540232},
 {-0.005192}, {0.0329559}, {-0.0232222}, {-0.0648344}, {-0.0239867},
 {0.119308}, {0.138402}, {-0.0163199}, {-0.0808941}, {-0.0533655}, {-0.078021}}
```

## Symbolic Identification of Parameters of the Unknown System

We assumed that the unknown system has 8 parameters. Therefore, 8 equations are required to determine the parameters.

For given `inputSignal` and `desiredSignal` we set up the linear system of equations and compute the unknown system parameters. For example, for 8 samples starting from the 9th sample, we obtain

```
i = 9;
Solve[{desiredSignalSymbolic[[i]] == desiredSignal[[i]],
  desiredSignalSymbolic[[i + 1]] == desiredSignal[[i + 1]],
  desiredSignalSymbolic[[i + 2]] == desiredSignal[[i + 2]],
  desiredSignalSymbolic[[i + 3]] == desiredSignal[[i + 3]],
  desiredSignalSymbolic[[i + 4]] == desiredSignal[[i + 4]],
  desiredSignalSymbolic[[i + 5]] == desiredSignal[[i + 5]],
  desiredSignalSymbolic[[i + 6]] == desiredSignal[[i + 6]],
  desiredSignalSymbolic[[i + 7]] == desiredSignal[[i + 7]]},
 parameterSymbols]

{{b0 → -0.005, b1 → -0.02, b2 → 0.1, b3 → 0.4, b4 → 0.5, b5 → 0.08, b6 → -0.03, b7 → -0.002}}
```

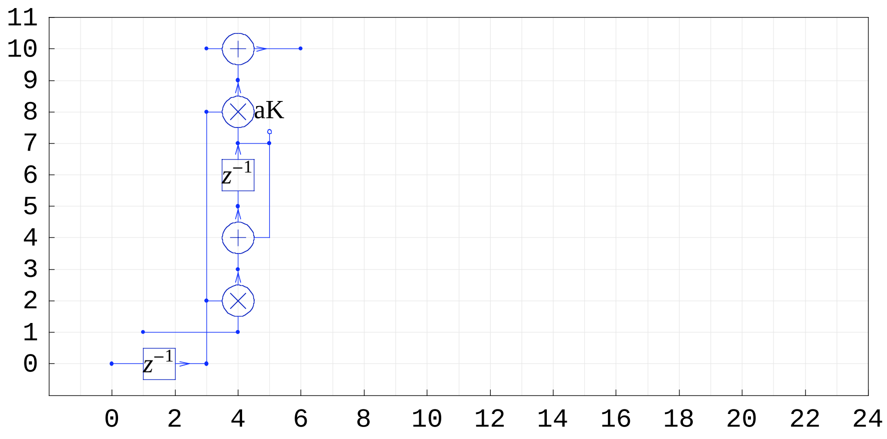
The same result appears if we take the 8 samples starting from the 11th sample:

```
i = 11;
Solve[{desiredSignalSymbolic[[i]] == desiredSignal[[i]],
  desiredSignalSymbolic[[i + 1]] == desiredSignal[[i + 1]],
  desiredSignalSymbolic[[i + 2]] == desiredSignal[[i + 2]],
  desiredSignalSymbolic[[i + 3]] == desiredSignal[[i + 3]],
  desiredSignalSymbolic[[i + 4]] == desiredSignal[[i + 4]],
  desiredSignalSymbolic[[i + 5]] == desiredSignal[[i + 5]],
  desiredSignalSymbolic[[i + 6]] == desiredSignal[[i + 6]],
  desiredSignalSymbolic[[i + 7]] == desiredSignal[[i + 7]]},
 parameterSymbols]

{{b0 → -0.005, b1 → -0.02, b2 → 0.1, b3 → 0.4, b4 → 0.5, b5 → 0.08, b6 → -0.03, b7 → -0.002}}
```

## Adaptive System

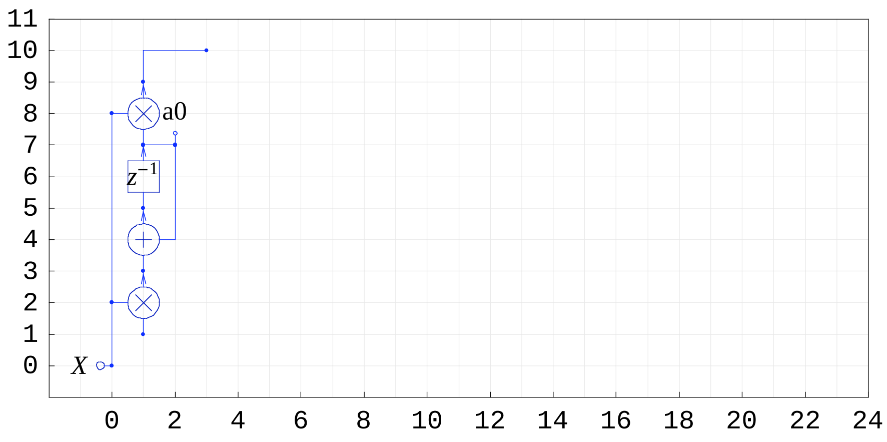
First, we draw smaller parts that constitute the adaptive system. Here is the basic stage:

```
stageSubsystem = {{"Output", {5, 7}, aK, "", TextOffset → {0, -1}},
   {"Modulator", {{3, 2}, {4, 1}, {5, 2}, {4, 3}}, {1, 1, 0, 2}},
   {"Modulator", {{3, 8}, {4, 7}, {5, 8}, {4, 9}}, {1, 1, 0, 2}},
   {"Adder", {{3, 4}, {4, 3}, {5, 7}, {4, 5}}, {0, 1, 1, 2}},
   {"Adder", {{3, 10}, {4, 9}, {6, 10}, {4, 11}}, {1, 1, 2, 0}},
   {"Line", {{3, 0}, {3, 2}}}, {"Line", {{3, 2}, {3, 8}}},
   {"Line", {{4, 7}, {5, 7}}}, {"Line", {{1, 1}, {4, 1}}},
   {"Delay", {{0, 0}, {3, 0}}, 1}, {"Delay", {{4, 5}, {4, 7}}, 1}};
ShowSchematic[%, PlotRange → {{-2, 24}, {-1, 11}}];
```

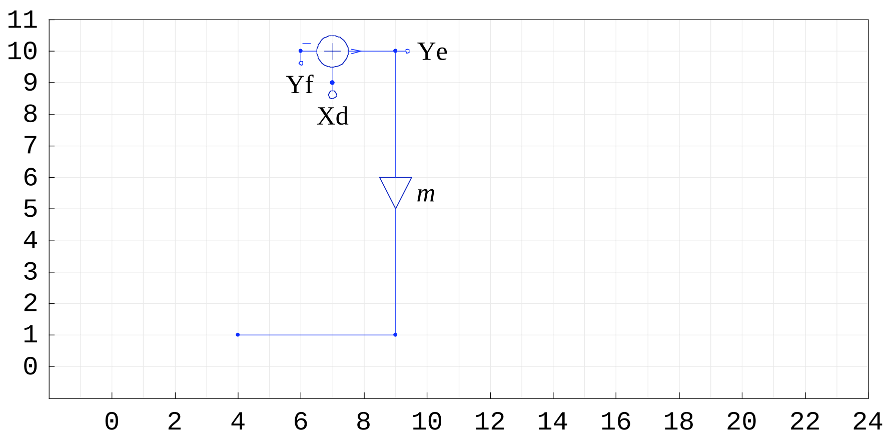


The input and the output parts of the adaptive system look like these:

```
inputSubsystem =  {{"Output", {2, 7}, a0, "", TextOffset → {0, -1}},
   {"Modulator", {{0, 2}, {1, 1}, {2, 2}, {1, 3}}, {1, 1, 0, 2}},
   {"Modulator", {{0, 8}, {1, 7}, {2, 8}, {1, 9}}, {1, 1, 0, 2}},
   {"Adder", {{0, 4}, {1, 3}, {2, 7}, {1, 5}}, {0, 1, 1, 2}},
   {"Line", {{0, 0}, {0, 2}}}, {"Line", {{0, 2}, {0, 8}}},
   {"Line", {{1, 7}, {2, 7}}}, {"Line", {{1, 9}, {1, 10}, {3, 10}}},
   {"Input", {0, 0}, X}, {"Delay", {{1, 5}, {1, 7}}, 1}};
ShowSchematic[%, PlotRange → {{-2, 24}, {-1, 11}}];
```

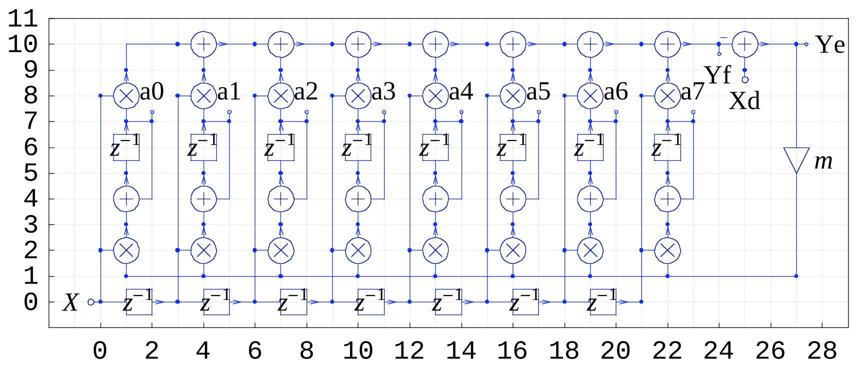


```
outputSubsystem = {{"Input", {7, 9}, Xd, "", TextOffset → {0, 1}},
   {"Multiplier", {{9, 10}, {9, 1}}, m},
   {"Adder", {{6, 10}, {7, 9}, {9, 10}, {7, 11}}, {-1, 1, 2, 0}},
   {"Output", {9, 10}, Ye},
   {"Output", {6, 10}, Yf, "", TextOffset → {0, 1}}, {"Line", {{9, 1}, {4, 1}}}};
ShowSchematic[%, PlotRange → {{-2, 24}, {-1, 11}}];
```



We generate the adaptive system by replicating the basic stage and by adding the input and the output parts:

```
numberOfStages = 7;
adaptiveSystem = TranslateSchematic[outputSubsystem, {(numberOfStages - 1) * 3, 0}];
adaptiveSystem = Join[adaptiveSystem, inputSubsystem];
Do[adaptiveSystem = Join[adaptiveSystem, TranslateSchematic[stageSubsystem /.
        aK → ToExpression["a" ~ StringJoin ~ ToString[k]], {(k - 1) * 3, 0}]];
  , {k, numberOfStages}];
ShowSchematic[adaptiveSystem, PlotRange → {{-2, numberOfStages * 3 + 8}, {-1, 11}}];
```



The system summary points out the input, initial state, parameter set, output, and final state of the adaptive system.

```
DiscreteSystemImplementationSummary[adaptiveSystem]
```

```
    Input: {Y[{25, 9}], Y[{0, 0}]}

    Initial state: {Y[{1, 7}], Y[{3, 0}], Y[{4, 7}], Y[{6, 0}],
      Y[{7, 7}], Y[{9, 0}], Y[{10, 7}], Y[{12, 0}], Y[{13, 7}], Y[{15, 0}],
      Y[{16, 7}], Y[{18, 0}], Y[{19, 7}], Y[{21, 0}], Y[{22, 7}]}

    Parameter: {m}

    Output: {Y[{27, 10}], Y[{24, 10}], Y[{1, 7}], Y[{4, 7}],
      Y[{7, 7}], Y[{10, 7}], Y[{13, 7}], Y[{16, 7}], Y[{19, 7}], Y[{22, 7}]}

    Final state: {Y[{1, 5}], Y[{0, 0}], Y[{4, 5}], Y[{3, 0}],
      Y[{7, 5}], Y[{6, 0}], Y[{10, 5}], Y[{9, 0}], Y[{13, 5}], Y[{12, 0}],
      Y[{16, 5}], Y[{15, 0}], Y[{19, 5}], Y[{18, 0}], Y[{22, 5}]}
```

errorSignal appears at Y[{27,10}] and filteredSignal appears at Y[{24,10}].

The adaptive filter coefficients are computed from the signals at the remaining outputs Y[{1,7}], Y[{4,7}], Y[{7,7}], Y[{10,7}], Y[{13,7}], Y[{16,7}], Y[{19,7}], and Y[{22,7}].

## Specifying the Parameter of Adaptive System

Assume the numeric value of the system parameter

```
m0 = 0.4;
```

Input sequence to the adaptive system is the multiplex sequence formed by desiredSignal and inputSignal.

```
inpSeq = MultiplexSequence[desiredSignal, inputSignal];
% // MatrixForm
```

```
  0.000013    -0.0026
 0.0006075   -0.1111
 0.0015865    0.0751
 -0.013902     0.05
-0.0389715   -0.0517
-0.0194045   -0.0559
 0.0450645   -0.0753
 0.0091192    0.0926
-0.0554983   -0.0249
-0.0540232    -0.015
 -0.005192   -0.1258
 0.0329559    0.0313
-0.0232222     0.269
-0.0648344     0.029
-0.0239867   -0.1423
  0.119308    0.0247
  0.138402   -0.1436
-0.0163199    0.0149
-0.0808941   -0.1693
-0.0533655    0.0719
 -0.078021       0
```

## Processing with Adaptive System

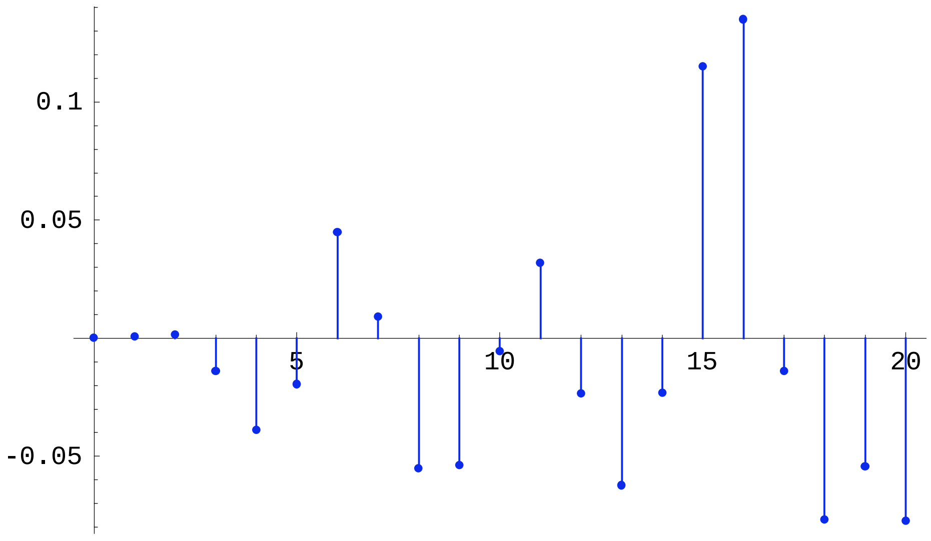
The input sequence to the adaptive system is processed for the specified parameter as follows:

```
outSeq = DiscreteSystemSimulation[adaptiveSystem /. m → m0, inpSeq];
```

The output sequence from the adaptive system consists of 10 sequences: errorSignal, filteredSignal, and 8 sequences that have the values of adapted coefficients at each processing step.

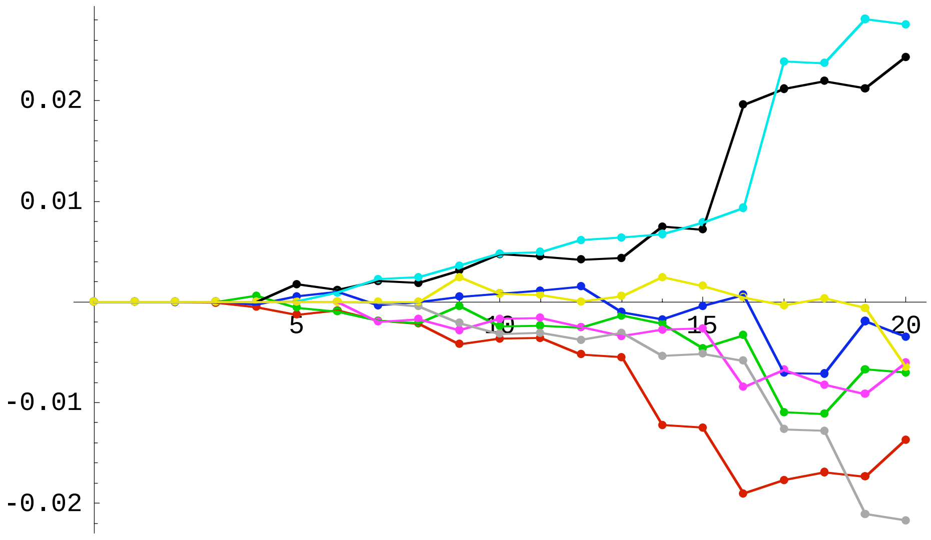
```
{errorSignal, filteredSignal, a0Seq, a1Seq, a2Seq, a3Seq, a4Seq, a5Seq, a6Seq, a7Seq} =
  DemultiplexSequence[outSeq];
```

```
SequencePlot[errorSignal];
```



The error signal increases because the coefficients are very small at the beginning of the adaptive process. The number of processed samples should be larger.

```
coefSeq = MultiplexSequence[a0Seq, a1Seq, a2Seq, a3Seq, a4Seq, a5Seq, a6Seq, a7Seq];
SequencePlot[coefSeq, PlotJoined → True, StemPlot → False];
```



The values of the adapted coefficients at the end of the process are

```
coefValues = coefSeq // Last
```

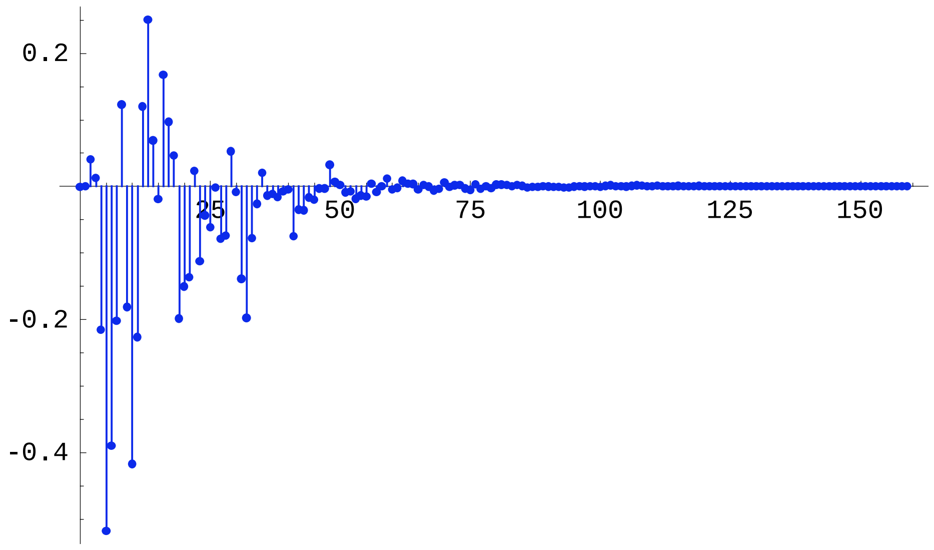
```
{-0.00347891, -0.0136919, -0.0070148,
 0.0243012, 0.0275463, -0.00605629, -0.021713, -0.00644811}
```

## Finding Adapted Coefficients

The coefficients of the adaptive system can be successfully found if the input data has sufficient number of samples. Let us repeat the same procedure with 160 random samples.

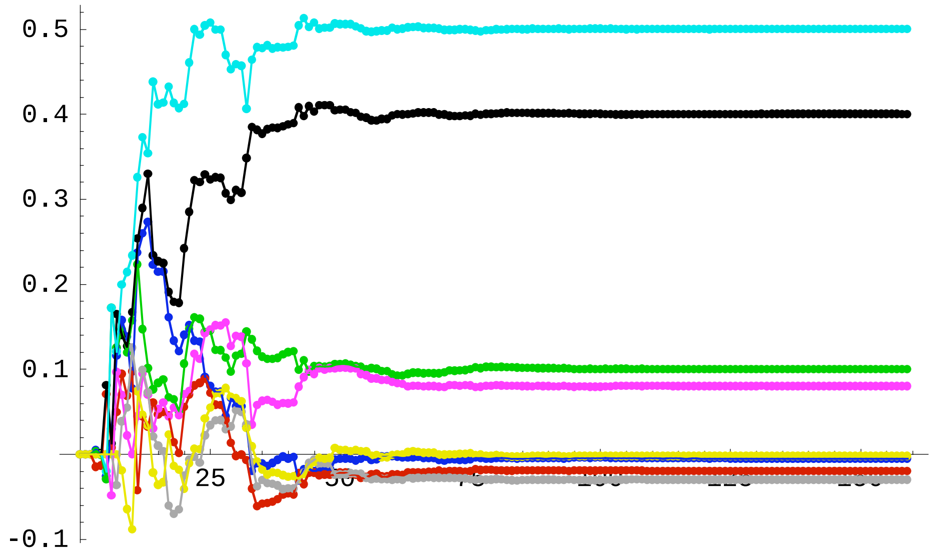
```
inputSignal2 = UnitNoiseSequence[160];
```

```
desiredSignal2 =
  DiscreteSystemSimulation[unknownSystem /. parameterSubstitution, inputSignal2];

inpSeq2 = MultiplexSequence[desiredSignal2, inputSignal2];
outSeq2 = DiscreteSystemSimulation[adaptiveSystem /. m → m0, inpSeq2];

{errorSignal2, filteredSignal2, a0Seq2, a1Seq2, a2Seq2, a3Seq2,
   a4Seq2, a5Seq2, a6Seq2, a7Seq2} = DemultiplexSequence[outSeq2];
SequencePlot[errorSignal2];
```



The error signal converges to zero after, say, 100 samples.

```
coefSeq2 =
  MultiplexSequence[a0Seq2, a1Seq2, a2Seq2, a3Seq2, a4Seq2, a5Seq2, a6Seq2, a7Seq2];
SequencePlot[coefSeq2, PlotJoined → True, StemPlot → False];
```



The values of the coefficients of the adaptive system converge to

```
coefValues2 = coefSeq2 // Last

{-0.00500044, -0.0200021, 0.100004,
 0.400004, 0.499999, 0.0800009, -0.0300005, -0.00200969}
```

and are close to the assumed parameters of the example unknown system

```
coefError2 = coefValues2 - parameterValues
```

$\{-4.43942\times10^{-7}, -2.09705\times10^{-6}, 4.01344\times10^{-6}, 3.93388\times10^{-6}, -6.8361\times10^{-7}, 8.60554\times10^{-7}, -5.14236\times10^{-7}, -9.69096\times10^{-6}\}$

## Benefits of *SchematicSolver*

*SchematicSolver* clearly visualizes the sophisticated adaptive algorithm.

*SchematicSolver* symbolically processes data samples keeping the system parameters as symbols. Consequently, we can identify the parameters of the unknown system with small number of samples.

In addition, *SchematicSolver* can process samples in a traditional numerical way, which requires large number of samples.

## 9.2. Automatic Gain Control

*Automatic Gain Control* (AGC) system scales the signal to a required level. AGC is typically handled in the analog domain to properly scale the signal for analog-to-digital (A/D) conversion because A/D converters have a limited dynamic range. If the signal strength is too high, the A/D conversion process will introduce a type of distortion known as clipping. If the signal strength is too low, the signal variations will toggle only a few bits at the A/D, and distortion will occur because of severe quantization.

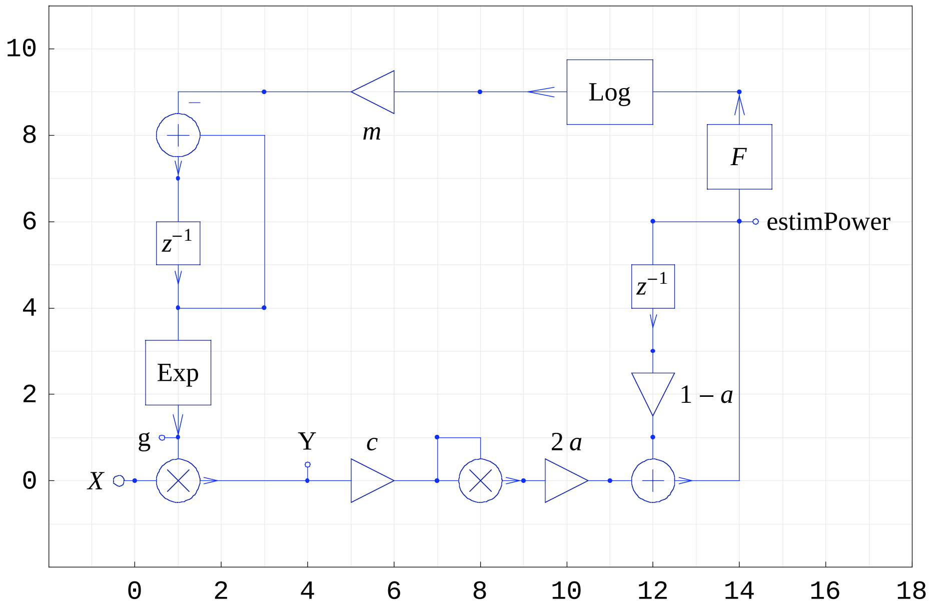
Many systems implement AGC in the digital domain for fine signal scaling. AGC is an adaptive system that operates over a wide dynamic range while maintaining the output signal at a nearly constant level. AGC is needed because some systems use amplitude thresholds to make decisions. These thresholds must remain constant over the entire dynamic range of input signals. This is achieved through use of an AGC system that adjusts the signal gain according to the actual input signal level.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

Here is an example AGC system:

```
agcSystem = {{"Input", {0, 0}, X},
   {"Output", {4, 0}, "Y", "", TextOffset → {0, -1}},
   {"Output", {1, 1}, "g", "", TextOffset → {1, 0}},
   {"Output", {14, 6}, "estimPower"},
   {"Multiplier", {{8, 9}, {3, 9}}, m},
   {"Multiplier", {{4, 0}, {7, 0}}, c},
   {"Multiplier", {{9, 0}, {11, 0}}, 2 a},
   {"Multiplier", {{12, 3}, {12, 1}}, 1 - a},
   {"Function", {{1, 4}, {1, 1}}, Exp, "", ElementSize → {2, 2}},
   {"Function", {{14, 9}, {8, 9}}, Log, "", ElementSize → {2, 1.5}},
   {"Function", {{14, 6}, {14, 9}}, F, "", ElementSize → {2, 2}},
   {"Adder", {{0, 8}, {1, 7}, {3, 4}, {3, 9}}, {0, 2, 1, -1}},
   {"Adder", {{11, 0}, {11, -1}, {14, 6}, {12, 1}}, {1, 0, 2, 1}},
   {"Modulator", {{0, 0}, {1, -1}, {4, 0}, {1, 1}}, {1, 0, 2, 1}},
   {"Modulator", {{7, 0}, {8, -1}, {9, 0}, {7, 1}}, {1, 0, 2, 1}},
   {"Line", {{1, 4}, {3, 4}}},
   {"Line", {{12, 6}, {14, 6}}},
   {"Line", {{7, 0}, {7, 1}}},
   {"Delay", {{1, 7}, {1, 4}}, 1},
   {"Delay", {{12, 6}, {12, 3}}, 1}};
 ShowSchematic[%, PlotRange → {{-2, 18}, {-2, 11}}];
```



The system summary, generated by `DiscreteSystemImplementationSummary`, points out the system input, initial state, parameter set, output, and final state.

```
DiscreteSystemImplementationSummary[agcSystem]

    Input: {Y[{0, 0}]}

    Initial state: {Y[{1, 4}], Y[{12, 3}]}

    Parameter: {a, c, F, m}

    Output: {Y[{4, 0}], Y[{1, 1}], Y[{12, 6}]}

    Final state: {Y[{1, 7}], Y[{12, 6}]}
```

The input signal `X` is multiplied by a variable gain signal `g`. `Y` is the output signal that we call the scaled signal. We have used the *Modulator* element, instead of the *Multiplier* element, to multiply `X` by `g` because `g` is not a constant. The scaled signal, `Y`, is multiplied by itself, by using another Modulator element, to obtain the signal power. The average power is computed by using a first order IIR filter that consists of two multipliers (with coefficients `2a` and `(1-a)`), an adder and a Delay element. The maximal level of the scaled signal can be controlled by another multiplier with coefficient `c`. If the average power is too small, the gain g is approximately equal to 1. If X increases, the average power also increases, g goes to zero, and Y is kept in the predefined range.

The AGC system should ignore small input signal levels. If the level of the input signal is too low (no input signal or noise) the gain should be set to 1. The corresponding nonlinear function for this purpose is defined as follows:

```
clippingFunction[x_] := x * (1 + Sign[x - 0.02]) / 2 +
   (1 - Sign[x - 0.02]) / 2;
```

Consider an example AGC system with the following parameters:

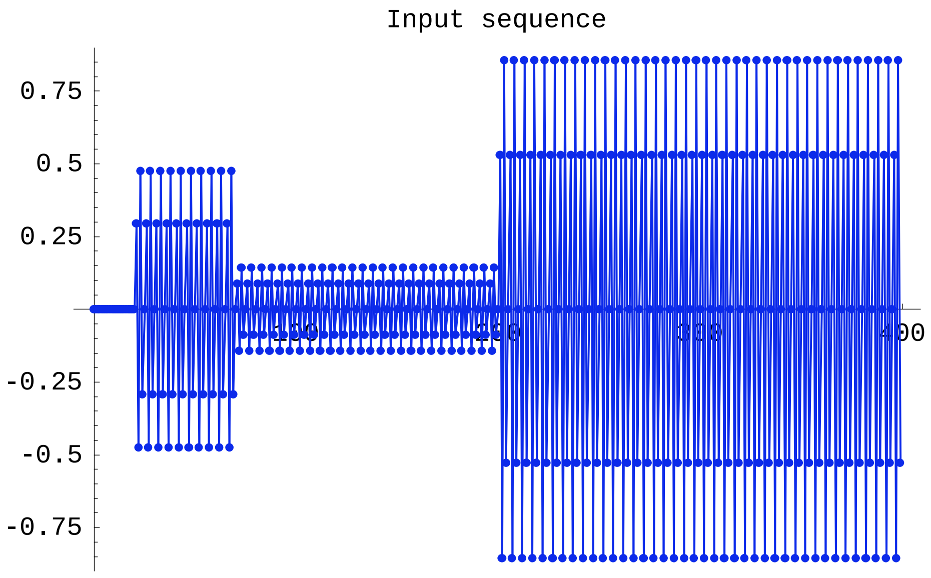
```
parameters = {
   a → estimatePowerParameter,
   c → 1 / nominalLevel,
   m → loopGainParameter,
   F → clippingFunction};

nominalLevel = 0.5;
loopGainParameter = 0.015;
estimatePowerParameter = 0.1;
```

Assume the following input signal:

```
numberOfSamples = 400;

inputSequence = nominalLevel * UnitSineSequence[numberOfSamples, 0.4] *
   (UnitStepSequence[numberOfSamples] -
     0.7 * UnitStepSequence[numberOfSamples, 70] +
     1.5 * UnitStepSequence[numberOfSamples, 200]) *
   UnitStepSequence[numberOfSamples, 20];
```

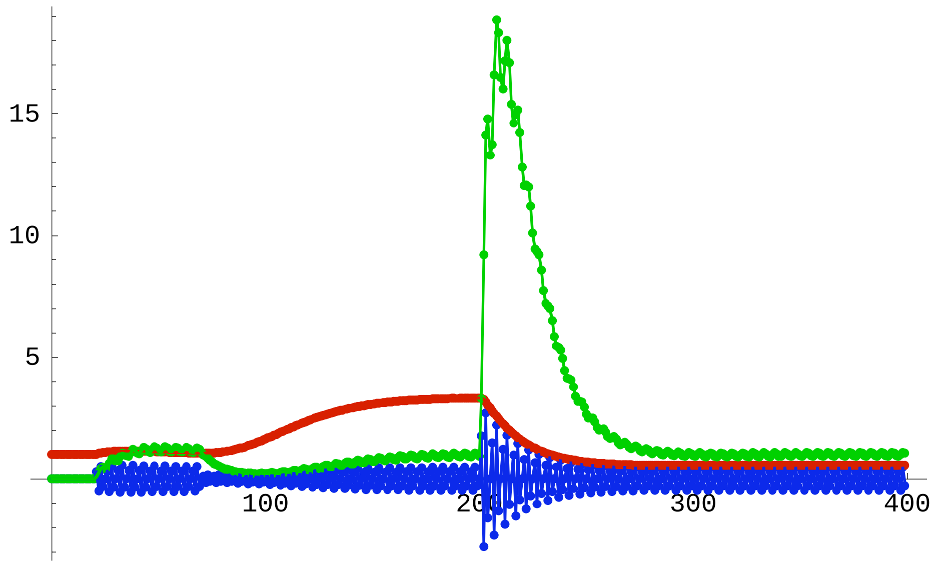
In this case, you can plot the discrete signals more clearly by setting the SequencePlot options to StemPlot→False and PlotJoined→True.

```
SetOptions[SequencePlot, StemPlot → False, PlotJoined → True];

SequencePlot[inputSequence, PlotLabel → "Input sequence"];
```



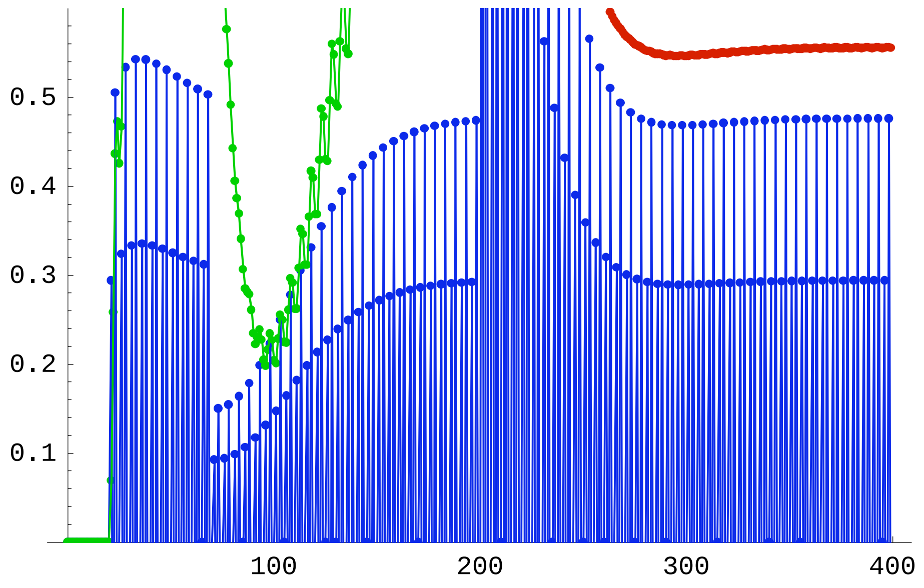
`DiscreteSystemSimulation` finds the system output: the scaled signal, the gain signal, and the estimated-power signal.

```
outputSequence = DiscreteSystemSimulation[agcSystem /. parameters, inputSequence];

SequencePlot[outputSequence];
```



The scaled signal is plotted in blue, the gain signal is plotted in red and the estimated average power is plotted in green.

```
SequencePlot[outputSequence, PlotRange → {0, 1.2 * nominalLevel}];
```



Note that the AGC system adjusts the gain and tries to scale the signal to the given level

```
nominalLevel
```

```
0.5
```

## 9.3. Quadrature Amplitude Modulation

### Introduction

*Quadrature Amplitude Modulation* (QAM) is a widely used method for transmitting digital data over bandpass channels. The simulation of a simplified and idealized QAM system follows.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

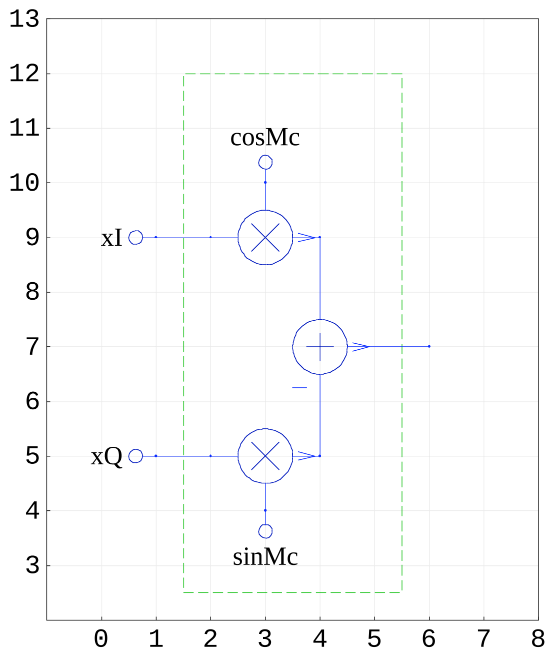
We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[InputNotebook[], ImageSize → {350, 250}];
SetOptions[ShowSchematic, FontSize → 10];
SetOptions[SequencePlot, StemPlot → False, PlotJoined → True];
```

### QAM Transmitter

The basic QAM transmitter can be represented by the schematic `modulatorQAM`

```
modulatorQAM = {{"Polyline", {{1.5, 2.5}, {5.5, 2.5}, {5.5, 12}, {1.5, 12}, {1.5, 2.5}},
    PlotStyle → {{RGBColor[0, 1, 0]}, {RGBColor[0, .7, 0]}}},
   {"Modulator", {{2, 5}, {3, 4}, {4, 5}, {3, 6}}, {1, 1, 2, 0}},
   {"Modulator", {{2, 9}, {3, 8}, {4, 9}, {3, 10}}, {1, 0, 2, 1}},
   {"Adder", {{3, 7}, {4, 5}, {6, 7}, {4, 9}}, {0, -1, 2, 1}, ""},
   {"Line", {{1, 9}, {2, 9}}}, {"Line", {{1, 5}, {2, 5}}}, {"Input", {1, 5}, xQ},
   {"Input", {1, 9}, xI}, {"Input", {3, 4}, sinMc, "", TextOffset → {0, 1}},
   {"Input", {3, 10}, cosMc, "", TextOffset → {0, -1}}};
ShowSchematic[%, PlotRange → {{-1, 8}, {2, 13}}];
```

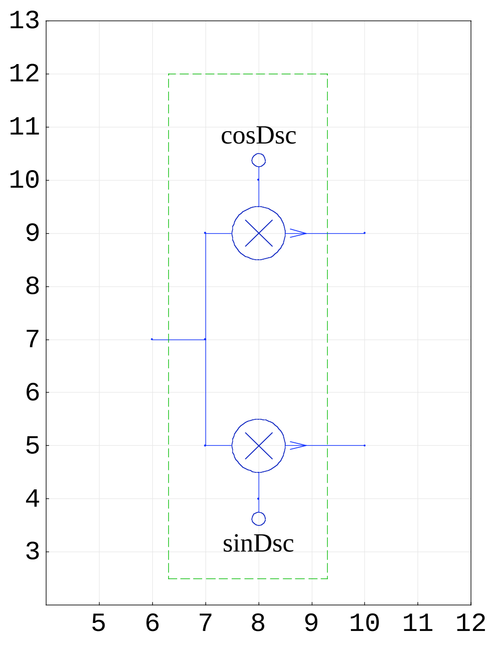


`xI` is the in-phase signal and `xQ` is the quadrature signal. `xI` and `xQ` are modulated by the quadrature carriers `cosMc` and `sinMc` and subtracted to form the transmitted QAM signal.

## QAM Receiver: Stage 1

The received signal is modulated by the locally generated quadrature carriers `cosDsc` and `sinDsc` to demodulate the signal to two baseband signals.

```
receiver1 =
  {{"Line", {{6, 7}, {7, 7}}}, {"Line", {{7, 7}, {7, 5}}}, {"Line", {{7, 7}, {7, 9}}},
   {"Polyline", {{6.3, 2.5}, {9.3, 2.5}, {9.3, 12}, {6.3, 12}, {6.3, 2.5}},
    PlotStyle → {{RGBColor[0, 1, 0]}, {RGBColor[0, .7, 0]}}},
   {"Modulator", {{7, 5}, {8, 4}, {10, 5}, {8, 6}}, {1, 1, 2, 0}},
   {"Modulator", {{7, 9}, {8, 8}, {10, 9}, {8, 10}}, {1, 0, 2, 1}},
   {"Input", {8, 4}, sinDsc, "", TextOffset → {0, 1}},
   {"Input", {8, 10}, cosDsc, "", TextOffset → {0, -1}}};
ShowSchematic[%, PlotRange → {{4, 12}, {2, 13}}];
```



## QAM Receiver: Stage 2

The in-phase and quadrature signals are detected at the output of the lowpass filter:

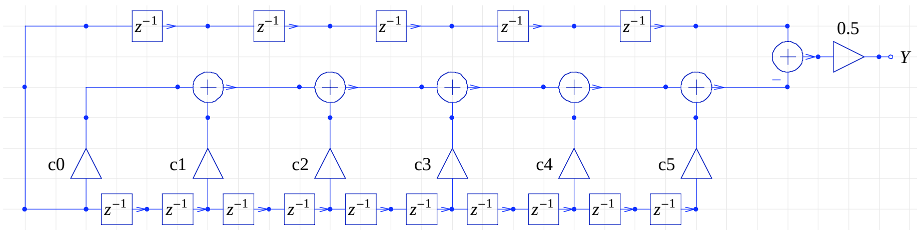
```
parameterSymbols = UnitSymbolicSequence[6, c, 0] // Flatten;
parameterValues = {-0.10744, 0.18315, -0.62626, -0.62626, 0.18315, -0.10744};
parameterSubstitution = parameterSymbols → parameterValues // Thread
```

```
{c0 → -0.10744, c1 → 0.18315, c2 → -0.62626, c3 → -0.62626, c4 → 0.18315, c5 → -0.10744}
```

```
{lowPassFilterSchematic1, inpCoords1, outCoords1} =
  HalfbandDirectFormFIRFilterSchematic[parameterSymbols, {2, 6}];
```

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];
```

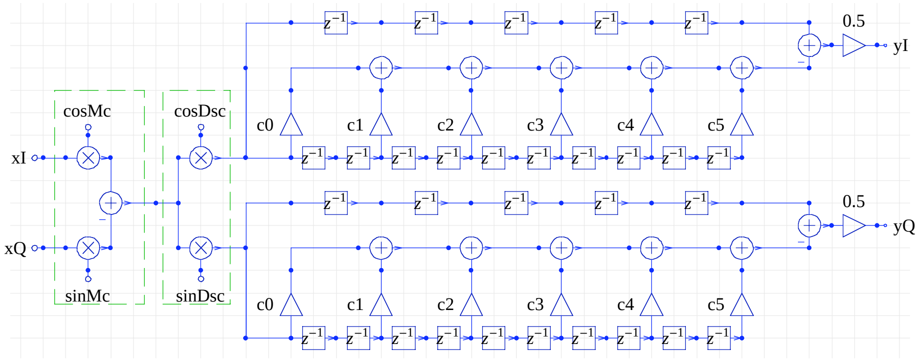
```
lowPassFilter = Join[
   lowPassFilterSchematic1,
   {{"Line", {inpCoords1[[1]], inpCoords1[[1]] + {0, 4}}},
    {"Output", outCoords1[[1]], Y}}];
ShowSchematic[%, FontSize → 7, Frame → False];
```



## QAM Transmitter/Receiver System

The simplified ideal QAM modulator and demodulator can be represented by the schematic `idealSystemQAM`

```
idealSystemQAM = Join[
   modulatorQAM,
   receiver1,
   TranslateSchematic[lowPassFilter /. Y → "yQ", {8, -5}],
   TranslateSchematic[lowPassFilter /. Y → "yI", {8, 3}]
  ];
ShowSchematic[%, FontSize → 7, Frame → False];
```



## Implementation of QAM

The system summary, generated by `DiscreteSystemImplementationSummary`, points out the system input, initial state, parameter set, output, and final state.

```
DiscreteSystemImplementationSummary[idealSystemQAM];
```

```
Input: {Y[{1, 5}], Y[{1, 9}], Y[{3, 4}], Y[{3, 10}], Y[{8, 4}], Y[{8, 10}]}

Initial state: {Y[{14, 1}], Y[{16, 1}], Y[{16, 7}], Y[{18, 1}], Y[{20, 1}],
  Y[{20, 7}], Y[{22, 1}], Y[{24, 1}], Y[{24, 7}], Y[{26, 1}], Y[{28, 1}], Y[{28, 7}],
  Y[{30, 1}], Y[{32, 1}], Y[{35, 7}], Y[{14, 9}], Y[{16, 9}], Y[{16, 15}],
  Y[{18, 9}], Y[{20, 9}], Y[{20, 15}], Y[{22, 9}], Y[{24, 9}], Y[{24, 15}],
  Y[{26, 9}], Y[{28, 9}], Y[{28, 15}], Y[{30, 9}], Y[{32, 9}], Y[{35, 15}]}

Parameter: {c0, c1, c2, c3, c4, c5}

Output: {Y[{38, 6}], Y[{38, 14}]}

Final state: {Y[{10, 1}], Y[{14, 1}], Y[{10, 1}], Y[{16, 1}], Y[{18, 1}],
  Y[{16, 7}], Y[{20, 1}], Y[{22, 1}], Y[{20, 7}], Y[{24, 1}], Y[{26, 1}], Y[{24, 7}],
  Y[{28, 1}], Y[{30, 1}], Y[{28, 7}], Y[{10, 9}], Y[{14, 9}], Y[{10, 9}],
  Y[{16, 9}], Y[{18, 9}], Y[{16, 15}], Y[{20, 9}], Y[{22, 9}], Y[{20, 15}],
  Y[{24, 9}], Y[{26, 9}], Y[{24, 15}], Y[{28, 9}], Y[{30, 9}], Y[{28, 15}]}
```

`DiscreteSystemImplementationEquations` is used to extract the system input, initial state, and parameter set:

```
eqns = DiscreteSystemImplementationEquations[idealSystemQAM];
numberOfInputs = Length[eqns[[1]]];
initialConditions = 0 * eqns[[2]]
systemParameters = eqns[[3]]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

{c0, c1, c2, c3, c4, c5}
```

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system:

```
DiscreteSystemImplementation[idealSystemQAM, "implementationQAM"];
```

```
    Implementation procedure name: implementationQAM

    Implementation procedure usage:

     {{Y38p6, Y38p14}, {Y10p1, Y14p1, Y10p1, Y16p1, Y18p1, Y16p7, Y20p1, Y22p1,
      Y20p7, Y24p1, Y26p1, Y24p7, Y28p1, Y30p1, Y28p7, Y10p9, Y14p9, Y10p9,
      Y16p9, Y18p9, Y16p15, Y20p9, Y22p9, Y20p15, Y24p9, Y26p9, Y24p15,
      Y28p9, Y30p9, Y28p15}} = implementationQAM[{Y1p5, Y1p9, Y3p4, Y3p10,
      Y8p4, Y8p10},{Y14p1, Y16p1, Y16p7, Y18p1, Y20p1, Y20p7, Y22p1, Y24p1,
      Y24p7, Y26p1, Y28p1, Y28p7, Y30p1, Y32p1, Y35p7, Y14p9, Y16p9, Y16p15,
      Y18p9, Y20p9, Y20p15, Y22p9, Y24p9, Y24p15, Y26p9, Y28p9, Y28p15,
      Y30p9, Y32p9, Y35p15},{c0, c1, c2, c3, c4, c5}] is the template
      for calling the procedure.  The general template is {outputSamples,
      finalConditions} = procedureName[inputSamples, initialConditions,
      systemParameters]. See also: DiscreteSystemImplementationProcessing
```

## Input Sequence

Consider 100 samples of the signal S of the modem V.34, for a symbol rate of 2400, the carrier frequency 1600 Hz, and the sampling rate of 8000 Hz:

```
numberOfSamples = 100;
Fc = 1600;
Fs = 8000;
Fy = 2400 / 2;
```

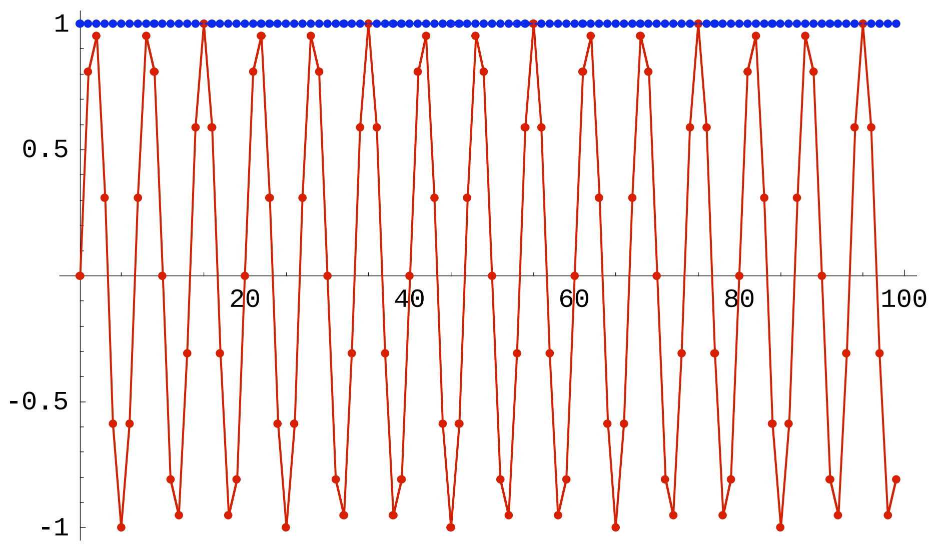
Digital frequency of the in-phase sinusoidal sequence is `Fy/Fs` (`Fy` is a half of the symbol rate), and the quadrature signal is a step sequence. The quadrature carriers `cosMc` and `sinMc` are generated as sinusoidal sequences with the phase shift of $\pi/2$.

```
xI = UnitSineSequence[numberOfSamples, Fy / Fs];
xQ = UnitStepSequence[numberOfSamples];
cosMc = UnitSineSequence[numberOfSamples, Fc / Fs, Pi / 2];
sinMc = UnitSineSequence[numberOfSamples, Fc / Fs];
```

The locally generated quadrature carriers `cosDsc` and `sinDsc` are generated as sinusoidal sequences with the phase shift of $(\pi+\pi/2)$.

```
cosDsc = UnitSineSequence[numberOfSamples, Fc / Fs, Pi + Pi / 2];
sinDsc = UnitSineSequence[numberOfSamples, Fc / Fs, Pi];
```

```
QIsequence = MultiplexSequence[xQ, xI];
SequencePlot[%];
```



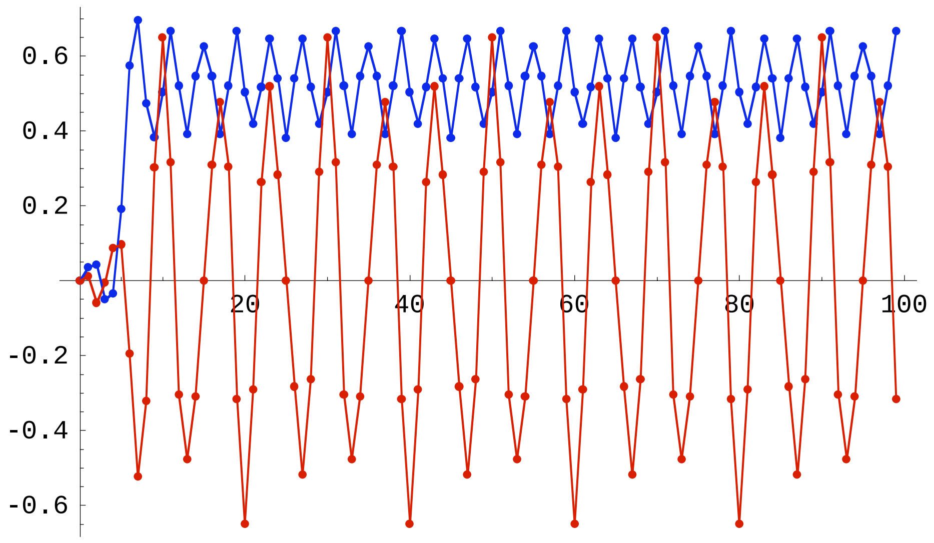
Here is the input sequence to the system:

```
inputSequence = MultiplexSequence[QIsequence, sinMc, cosMc, sinDsc, cosDsc];
```

## Processing

`DiscreteSystemImplementationProcessing` processes `inputSequence` with the function `implementationQAM`.

```
{outputSequence, finalConditions} = DiscreteSystemImplementationProcessing[
   inputSequence, initialConditions, systemParameters, implementationQAM];

SequencePlot[outputSequence /. parameterSubstitution];
```



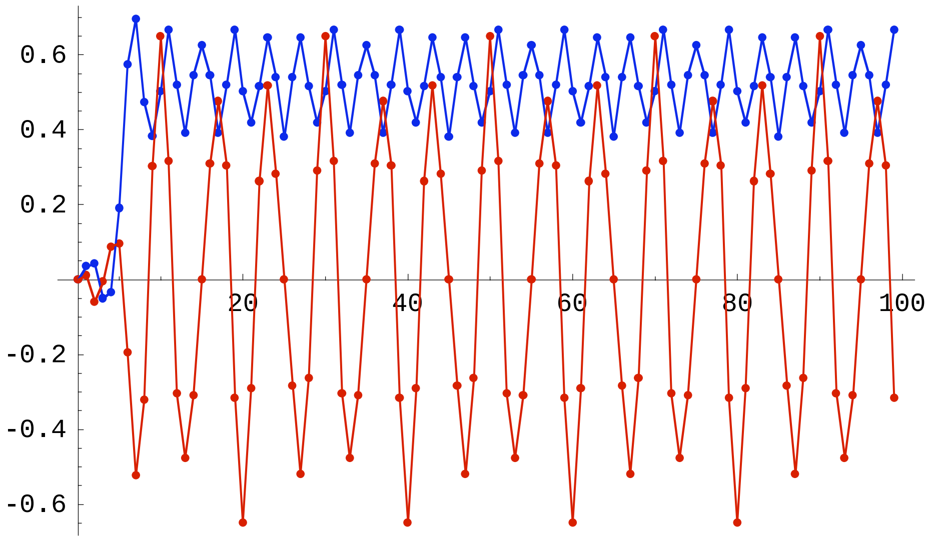
The plot demonstrates that we have detected both signals.

Note that the received in-phase sequence has smaller amplitude and that the received quadrature sequence has some distortion.

## Simulation

The same result is obtained with the *SchematicSolver*'s function `DiscreteSystemSimulation`:

```
DiscreteSystemSimulation[
   idealSystemQAM /. parameterSubstitution, inputSequence] // SequencePlot;
```



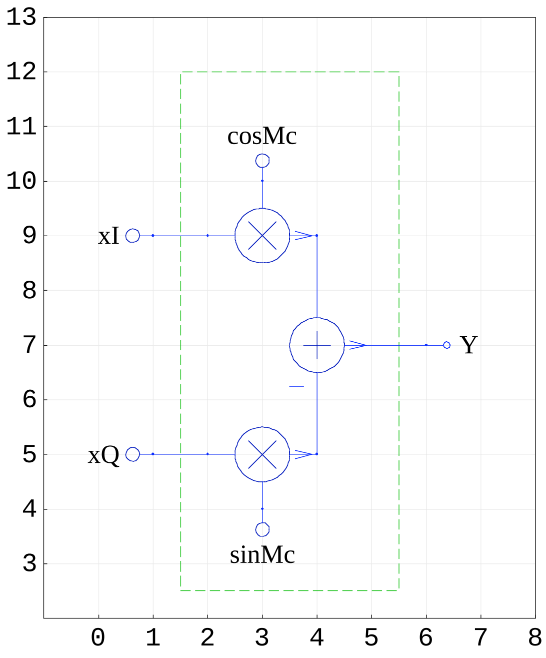
## QAM System Implementation by Subsystems

Consider a system that can be represented by subsystems, and assume that there are no feedback paths between the subsystems. We can simulate the system by implementing and simulating the subsystems individually.

In order to process signals with subsystems, the appropriate inputs and outputs should be added to the schematics of the QAM subsystems. An example follows.

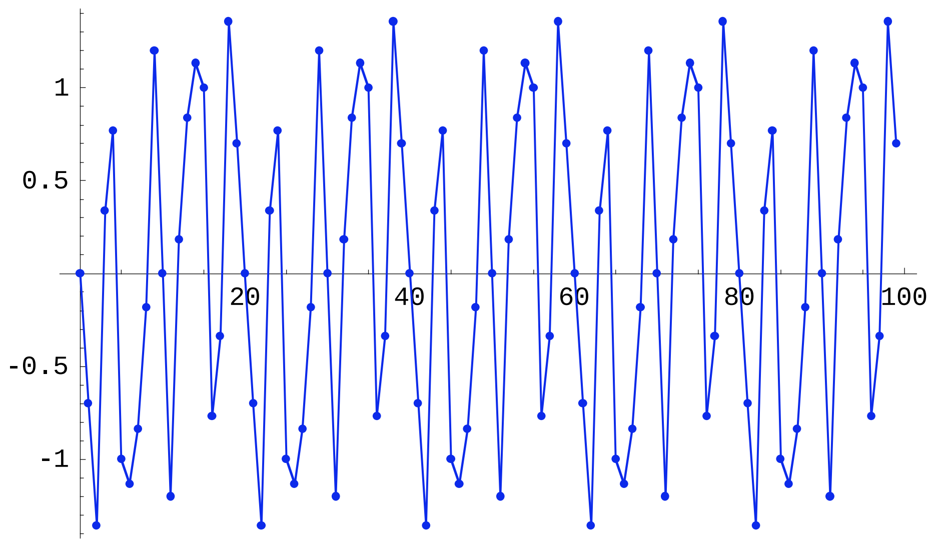
## QAM Transmitter (Modulator)

```
Clear[xI, xQ, cosMc, sinMc]

modulatorSubSystem = modulatorQAM ~ Join ~ {{"Output", {6, 7}, "Y"}};
ShowSchematic[%, PlotRange → {{-1, 8}, {2, 13}}];
```



The transmitted QAM signal, `outSeq1`, is computed by using the schematic of the modulator:

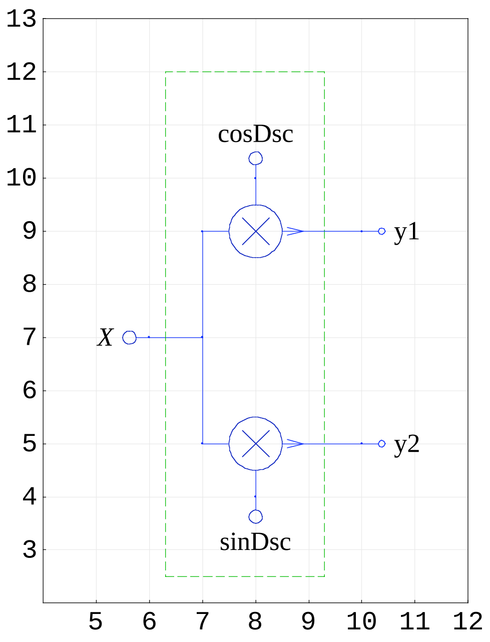
```
xI = UnitSineSequence[numberOfSamples, Fy / Fs];
xQ = UnitStepSequence[numberOfSamples];
cosMc = UnitSineSequence[numberOfSamples, Fc / Fs, Pi / 2];
sinMc = UnitSineSequence[numberOfSamples, Fc / Fs];
inpSeq1 = MultiplexSequence[xQ, xI, sinMc, cosMc];

outSeq1 = DiscreteSystemSimulation[modulatorSubSystem, inpSeq1];
SequencePlot[%];
```



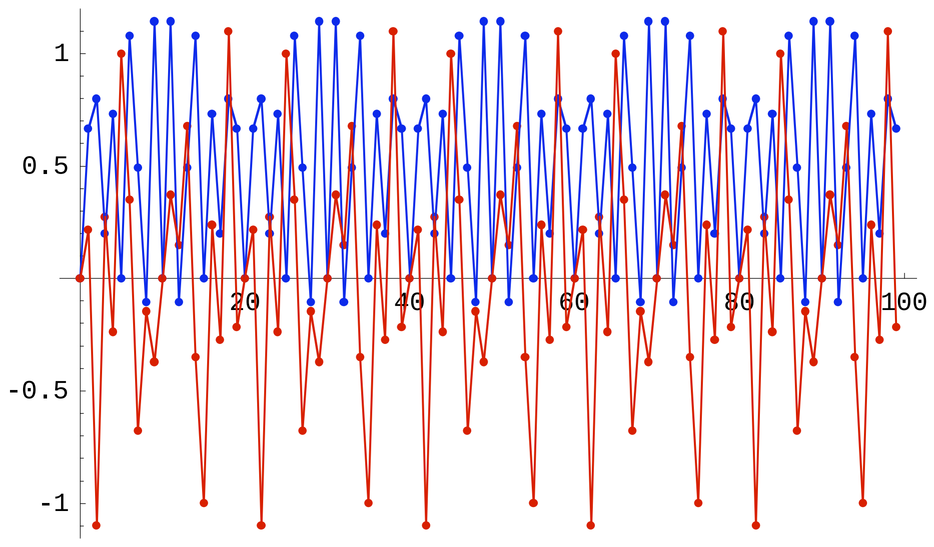
## QAM Receiver (Demodulator): Stage 1

The transmitted QAM signal is used to compute the two signals at the outputs of the first stage of the demodulator:

```
Clear[cosDsc, sinDsc]

receiverSubSystem = {{"Input", {6, 7}, X},
    {"Output", {10, 5}, y2}, {"Output", {10, 9}, y1}} ~
   Join ~ receiver1;
ShowSchematic[receiverSubSystem, PlotRange → {{4, 12}, {2, 13}}];
```



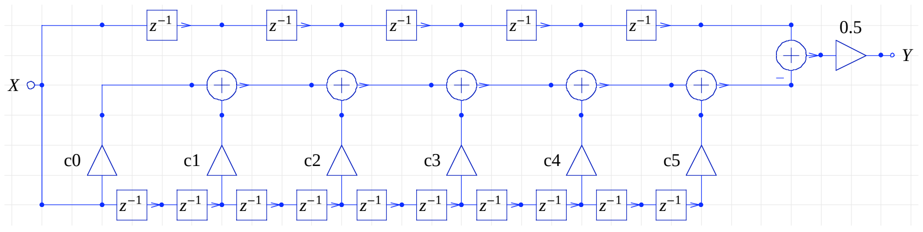
```
cosDsc = UnitSineSequence[numberOfSamples, Fc / Fs, Pi + Pi / 2];
sinDsc = UnitSineSequence[numberOfSamples, Fc / Fs, Pi];
inpSeq2 = MultiplexSequence[outSeq1, sinDsc, cosDsc];

outSeq2 = DiscreteSystemSimulation[receiverSubSystem, inpSeq2];
SequencePlot[%];
```



## QAM Receiver (Demodulator): Stage 2

The next stage of the demodulator is the lowpass filter. The filter may have feedback paths, but the system do not see the feedback path if we consider the filter as a subsystem.

```
filterSubSystem = {{"Input", {2, 10}, X}}~Join~lowPassFilter;
ShowSchematic[%, FontSize → 7, Frame → False];
```

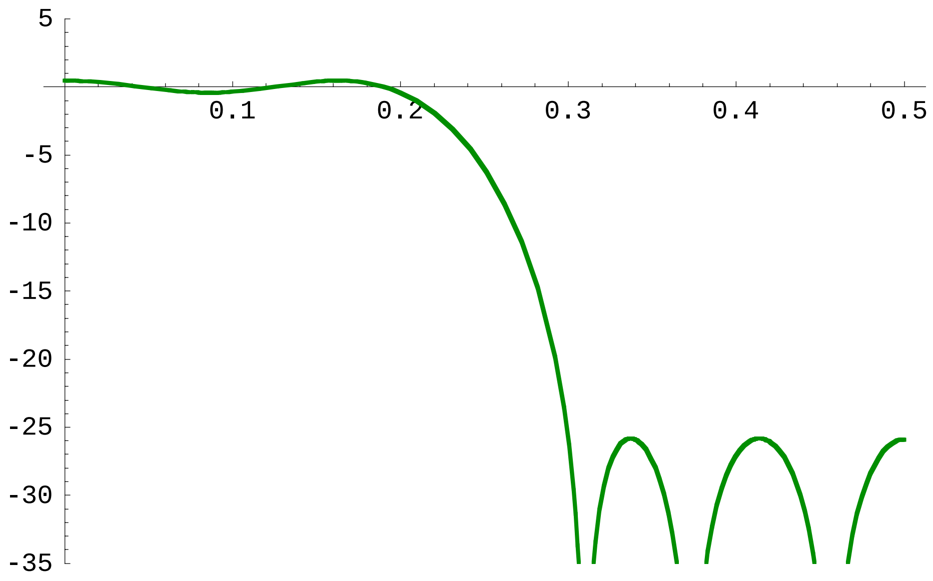


The lowpass filter is a linear system and we can compute its frequency response.

```
{tfMatrix, systemInp, systemOut} =
  DiscreteSystemTransferFunction[filterSubSystem /. parameterSubstitution];
tf = tfMatrix[[1, 1]] // Simplify
```

$$\frac{1}{z^{10}}\left(0.05372 - 0.091575\, z^2 + 0.31313\, z^4 + 0.5\, z^5 + 0.31313\, z^6 - 0.091575\, z^8 + 0.05372\, z^{10}\right)$$

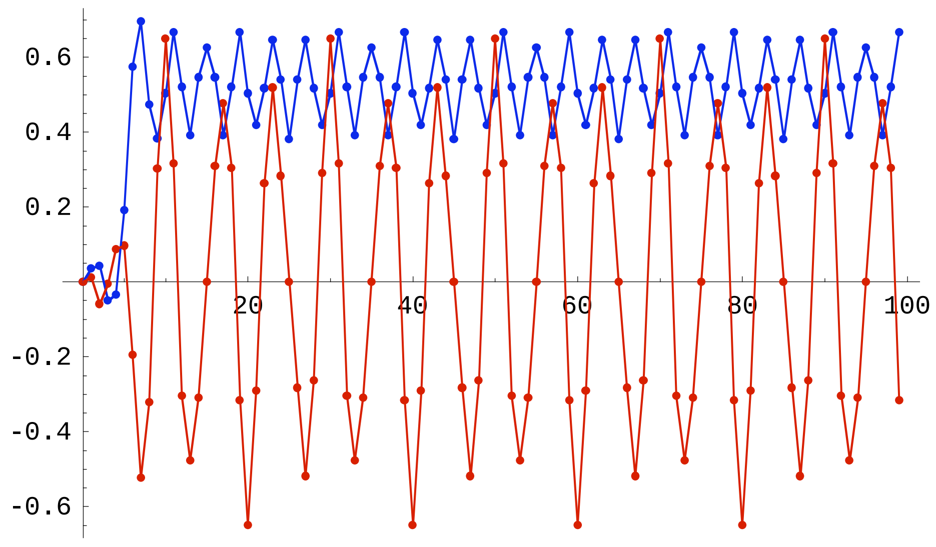
```
DiscreteSystemMagnitudeResponsePlot[tf, {0, 0.5}, PlotRange → {-35, 5}];
```



The lowpass filter attenuates the signals at higher frequencies and we can expect the smaller amplitude of sinusoidal sequences with higher digital frequency. The two sequences at the output of the first stage of the receiver are processed with two identical filters:

```
{inpSeq3, inpSeq4} = DemultiplexSequence[outSeq2];

outSeq3 = DiscreteSystemSimulation[filterSubSystem, inpSeq3];

outSeq4 = DiscreteSystemSimulation[filterSubSystem, inpSeq4];
```

```
MultiplexSequence[outSeq3, outSeq4] /. parameterSubstitution // SequencePlot;
```



The received quadrature and in-phase signals are delayed due to the Delay elements in the filter subsystem.

## 9.4. Square-Law Envelope Detector

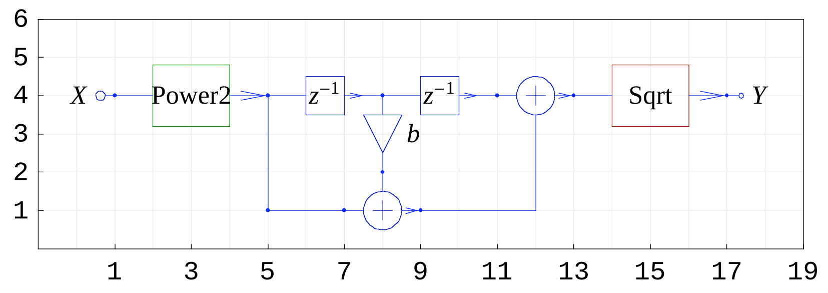
*Square-law envelope detector* is a system that demodulates an AM (Amplitude Modulation) signal.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

Here is an example square-law envelope detector system:

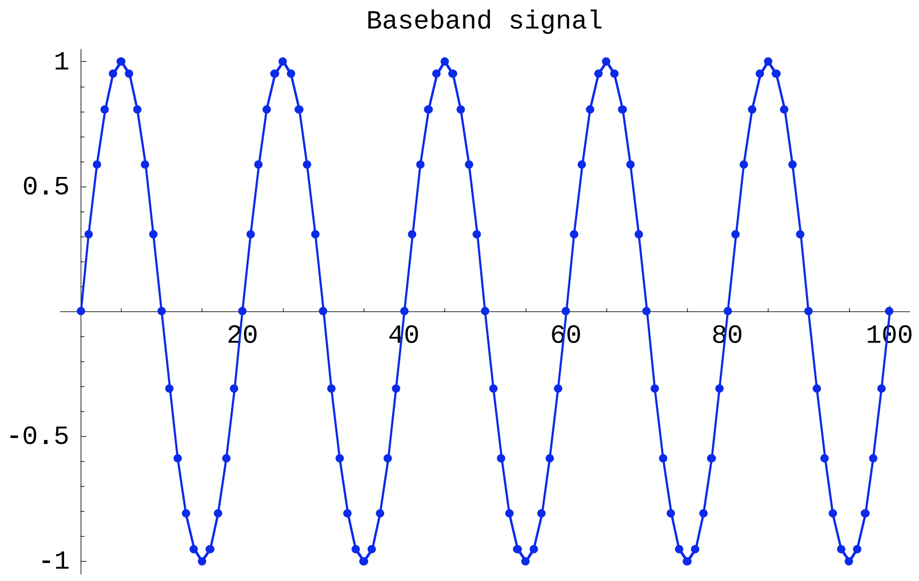
```
envelopeDetectorSystem = {
   {"Input", {1, 4}, X}, {"Output", {17, 4}, Y},
   {"Function", {{1, 4}, {5, 4}}, Power2, "", ElementSize → {2, 1.6},
    PlotStyle → {{RGBColor[0, 0.5, 0]}, {RGBColor[0, 0, 1]}}},
   {"Function", {{13, 4}, {17, 4}}, Sqrt, "", ElementSize → {2, 1.6},
    PlotStyle → {{RGBColor[0.5, 0, 0]}, {RGBColor[0, 0, 1]}}},
   {"Multiplier", {{8, 4}, {8, 2}}, b},
   {"Adder", {{7, 1}, {8, 0}, {9, 1}, {8, 2}}, {1, 0, 2, 1}},
   {"Delay", {{5, 4}, {8, 4}}, 1}, {"Delay", {{8, 4}, {11, 4}}, 1},
   {"Adder", {{11, 4}, {9, 1}, {13, 4}, {12, 5}}, {1, 1, 2, 0}},
   {"Line", {{5, 1}, {7, 1}}}, {"Line", {{5, 4}, {5, 1}}}};
 ShowSchematic[%, PlotRange → {{-1, 19}, {0, 6}}];
```



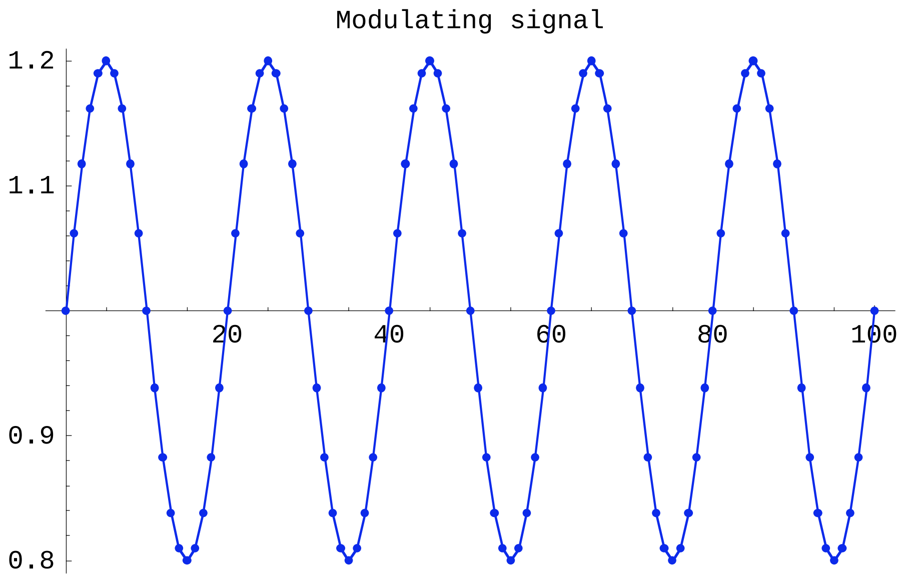
The input signal is an AM signal $x(n) = (1 + k\, m(n))\, c(n)$, where $c(n) = A \sin\left(2\, \pi\, n\, \frac{F_c}{F_s}\right)$ is called the carrier signal, $k$ is the amplitude sensitivity of the modulator, and $m(n) = \sin\left(2\, \pi\, n\, \frac{F_m}{F_s}\right)$ is the baseband signal.

```
numberSamples = 101;
A = 1 / Sqrt[2];
k = 0.2;
Fs = 8000;
Fc = 2000 / Fs;
Fm = 400 / Fs;
b = -2 * Cos[4 * Pi * Fc];

m = UnitSineSequence[numberSamples, Fm];
```

```
SequencePlot[m, PlotLabel → "Baseband signal"];
```



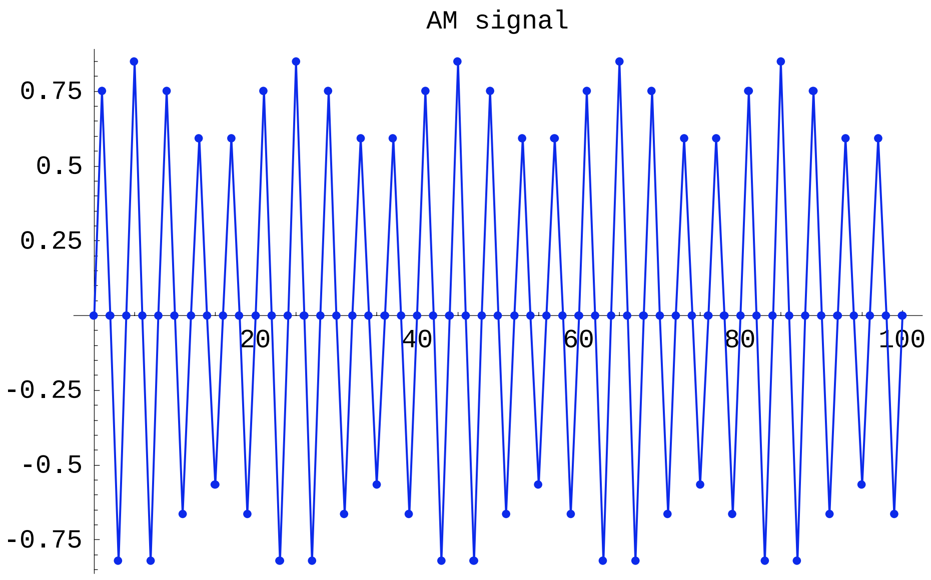
```
SequencePlot[1 + k * m, PlotLabel → "Modulating signal"];
```



```
c = A * UnitSineSequence[numberSamples, Fc];
```

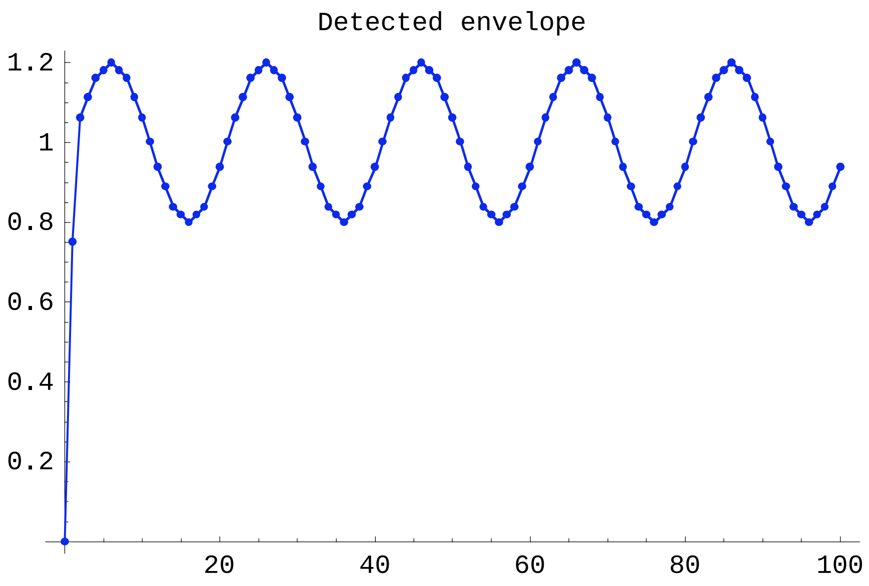
```
x = (1 + k * m) * c;
```

```
SequencePlot[x, PlotLabel → "AM signal"];
```



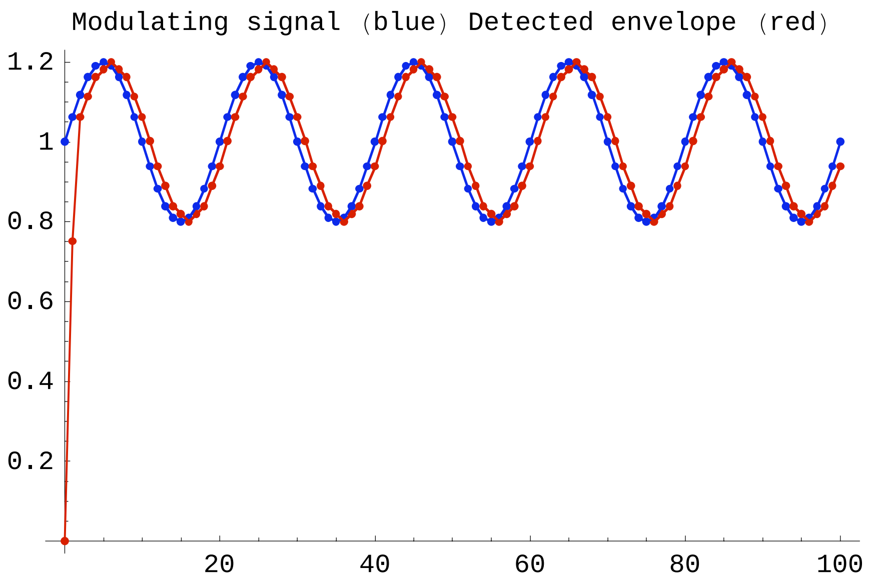
`DiscreteSystemSimulation` finds the envelope signal:

```
y = DiscreteSystemSimulation[envelopeDetectorSystem, x];
SequencePlot[y, PlotLabel → "Detected envelope"];
```



Make sure that the modulating signal and the envelope signal have the same shape:

```
SequencePlot[MultiplexSequence[1 + k * m, y],
  PlotLabel → "Modulating signal (blue) Detected envelope (red)",
  StemPlot → False, PlotJoined → True];
```



## 9.5. Nonlinear System with Hysteresis

### Introduction

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

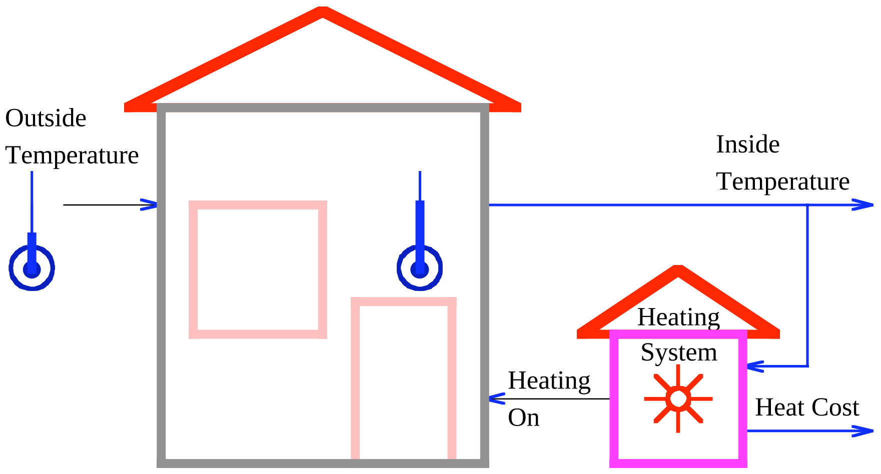
We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[InputNotebook[],
  ImageSize → {350, 250},
  ImageMargins → {{0, 0}, {0, 0}}];

SetOptions[ShowSchematic,
  ElementScale → 1,
  FontSize → Automatic,
  Frame → True,
  GridLines → Automatic,
  PlotRange → All];

SetOptions[DrawElement,
  ElementSize → {1, 1},
  PlotStyle → {
    {RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
    {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
  ShowArrowTail → True,
  ShowNodes → False,
  TextOffset → Automatic,
  TextStyle → {FontFamily → Times, FontSize → 10}];

SetOptions[SequencePlot, StemPlot → False, PlotJoined → True];
```

### Description of Heating System

Let us consider a simple model of the thermodynamics of a house:

```
ShowSchematic[SchematicSolverFigureImplementationExamplesHouseHeating,
 GridLines → None, Frame → False]
```



The out-door thermometer measures the outside temperature, `tempOut`, and the in-door thermometer measures the inside temperature, `tempIn`. The temperatures have been obtained by taking samples at discrete instants of time. We are concerned with uniform samples by sampling every $T$ units of time.

The next sample of the inside temperature is obtained by adding two terms to the current sample of the inside temperature `tempIn`:

```
(tempOut - tempIn) * coefHouse
```

and

```
heatOn * coefHeat
```

`coefHouse` denotes a parameter of the house, `coefHeat` denotes a parameter of the heating system, and `heatOn` can be 1 (heating system turned on) or 0 (heating system turned off). The next sample of the cumulative heating cost is computed by adding `unitCost` to the cumulative heating cost if the heating system is turned on.

## Two-Input Two-Output Linear Heating System

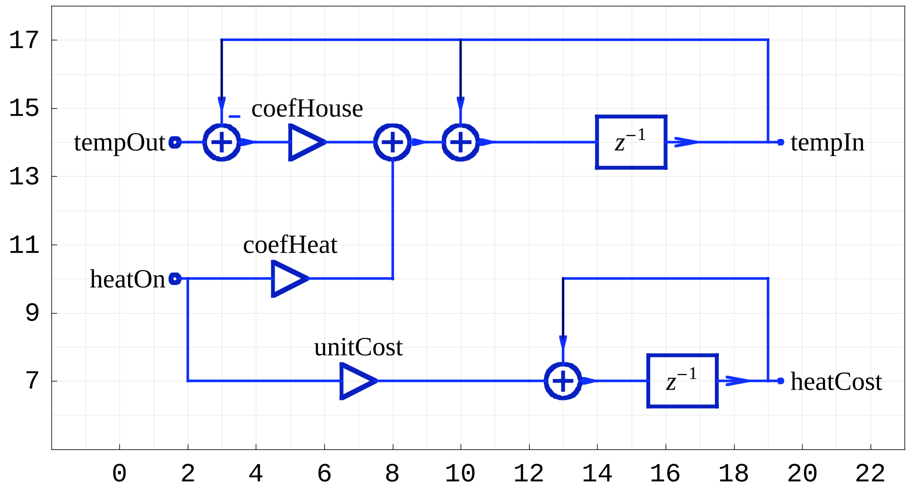
The schematic of the heating system can be drawn according to the system description. First, we use the Input element to describe the outside temperature `tempOut`. We employ an adder to perform the operation of subtraction of the outside temperature and the inside temperature. The difference of those two temperatures is multiplied by `coefHouse` using the Multiplier element. Another Input element is used for `heatOn` and a multiplier is used for multiplying it by `coefHeat`. An adder sums the two products. This value is added to the current inside temperature `tempIn` by using another adder. The computed value becomes the next sample of the inside temperature, therefore we use the Delay element. The output of the Delay element is fed back to the adders as the value of the current inside temperature.

The value of `heatOn` is multiplied by `unitCost` using the Multiplier element. This value is added to the current cumulative heating cost by using an adder. The computed cost becomes the next sample of the cumulative heating cost, therefore we use another Delay element. The output of the Delay element has the value of the current cumulative heating cost.

*SchematicSolver* describes a system as a list of elements; this list specifies what elements are in the system and how they are interconnected. A list describing a system will be referred to as the *system specification*. Each element in the system is also described as a list that states what the element is, to which other elements it is connected, and what its value is. A list describing an element will be referred to as the *element specification*.

Here is the schematic specification of the thermodynamics of a house:

```
linearHeatingSystem = {{"Input", {2, 14}, tempOut}, {"Input", {2, 10}, heatOn},
   {"Output", {19, 14}, tempIn},
   {"Output", {19, 7}, heatCost, "", TextOffset → {-1, 0}},
   {"Adder", {{2, 14}, {3, 13}, {4, 14}, {10, 17}}, {1, 0, 2, -1}},
   {"Adder", {{7, 14}, {8, 10}, {9, 14}, {8, 17}}, {1, 1, 2, 0}},
   {"Adder", {{9, 14}, {9, 10}, {11, 14}, {10, 17}}, {1, 0, 2, 1}},
   {"Adder", {{12, 7}, {13, 6}, {14, 7}, {13, 10}}, {1, 0, 2, 1}},
   {"Multiplier", {{4, 14}, {7, 14}}, coefHouse},
   {"Multiplier", {{2, 10}, {8, 10}}, coefHeat},
   {"Multiplier", {{2, 7}, {12, 7}}, unitCost},
   {"Delay", {{11, 14}, {19, 14}}, 1, "", ElementSize → {2, 3 / 2}},
   {"Delay", {{14, 7}, {19, 7}}, 1, "", ElementSize → {2, 3 / 2}},
   {"Line", {{10, 17}, {19, 17}, {19, 14}}},
   {"Line", {{2, 7}, {2, 10}}}, {"Line", {{13, 10}, {19, 10}, {19, 7}}},
   {"Arrow", {{10, 15}, {10, 17}}}, {"Arrow", {{3, 15}, {3, 17}}},
   {"Arrow", {{13, 8}, {13, 10}}}};
ShowSchematic[%, PlotRange → {{-2, 23}, {5, 18}}];
```



`DiscreteSystemImplementationSummary` points out the system input, initial state, parameter set, output, and final state:

```
DiscreteSystemImplementationSummary[linearHeatingSystem]
```

```
    Input: {Y[{2, 14}], Y[{2, 7}]}

    Initial state: {Y[{10, 17}], Y[{13, 10}]}

    Parameter: {coefHeat, coefHouse, unitCost}

    Output: {Y[{10, 17}], Y[{13, 10}]}

    Final state: {Y[{11, 14}], Y[{14, 7}]}
```

## Simulation: Step Response without Heating

Assume that the outside temperature abruptly changes from zero to 70

```
tempOutMax = 70;
```

Here are the first 200 samples of the outside temperature:

```
numberOfSamples = 200;
inpSeq1 = tempOutMax * UnitStepSequence[numberOfSamples];
```

Assume no heating

```
inpSeq2 = 0 * inpSeq1;
```

`MultiplexSequence` forms the input sequence to the system:

```
inputSequence = MultiplexSequence[inpSeq1, inpSeq2];
```

For given system parameters

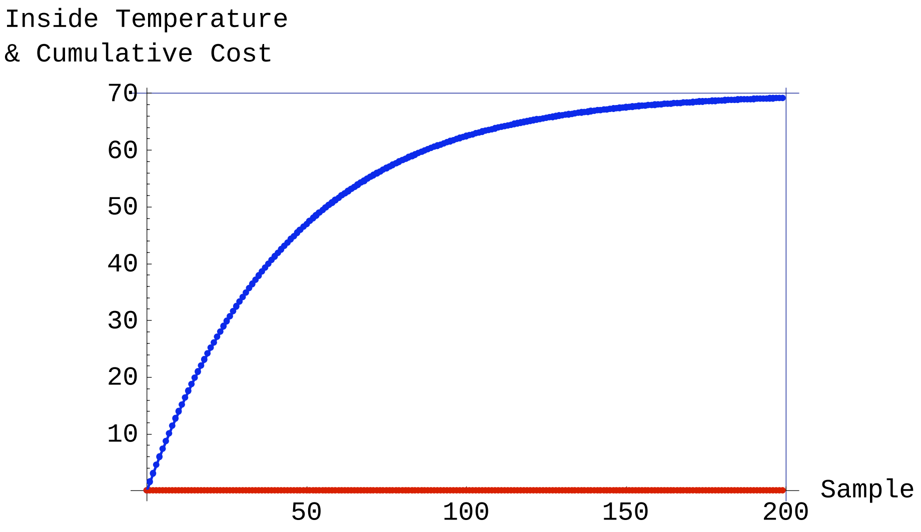
```
parameterValues = {coefHeat → 1.022, coefHouse → 0.022, unitCost → 0.025};
```

`DiscreteSystemSimulation` finds the system output, for zero initial conditions, as follows

```
outputSequence =
  DiscreteSystemSimulation[linearHeatingSystem /. parameterValues, inputSequence];
```

`SequencePlot` plots `outputSequence` that contains two signals, the inside temperature (blue) and the cumulative heating cost (red):

```
SequencePlot[outputSequence,
  AxesLabel → {"Sample", "Inside Temperature\n& Cumulative Cost"},
  GridLines → {{numberOfSamples}, {tempOutMax}}];
```



After 200 samples, the inside temperature is practically equal to the outside temperature. The cumulative heating cost is zero.

## Implementation of Linear Heating System

*Software implementation* is a sequence of statements that are executed on a general-purpose computer or on a dedicated hardware.

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system.

```
DiscreteSystemImplementation[linearHeatingSystem, "linearSystemImplementation"];
```

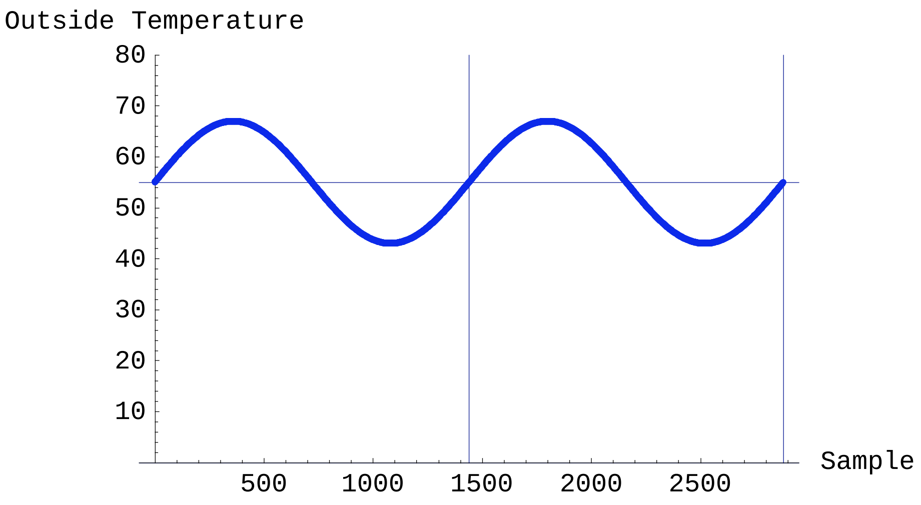
```
Implementation procedure name: linearSystemImplementation

Implementation procedure usage:

{{Y10p17, Y13p10}, {Y11p14, Y14p7}} = linearSystemImplementation[{Y2p14,
 Y2p7},{Y10p17, Y13p10},{coefHeat, coefHouse, unitCost}] is the template
 for calling the procedure.  The general template is {outputSamples,
 finalConditions} = procedureName[inputSamples, initialConditions,
 systemParameters]. See also: DiscreteSystemImplementationProcessing
```

The name of the implementation function is arbitrary and it is given as the second argument to `DiscreteSystemImplementation`. The name of the implementation function is `linearSystemImplementation` and it should be enclosed within double quotation marks.

## Generating Stimulus: Sine Temperature and Pulse-Train Heating

We can simulate daily temperature fluctuations applying a sinusoidal term with amplitude of 12 to a base temperature of 55. Assume that we sample temperature every minute, and that we observe an interval of two days.

```
baseTemperature = 55;
amplitudeTemperature = 12;
sinePeriod = 60 * 24;
numberOfSamples = 60 * 24 * 2;

inpSeq1 = baseTemperature +
   amplitudeTemperature * UnitSineSequence[numberOfSamples, 1 / sinePeriod];

SequencePlot[inpSeq1,
  AxesLabel → {"Sample", "Outside Temperature"}, GridLines →
   {{sinePeriod, numberOfSamples}, {0, baseTemperature}}, PlotRange → {0, 80}];
```



Assume that heating is periodically turned on for 1 minute, and then turned off for 2 minutes:

```
inpSeq2 = (1 - Sign[(0.1 +
        UnitSineSequence[numberOfSamples, (24 * 20) / sinePeriod])]) / 2;
```

`MultiplexSequence` forms the input sequence to the system:

```
inputSequence = MultiplexSequence[inpSeq1, inpSeq2];
```

## Processing with Linear System

Assume the following initial conditions (inside temperature of 60 and zero cumulative heating cost):

```
initialValues = {tempInitialCondition → 60, costInitialCondition → 0};
initialConditions = {tempInitialCondition, costInitialCondition} /. initialValues

{60, 0}
```

Assume the following values for the system parameters:

```
parameterValues = {coefHeat → 1.022, coefHouse → 0.022, unitCost → 0.025};
```

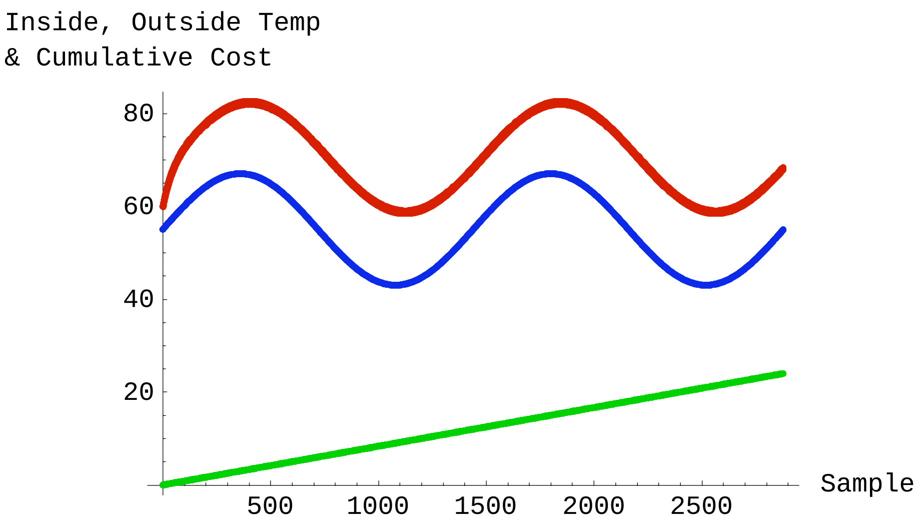
`DiscreteSystemImplementationEquations` finds the required order of parameters for a given schematic:

```
eqns = DiscreteSystemImplementationEquations[linearHeatingSystem];
systemParameters = eqns[[3]] /. parameterValues
```

```
{1.022, 0.022, 0.025}
```

`DiscreteSystemImplementationProcessing` processes `inputSequence` with the function `linearSystemImplementation`.

```
{outputSequence, finalConditions} =
  DiscreteSystemImplementationProcessing[inputSequence,
   initialConditions, systemParameters, linearSystemImplementation];

MultiplexSequence[inpSeq1, outputSequence];
SequencePlot[%,
  AxesLabel → {"Sample", "Inside, Outside Temp\n& Cumulative Cost"}];
```



Outside temperature is plotted in blue, inside temperature is plotted in red, and cumulative heating cost appears in green.

`DemultiplexSequence` extracts individual output sequences:

```
{insideTemperatureSeq, costSeq} = DemultiplexSequence[outputSequence];
```

Here is the final value of the cumulative heating cost:

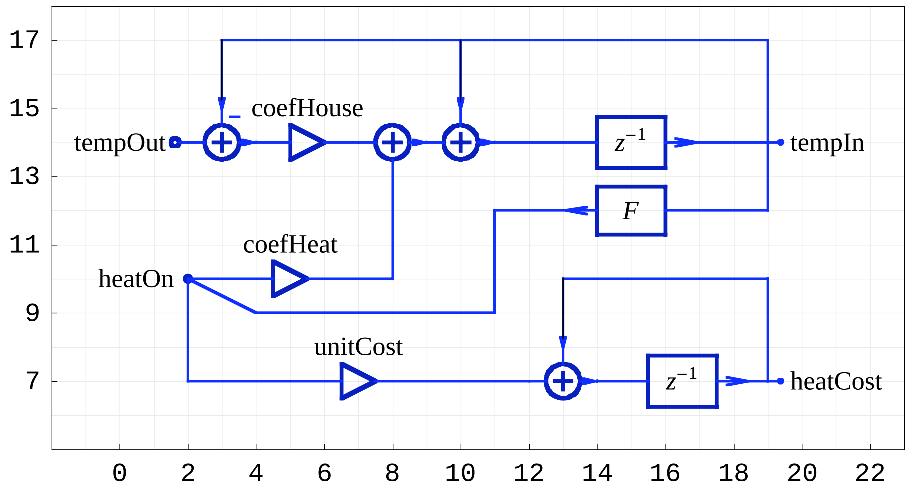
```
finalCost = Last[costSeq]
```

```
{23.975}
```

## Nonlinear Heating System

Here is a nonlinear model of the heating system:

```
nonlinearHeatingSystem = {{"Input", {2, 14}, tempOut},
   {"Node", {2, 10}, "heatOn ", "", TextOffset → {1, 0}},
   {"Output", {19, 14}, tempIn},
   {"Output", {19, 7}, heatCost, "", TextOffset → {-1, 0}},
   {"Adder", {{2, 14}, {3, 13}, {4, 14}, {10, 17}}, {1, 0, 2, -1}},
   {"Adder", {{7, 14}, {8, 10}, {9, 14}, {8, 17}}, {1, 1, 2, 0}},
   {"Adder", {{9, 14}, {9, 10}, {11, 14}, {10, 17}}, {1, 0, 2, 1}},
   {"Adder", {{12, 7}, {13, 6}, {14, 7}, {13, 10}}, {1, 0, 2, 1}},
   {"Multiplier", {{4, 14}, {7, 14}}, coefHouse},
   {"Multiplier", {{2, 10}, {8, 10}}, coefHeat},
   {"Multiplier", {{2, 7}, {12, 7}}, unitCost},
   {"Delay", {{11, 14}, {19, 14}}, 1, "", ElementSize → {2, 3 / 2}},
   {"Delay", {{14, 7}, {19, 7}}, 1, "", ElementSize → {2, 3 / 2}},
   {"Line", {{10, 17}, {19, 17}, {19, 14}}},
   {"Line", {{2, 7}, {2, 10}}}, {"Line", {{13, 10}, {19, 10}, {19, 7}}},
   {"Arrow", {{10, 15}, {10, 17}}}, {"Arrow", {{3, 15}, {3, 17}}},
   {"Arrow", {{13, 8}, {13, 10}}},
   {"Function", {{19, 12}, {11, 12}}, F, "", ElementSize → {2, 1.4}},
   {"Line", {{19, 12}, {19, 14}}},
   {"Line", {{2, 10}, {4, 9}, {11, 9}, {11, 12}}}};
ShowSchematic[%, PlotRange → {{-2, 23}, {5, 18}}];
```



`DiscreteSystemImplementationSummary` points out the nonlinear system input, initial state, parameter set, output, and final state:

```
DiscreteSystemImplementationSummary[nonlinearHeatingSystem]
```

```
    Input: {Y[{2, 14}]}

    Initial state: {Y[{19, 12}], Y[{13, 10}]}

    Parameter: {coefHeat, coefHouse, F, unitCost}

    Output: {Y[{19, 12}], Y[{13, 10}]}

    Final state: {Y[{11, 14}], Y[{14, 7}]}
```

## Implementation of Nonlinear Heating System

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the nonlinear system.

```
DiscreteSystemImplementation[
  nonlinearHeatingSystem, "nonlinearSystemImplementation"];

    Implementation procedure name: nonlinearSystemImplementation

    Implementation procedure usage:

     {{Y19p12, Y13p10}, {Y11p14, Y14p7}} = nonlinearSystemImplementation[{Y2p14},
      {Y19p12, Y13p10},{coefHeat, coefHouse, F, unitCost}] is the template
      for calling the procedure.  The general template is {outputSamples,
      finalConditions} = procedureName[inputSamples, initialConditions,
      systemParameters]. See also: DiscreteSystemImplementationProcessing
```

The name of the implementation function is arbitrary and it is given as the second argument to `DiscreteSystemImplementation`. The name of the implementation function is `nonlinearSystemImplementation` and it should be enclosed within double quotation marks.

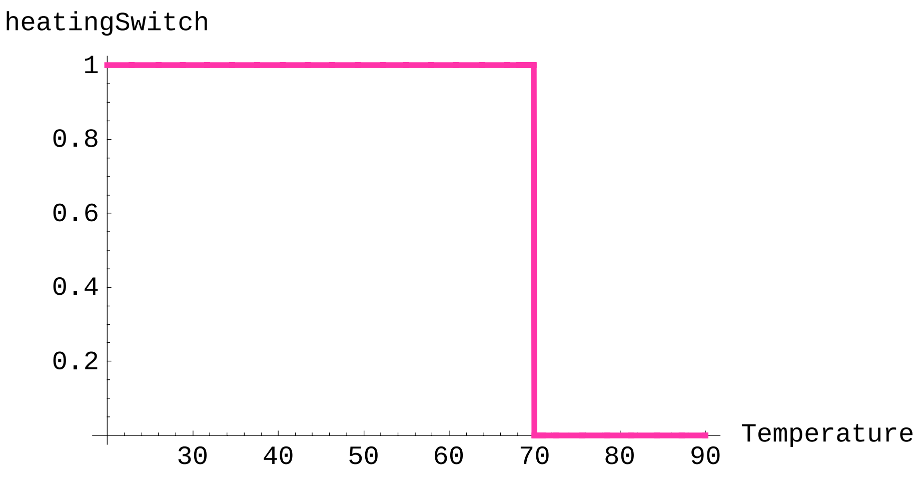
## User-Defined Nonlinear On-Off Function

Assume that we want to set up our thermostat at

```
tempThermostat = 70;
```

The nonlinear model of the heating system can be implemented with the on-off function

```
Clear[onOffFunction]
onOffFunction[t_] := Module[{heatingSwitch},
  If[t < tempThermostat, heatingSwitch = 1, heatingSwitch = 0];
  heatingSwitch]

Plot[onOffFunction[t], {t, 20, 90},
  AxesLabel → {"Temperature", "heatingSwitch"},
  PlotStyle → {Hue[0.9], Thickness[0.01]}];
```



## Processing with On-Off Function

Assume the following initial conditions (inside temperature of 60 and zero cumulative heating cost):

```
initialConditions

{60, 0}
```

The system parameters now contain the function name *F*:

```
eqns = DiscreteSystemImplementationEquations[nonlinearHeatingSystem];
systemParameters = eqns[[3]] /. parameterValues /. F → onOffFunction
```

```
{1.022, 0.022, onOffFunction, 0.025}
```

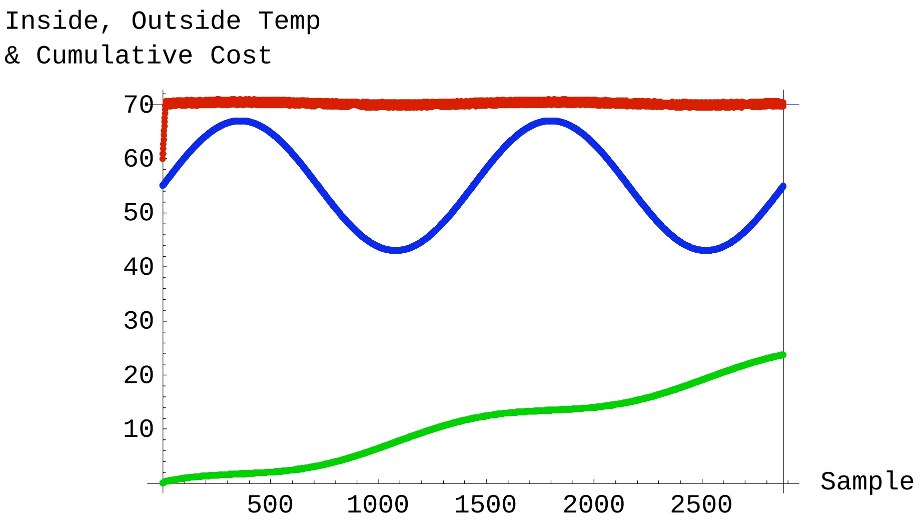
The system input is the previously generated stimulus `inpSeq1`.

```
inputSequence = inpSeq1;
```

`DiscreteSystemImplementationProcessing` processes `inputSequence` with the function `nonlinear-SystemImplementation`.

```
{outputSequence, finalConditions} =
  DiscreteSystemImplementationProcessing[inputSequence,
   initialConditions, systemParameters, nonlinearSystemImplementation];
```

```
MultiplexSequence[inpSeq1, outputSequence];
SequencePlot[%,
  AxesLabel → {"Sample", "Inside, Outside Temp\n& Cumulative Cost"},
  GridLines → {{numberOfSamples}, {tempThermostat}}];
```



Outside temperature is plotted in blue, inside temperature is plotted in red, and cumulative heating cost appears in green.

`DemultiplexSequence` extracts individual output sequences:

```
{insideTemperatureSeq, costSeq} = DemultiplexSequence[outputSequence];
```

Here is the final value of the cumulative heating cost:

```
finalCost = Last[costSeq]
```

```
{23.725}
```

## User-Defined Hysteresis Function

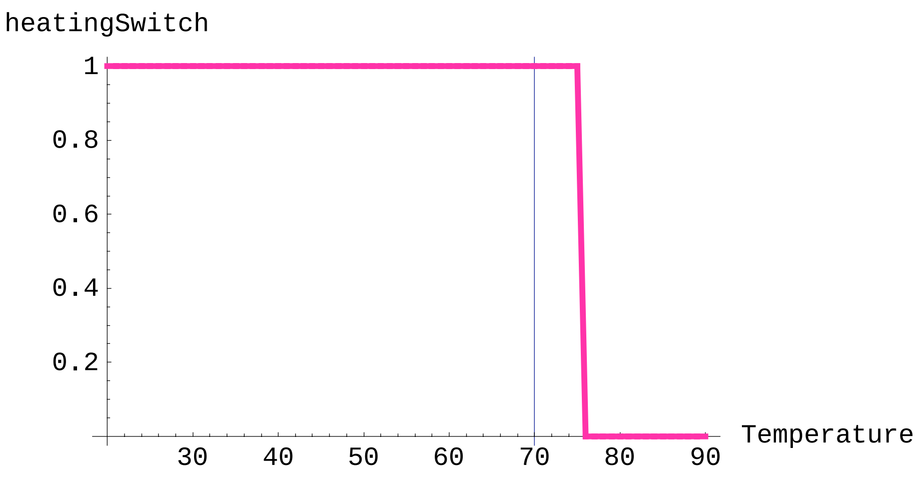
Consider a function that should keep the inside temperature within a predefined temperature range:

```
tempThermostat = 70;
tempDelta = 5;
tempHeatOn = tempThermostat - tempDelta;
tempHeatOff = tempThermostat + tempDelta;
heatingFlag = 0;

Clear[Fhysteresis];
Fhysteresis[t_] := Module[{heatingSwitch},
  If[heatingFlag == 0,
   If[t < tempHeatOn, heatingFlag = 1; heatingSwitch = 1, heatingSwitch = heatingFlag],
   If[t > tempHeatOff, heatingFlag = 0;
    heatingSwitch = 0, heatingSwitch = heatingFlag]];
  heatingSwitch]
```

The function $F_{\text{hysteresis}}$ uses the global variable `heatingFlag` and changes its value during processing.

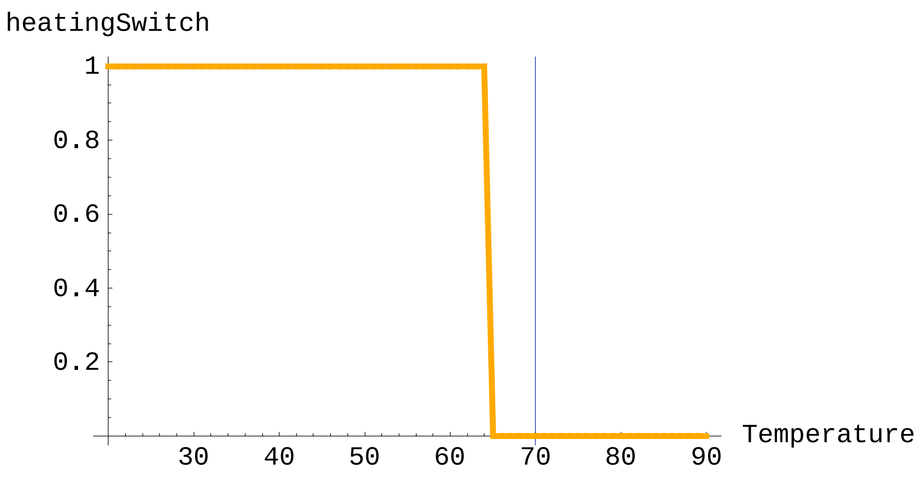
If temperature increases from 20 to 90, the heating switch is off after 75.

```
heatingFlag = 0;
ListPlot[Table[{t, Fhysteresis[t]}, {t, 20, 90}],
  AxesLabel → {"Temperature", "heatingSwitch"}, PlotJoined → True,
  PlotStyle → {Hue[0.9], Thickness[0.01]}, GridLines → {{tempThermostat}, {}}];
```



If temperature decreases from 90 to 20, the heating switch is on below 65.

```
heatingFlag = 0;
ListPlot[Table[{t, Fhysteresis[t]}, {t, 90, 20, -1}],
  AxesLabel → {"Temperature", "heatingSwitch"}, PlotJoined → True,
  PlotStyle → {Hue[0.1], Thickness[0.01]}, GridLines → {{tempThermostat}, {}}];
```



## Processing with Hysteresis Function

Assume the following initial conditions (inside temperature of 60 and zero cumulative heating cost):

```
initialConditions
```

```
{60, 0}
```

The system parameters now contain the function name *F*:

```
eqns = DiscreteSystemImplementationEquations[nonlinearHeatingSystem];
systemParameters = eqns[[3]] /. parameterValues /. F → Fhysteresis
```

```
{1.022, 0.022, Fhysteresis, 0.025}
```

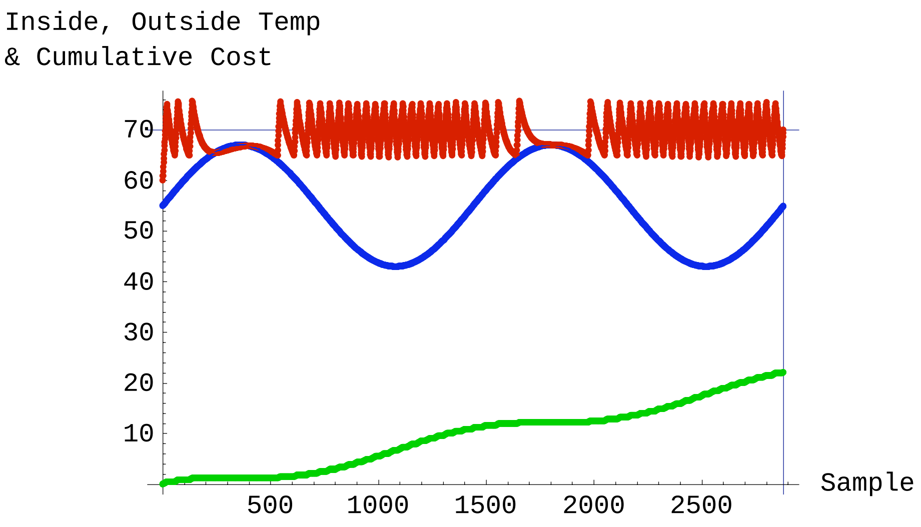
The system input is the previously generated stimulus `inpSeq1`.

```
inputSequence = inpSeq1;
```

`DiscreteSystemImplementationProcessing` processes `inputSequence` with the function `nonlinear-SystemImplementation`.

```
{outputSequence, finalConditions} =
  DiscreteSystemImplementationProcessing[inputSequence,
   initialConditions, systemParameters, nonlinearSystemImplementation];
```

```
MultiplexSequence[inpSeq1, outputSequence];
SequencePlot[%,
  AxesLabel → {"Sample", "Inside, Outside Temp\n& Cumulative Cost"},
  GridLines → {{numberOfSamples}, {tempThermostat}}];
```



Outside temperature is plotted in blue, inside temperature is plotted in red, and cumulative heating cost appears in green.

`DemultiplexSequence` extracts individual output sequences:

```
{insideTemperatureSeq, costSeq} = DemultiplexSequence[outputSequence];
```

Here is the final value of the cumulative heating cost:

```
finalCost = Last[costSeq]
```

```
{22.1}
```

## 9.6. High-Speed Recursive Filters

### High-Speed IIR-FIR Filters

With the increasing demand for high-speed and low-power applications such as mobile communications systems, it will become more important to use filters with maximal sampling frequency as well as low arithmetic complexity.

In this section, we shall consider special single-input two-output filters composed of identical allpass subfilters, that are interconnected via extra multipliers, referred to as *high-speed recursive filters*.

The major advantage of the high-speed filters over the corresponding conventional filters is that the required coefficient word length for the allpass subfilters is substantially reduced. For a given VLSI technology, it implies the substantially increased maximal sampling frequency at which the filter can operate. For interpolation and decimation, the arithmetic complexity can be reduced in comparison with the conventional filters.

The extra multipliers correspond to the coefficients of the *optimal finite impulse response* (FIR) filter, and allpass subfilters correspond to the half-band *infinite impulse response* (IIR) filter.

The high-speed filters can be used for interpolation and decimation with factor of two, and for *quadrature mirror filter* (QMF) banks with perfect magnitude reconstruction.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];

SetOptions[SequencePlot, StemPlot → True, PlotJoined → False];
```

### Draw Schematic of High-Speed Filter

Assume that we want to design a high-speed filter with the third-order half-band IIR filter and three extra multipliers.

Symbolic names of the coefficients are

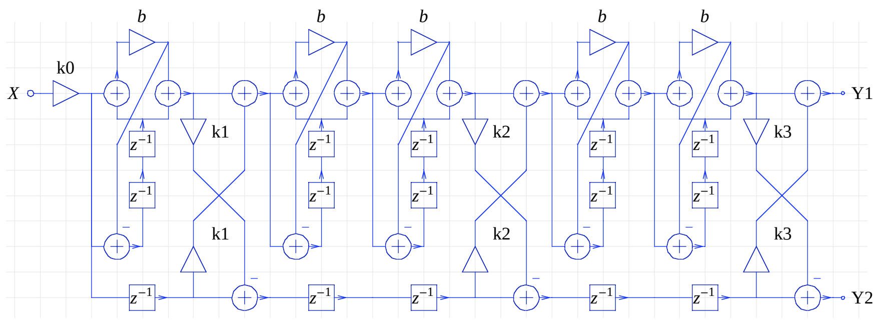
```
Clear[b, k0, k1, k2, k3]

parameterSymbols = {b, k0, k1, k2, k3}

{b, k0, k1, k2, k3}
```

where `b` is the coefficient of the IIR half-band filter, `k0` is the normalized coefficient and `k1`, `k2`, `k3` are the coefficients of the FIR filter.

`HighSpeedIIR3FIRHalfbandFilterSchematic` generates the schematic specification of the high-speed filter that is composed of allpass filters and extra multipliers specified by `parameterSymbols`.

```
{hsSchematic, inpCoordsHS, outCoordsHS} =
  HighSpeedIIR3FIRHalfbandFilterSchematic[parameterSymbols, {0, 0}];
```

```
hsSystem = Join[hsSchematic,
   {{"Input", inpCoordsHS[[1]], X},
    {"Output", outCoordsHS[[1]], "Y1"},
    {"Output", outCoordsHS[[2]], "Y2"}
   }];
ShowSchematic[hsSystem, Frame → False, FontSize → 7];
```



## Transfer Function of High-Speed Filter

`DiscreteSystemTransferFunction` computes the transfer function of the high-speed filter:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[hsSystem];
hsTF1 = tfMatrix[[1, 1]] // Together
```

$$
\begin{aligned}
&\frac{1}{z^5\,(b+z^2)^5}\\
&(b^5\,\mathrm{k0}\,\mathrm{k3} - b^4\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z + b^3\,\mathrm{k0}\,\mathrm{k2}\,z^2 + 5\,b^4\,\mathrm{k0}\,\mathrm{k3}\,z^2 - b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^2 - b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^3 -\\
&\quad 4\,b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^3 - b^5\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^3 - b^2\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^3 + b\,\mathrm{k0}\,\mathrm{k1}\,z^4 + 3\,b^2\,\mathrm{k0}\,\mathrm{k2}\,z^4 + 2\,b^4\,\mathrm{k0}\,\mathrm{k2}\,z^4 +\\
&\quad 10\,b^3\,\mathrm{k0}\,\mathrm{k3}\,z^4 - 3\,b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^4 - 2\,b^4\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^4 + \mathrm{k0}\,z^5 - 2\,b\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^5 -\\
&\quad 3\,b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^5 - 6\,b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^5 - 4\,b^4\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^5 - 2\,b\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^5 - 3\,b^3\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^5 +\\
&\quad \mathrm{k0}\,\mathrm{k1}\,z^6 + 4\,b^2\,\mathrm{k0}\,\mathrm{k1}\,z^6 + 3\,b\,\mathrm{k0}\,\mathrm{k2}\,z^6 + 6\,b^3\,\mathrm{k0}\,\mathrm{k2}\,z^6 + b^5\,\mathrm{k0}\,\mathrm{k2}\,z^6 + 10\,b^2\,\mathrm{k0}\,\mathrm{k3}\,z^6 -\\
&\quad 3\,b\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^6 - 6\,b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^6 - b^5\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^6 + 5\,b\,\mathrm{k0}\,z^7 - \mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^7 -\\
&\quad 6\,b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^7 - 3\,b^4\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^7 - 4\,b\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^7 - 6\,b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^7 - \mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^7 -\\
&\quad 6\,b^2\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^7 - 3\,b^4\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^7 + 4\,b\,\mathrm{k0}\,\mathrm{k1}\,z^8 + 6\,b^3\,\mathrm{k0}\,\mathrm{k1}\,z^8 + \mathrm{k0}\,\mathrm{k2}\,z^8 + 6\,b^2\,\mathrm{k0}\,\mathrm{k2}\,z^8 +\\
&\quad 3\,b^4\,\mathrm{k0}\,\mathrm{k2}\,z^8 + 5\,b\,\mathrm{k0}\,\mathrm{k3}\,z^8 - \mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^8 - 6\,b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^8 - 3\,b^4\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^8 +\\
&\quad 10\,b^2\,\mathrm{k0}\,z^9 - 3\,b\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^9 - 6\,b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^9 - b^5\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^9 - \mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^9 -\\
&\quad 4\,b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^9 - 3\,b\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^9 - 6\,b^3\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^9 - b^5\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^9 + 6\,b^2\,\mathrm{k0}\,\mathrm{k1}\,z^{10} +\\
&\quad 4\,b^4\,\mathrm{k0}\,\mathrm{k1}\,z^{10} + 2\,b\,\mathrm{k0}\,\mathrm{k2}\,z^{10} + 3\,b^3\,\mathrm{k0}\,\mathrm{k2}\,z^{10} + \mathrm{k0}\,\mathrm{k3}\,z^{10} - 2\,b\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^{10} -\\
&\quad 3\,b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^{10} + 10\,b^3\,\mathrm{k0}\,z^{11} - 3\,b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^{11} - 2\,b^4\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^{11} - b\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k3}\,z^{11} -\\
&\quad 3\,b^2\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^{11} - 2\,b^4\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^{11} + 4\,b^3\,\mathrm{k0}\,\mathrm{k1}\,z^{12} + b^5\,\mathrm{k0}\,\mathrm{k1}\,z^{12} + b^2\,\mathrm{k0}\,\mathrm{k2}\,z^{12} -\\
&\quad b^2\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,\mathrm{k3}\,z^{12} + 5\,b^4\,\mathrm{k0}\,z^{13} - b^3\,\mathrm{k0}\,\mathrm{k1}\,\mathrm{k2}\,z^{13} - b^3\,\mathrm{k0}\,\mathrm{k2}\,\mathrm{k3}\,z^{13} + b^4\,\mathrm{k0}\,\mathrm{k1}\,z^{14} + b^5\,\mathrm{k0}\,z^{15})
\end{aligned}
$$

```
hsTF2 = tfMatrix[[2, 1]] // Together;
```

The system has one input and two outputs, so its transfer function is a 2-by-1 matrix.

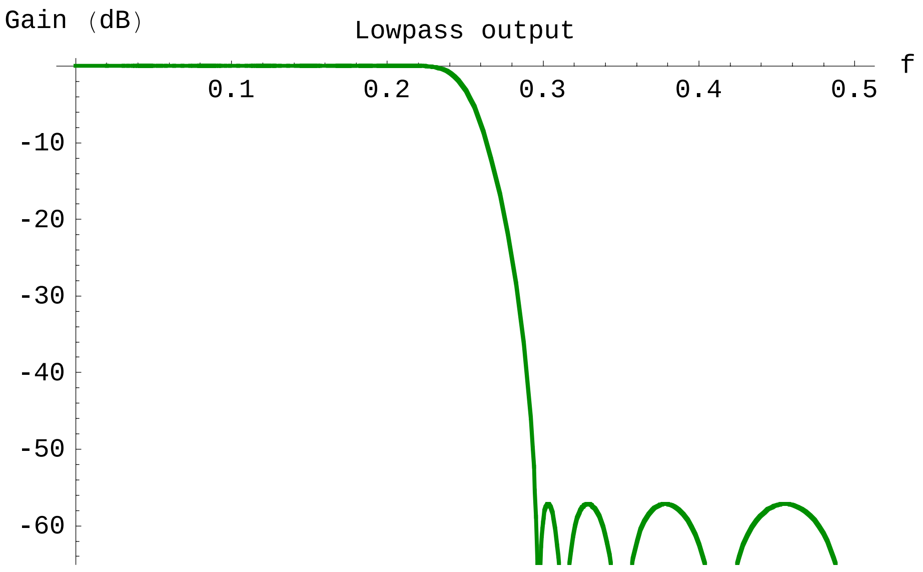
For the second-order allpass IIR filter, the maximal sampling frequency becomes critical due to the multiplier $b$ in the loop. Implementing the multiplier with a small number of adders can increase the maximal sampling frequency because the latency due to the multiplication is considerably larger than the latency due to an arithmetic operation of addition. When the value of the multiplier coefficient is a power of two, it can be implemented as a simple binary shift. By implementing the multipliers as binary shifters, that is without any hardware in FPGA (*Field Programmable Gate Array*) or VLSI (*Very Large Scale Integration*) implementations, we remove the multipliers from the critical loop and the maximal sampling frequency of the second-order IIR section reaches its maximal value. For example, the multiplier $b = 1/2 + 1/2^4$ can be implemented using an adder and two binary shifts. A digital filter whose multiplier coefficients are implemented with a small number of shift-and-add operations is called a *multiplierless filter*.

Let us define the numeric values of the system parameters:

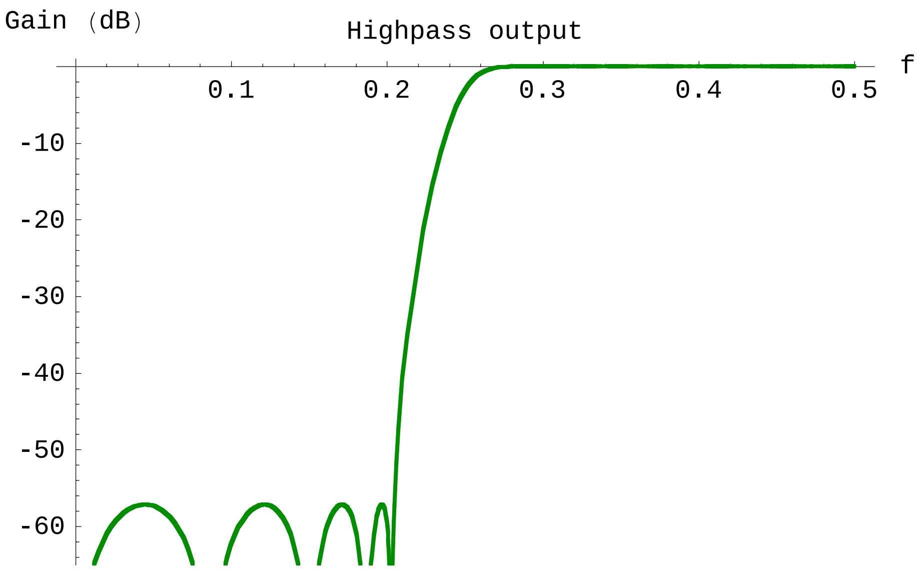
```
parameterSubstitution = {
   b → 1 / 2 + 1 / 16,
   k0 → 0.24000685,
   k1 → 2.37428361,
   k2 → -0.54068333,
   k3 →  0.10932683};
```

The magnitude characteristic, gain in decibels, of the high-speed filter is

```
DiscreteSystemMagnitudeResponsePlot[
 hsTF1 /. parameterSubstitution, {0, 0.5}, PlotRange → {-65, 1},
 AxesLabel → {"f", "Gain (dB)"}, PlotLabel → "Lowpass output"];
```



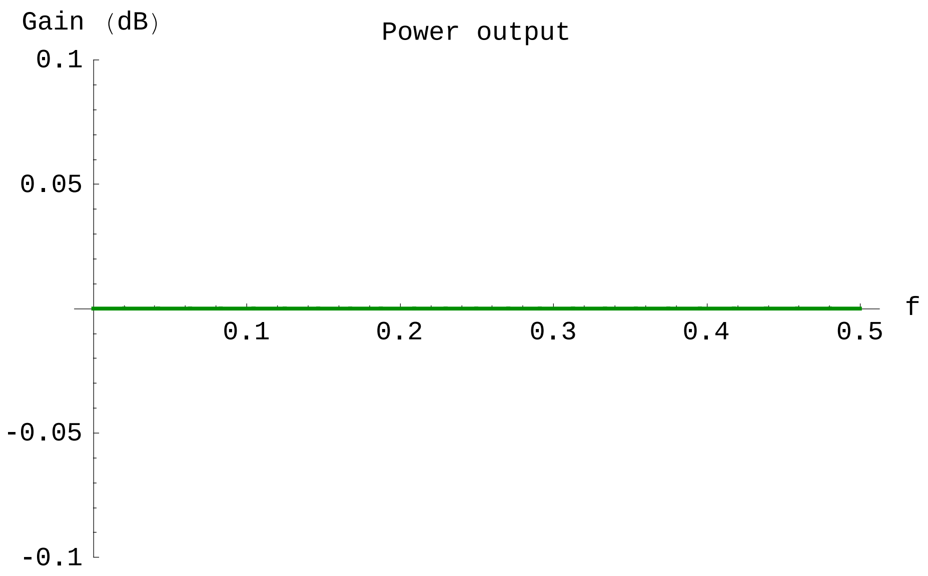
```
DiscreteSystemMagnitudeResponsePlot[
 hsTF2 /. parameterSubstitution, {0, 0.5}, PlotRange → {-65, 1},
 AxesLabel → {"f", "Gain (dB)"}, PlotLabel → "Highpass output"];
```



The sum of the squares of the magnitudes is constant:

```
tfSum2 = hsTF1 * (hsTF1 /. z → 1 / z) + hsTF2 * (hsTF2 /. z → 1 / z) // FullSimplify;
```

```
DiscreteSystemMagnitudeResponsePlot[
 tfSum2 /. parameterSubstitution, {0, 0.5}, PlotRange → {-0.1, 0.1},
 AxesLabel → {"f", "Gain (dB)"}, PlotLabel → "Power output"];
```



Here is the value of `tfSum2`:

```
tfSum2
```

$2\,\text{k0}^2\,(1+\text{k1}^2)\,(1+\text{k2}^2)\,(1+\text{k3}^2)$

Note that `tfSum2` does not depend on `z` or `b`. You can symbolically compute `k0` for which `tfSum2` equals 1.

```
k0Rule = Solve[tfSum2 == 1, k0]
```

$$\left\{\left\{\text{k0}\to -\frac{1}{\sqrt{2}\,\sqrt{1+\text{k1}^2}\,\sqrt{1+\text{k2}^2}\,\sqrt{1+\text{k3}^2}}\right\},\ \left\{\text{k0}\to \frac{1}{\sqrt{2}\,\sqrt{1+\text{k1}^2}\,\sqrt{1+\text{k2}^2}\,\sqrt{1+\text{k3}^2}}\right\}\right\}$$

A single-input two-output filter of the transfer function matrix

$$H(z) = \begin{pmatrix} H_1(z) \\ H_2(z) \end{pmatrix}$$

is said to be *power-complementary* if its transfer functions satisfy the following relation

$$|H_1(e^{i\omega})|^2 + |H_2(e^{i\omega})|^2 = 1$$

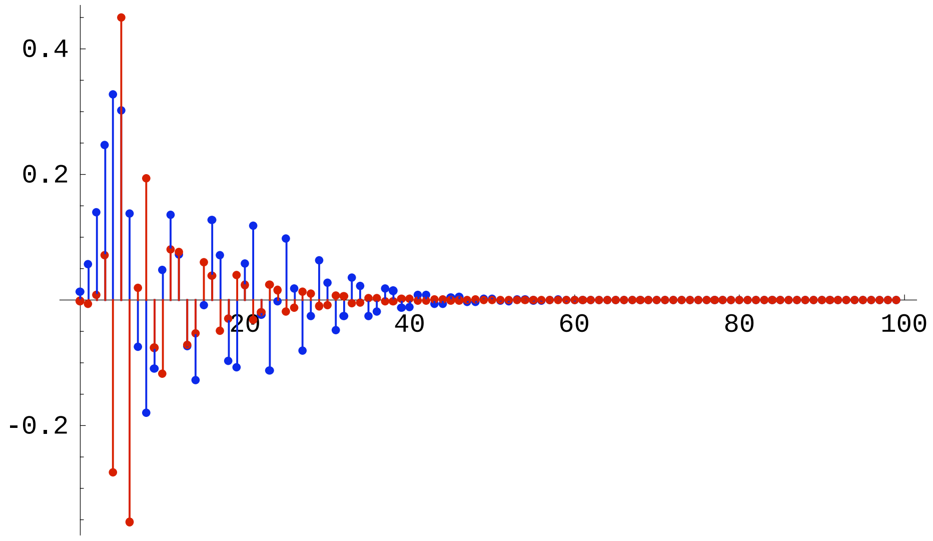
that is

$$H_1(z)\,H_1\left(\tfrac{1}{z}\right) + H_2(z)\,H_2\left(\tfrac{1}{z}\right) = 1$$

for $z = e^{i\omega}$.

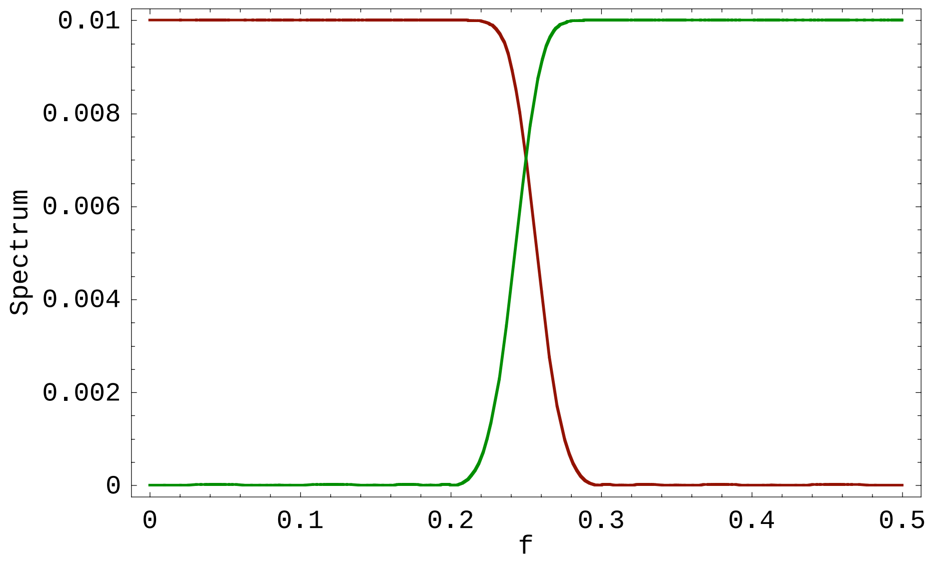
Here is the impulse response of the filter:

```
impulseResponseSeq = DiscreteSystemSimulation[
   hsSystem /. parameterSubstitution, UnitImpulseSequence[100]];
SequencePlot[impulseResponseSeq];
```



`SequenceFourierTransformMagnitudePlot` computes the spectrum of the impulse response:

```
SequenceFourierTransformMagnitudePlot[impulseResponseSeq, {0, 1/2}, Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



*SchematicSolver*'s functions help us symbolically derive an important relation between coefficients of the high-speed filter. A closed-form relation, such as

```
tfSum2
```

$2\,k0^2\,(1+k1^2)\,(1+k2^2)\,(1+k3^2)$

or

```
k0Rule
```

$$\left\{\left\{k0 \to -\frac{1}{\sqrt{2}\sqrt{1+k1^2}\sqrt{1+k2^2}\sqrt{1+k3^2}}\right\}, \left\{k0 \to \frac{1}{\sqrt{2}\sqrt{1+k1^2}\sqrt{1+k2^2}\sqrt{1+k3^2}}\right\}\right\}$$

cannot be identified with a purely numeric simulation of the filter.

# 10. Hilbert Transformer

## 10.1. Discrete Analytic Signal

### Real, Complex, and Analytic Signals

All naturally generated signals are *real-valued* and are referred to as *real signals*. In some applications, it is desirable to generate signals that are *complex-valued*, also called *complex signals*.

Consider a complex discrete signal $x(n)$ formed from two real signals $x_{\text{inphase}}(n)$ and $x_{\text{quad}}(n)$:

$x(n) = x_{\text{inphase}}(n) + i\, x_{\text{quad}}(n)$

The signal $x_{\text{inphase}}(n)$ is called the *in-phase component* and the signal $x_{\text{quad}}(n)$ is called the *quadrature component*.

*Discrete analytic signal* is a complex signal that has zero-valued spectrum for digital frequencies $-\frac{1}{2} < f < 0$.

Assume that we can find the Fourier Transform, $X(e^{i\omega})$, $X_{\text{inphase}}(e^{i\omega})$, and $X_{\text{quad}}(e^{i\omega})$, of signals $x(n)$, $x_{\text{inphase}}(n)$, and $x_{\text{quad}}(n)$, respectively. It follows:

$X(e^{i\omega}) = X_{\text{inphase}}(e^{i\omega}) + i\, X_{\text{quad}}(e^{i\omega}), \quad \omega = 2\pi f$

Fourier Transform of the analytic signal, $X(e^{i\omega})$ , is determined by the transform of the in-phase signal, $X_{\text{inphase}}(e^{i\omega})$:

$X(e^{i\omega}) = 2\,\text{Re}(X_{\text{inphase}}(e^{i\omega}))$.

A more detailed introduction to this topic can be found in the book:

Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, New York, NY, 1998.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

### Spectrum of Analytic Signal

Assume that we want to process a signal that is composed of two complex exponential signals.

```
numberOfSamples = 100;

amplitude1 = 1;
frequency1 = 0.05;
phase1 = Pi / 2;
expSeq1 =
  amplitude1 * ⅇ^(ⅈ phase1) * UnitExponentialSequence[numberOfSamples, I 2 π frequency1, E];
```

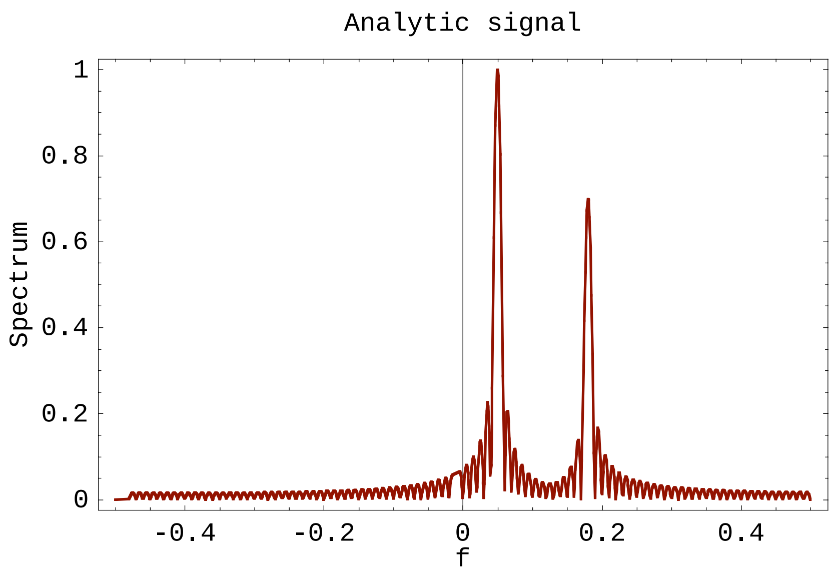
```
amplitude2 = 0.7;
frequency2 = 0.18;
phase2 = Pi / 3;
expSeq2 =
  amplitude2 * ⅇ^(ⅈ phase2) * UnitExponentialSequence[numberOfSamples, I 2 π frequency2, E];
```

Here is the composite signal:

```
compositeSeq = expSeq1 + expSeq2;
```

The spectrum of the composite signal can be computed using `SequenceFourierTransform` and illustrated using `SequenceFourierTransformMagnitudePlot`:

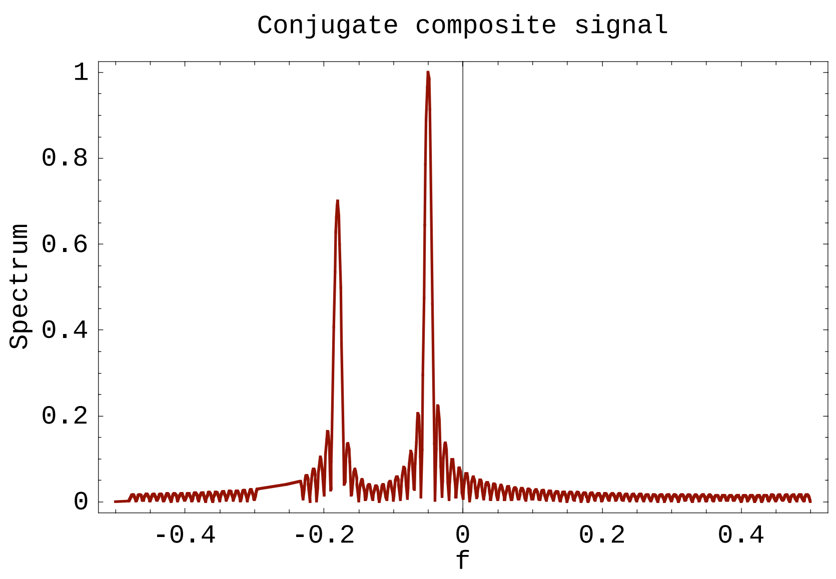
```
SequenceFourierTransformMagnitudePlot[compositeSeq,
 AxesLabel → "Analytic signal", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



The composite signal has practically zero-valued spectrum for $-\frac{1}{2} < f < 0$, therefore it is an analytical signal.

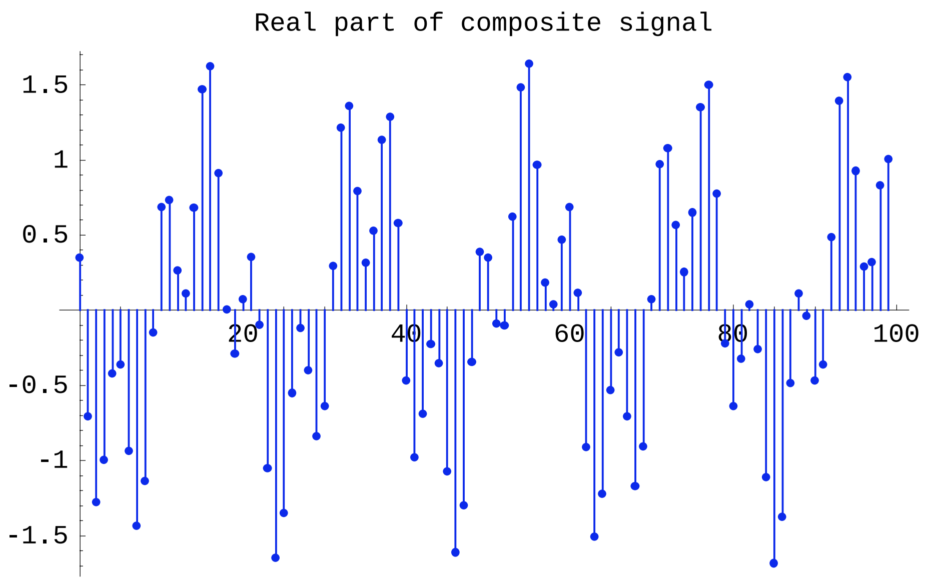
The conjugate composite signal has zero-valued spectrum for $0 < f < \frac{1}{2}$:

```
SequenceFourierTransformMagnitudePlot[Conjugate[compositeSeq],
 AxesLabel → "Conjugate composite signal", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



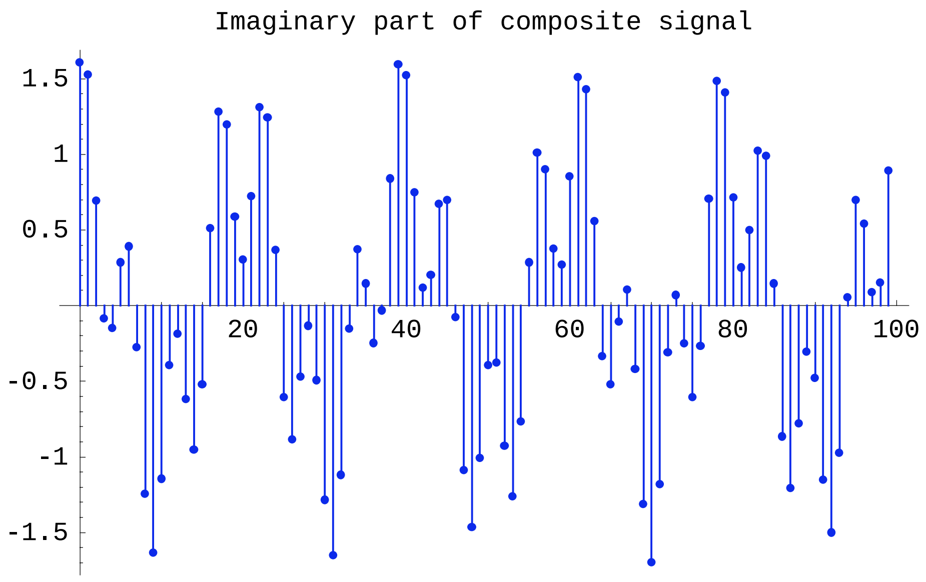
Here is the plot of the real part of the composite signal:

```
SequencePlot[Re[compositeSeq], PlotLabel → "Real part of composite signal"];
```



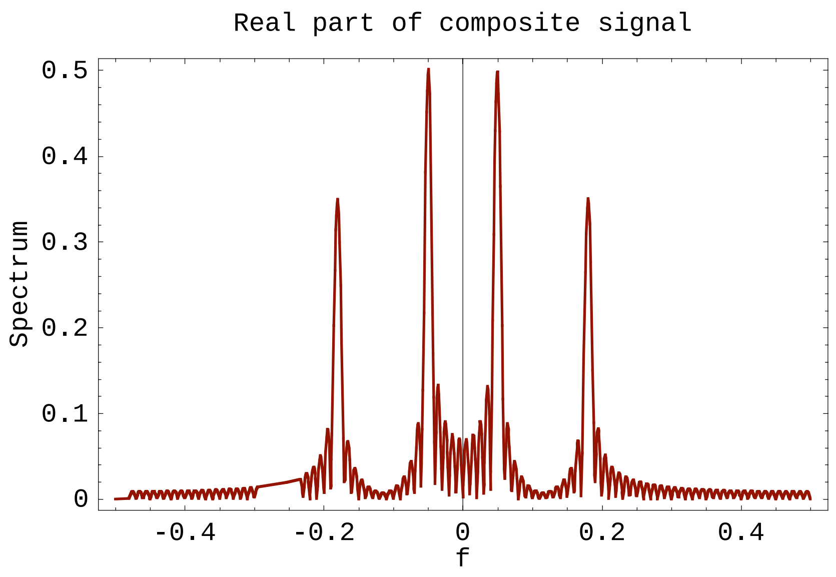
The plot of the imaginary part of the composite signal follows:

```
SequencePlot[Im[compositeSeq], PlotLabel → "Imaginary part of composite signal"];
```



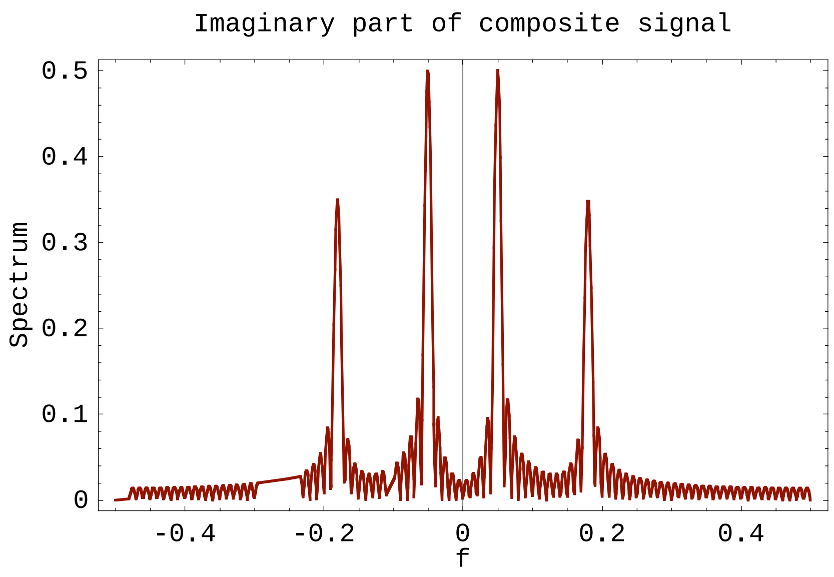
The magnitude spectrum of the real part of the composite signal is

```
SequenceFourierTransformMagnitudePlot[Re[compositeSeq],
 AxesLabel → "Real part of composite signal", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



and it is the same as the magnitude spectrum of the imaginary part of the composite signal:

```
SequenceFourierTransformMagnitudePlot[Im[compositeSeq],
 AxesLabel → "Imaginary part of composite signal", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```

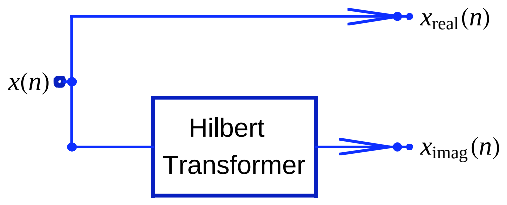


## 10.2. Hilbert Transformer

### Ideal Discrete Hilbert Transformer

Analytic signal can be generated from a real discrete signal by passing the real signal through a linear discrete system shown in the following figure:

```
Needs["SchematicSolver`"];

ShowSchematic[SchematicSolverFigureHilbertTransformerIdeal,
  Frame → False, GridLines → None, PlotRange → {{-3, 25}, All}];
```



Frequency response of an *ideal Hilbert transformer* is given by

$$H_{\text{Hilbert}}(e^{i2\pi f}) = \begin{cases} i, & -\frac{1}{2} \le f < 0 \\ -i, & 0 \le f < \frac{1}{2} \end{cases}$$

The ideal Hilbert transformer has a +90-degree phase-shift for $-\frac{1}{2} \le f < 0$, and a –90-degree phase-shift for $0 \le f < \frac{1}{2}$. An ideal Hilbert transformer is also called a *90–degree phase–shifter* or $\frac{\pi}{2}$ *phase splitter*.

### Design of Hilbert Transformer

Any realization that approximates the ideal Hilbert transformer is referred to as a *Hilbert transformer*. In this section, we present a realization of Hilbert transformer using a *half-band filter*.

The frequency response of an ideal half-band filter is given by

$$H_{\text{halfband}}(e^{i 2\pi f}) = \begin{cases} 1, & 0 \leq f < \frac{1}{4} \\ 0, & \frac{1}{4} < f < \frac{1}{2} \end{cases}$$

Assume that we want to design a 14th-order Hilbert transformer. We can identify 7 subschematics that we simply call the stages.

```
numberOfHilbertStages = 7;
```

The corresponding half-band filter has the length of 15.

```
lengthHBF = 2 * numberOfHilbertStages + 1;
```

Assume that we chose the lowest passband frequency of Hilbert transformer as

```
passbandEdgeFreqHilbert = 0.05;
```

The corresponding half-band passband and stopband edge frequencies are

```
passbandEdgeFreqHalfband = 0.25 - passbandEdgeFreqHilbert;
stopbandEdgeFreqHalfband = 0.25 + passbandEdgeFreqHilbert;
```

The numeric coefficient values can be computed with filter design software, such as *Digital Image Processing* – a *Mathematica* application package:

```
Needs["ImageProcessing`"];
h = EquirippleFilter[lengthHBF,
   {0.0, passbandEdgeFreqHalfband, stopbandEdgeFreqHalfband, 0.5}, {1., 0.}];
```

The even coefficients are equal to zero, and we take only nonzero coefficients.

```
h1 = Take[2 * h, {1, lengthHBF, 2}]

{-0.0529753, 0.0882208, -0.186807,
 0.627852, 0.627852, -0.186807, 0.0882208, -0.0529753}
```

Hilbert transformer can be designed from half-band filter in a straightforward manner by replacing $z$ with $-i z$ or by replacing $z^2$ with $-z^2$

```
replacingSign = ((-1) ^ (Mod[numberOfHilbertStages + 1, 4] / 2)) *
   UnitExponentialSequence[numberOfHilbertStages + 1, 1, -1] // Flatten

{1, -1, 1, -1, 1, -1, 1, -1}

parameterValues = h1 * replacingSign

{-0.0529753, -0.0882208, -0.186807,
 -0.627852, 0.627852, 0.186807, 0.0882208, 0.0529753}
```

Symbolic names of the coefficients can be automatically generated as follows:

```
parameterSymbols = UnitSymbolicSequence[numberOfHilbertStages + 1, c, 0] // Flatten

{c0, c1, c2, c3, c4, c5, c6, c7}
```

Here is the parameter substitution list:

```
parameterSubstitution = parameterSymbols → parameterValues // Thread
```

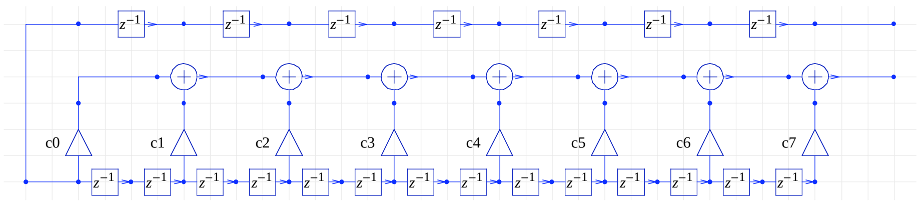
```
{c0 → -0.0529753, c1 → -0.0882208, c2 → -0.186807, c3 → -0.627852,
 c4 → 0.627852, c5 → 0.186807, c6 → 0.0882208, c7 → 0.0529753}
```

The coefficient values of Hilbert transformer can be computed, also, with *Digital Image Processing* application package using the optional type keyword "hilbert".

## Draw Schematic of System with Hilbert Transformer

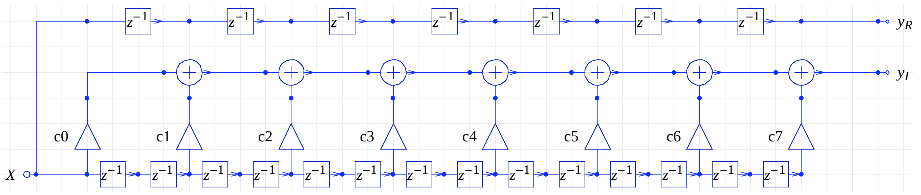
First, we draw the schematic of the Hilbert transformer.

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];

{htSchematic, inpCoordsHT, outCoordsHT} =
  HilbertTransformerDirectFormFIRSchematic[parameterSymbols];
ShowSchematic[htSchematic, FontSize → 6, Frame → False];
```



We draw the system with the Hilbert transformer by adding the input and the output parts:

```
htSystem = Join[
  htSchematic,
  {{"Input", inpCoordsHT[[1]], X}},
  {{"Output", outCoordsHT[[1]], "yI"}},
  {{"Output", outCoordsHT[[2]], "yR"}}];
ShowSchematic[%, FontSize → 6, Frame → False];
```



## Transfer Function of Hilbert Transformer

*SchematicSolver*'s function `DiscreteSystemTransferFunction` computes the transfer function of the system with Hilbert transformer:

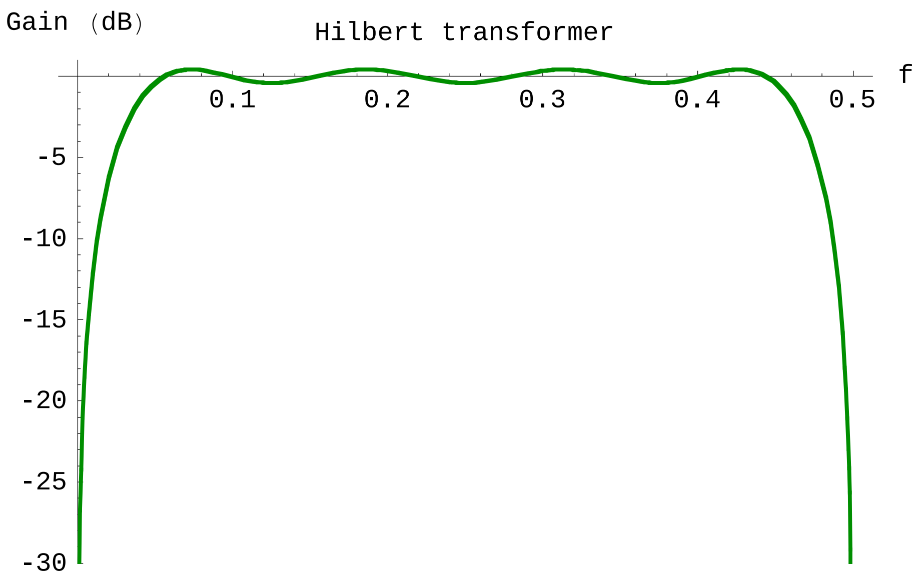
```
{htTF, systemInp, systemOut} = DiscreteSystemTransferFunction[htSystem] // Together;
htTF // MatrixForm
```

$$\begin{pmatrix} \frac{c7+c6\,z^2+c5\,z^4+c4\,z^6+c3\,z^8+c2\,z^{10}+c1\,z^{12}+c0\,z^{14}}{z^{14}} \\ \frac{1}{z^7} \end{pmatrix}$$

The system has one input and two outputs. Therefore, its transfer function is a 2-by-1 matrix.

The magnitude characteristic, gain in decibels, of the Hilbert transformer is

```
DiscreteSystemMagnitudeResponsePlot[
 htTF[[1, 1]] /. parameterSubstitution, {0, 0.5}, PlotRange → {-30, 1},
 AxesLabel → {"f", "Gain (dB)"}, PlotLabel → "Hilbert transformer"];
```



## Processing with Hilbert Transformer System

Consider the following example complex signal:

```
numberOfSamples = 200;

amplitude3 = 1;
frequency3 = 0.12;
phase3 = 0;
expSeq3 =
  amplitude3 * ⅇ^(ⅈ phase3) * UnitExponentialSequence[numberOfSamples, I 2 π frequency3, E];
```

The in-phase component of the signal is

```
inphaseSeq = Re[expSeq3];
```

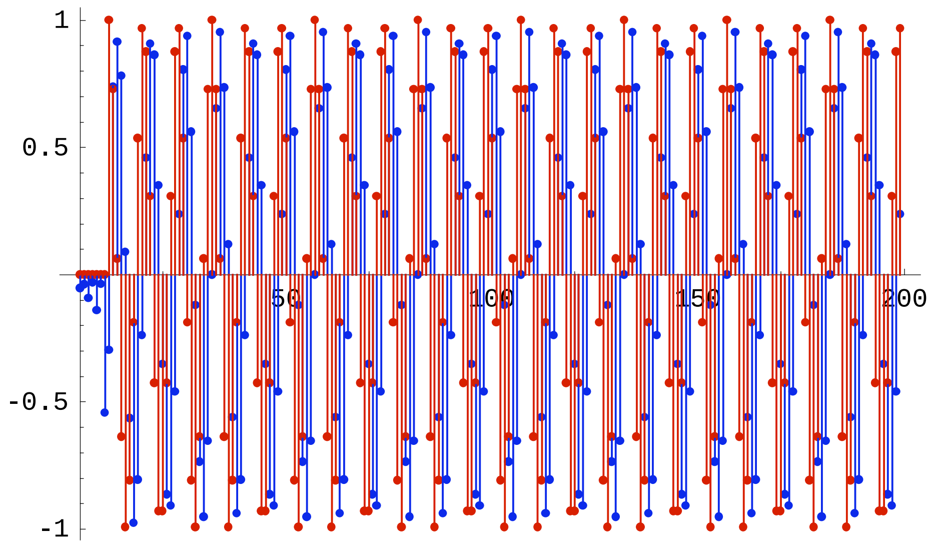
and it is inputted into `htSystem`.

`DiscreteSystemSimulation` finds the signals at both outputs of `htSystem`:

```
outSeq = DiscreteSystemSimulation[htSystem /. parameterSubstitution, inphaseSeq];
```

The first output signal, `outImagSeq`, corresponds to the quadrature component, and the second output signal, `outRealSeq`, corresponds to the in-phase component:

```
{outImagSeq, outRealSeq} = DemultiplexSequence[outSeq];
SequencePlot[outSeq];
```



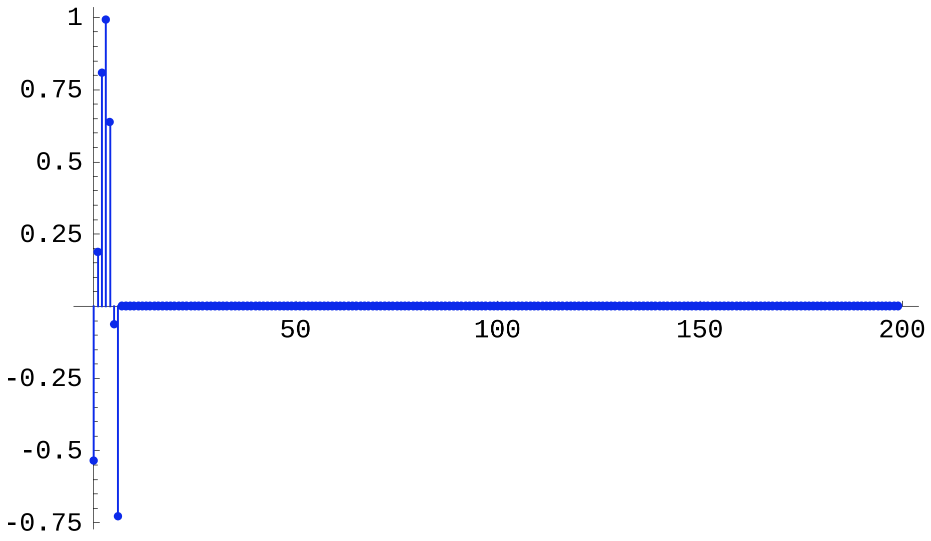
In fact, at the two outputs of `htSystem`, we have reconstructed the complex signal from its in-phase component.

The reconstructed signal is delayed due to processing. Let us generate the delayed original complex signal:

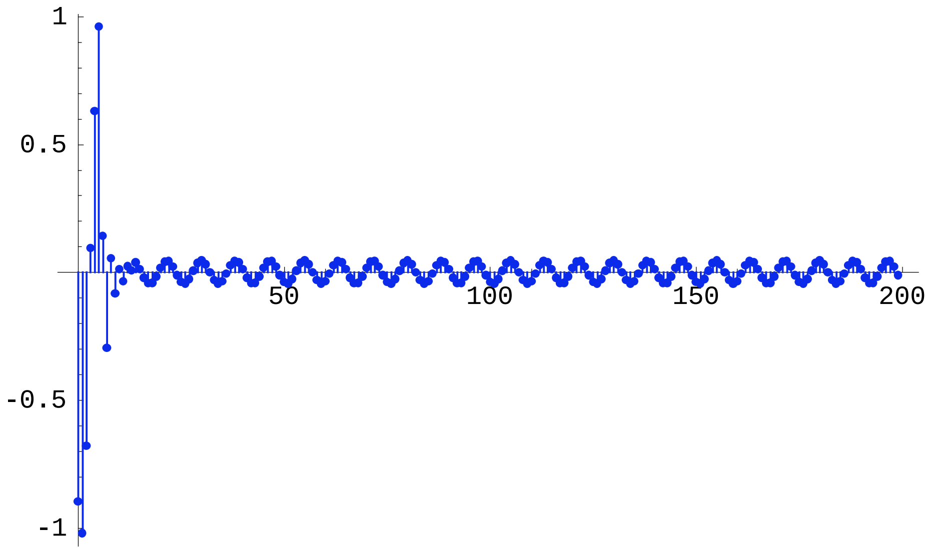
```
phaseShift = -2 * numberOfHilbertStages * frequency3 * Pi // N;

expSeq4 = ⅇ^(ⅈ phaseShift) * expSeq3;
```

The reconstructed signal and the delayed signal are practically the same, after some number of samples:

```
SequencePlot[outRealSeq - Re[expSeq4]];
```



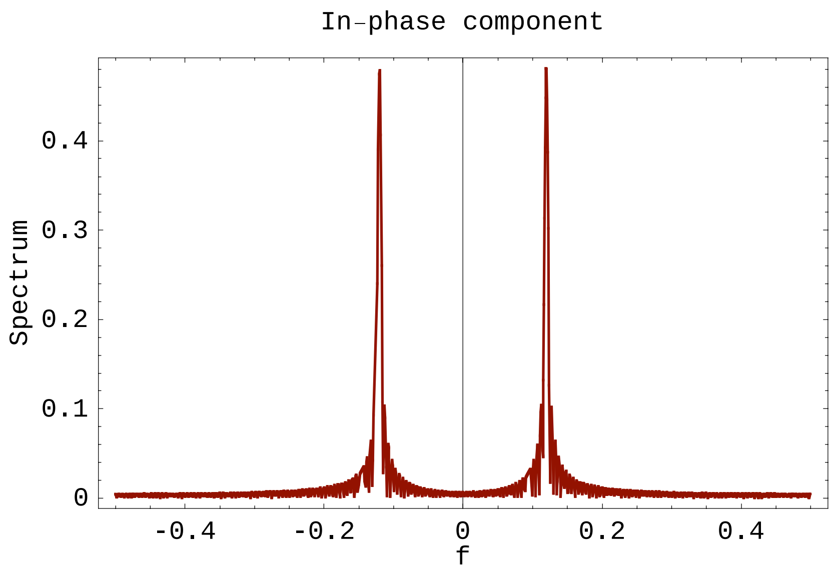
```
SequencePlot[outImagSeq - Im[expSeq4]];
```



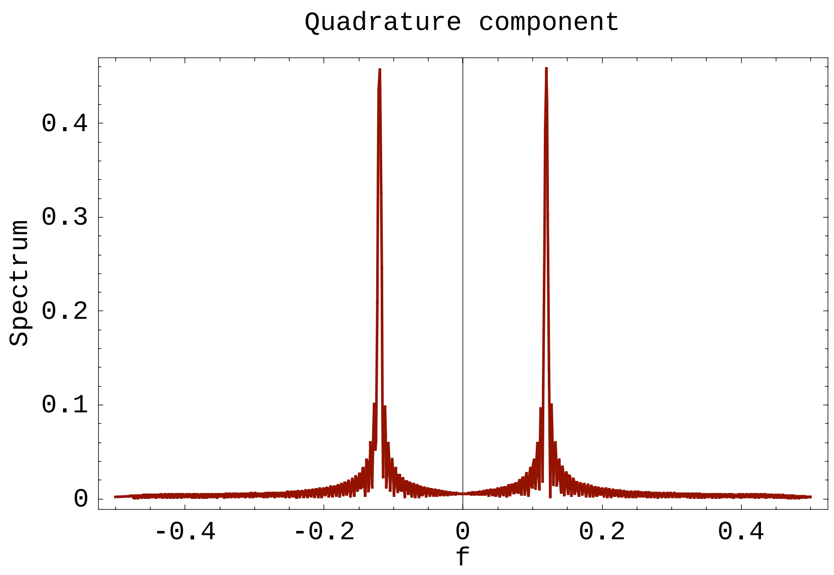
## Spectra of Reconstructed Signals

The spectra of the reconstructed signals can be computed using `SequenceFourierTransformMagnitudePlot`:

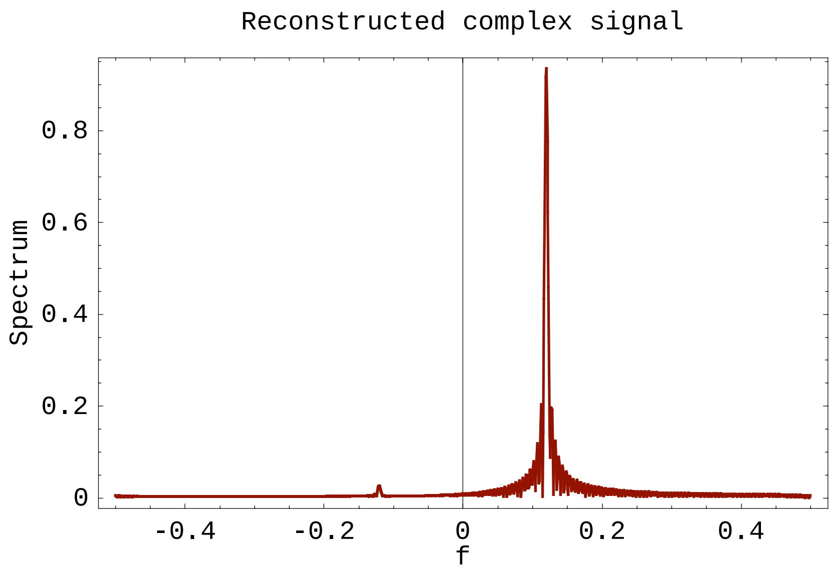
```
SequenceFourierTransformMagnitudePlot[outRealSeq,
 AxesLabel → "In-phase component", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



```
SequenceFourierTransformMagnitudePlot[outImagSeq,
 AxesLabel → "Quadrature component", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



```
SequenceFourierTransformMagnitudePlot[outRealSeq + ⅈ outImagSeq,
 AxesLabel → "Reconstructed complex signal", Frame → True,
 FrameLabel -> { "f", "Spectrum"}];
```



The reconstructed complex signal has, practically, zero-valued spectrum for $-\frac{1}{2} < f < 0$, consequently it is an analytical signal.

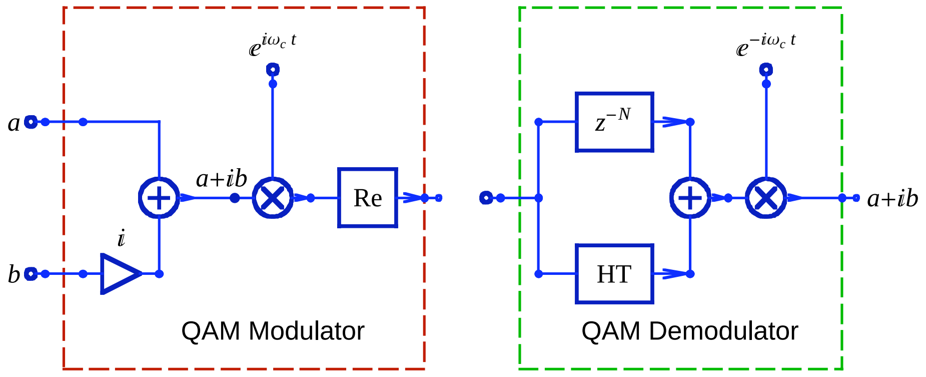
## 10.3. Hilbert Transformer in Quadrature Amplitude Modulation

### Implementation of QAM using Hilbert Transformer

*Quadrature Amplitude Modulation* (QAM) is a widely used method for transmitting digital data over bandpass channels. The simulation of a simplified and idealized QAM system, in which the Hilbert transformer is used, follows.

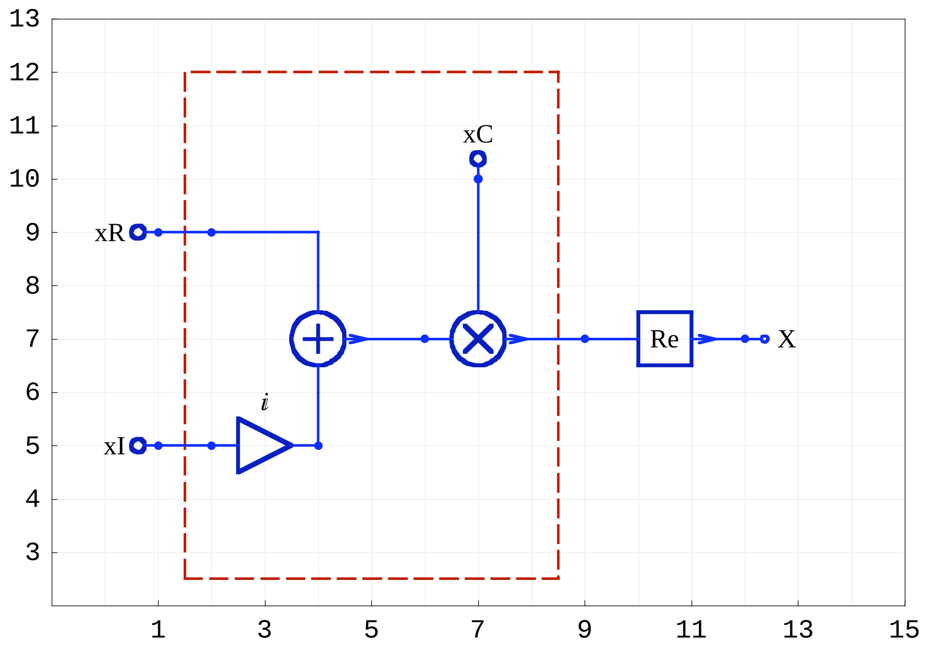
Here is the representation of the QAM modulator and demodulator in terms of complex signals.

```
Needs["SchematicSolver`"];

SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleDefault];

ShowSchematic[SchematicSolverFigureHilbertTransformerQAM,
  Frame → False, GridLines → None];
```



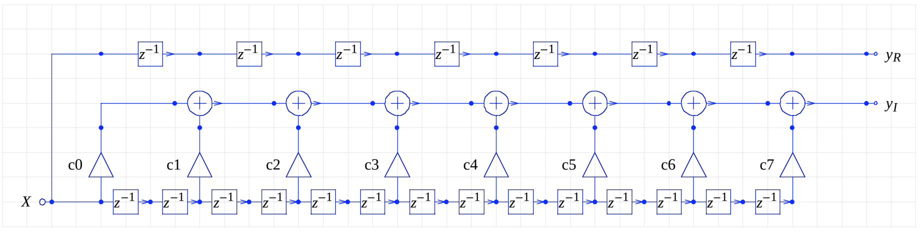
The modulator QAM system follows:

```
modulatorSystem =
  {{"Polyline", {{1.5, 2.5}, {8.5, 2.5}, {8.5, 12}, {1.5, 12}, {1.5, 2.5}},
    PlotStyle → {{RGBColor[0, 1, 0]}, {RGBColor[.7, 0, 0]}}},
   {"Multiplier", {{2, 5}, {4, 5}}, I, ""},
   {"Modulator", {{6, 7}, {7, 6}, {9, 7}, {7, 10}}, {1, 0, 2, 1}, ""},
   {"Output", {12, 7}, "X"},
   {"Adder", {{3, 7}, {4, 5}, {6, 7}, {2, 9}}, {0, 1, 2, 1}, ""},
   {"Line", {{1, 9}, {2, 9}}}, {"Line", {{1, 5}, {2, 5}}},
   {"Function", {{9, 7}, {12, 7}}, Re},
   {"Input", {1, 5}, "xI"},
   {"Input", {1, 9}, "xR"},
   {"Input", {7, 10}, "xC", "", TextOffset → {0, -1}}};
ShowSchematic[%, PlotRange → {{-1, 15}, {2, 13}}];
```

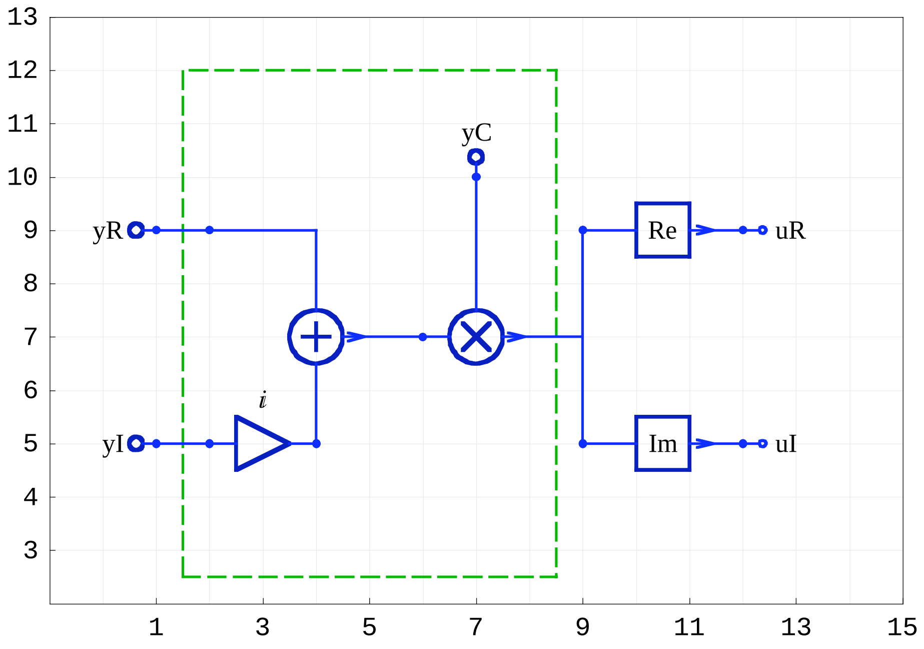


The demodulator QAM system consists of two subsystems `htSystem` and `demodulatorSubsystem`:

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];

ShowSchematic[htSystem, PlotRange → {{-2, numberOfHilbertStages * 4 + 7}, {-1, 8}},
  FontSize → 6, Frame → False];
```



```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleDefault];

demodulatorSubsystem =
  {{"Polyline", {{1.5, 2.5}, {8.5, 2.5}, {8.5, 12}, {1.5, 12}, {1.5, 2.5}},
    PlotStyle → {{RGBColor[0, 1, 0]}, {RGBColor[0, .7, 0]}}},
   {"Multiplier", {{2, 5}, {4, 5}}, I, ""},
   {"Modulator", {{6, 7}, {7, 6}, {9, 9}, {7, 10}}, {1, 0, 2, 1}},
   {"Output", {12, 9}, "uR"},
   {"Output", {12, 5}, "uI"},
   {"Adder", {{3, 7}, {4, 5}, {6, 7}, {2, 9}}, {0, 1, 2, 1}, ""},
   {"Line", {{1, 9}, {2, 9}}}, {"Line", {{1, 5}, {2, 5}}},
   {"Line", {{9, 5}, {9, 9}}},
   {"Function", {{9, 9}, {12, 9}}, Re},
   {"Function", {{9, 5}, {12, 5}}, Im},
   {"Input", {1, 9}, "yR"},
   {"Input", {1, 5}, "yI"},
   {"Input", {7, 10}, "yC", "", TextOffset → {0, -1}}};
ShowSchematic[%, PlotRange → {{-1, 15}, {2, 13}}];
```



Consider 100 samples of the signal S of the modem V.34, for a symbol rate of 2400, the carrier frequency 1600 Hz, and the sampling rate of 8000 Hz:

```
numberOfSamples = 100;
Fc = 1600;
Fs = 8000;
Fy = 2400 / 2;
```

Digital frequency of the in-phase sinusoidal sequence is `Fy/Fs` (`Fy` is a half of the symbol rate), and the quadrature signal is a step sequence. The carrier `xC` is generated as an exponential complex sequence.

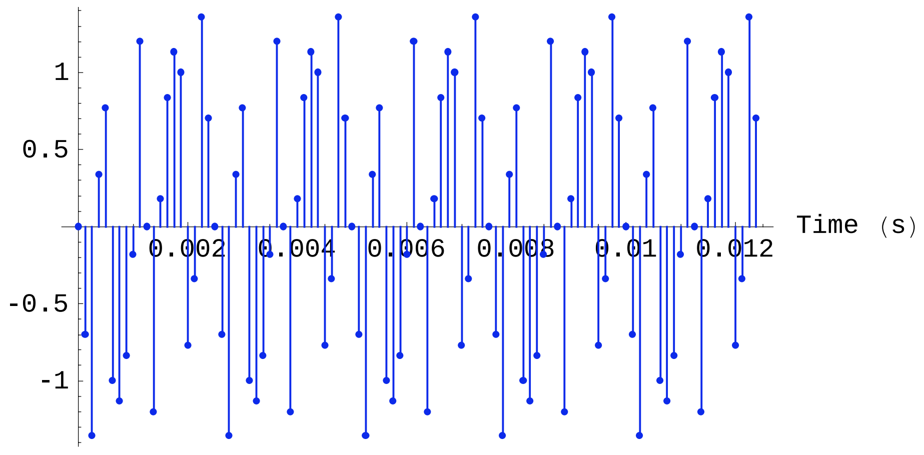
```
xI = UnitSineSequence[numberOfSamples, Fy / Fs];
xQ = UnitStepSequence[numberOfSamples];
xC = UnitExponentialSequence[numberOfSamples, I 2 π Fc / Fs, E];
```

The input sequence to the modulator system is

```
inpModSeq = MultiplexSequence[xQ, xI, xC];
```

`DiscreteSystemSimulation` finds the output sequence of `modulatorSystem` that is a real signal

```
outModSeq = DiscreteSystemSimulation[modulatorSystem, inpModSeq];
SequencePlot[outModSeq,
  SequenceSamplingFrequency → Fs, AxesLabel → {"Time (s)", ""}];
```



which is the input to `htSystem`.

`DiscreteSystemSimulation` finds the signals at both outputs of `htSystem`:

```
outHTSeq = DiscreteSystemSimulation[htSystem /. parameterSubstitution, outModSeq];
```

The carrier `yC` is generated as a delayed exponential complex sequence.

```
phaseShiftDem = -2 * numberOfHilbertStages * Fc / Fs * Pi // N;
```

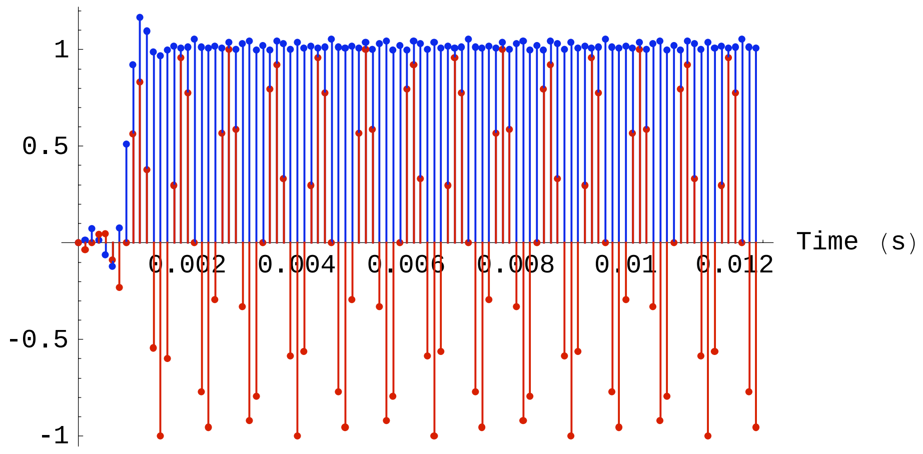
$$yC = e^{i\ phaseShiftDem} * xC;$$

The input sequence to the demodulator subsystem is

```
inpDemSeq = MultiplexSequence[outHTSeq, yC];
```

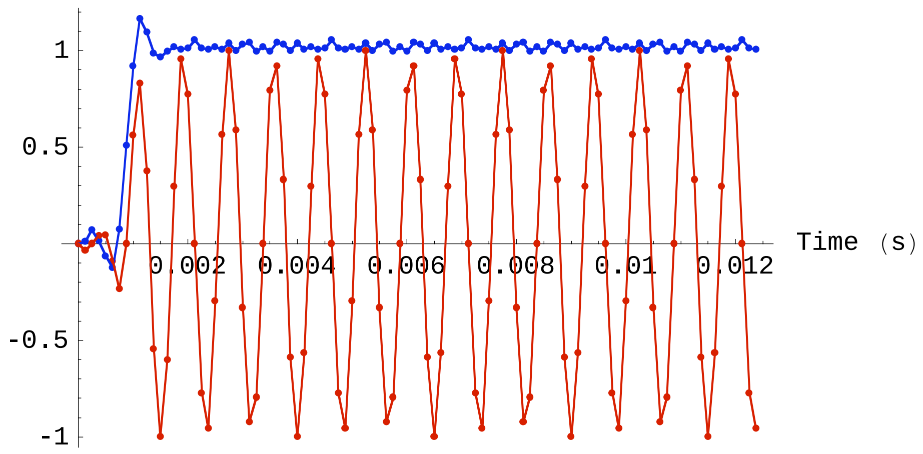
`DiscreteSystemSimulation` finds the signals at both outputs of `demodulatorSubsystem`:

```
outDemSeq = DiscreteSystemSimulation[demodulatorSubsystem, inpDemSeq];
SequencePlot[outDemSeq,
  SequenceSamplingFrequency → Fs, AxesLabel → {"Time (s)", ""}];
```



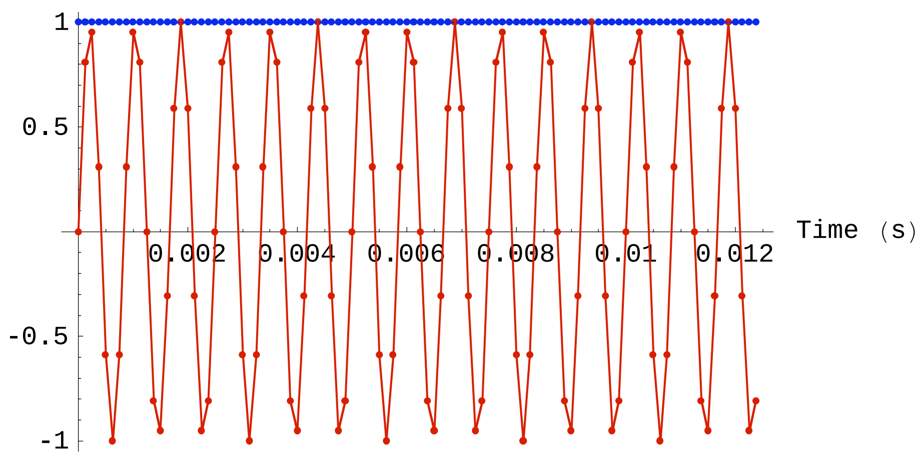
You can plot the discrete signals more clearly by setting the SequencePlot options to StemPlot→False and PlotJoined→True.

```
SetOptions[SequencePlot, StemPlot → False, PlotJoined → True];

SequencePlot[outDemSeq, SequenceSamplingFrequency → Fs, AxesLabel → {"Time (s)", ""}];
```



Let us replot the two sequences inputted to the modulator system.

```
SequencePlot[MultiplexSequence[xQ, xI],
  SequenceSamplingFrequency → Fs, AxesLabel → {"Time (s)", ""}];
```

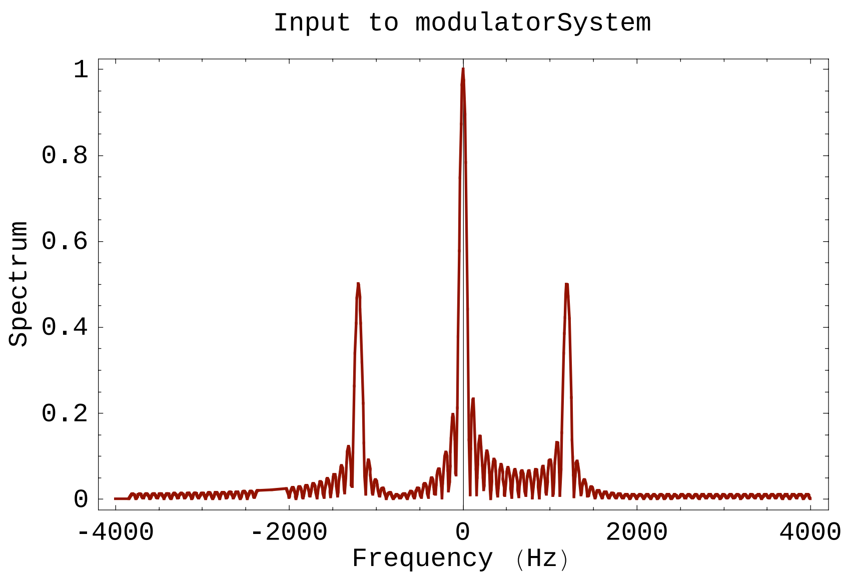


The demodulated signal and the original signal are practically the same, after some number of samples.

## Spectra of QAM Signals

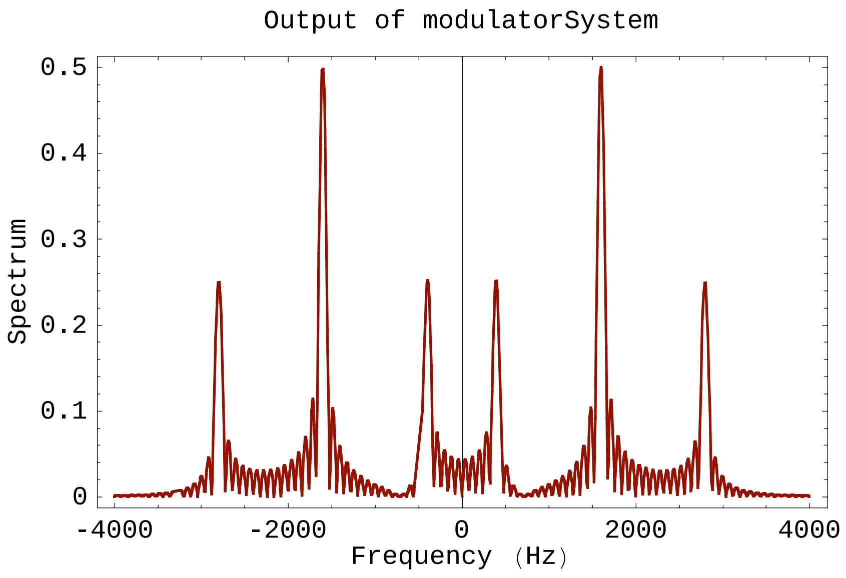
Let us plot the spectrum of the complex sequence inputted to the `modulatorSystem`.

```
SequenceFourierTransformMagnitudePlot[xI + ⅈ xQ, SequenceSamplingFrequency → Fs,
 AxesLabel → "Input to modulatorSystem", Frame → True,
 FrameLabel -> { "Frequency (Hz)", "Spectrum"}];
```



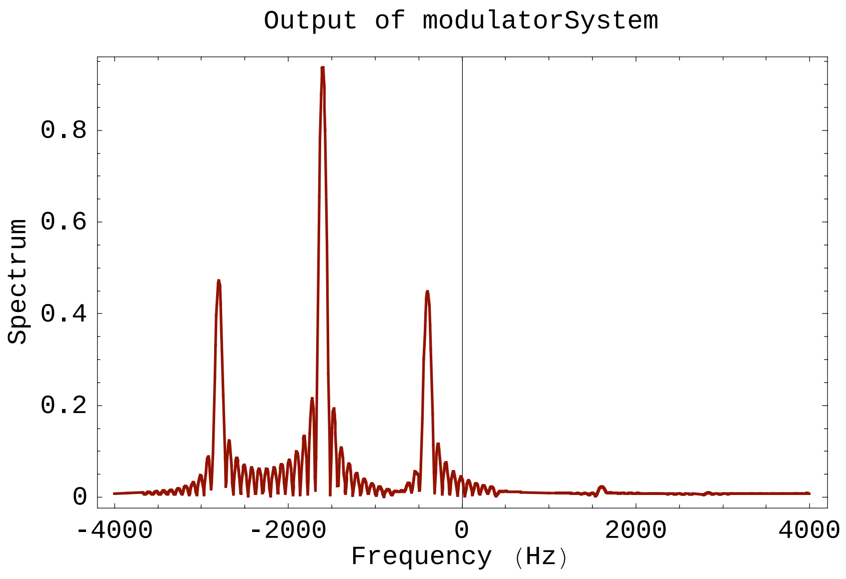
The spectrum of the real signal at the output of `modulatorSystem` follows:

```
SequenceFourierTransformMagnitudePlot[outModSeq,
 SequenceSamplingFrequency → Fs,
 AxesLabel → "Output of modulatorSystem", Frame → True,
 FrameLabel -> { "Frequency (Hz)", "Spectrum"}];
```



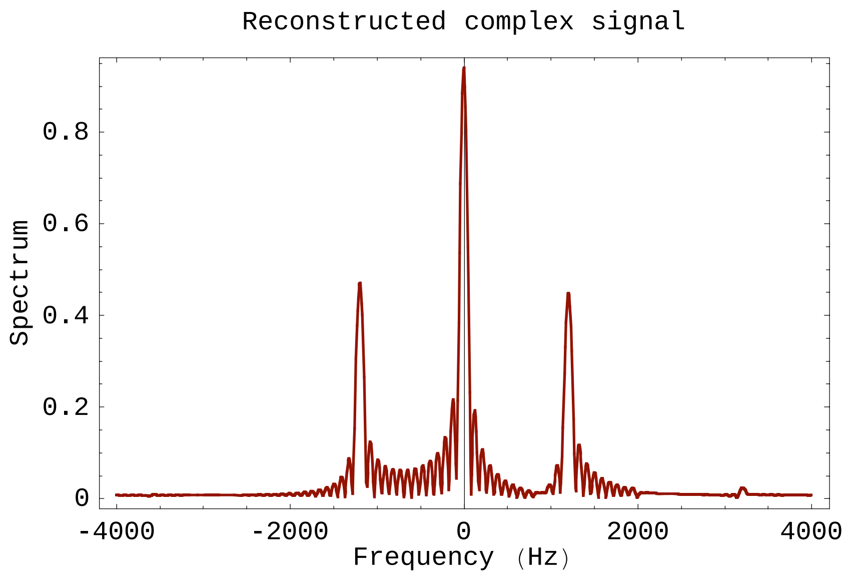
The spectrum of the complex signal, that is a sum of the signals at both outputs of `htSystem`, is zero-valued for $0 < f < \frac{\text{Fs}}{2}$.

```
SequenceFourierTransformMagnitudePlot[
 First[DemultiplexSequence[outHTSeq]] + ⅈ Last[DemultiplexSequence[outHTSeq]],
 SequenceSamplingFrequency → Fs,
 AxesLabel → "Output of modulatorSystem", Frame → True,
 FrameLabel -> { "Frequency (Hz)", "Spectrum"}];
```



Here is the spectrum of the complex signal at the outputs of `demodulatorSubsystem`:

```
SequenceFourierTransformMagnitudePlot[
 First[DemultiplexSequence[outDemSeq]] + ⅈ Last[DemultiplexSequence[outDemSeq]],
 SequenceSamplingFrequency → Fs,
 AxesLabel → "Reconstructed complex signal", Frame → True,
 FrameLabel -> { "Frequency (Hz)", "Spectrum"}];
```



The spectrum of the complex signal at the outputs of `demodulatorSubsystem` is similar to the spectrum of the complex sequence inputted to `modulatorSystem`.

# 11. Multirate Systems

## 11.1. Introduction

### What are Multirate systems?

Classic signal processing assumes a *single-rate system* in which the sampling rates are the same at all nodes of the system. *Multirate system* works with two or more sampling rates.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

### Decimation

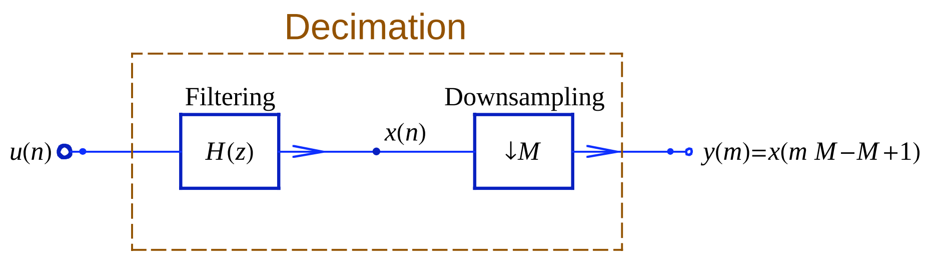
The reduction of a sampling rate is called *decimation*. Decimation consists of two stages:

1) *filtering*

2) *downsampling*

as shown in the figure below.

```
ShowSchematic[SchematicSolverFigureMultirateDecimation,
  GridLines → None, Frame → False];
```

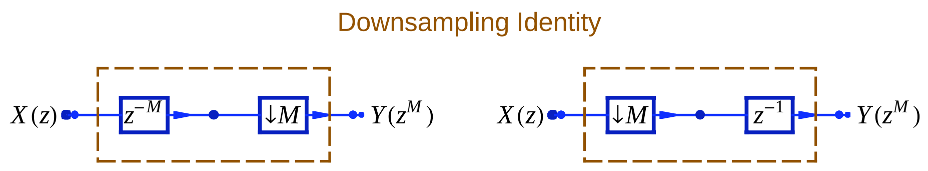


*Downsampling* reduces the sampling rate by an integer factor $M$, which is known as a *downsampling factor*, also called a *decimation factor*.

## Downsampling Identity

The $M$-sample delay before downsampling is equivalent to a single-sample delay after downsampling.

```
ShowSchematic[SchematicSolverFigureMultirateDownsamplingIdentity,
  GridLines → None, Frame → False];
```



## Interpolation

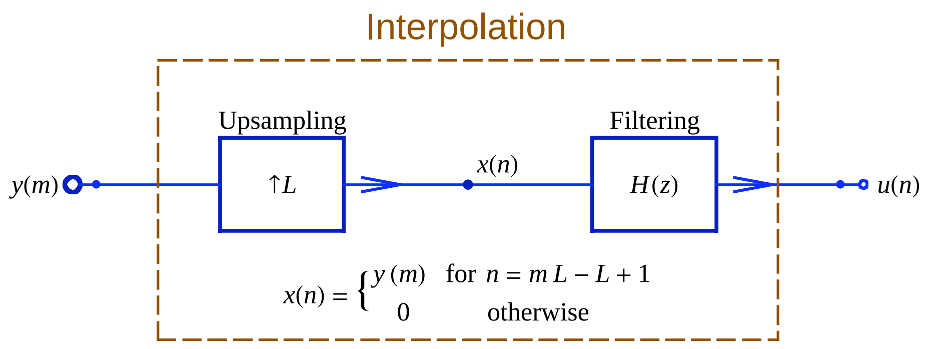
The increasing of a sampling rate is called *interpolation*. Interpolation consists of two stages:

1) *upsampling*

2) *filtering*

as shown in the figure below.

```
ShowSchematic[SchematicSolverFigureMultirateInterpolation,
  GridLines → None, Frame → False];
```



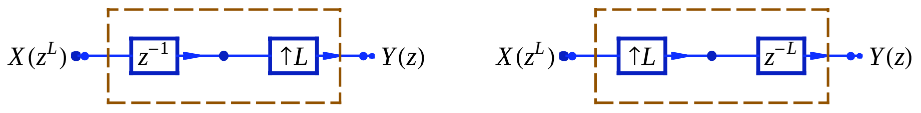
*Upsampling* increases the sampling rate by an integer factor $L$, which is known as an *upsampling factor*, also called an *interpolation factor*.

## Upsampling Identity

The single-sample delay before upsampling is equivalent to an *L*-sample delay after upsampling.

```
ShowSchematic[SchematicSolverFigureMultirateUpsamplingIdentity,
  GridLines → None, Frame → False];
```

Upsampling Identity



## References

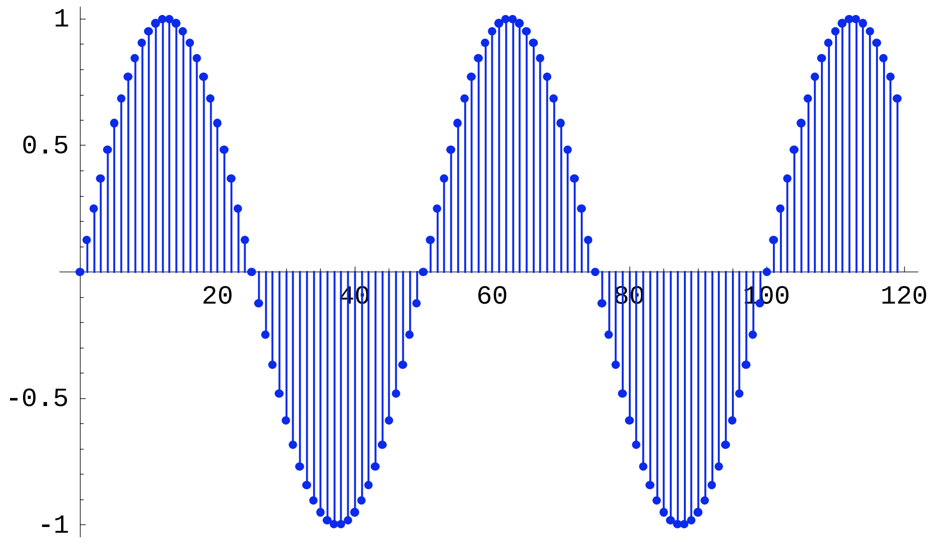
A more detailed introduction to multirate systems can be found in the book:

Dolecek, G., *Multirate Systems: Design and Applications*, Idea Group Publishers, Hershey, PA, 2002.

## 11.2. Downsampling and Upsampling

Consider an example sequence

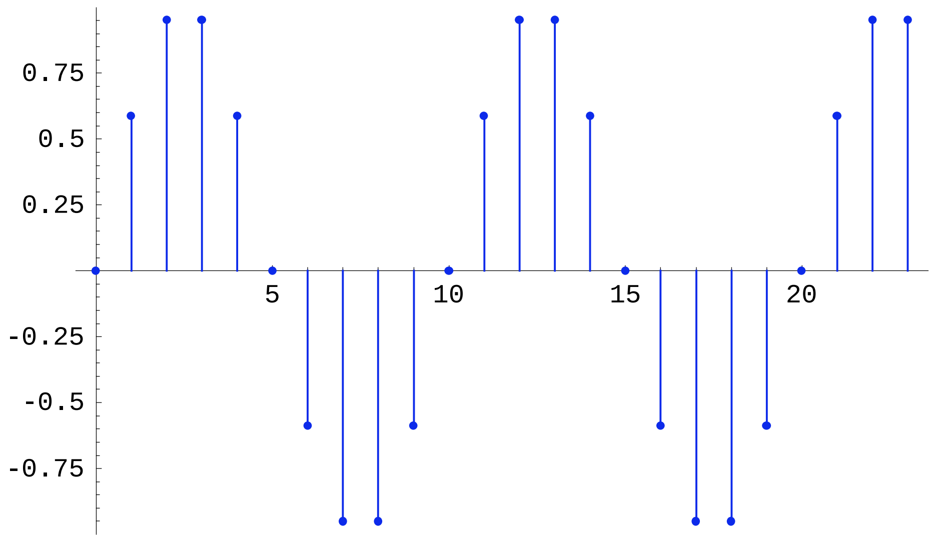
```
dataSeq = UnitSineSequence[120, 0.02];
% // SequencePlot;
```



*SchematicSolver*'s function `DownsampleSequence` implements downsampling:

```
M = 5;

downSeq5 = DownsampleSequence[dataSeq, M]
% // SequencePlot;
```

```
{{0}, {0.587785}, {0.951057}, {0.951057}, {0.587785}, {1.22461×10^-16}, {-0.587785},
 {-0.951057}, {-0.951057}, {-0.587785}, {-2.44921×10^-16}, {0.587785},
 {0.951057}, {0.951057}, {0.587785}, {3.67382×10^-16}, {-0.587785}, {-0.951057},
 {-0.951057}, {-0.587785}, {-4.89843×10^-16}, {0.587785}, {0.951057}, {0.951057}}
```



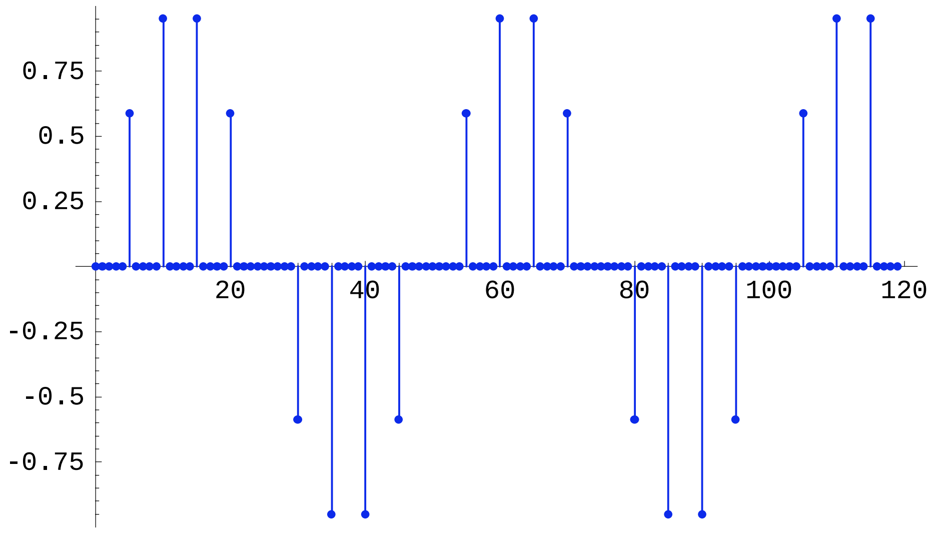
```
downSeq10 = DownsampleSequence[dataSeq, 2 * M]
```

```
{{0}, {0.951057}, {0.587785}, {-0.587785}, {-0.951057}, {-2.44921×10^-16},
 {0.951057}, {0.587785}, {-0.587785}, {-0.951057}, {-4.89843×10^-16}, {0.951057}}
```

*SchematicSolver*'s function `UpsampleSequence` implements upsampling:

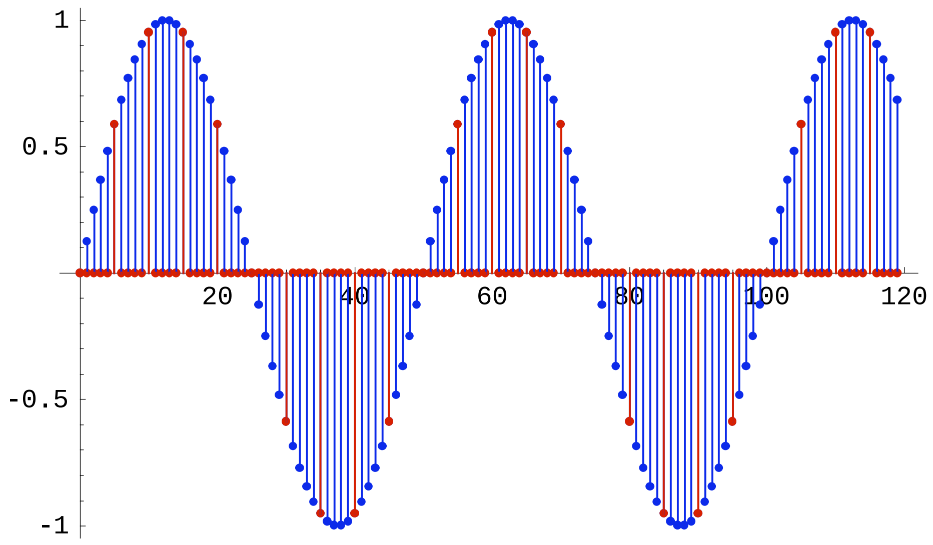
```
L = 5;
```

```
upSeq5 = UpsampleSequence[downSeq5, L];
% // SequencePlot;
```



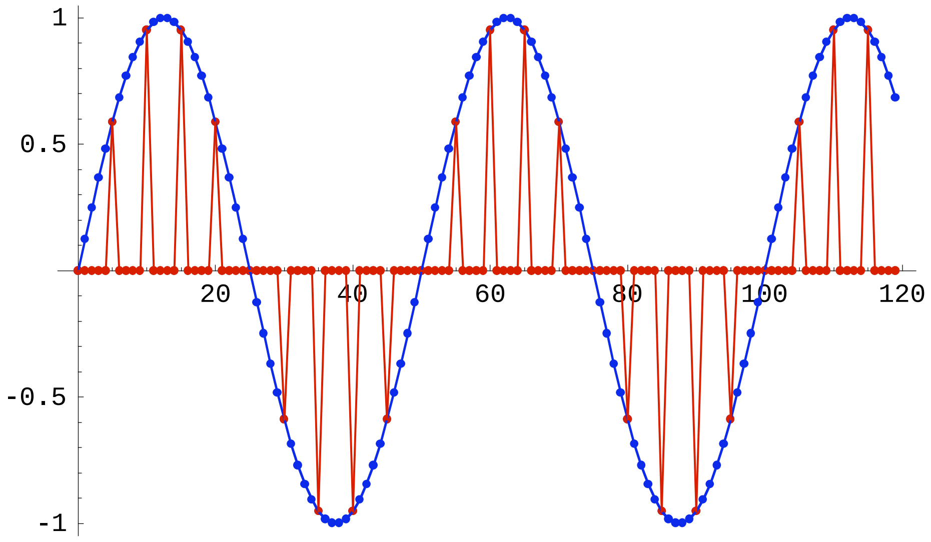
Here are the original signal and the upsampled signal:

```
MultiplexSequence[dataSeq, upSeq5] // SequencePlot;
```

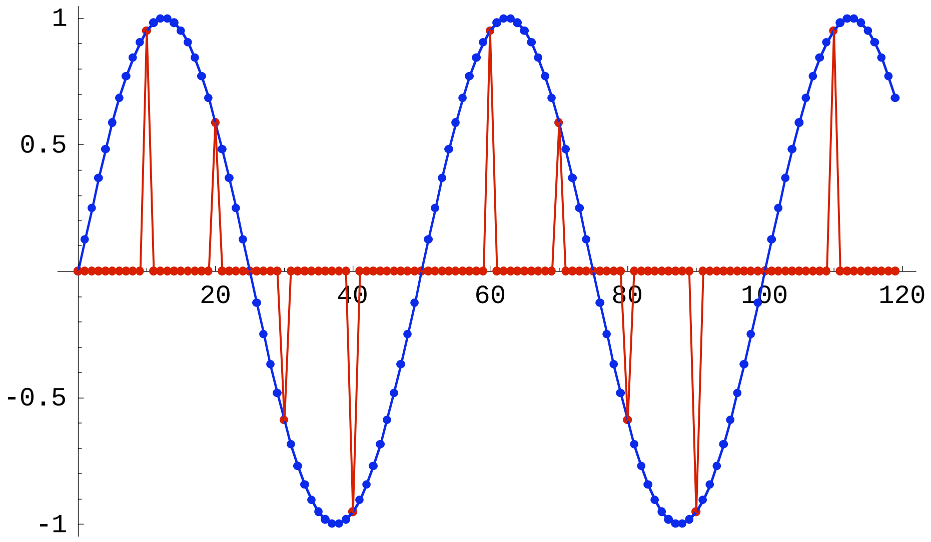


To illustrate the effect of downsampling and upsampling more clearly, the two signals have been plotted using the options `StemPlot→False` and `PlotJoined→True`.

```
SequencePlot[MultiplexSequence[dataSeq, upSeq5],
  StemPlot → False, PlotJoined → True];
```



```
upSeq10 = UpsampleSequence[downSeq10, 2 * L];
SequencePlot[MultiplexSequence[dataSeq, upSeq10],
  StemPlot → False, PlotJoined → True];
```



## 11.3. Spectra of Downsampled Signals

### Composite Signal

Assume that we want to process a composite signal that is composed of two sinusoidal sequences and a noise sequence.

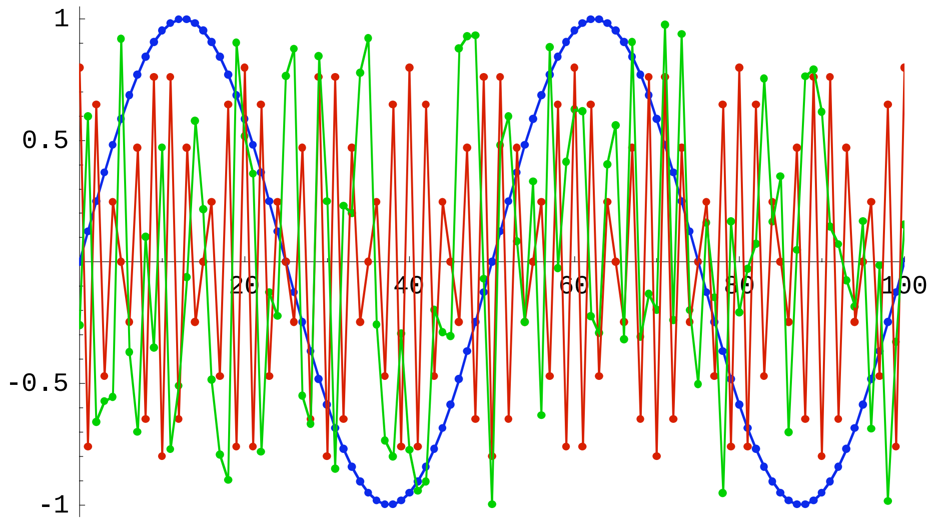
```
numberOfSamples = 1000;

amplitude1 = 1;
frequency1 = 0.02;
phase1 = 0;
sineSeq1 = amplitude1 * UnitSineSequence[numberOfSamples, frequency1, phase1];

amplitude2 = 0.8;
frequency2 = 0.45;
phase2 = Pi / 2;
sineSeq2 = amplitude2 * UnitSineSequence[numberOfSamples, frequency2, phase2];

noiseSeq = UnitNoiseSequence[numberOfSamples];

compositeSeq = sineSeq1 + sineSeq2 + noiseSeq;
```

To plot multiplex sequences and large sequences more clearly, the `SequencePlot` options are set to `StemPlot→False` and `PlotJoined→True`.

```
SetOptions[SequencePlot, StemPlot → False, PlotJoined → True];
```

Here is a portion of the three sequences:

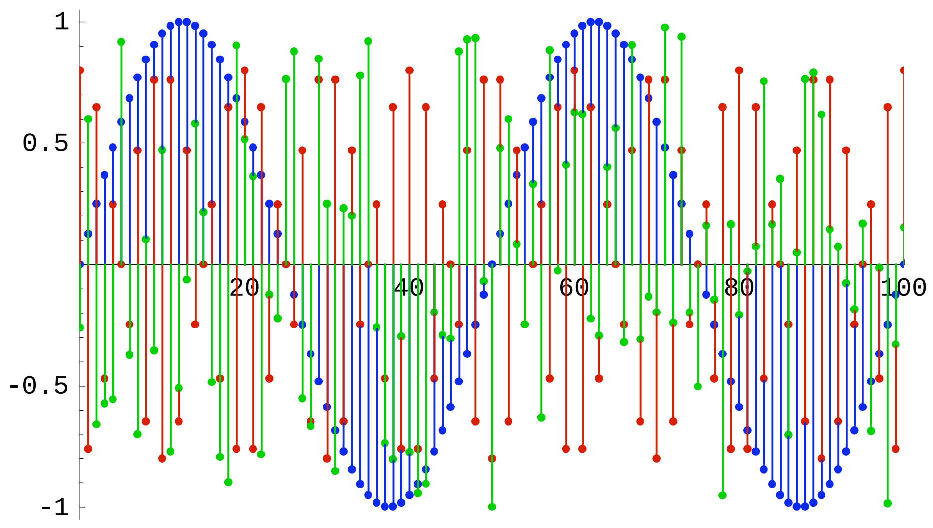
```
MultiplexSequence[sineSeq1, sineSeq2, noiseSeq];
SequencePlot[%, PlotRange → {{0, 100}, All}];
```



You can plot discrete signals in a traditional way by setting the `SequencePlot` options to `StemPlot→True` and `PlotJoined→False`.

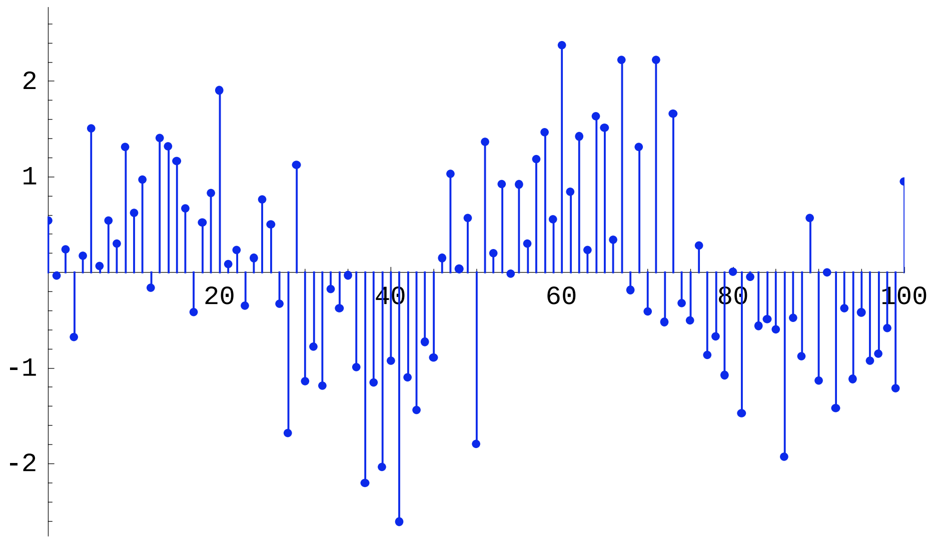
```
SetOptions[SequencePlot, StemPlot → True, PlotJoined → False];
```

```
MultiplexSequence[sineSeq1, sineSeq2, noiseSeq];
SequencePlot[%, PlotRange → {{0, 100}, All}];
```



Here is the plot of the composite signal:

```
SequencePlot[compositeSeq, PlotRange → {{0, 100}, All}];
```

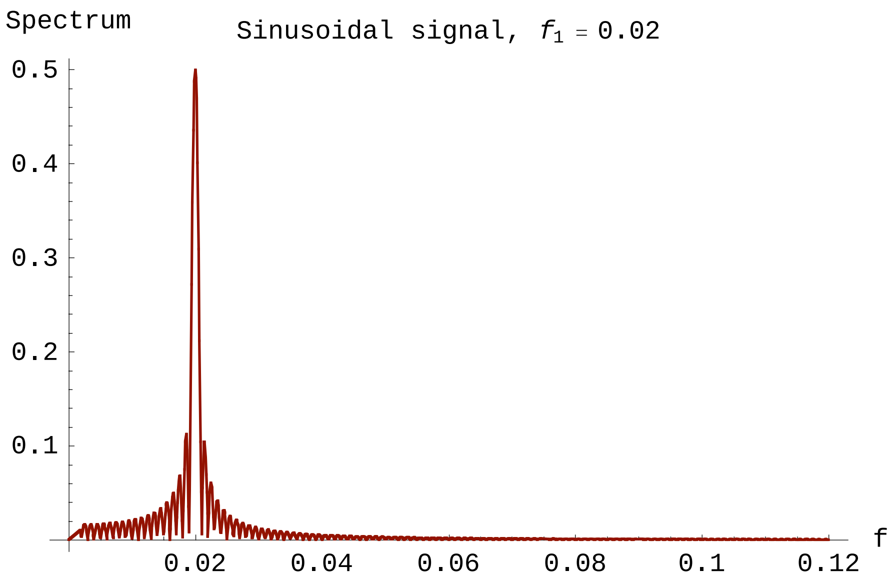


## Downsampled Signal

*SchematicSolver*'s function `SequenceFourierTransformMagnitudePlot` computes and plots the magnitude spectrum of discrete signals.

Here is the spectrum of the first sinusoidal sequence of the frequency $f_1 = 0.02$:

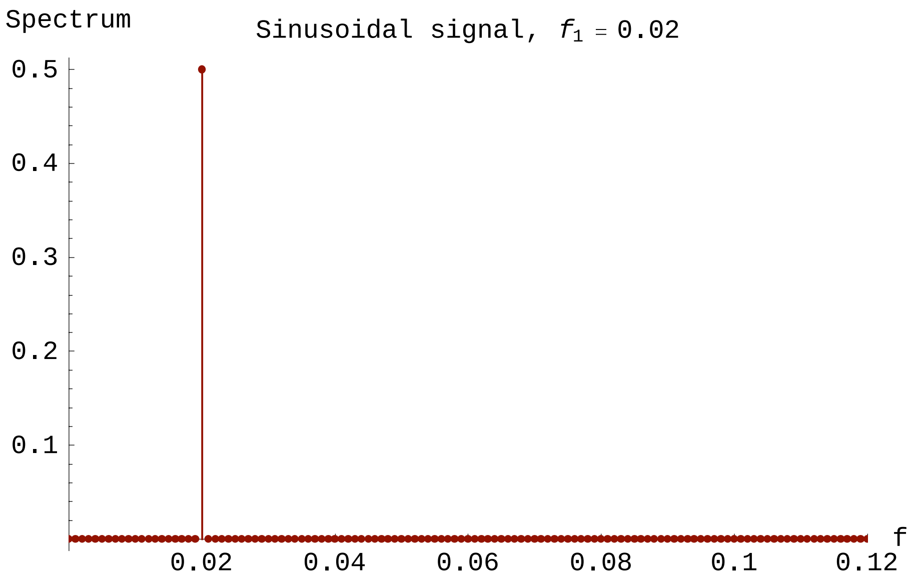
```
SequenceFourierTransformMagnitudePlot[sineSeq1, {0, 0.12},
 AxesLabel → { "f", "Spectrum"},
 PlotLabel → "Sinusoidal signal, f₁ = 0.02"];
```



`SequenceFourierTransformMagnitudePlot` shows a strong peak at $f_1 = 0.02$.

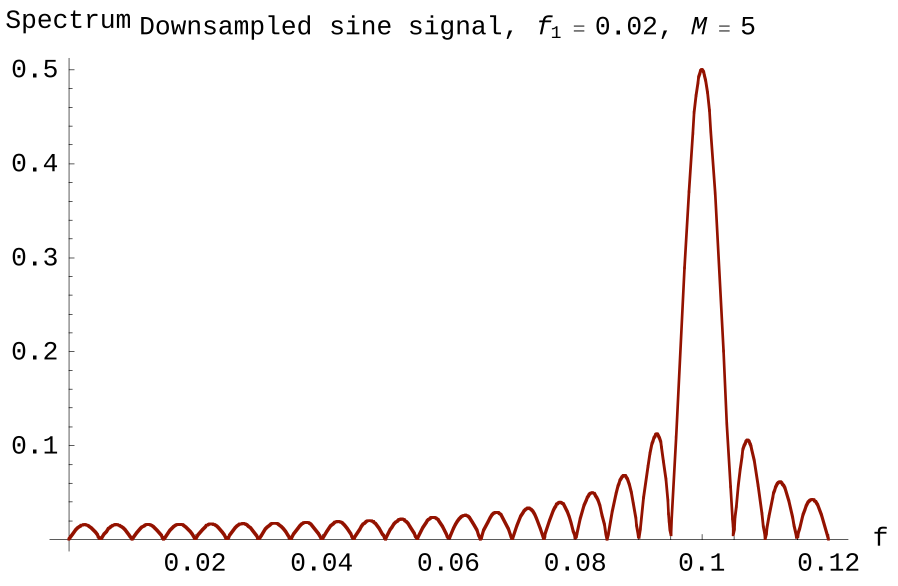
You can plot the discrete spectrum of discrete signals with `SequenceDiscreteFourierTransformMagnitude-Plot`.

```
SequenceDiscreteFourierTransformMagnitudePlot[sineSeq1,
 PlotRange → {{0, 0.12}, All},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Sinusoidal signal, f₁ = 0.02"];
```



Spectrum of the downsampled sinusoidal sequence of frequency $f_1 = 0.02$ follows:

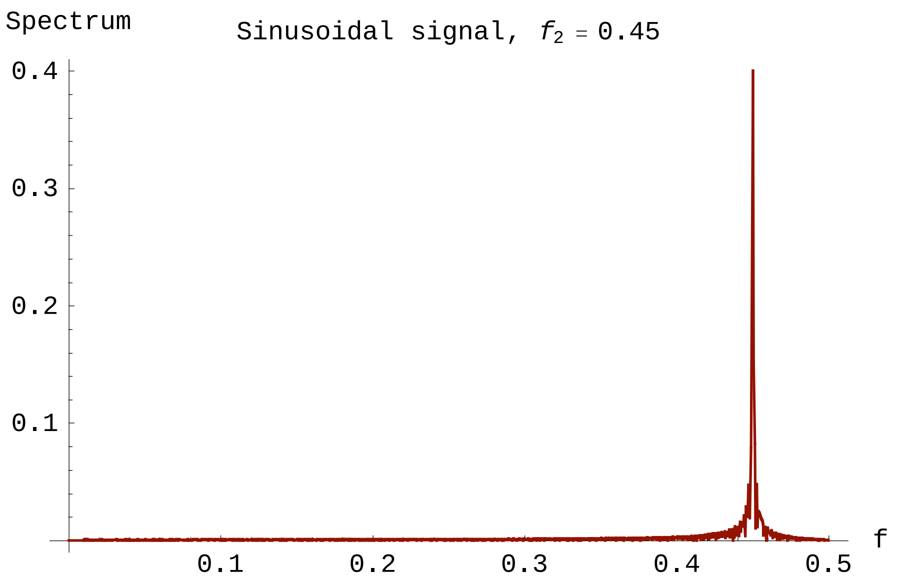
```
M = 5;

downSineSeq1 = DownsampleSequence[sineSeq1, M];

SequenceFourierTransformMagnitudePlot[downSineSeq1, {0, 0.12},
 AxesLabel → { "f", "Spectrum"},
 PlotLabel → "Downsampled sine signal, f1 = 0.02, M = 5"];
```



SequenceFourierTransformMagnitudePlot shows a strong peak at the frequency $M\ f_1 = 5\ f_1 = 0.1$.

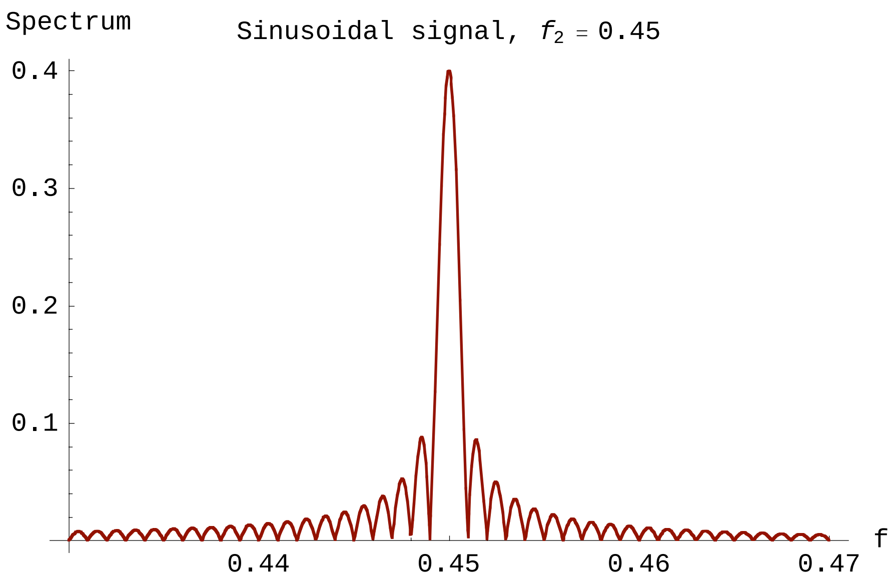
Spectrum of the second sinusoidal sequence of the frequency $f_2 = 0.45$ is

```
SequenceFourierTransformMagnitudePlot[sineSeq2, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Sinusoidal signal, f2 = 0.45"];
```



SequenceFourierTransformMagnitudePlot shows a strong peak at $f_2 = 0.45$ in the signal spectrum.

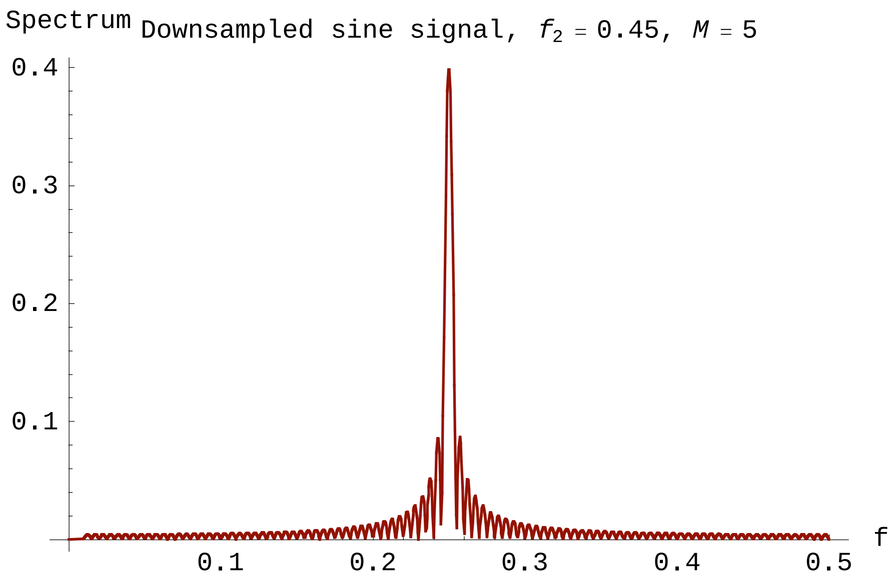
```
SequenceFourierTransformMagnitudePlot[sineSeq2, {0.43, 0.47},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Sinusoidal signal, f2 = 0.45"];
```



Spectrum of the downsampled sinusoidal sequence of frequency $f_2 = 0.45$ follows:

```
downSineSeq2 = DownsampleSequence[sineSeq2, M];
```

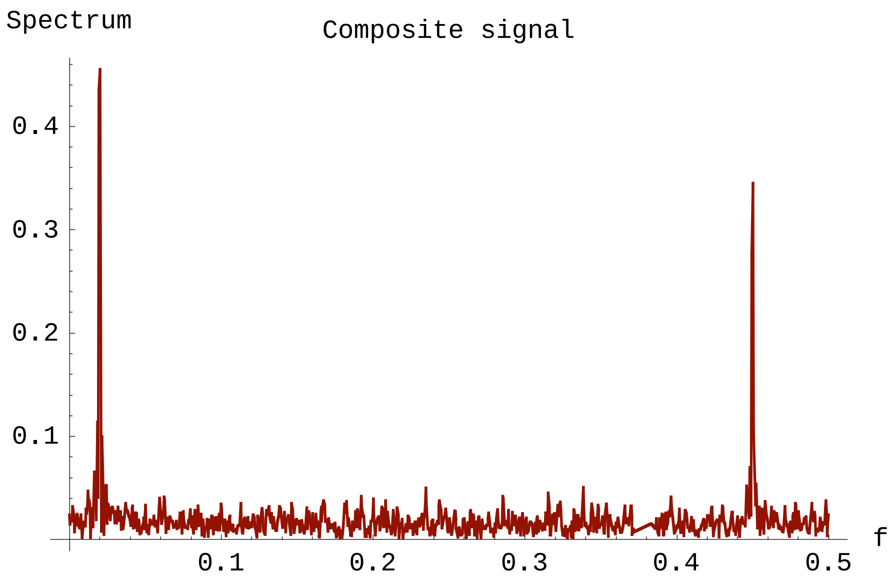
```
SequenceFourierTransformMagnitudePlot[downSineSeq2, {0, 0.5},
 AxesLabel → { "f", "Spectrum"},
 PlotLabel → "Downsampled sine signal, f2 = 0.45, M = 5"];
```



`SequenceFourierTransformMagnitudePlot` shows a strong peak at the frequency $f_{\text{folded}} = M\, f_2 - k\,\frac{1}{2} = 5\, f_2 - k\,\frac{1}{2} = 2.25 - 4\,\frac{1}{2} = 0.25$, which is the folded (aliased) spectral component. The same spectral component appears for a sinusoidal sequence of the frequency $f_{\sin} = \frac{1}{M}\, f_{\text{folded}} = 0.05$.

Here is the spectrum of the composite signal:

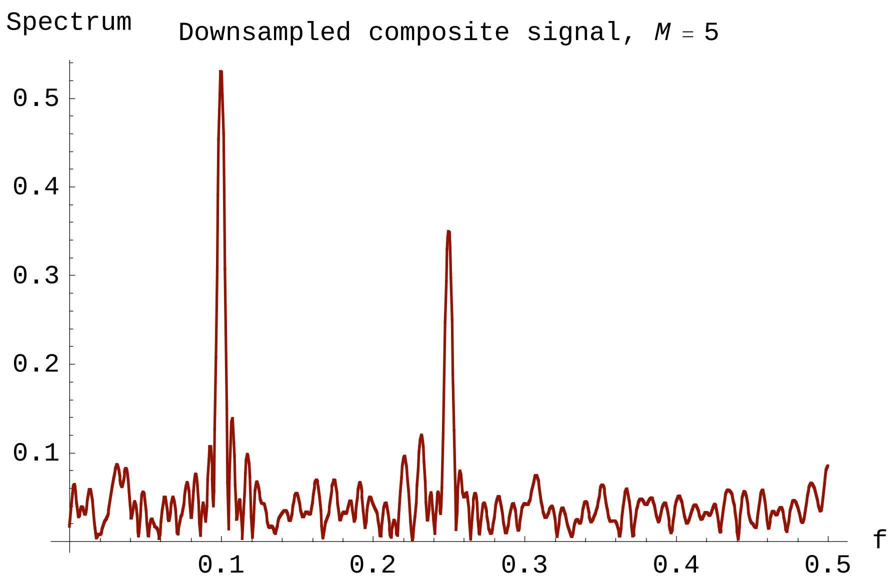
```
SequenceFourierTransformMagnitudePlot[compositeSeq, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Composite signal"];
```



`SequenceFourierTransformMagnitudePlot` shows two strong peaks at $f_1 = 0.02$ and $f_2 = 0.45$.

Here is the spectrum of the downsampled composite signal:

```
M = 5;

downCompositeSeq = DownsampleSequence[compositeSeq, M];

SequenceFourierTransformMagnitudePlot[downCompositeSeq, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Downsampled composite signal, M = 5"];
```



`SequenceFourierTransformMagnitudePlot` shows two strong peaks at $5\,f_1 = 0.1$ and at $f_{\text{folded}} = 0.25$.

If the composite signal is not bandlimited to the frequency $\frac{1}{2\,M}$, then the downsampled signal spectrum will contain folded (aliased) components. In order to avoid aliasing, it is necessary to bandlimit the spectrum of the composite signal, before downsampling, to a frequency below $\frac{1}{2M}$. This is why a lowpass filter should precede the downsampler. That lowpass filter is called a *decimation filter*.

## 11.4. Decimation FIR Filter

### Generate Parameter Names

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

Assume that we want to design a 59th-order FIR filter for decimation and interpolation. The filter has 59 stages

```
numberOfStages = 59;
```

and 60 coefficients. Names of the coefficients can be automatically generated as follows:

```
parameterSymbols = UnitSymbolicSequence[numberOfStages + 1, c, 0] // Flatten
```

```
{c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15,
 c16, c17, c18, c19, c20, c21, c22, c23, c24, c25, c26, c27, c28, c29,
 c30, c31, c32, c33, c34, c35, c36, c37, c38, c39, c40, c41, c42, c43, c44,
 c45, c46, c47, c48, c49, c50, c51, c52, c53, c54, c55, c56, c57, c58, c59}
```

The coefficients are parameters of the multirate system.

### Draw Schematic of Classic Realization

We choose a filter realization known as *direct form FIR*.

`DirectFormFIRFilterSchematic` creates the schematic of the Direct Form FIR filter.

```
{classicFIRSchematic, inpCoords, outCoords} =
  DirectFormFIRFilterSchematic[parameterSymbols];
```

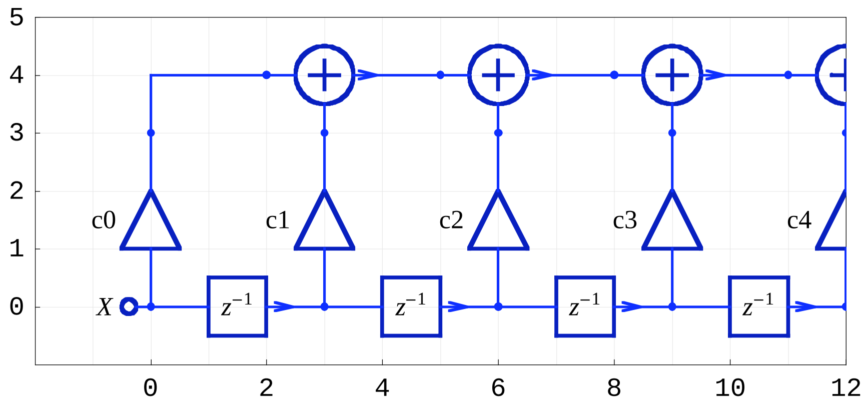
The coordinates of input and output are returned by `DirectFormFIRFilterSchematic`.

You can add input and output to form the system:

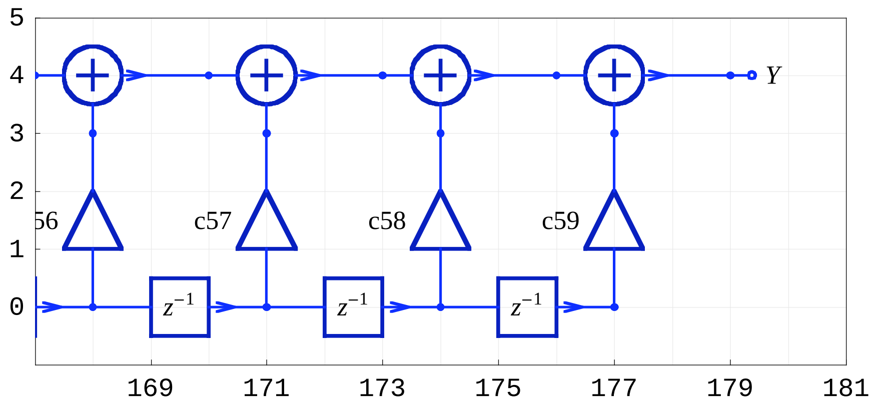
```
classicFIR = Join[
   classicFIRSchematic,
   {{"Input", First[inpCoords], X}},
   {{"Output", First[outCoords], Y}}
  ];
```

`PlotRange` refines the drawing to show a portion of the system:

```
ShowSchematic[classicFIR, PlotRange → {{-2, 12}, {-1, 5}}];
```



```
ShowSchematic[classicFIR,
  PlotRange → {{numberOfStages * 3 - 10, numberOfStages * 3 + 4}, {-1, 5}}];
```



## Transfer Function of Classic Realization

*SchematicSolver*'s function `DiscreteSystemTransferFunction` computes the filter transfer function:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[classicFIR];
classicTF = tfMatrix[[1, 1]] // Together
```

$$\frac{1}{z^{59}}(c59 + c58\, z + c57\, z^2 + c56\, z^3 + c55\, z^4 + c54\, z^5 + c53\, z^6 + c52\, z^7 + c51\, z^8 + c50\, z^9 + c49\, z^{10} + c48\, z^{11} + c47\, z^{12} + c46\, z^{13} + c45\, z^{14} + c44\, z^{15} + c43\, z^{16} + c42\, z^{17} + c41\, z^{18} + c40\, z^{19} + c39\, z^{20} + c38\, z^{21} + c37\, z^{22} + c36\, z^{23} + c35\, z^{24} + c34\, z^{25} + c33\, z^{26} + c32\, z^{27} + c31\, z^{28} + c30\, z^{29} + c29\, z^{30} + c28\, z^{31} + c27\, z^{32} + c26\, z^{33} + c25\, z^{34} + c24\, z^{35} + c23\, z^{36} + c22\, z^{37} + c21\, z^{38} + c20\, z^{39} + c19\, z^{40} + c18\, z^{41} + c17\, z^{42} + c16\, z^{43} + c15\, z^{44} + c14\, z^{45} + c13\, z^{46} + c12\, z^{47} + c11\, z^{48} + c10\, z^{49} + c9\, z^{50} + c8\, z^{51} + c7\, z^{52} + c6\, z^{53} + c5\, z^{54} + c4\, z^{55} + c3\, z^{56} + c2\, z^{57} + c1\, z^{58} + c0\, z^{59})$$

Coefficient numeric values can be computed with filter design software such as *Digital Image Processing* – a *Mathematica* application package:

```
Needs["ImageProcessing`"];
parameterValues =
  EquirippleFilter[numberOfStages + 1, {0.0, 0.04, 0.1, 0.5}, {1., 0.}];
```

Assume symmetric coefficients:

```
parameterValues = (parameterValues + Take[parameterValues,
      {Length[parameterValues], 1, -1}]) / 2;
```

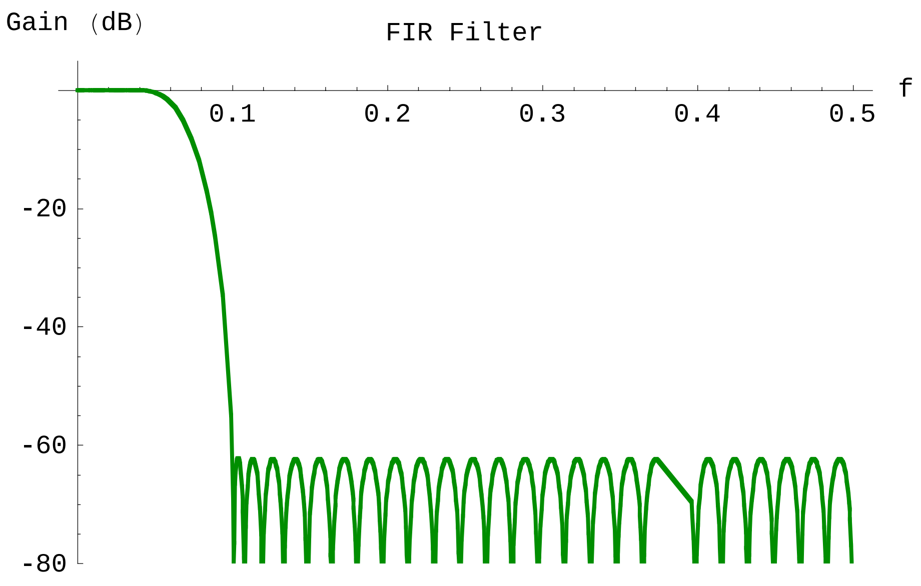
Here is the parameter substitution list:

```
parameterSubstitution = parameterSymbols → parameterValues // Thread
```

```
{c0 → 0.000402511, c1 → -0.0000577266, c2 → -0.000352446, c3 → -0.000809402,
 c4 → -0.00132471, c5 → -0.0017228, c6 → -0.00178407, c7 → -0.00130033,
 c8 → -0.00014924, c9 → 0.00163053, c10 → 0.00378699, c11 → 0.0058506, c12 → 0.00719502,
 c13 → 0.00716018, c14 → 0.00522332, c15 → 0.0011915, c16 → -0.00464914,
 c17 → -0.0114503, c18 → -0.0178415, c19 → -0.0220959, c20 → -0.0224119,
 c21 → -0.0172665, c22 → -0.00577023, c23 → 0.0120502, c24 → 0.0351138, c25 → 0.0613364,
 c26 → 0.0878783, c27 → 0.111555, c28 → 0.129345, c29 → 0.138885, c30 → 0.138885,
 c31 → 0.129345, c32 → 0.111555, c33 → 0.0878783, c34 → 0.0613364, c35 → 0.0351138,
 c36 → 0.0120502, c37 → -0.00577023, c38 → -0.0172665, c39 → -0.0224119,
 c40 → -0.0220959, c41 → -0.0178415, c42 → -0.0114503, c43 → -0.00464914,
 c44 → 0.0011915, c45 → 0.00522332, c46 → 0.00716018, c47 → 0.00719502,
 c48 → 0.0058506, c49 → 0.00378699, c50 → 0.00163053, c51 → -0.00014924,
 c52 → -0.00130033, c53 → -0.00178407, c54 → -0.0017228, c55 → -0.00132471,
 c56 → -0.000809402, c57 → -0.000352446, c58 → -0.0000577266, c59 → 0.000402511}
```

The corresponding magnitude characteristic, gain in decibels, is

```
DiscreteSystemMagnitudeResponsePlot[
 classicTF /. parameterSubstitution, {0, 0.5}, PlotRange → {-80, 5},
 AxesLabel → {"f", "Gain (dB)"}, PlotLabel → "FIR Filter"];
```



## 11.5. Polyphase Decimation FIR Filter

### Draw Subsystem Schematics of Polyphase Realization

Consider a multirate system with the downsampling factor $M = 5$ and generate parameter names for the polyphase FIR filter:

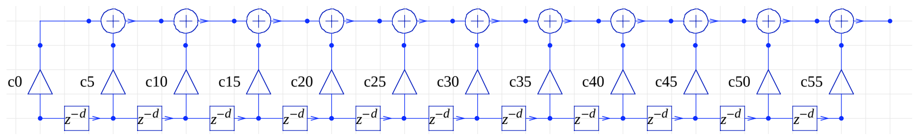
```
M = 5;
parameterSymbols1 = Take[parameterSymbols, {1, Length[parameterSymbols], M}]
parameterSymbols2 = Take[parameterSymbols, {2, Length[parameterSymbols], M}]
parameterSymbols3 = Take[parameterSymbols, {3, Length[parameterSymbols], M}]
parameterSymbols4 = Take[parameterSymbols, {4, Length[parameterSymbols], M}]
parameterSymbols5 = Take[parameterSymbols, {5, Length[parameterSymbols], M}]

{c0, c5, c10, c15, c20, c25, c30, c35, c40, c45, c50, c55}

{c1, c6, c11, c16, c21, c26, c31, c36, c41, c46, c51, c56}

{c2, c7, c12, c17, c22, c27, c32, c37, c42, c47, c52, c57}

{c3, c8, c13, c18, c23, c28, c33, c38, c43, c48, c53, c58}

{c4, c9, c14, c19, c24, c29, c34, c39, c44, c49, c54, c59}
```

We specify some draw options to better present the systems:

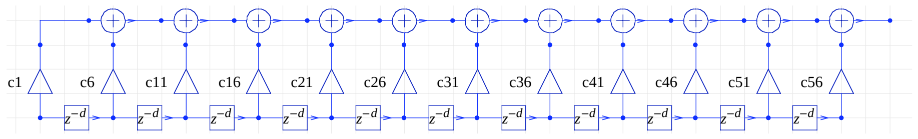
```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];
```

Here are schematic specifications for the 5 FIR filters:

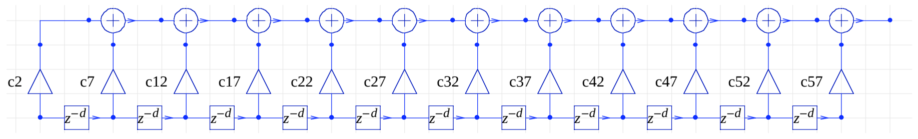
```
{classicFIRschematic1, inpCoords1, outCoords1} =
  DirectFormFIRFilterSchematic[parameterSymbols1, {2, 0}, DelayElementValue → d];
ShowSchematic[classicFIRschematic1, FontSize → 6, Frame → False];
```



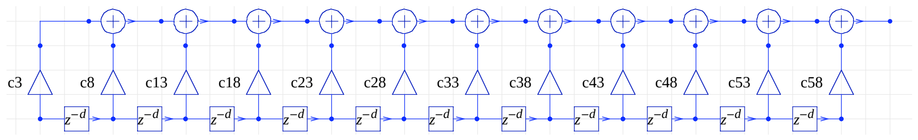
```
{classicFIRschematic2, inpCoords2, outCoords2} =
  DirectFormFIRFilterSchematic[parameterSymbols2, {2, 6}, DelayElementValue → d];
ShowSchematic[classicFIRschematic2, FontSize → 6, Frame → False];
```



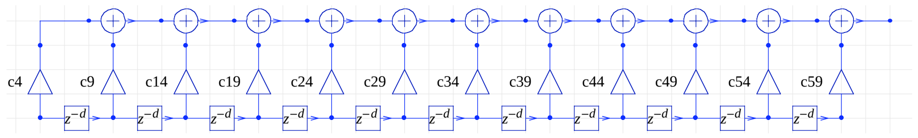
```
{classicFIRschematic3, inpCoords3, outCoords3} =
  DirectFormFIRFilterSchematic[parameterSymbols3, {2, 12}, DelayElementValue → d];
ShowSchematic[classicFIRschematic3, FontSize → 6, Frame → False];
```



```
{classicFIRschematic4, inpCoords4, outCoords4} =
  DirectFormFIRFilterSchematic[parameterSymbols4, {2, 18}, DelayElementValue → d];
ShowSchematic[classicFIRschematic4, FontSize → 6, Frame → False];
```



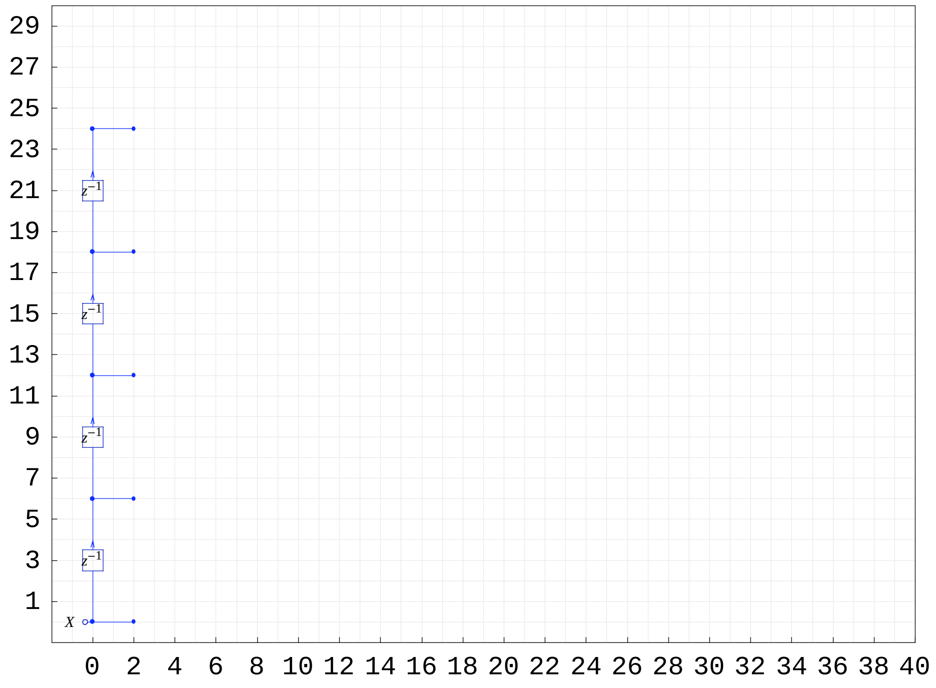
```
{classicFIRschematic5, inpCoords5, outCoords5} =
  DirectFormFIRFilterSchematic[parameterSymbols5, {2, 24}, DelayElementValue → d];
ShowSchematic[classicFIRschematic5, FontSize → 6, Frame → False];
```



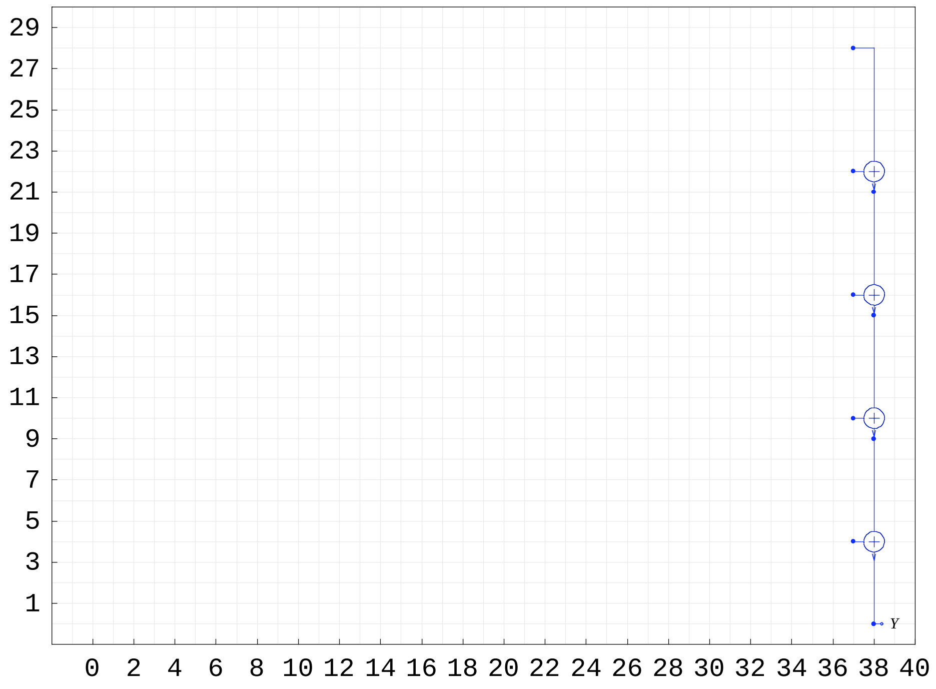
## Draw Schematic of Polyphase Realization

We draw the polyphase FIR filter by combining the FIR schematics and by adding the input and the output parts:

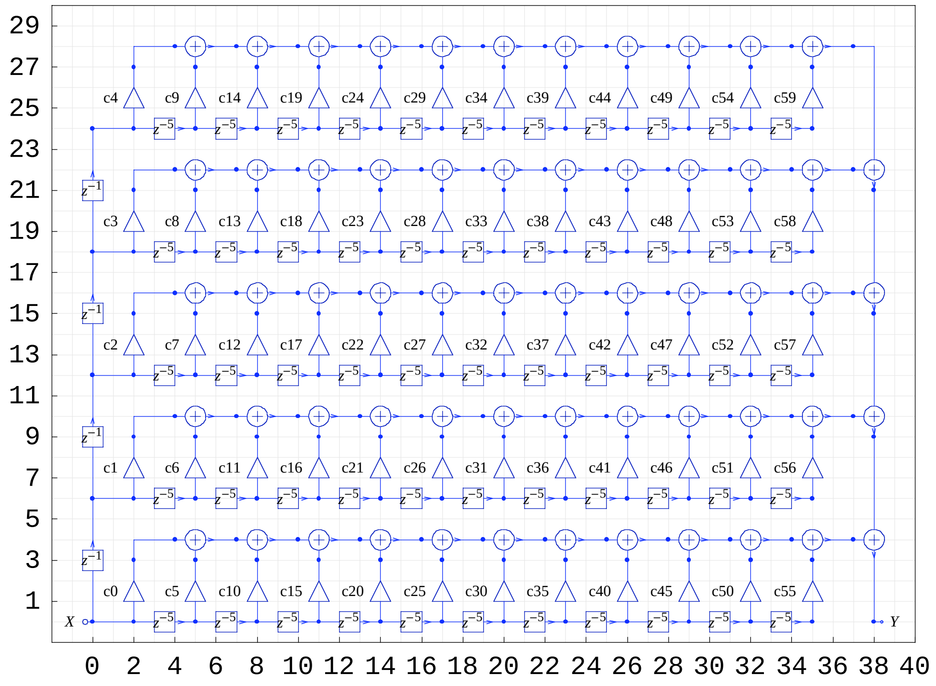
```
inputPolyphaseSchematic = {{"Input", {0, 0}, X},
   {"Delay", {{0, 0}, {0, 6}}, 1}, {"Delay", {{0, 6}, {0, 12}}, 1},
   {"Delay", {{0, 12}, {0, 18}}, 1}, {"Delay", {{0, 18}, {0, 24}}, 1},
   {"Line",  {{0, 24}, {2, 24}}}, {"Line",  {{0, 18}, {2, 18}}},
   {"Line",  {{0, 12}, {2, 12}}},
   {"Line",  {{0, 6}, {2, 6}}},
   {"Line",  {{0, 0}, {2, 0}}}};
 ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



```
outputPolyphaseSchematic = {
   {"Adder", {{37, 22}, {38, 21}, {39, 22}, {37, 28}}, {1, 2, 0, 1}},
   {"Adder", {{37, 16}, {38, 15}, {39, 16}, {38, 21}}, {1, 2, 0, 1}},
   {"Adder", {{37, 10}, {38, 9}, {39, 10}, {38, 15}}, {1, 2, 0, 1}},
   {"Adder", {{37, 4}, {38, 0}, {39, 4}, {38, 9}}, {1, 2, 0, 1}},
   {"Output", {38, 0}, Y}};
ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



```
polyphaseFIR = Join[
    inputPolyphaseSchematic,
    outputPolyphaseSchematic,
    classicFIRschematic1,
    classicFIRschematic2,
    classicFIRschematic3,
    classicFIRschematic4,
    classicFIRschematic5] /. d → 5;
 ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



c0 c5 c10 c15 c20 c25 c30 c35 c40 c45 c50 c55
c1 c6 c11 c16 c21 c26 c31 c36 c41 c46 c51 c56
c2 c7 c12 c17 c22 c27 c32 c37 c42 c47 c52 c57
c3 c8 c13 c18 c23 c28 c33 c38 c43 c48 c53 c58
c4 c9 c14 c19 c24 c29 c34 c39 c44 c49 c54 c59
$z^{-5}$
$z^{-1}$
X
Y

## Transfer Function of Polyphase Realization

*SchematicSolver*'s function `DiscreteSystemTransferFunction` computes the filter transfer function:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[polyphaseFIR];
polyphaseTF = tfMatrix[[1, 1]] // Together
```

$$\frac{1}{z^{59}}\left(c59 + c58\,z + c57\,z^{2} + c56\,z^{3} + c55\,z^{4} + c54\,z^{5} + c53\,z^{6} + c52\,z^{7} + c51\,z^{8} + c50\,z^{9} + c49\,z^{10} + c48\,z^{11} + c47\,z^{12} + c46\,z^{13} + c45\,z^{14} + c44\,z^{15} + c43\,z^{16} + c42\,z^{17} + c41\,z^{18} + c40\,z^{19} + c39\,z^{20} + c38\,z^{21} + c37\,z^{22} + c36\,z^{23} + c35\,z^{24} + c34\,z^{25} + c33\,z^{26} + c32\,z^{27} + c31\,z^{28} + c30\,z^{29} + c29\,z^{30} + c28\,z^{31} + c27\,z^{32} + c26\,z^{33} + c25\,z^{34} + c24\,z^{35} + c23\,z^{36} + c22\,z^{37} + c21\,z^{38} + c20\,z^{39} + c19\,z^{40} + c18\,z^{41} + c17\,z^{42} + c16\,z^{43} + c15\,z^{44} + c14\,z^{45} + c13\,z^{46} + c12\,z^{47} + c11\,z^{48} + c10\,z^{49} + c9\,z^{50} + c8\,z^{51} + c7\,z^{52} + c6\,z^{53} + c5\,z^{54} + c4\,z^{55} + c3\,z^{56} + c2\,z^{57} + c1\,z^{58} + c0\,z^{59}\right)$$

The transfer functions of the classic realization and the polyphase realization should be the same:

```
SameQ[classicTF, polyphaseTF]
```

```
True
```

## 11.6. Spectra of Decimated Signals

### Processing with Classic FIR Filter

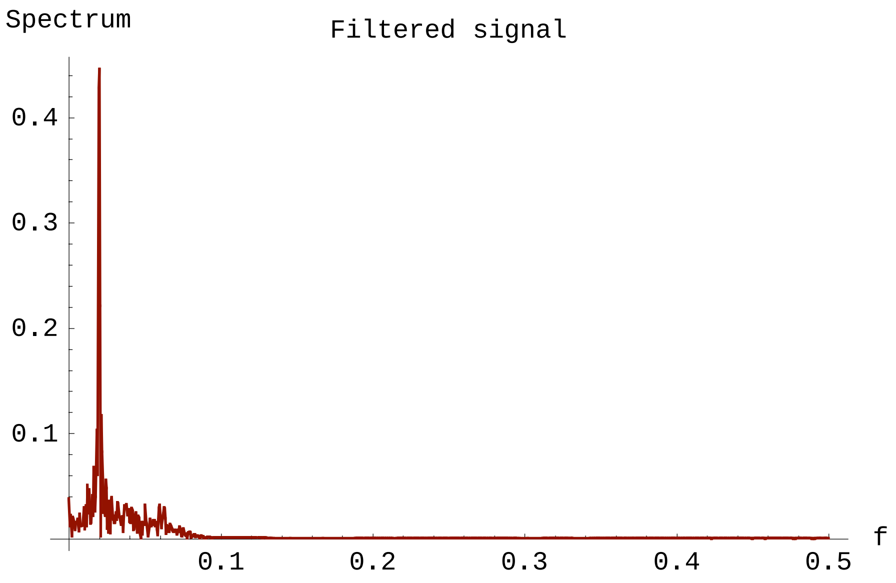
In order to avoid aliasing in multirate systems, it is necessary to bandlimit the spectrum of the input signal, before downsampling, to a frequency below $\frac{1}{2M}$. This is accomplished with a lowpass filter that we implement as `classicFIR`.

`DiscreteSystemSimulation` finds the filtered signal at the output of `classicFIR`:

```
outClassicSeq =
  DiscreteSystemSimulation[classicFIR /. parameterSubstitution, compositeSeq];
```

Here is the spectrum of the filtered composite signal:

```
SequenceFourierTransformMagnitudePlot[outClassicSeq, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Filtered signal"];
```

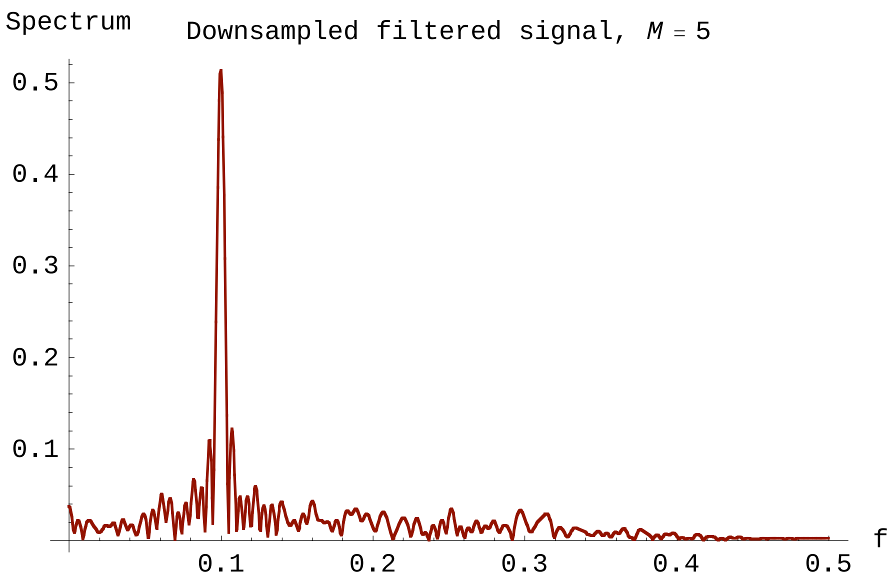


`SequenceFourierTransformMagnitudePlot` shows only one strong peak at $f_1 = 0.02$.

Here is the spectrum of the downsampled filtered composite signal:

```
M = 5;
downOutClassicSeq = DownsampleSequence[outClassicSeq, M];
```

```
SequenceFourierTransformMagnitudePlot[downOutClassicSeq, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Downsampled filtered signal, M = 5"];
```



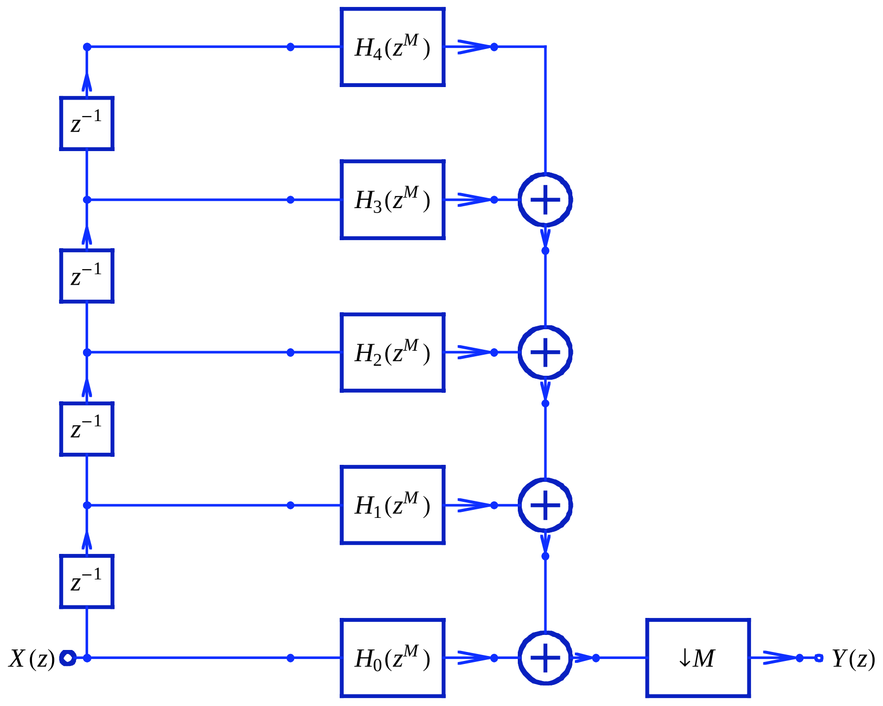
`SequenceFourierTransformMagnitudePlot` shows only one strong peak at $5\ f_1 = 0.1$.

The filtered composite signal is bandlimited to the frequency $\frac{1}{2M}$, so the downsampled signal spectrum does not contain folded (aliased) components.

## 11.7. Efficient Decimation FIR Filter

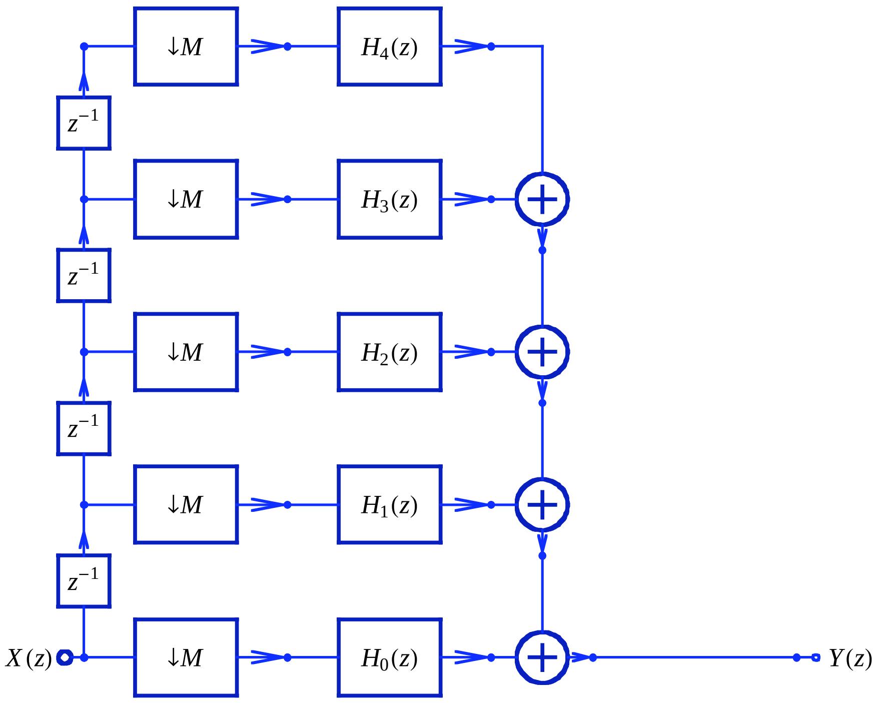
Decimation system can be implemented by using the classic polyphase realization

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleDefault];

SchematicSolverFigureMultirateDownsamplingClassic;
 ShowSchematic[%, GridLines → None, Frame → False];
```



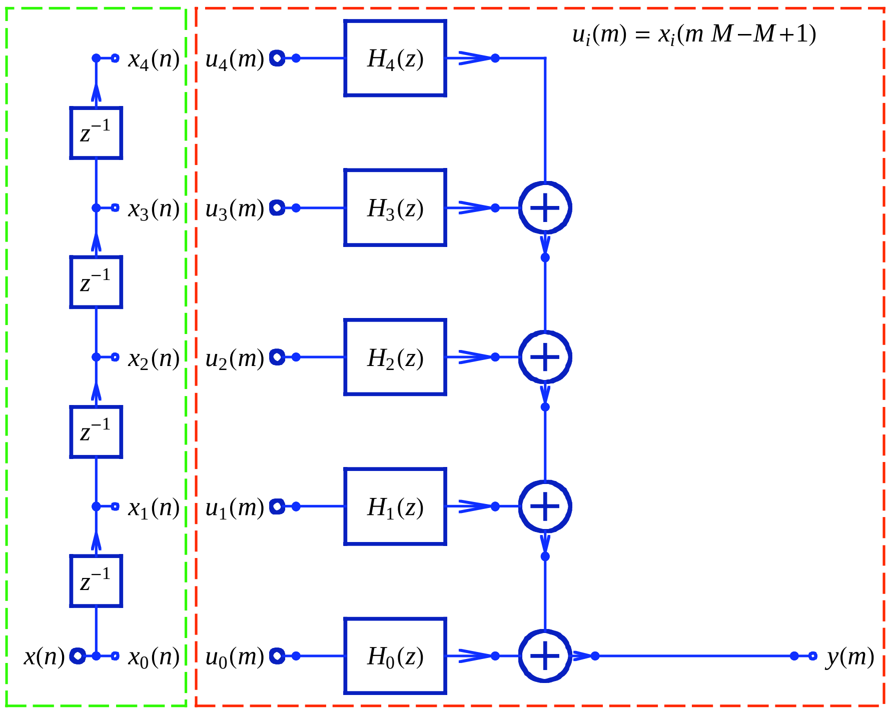
More efficient implementation can be achieved by using the decimation identity as follows:

```
SchematicSolverFigureMultirateDownsamplingEfficient;
 ShowSchematic[%, GridLines → None, Frame → False];
```



Here is a realization of the efficient multirate system that is implemented in *SchematicSolver*:

```
SchematicSolverFigureMultirateDownsamplingImplemented;
 ShowSchematic[%, GridLines → None, Frame → False];
```



A system with a small number of multiplications is said to be the *efficient system* if the multiplication is the most time-consuming operation, and if time is the most critical resource. A figure of merit should be used to quantify the computational complexity. In this section, `FigureOfMeritOutput` is the figure of merit defined as the number of multiplications per output sample. For the classic implementation, it can be computed as a product of the number of multiplications and the downsampling factor:

```
FigureOfMeritOutputClassic = Length[parameterSymbols] * M
```

```
300
```

`FigureOfMeritOutputEfficient`, the figure of merit of the efficient implementation, is equal to the number of multiplications:

```
FigureOfMeritOutputEfficient = Length[parameterSymbols]
```

```
60
```

`FigureOfMeritOutputEfficient` is `M` times smaller than `FigureOfMeritOutputClassic`.

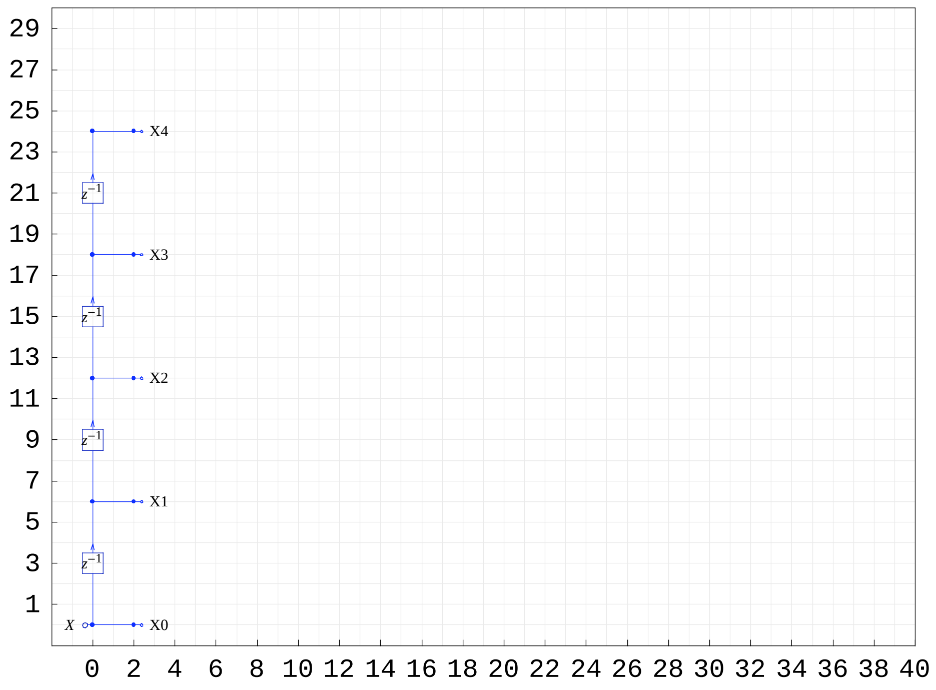
## 11.8. Implementation of Efficient Decimation

### Processing with Input Decimation Subsystem

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];
```

Here is the input decimation subsystem:

```
inputDecimationSubsystem = Join[
   inputPolyphaseSchematic,
   {{"Output", {2, 0}, X0}},
   {{"Output", {2, 6}, X1}},
   {{"Output", {2, 12}, X2}},
   {{"Output", {2, 18}, X3}},
   {{"Output", {2, 24}, X4}}];
 ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



`DiscreteSystemSimulation` finds the output of `inputDecimationSubsystem`:

```
outDecSeq = DiscreteSystemSimulation[inputDecimationSubsystem, compositeSeq];
```

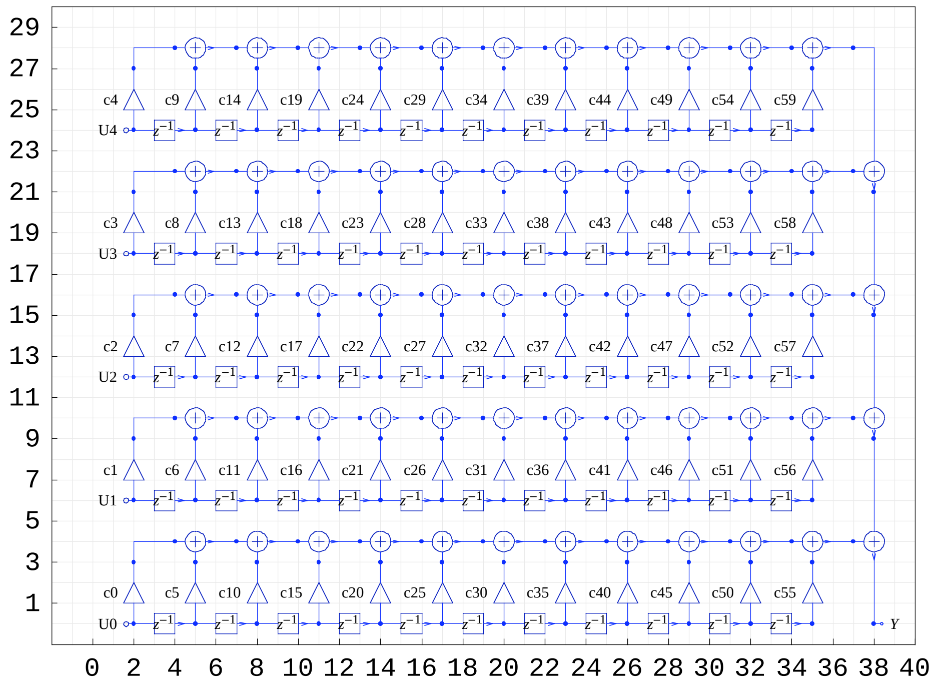
`DownsampleSequence` implements the downsampler:

```
downOutDecSeq = DownsampleSequence[outDecSeq, M];
```

## Processing with Polyphase Decimation Subsystem

Here is the polyphase decimation subsystem:

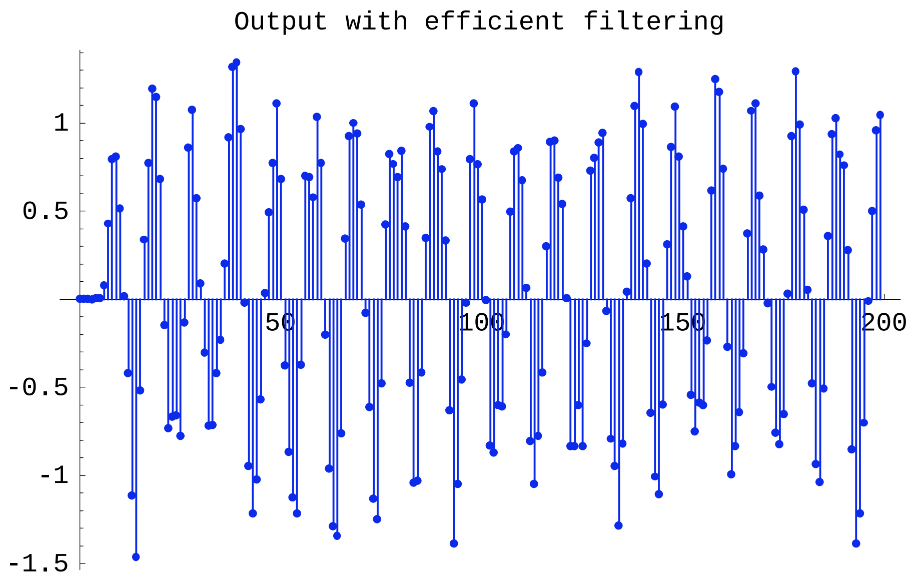
```
polyphaseDecimationSubsystem = Join[
    outputPolyphaseSchematic,
    classicFIRschematic1,
    classicFIRschematic2,
    classicFIRschematic3,
    classicFIRschematic4,
    classicFIRschematic5,
    {{"Input", inpCoords1[[1]], U0}},
    {{"Input", inpCoords2[[1]], U1}},
    {{"Input", inpCoords3[[1]], U2}},
    {{"Input", inpCoords4[[1]], U3}},
    {{"Input", inpCoords5[[1]], U4}}
   ] /. d → 1;
ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



`DiscreteSystemSimulation` finds the output of `polyphaseDecimationSubsystem`:

```
outDecimationSeq = DiscreteSystemSimulation[
   polyphaseDecimationSubsystem /. parameterSubstitution, downOutDecSeq];
```

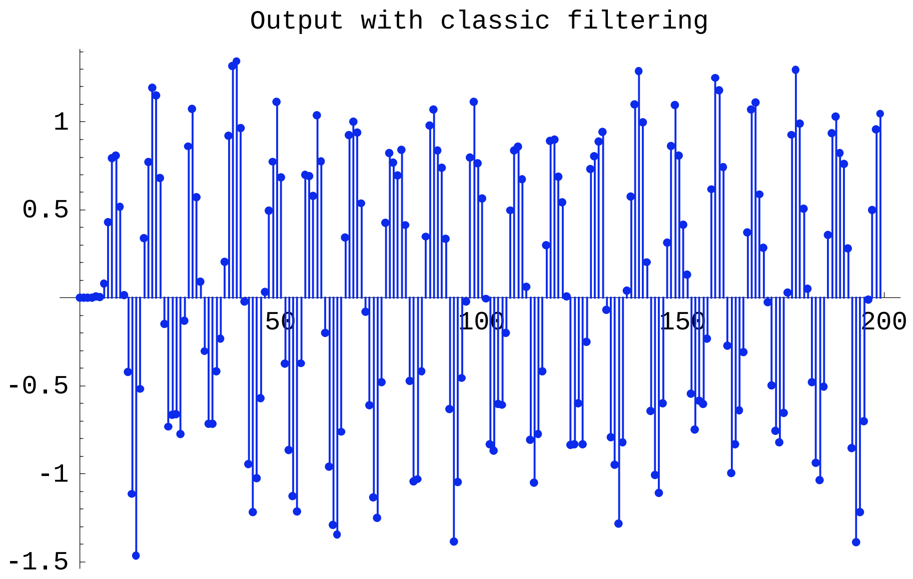
```
SequencePlot[outDecimationSeq,
  PlotLabel -> "Output with efficient filtering"];
```



Here is the output signal of the classic realization:

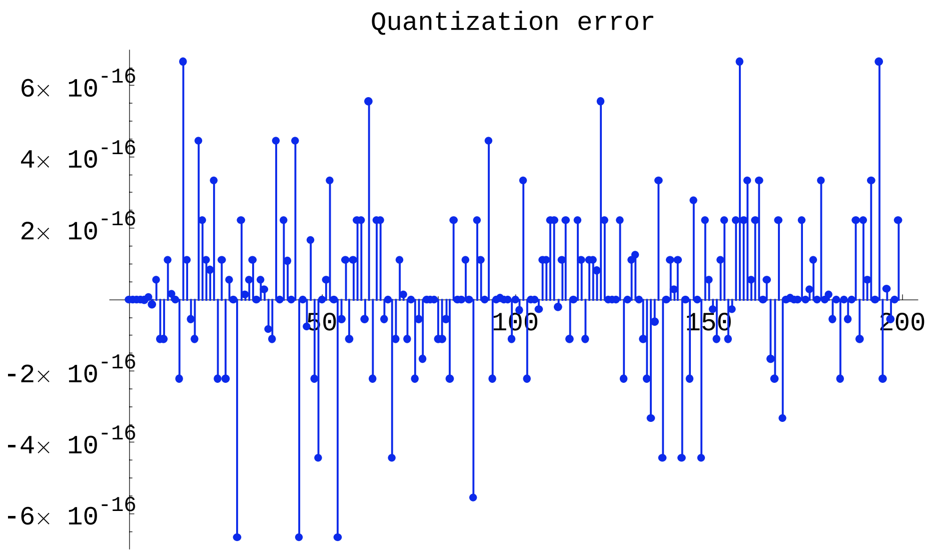
```
M = 5;
```

```
SequencePlot[DownsampleSequence[outClassicSeq, M],
  PlotLabel -> "Output with classic filtering"];
```



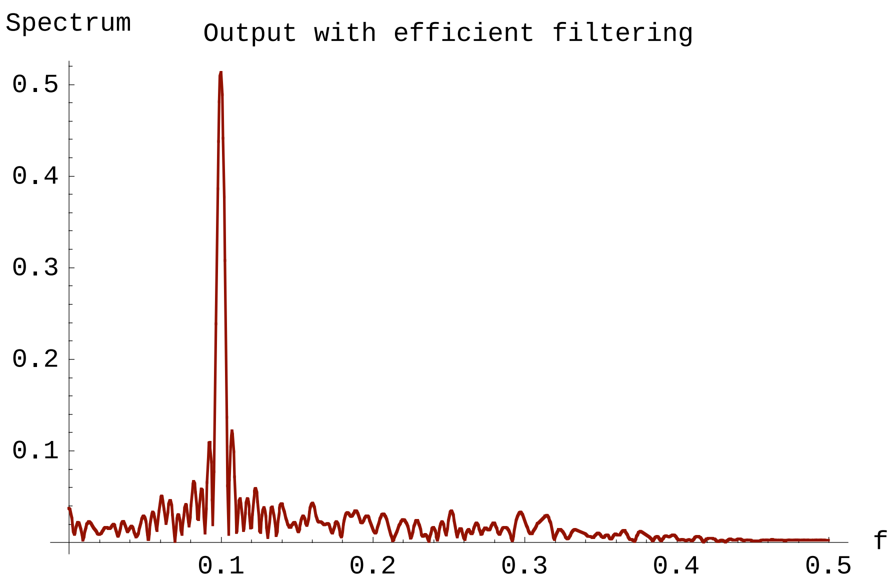
The two output signals are practically the same; the difference of the signals is attributed to the quantization error:

```
SequencePlot[outDecimationSeq - DownsampleSequence[outClassicSeq, M],
  PlotLabel -> "Quantization error"];
```



Here is the spectrum of the output signal:

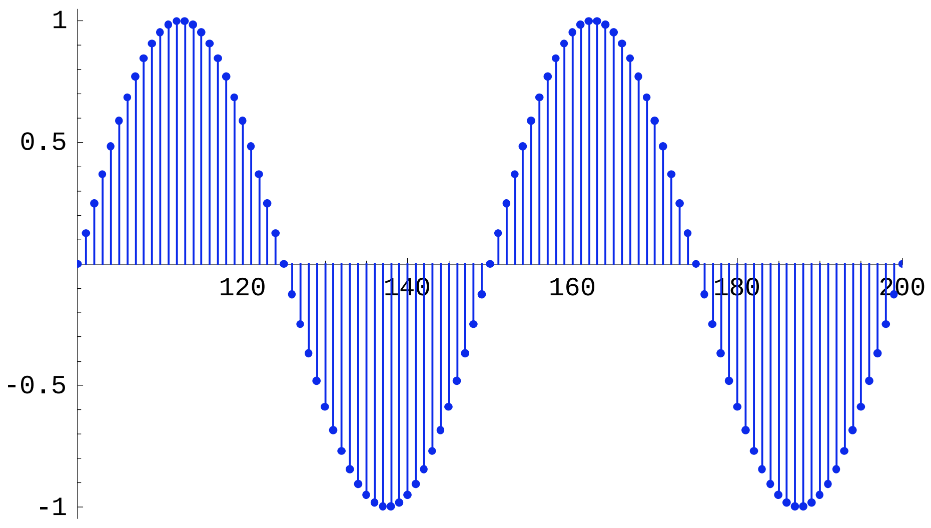
```
SequenceFourierTransformMagnitudePlot[outDecimationSeq, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Output with efficient filtering"];
```



## 11.9. Spectra of Upsampled Signals and Signal Reconstruction

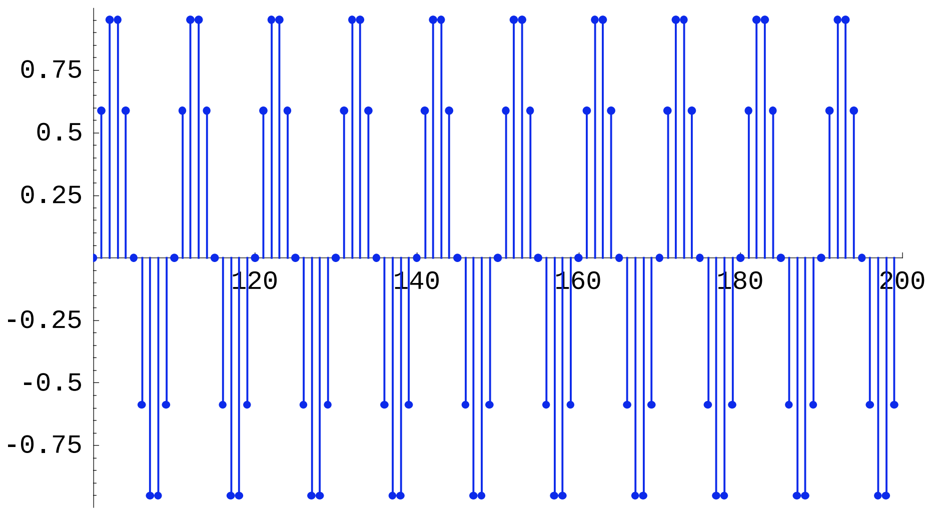
Here is a portion of the first sinusoidal signal of $f_1 = 0.02$:

```
SequencePlot[sineSeq1, PlotRange → {{100, 200}, All}];
```



The corresponding downsampled signal is

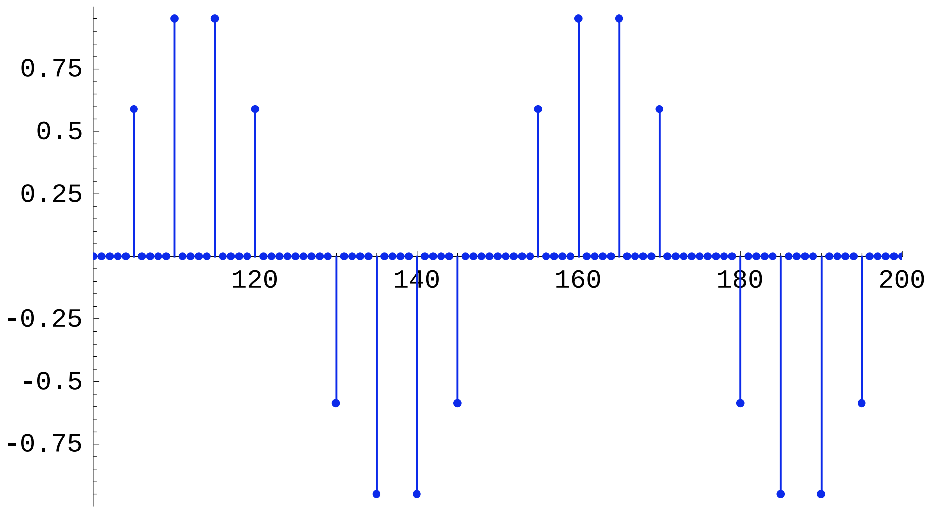
```
SequencePlot[downSineSeq1, PlotRange → {{100, 200}, All}];
```



If we upsample the downsampled signal, we do not obtain the original signal:

```
M = 5;
L = M;
```

```
updownSineSeq1 = UpsampleSequence[downSineSeq1, L];
SequencePlot[%, PlotRange → {{100, 200}, All}];
```

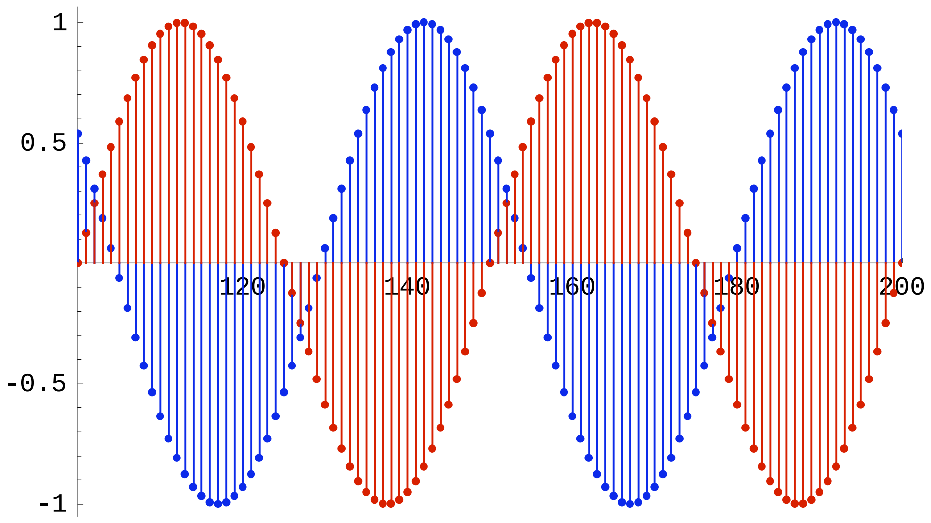


The upsampled signal `upDownSineSeq1` can be processed with the lowpass filter `classicFIR` to reconstruct the original signal.

```
reconstructedSeq =
  DiscreteSystemSimulation[classicFIR /. parameterSubstitution, upDownSineSeq1];
```

The reconstructed signal is delayed due to processing. In addition, it is of a smaller amplitude by the factor $\frac{1}{L} = \frac{1}{5}$. Here is the plot of the reconstructed signal (blue) and the original signal (red):

```
SequencePlot[MultiplexSequence[5 * reconstructedSeq, sineSeq1],
  PlotRange → {{100, 200}, All}];
```



The equivalent phase shift, which takes into account the processing delay, is

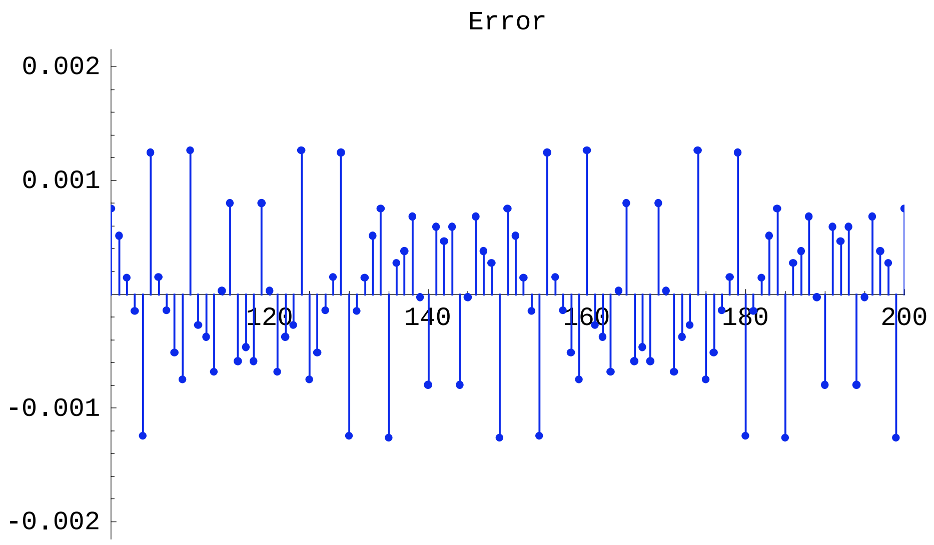
```
phaseShift = -numberOfStages * frequency1 * Pi;
```

Let us generate a delayed sinusoidal sequence

```
delayedSineSeq1 =
  amplitude1 * UnitSineSequence[numberOfSamples, frequency1, phase1 + phaseShift];
```

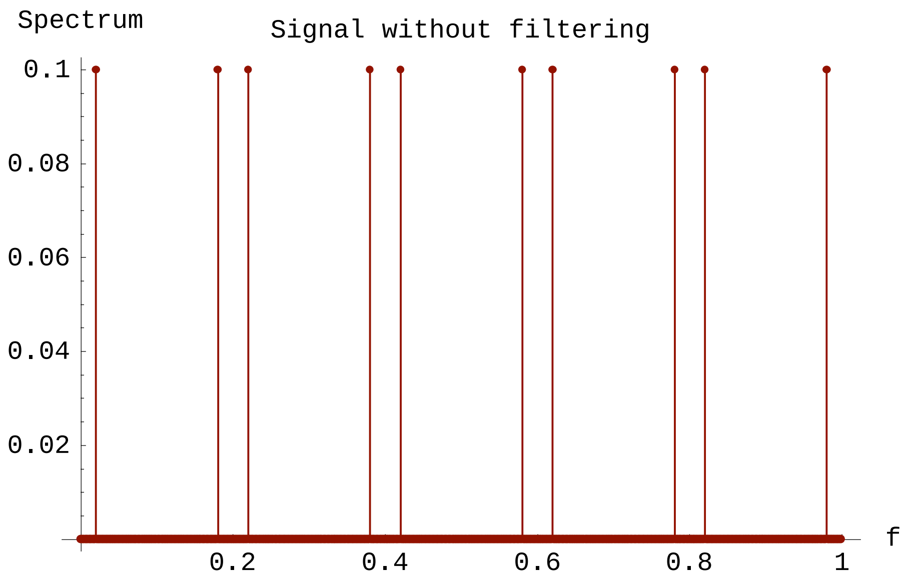
The reconstructed signal and the delayed signal are practically the same:

```
SequencePlot[5 * reconstructedSeq - delayedSineSeq1,
  PlotRange → {{100, 200}, Automatic}, PlotLabel → "Error"];
```



Here is the discrete spectrum of the signal `upDownSineSeq1`:

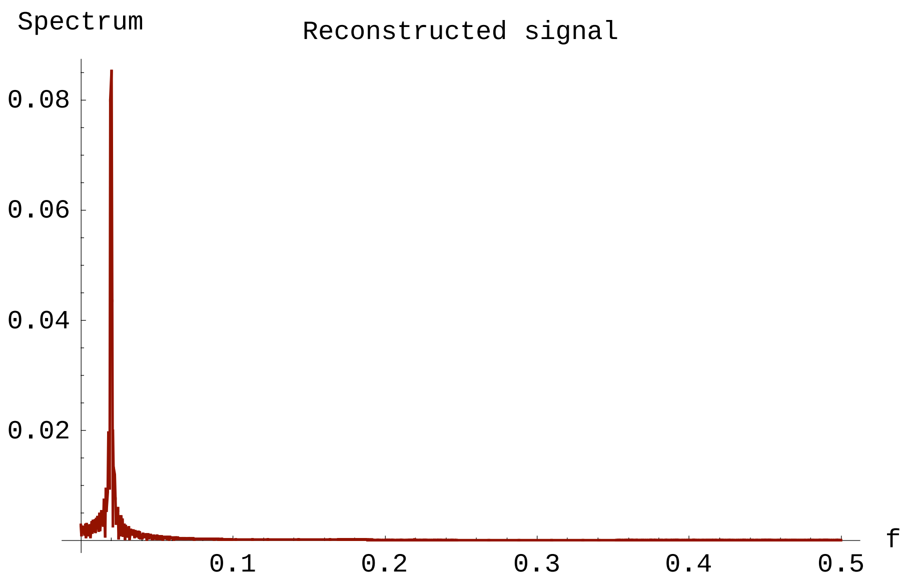
```
SequenceDiscreteFourierTransformMagnitudePlot[upDownSineSeq1,
 AxesLabel → { "f", "Spectrum"},
 PlotLabel → "Signal without filtering", PlotRange → All];
```



Note that the spectral components of the upsampled signal are $\frac{1}{L} = \frac{1}{5}$ smaller in amplitude.

Here is the spectrum of the reconstructed signal:

```
SequenceFourierTransformMagnitudePlot[reconstructedSeq, {0, 0.5},
 AxesLabel → { "f", "Spectrum"}, PlotLabel → "Reconstructed signal"];
```



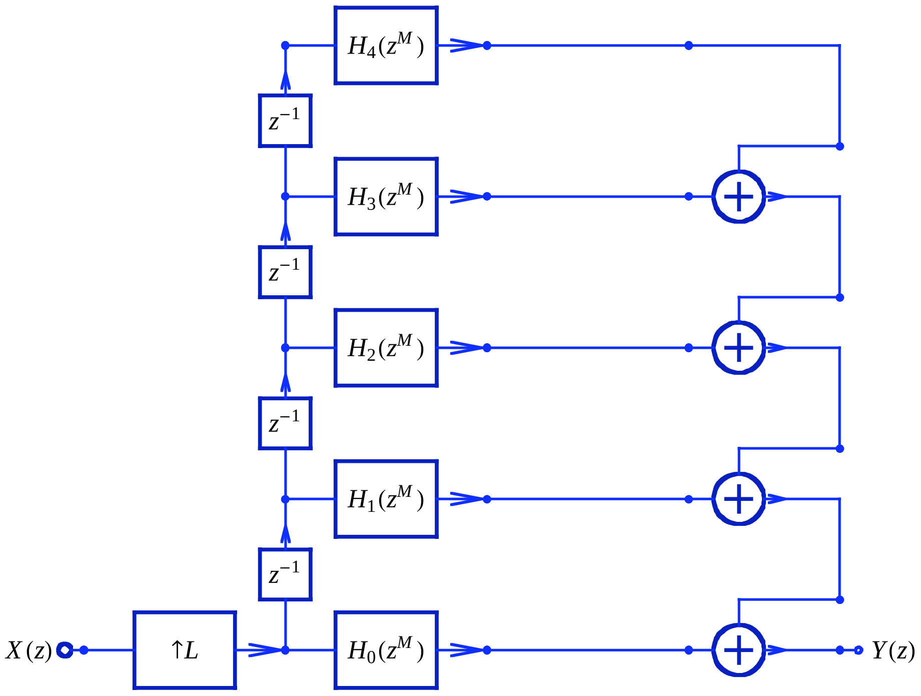
The process of upsampling introduces the replicas of the main spectra. This is called *imaging*. In order to remove the unwanted image spectra, we need a lowpass filter immediately after upsampling. This filter is called an *anti-imaging filter*, also referred to as an *interpolation filter*.

## 11.10. Efficient Interpolation FIR Filter

```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleDefault];
```

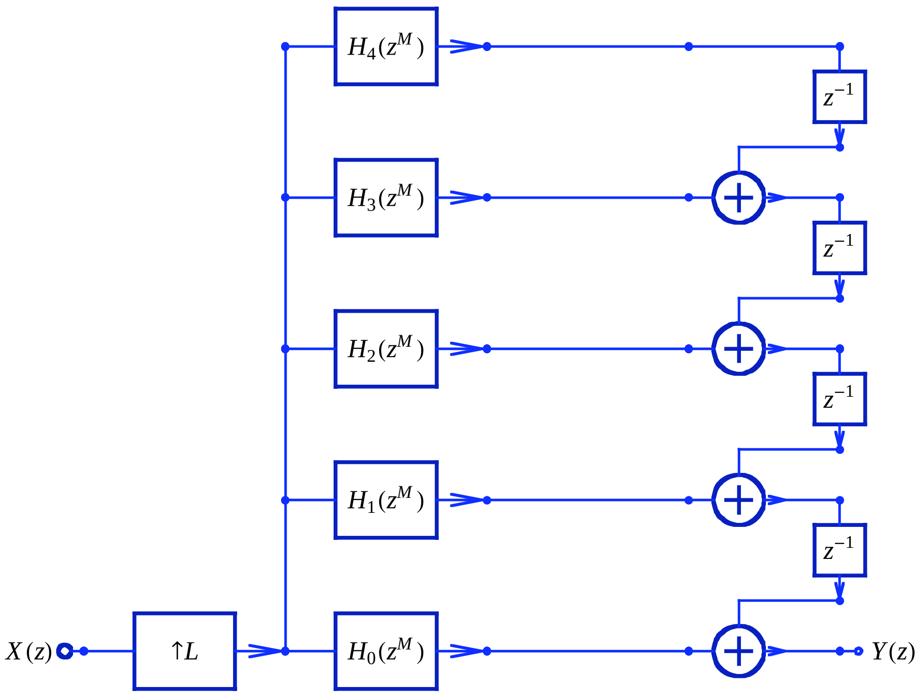
Interpolation system can be implemented by using the classic polyphase realization

```
SchematicSolverFigureMultirateUpsamplingClassic;
 ShowSchematic[%, GridLines → None, Frame → False];
```



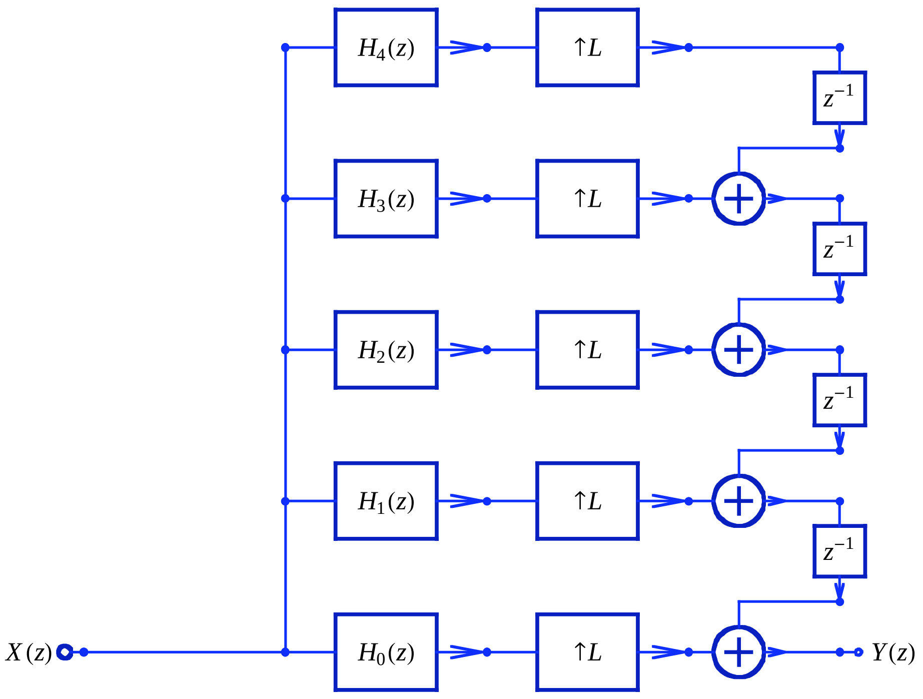
Here is the transposed classic polyphase realization:

```
SchematicSolverFigureMultirateUpsamplingTransposed;
 ShowSchematic[%, GridLines → None, Frame → False];
```



More efficient implementation can be achieved using the interpolation identity as follows:

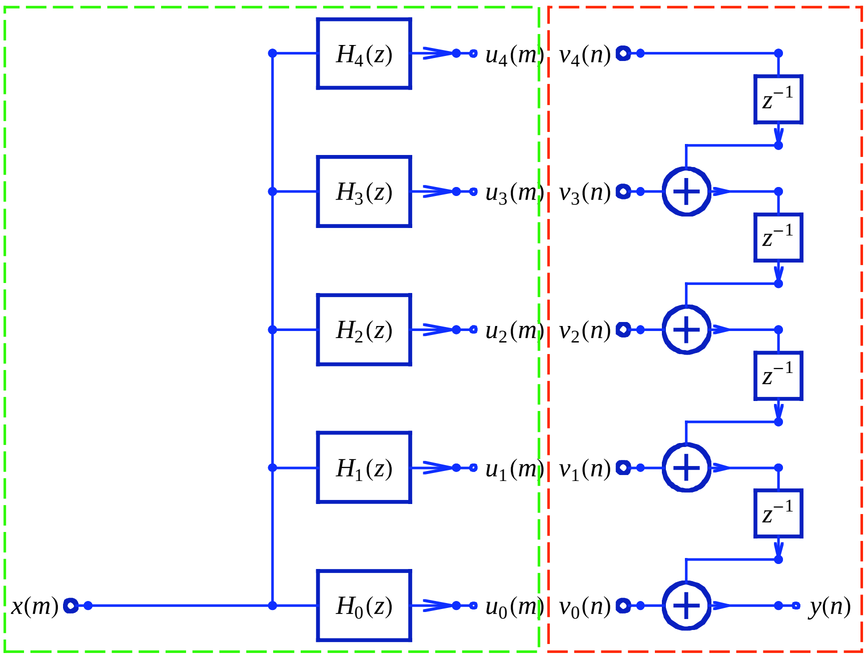
```
SchematicSolverFigureMultirateUpsamplingEfficient;
 ShowSchematic[%, GridLines → None, Frame → False];
```



Here is a realization of the efficient multirate system that is implemented in *SchematicSolver*:

```
SchematicSolverFigureMultirateUpsamplingImplemented;
 ShowSchematic[%, GridLines → None, Frame → False];
```

$$v(n) = \begin{cases} u(m) & \text{for } n = m\,L - L + 1 \\ 0 & \text{otherwise} \end{cases}$$



A system with a small number of multiplications is said to be the *efficient system* if the multiplication is the most time-consuming operation, and if time is the most critical resource. A figure of merit should be used to quantify the computational complexity. In this section, `FigureOfMeritInput` is the figure of merit defined as the number of multiplications per input sample. For the classic implementation, it can be computed as a product of the number of multiplications and the upsampling factor:

```
FigureOfMeritInputClassic = Length[parameterSymbols] * L
```

```
300
```

`FigureOfMeritInputEfficient`, the figure of merit of the efficient implementation, is equal to the number of multiplications:

```
FigureOfMeritInputEfficient = Length[parameterSymbols]
```

```
60
```

`FigureOfMeritInputEfficient` is `L` times smaller than `FigureOfMeritInputClassic`.

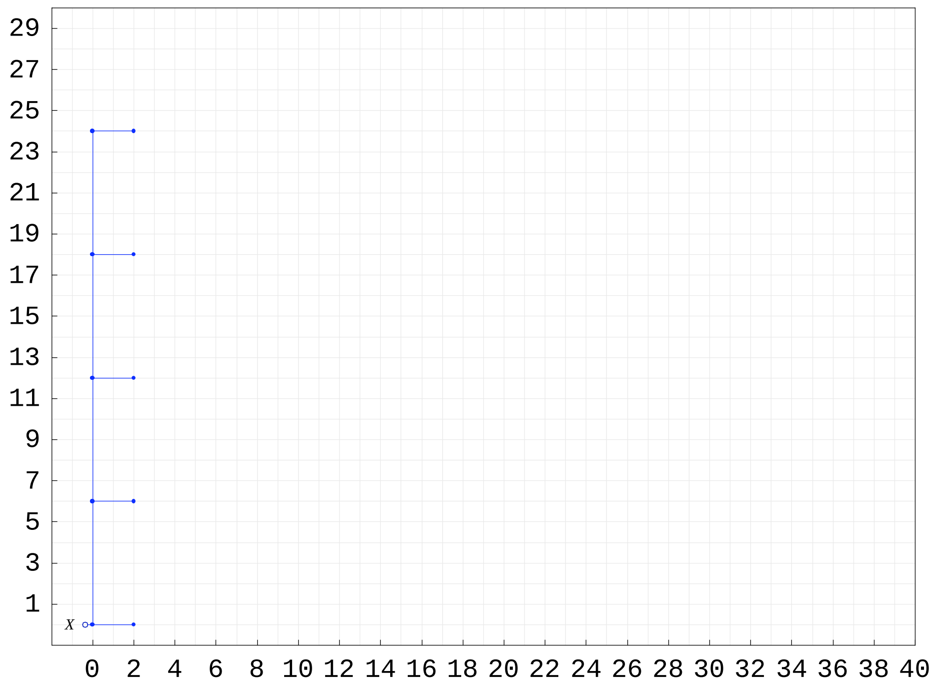
## 11.11. Implementation of Efficient Interpolation

### Draw Interpolation Transposed Filter

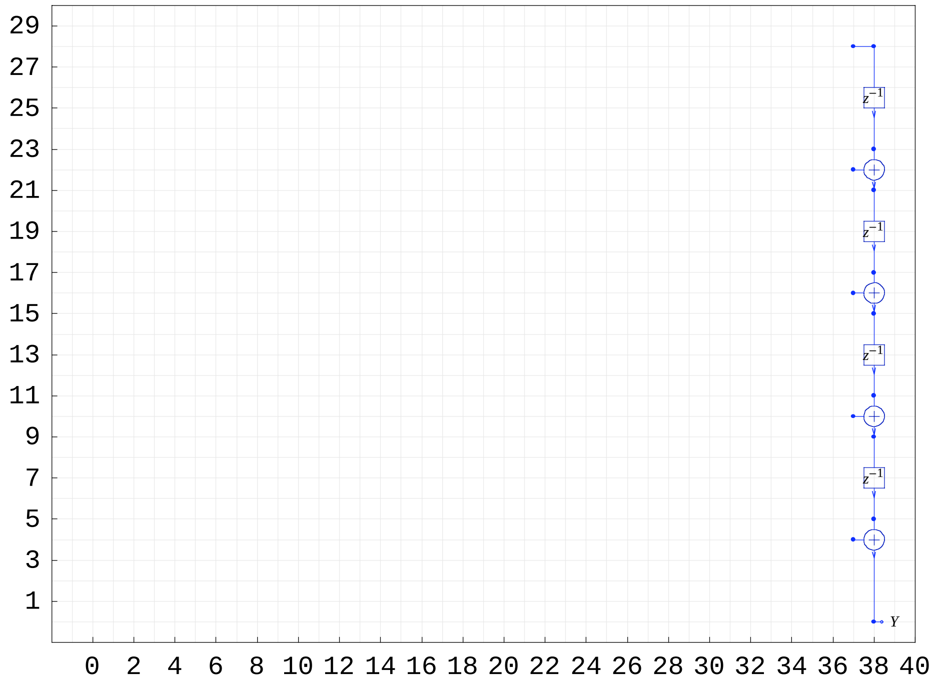
```
SetOptions[DrawElement, PlotStyle → DrawElementPlotStyleLight];
```

Here is the transposed interpolation filter:

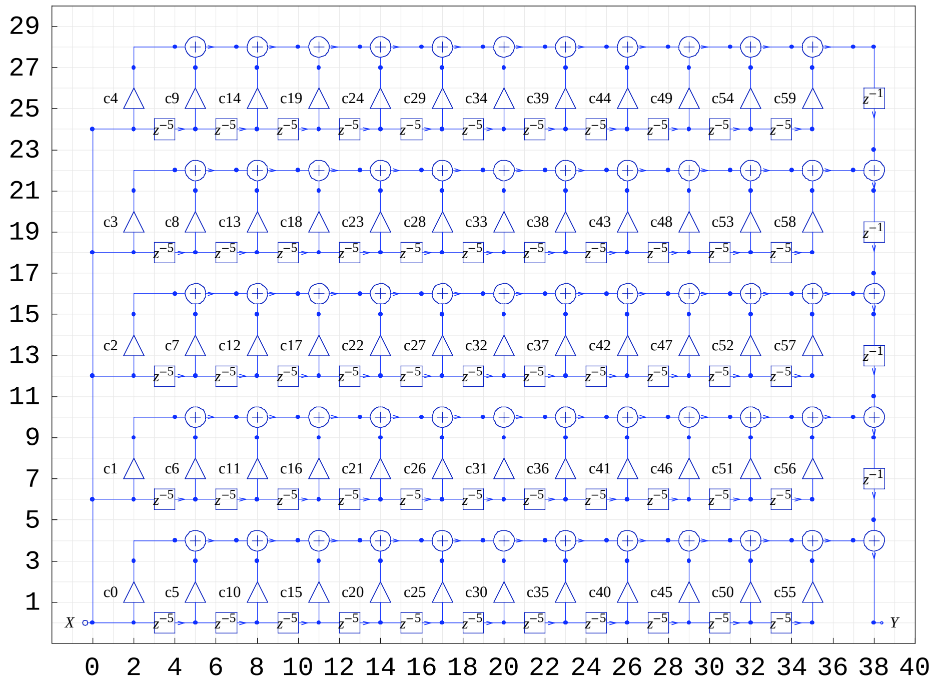
```
inputTransposedPolyphaseSchematic = {
   {"Line", {{0, 0}, {0, 6}}}, {"Line", {{0, 6}, {0, 12}}},
   {"Line", {{0, 12}, {0, 18}}}, {"Line", {{0, 18}, {0, 24}}},
   {"Line", {{0, 24}, {2, 24}}}, {"Line", {{0, 18}, {2, 18}}},
   {"Line", {{0, 12}, {2, 12}}}, {"Line", {{0, 6}, {2, 6}}},
   {"Line", {{0, 0}, {2, 0}}}, {"Input", {0, 0}, X}};
 ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



```
outputTransposedPolyphaseSchematic = {{"Line", {{37, 28}, {38, 28}}},
   {"Adder", {{37, 22}, {38, 21}, {39, 22}, {38, 23}}, {1, 2, 0, 1}},
   {"Adder", {{37, 16}, {38, 15}, {39, 16}, {38, 17}}, {1, 2, 0, 1}},
   {"Adder", {{37, 10}, {38, 9}, {39, 10}, {38, 11}}, {1, 2, 0, 1}},
   {"Adder", {{37, 4}, {38, 0}, {39, 4}, {38, 5}}, {1, 2, 0, 1}},
   {"Delay", {{38, 28}, {38, 23}}, 1}, {"Delay", {{38, 21}, {38, 17}}, 1},
   {"Delay", {{38, 15}, {38, 11}}, 1}, {"Delay", {{38, 9}, {38, 5}}, 1},
   {"Output", {38, 0}, Y}};
ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



```
transposedPolyphaseFIR = Join[
    inputTransposedPolyphaseSchematic,
    outputTransposedPolyphaseSchematic,
    classicFIRschematic1,
    classicFIRschematic2,
    classicFIRschematic3,
    classicFIRschematic4,
    classicFIRschematic5] /. d → 5;
 ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



## Transfer Function of Interpolation Transposed Filter

*SchematicSolver*'s function `DiscreteSystemTransferFunction` computes the filter transfer function:

```
{tfMatrix, systemInp, systemOut} =
  DiscreteSystemTransferFunction[transposedPolyphaseFIR];
transposedTF = tfMatrix[[1, 1]] // Together
```

$$\frac{1}{z^{59}}\left(c59 + c58\, z + c57\, z^{2} + c56\, z^{3} + c55\, z^{4} + c54\, z^{5} + c53\, z^{6} + c52\, z^{7} + c51\, z^{8} + c50\, z^{9} + c49\, z^{10} + c48\, z^{11} + c47\, z^{12} + c46\, z^{13} + c45\, z^{14} + c44\, z^{15} + c43\, z^{16} + c42\, z^{17} + c41\, z^{18} + c40\, z^{19} + c39\, z^{20} + c38\, z^{21} + c37\, z^{22} + c36\, z^{23} + c35\, z^{24} + c34\, z^{25} + c33\, z^{26} + c32\, z^{27} + c31\, z^{28} + c30\, z^{29} + c29\, z^{30} + c28\, z^{31} + c27\, z^{32} + c26\, z^{33} + c25\, z^{34} + c24\, z^{35} + c23\, z^{36} + c22\, z^{37} + c21\, z^{38} + c20\, z^{39} + c19\, z^{40} + c18\, z^{41} + c17\, z^{42} + c16\, z^{43} + c15\, z^{44} + c14\, z^{45} + c13\, z^{46} + c12\, z^{47} + c11\, z^{48} + c10\, z^{49} + c9\, z^{50} + c8\, z^{51} + c7\, z^{52} + c6\, z^{53} + c5\, z^{54} + c4\, z^{55} + c3\, z^{56} + c2\, z^{57} + c1\, z^{58} + c0\, z^{59}\right)$$

The transfer functions of the classic realization and the transposed realization should be the same:

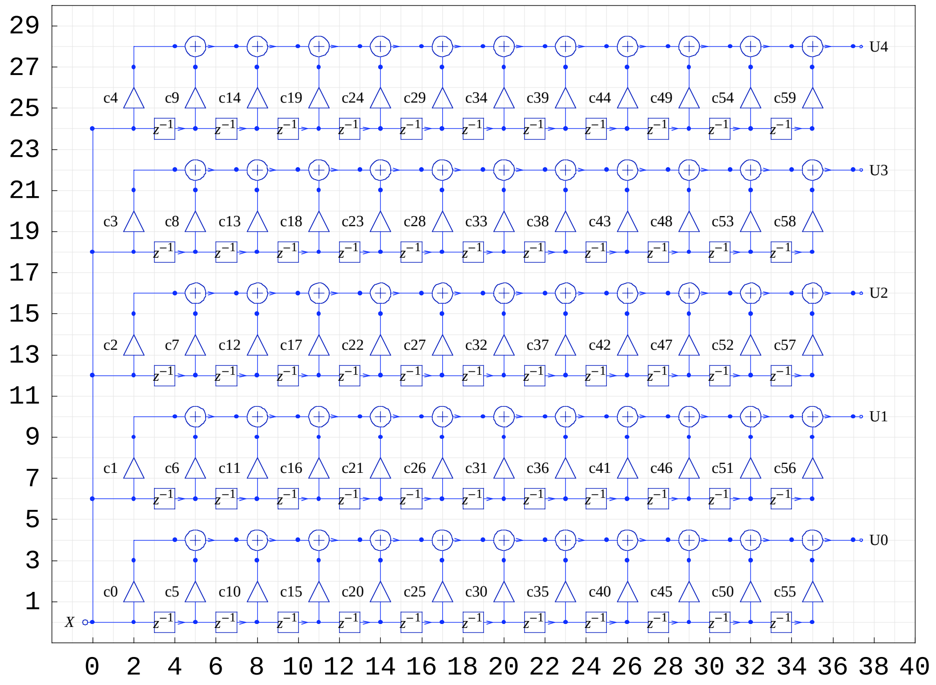
```
SameQ[classicTF, transposedTF]
```

```
True
```

## Processing with Polyphase Interpolation Subsystem

Here is the polyphase interpolation subsystem:

```
polyphaseInterpolationSubsystem = Join[
    inputTransposedPolyphaseSchematic,
    classicFIRschematic1,
    classicFIRschematic2,
    classicFIRschematic3,
    classicFIRschematic4,
    classicFIRschematic5,
    {{"Output", outCoords1[[1]], U0}},
    {{"Output", outCoords2[[1]], U1}},
    {{"Output", outCoords3[[1]], U2}},
    {{"Output", outCoords4[[1]], U3}},
    {{"Output", outCoords5[[1]], U4}}
   ] /. d → 1;
ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



`DiscreteSystemSimulation` finds the output of `polyphaseInterpolationSubsystem`:

```
outIntSeq = DiscreteSystemSimulation[
   polyphaseInterpolationSubsystem /. parameterSubstitution, downSineSeq1];
```

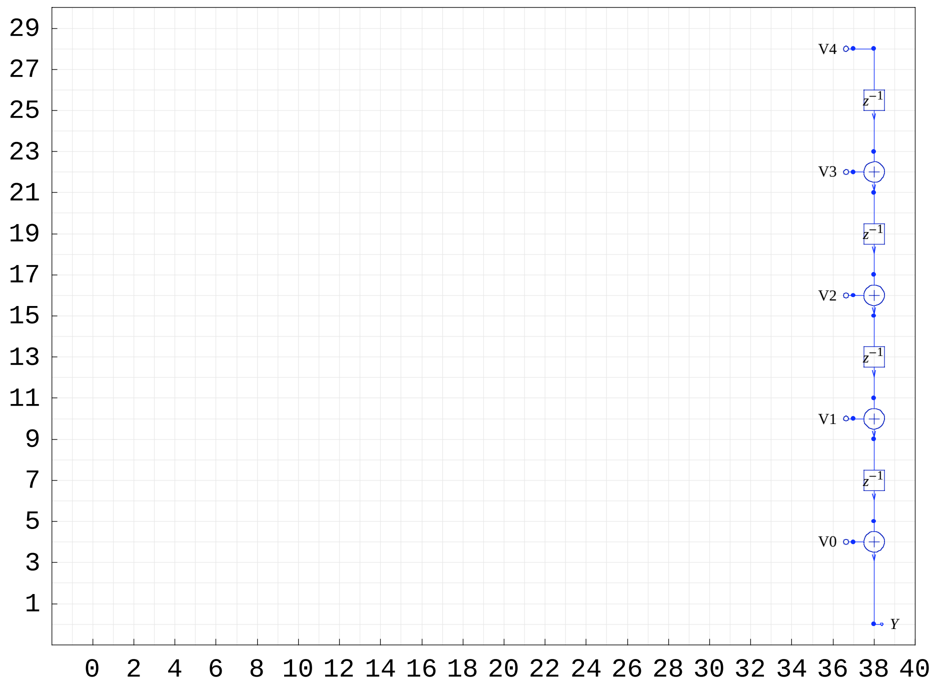
`UpsampleSequence` implements the upsampler:

```
L = 5;
upOutIntSeq = UpsampleSequence[outIntSeq, L];
```

## Processing with Output Interpolation Subsystem

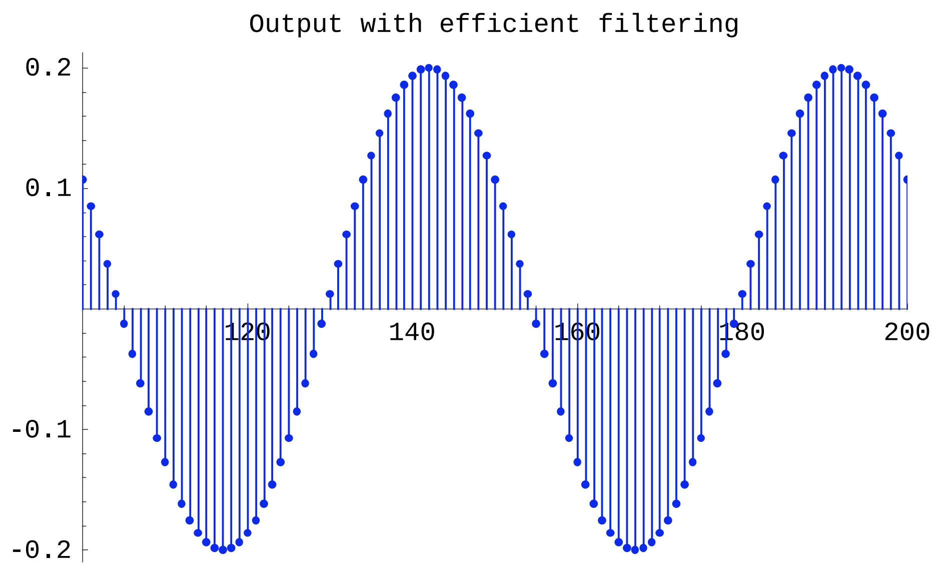
Here is the output interpolation subsystem:

```
outputInterpolationSubsystem = Join[
   outputTransposedPolyphaseSchematic,
   {{"Input", {37, 4}, V0}},
   {{"Input", {37, 10}, V1}},
   {{"Input", {37, 16}, V2}},
   {{"Input", {37, 22}, V3}},
   {{"Input", {37, 28}, V4}}];
ShowSchematic[%, PlotRange → {{-2, 40}, {-1, 30}}, FontSize → 6];
```



`DiscreteSystemSimulation` finds the output of `outputInterpolationSubsystem`:

```
outInterpolationSeq =
  DiscreteSystemSimulation[outputInterpolationSubsystem, upOutIntSeq];

SequencePlot[outInterpolationSeq, PlotRange → {{100, 200}, All},
  PlotLabel -> "Output with efficient filtering"];
```



*SchematicSolver*'s functions demonstrate the well-known benefits of the efficient multirate approach: the same output samples are obtained with `L` times less multiplications. This might

1) increase the computation speed, or

2) decrease the power consumption.

## 11.12. Symbolic Multirate Processing

### Symbolic Stimulus

Assume that we want to process a symbolic sequence

```
numberOfSymbolicSamples = 200;

symbolicSeq = UnitSymbolicSequence[numberOfSymbolicSamples, x];
```

### Downsampled Symbolic Signal

*SchematicSolver*'s function `DownsampleSequence` implements downsampling:

```
M = 5;
downSymbolicSeq = DownsampleSequence[symbolicSeq, M]
```

```
{{x1}, {x6}, {x11}, {x16}, {x21}, {x26}, {x31}, {x36}, {x41}, {x46}, {x51},
 {x56}, {x61}, {x66}, {x71}, {x76}, {x81}, {x86}, {x91}, {x96}, {x101}, {x106},
 {x111}, {x116}, {x121}, {x126}, {x131}, {x136}, {x141}, {x146}, {x151},
 {x156}, {x161}, {x166}, {x171}, {x176}, {x181}, {x186}, {x191}, {x196}}
```

### Symbolic Decimation with Classic FIR Filter

In order to avoid aliasing in multirate systems, it is necessary to bandlimit the spectrum of the input signal, before downsampling, to a frequency below $\frac{1}{2M}$. This is accomplished with a lowpass filter that we implement as `classicFIR`.

*SchematicSolver* performs classic symbolic decimation in two steps.

1) `DiscreteSystemSimulation` finds the filtered signal at the output of `classicFIR`:

```
outClassicSymbolicSeq = DiscreteSystemSimulation[classicFIR, symbolicSeq];
```

2) `DownsampleSequence` implements the downsampler:

```
downOutClassicSymbolicSeq = DownsampleSequence[outClassicSymbolicSeq, M];
```

### Symbolic Decimation with Efficient FIR Filter

*SchematicSolver* performs symbolic decimation with efficient FIR filter in three steps.

1) `DiscreteSystemSimulation` finds the output of `inputDecimationSubsystem`:

```
outDecSymbolicSeq = DiscreteSystemSimulation[inputDecimationSubsystem, symbolicSeq];
```

2) `DownsampleSequence` implements the downsampler:

```
downOutDecSymbolicSeq = DownsampleSequence[outDecSymbolicSeq, M];
```

3) `DiscreteSystemSimulation` finds the output of `polyphaseDecimationSubsystem`:

```
outDecimationSymbolicSeq =
  DiscreteSystemSimulation[polyphaseDecimationSubsystem, downOutDecSymbolicSeq];
```

The output signals `downOutClassicSymbolicSeq` (classic decimation) and `outDecimationSymbolicSeq` (efficient decimation) should be the same; there are no differences attributed to the quantization error:

```
SameQ[downOutClassicSymbolicSeq, outDecimationSymbolicSeq]
```

```
True
```

Here is the 21st output sample:

```
outDecimationSymbolicSeq[[21]]
```

```
{c1 x100 + c0 x101 + c59 x42 + c58 x43 + c57 x44 + c56 x45 + c55 x46 + c54 x47 + c53 x48 + c52 x49 +
  c51 x50 + c50 x51 + c49 x52 + c48 x53 + c47 x54 + c46 x55 + c45 x56 + c44 x57 + c43 x58 + c42 x59 +
  c41 x60 + c40 x61 + c39 x62 + c38 x63 + c37 x64 + c36 x65 + c35 x66 + c34 x67 + c33 x68 + c32 x69 +
  c31 x70 + c30 x71 + c29 x72 + c28 x73 + c27 x74 + c26 x75 + c25 x76 + c24 x77 + c23 x78 + c22 x79 +
  c21 x80 + c20 x81 + c19 x82 + c18 x83 + c17 x84 + c16 x85 + c15 x86 + c14 x87 + c13 x88 +
  c12 x89 + c11 x90 + c10 x91 + c9 x92 + c8 x93 + c7 x94 + c6 x95 + c5 x96 + c4 x97 + c3 x98 + c2 x99}
```

Note that the sample is a fully symbolic expression in terms of the input symbolic samples and the symbolic filter coefficients.

## Symbolic Interpolation with Classic FIR Filter

*SchematicSolver* performs classic symbolic interpolation in two steps.

1) `UpsampleSequence` implements the upsampler:

```
L = 5;
upDownSymbolicSeq = UpsampleSequence[downSymbolicSeq, L]
```

```
{{x1}, {0}, {0}, {0}, {0}, {x6}, {0}, {0}, {0}, {0}, {x11}, {0}, {0}, {0}, {0},
 {x16}, {0}, {0}, {0}, {0}, {x21}, {0}, {0}, {0}, {0}, {x26}, {0}, {0}, {0}, {0},
 {x31}, {0}, {0}, {0}, {0}, {x36}, {0}, {0}, {0}, {0}, {x41}, {0}, {0}, {0}, {0},
 {x46}, {0}, {0}, {0}, {0}, {x51}, {0}, {0}, {0}, {0}, {x56}, {0}, {0}, {0}, {0},
 {x61}, {0}, {0}, {0}, {0}, {x66}, {0}, {0}, {0}, {0}, {x71}, {0}, {0}, {0},
 {0}, {x76}, {0}, {0}, {0}, {0}, {x81}, {0}, {0}, {0}, {0}, {x86}, {0}, {0},
 {0}, {0}, {x91}, {0}, {0}, {0}, {0}, {x96}, {0}, {0}, {0}, {0}, {x101}, {0},
 {0}, {0}, {0}, {x106}, {0}, {0}, {0}, {0}, {x111}, {0}, {0}, {0}, {0}, {x116},
 {0}, {0}, {0}, {0}, {x121}, {0}, {0}, {0}, {0}, {x126}, {0}, {0}, {0}, {0},
 {x131}, {0}, {0}, {0}, {0}, {x136}, {0}, {0}, {0}, {0}, {x141}, {0}, {0}, {0},
 {0}, {x146}, {0}, {0}, {0}, {0}, {x151}, {0}, {0}, {0}, {0}, {x156}, {0}, {0},
 {0}, {0}, {x161}, {0}, {0}, {0}, {0}, {x166}, {0}, {0}, {0}, {0}, {x171}, {0},
 {0}, {0}, {0}, {x176}, {0}, {0}, {0}, {0}, {x181}, {0}, {0}, {0}, {0}, {x186},
 {0}, {0}, {0}, {0}, {x191}, {0}, {0}, {0}, {0}, {x196}, {0}, {0}, {0}, {0}}
```

2) `DiscreteSystemSimulation` finds the filtered signal at the output of `classicFIR`:

```
outClassicInterpolationSymbolicSeq =
  DiscreteSystemSimulation[classicFIR, upDownSymbolicSeq];
```

Here is the 100th output sample:

```
outClassicInterpolationSymbolicSeq[[100]]
```

```
{c59 x41 + c54 x46 + c49 x51 + c44 x56 + c39 x61 +
  c34 x66 + c29 x71 + c24 x76 + c19 x81 + c14 x86 + c9 x91 + c4 x96}
```

Note that the sample is a fully symbolic expression in terms of the input symbolic samples and the symbolic filter coefficients.

## Symbolic Interpolation with Efficient FIR Filter

*SchematicSolver* performs efficient symbolic interpolation in three steps.

1) `DiscreteSystemSimulation` finds the output of `polyphaseInterpolationSubsystem`:

```
outIntSymbolicSeq =
  DiscreteSystemSimulation[polyphaseInterpolationSubsystem, downSymbolicSeq];
```

2) `UpsampleSequence` implements the upsampler:

```
upOutIntSymbolicSeq = UpsampleSequence[outIntSymbolicSeq, L];
```

3) `DiscreteSystemSimulation` finds the output of `outputInterpolationSubsystem`:

```
outputInterpolationSymbolicSeq =
  DiscreteSystemSimulation[outputInterpolationSubsystem, upOutIntSymbolicSeq];
```

The output signals `outClassicInterpolationSymbolicSeq` (classic interpolation) and `outputInterpolationSymbolicSeq` (efficient interpolation) should be the same; there are no differences attributed to the quantization error:

```
SameQ[outClassicInterpolationSymbolicSeq, outputInterpolationSymbolicSeq]
```

```
True
```

Here is the 100th output sample:

```
outputInterpolationSymbolicSeq[[100]]
```

```
{c59 x41 + c54 x46 + c49 x51 + c44 x56 + c39 x61 +
  c34 x66 + c29 x71 + c24 x76 + c19 x81 + c14 x86 + c9 x91 + c4 x96}
```

Note that the sample is a fully symbolic expression in terms of the input symbolic samples and the symbolic filter coefficients.

## Numeric Processing is Special Case of Symbolic Processing

After fully symbolic processing, numeric values can be assigned to the filter coefficients:

```
outputInterpolationSymbolicSeq[[100]] /. parameterSubstitution
```

```
{0.000402511 x41 - 0.0017228 x46 + 0.00378699 x51 +
  0.0011915 x56 - 0.0224119 x61 + 0.0613364 x66 + 0.138885 x71 + 0.0351138 x76 -
  0.0220959 x81 + 0.00522332 x86 + 0.00163053 x91 - 0.00132471 x96}
```

After fully symbolic processing, numeric values can be assigned to the samples:

```
sampleSubstitution =
  Flatten[symbolicSeq] → Flatten[UnitRampSequence[numberOfSymbolicSamples]] // Thread;
```

```
outputInterpolationSymbolicSeq[[100]] /. sampleSubstitution
```

```
{85 c14 + 80 c19 + 75 c24 + 70 c29 + 65 c34 +
  60 c39 + 95 c4 + 55 c44 + 50 c49 + 45 c54 + 40 c59 + 90 c9}
```

In addition, you can assign numeric value to both the samples and the coefficients:

```
outputInterpolationSymbolicSeq[[100]] /. parameterSubstitution /. sampleSubstitution
```

```
{13.8883}
```

Symbolic system simulation is the *SchematicSolver*'s unique feature not available in other simulation software. This section demonstrates that `DiscreteSystemSimulation` returns the output sequence with symbolic sample values.

*SchematicSolver* works with symbolic input, symbolic parameters, and symbolic states.

# 12. Hierarchical Systems

## 12.1. Introduction

*SchematicSolver* can implement a composite discrete system described by standard *SchematicSolver*'s elements and single-input single-output (SISO) subsystems.

SISO subsystems are made out of the standard *SchematicSolver*'s discrete elements.

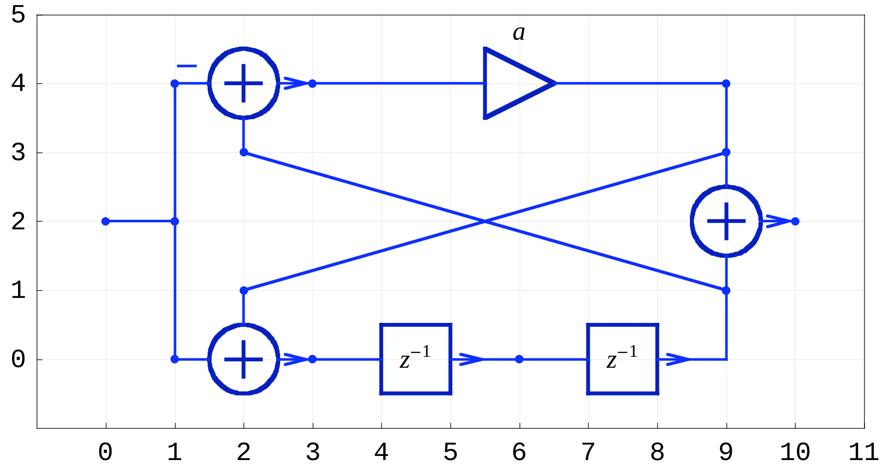
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

## 12.2. Draw Subschematic of Composite System

Here is a schematic of a stage that is the building block of a composite system. We call that schematic a *subschematic*.

```
subschematic = {{"Line", {{1, 0}, {1, 2}}}, {"Line", {{1, 2}, {1, 4}}},
   {"Line", {{2, 1}, {9, 3}}}, {"Line", {{2, 3}, {9, 1}}},
   {"Line", {{0, 2}, {1, 2}}}, {"Line", {{9, 4}, {9, 3}}},
   {"Adder", {{1, 0}, {2, -1}, {3, 0}, {2, 1}}, {1, 0, 2, 1}, ""},
   {"Adder", {{1, 4}, {2, 3}, {3, 4}, {2, 5}}, {-1, 1, 2, 0}, ""},
   {"Adder", {{8, 2}, {9, 1}, {10, 2}, {9, 3}}, {0, 1, 2, 1}, ""},
   {"Multiplier", {{3, 4}, {9, 4}}, a, ""},
   {"Delay", {{3, 0}, {6, 0}}, 1, ""}, {"Delay", {{6, 0}, {9, 1}}, 1, ""}};
 ShowSchematic[%, PlotRange → {{-1, 11}, {-1, 5}}];
```

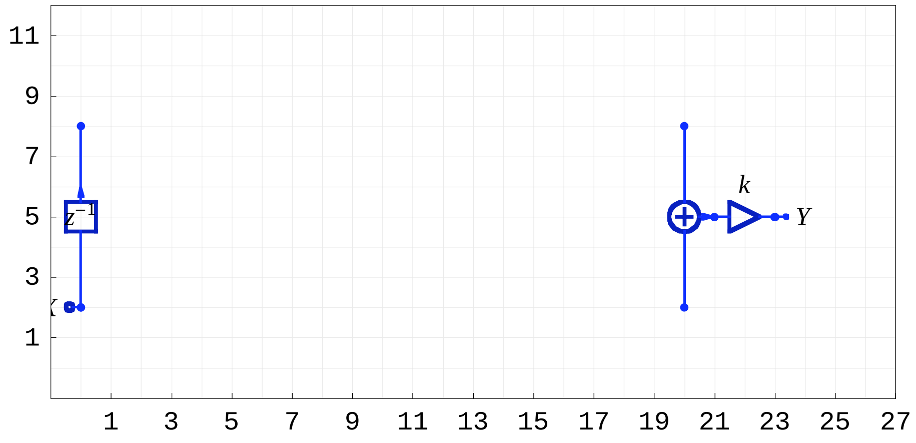


Subschematic has no inputs and outputs.

## 12.3. Draw and Simulate Composite System

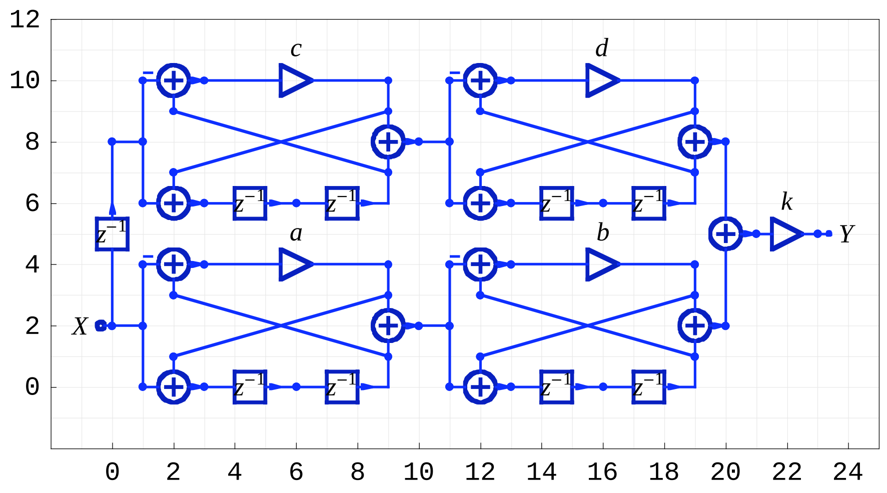
The composite system can be broken into several subschematics. Here is the input-output subschematic:

```
inOutSubschematic = {{"Delay", {{0, 2}, {0, 8}}, 1, ""},
   {"Adder", {{19, 5}, {20, 2}, {21, 5}, {20, 8}}, {0, 1, 2, 1}, ""},
   {"Multiplier", {{21, 5}, {23, 5}}, k, ""},
   {"Input", {0, 2}, X, ""}, {"Output", {23, 5}, Y, ""}};
 ShowSchematic[%, PlotRange → {{-1, 27}, {-1, 12}}];
```



`TranslateSchematic` is used to change the subschematic coordinates and to put the subschematic at the proper place. `Join` is used to form the schematic specification of the composite system. In addition, we can change the values of the subschematics' elements.

```
compositeSystem = Join[inOutSubschematic,
   subschematic,
   TranslateSchematic[subschematic /. a → b, {10, 0}],
   TranslateSchematic[subschematic /. a → c, {0, 6}],
   TranslateSchematic[subschematic /. a → d, {10, 6}]];
ShowSchematic[%, PlotRange → {{-2, 25}, {-2, 12}}];
```



`DiscreteSystemSimulation` finds the output sequence of a discrete system given by a schematic, assuming zero initial conditions. The second argument to `DiscreteSystemSimulation` specifies the input sequence to the system. Here we process 6 samples of an impulse sequence:

```
inpSeq = UnitImpulseSequence[6];
outSeq = DiscreteSystemSimulation[compositeSystem, inpSeq];
% // FullSimplify // TraditionalForm
```

$$\begin{pmatrix} a\,b\,k \\ c\,d\,k \\ (a+b)\,(a\,b-1)\,k \\ (c+d)\,(c\,d-1)\,k \\ (a\,b-1)\,(a^2+b\,a+b^2-1)\,k \\ (c\,d-1)\,(c^2+d\,c+d^2-1)\,k \end{pmatrix}$$

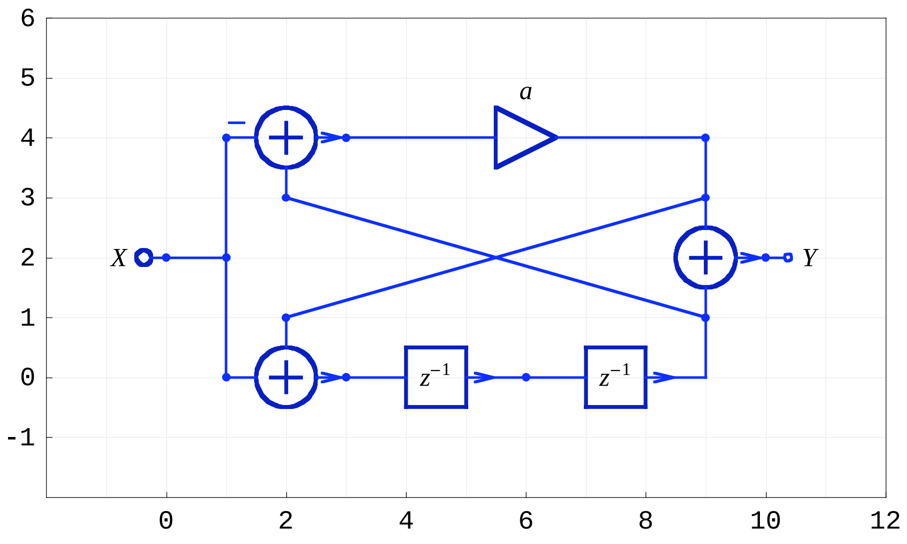
Symbolic system simulation is the *SchematicSolver*'s unique feature not available in other simulation software. The above example demonstrates that `DiscreteSystemSimulation` returns the output sequence with symbolic sample values.

## 12.4. Implementation of Hierarchical System

### Draw and Implement Subsystem

Here is a schematic of a SISO subsystem that is the building block of a composite discrete system. The subsystem is constructed by adding one input element and one output element to the subschematic:

```
discreteSubsystem = Join[subschematic,
   {{"Input", {0, 2}, X, ""},
    {"Output", {10, 2}, Y, ""}}];
ShowSchematic[%, PlotRange → {{-2, 12}, {-2, 6}}];
```



The system summary, generated by `DiscreteSystemImplementationSummary`, points out the subsystem input, initial state, parameter set, output, and final state.

```
DiscreteSystemImplementationSummary[discreteSubsystem];
```

```
Input: {Y[{0, 2}]}

Initial state: {Y[{6, 0}], Y[{2, 3}]}

Parameter: {a}

Output: {Y[{10, 2}]}

Final state: {Y[{3, 0}], Y[{6, 0}]}
```

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the subsystem, and returns a string that is the *Mathematica* code of that function.

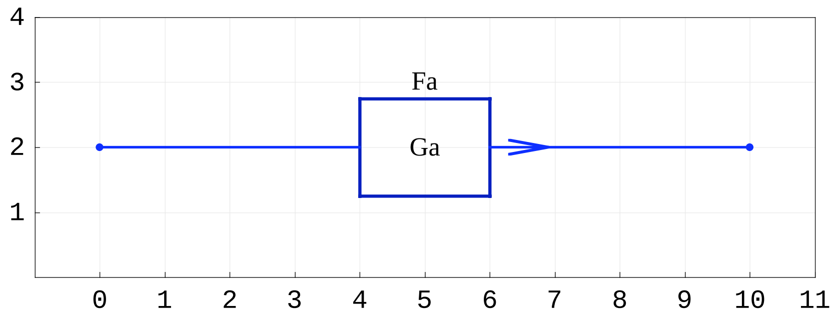
```
DiscreteSystemImplementation[discreteSubsystem, "implementationSubsystem"];

   Implementation procedure name: implementationSubsystem

   Implementation procedure usage:

    {{Y10p2}, {Y3p0, Y6p0}} = implementationSubsystem[
     {Y0p2},{Y6p0, Y2p3},{a}] is the template for calling
     the procedure.  The general template is {outputSamples,
     finalConditions} = procedureName[inputSamples, initialConditions,
     systemParameters]. See also: DiscreteSystemImplementationProcessing
```

The name of the implementation function is arbitrary and it is given as the second argument to `DiscreteSystemImplementation`. In this case, the name of the implementation function is `implementationSubsystem` and it should be enclosed within double quotation marks.

## Draw Hierarchical System

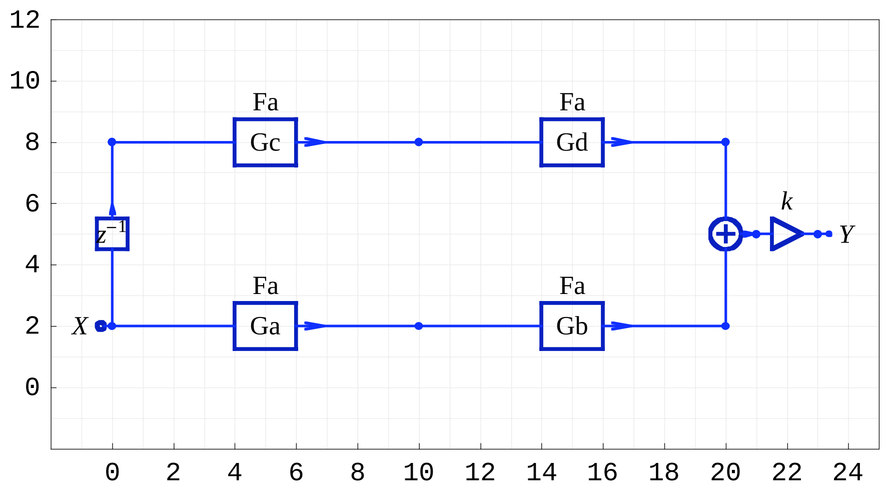
Each subschematic is represented by the Function element.

```
functionSubschematic = {
   {"Function", {{0, 2}, {10, 2}}, Ga, "Fa", ElementSize → {2, 1.5}}};
ShowSchematic[%, PlotRange → {{-1, 11}, {0, 4}}];
```



`TranslateSchematic` is used to change the subschematic coordinates and to put the subschematic at the proper place. `Join` is used to form the schematic specification of the hierarchical system. In addition, we may change the values of the Function elements.

```
hierarchicalSystem  = Join[inOutSubschematic,
   functionSubschematic,
   TranslateSchematic[functionSubschematic /. Ga → Gb, {10, 0}],
   TranslateSchematic[functionSubschematic /. Ga → Gc, {0, 6}],
   TranslateSchematic[functionSubschematic /. Ga → Gd, {10, 6}]];
ShowSchematic[%, PlotRange → {{-2, 25}, {-2, 12}}];
```



The system summary, generated by `DiscreteSystemImplementationSummary`, points out the hierarchical system input, initial state, parameter set, output, and final state.

```
DiscreteSystemImplementationSummary[hierarchicalSystem];

    Input: {Y[{0, 2}]}

    Initial state: {Y[{0, 8}]}

    Parameter: {Ga, Gb, Gc, Gd, k}

    Output: {Y[{23, 5}]}

    Final state: {Y[{0, 2}]}
```

## Implement Hierarchical System

First, we generate a preliminary implementation of the hierarchical system. `DiscreteSystemImplementation` creates a *Mathematica* function that implements the system, and returns a string that is the *Mathematica* code of that function.

```
preliminaryCode =
  DiscreteSystemImplementation[hierarchicalSystem, "implementationPreliminary"];

    Implementation procedure name: implementationPreliminary

    Implementation procedure usage:

     {{Y23p5}, {Y0p2}} = implementationPreliminary[
      {Y0p2},{Y0p8},{Ga, Gb, Gc, Gd, k}] is the template for calling
      the procedure.  The general template is {outputSamples,
      finalConditions} = procedureName[inputSamples, initialConditions,
      systemParameters]. See also: DiscreteSystemImplementationProcessing
```

The name of the implementation function is given as the second argument to `DiscreteSystemImplementation`. In this case, the name of the implementation function is `implementationPreliminary` and it should be enclosed within double quotation marks.

Here is the string that contains the preliminary code of the implementation function:

**preliminaryCode**

```
implementationPreliminary::usage = " {{Y23p5}, {Y0p2}} = implementationPreliminary[
  {Y0p2},{Y0p8},{Ga, Gb, Gc, Gd, k}] is the template for calling the
  procedure.  The general template is {outputSamples, finalConditions} =
  procedureName[inputSamples, initialConditions, systemParameters]. See also:
  DiscreteSystemImplementationProcessing"; implementationPreliminary[] :=
  {1, 1, 5, 8, 1, 1}; implementationPreliminary[dataSamples_List,
  initialConditions_List, systemParameters_List] := Module[ {Y0p2, Y0p8,
  Y10p2, Y20p2, Y10p8, Y20p8, Y21p5, Y23p5, Ga, Gb, Gc, Gd, k}, {Ga, Gb, Gc,
  Gd, k} = systemParameters; {Y0p2} = dataSamples; {Y0p8} = initialConditions;
  Y10p2 = Ga[Y0p2]; Y20p2 = Gb[Y10p2]; Y10p8 = Gc[Y0p8]; Y20p8 = Gd[
  Y10p8]; Y21p5 = Y20p2 + Y20p8; Y23p5 = k*Y21p5; {{Y23p5}, {Y0p2}} ];
```

Edit the preliminary code to include the subsystem definitions:

```
implementationPreliminary[] := {1, 1, 5, 8, 1, 1};

implementationPreliminary[dataSamples_List,
  initialConditions_List, systemParameters_List] := Module[
  {Y0p2, Y0p8, Y10p2, Y20p2, Y10p8, Y20p8, Y21p5, Y23p5,
   Ga, Gb, Gc, Gd, k},
  {Ga, Gb, Gc, Gd, k} = systemParameters;
  {Y0p2} = dataSamples;
  {Y0p8} = initialConditions;
  Y10p2 = Ga[Y0p2];
  Y20p2 = Gb[Y10p2];
  Y10p8 = Gc[Y0p8];
  Y20p8 = Gd[Y10p8];
  Y21p5 = Y20p2 + Y20p8;
  Y23p5 = k*Y21p5;
{{Y23p5}, {Y0p2}}
];
```

The implementation code for the hierarchical system is as follows:

```
implementationHierarchical[] := {1,5,5,8,1,5};

implementationHierarchical[dataSamples_List,
  initialConditions_List, systemParameters_List] := Module[
  {Y0p2, Y0p8, Y10p2, Y20p2, Y10p8, Y20p8, Y21p5, Y23p5,
   Ga, Gb, Gc, Gd, k, stateY10p2, stateY20p2, stateY10p8,
stateY20p8},
  {Ga, Gb, Gc, Gd, k} = systemParameters;
  {Y0p2} = dataSamples;
  {Y0p8, stateY10p2, stateY20p2, stateY10p8, stateY20p8} =
initialConditions;
  {{Y10p2}, stateY10p2} = Ga[Y0p2, stateY10p2];
  {{Y20p2}, stateY20p2} = Gb[Y10p2, stateY20p2];
  {{Y10p8}, stateY10p8} = Gc[Y0p8, stateY10p8];
  {{Y20p8}, stateY20p8} = Gd[Y10p8, stateY20p8];
  Y21p5 = Y20p2 + Y20p8;
  Y23p5 = k*Y21p5;
{{Y23p5}, {Y0p2, stateY10p2, stateY20p2, stateY10p8, stateY20p8}}
];
```

The letters marked in red are inserted into the preliminary code. We added second argument (states) to the functions that represent subsystem implementation.

```
  {{Y10p2}, stateY10p2} = Ga[Y0p2, stateY10p2];
  {{Y20p2}, stateY20p2} = Gb[Y10p2, stateY20p2];
  {{Y10p8}, stateY10p8} = Gc[Y0p8, stateY10p8];
  {{Y20p8}, stateY20p8} = Gd[Y10p8, stateY20p8];
```

Next, we added the state variables to the local variables list

```
   Ga, Gb, Gc, Gd, k, stateY10p2, stateY20p2, stateY10p8,
stateY20p8},
```

to the initial condition list

```
  {Y0p8, stateY10p2, stateY20p2, stateY10p8, stateY20p8} =
initialConditions;
```

and to the final condition list that is returned by the implementation procedure

```
{{Y23p5}, {Y0p2, stateY10p2, stateY20p2, stateY10p8, stateY20p8}}
```

Here is the actual implementation code

```
Clear[implementationHierarchical];
implementationHierarchical[] := {1, 5, 5, 8, 1, 5};
implementationHierarchical[dataSamples_List,
   initialConditions_List, systemParameters_List] := Module[
   {Y0p2, Y0p8, Y10p2, Y20p2, Y10p8, Y20p8, Y21p5, Y23p5,
    Ga, Gb, Gc, Gd, k, stateY10p2, stateY20p2, stateY10p8, stateY20p8},
   {Ga, Gb, Gc, Gd, k} = systemParameters;
   {Y0p2} = dataSamples;
   {Y0p8, stateY10p2, stateY20p2, stateY10p8, stateY20p8} = initialConditions;
   {{Y10p2}, stateY10p2} = Ga[Y0p2, stateY10p2];
   {{Y20p2}, stateY20p2} = Gb[Y10p2, stateY20p2];
   {{Y10p8}, stateY10p8} = Gc[Y0p8, stateY10p8];
   {{Y20p8}, stateY20p8} = Gd[Y10p8, stateY20p8];
   Y21p5 = Y20p2 + Y20p8;
   Y23p5 = k * Y21p5;
   {{Y23p5}, {Y0p2, stateY10p2, stateY20p2, stateY10p8, stateY20p8}}
  ];
```

Each function that represents a subsystem (`Ga`, `Gb`, `Gc`, `Gd`) in the implementation code (`implementationHierarchical`) should be defined according to the usage template for the subsystem implementation procedures (`implementationSubsystem`):

```
Clear[Fa, Fb, Fc, Fd];
Fa[x_, y_: {0, 0}] := implementationSubsystem[{x}, y, {a}];
Fb[x_, y_: {0, 0}] := implementationSubsystem[{x}, y, {b}];
Fc[x_, y_: {0, 0}] := implementationSubsystem[{x}, y, {c}];
Fd[x_, y_: {0, 0}] := implementationSubsystem[{x}, y, {d}];
```

### Processing with Hierarchical System

Let us process a unit impulse sequence with the hierarchical system.

```
inputSequence = UnitImpulseSequence[6];
```

Assume zero initial conditions

```
initialConditions = {0, {0, 0}, {0, 0}, {0, 0}, {0, 0}};
```

and the following system parameters:

```
systemParameters = {Fa, Fb, Fc, Fd, k};
```

`DiscreteSystemImplementationProcessing` processes `inputSequence` for created `implementation-Hierarchical`.

```
{outputSequence, finalConditions} = DiscreteSystemImplementationProcessing[
   inputSequence, initialConditions, systemParameters, implementationHierarchical];
outputSequence // FullSimplify // TraditionalForm
```

$$\begin{pmatrix} a\,b\,k \\ c\,d\,k \\ (a+b)\,(a\,b-1)\,k \\ (c+d)\,(c\,d-1)\,k \\ (a\,b-1)\,(a^2+b\,a+b^2-1)\,k \\ (c\,d-1)\,(c^2+d\,c+d^2-1)\,k \end{pmatrix}$$

The same result has been obtained, already, by simulation of the composite system.

```
SameQ[outSeq, outputSequence]
```

```
True
```

Symbolic processing is the *SchematicSolver*'s unique feature not available in other simulation software. The above example demonstrates that `DiscreteSystemImplementationProcessing` returns the output sequence with symbolic sample values.

# 13. Palettes for Drawing and Solving Systems

## 13.1. Introduction

Palettes provide a simple way to access the full range of *SchematicSolver*'s drawing and solving capabilities.

The *SchematicSolver*'s palettes provide an easy point-and-click interface for performing the most common drawing tasks. However, advanced users might prefer to type and evaluate functions directly. But for users who only want to perform the basic operations, the *SchematicSolver*'s palettes provide the simplest alternative.

You can use the palettes to process

(a) a single notebook with a new schematic,

(b) a new schematic based on existing schematic,  and

(c) an old schematic by adding new elements or removing elements from the schematic.

*SchematicSolver* provides four palettes:

- Palette for drawing and solving continuous-time systems,
  the *Continuous Elements* palette,

- Palette for drawing, solving, simulating, and implementing discrete systems,
  the *Discrete Elements* palette,

- Palette for drawing, simulating, and implementing discrete nonlinear systems, the *Discrete Nonlinear* palette, and
- Palette for specifying drawing options and schematic plot range, the *Schematic Options* palette.

## 13.2. Opening Palettes

If a palette is not open, choose it from the **File** menu:

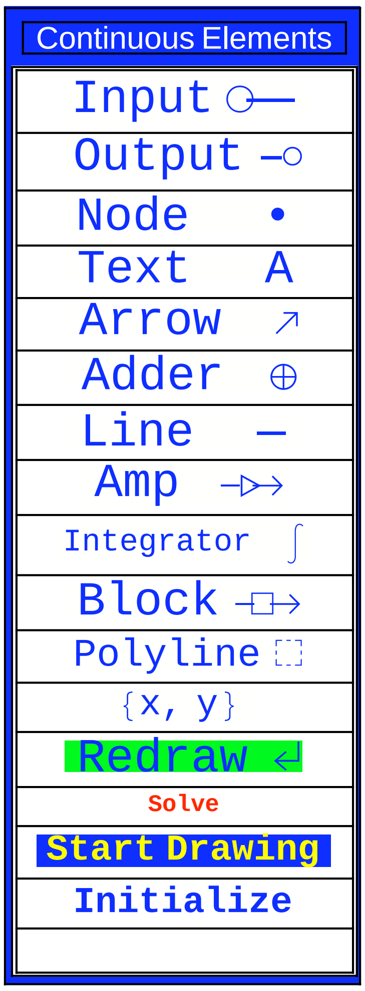
(a) open the **Palettes** submenu of the **File** menu, and choose the **ContinuousElements** command to open the palette Continuous Elements

(b) open the **Palettes** submenu of the **File** menu, and choose the **DiscreteElements** command to open the palette Discrete Elements

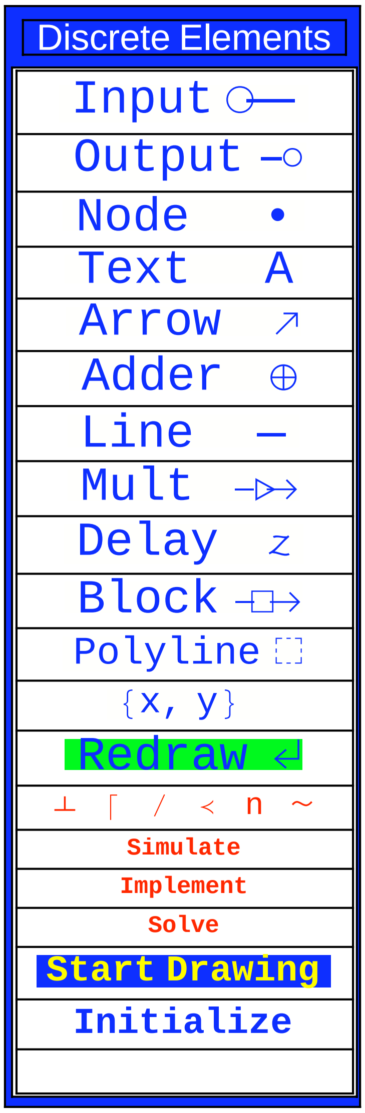
(c) open the **Palettes** submenu of the **File** menu, and choose the **DiscreteNonlinear** command to open the palette Discrete Nonlinear

(d) open the **Palettes** submenu of the **File** menu, and choose the **SchematicOptions** command to open the palette Schematic Options

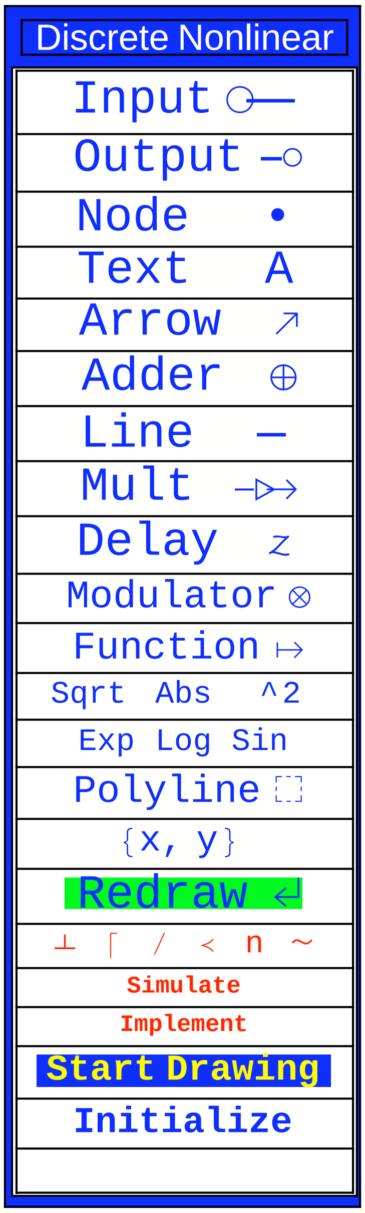
Here is the Continuous Elements palette:



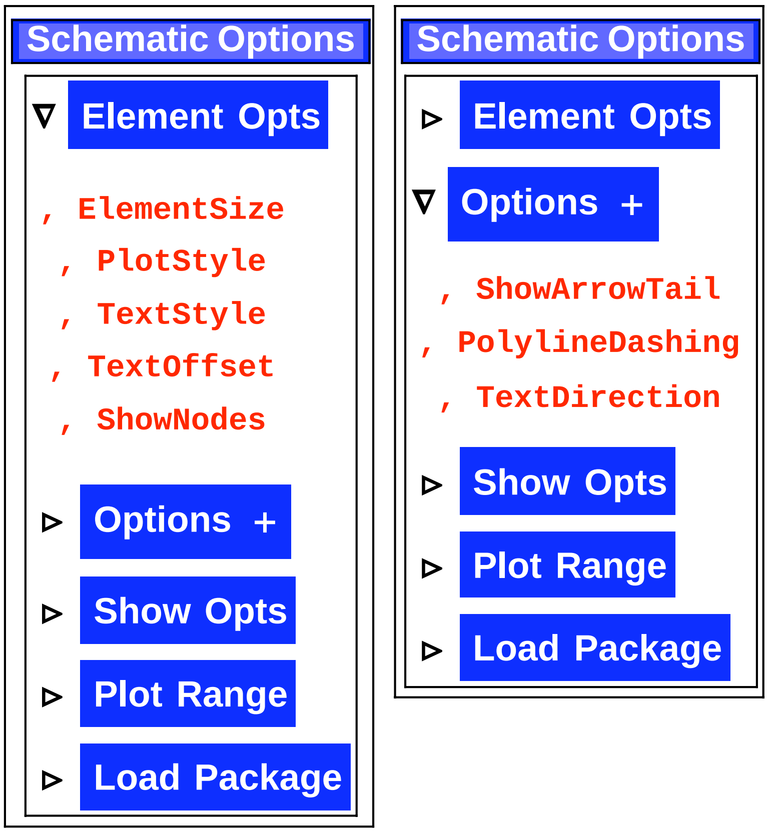
Here is the Discrete Elements palette:

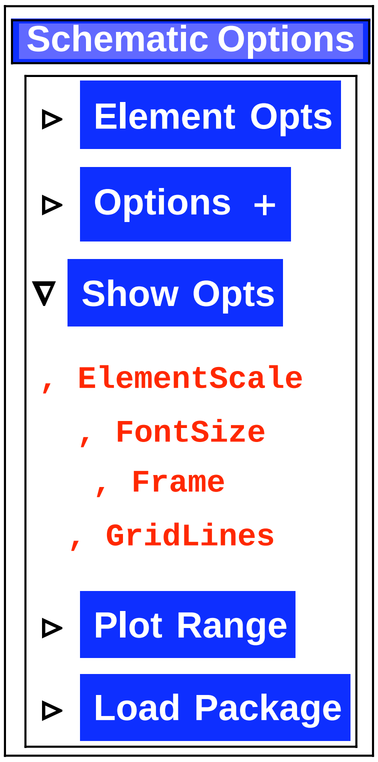


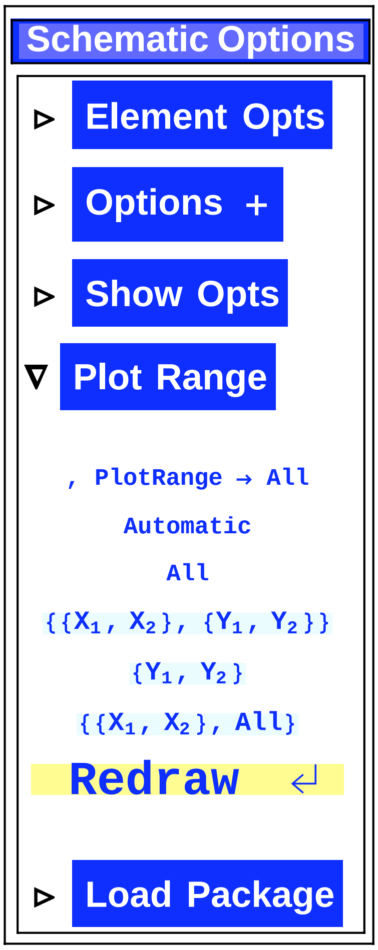
Here is the Discrete Nonlinear palette:

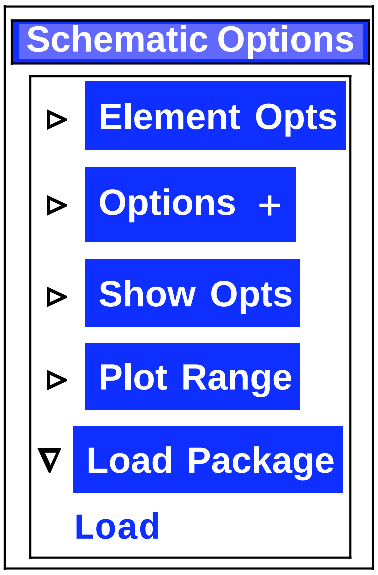


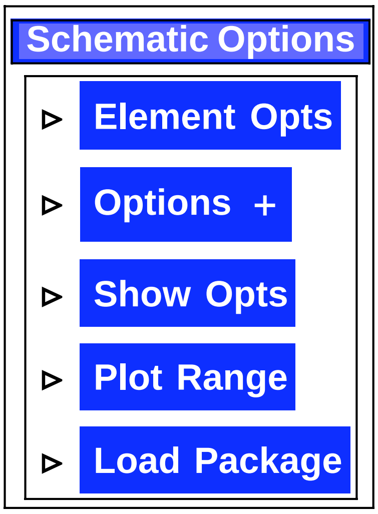
Here is the Schematic Options palette:











You can open all four palettes, and you can use all palettes for drawing the same schematic. However, the solving, simulation, and implementation functions will report error messages if you use elements that do not exist on the corresponding palette.

## 13.3. Contents of Palettes

The Continuous Elements palette, the Discrete Elements palette, and the Discrete Nonlinear palette contain drawing buttons (from the top button `Input` to the button `Polyline`).

The button `Redraw` serves to redraw the schematic.

The button for changing the position of an element on the schematic is `{x, y}`.

The button `Solve` invokes commands for solving a linear system.

The buttons `Simulate` and `Implement` invoke commands for simulating and implementing a discrete system.

Six small buttons above the `Simulate` button generate various sequences.

A new drawing is started by the button `Start Drawing`.

The button `Initialize` loads the package *SchematicSolver*.

Palette heading shows the palette name.

Palette footer provides a visual cue to indicate the function of the button when the mouse cursor is over the button.

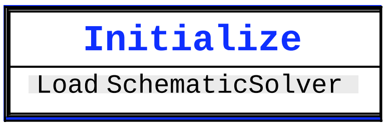
The Schematic Options palette contains buttons for setting drawing options. In addition, this palette automates the plot range selection by mouse point-and-click.

## 13.4. Using Palettes

Here is a step-by-step procedure for using a palette to draw a new schematic.

**1.** (a) Open a new notebook, or (b) place the insertion point in a new empty cell in your notebook, or (c) click once in the blank section of your notebook; a horizontal line appears (this horizontal line is called the *cell insertion bar*).

**2.** Click the button `Initialize`



An input cell will be opened with pasted text, as shown below, and then the whole cell will be evaluated:

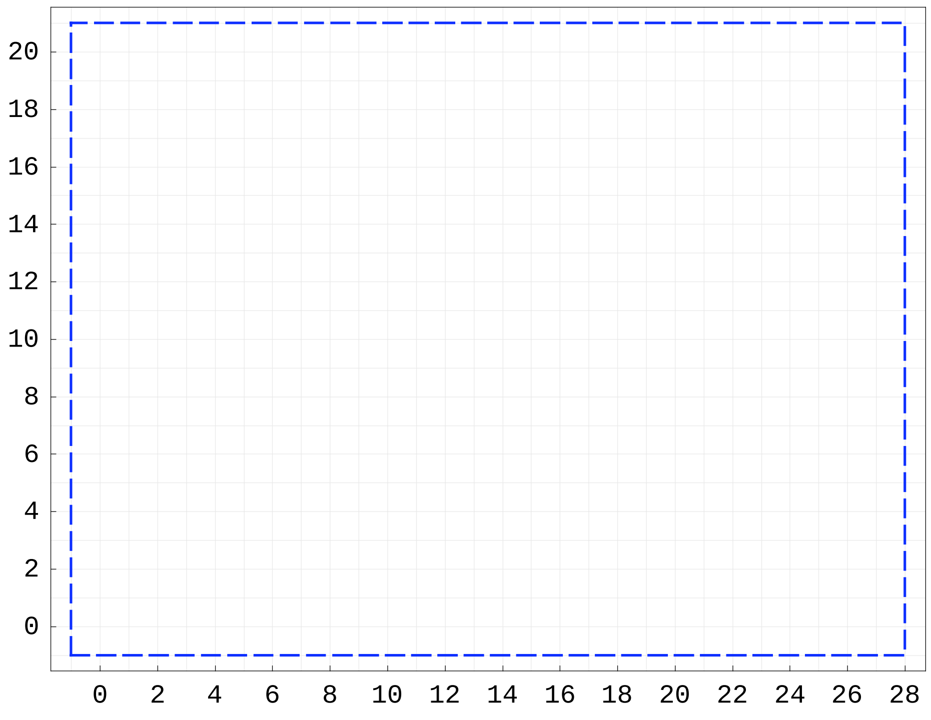
```
Needs["SchematicSolver`"];
      SetOptions[InputNotebook[], ImageSize → {350, 300}, WindowSize → {500, 600}];
```

By clicking the button `Initialize`, the package *SchematicSolver* is loaded and the *SchematicSolver* functions are available. `WindowSize` specifies the size of window that should be used to display the current notebook on the screen. `ImageSize` specifies the absolute size of images to render. Palette footer, below the button `Initialize`, indicates the function of this button — the initialization of a new drawing.

**3.** Click the button `Start Drawing`

A new input cell will be opened with pasted text, as shown below; then the whole cell will be evaluated producing a new graphic output cell below the input cell:

```
mySchematic = {

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



By clicking the button `Start Drawing` a new schematic (typically, a system specification) is generated with only one annotation element — Polyline. The `ShowSchematic` function shows the *drawing workspace* with grid lines. By default, the list of elements that describe the schematic is named `mySchematic`. We call this list the *schematic specification*.

**4.** After clicking the button `Start Drawing`, the horizontal line, the cell insertion bar, will appear below the schematic.

Place the insertion point in the empty line in your schematic specification, above the drawing workspace. In the example below, we marked the insertion point by red vertical separator |.

```
mySchematic = {
 |
 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

**5.** Once you have a schematic specification like this, you can start filling in new elements. For example, to add an input element, click the button `Input`

The text `"Click a point"` is displayed in the window's status area. The *status area* is an area used to display status messages. Usually, the status area appears on the left-hand side in the bottom line of the window.

Move the mouse over the drawing workspace. Click once, say when the mouse position is over the coordinate {5, 10}. The coordinate {5,10} is selected, and it appears in the Input-element specification that is pasted at the current insertion point:

```
{"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
```

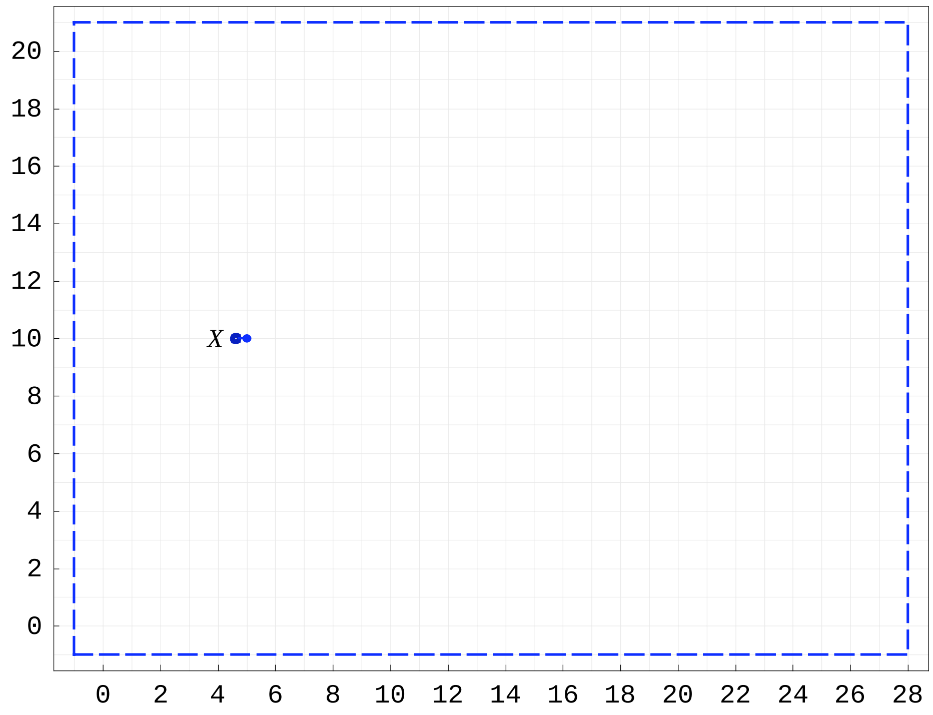
The schematic specification changes, and it has a new element above the empty line:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
 |
 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

The insertion point remains in the empty line. The drawing workspace does not change until you evaluate the cell with the schematic specification.

**6.** Click the button `Redraw` to redraw the schematic:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



The cell insertion bar appears below the drawing workspace.

**7.** Place the insertion point in the empty line in your schematic specification, above the drawing workspace:

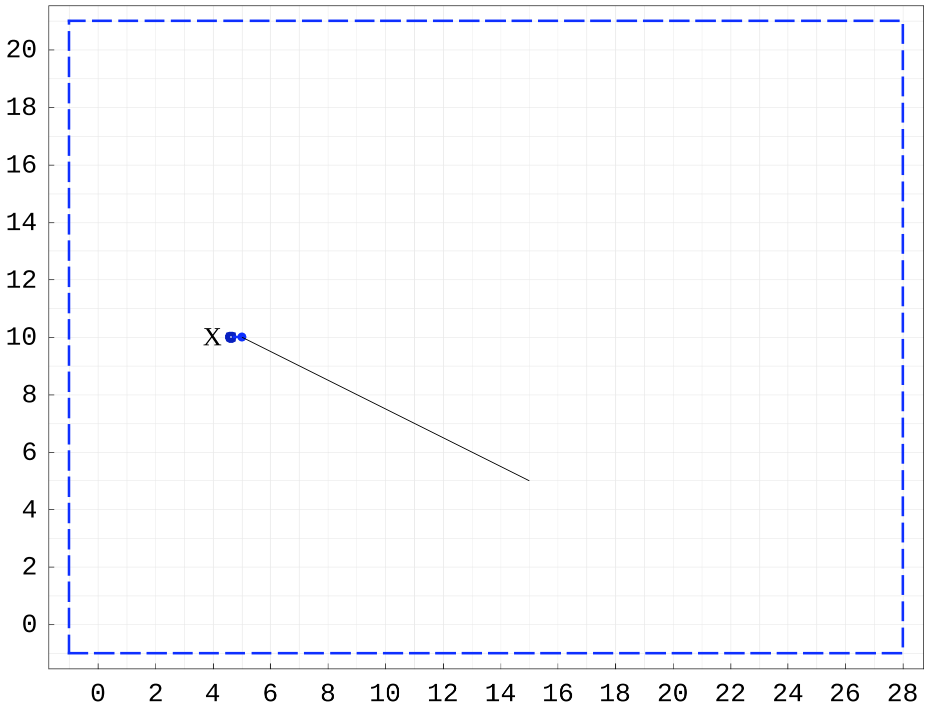
```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
 |
 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

**8.** You can continue filling in your schematic specification with other elements. For example, to add the Line element, click the button `Line`

The text `"Click and Drag"` is displayed in the window's status area.

Move the mouse over the drawing workspace. Press and hold the mouse button, say when the mouse position is over the coordinate {5, 10}. Drag the mouse to specify the second coordinate. Release the mouse button, say at {15, 5}.

```
ShowSchematic[SchematicSolverFigurePalettesDrawLine]
```



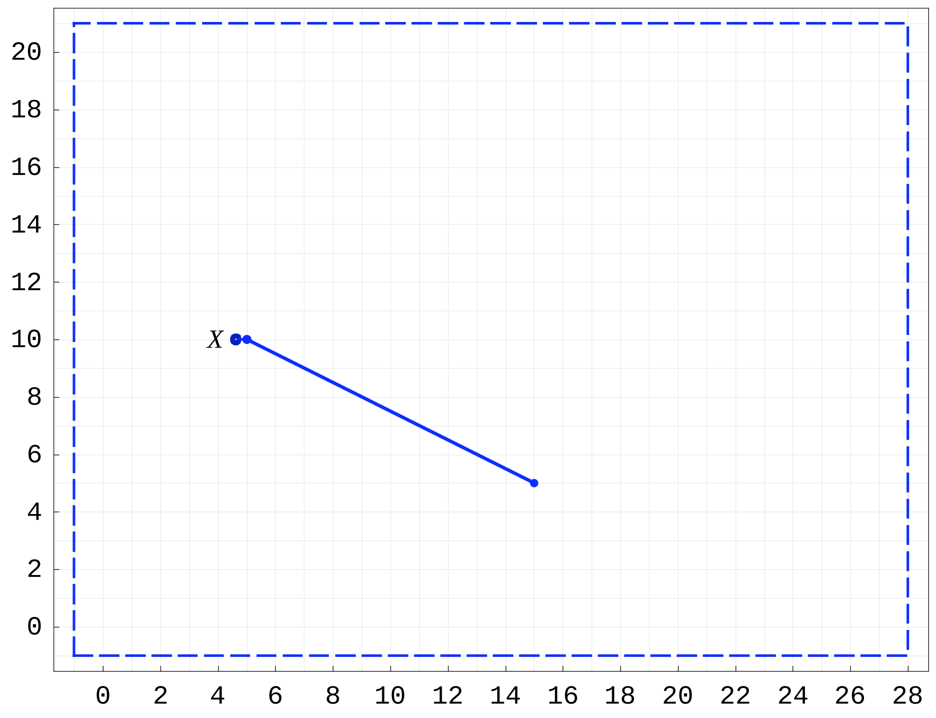
The coordinates {5,10} and {15, 5} are selected, and they appear in the element specification that is pasted at the current insertion point:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
 |
 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

The insertion point remains in the empty line. The drawing workspace does not change until you evaluate the cell with the schematic specification.

**9.** Click the button `Redraw` to redraw the schematic:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



Again, the cell insertion bar appears below the drawing workspace.

## 13.5. Draw Single-Node Elements Using Palettes

Single-node elements are Input, Output, Node, and Text. Here is a step-by-step procedure for using the *SchematicSolver*'s palettes to add a single-node element to the existing schematic:

1. Place the insertion point in the empty line of your schematic specification.

2. Click the corresponding palette button.

3. Move the mouse over the drawing workspace.

4. Click once when the mouse position is over the desired coordinate.

The selected coordinate appears in the element specification that is pasted at the current insertion point. The pasted text is a typical element specification most frequently encountered in practice.

You can edit the pasted element specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in an element specification and start typing. Or you can select a part of the expression, then remove it using the Delete key, or insert a new version by typing it in.

## 13.6. Draw Two-Node Elements Using Palettes

Two-node elements are Line, Block, Multiplier, Delay, Function, Amplifier, Integrator, and Arrow. Here is a step-by-step procedure for using the *SchematicSolver*'s palettes to add a two-node element to the existing schematic:

1. Place the insertion point in the empty line of your schematic specification.

2. Click the corresponding palette button.

3. Move the mouse over the drawing workspace.

4. Press and hold the mouse button when the mouse position is over the first coordinate.

5. Drag the mouse to specify the second coordinate.

6. Release the mouse button when the mouse position is over the second coordinate.

The selected coordinates appear in the element specification that is pasted at the current insertion point. The pasted text is a typical element specification most frequently encountered in practice.

The Discrete Nonlinear palette has six small buttons labeled Sqrt, Abs, ^2, Exp, Log, and Sin for drawing the Function element with the corresponding function.

You can edit the pasted element specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in an element specification and start typing. Or you can select a part of the expression, then remove it using the Delete key, or insert a new version by typing it in.

## 13.7. Draw Adder and Modulator Using Palettes

Here is a step-by-step procedure for using the *SchematicSolver*'s palettes to place the Adder element, or the Modulator element, in the existing schematic:

1. Place the insertion point in the empty line of your schematic specification.

2. Click the button `Adder`

3. Move the mouse over the drawing workspace.

4. Press and hold the mouse button when the mouse position is over the left adder coordinate.

5. Drag the mouse to specify the right adder coordinate.

6. Release the mouse button when the mouse position is over the second coordinate.

The selected coordinates appear in the element specification that is pasted at the current insertion point. The pasted text is a typical element specification most frequently encountered in practice. The coordinates for the upper and lower adder nodes are automatically computed.

You can edit the pasted element specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in an element specification and start typing. Or you can select a part of the expression, then remove it using the Delete key, or insert a new version by typing it in.

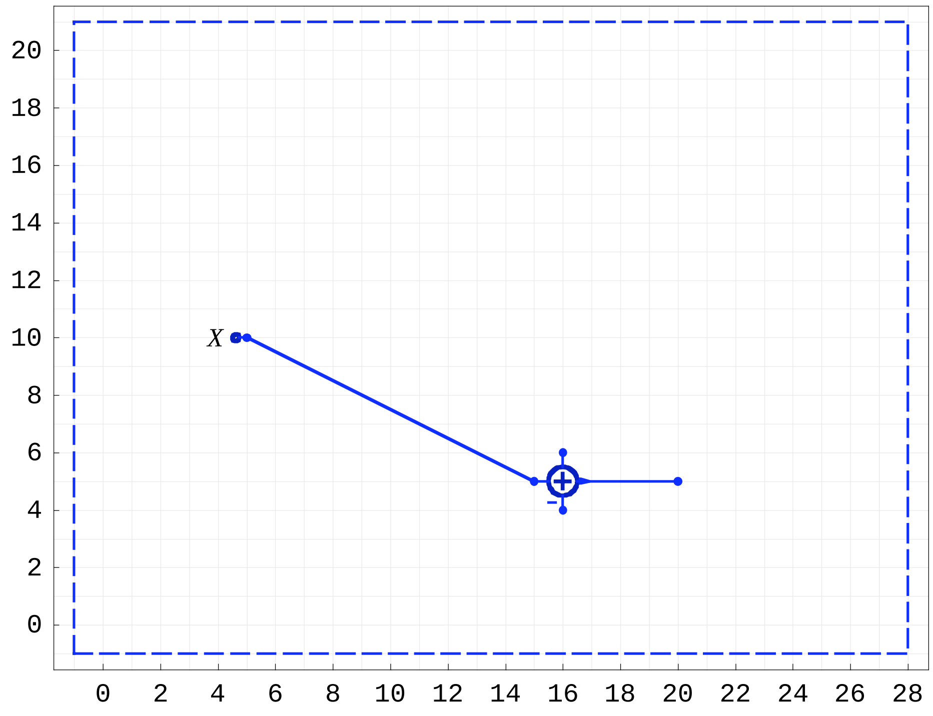
Here is an adder specification created with the palette. Assume that the left node coordinate is {15, 5} and that the right node coordinate is {20, 5}:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, -1, 2, 1}, " "},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

The lower node coordinate {16,4} is automatically generated from the left node coordinate by {16,4}={15,5}+{1,-1}. The upper node coordinate {16,6} is automatically generated from the left node coordinate by {16,6}={15,5}+{1,1}.

Click the button `Redraw` to update the drawing workspace

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, -1, 2, 1}, " "},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

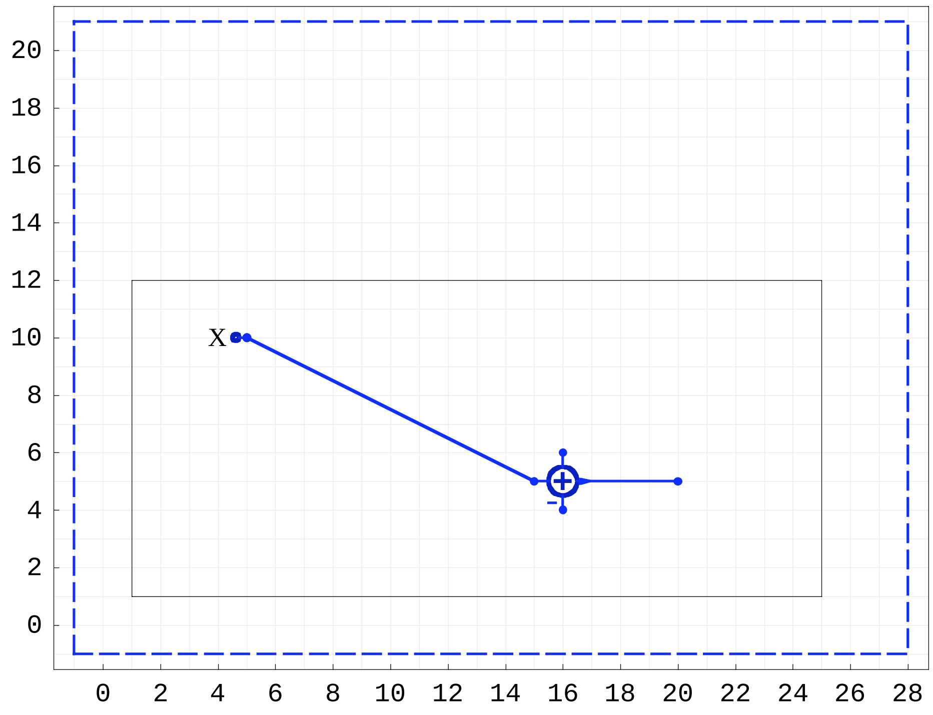


## 13.8. Draw Polyline Element Using Palettes

Here is a step-by-step procedure for using the *SchematicSolver*'s palettes to add a Polyline element in the existing schematic:

1. Place the insertion point in the empty line of your schematic specification.
2. Click the button `Polyline`
3. Move the mouse over the drawing workspace.
4. Press and hold the mouse button when the mouse position is over the lower-left polyline coordinate.
5. Drag the mouse to specify the upper-right polyline coordinate.
6. Release the mouse button when the mouse position is over the second coordinate.

```
ShowSchematic[SchematicSolverFigurePalettesDrawPolyline]
```



The selected coordinates appear in the element specification that is pasted at the current insertion point. The pasted text is a typical element specification most frequently encountered in practice. The coordinates for the upper-left and lower-right polyline nodes are automatically computed.

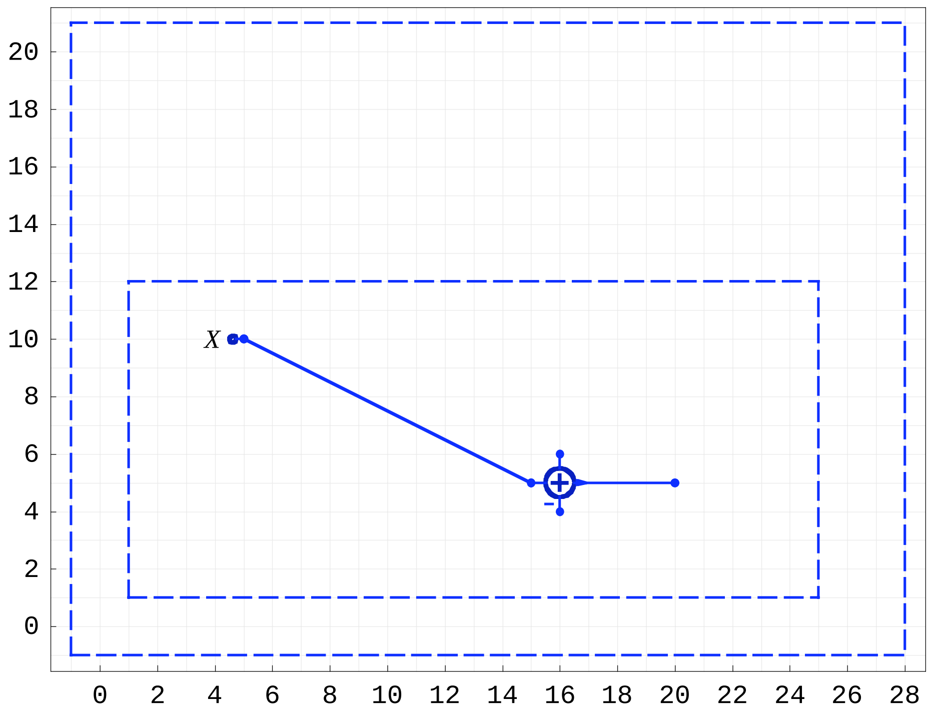
You can edit the pasted element specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in an element specification and start typing. Or you can select a part of the expression, then remove it using the Delete key, or insert a new version by typing it in.

Here is a polyline specification created with the palette. Assume that the lower-left node coordinate is {1,1} and that the upper-right node coordinate is {25,12}:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, -1, 2, 1}, " "},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

Click the button `Redraw` to update the drawing workspace

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, -1, 2, 1}, " "},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



## 13.9. Editing Schematic Specification Using Palettes

### Adding Two or More Elements in Succession

Here is a step-by-step procedure for using the palettes to add one two-node element and one single-node element to the existing schematic specification:

1. Place the insertion point in the empty line of your schematic specification.

2. Click the button `Block`.

3. Move the mouse over the drawing workspace.

4. Press and hold the mouse button when the mouse position is over the first coordinate — the block input.

5. Drag the mouse to specify the second coordinate — the block output.

6. Release the mouse button when the mouse position is over the second coordinate.

The selected coordinates appear in the element specification that is pasted at the current insertion point. The insertion point remains in the empty line. The drawing workspace does not change.

7. Click the button `Output`.

8. Move the mouse over the drawing workspace.

9. Click once when the mouse position is over the desired coordinate.

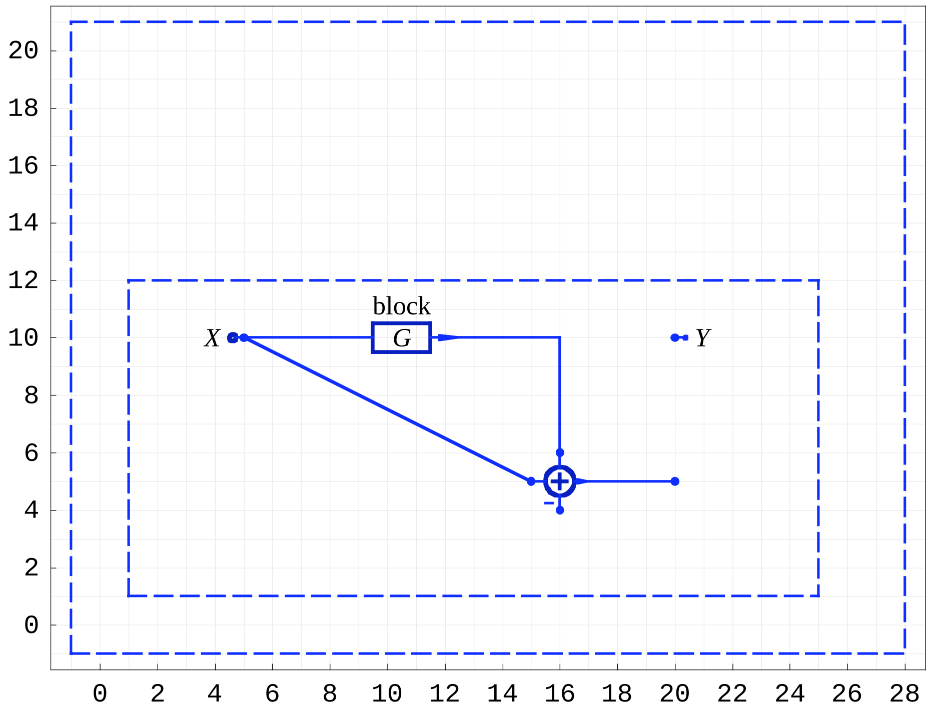
The selected coordinate appears in the element specification that is pasted at the current insertion point. The insertion point remains in the empty line. The drawing workspace does not change.

Here is an example of the schematic specification after adding the Block and Output elements:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, -1, 2, 1}, ""},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},
{"Block", {{5, 10}, {16, 6}}, G, "block"},
{"Output", {20, 10}, Y, "", TextOffset -> {-1, 0}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```

Click the button `Redraw` to update the drawing workspace

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, -1, 2, 1}, ""},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},
{"Block", {{5, 10}, {16, 6}}, G, "block", ElementSize → {2, 1}},
{"Output", {20, 10}, Y, "", TextOffset -> {-1, 0}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



## Changing Element Values

Element specifications can be readily edited. For example, to remove a negative adder input

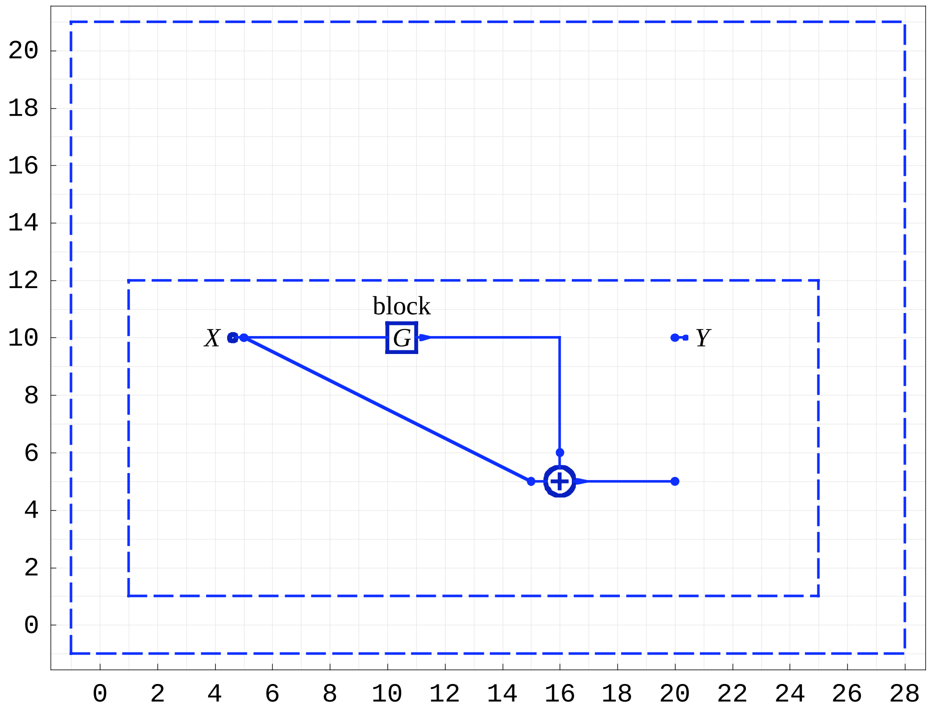
1. select **-1** in the second list of the adder specification, then

2. insert a new value by typing **0**, and

3. evaluate the cell by clicking the button Redraw.

Here is the new schematic specification:

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, 0, 2, 1}, " "},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},
{"Block", {{5, 10}, {16, 6}}, G, "block"},
{"Output", {20, 10}, Y, "", TextOffset -> {-1, 0}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



## Changing Element Coordinates

Assume that we want to connect the adder output to the Output element *Y*:

1. Select the third coordinate pair in the adder specification

```
{"Adder", {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, 0, 2, 1}, " "}
```

```
{Adder, {{15, 5}, {16, 4}, {20, 5}, {16, 6}}, {1, 0, 2, 1},  }
```

2. Click the button `{x, y}`

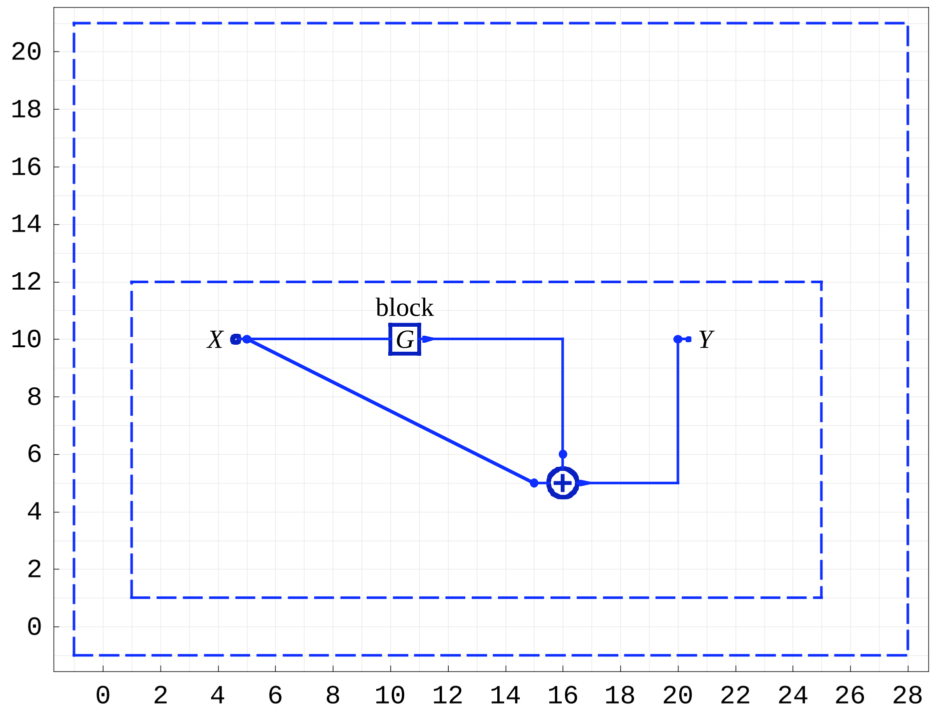
3. Move the mouse over the drawing workspace.

4. Click once when the mouse position is over the desired coordinate, say {20,10}.

The selected coordinate is pasted into the adder specification. The drawing workspace does not change.

Click the button `Redraw` to update the drawing workspace

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 10}, {16, 6}}, {1, 0, 2, 1}, " "},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},
{"Block", {{5, 10}, {16, 6}}, G, "block"},
{"Output", {20, 10}, Y, "", TextOffset -> {-1, 0}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
 ShowSchematic[%];
```



## Removing Elements from Schematic

You can edit element specifications in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in an element specification and start typing. Or you can select a part of the expression, then remove it using the Delete key, or insert a new version by typing it in.

To remove an entire element from the schematic, select the element specification, and the comma following the specification, and press the Delete key.

Alternatively, you can use the *Mathematica* `Delete` function to drop out items from the schematic specification.

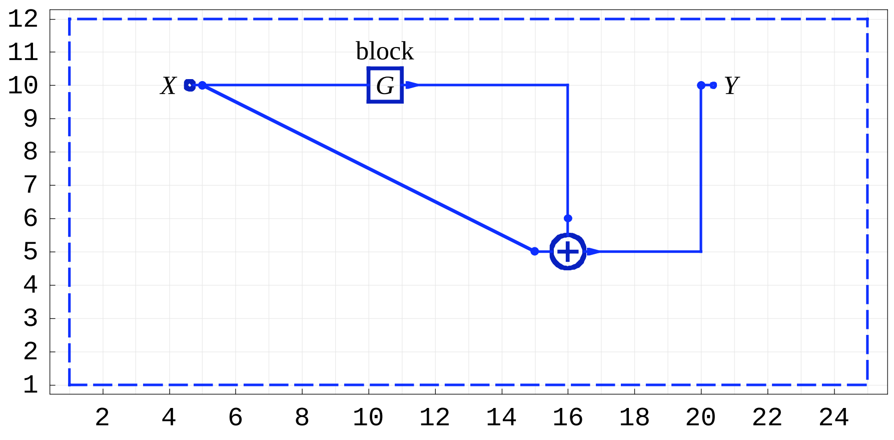
For example, consider the specification

```
mySchematic = {
 {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},
{"Line", {{5, 10}, {15, 5}}},
{"Adder", {{15, 5}, {16, 4}, {20, 10}, {16, 6}}, {1, 0, 2, 1}, " "},
{"Polyline", {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},
{"Block", {{5, 10}, {16, 6}}, G, "block"},
{"Output", {20, 10}, Y, "", TextOffset -> {-1, 0}},

 {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}}};
```

To remove the last polyline element, use the command

```
mySchematic = Delete[mySchematic, Length[mySchematic]]
% // ShowSchematic
```

```
{{Input, {5, 10}, X, , TextOffset → {1, 0}}, {Line, {{5, 10}, {15, 5}}},
 {Adder, {{15, 5}, {16, 4}, {20, 10}, {16, 6}}, {1, 0, 2, 1},  },
 {Polyline, {{1, 1}, {25, 1}, {25, 12}, {1, 12}, {1, 1}}},
 {Block, {{5, 10}, {16, 6}}, G, block}, {Output, {20, 10}, Y, , TextOffset → {-1, 0}}}
```



## 13.10. Solving Linear Systems Using Palettes

Schematic specification can be used to describe a system. Typically, we want to solve the system: to find the system response, or to compute the transfer function. The palette button **Solve** pastes and evaluates a template for general solving a linear system. The button **Solve** assumes that the name of the schematic specification is `mySchematic`. Here is what you get after clicking the button **Solve**:

```
Print["Equations of the System:"];
{myEquations, myVars} = DiscreteSystemEquations[mySchematic];
myEquations // ColumnForm
Print["Response of the System:"];
{myResponse, myVars} = DiscreteSystemResponse[mySchematic];
myResponse // ColumnForm
Print["Signals of the System:"];
{mySignals, myVars} = DiscreteSystemSignals[mySchematic];
% // Transpose // TableForm
Print["Transfer Function Matrix:"];
{myTF, myInputs, myOutputs} = DiscreteSystemTransferFunction[mySchematic];
myTF // MatrixForm
Print["Inputs of the System:"];
myInputs
Print["Outputs of the System:"];
myOutputs
```

```
   Equations of the System:

Y[{5, 10}] == X
Y[{20, 10}] == Y[{5, 10}] + Y[{16, 6}]
Y[{16, 6}] == G Y[{5, 10}]

   Response of the System:

Y[{20, 10}] → X + G X
Y[{16, 6}] → G X
Y[{5, 10}] → X

   Signals of the System:

X + G X     Y[{20, 10}]
G X         Y[{16, 6}]
X           Y[{5, 10}]

   Transfer Function Matrix:

( 1 + G )

   Inputs of the System:

{Y[{5, 10}]}

   Outputs of the System:

{Y[{20, 10}]}
```

**End of SchematicSolver Solving**

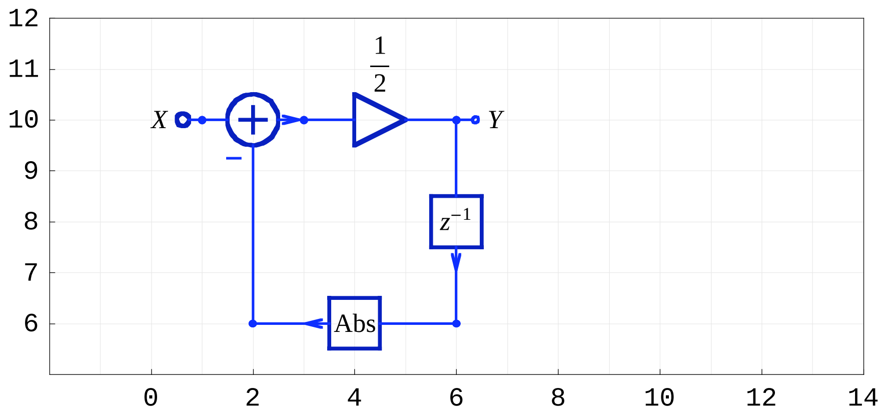
Further processing can be applied to the results returned by **Solve**, say by using *Control System Professional*.

## 13.11. Simulating and Implementing Systems Using Palettes

*SchematicSolver* can be used for simulation and generation of software implementation of discrete systems.

Here is an example schematic specification:

```
mySchematic = {
   {"Input", {1, 10}, X, ""},
   {"Output", {6, 10}, Y, ""},
   {"Adder", {{1, 10}, {2, 6}, {3, 10}, {2, 11}}, {1, -1, 2, 0}, ""},
   {"Multiplier", {{3, 10}, {6, 10}}, 1/2, ""},
   {"Delay", {{6, 10}, {6, 6}}, 1, ""},
   {"Function", {{6, 6}, {2, 6}}, Abs, ""}};
 ShowSchematic[%, PlotRange → {{-2, 14}, {5, 12}}];
```



The palette button **Simulate** pastes and evaluates a template for simulating a system. The button **Simulate** assumes that the name of the schematic specification is `mySchematic` and the unit impulse sequence for input sequence. Here is what you get after clicking the button **Simulate**:

```
DiscreteSystemSimulation[mySchematic]
```

$\{\{\frac{1}{2}\}, \{-\frac{1}{4}\}, \{-\frac{1}{8}\}, \{-\frac{1}{16}\}, \{-\frac{1}{32}\}, \{-\frac{1}{64}\}, \{-\frac{1}{128}\}, \{-\frac{1}{256}\}\}$

```
--- End of SchematicSolver Simulation ---
```

`DiscreteSystemSimulation` simulates a system with zero initial conditions.

The palette button **Implement** pastes and evaluates a template for implementing a system. The **Implement** button assumes that the name of the schematic specification is `mySchematic`, the unit impulse sequence for input sequence, the zero initial conditions, and `implementationProcedure` as the name of the *Mathematica* function that implements the system. Here is what you get after clicking the button **Implement**:

```
procedureName = implementationProcedure;
DiscreteSystemImplementation[mySchematic, ToString[procedureName]];
DiscreteSystemImplementationSummary[mySchematic, Verbose -> True]
Print["--- EXAMPLE: Input Sequence, Initial Conditions, System Parameters"];
{inpVec, initCond, params, eqns, outVec, finalCond} =
  DiscreteSystemImplementationEquations[mySchematic];
numberOfInputs = Length[inpVec];
inputSequence = MultiplexSequence @@ Table[UnitImpulseSequence[], {numberOfInputs}]
initialConditions = 0 * initCond
systemParameters = params
Print["--- PROCESSING: Output Sequence, Final Conditions"];
{outputSequence, finalConditions} = DiscreteSystemImplementationProcessing[
   inputSequence, initialConditions, systemParameters, procedureName];
outputSequence
finalConditions
```

```
    Implementation procedure name: implementationProcedure

    Implementation procedure usage:

     {{Y6p10}, {Y6p10}} = implementationProcedure[{Y1p10},{Y6p6},{}] is the
      template for calling the procedure.  The general template is {outputSamples,
      finalConditions} = procedureName[inputSamples, initialConditions,
      systemParameters]. See also: DiscreteSystemImplementationProcessing

    Input: {Y[{1, 10}]}

    Initial state: {Y[{6, 6}]}

    Parameter: {}

    Equations:  Y[{1, 10}] == X
                Y[{6, 6}] == previousSample[Y[{6, 10}]]
                Y[{2, 6}] == Abs[Y[{6, 6}]]
                Y[{3, 10}] == Y[{1, 10}] - Y[{2, 6}]
                Y[{6, 10}] == 1/2 Y[{3, 10}]

    Output: {Y[{6, 10}]}

    Final state: {Y[{6, 10}]}

    --- EXAMPLE: Input Sequence, Initial Conditions, System Parameters

{{1}, {0}, {0}, {0}, {0}, {0}, {0}, {0}}

{0}

{}

    --- PROCESSING: Output Sequence, Final Conditions
```

$\{\{\frac{1}{2}\}, \{-\frac{1}{4}\}, \{-\frac{1}{8}\}, \{-\frac{1}{16}\}, \{-\frac{1}{32}\}, \{-\frac{1}{64}\}, \{-\frac{1}{128}\}, \{-\frac{1}{256}\}\}$

$\{-\frac{1}{256}\}$

**--- End of SchematicSolver Implementation ---**

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system.

`DiscreteSystemImplementationProcessing` processes a data sequence for the created function.

`DiscreteSystemImplementationSummary` prints a summary of the system implementation.

## 13.12. Setting Element Drawing Options

The Schematic Options palette contains buttons for setting element drawing options.

*SchematicSolver* can draw elements in different colors and sizes by means of *element options*. The following options are available for all elements:

`ElementSize`

`PlotStyle`

`TextStyle`

`ShowNodes`

`TextOffset`

Special options are provided for controlling some elements:

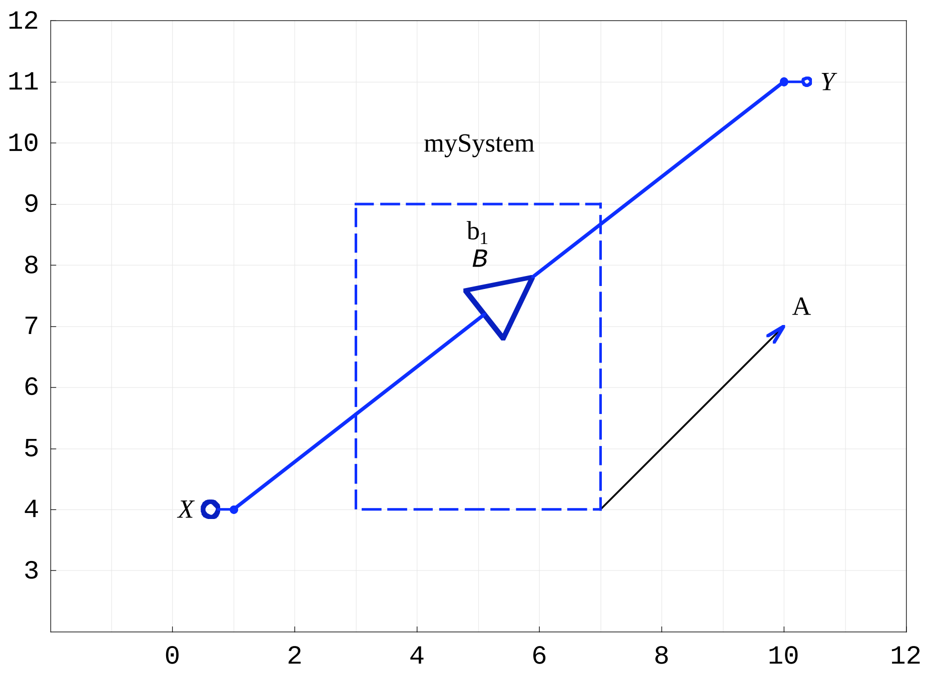
`ShowArrowTail` for the Arrow element,

`PolylineDashing` for the Polyline element,

`TextDirection` for the Text element.

Here is an example schematic:

```
mySchematic = {
 {"Input", {1, 4}, X, "", TextOffset → {1, 0}},
{"Multiplier", {{1, 4}, {10, 11}}, B, "b1"},
{"Output", {10, 11}, Y, "", TextOffset → {-1, 0}},
{"Arrow", {{10, 7}, {7, 4}}, "A"},
{"Polyline", {{3, 9}, {7, 9}, {7, 4}, {3, 4}, {3, 9}}},
{"Text", {5, 10}, "mySystem"}
  };
 ShowSchematic[%, PlotRange → {{-2, 12}, {2 , 12}}];
```



Step-by-step procedure for setting the element drawing options follows.

**1.** In your schematic specification, place the insertion point at the end of the element specification just before the right-most curly brace. In the example below, in the Input-element specification, we marked the insertion point by red vertical separator |.

```
{"Input",{1,4},X,"",TextOffset→{1,0}|},
```

**2.** Click the button `, PlotStyle` to change the element color

The schematic specification changes, and it has a new text in the Input-element specification:

```
{"Input",{1,4},X,"",TextOffset→{1,0},PlotStyle→{{RGBColor[0,1,0]},{-
RGBColor[1,0,0]}}},
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

**3.** Place the insertion point at the end of the Multiplier-element specification, just before the last "}". We marked the insertion point by red vertical separator |.

```
{Multiplier, {{1, 4}, {10, 8}}, B, b1 |},
```

**4.** Click the button `, ElementSize` to change the element size

The schematic specification changes, and it has a new text in the Multiplier-element specification:

```
{Multiplier, {{1, 4}, {10, 8}}, B, b1, ElementSize → {2, 1.5}},
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

**5.** Place the insertion point at the end of the Arrow-element specification, just before the last "}". We marked the insertion point by red vertical separator |.

```
{"Arrow", {{10,7},{7,4}}, "A"|},
```

**6.** Click the button `, ShowArrowTail` to draw only the arrowhead.

The schematic specification changes, and it has a new text in the Arrow-element specification:

```
{"Arrow", {{10,7},{7,4}}, "A",ShowArrowTail→False},
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

**7.** Place the insertion point at the end of the Polyline-element specification, just before the last "}". We marked the insertion point by red vertical separator |.

```
{Polyline, {{3, 9}, {7, 9}, {7, 4}, {3, 4}, {3, 9}} |},
```

**8.** Click the button `, PolylineDashing` to change the dashing style of the Polyline element

The schematic specification changes, and it has a new text in the Polyline-element specification:

```
{"Polyline",{{3,9},{7,9},{7,4},{3,4},{3,9}},PolylineDashing→Dashing[{0.04,
0.03}]},
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

**9.** Place the insertion point at the end of the Text-element specification, just before the last "}". We marked the insertion point by red vertical separator |.

```
{"Text",{5,10},"mySystem"|}
```

**10.** Click the button `, TextDirection` to rotate text

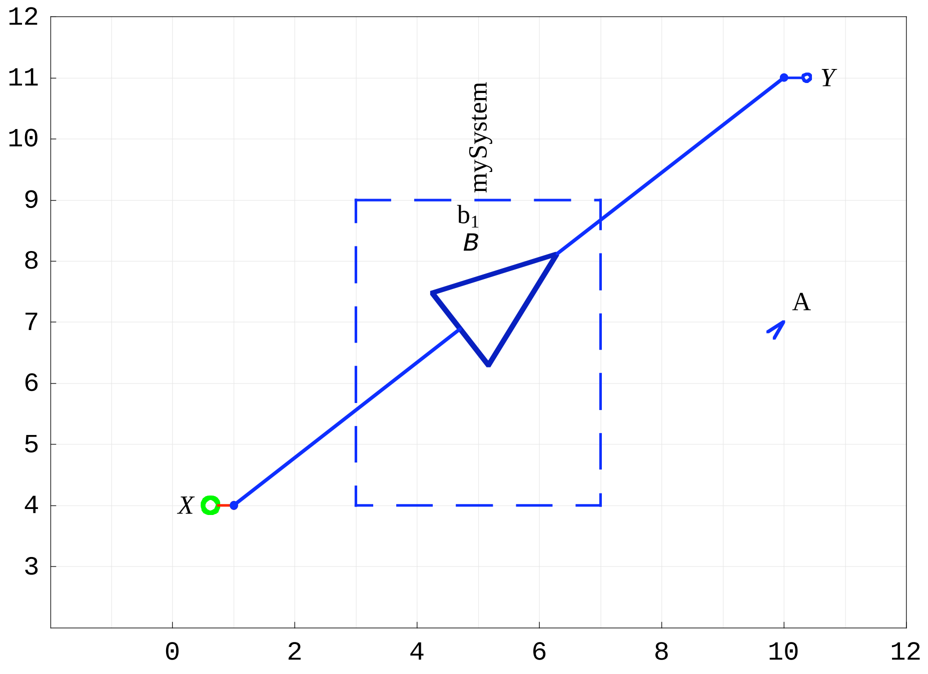
The element specification changes to

```
{"Text",{5,10},"mySystem",TextDirection→{0, 1}}
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

Here is the updated schematic specification:

```
mySchematic = {
 {"Input", {1, 4}, X, "", TextOffset → {1, 0},
    PlotStyle → {{RGBColor[0, 1, 0]}, {RGBColor[1, 0, 0]}}},
{"Multiplier", {{1, 4}, {10, 11}}, B, "b1", ElementSize → {2, 1.5}},
{"Output", {10, 11}, Y, "", TextOffset → {-1, 0}},
{"Arrow", {{10, 7}, {7, 4}}, "A", ShowArrowTail → False},
{"Polyline", {{3, 9}, {7, 9}, {7, 4}, {3, 4}, {3, 9}},
    PolylineDashing → Dashing[{0.04, 0.03}]},
{"Text", {5, 10}, "mySystem", TextDirection → {0, 1}}
  };
 ShowSchematic[%, PlotRange → {{-2, 12}, {2 , 12}}];
```



You can edit the specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in the specification and start editing.

## 13.13. Setting `ShowSchematic` Drawing Options

The Schematic Options palette contains buttons for fine-tuning graphics created by `ShowSchematic`:

`ElementScale`

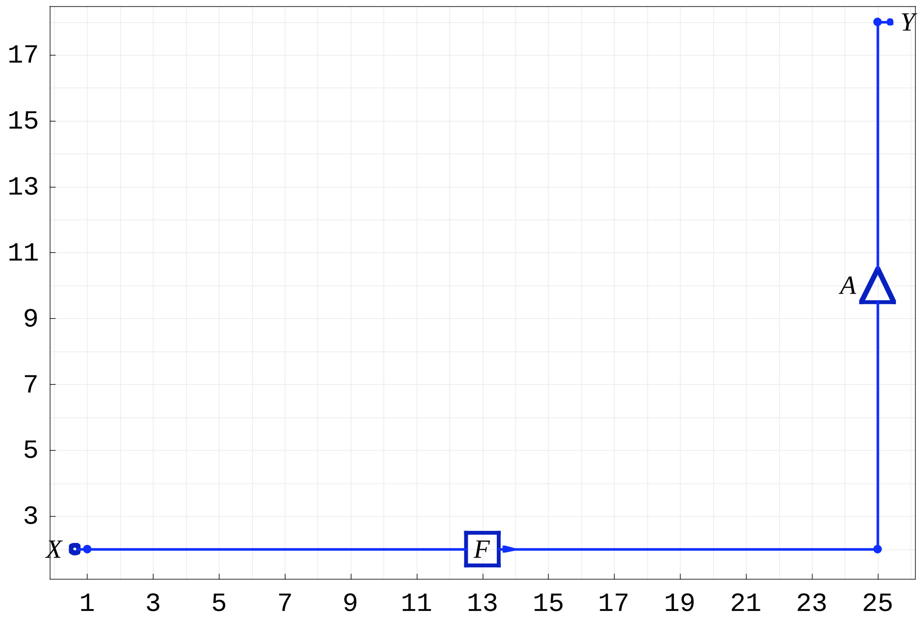
`FontSize`

`Frame`

`GridLines`

Consider an example schematic:

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%];
```



Here is a step-by-step procedure for setting the `ShowSchematic` drawing options.

**1.** Place the insertion point at the end of the `ShowSchematic` command just before the rightmost "]". In the example below, we marked the insertion point by red vertical separator |.

```
ShowSchematic[%|];
```

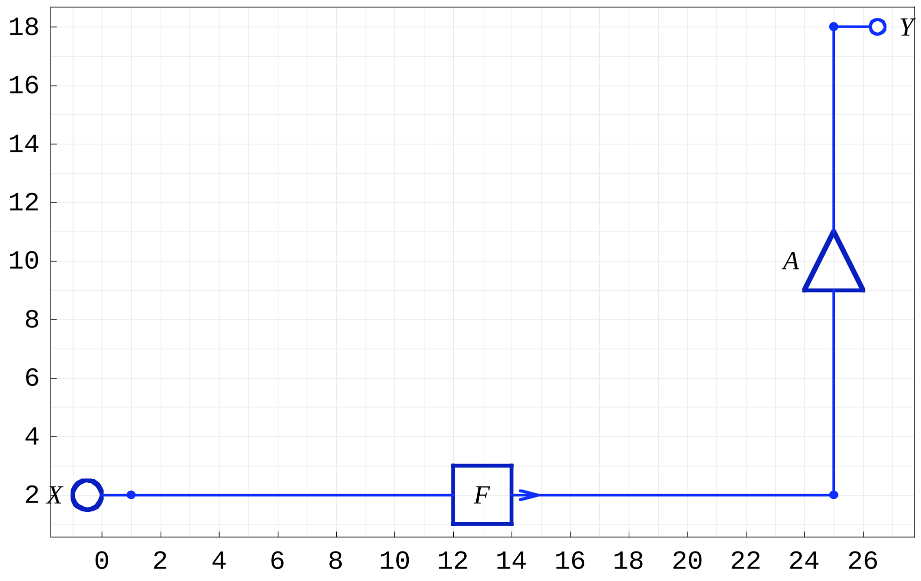
**2.** Click the button `, ElementScale` to change the element size of all elements.

The line with `ShowSchematic` changes:

```
ShowSchematic[%,ElementScale → 2];
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%, ElementScale → 2];
```



**3.** Place the insertion point at the end of the `ShowSchematic` command just before the last "]". In the example below, we marked the insertion point by red vertical separator |.

```
ShowSchematic[%,ElementScale → 2|];
```

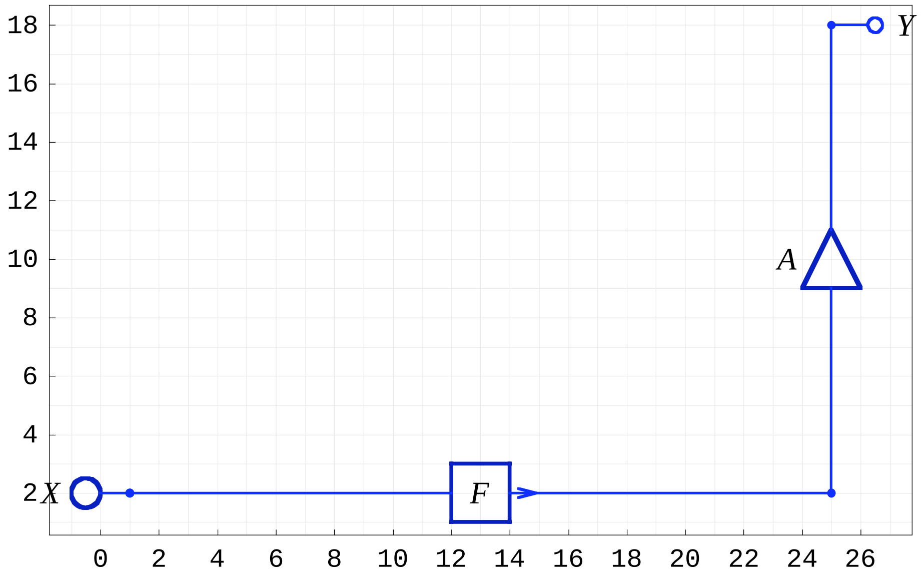
**4.** Click the button `, FontSize` to change the font size of all elements.

The line with `ShowSchematic` changes:

```
ShowSchematic[%,ElementScale → 2,FontSize → 12];
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%, ElementScale → 2, FontSize → 12];
```



**5.** Place the insertion point at the end of the `ShowSchematic` command just before the rightmost "]". In the example below, we marked the insertion point by red vertical separator |.

```
ShowSchematic[%,ElementScale → 2,FontSize → 12|];
```

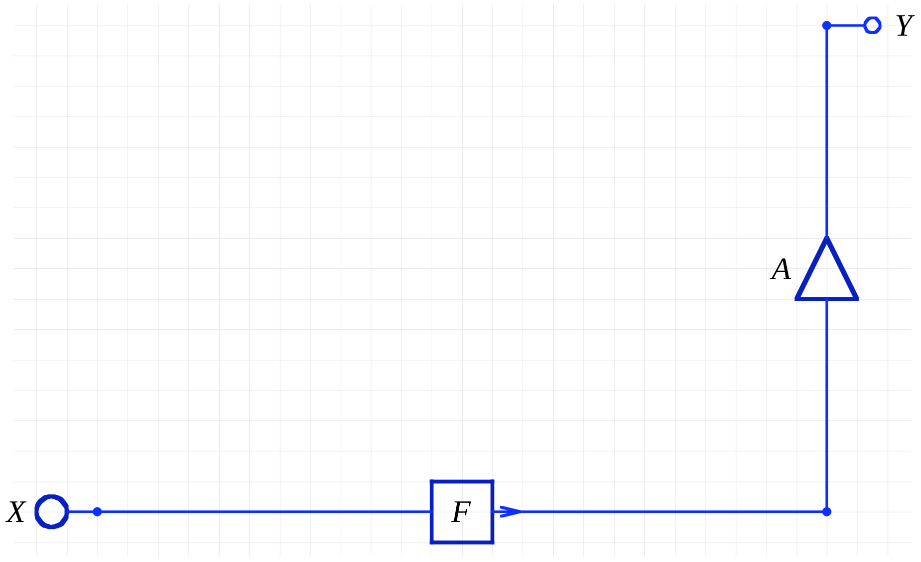
**6.** Click the button `, Frame` to remove the frame.

The line with `ShowSchematic` changes:

```
ShowSchematic[%,ElementScale→2,FontSize→12,Frame → False];
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%, ElementScale → 2, FontSize → 12, Frame → False];
```



**7.** Place the insertion point at the end of the `ShowSchematic` command just before the last "]". In the example below, we marked the insertion point by red vertical separator |.

```
ShowSchematic[%,ElementScale→2,FontSize→12,Frame→False|];
```

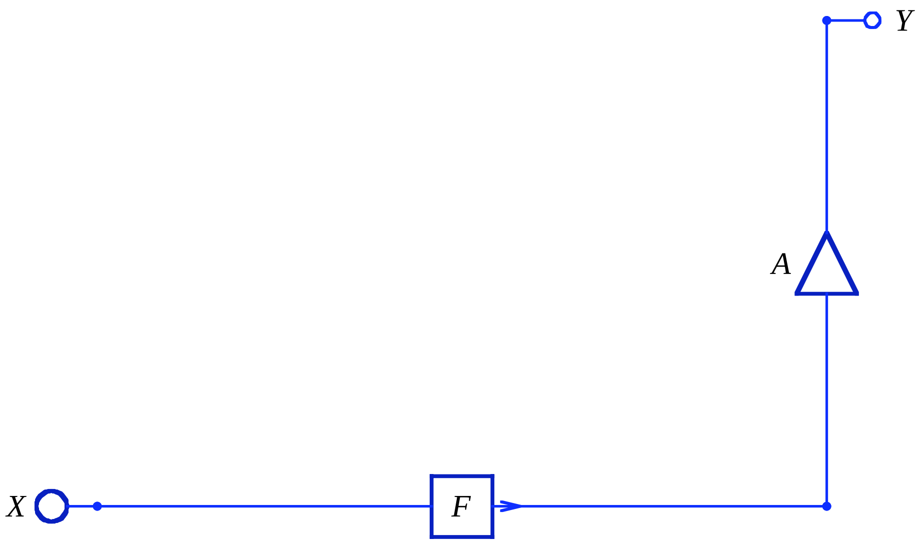
**8.** Click the button `, GridLines` to remove grid.

The line with `ShowSchematic` changes:

```
ShowSchematic[%,ElementScale→2,FontSize→12,Frame→False,GridLines → None];
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%, ElementScale → 2, FontSize → 12, Frame → False, GridLines → None];
```



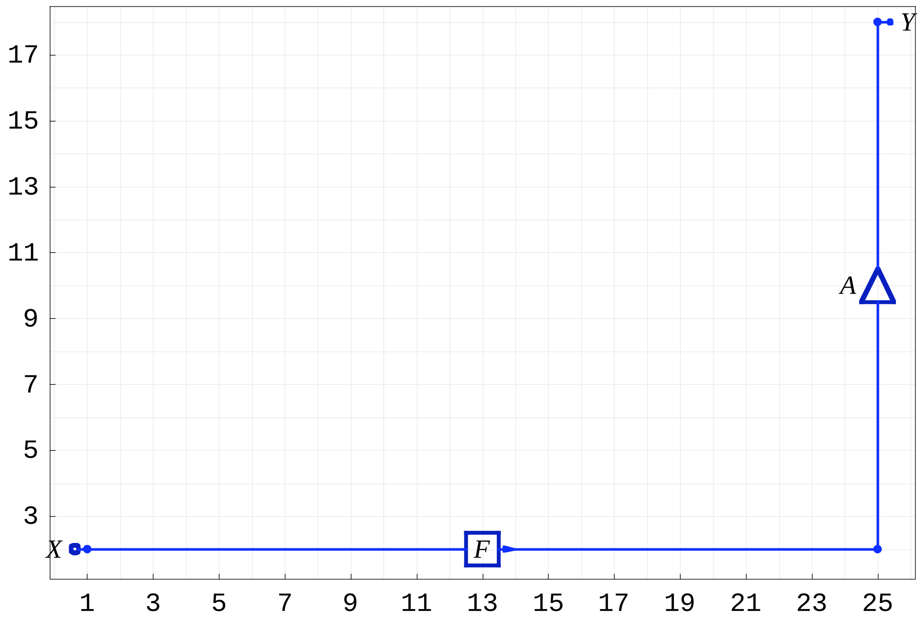
You can edit the specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in the specification and start editing.

## 13.14. Setting `PlotRange`

Schematic Options palette automates the plot range selection by mouse point-and-click.

Consider an example schematic:

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%];
```



Here is a step-by-step procedure for setting plot range.

**1.** Place the insertion point at the end of the `ShowSchematic` command just before the rightmost "]". In the example below, we marked the insertion point by red vertical separator |.

```
ShowSchematic[%|];
```

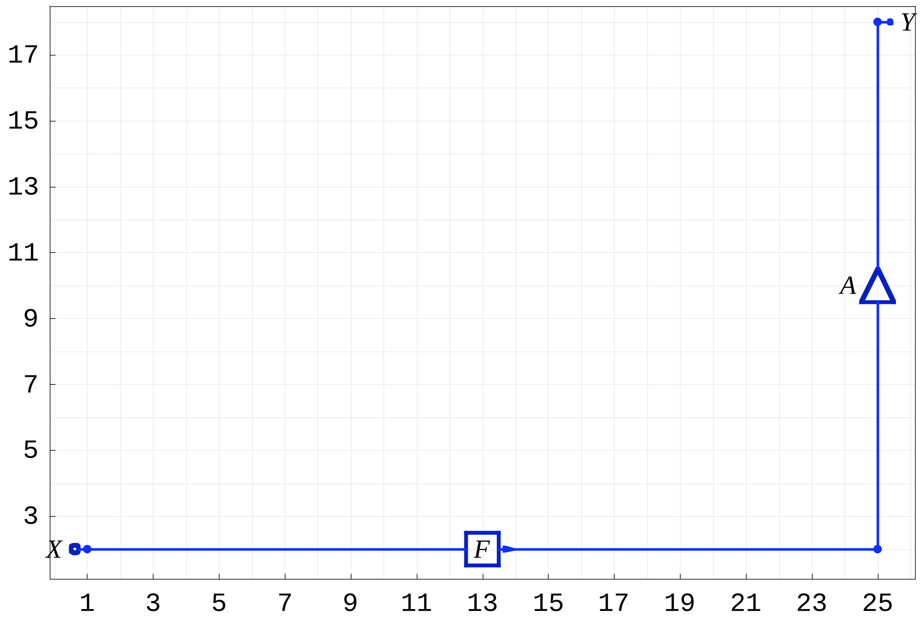
**2.** Click the button `, PlotRange → All` to insert the `PlotRange` option.

The line with `ShowSchematic` changes:

```
 ShowSchematic[%,PlotRange→All];
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%, PlotRange → All];
```



**3.** Select the option parameter `All`.

```
 ShowSchematic[%,PlotRange→ All ];
```

**4.** Click the button `{{x1, x2}, {y1, y2}}` to insert range by specifying a rectangular area.

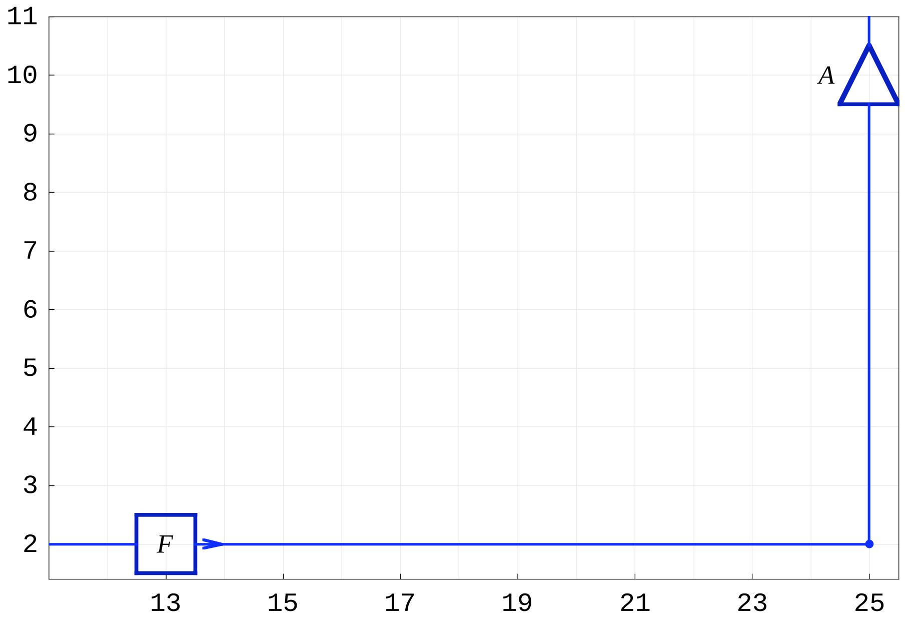
**5.** Move the mouse over the drawing workspace. The text `"Click and Drag"` is displayed in the window's status area.

**6.** Press and hold the mouse button, say when the mouse position is over the coordinate {11, 1.4}. Drag the mouse to specify the second coordinate. Release the mouse button, say at {25.5, 11}.

The selected coordinate is pasted into the line with `ShowSchematic` instead of the text `All`. The drawing workspace does not change.

Click the button `Redraw` to update the drawing workspace. The cell will be evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
ShowSchematic[%, PlotRange → {{11, 25.5}, {1.4, 11}}];
```



**7.** Place the insertion point in the line with the `ShowSchematic` command after the text `PlotRange→.` Click three times, and the plot range coordinates are selected:

```
{{11, 25.5}, {1.4, 11}}
```

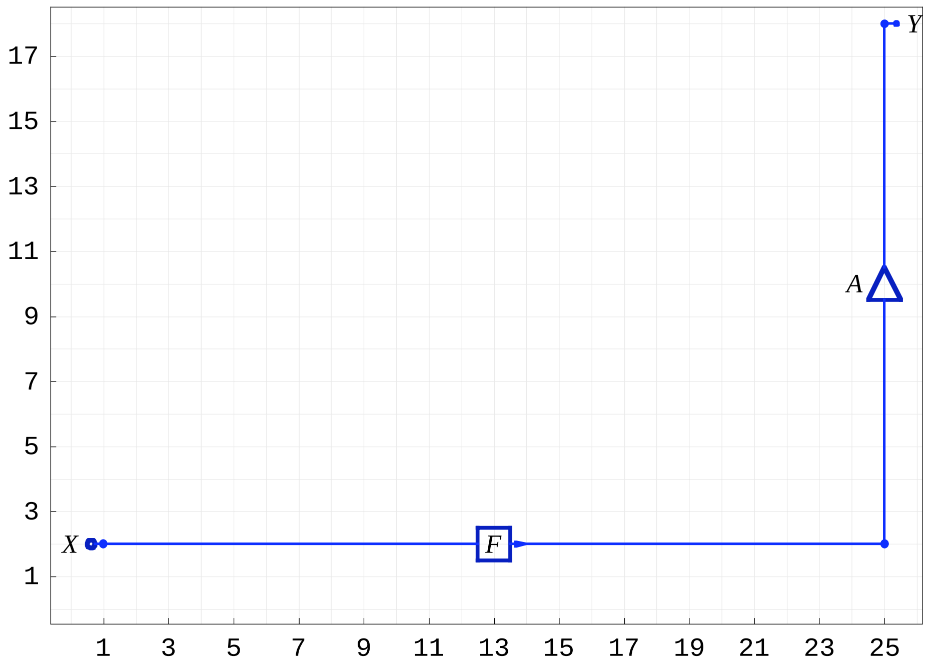
**8.** Click the button `Automatic` to insert the `PlotRange` option Automatic.

The line with `ShowSchematic` changes:

```
ShowSchematic[%,PlotRange→Automatic];
```

The whole cell will be automatically evaluated producing a new graphic output cell below the input cell.

```
mySchematic = {
{"Input", {1, 2}, X}, {"Output", {25, 18}, Y},
{"Function", {{1, 2}, {25, 2}}, F, ""},
{"Multiplier", {{25, 2}, {25, 18}}, A, ""}};
 ShowSchematic[%, PlotRange → Automatic];
```



You can edit the specification, the values and options, in the same way you edit *Mathematica* cells. For example, you can place your cursor somewhere in the specification and start editing.

## 13.15. Simultaneous Drawing of Combined Schematics

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

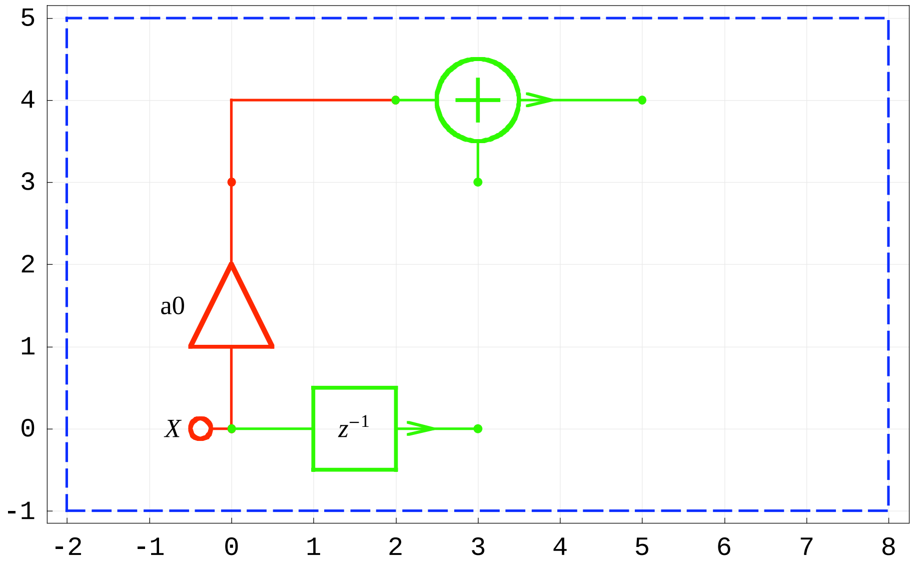
Suppose that you want to draw two or more schematics that will be combined into another schematic.

Consider three schematics `firstStageSchematic`, `lastStageSchematic`, and `basicStageSchematic`.

```
firstStageSchematic = {{"Input", {0, 0}, X, "", PlotStyle → {{Hue[0]}, {Hue[0]}}},
   {"Multiplier", {{0, 0}, {0, 3}}, a0, "", PlotStyle → {{Hue[0]}, {Hue[0]}}},
   {"Line", {{0, 3}, {0, 4}, {2, 4}}, PlotStyle → {{Hue[0]}, {Hue[0]}}}};

lastStageSchematic = {

   {"Polyline", {{-2, -1}, {8, -1}, {8, 5}, {-2, 5}, {-2, -1}}}};

basicStageSchematic =
  {{"Delay", {{0, 0}, {3, 0}}, 1, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}},

   {"Adder", {{2, 4}, {3, 3}, {5, 4}, {3, 5}},
    {1, 1, 2, 0}, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}}};
```

A combined schematic `combinedSchematic` can be made of individual schematics using the *Mathematica* function `Join`.

```
combinedSchematic =
  Join[firstStageSchematic, basicStageSchematic, lastStageSchematic];
ShowSchematic[%];
```



Suppose that you want to add a new Output element in `lastStageSchematic` with the same coordinate as the output of the Adder element in `combinedSchematic`. Here is a step-by-step procedure to accomplish this task.

**1.** Place the insertion point in the empty line in the specification `lastStageSchematic`. In the example below, we marked the insertion point by red vertical separator |.

```
lastStageSchematic = {
   |
   {"Polyline", {{-2, -1}, {8, -1}, {8, 5}, {-2, 5}, {-2, -1}}}};
```

**2.** Move the mouse over the drawing workspace below `combinedSchematic`. Click the button `Output` on the palette.

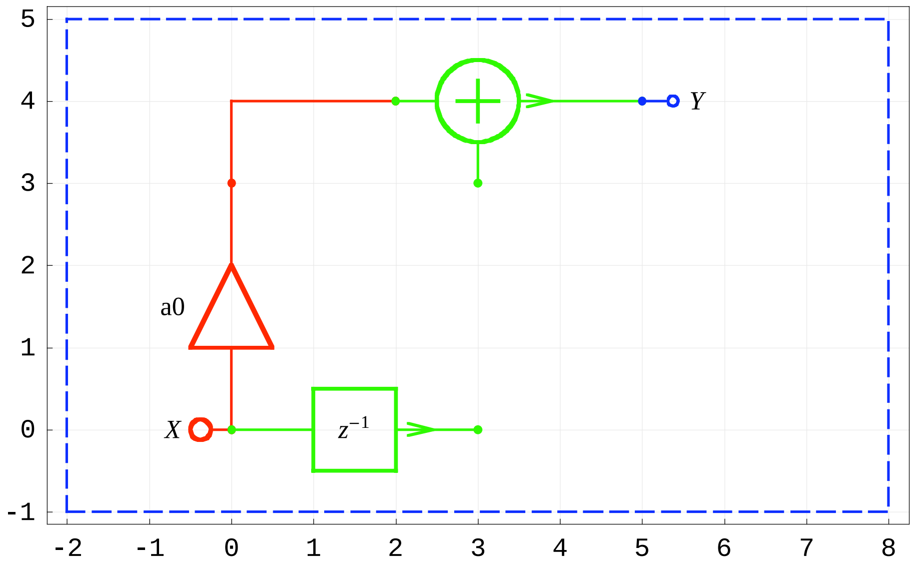
**3.** Click once when the mouse position is over the coordinate {5, 4} — the output of the Adder element. The coordinate {5,4} is selected, and it appears in the Output-element specification that is pasted at the current insertion point:

```
lastStageSchematic = {
   {"Output", {5, 4}, Y, "", TextOffset -> {-1, 0}},

   {"Polyline", {{-2, -1}, {8, -1}, {8, 5}, {-2, 5}, {-2, -1}}}};
```

**4.** Click the button `Redraw` to update the `lastStageSchematic` specification.

**5.** Place the insertion point into the `combinedSchematic` specification and click the button `Redraw` to draw `combinedSchematic`:

```
combinedSchematic =
  Join[firstStageSchematic, basicStageSchematic, lastStageSchematic];
ShowSchematic[%];
```



Assume that you want to add a new Multiplier element in `basicStageSchematic`. Here is a step-by-step procedure to do this.

**1.** Place the insertion point in the empty line in the `basicStageSchematic` specification. In the example below, we marked the insertion point by red vertical separator |.

```
basicStageSchematic =
 {{"Delay", {{0, 0}, {3, 0}}, 1, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}},
  |
   {"Adder", {{2, 4}, {3, 3}, {5, 4}, {3, 5}},
    {1, 1, 2, 0}, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}}};
```

**2.** Move the mouse over the drawing workspace below `combinedSchematic`. Click the button `Mult` on the palette.

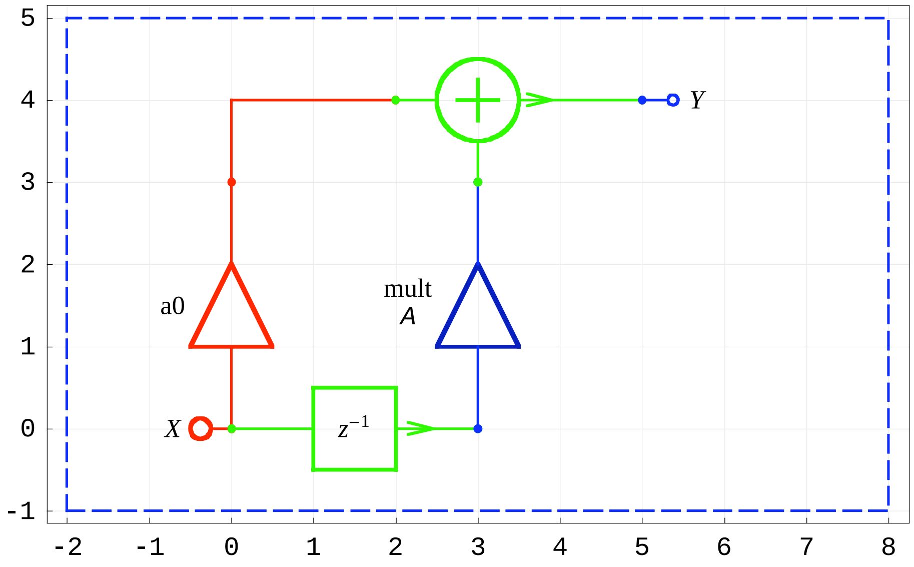
**3.** Press and hold the mouse button when the mouse position is over the coordinate {3,0} — the output of the Delay element. Drag the mouse to specify the second coordinate — input of the Adder element. Release the mouse button when the mouse position is over the coordinate {3,3}. The selected coordinates appear in the Multiplier-element specification that is pasted at the current insertion point.

```
basicStageSchematic =
  {{"Delay", {{0, 0}, {3, 0}}, 1, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}},
   {"Multiplier", {{3, 0}, {3, 3}}, A, "mult"},

   {"Adder", {{2, 4}, {3, 3}, {5, 4}, {3, 5}},
    {1, 1, 2, 0}, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}}};
```

**4.** Click the button `Redraw` to update the `basicStageSchematic` specification.

**5.** Place the insertion point in the `combinedSchematic` specification and click the button `Redraw` to draw `combinedSchematic`:

```
combinedSchematic =
  Join[firstStageSchematic, basicStageSchematic, lastStageSchematic];
ShowSchematic[%];
```



You can edit the specifications, the values and options, in the same way you edit *Mathematica* cells.

## 13.16. Draw Large Schematics Using `PlotRange`

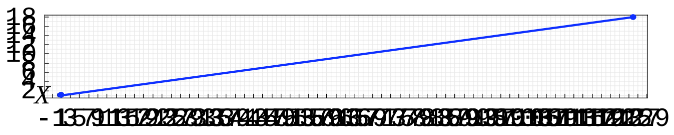
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

Suppose that you want to add a new Output element in a large schematic that cannot be nicely presented in the drawing workspace.

Consider the schematic `largeSchematic`.

```
largeSchematic = {{"Input", {1, 1}, X},

   {"Line", {{1, 1}, {126, 18}}}};
ShowSchematic[%];
```

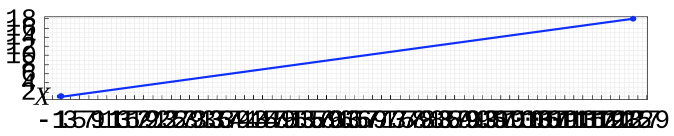


Here is a step-by-step procedure how to add an element when the grid is too dense.

**1.** Place the insertion point at the end of the `ShowSchematic` command just before the rightmost "]".

**2.** Click the button `, PlotRange → All` on the palette to insert the `PlotRange` option.

```
largeSchematic = {{"Input", {1, 1}, X},

   {"Line", {{1, 1}, {126, 18}}}};
ShowSchematic[%, PlotRange → All];
```



**3.** Select the option parameter `All`.

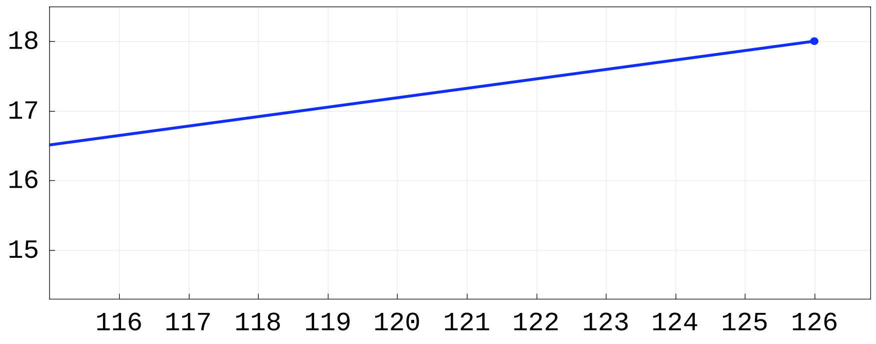
```
ShowSchematic[%,PlotRange→ All ];
```

**4.** Click the button `{{x1, x2}, {y1, y2}}` to insert range by specifying a rectangular area.

**5.** Move the mouse over the drawing workspace.

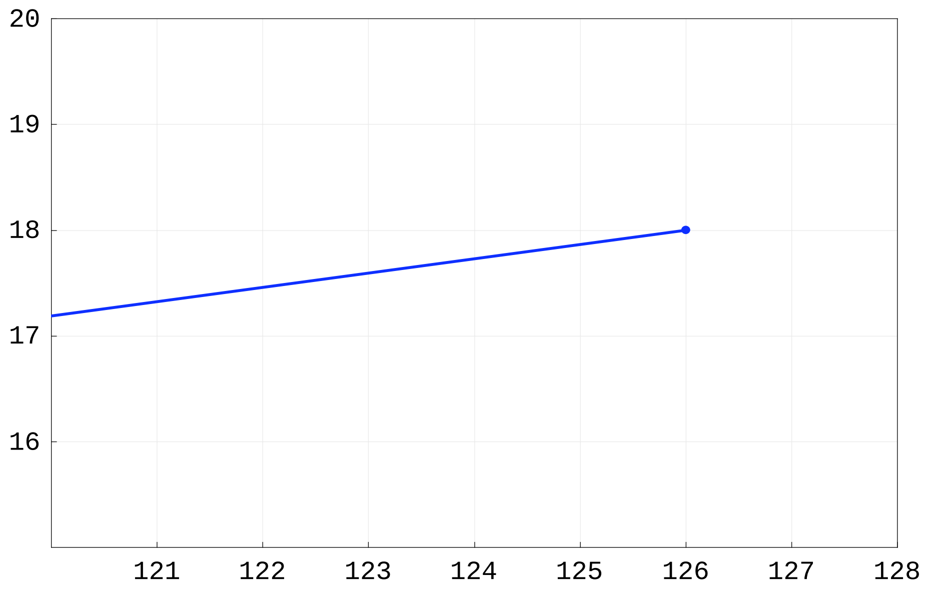
**6.** Press and hold the mouse button when the mouse position is over the first coordinate, say {115, 126.8}. Drag the mouse to specify the second coordinate. Release the mouse button, say at {14.3, 18.5}. Click the button `Redraw` to update the drawing workspace. The cell will be evaluated producing a new graphic output cell below the input cell.

```
largeSchematic = {{"Input", {1, 1}, X},

   {"Line", {{1, 1}, {126, 18}}}};
ShowSchematic[%, PlotRange → {{115, 126.8}, {14.3, 18.5}}];
```



**7.** You can manually edit the plot range. For example, you can change the plot range to {{120, 128}, {15, 20}}. Click the button `Redraw` to update the drawing workspace.

```
largeSchematic = {{"Input", {1, 1}, X},

   {"Line", {{1, 1}, {126, 18}}}};
ShowSchematic[%, PlotRange → {{120, 128}, {15, 20}}];
```

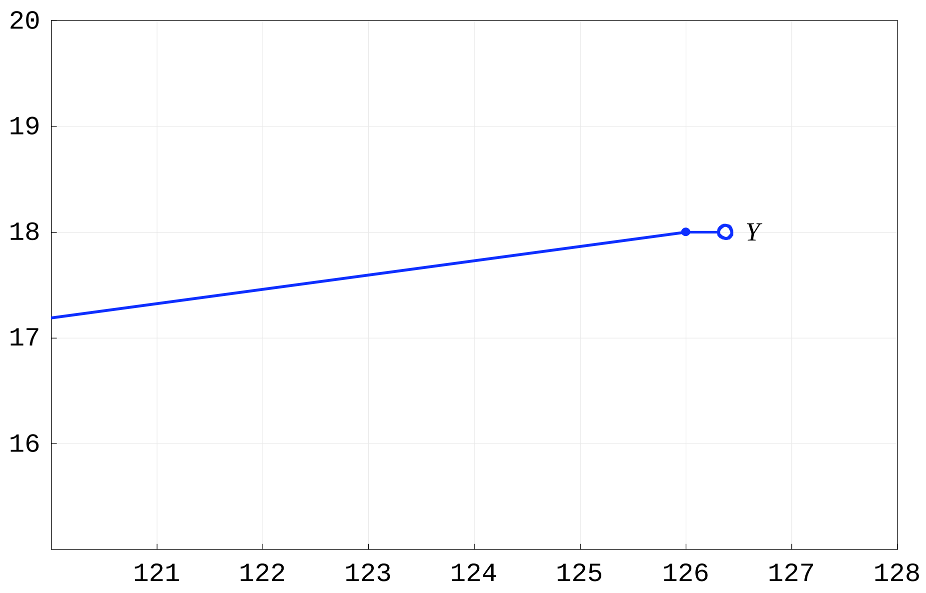


**8.** Place the insertion point in the empty line in the `largeSchematic` specification.

**9.** Move the mouse over the drawing workspace below `largeSchematic` specification. Click the button `Output`.

**10.** Click once when the mouse position is over the coordinate {126, 18}. The coordinate {126,18} is selected, and it appears in the Output-element specification that is pasted at the current insertion point. Click the button `Redraw` to update the drawing workspace.

```
largeSchematic = {{"Input", {1, 1}, X},
   {"Output", {126, 18}, Y, "", TextOffset -> {-1, 0}},

   {"Line", {{1, 1}, {126, 18}}}};
ShowSchematic[%, PlotRange → {{120, 128}, {15, 20}}];
```



## 13.17. Draw Large Schematics with Repeated Subschematics

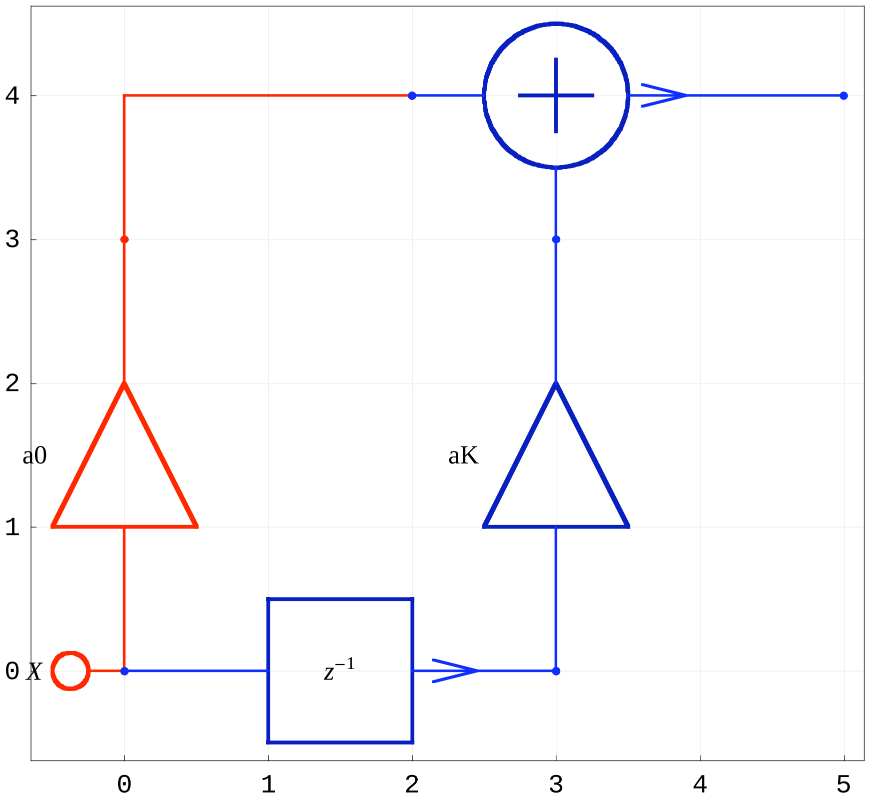
Some large schematics consist of replicas of the subschematics. It is not necessary to manually insert all elements. Instead, you can draw smaller parts that constitute the large system, and combine them into the desired schematic.

Consider three subschematics `firstStageSchematic`, `lastStageSchematic`, and `basicStageSchematic`.

```
firstStageSchematic = {{"Input", {0, 0}, X, "", PlotStyle → {{Hue[0]}, {Hue[0]}}},
   {"Multiplier", {{0, 0}, {0, 3}}, a0, "", PlotStyle → {{Hue[0]}, {Hue[0]}}},
   {"Line", {{0, 3}, {0, 4}, {2, 4}}, PlotStyle → {{Hue[0]}, {Hue[0]}}}};

lastStageSchematic = {{"Output", {5, 4}, Y, "", PlotStyle → {{Hue[0.3]}, {Hue[0.3]}}}};

basicStageSchematic = {{"Delay", {{0, 0}, {3, 0}}, 1},
   {"Multiplier", {{3, 0}, {3, 3}}, aK},
   {"Adder", {{2, 4}, {3, 3}, {5, 4}, {3, 5}}, {1, 1, 2, 0}}};
```

A combined schematic `combinedSchematic` can be made of individual schematics by using the *Mathematica* function `Join`.

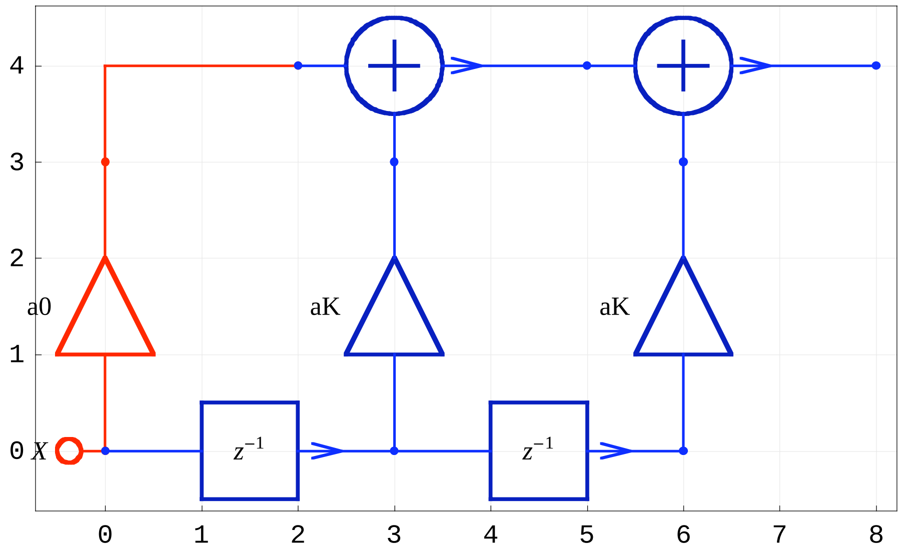
```
combinedSchematic = Join[
   firstStageSchematic,
   basicStageSchematic];
ShowSchematic[%];
```



Suppose that you want to add the second `basicStageSchematic` in such a way that the Delay-element input of the second `basicStageSchematic` is connected to the Delay-element output of the first `basicStageSchematic`. The coordinates of the Delay-element input of `basicStageSchematic` is {0,0}; therefore, the coordinates of the Delay-element input of the second `basicStageSchematic` should be translated to the right by {3,0}.

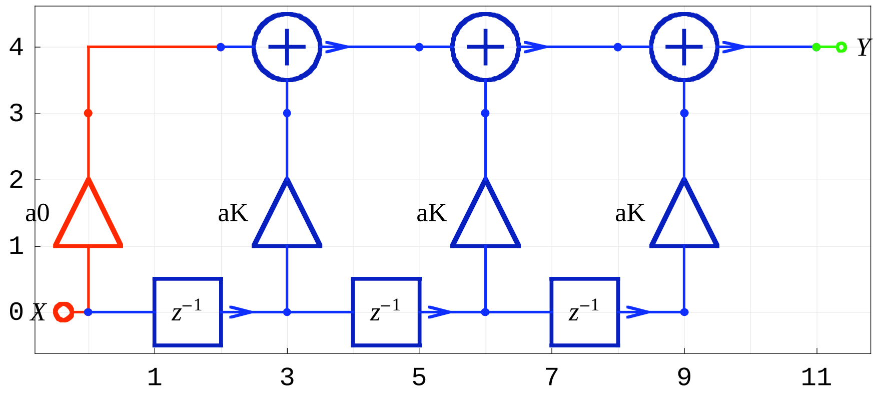
`TranslateSchematic` is used to translate `basicStageSchematic`. The combined schematic `combinedSchematic` contains now one instance of `firstStageSchematic` and two instances of `basicStageSchematic`.

```
combinedSchematic = Join[
   firstStageSchematic,
   basicStageSchematic,
   TranslateSchematic[basicStageSchematic, {3, 0}]];
ShowSchematic[%];
```



Here is `combinedSchematic` with three instances of `basicStageSchematic` and the additional subschematic `lastStageSchematic`.

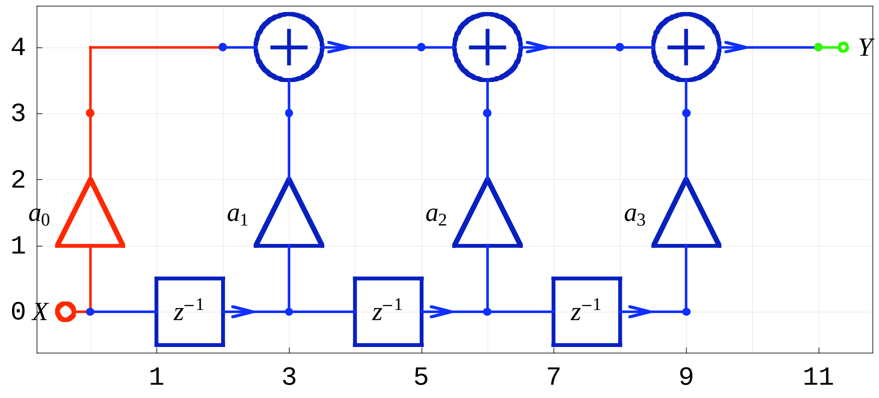
```
combinedSchematic = Join[
   firstStageSchematic,
   basicStageSchematic,
   TranslateSchematic[basicStageSchematic, {3, 0}],
   TranslateSchematic[basicStageSchematic, {6, 0}],
   TranslateSchematic[lastStageSchematic, {6, 0}]];
ShowSchematic[%];
```



`TranslateSchematic` is be used to translate `lastStageSchematic` to the proper place.

You can edit the specifications, the values and options, in the same way you edit *Mathematica* cells. For example, multiplier coefficients can be replaced by more appropriate expressions:

```
combinedSchematic = Join[
   firstStageSchematic /. a0 → a₀,
   basicStageSchematic /. aK → a₁,
   TranslateSchematic[basicStageSchematic /. aK → a₂, {3, 0}],
   TranslateSchematic[basicStageSchematic /. aK → a₃, {6, 0}],
   TranslateSchematic[lastStageSchematic, {6, 0}]];
ShowSchematic[%];
```



## 13.18. Automated Drawing of Systems with Repeated Subschematics

Once when you have drawn subschematics, and when you find out that they can be used to build large schematics with repeated subschematics, you can write a code to automate drawing for an arbitrary number of repeated stages. Assume that you want to design a system with 7 stages and 8 parameters.

```
numberOfStages = 7;
```

The parameter symbols can be automatically generated as follows:

```
parameterSymbols = UnitSymbolicSequence[numberOfStages + 1, a, 0] // Flatten
```

```
{a0, a1, a2, a3, a4, a5, a6, a7}
```

Consider three subschematics `firstStageSchematic`, `lastStageSchematic`, and `basicStageSchematic`.

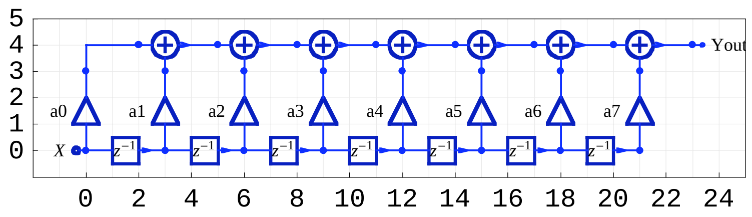
```
firstStageSchematic = {{"Input", {0, 0}, X},
   {"Multiplier", {{0, 0}, {0, 3}}, a0},
   {"Line", {{0, 3}, {0, 4}, {2, 4}}}};

lastStageSchematic = {{"Output", {5, 4}, Yout}};

basicStageSchematic = {{"Delay", {{0, 0}, {3, 0}}, 1},
   {"Multiplier", {{3, 0}, {3, 3}}, aK},
   {"Adder", {{2, 4}, {3, 3}, {5, 4}, {3, 5}}, {1, 1, 2, 0}}};
```

A combined schematic `combinedSchematic` can be made of subschematics using `Join` and `TranslateSchematic`.

```
combinedSchematic = Join[firstStageSchematic,
   TranslateSchematic[lastStageSchematic, {(numberOfStages - 1) * 3, 0}]];
Do[combinedSchematic = Join[combinedSchematic,
     TranslateSchematic[
      basicStageSchematic /. d → 1 /. aK → parameterSymbols[[k + 1]], {(k - 1) * 3, 0}]];
  , {k, numberOfStages}];
```

`PlotRange` and `FontSize` refine the drawing:

```
ShowSchematic[combinedSchematic,
  PlotRange → {{-2, numberOfStages * 3 + 4}, {-1, 5}}, FontSize → 7];
```



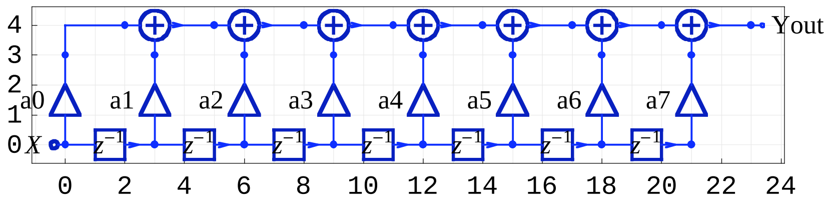
## 13.19. Save and Load Schematic Specification

Sometimes you may prefer to draw a previously generated schematic without seeing the drawing procedure. In that case, you should save the schematic specification in a disk file, say `"c:\\temp\\mySavedSchematics.m"`, with

```
Save["c:\\temp\\mySavedSchematics.m", combinedSchematic]
```

You can load the saved schematic with

```
Get["c:\\temp\\mySavedSchematics.m"];
ShowSchematic[combinedSchematic]
```



## 13.20. Predefined Schematics

*SchematicSolver* comes with functions that create schematics important for practice.

This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
Needs["SchematicSolver`"];
```

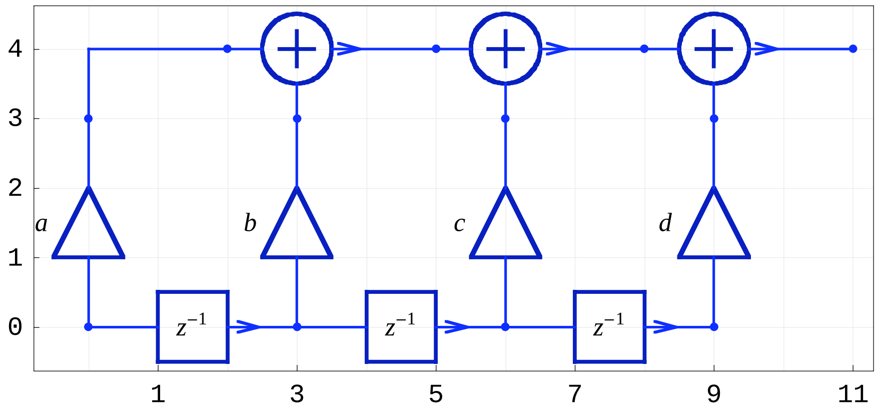
`DirectFormFIRFilterSchematic` creates the schematic of the *Direct Form FIR* filter with an arbitrary order and parameters.

Here is an example schematic specification with 4 symbolic parameters:

```
{mySchematic, inpCoords, outCoords} = DirectFormFIRFilterSchematic[{a, b, c, d}]
```

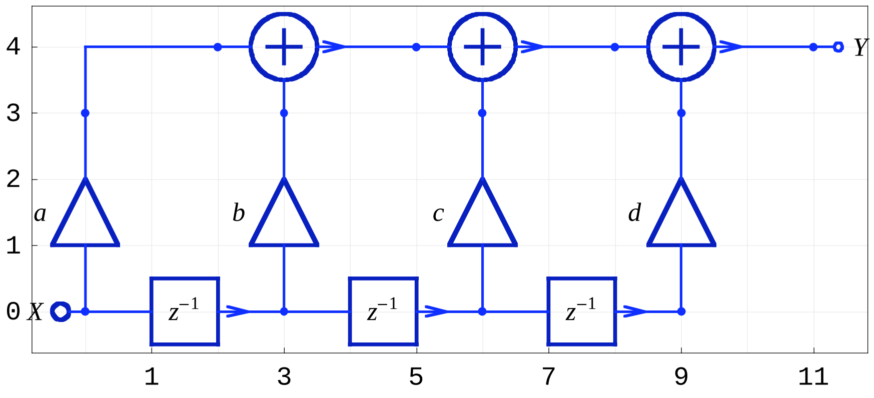
```
{{{Multiplier, {{0, 0}, {0, 3}}, a}, {Line, {{0, 3}, {0, 4}, {2, 4}}},
  {Delay, {{0, 0}, {3, 0}}, 1}, {Multiplier, {{3, 0}, {3, 3}}, b},
  {Adder, {{2, 4}, {3, 3}, {5, 4}, {3, 5}}, {1, 1, 2, 0}},
  {Delay, {{3, 0}, {6, 0}}, 1}, {Multiplier, {{6, 0}, {6, 3}}, c},
  {Adder, {{5, 4}, {6, 3}, {8, 4}, {6, 5}}, {1, 1, 2, 0}},
  {Delay, {{6, 0}, {9, 0}}, 1}, {Multiplier, {{9, 0}, {9, 3}}, d},
  {Adder, {{8, 4}, {9, 3}, {11, 4}, {9, 5}}, {1, 1, 2, 0}}}, {{0, 0}}, {{11, 4}}}
```

```
ShowSchematic[mySchematic]
```



You can add input and output to form the system:

```
mySystem = Join[
   {{"Input", First[inpCoords], X}},
   mySchematic,
   {{"Output", First[outCoords], Y}}
  ];
% // ShowSchematic
```



Note that the coordinates of input and output have been returned by `DirectFormFIRFilterSchematic`.

# 14. Reference Guide

## 14.1. List of *SchematicSolver* Functions

Load the *SchematicSolver* package with

```
Needs["SchematicSolver`"]
```

*SchematicSolver* functions have short descriptions that document their basic usage. The usage message for a function *fnct* is retrieved when you type ?*fnct*. When you click on a function name below, the usage of that function appears in the next cell.

## Elements

| | |
|---|---|
| DrawElement | ShowSchematic |
| ElementScale | TextDirection |
| ElementSize | TextOffset |
| PolylineDashing | z |
| ShowArrowTail | $VersionSchematicSolverSchematicElements |
| ShowNodes | |

## Analysis

ContinuousSystemEquations
ContinuousSystemResponse
ContinuousSystemSignals
ContinuousSystemTransferFunction
DiscreteSystemEquations
DiscreteSystemResponse
DiscreteSystemSignals
DiscreteSystemTransferFunction
PrintFloatingPorts
s
Verbose
Y
z
$VersionSchematicSolverSchematicAnalysis

## Utilities

CheckElementSyntax
CheckSchematicSyntax
dBMagnitudePlot
DiscreteSystemDisplayForm
DiscreteSystemFrequencyResponse
DiscreteSystemMagnitudeResponsePlot
DiscreteSystemProcessingSISO
f
TranslateSchematic
$VersionSchematicSolverSchematicUtilities

## Implementation

AdjustSchematicCoordinates
DemultiplexSequence
DiscreteSystemImplementation
DiscreteSystemImplementationEquations
DiscreteSystemImplementationModule
DiscreteSystemImplementationProcessing
DiscreteSystemImplementationSummary
DiscreteSystemSimulation
DownsampleSequence
FirstSampleIndex
ListToSequence
MultiplexDataList
MultiplexSequence
MultirateDownsampleSequence
NormalizedSpectrum
Power2
previousSample
SequenceDiscreteFourierTransform
SequenceDiscreteFourierTransformMagnitudePlot
SequenceFourierTransform
SequenceFourierTransformMagnitudePlot
SequenceLineThickness
SequencePlot
SequencePointSize
SequenceSamplingFrequency
SequenceToList
StemPlot
UndefinedSymbolQ
UnitExponentialSequence
UnitImpulseSequence
UnitNoiseSequence
UnitRampSequence
UnitSineSequence
UnitStepSequence
UnitSymbolicSequence
UpsampleSequence
ValidImplementationModuleNameQ
$VersionSchematicSolverSchematicImplementation

## Album

DelayElementValue
DirectFormFIRFilterSchematic
DoubleDelayDirectFormFIRFilterSchematic
HalfbandDirectFormFIRFilterSchematic
HighSpeedIIR3FIRHalfbandFilterSchematic
HilbertTransformerDirectFormFIRSchematic
TestDiscreteLinearSISOAlbumSchematic
TransposedDirectForm2IIRBiquadSchematic
TransposedDirectForm2IIRFilterSchematic
$VersionSchematicSolverSchematicAlbum

## Figures

## 14.2. Palettes

### Continuous Elements Palette

Open the **Palettes** submenu of the **File** menu, and choose the **ContinuousElements** command to open the palette Continuous Elements.

### Discrete Elements Palette

Open the **Palettes** submenu of the **File** menu, and choose the **DiscreteElements** command to open the palette Discrete Elements.

### Discrete Nonlinear Elements Palette

Open the **Palettes** submenu of the **File** menu, and choose the **DiscreteNonlinear** command to open the palette Discrete Nonlinear Elements.

### Schematic Options Palette

Open the **Palettes** submenu of the **File** menu, and choose the **SchematicOptions** command to open the palette Schematic Options.

## 14.3. Showing Schematics and Schematic Elements

### ShowSchematic

`ShowSchematic` draws schematic from a schematic specification.

`ShowSchematic`[*schematicSpecification*, *options*]

`ShowSchematic`[*schematicSpecification*] defaults to `ShowSchematic`[*schematicSpecification*, `ElementScale→1, FontSize→Automatic, Frame→True, GridLines→Automatic, PlotRange→All`]

*schematicSpecification* is a schematic specification that represents the system; it is a list of element specifications.

Supported elements are Adder, Amplifier, Arrow, Block, Delay, Function, Input, Integrator, Line, Modulator, Multiplier, Node, Output, Polyline, and Text.

*options* are the `ShowSchematic` options: `ElementScale`, `FontSize`, `Frame`, `GridLines`, and `PlotRange`.

`ElementScale` specifies the scale of all schematic elements.

`FontSize` specifies the font size of text for all schematic elements. See *The Mathematica Book* for details.

`Frame` specifies whether a frame should be drawn around the plot. `Frame → True` draws a frame with tick marks. `Frame → False` does not draw a frame with tick marks. See *The Mathematica Book* for details.

`GridLines` specifies the grid lines of the schematic. `GridLines->None` does not draw the grid lines. See *The Mathematica Book* for details.

`PlotRange` specifies the plot range of the schematic. See *The Mathematica Book* for details.

`Options[ShowSchematic]` gives a list of the current default settings for all options. You can reset the default using `SetOptions`[*function*, *option*→*value*]. For example, `SetOptions[ShowSchematic, Frame→False]`.

Element specification is a list of the form

{"*name*", *coordinates*, *value*, *label*, *elementOpts*}.

*name* is the name of an element: `Adder`, `Amplifier`, `Arrow`, `Block`, `Delay`, `Function`, `Input`, `Integrator`, `Line`, `Modulator`, `Multiplier`, `Node`, `Output`, `Polyline`, or `Text`. *name* should be enclosed within double quotation marks.

*coordinates* is a pair of numbers {*x*, *y*} or a list of pairs of numbers {{*x1*,*y1*}, {*x2*,*y2*}, ...} .

*value* is an expression that is the element value (e.g., the multiplier coefficient).

*label* is a string or expression to annotate the element.

The Arrow and Text elements do not have the label item. The Line and Polyline elements do not have the value and label items.

*elementOpts* are element options: `ElementSize`, `PlotStyle`, `PolylineDashing`, `ShowArrowTail`, `ShowNodes`, `TextDirection`, `TextOffset`, and `TextStyle`. See `DrawElement` for details.

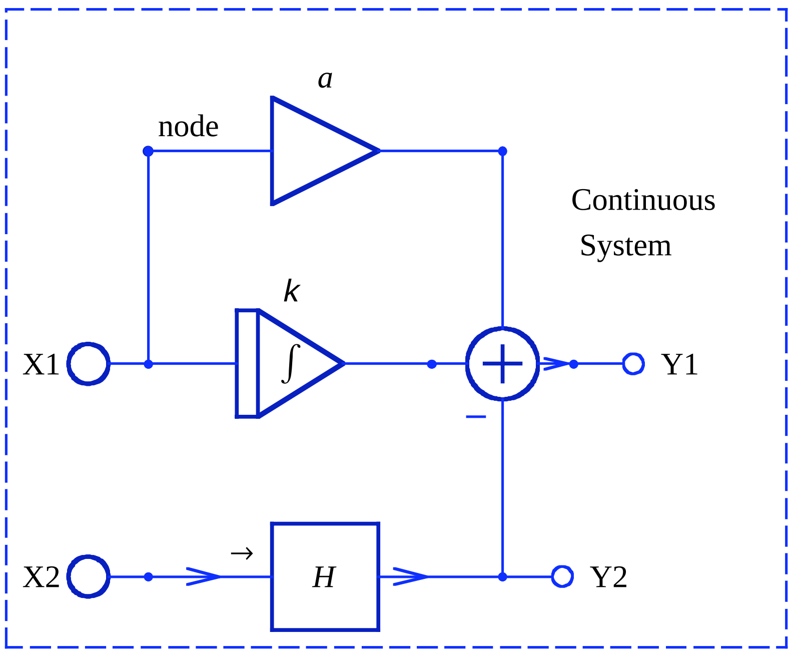
See also: `DrawElement`

## Examples

```
Needs["SchematicSolver`"]
```

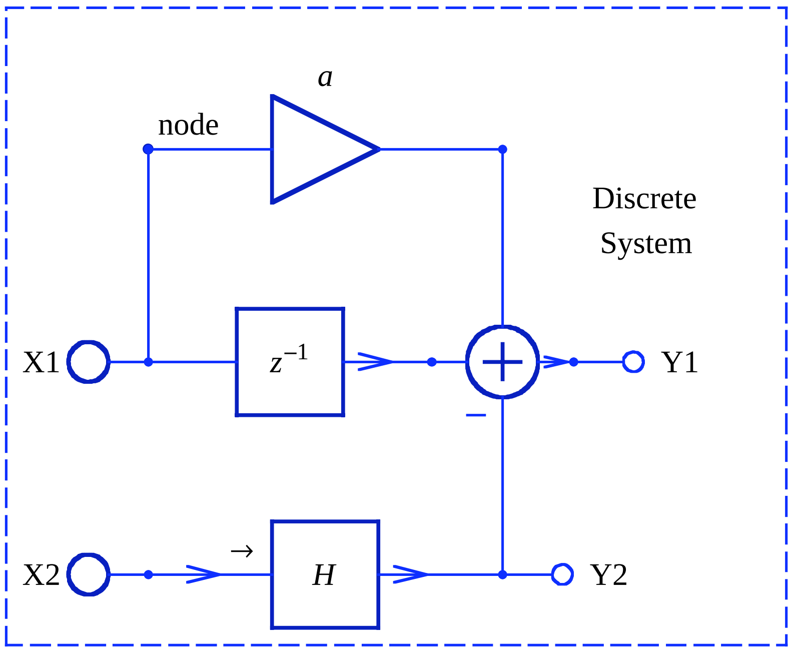
Here is the schematic specification of a continuous system:

```
continuousSchematic = {
  {"Input", {1, 4}, X1}, {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1}, {"Output", {6, 1}, Y2},
  {"Node", {1, 7}, node},
  {"Text", {8, 6}, "Continuous\n System"},
  {"Arrow", {{2, 1}, {1, 1}}, "→"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Amplifier", {{1, 7}, {6, 7}}, a},
  {"Integrator", {{1, 4}, {5, 4}}, k},
  {"Block", {{1, 1}, {6, 1}}, H},
  {"Polyline", {{-1, 0}, {-1, 9}, {10, 9}, {10, 0}, {-1, 0}}}};

ShowSchematic[continuousSchematic, ElementScale → 1.5, FontSize → 12,
 Frame → False, GridLines → None, PlotRange → {{-2, 11}, {-1, 10}}];
```



Here is the schematic specification of a discrete system:

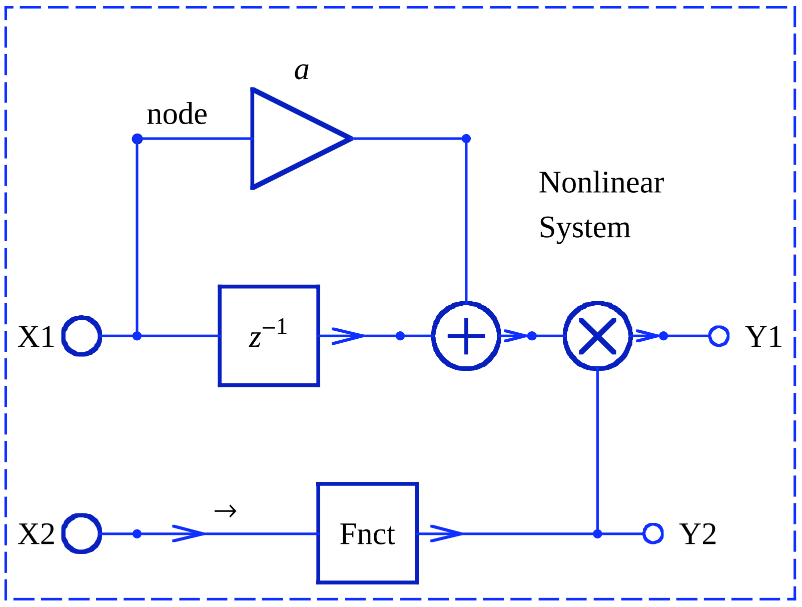
```
discreteSchematic = {
  {"Input", {1, 4}, X1}, {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1}, {"Output", {6, 1}, Y2},
  {"Node", {1, 7}, node},
  {"Text", {8, 6}, "Discrete\n System"},
  {"Arrow", {{2, 1}, {1, 1}}, "→"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Multiplier", {{1, 7}, {6, 7}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1},
  {"Block", {{1, 1}, {6, 1}}, H},
  {"Polyline", {{-1, 0}, {-1, 9}, {10, 9}, {10, 0}, {-1, 0}}}};

ShowSchematic[discreteSchematic, ElementScale → 1.5, FontSize → 12,
 Frame → False, GridLines → None, PlotRange → {{-2, 11}, {-1, 10}}];
```



Here is the schematic specification of a nonlinear system:

```
nonlinearSchematic = {
   {"Input", {1, 4}, X1}, {"Input", {1, 1}, X2},
   {"Output", {9, 4}, Y1}, {"Output", {8, 1}, Y2},
   {"Node", {1, 7}, node},
   {"Text", {8, 6}, " Nonlinear\n System"},
   {"Arrow", {{2, 1}, {1, 1}}, "→"},
   {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, 0, 2, 1}},
   {"Line", {{1, 4}, {1, 7}}},
   {"Multiplier", {{1, 7}, {6, 7}}, a},
   {"Delay", {{1, 4}, {5, 4}}, 1},
   {"Modulator", {{7, 4}, {8, 1}, {9, 4}, {8, 5}}, {1, 1, 2, 0}},
   {"Function", {{1, 1}, {8, 1}}, Fnct},
   {"Polyline", {{-1, 0}, {-1, 9}, {11, 9}, {11, 0}, {-1, 0}}}};
```

```
ShowSchematic[nonlinearSchematic, ElementScale → 1.5, FontSize → 12,
  Frame → False, GridLines → None, PlotRange → {{-2, 12}, {-1, 10}}];
```



## `DrawElement`

`DrawElement` creates a list of graphics specifications from which to draw an element.

`DrawElement[`*elementSpec*`]`

*elementSpec* is an element specification. Element specification is a list of the form

{"*name*", *coordinates*, *value*, *label*, *elementOpts*}.

*name* is the name of an element: `Adder`, `Amplifier`, `Arrow`, `Block`, `Delay`, `Function`, `Input`, `Integrator`, `Line`, `Modulator`, `Multiplier`, `Node`, `Output`, `Polyline`, or `Text`. *name* should be enclosed within double quotation marks.

*coordinates* is a pair of numbers {*x*, *y*} or a list of pairs of numbers {{*x1*,*y1*}, {*x2*,*y2*}, ...} .

*value* is an expression that is the element value (e.g., the multiplier coefficient).

*label* is a string or expression to annotate the element.

The Arrow and Text elements do not have the label item. The Line and Polyline elements do not have the value and label items.

*elementOpts* are element options: `ElementSize`, `PlotStyle`, `PolylineDashing`, `ShowArrowTail`, `Show-Nodes`, `TextDirection`, `TextOffset`, and `TextStyle`.

{"*name*", *coordinates*, *value*, *label*} defaults to

{"*name*", *coordinates*, *value*, *label*, `ElementSize→{1,1}`,

`PlotStyle→{{RGBColor[0,0,0.7]`, `Thickness[0.005]`, `PointSize[0.012]}`, `{RGBColor[0,0,1]`, `Thickness[0.0035]`, `PointSize[0.01]}}`,

`ShowNodes→True`, `TextDirection→{1,0}`, `TextOffset→Automatic`,

`TextStyle→{FontFamily→"Times"`, `FontSize→10}`,

`PolylineDashing→Dashing[{0.02,0.01}]`, `ShowArrowTail→True`}.

`ElementSize` specifies the size and aspect ratio of a schematic element.

`PlotStyle` specifies the style of lines and points to be plotted. Two specifications are given: one for the element shape (graphic symbol), and one for the element ports (lines connecting the graphic symbol and nodes). See the *Mathematica* help for details.

`ShowNodes` controls the appearance of element nodes.

`TextDirection` specifies the angle of the text rotation.

`TextOffset` specifies the position of the element value and label.

`TextStyle` specifies the text style and font options. See the *Mathematica* help for details.

`PolylineDashing` is an option for the Polyline-element specification that controls dashing.

`ShowArrowTail` is an option for the Arrow-element specification that controls the appearance of the arrow tail.

`Options[DrawElement]` gives a list of the current default settings for all options.

You can reset the default using `SetOptions[function, option→value]`. For example: `SetOptions[DrawElement, TextStyle→{FontFamily→"Helvetica", FontSize→9, FontColor→Hue[0.1]}]`.

See also: `ShowSchematic`

## ElementScale

`ElementScale` is an option for `ShowSchematic` that specifies the magnification of element dimensions for all schematic elements.

`ElementScale` → *scale*

*scale* is a number that specifies the amount of scaling.

`ElementScale`→1 is default. You can reset the default using

`SetOptions`[*function*, *option*→*value*].

For example, `SetOptions`[`ShowSchematic`, `ElementScale`→2].

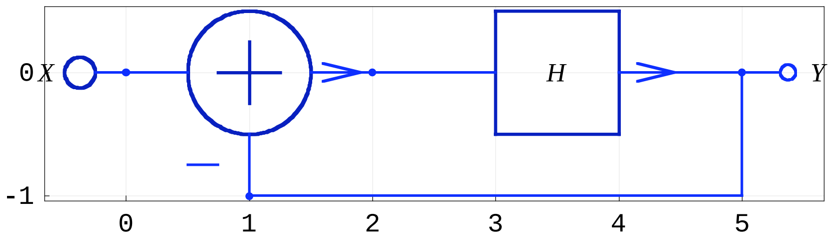
### Examples

```
Needs["SchematicSolver`"]
```

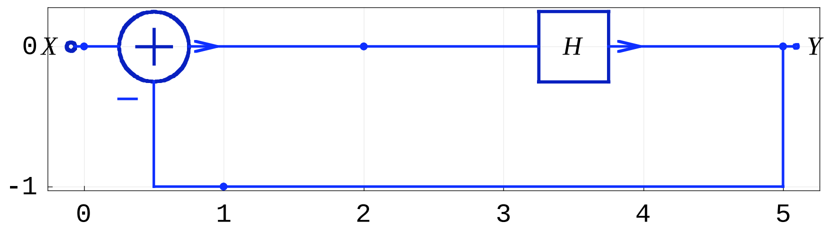
Here is the schematic specification of a system:

```
schematic = {{"Input", {0, 0}, X}, {"Output", {5, 0}, Y},
   {"Adder", {{0, 0}, {1, -1}, {2, 0}, {1, 1}}, {1, -1, 2, 0}},
   {"Block", {{2, 0}, {5, 0}}, H},
   {"Line", {{5, 0}, {5, -1}, {1, -1}}}};
```

```
ShowSchematic[schematic, ElementScale → 1]
```



```
ShowSchematic[schematic, ElementScale → 0.5]
```



## ElementSize

`ElementSize` is an option for element specification that controls the size and aspect ratio of a schematic element.

`ElementSize` → {*width*, *height*}

`ElementSize`→{*width*} defaults to `ElementSize`→{*width*, 1}

`ElementSize`→*width* defaults to `ElementSize`→{*width*, 1}

*width* is a number that represents the element width.

*height* is a number that represents the element height.

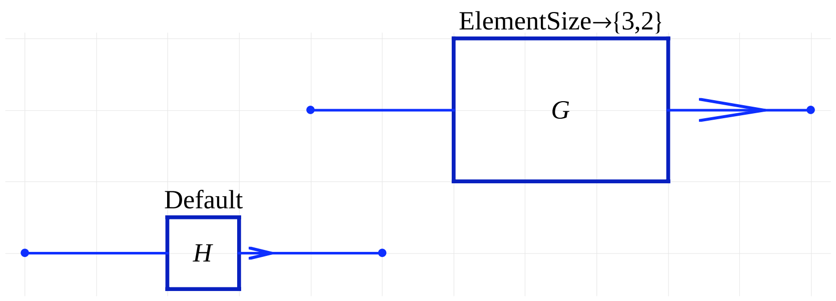
`ElementSize`→{1,1} is default.

You can reset the default using `SetOptions`[*function*, *option*→*value*].

For example, `SetOptions`[`DrawElement`, `ElementSize`→{2,1}].

### Examples

```
Needs["SchematicSolver`"]

{{"Block", {{0, 0}, {5, 0}}, H, "Default"},
  {"Block", {{4, 2}, {11, 2}}, G, "ElementSize→{3,2}",
   ElementSize → {3, 2}}};
ShowSchematic[%, Frame → False]
```



## PolylineDashing

`PolylineDashing` is an option for the Polyline-element specification that controls dashing.

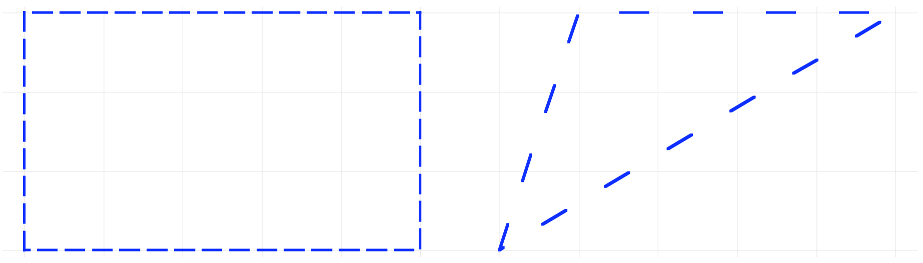
`PolylineDashing → Dashing[{`*markWidth*, *spaceWidth*`}]`

`PolylineDashing→Dashing[{0.02,0.01}]` is default.

See the *Mathematica* help for details about choosing the parameters of the `Dashing` function.

### Examples

```
Needs["SchematicSolver`"]

{{"Polyline", {{0, 0}, {0, 3}, {5, 3}, {5, 0}, {0, 0}},
   PolylineDashing → Dashing[{0.02, 0.01}]},
  {"Polyline", {{6, 0}, {7, 3}, {11, 3}, {6, 0}},
   PolylineDashing → Dashing[{0.03, 0.05}]}};
ShowSchematic[%, Frame → False]
```



## ShowArrowTail

ShowArrowTail is an option for the Arrow-element specification that controls the appearance of the arrow tail.

ShowArrowTail→True draws the arrow head and the arrow tail.

ShowArrowTail→False draws only the arrow head.

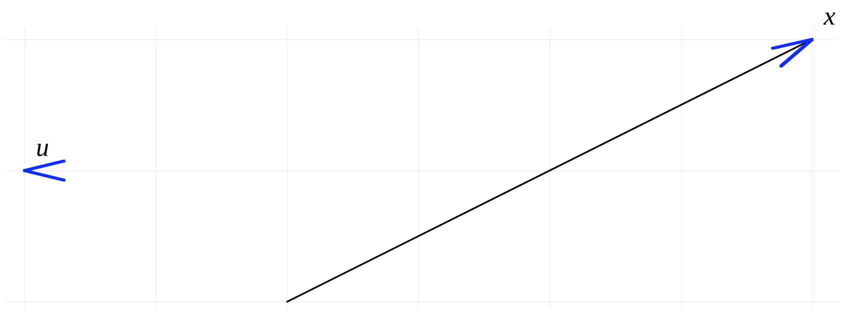
ShowArrowTail→True is default.

You can reset the default using SetOptions[*function*, *option*→*value*].

For example, SetOptions[DrawElement, ShowArrowTail→False].

## Examples

```
Needs["SchematicSolver`"]

{{"Arrow", {{8, 3}, {4, 1}}, x, ShowArrowTail → True},
  {"Arrow", {{2, 2}, {6, 2}}, u, ShowArrowTail → False}};
ShowSchematic[%, Frame → False]
```



## ShowNodes

ShowNodes is an option for element specification that controls the appearance of element nodes.

ShowNodes→True draws circles that represent nodes.

ShowNodes→False does not draw circles that represent nodes.

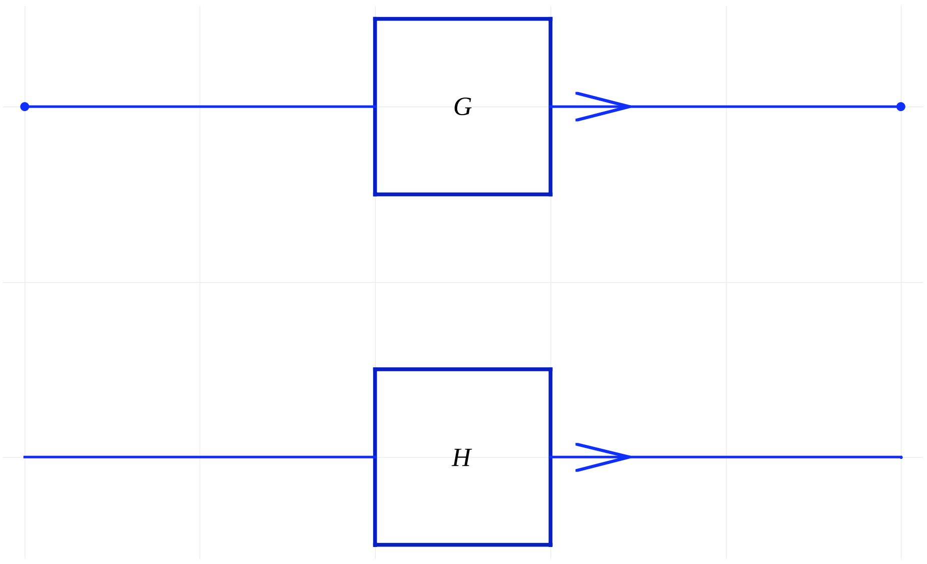
ShowNodes→True is default.

You can reset the default using SetOptions[*function*, *option*→*value*].

For example, SetOptions[DrawElement, ShowNodes→False].

## Examples

```
Needs["SchematicSolver`"]

{{"Block", {{0, 0}, {5, 0}}, H, "", ShowNodes → False},
  {"Block", {{0, 2}, {5, 2}}, G, "", ShowNodes → True}};
ShowSchematic[%, Frame → False]
```



## TextDirection

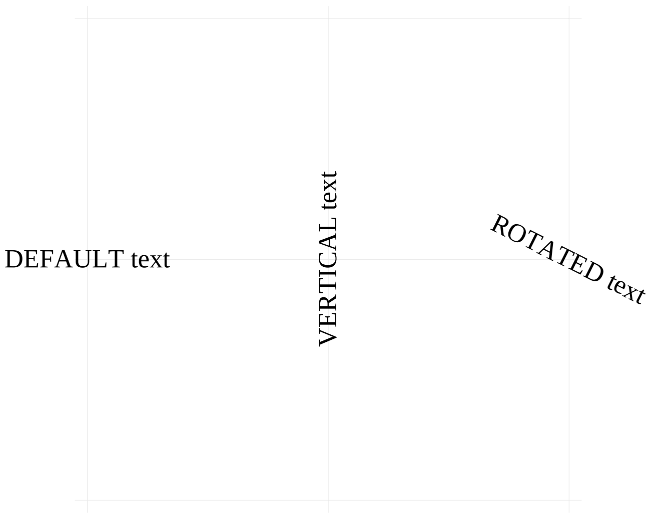
`TextDirection` is an option for the Text-element specification that controls the angle of the text rotation.

`TextDirection` → {*horizontalDirection*, *verticalDirection*}

`TextDirection`→{1,0} is default. See the *Mathematica* `Text` function for details about choosing the text direction.

### Examples

```
Needs["SchematicSolver`"]

{{"Text", {3, 0}, "DEFAULT text"},
  {"Text", {5, 0}, "ROTATED text",
   TextDirection → {2, -1}},
  {"Text", {4, 0}, "VERTICAL text",
   TextDirection → {0, 1}}};
ShowSchematic[%, Frame → False]
```



## TextOffset

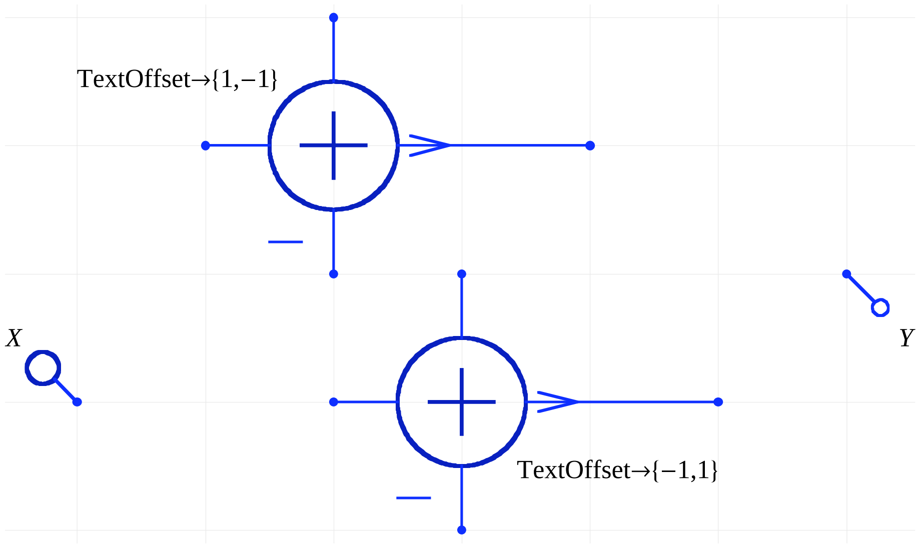
`TextOffset` is an option for element specification that controls the position of the element value and label.

`TextOffset` → {*horizontalOffset*, *verticalOffset*}

`TextOffset→Automatic` is default. See the *Mathematica* `Text` function for details about choosing the text offset. `TextOffset` also specifies the orientation of the Input element and the Output element.

### Examples

```
Needs["SchematicSolver`"]

{{"Input", {3, 0}, X, "", TextOffset → {1, -1}},
  {"Output", {9, 1}, Y, "", TextOffset → {-1, 1}},
  {"Adder", {{4, 2}, {5, 1}, {7, 2}, {5, 3}}, {1, -1, 2, 1}, "TextOffset→{1,-1}",
   TextOffset → {1, -1}},
  {"Adder", {{5, 0}, {6, -1}, {8, 0}, {6, 1}}, {1, -1, 2, 1}, "TextOffset→{-1,1}",
   TextOffset → {-1, 1}}};
ShowSchematic[%, Frame → False]
```



## $VersionSchematicSolverSchematicElements

`$VersionSchematicSolverSchematicElements` is a variable that contains information about the package version and release date.

```
Needs["SchematicSolver`"]

$VersionSchematicSolverSchematicElements

2.0 (February 03, 2004. 17:33)
```

## 14.4. Solving Continuous Systems

### `ContinuousSystemEquations`

`ContinuousSystemEquations` sets up the equations for a system represented by schematic specification.

---

{*systemEquations*, *systemVariables*} = `ContinuousSystemEquations`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`ContinuousSystemEquations`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `ContinuousSystemEquations`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`ContinuousSystemEquations`[*schematicSpec*, *signalTransformName*] defaults to `ContinuousSystemEquations`[*schematicSpec*, *signalTransformName*, `s`]

`ContinuousSystemEquations`[*schematicSpec*] defaults to `ContinuousSystemEquations`[*schematicSpec*, `Y`, `s`]

---

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.
Supported elements are Adder, Amplifier, Arrow, Block, Input, Integrator, Line, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex frequency (*s* for continuous-time systems).

*systemEquations* is a list of equations that describe the system.

*systemVariables* is a list of symbols that represent transforms of signals at nodes.

`s` is a reserved symbol for the complex frequency.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

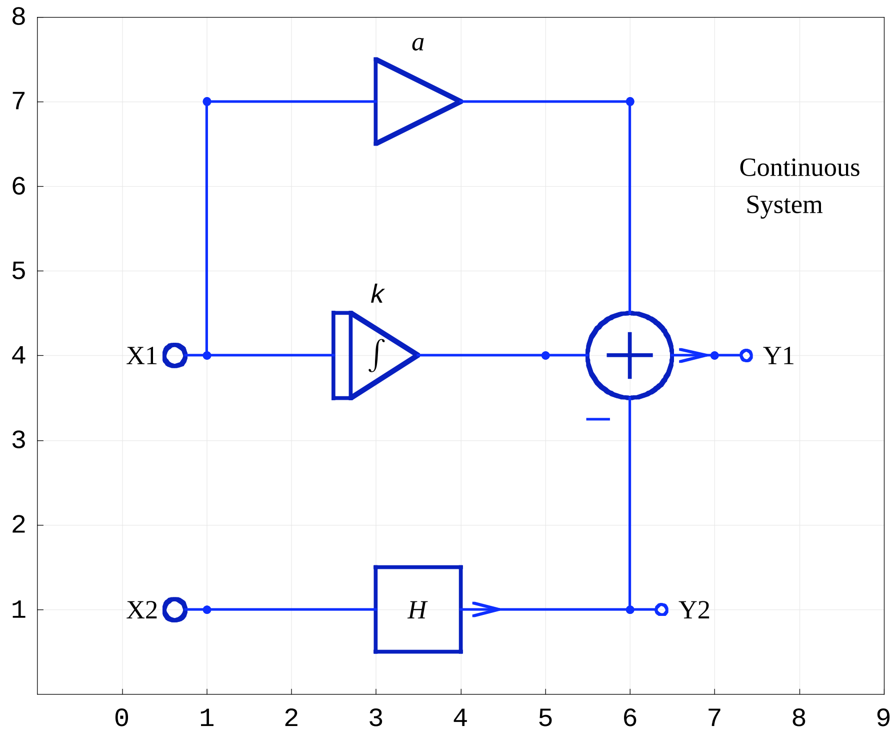
See also: `ContinuousSystemResponse`, `ContinuousSystemSignals`, `ContinuousSystemTransferFunction`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a continuous system:

```
continuousSchematic = {
  {"Input", {1, 4}, X1},
  {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1},
  {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Continuous\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Amplifier", {{1, 7}, {6, 7}}, a},
  {"Integrator", {{1, 4}, {5, 4}}, k},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
{eqns, vars} = ContinuousSystemEquations[continuousSchematic];
eqns // ColumnForm
```

```
Y[{1, 4}] == X1
Y[{1, 1}] == X2
Y[{7, 4}] == Y[{5, 4}] - Y[{6, 1}] + Y[{6, 7}]
Y[{6, 7}] == a Y[{1, 4}]
Y[{5, 4}] == k Y[{1,4}] / s
Y[{6, 1}] == H Y[{1, 1}]
```

## `ContinuousSystemResponse`

`ContinuousSystemResponse` finds the response of a system represented by schematic specification.

{*systemResponse*, *systemVariables*} = `ContinuousSystemResponse`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`ContinuousSystemResponse`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `ContinuousSystemResponse`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`ContinuousSystemResponse`[*schematicSpec*, *signalTransformName*] defaults to `ContinuousSystemResponse`[*schematicSpec*, *signalTransformName*, `s`]

`ContinuousSystemResponse`[*schematicSpec*] defaults to `ContinuousSystemResponse`[*schematicSpec*, `Y`, `s`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

Supported elements are Adder, Amplifier, Arrow, Block, Input, Integrator, Line, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex frequency (*s* for continuous-time systems).

*systemResponse* is a list of replacement rules that describe the system response.

*systemVariables* is a list of symbols that represent transforms of signals at nodes.

`s` is a reserved symbol for the complex frequency.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

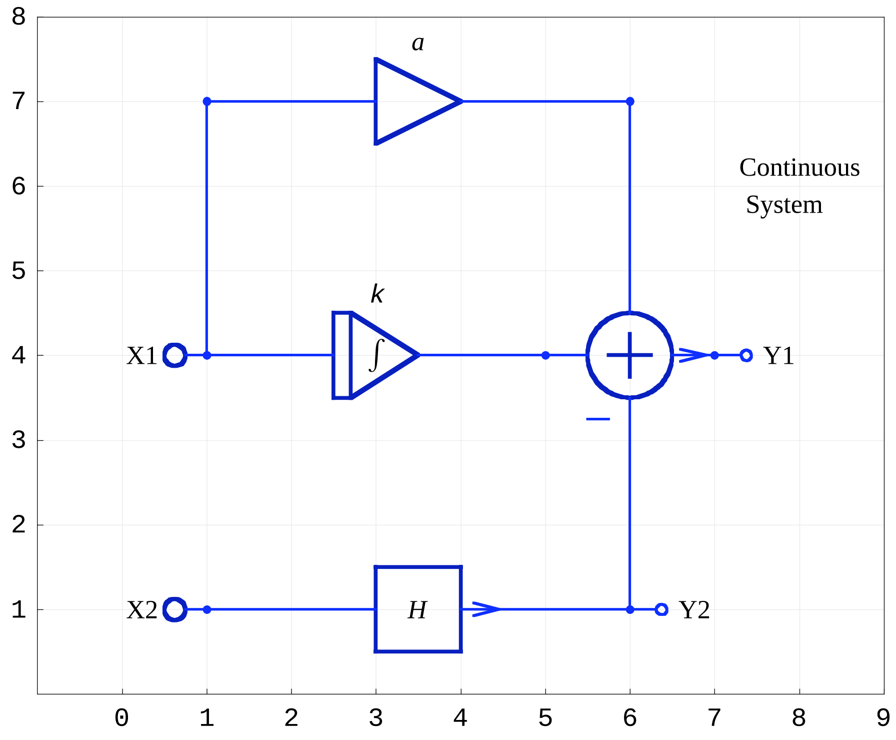
See also: `ContinuousSystemEquations`, `ContinuousSystemSignals`, `ContinuousSystemTrans-ferFunction`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a continuous system:

```
continuousSchematic = {
  {"Input", {1, 4}, X1},
  {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1},
  {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Continuous\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Amplifier", {{1, 7}, {6, 7}}, a},
  {"Integrator", {{1, 4}, {5, 4}}, k},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
{resp, vars} = ContinuousSystemResponse[continuousSchematic];
resp // ColumnForm
```

Y[{7, 4}] → $-\frac{-k\ X1 - a\ s\ X1 + H\ s\ X2}{s}$

Y[{5, 4}] → $\frac{k\ X1}{s}$

Y[{6, 7}] → a X1

Y[{6, 1}] → H X2

Y[{1, 1}] → X2

Y[{1, 4}] → X1

## ContinuousSystemSignals

`ContinuousSystemSignals` finds the signals at all nodes of a system represented by schematic specification.

{*systemSignals*, *systemVariables*} = `ContinuousSystemSignals`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`ContinuousSystemSignals`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `ContinuousSystemSignals`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`ContinuousSystemSignals`[*schematicSpec*, *signalTransformName*] defaults to `ContinuousSystemSignals`[*schematicSpec*, *signalTransformName*, `s`]

`ContinuousSystemSignals`[*schematicSpec*] defaults to `ContinuousSystemSignals`[*schematicSpec*, `Y`, `s`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications. Supported elements are Adder, Amplifier, Arrow, Block, Input, Integrator, Line, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex frequency (*s* for continuous-time systems).

*systemSignals* is a list of expressions that represent the signals at all nodes of the system.

*systemVariables* is a list of symbols that represent transforms of signals at nodes.

`s` is a reserved symbol for the complex frequency.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

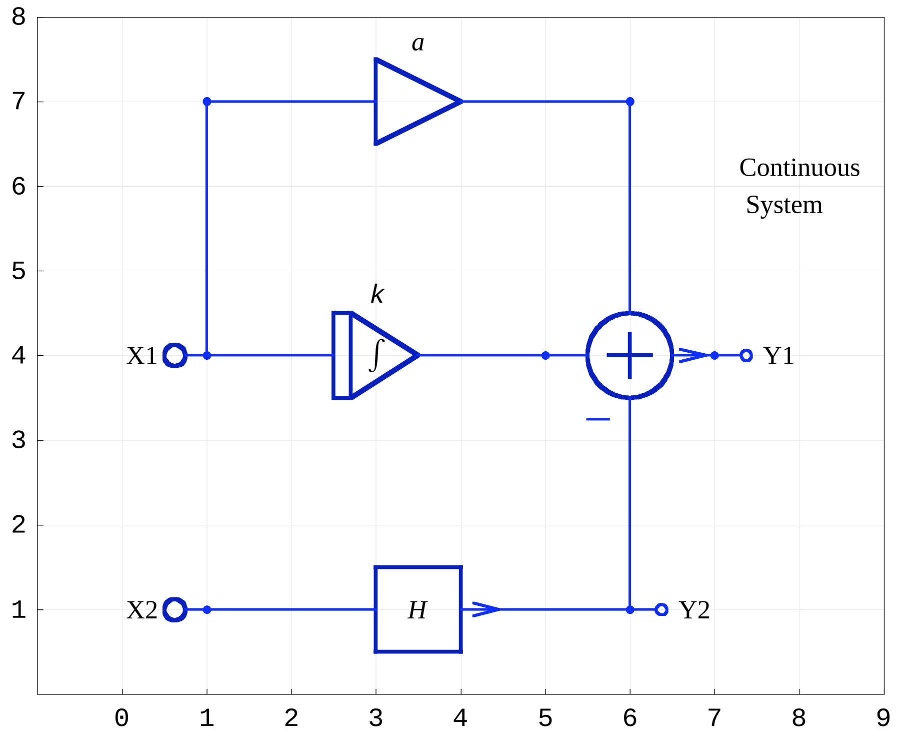
See also: `ContinuousSystemEquations`, `ContinuousSystemResponse`, `ContinuousSystemTrans-ferFunction`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a continuous system:

```
continuousSchematic = {
  {"Input", {1, 4}, X1},
  {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1},
  {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Continuous\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Amplifier", {{1, 7}, {6, 7}}, a},
  {"Integrator", {{1, 4}, {5, 4}}, k},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
ContinuousSystemSignals[continuousSchematic];
% // Transpose // TableForm
```

```
   -k X1-a s X1+H s X2
 - ───────────────────     Y[{7, 4}]
           s
 a X1                      Y[{6, 7}]
 H X2                      Y[{6, 1}]
 k X1
 ────                      Y[{5, 4}]
  s
 X1                        Y[{1, 4}]
 X2                        Y[{1, 1}]
```

## ContinuousSystemTransferFunction

`ContinuousSystemTransferFunction` finds the transfer function matrix of a system represented by schematic specification.

{*transferFunctionMatrix*, *systemInputs*, *systemOutputs*} = `ContinuousSystemTransferFunction`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`ContinuousSystemTransferFunction`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `ContinuousSystemTransferFunction`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`ContinuousSystemTransferFunction`[*schematicSpec*, *signalTransformName*] defaults to `ContinuousSystemTransferFunction`[*schematicSpec*, *signalTransformName*, `s`]

`ContinuousSystemTransferFunction`[*schematicSpec*] defaults to `ContinuousSystemTransferFunction`[*schematicSpec*, `Y`, `s`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

Supported elements are Adder, Amplifier, Arrow, Block, Input, Integrator, Line, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex frequency (*s* for continuous-time systems).

*transferFunctionMatrix* is the transfer function matrix of the system. Each row of this matrix corresponds to a system output, and each column of the matrix corresponds to a system input. The first input corresponds to the first Input element in *schematicSpec*, the second input corresponds to the second Input element in *schematicSpec*, and so on. The same convention applies to the numbering of outputs.

*systemInputs* is a list of symbols that represent system inputs.

*systemOutputs* is a list of symbols that represent system outputs.

`s` is a reserved symbol for complex frequency.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

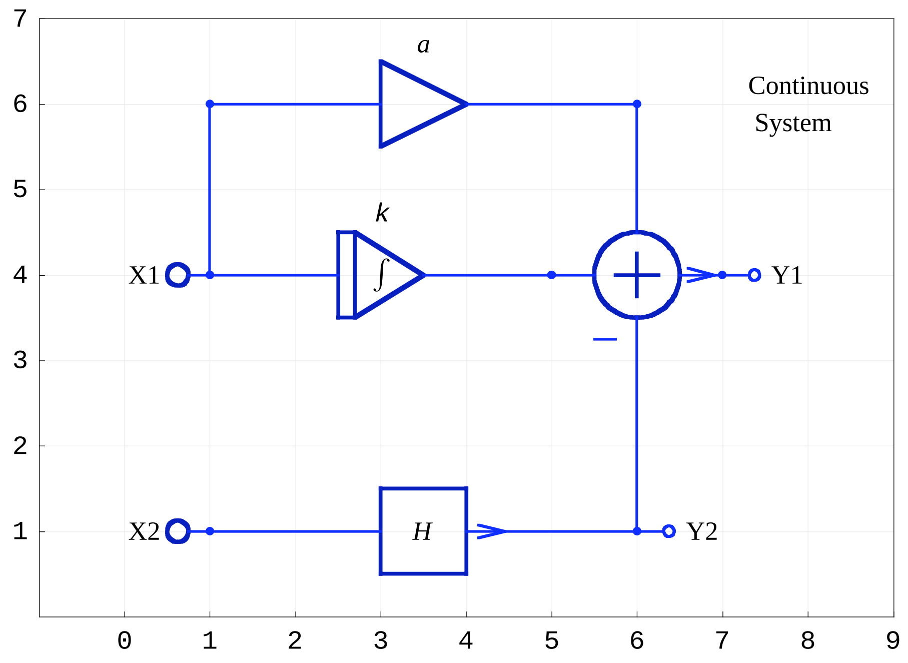
See also: `ContinuousSystemEquations`, `ContinuousSystemResponse`, `ContinuousSystemSignals`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a continuous system.

```
continuousSchematic = {
  {"Input", {1, 4}, X1}, {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1}, {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Continuous\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 6}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 6}}}, {"Block", {{1, 1}, {6, 1}}, H},
  {"Amplifier", {{1, 6}, {6, 6}}, a},
  {"Integrator", {{1, 4}, {5, 4}}, k}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 7}}]
```



```
{tfMatrix, systemInp, systemOut} =
 ContinuousSystemTransferFunction[continuousSchematic];
```

```
tfMatrix // Together // MatrixForm
```

$$\begin{pmatrix} \frac{k+a\,s}{s} & -H \\ 0 & H \end{pmatrix}$$

```
systemInp // ColumnForm
```

```
Y[{1, 4}]
Y[{1, 1}]
```

```
systemOut // ColumnForm
```

```
Y[{7, 4}]
Y[{6, 1}]
```

## `s`

`s` is a reserved symbol in *SchematicSolver*.

---

`s` is a symbol that represents the Laplace complex variable.

---

See also: `ContinuousSystemEquations`, `ContinuousSystemResponse`, `ContinuousSystemSignals`, `ContinuousSystemTransferFunction`

## 14.5. Solving Discrete Systems

### `DiscreteSystemEquations`

`DiscreteSystemEquations` sets up the equations for a system represented by schematic specification.

---

{*systemEquations*, *systemVariables*} = `DiscreteSystemEquations`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`DiscreteSystemEquations`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `DiscreteSystemEquations`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`DiscreteSystemEquations`[*schematicSpec*, *signalTransformName*] defaults to `DiscreteSystemEquations`[*schematicSpec*, *signalTransformName*, `z`]

`DiscreteSystemEquations`[*schematicSpec*] defaults to `DiscreteSystemEquations`[*schematicSpec*, `Y`, `z`]

---

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications. Supported elements are Adder, Arrow, Block, Delay, Input, Line, Multiplier, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex variable (*z* for discrete systems).

*systemEquations* is a list of equations that describe the system.

*systemVariables* is a list of symbols that represent transform of signals at nodes.

`z` is a reserved symbol for the complex variable.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

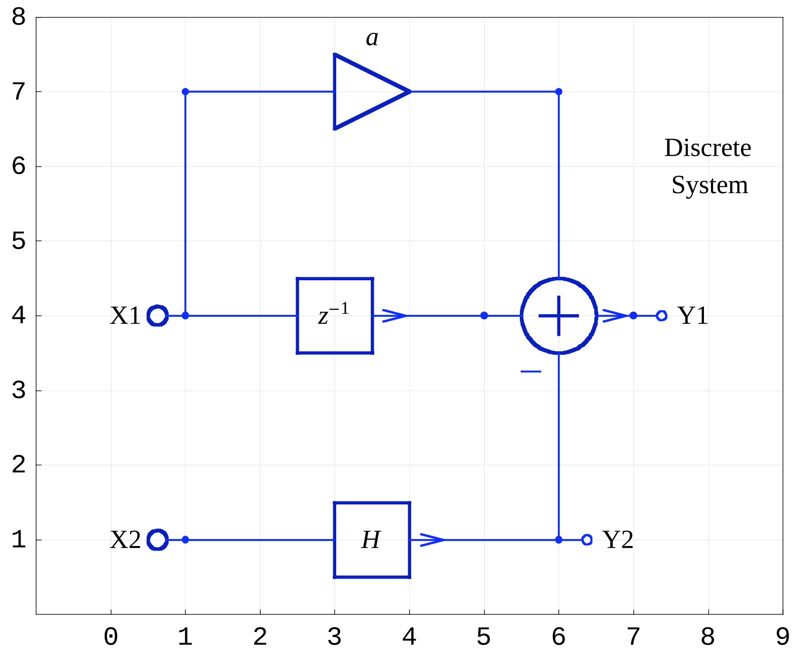
See also: `DiscreteSystemResponse`, `DiscreteSystemSignals`, `DiscreteSystemTransferFunc-tion`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a discrete system:

```
discreteSchematic = {
  {"Input", {1, 4}, X1},
  {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1},
  {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Discrete\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Multiplier", {{1, 7}, {6, 7}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
{eqns, vars} = DiscreteSystemEquations[discreteSchematic];
eqns // ColumnForm
```

```
Y[{1, 4}] == X1
Y[{1, 1}] == X2
Y[{7, 4}] == Y[{5, 4}] - Y[{6, 1}] + Y[{6, 7}]
Y[{6, 7}] == a Y[{1, 4}]
Y[{5, 4}] == Y[{1,4}]/z
Y[{6, 1}] == H Y[{1, 1}]
```

## DiscreteSystemResponse

DiscreteSystemResponse finds the response of a system represented by schematic specification.

---

{*systemResponse*, *systemVariables*} = DiscreteSystemResponse[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

DiscreteSystemResponse[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to DiscreteSystemResponse[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, PrintFloatingPorts→False, Verbose→False]

DiscreteSystemResponse[*schematicSpec*, *signalTransformName*] defaults to DiscreteSystemResponse[*schematicSpec*, *signalTransformName*, z]

DiscreteSystemResponse[*schematicSpec*] defaults to DiscreteSystemResponse[*schematicSpec*, Y, z]

---

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications. Supported elements are Adder, Arrow, Block, Delay, Input, Line, Multiplier, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex variable (*z* for discrete systems).

*systemResponse* is a list of replacement rules that describe the system response.

*systemVariables* is a list of symbols that represent transform of signals at nodes.

z is a reserved symbol for the complex variable.

Y is a reserved symbol that represents the transform of signals at nodes of the system.

PrintFloatingPorts→True prints a list of element inputs that are not connected to outputs of other elements.

Verbose→True prints solving details.

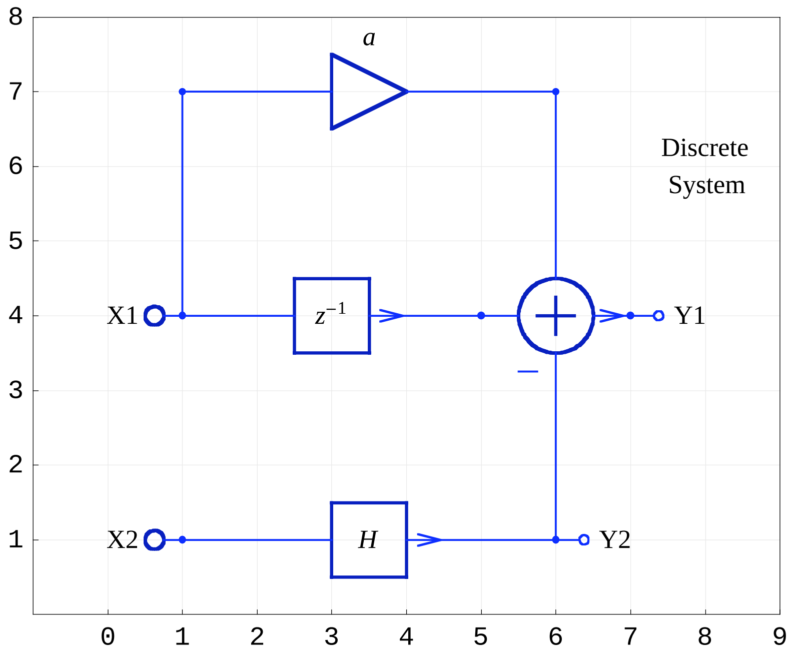
See also: DiscreteSystemEquations, DiscreteSystemSignals, DiscreteSystemTransferFunction

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a discrete system:

```
discreteSchematic = {
  {"Input", {1, 4}, X1},
  {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1},
  {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Discrete\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Multiplier", {{1, 7}, {6, 7}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
{resp, vars} = DiscreteSystemResponse[discreteSchematic];
resp // ColumnForm
```

Y[{7, 4}] → $-\frac{-X1 - a\ X1\ z + H\ X2\ z}{z}$

Y[{5, 4}] → $\frac{X1}{z}$

Y[{6, 7}] → a X1

Y[{6, 1}] → H X2

Y[{1, 1}] → X2

Y[{1, 4}] → X1

## DiscreteSystemSignals

`DiscreteSystemSignals` finds the signals at all nodes of a system represented by schematic specification.

{*systemSignals*, *systemVariables*} = `DiscreteSystemSignals`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`DiscreteSystemSignals`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `DiscreteSystemSignals`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`DiscreteSystemSignals`[*schematicSpec*, *signalTransformName*] defaults to `DiscreteSystemSignals`[*schematicSpec*, *signalTransformName*, `z`]

`DiscreteSystemSignals`[*schematicSpec*] defaults to `DiscreteSystemSignals`[*schematicSpec*, `Y`, `z`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications. Supported elements are Adder, Arrow, Block, Delay, Input, Line, Multiplier, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex variable (*z* for discrete systems).

*systemSignals* is a list of expressions that represent the signals at all nodes of the system.

*systemVariables* is a list of symbols that represent transform of signals at nodes.

`z` is a reserved symbol for the complex variable.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

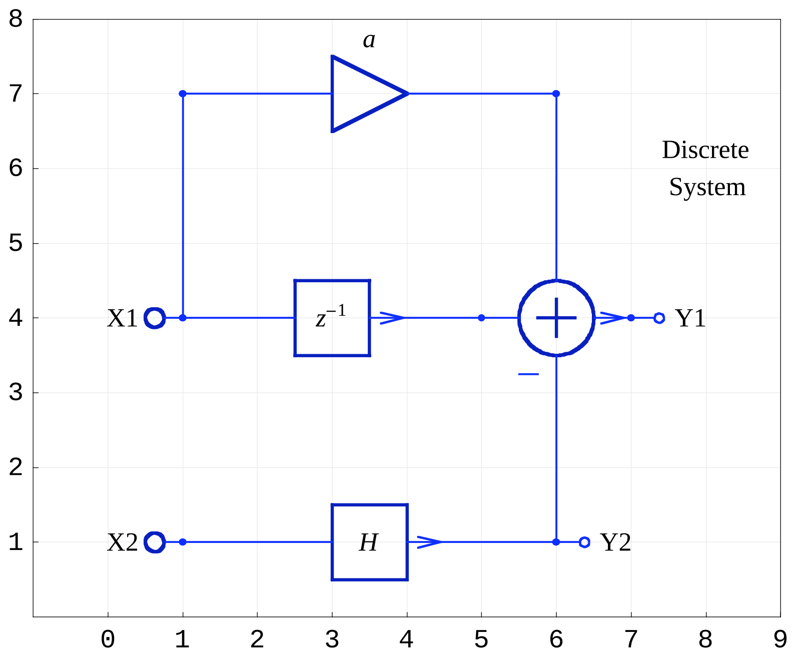
See also: `DiscreteSystemEquations`, `DiscreteSystemResponse`, `DiscreteSystemTransfer-Function`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a discrete system:

```
discreteSchematic = {
  {"Input", {1, 4}, X1},
  {"Input", {1, 1}, X2},
  {"Output", {7, 4}, Y1},
  {"Output", {6, 1}, Y2},
  {"Text", {8, 6}, "Discrete\n System"},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Multiplier", {{1, 7}, {6, 7}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
DiscreteSystemSignals[discreteSchematic];
% // Transpose // TableForm
```

| | |
|---|---|
| $-\frac{-X1 - a\,X1\,z + H\,X2\,z}{z}$ | Y[{7, 4}] |
| a X1 | Y[{6, 7}] |
| H X2 | Y[{6, 1}] |
| $\frac{X1}{z}$ | Y[{5, 4}] |
| X1 | Y[{1, 4}] |
| X2 | Y[{1, 1}] |

## DiscreteSystemTransferFunction

`DiscreteSystemTransferFunction` finds the transfer function matrix of a system represented by schematic specification.

{*transferFunctionMatrix*, *systemInputs*, *systemOutputs*} = `DiscreteSystemTransferFunction`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, *options*]

`DiscreteSystemTransferFunction`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*] defaults to `DiscreteSystemTransferFunction`[*schematicSpec*, *signalTransformName*, *frequencyVariableName*, `PrintFloatingPorts→False, Verbose→False`]

`DiscreteSystemTransferFunction`[*schematicSpec*, *signalTransformName*] defaults to `DiscreteSystemTransferFunction`[*schematicSpec*, *signalTransformName*, `z`]

`DiscreteSystemTransferFunction`[*schematicSpec*] defaults to `DiscreteSystemTransferFunction`[*schematicSpec*, `Y`, `z`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

Supported elements are Adder, Arrow, Block, Delay, Input, Line, Multiplier, Node, Output, Polyline, and Text.

*signalTransformName* is a symbol that represents the transform of signals at nodes of the system.

*frequencyVariableName* is a symbol that represents the complex variable (*z* for discrete systems).

*transferFunctionMatrix* is the transfer function matrix of the system. Each row of this matrix corresponds to a system output, and each column of the matrix corresponds to a system input. The first input corresponds to the first Input element in *schematicSpec*, the second input corresponds to the second Input element in *schematicSpec*, and so on. The same convention applies to the numbering of outputs.

*systemInputs* is a list of symbols that represent system inputs.

*systemOutputs* is a list of symbols that represent system outputs.

`z` is a reserved symbol for the complex variable.

`Y` is a reserved symbol that represents the transform of signals at nodes of the system.

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`Verbose→True` prints solving details.

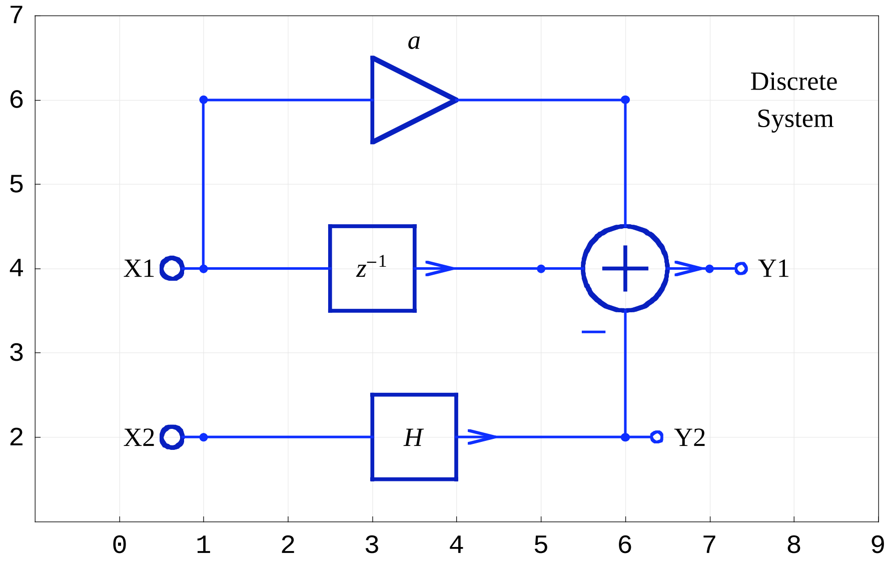
See also: `DiscreteSystemEquations`, `DiscreteSystemResponse`, `DiscreteSystemSignals`

### Example

```
Needs["SchematicSolver`"]
```

Here is the schematic specification of a discrete system:

```
discreteSchematic = {{"Text", {8, 6}, "Discrete\n System"},
  {"Input", {1, 4}, X1}, {"Input", {1, 2}, X2},
  {"Output", {7, 4}, Y1}, {"Output", {6, 2}, Y2},
  {"Adder", {{5, 4}, {6, 2}, {7, 4}, {6, 6}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 6}}}, {"Block", {{1, 2}, {6, 2}}, H},
  {"Multiplier", {{1, 6}, {6, 6}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1}};
ShowSchematic[%, PlotRange → {{-1, 9}, {1, 7}}]
```



```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[discreteSchematic];
```

```
tfMatrix // Together // MatrixForm
```

$$\begin{pmatrix} \frac{1+a\,z}{z} & -H \\ 0 & H \end{pmatrix}$$

```
systemInp // ColumnForm
```

```
Y[{1, 4}]
Y[{1, 2}]
```

```
systemOut // ColumnForm
```

```
Y[{7, 4}]
Y[{6, 2}]
```

## `f`

`f` is a reserved symbol in *SchematicSolver*.

`f` is a symbol that represents the digital frequency.

See also: `DiscreteSystemFrequencyResponse`, `DiscreteSystemMagnitudeResponsePlot`, `SequenceFourierTransform`, `SequenceFourierTransformMagnitudePlot`, `UnitSineSequence`

## PrintFloatingPorts

`PrintFloatingPorts` is an option for the solving functions.

---

`PrintFloatingPorts→True` prints a list of element inputs that are not connected to outputs of other elements.

`PrintFloatingPorts→False` does not report on the unconnected element inputs.

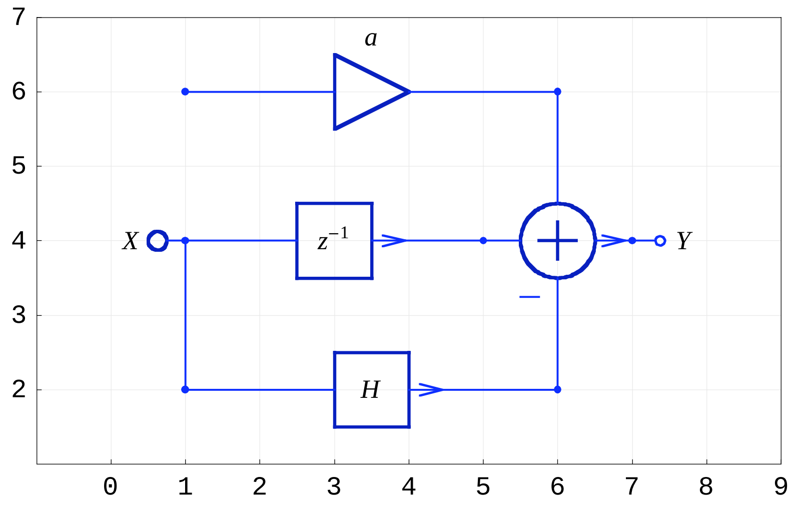
---

`PrintFloatingPorts→False` is default.

### Example

Here is the schematic specification of a discrete system:

```
discreteSchematic = {
  {"Input", {1, 4}, X}, {"Output", {7, 4}, Y},
  {"Adder", {{5, 4}, {6, 2}, {7, 4}, {6, 6}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 2}}}, {"Block", {{1, 2}, {6, 2}}, H},
  {"Multiplier", {{1, 6}, {6, 6}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1}};
ShowSchematic[%, PlotRange → {{-1, 9}, {1, 7}}]
```



```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[discreteSchematic,
  PrintFloatingPorts → True];
```

```
   Floating ports = {Y[{1, 6}]}
```

## Verbose

`Verbose` is an option for the solving functions.

`Verbose→True` prints solving details.

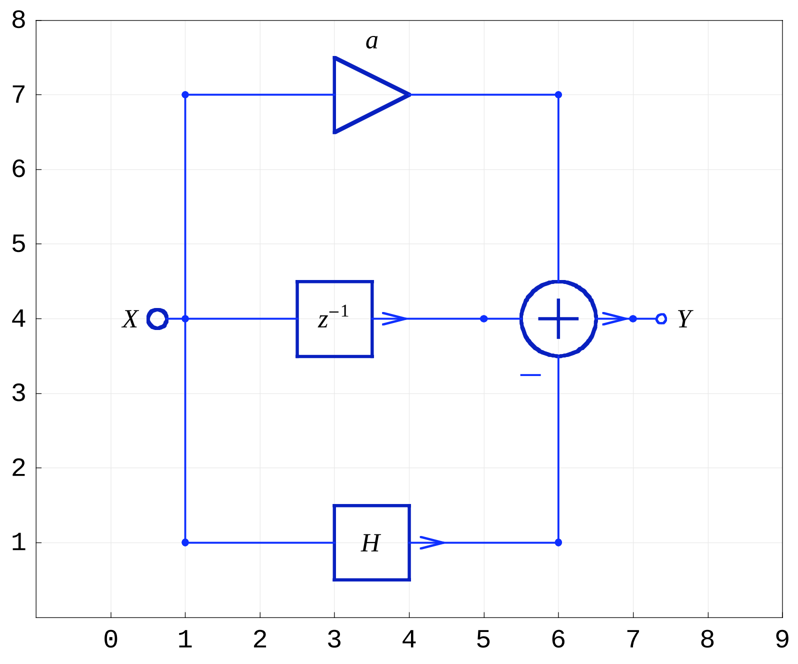
`Verbose→False` does not print solving details.

`Verbose→False` is default.

## Example

Here is the schematic specification of a discrete system:

```
discreteSchematic = {
  {"Input", {1, 4}, X}, {"Output", {7, 4}, Y},
  {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}},
  {"Line", {{1, 4}, {1, 1}}},
  {"Line", {{1, 4}, {1, 7}}},
  {"Multiplier", {{1, 7}, {6, 7}}, a},
  {"Delay", {{1, 4}, {5, 4}}, 1},
  {"Block", {{1, 1}, {6, 1}}, H}};
ShowSchematic[%, PlotRange → {{-1, 9}, {0, 8}}]
```



```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[discreteSchematic,
  Verbose → True];
```

```
SolveDiscreteSystem 2004 Feb 03, 19:18

Signals at nodes are designated by the symbol Y

Complex variable is represented by the symbol z

Schematic specification = {{Input, {1, 4}, X}, {Output, {7, 4}, Y},
  {Adder, {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}}, {Line, {{1, 4}, {1, 1}}},
  {Line, {{1, 4}, {1, 7}}}, {Multiplier, {{1, 7}, {6, 7}}, a},
  {Delay, {{1, 4}, {5, 4}}, 1}, {Block, {{1, 1}, {6, 1}}, H}}

Topology element list (raw) = {{Input, {1, 4}, {}, X}, {Output, {7, 4}, {}, Y},
  {Adder, {7, 4}, {{5, 4}, {6, 1}, {6, 7}}, {1, -1, 1}}, {Line, {1, 1}, {1, 4}, 1},
  {Line, {1, 7}, {1, 4}, 1}, {Multiplier, {6, 7}, {1, 7}, a},
  {Delay, {5, 4}, {1, 4}, 1}, {Block, {6, 1}, {1, 1}, H}}

Topology element list = {{Input, {1, 4}, {}, X}, {Output, {7, 4}, {}, Y},
  {Adder, {7, 4}, {{5, 4}, {6, 1}, {6, 7}}, {1, -1, 1}}, {Line, {1, 4}, {1, 4}, 1},
  {Line, {1, 4}, {1, 4}, 1}, {Multiplier, {6, 7}, {1, 4}, a},
  {Delay, {5, 4}, {1, 4}, 1}, {Block, {6, 1}, {1, 4}, H}}

Line-element position = {4, 5}

Input elements = {{Input, {1, 4}, {}, X}}

Input-element coordinates = {{1, 4}}

Input-element vector = {Y[{1, 4}]}

Output elements = {{Output, {7, 4}, {}, Y}}

Output-element coordinates = {{7, 4}}

Output-element vector = {Y[{7, 4}]}

Analysis elements =
 {{Input, {1, 4}, {}, X}, {Adder, {7, 4}, {{5, 4}, {6, 1}, {6, 7}}, {1, -1, 1}},
  {Multiplier, {6, 7}, {1, 4}, a},
  {Delay, {5, 4}, {1, 4}, 1}, {Block, {6, 1}, {1, 4}, H}}

Node coordinates (raw) = {{{1, 4}, {}}, {{7, 4}, {5, 4}, {6, 1}, {6, 7}},
  {{6, 7}, {1, 4}}, {{5, 4}, {1, 4}}, {{6, 1}, {1, 4}}}

Node coordinates = {{1, 4}, {7, 4}, {5, 4},
  {6, 1}, {6, 7}, {6, 7}, {1, 4}, {5, 4}, {1, 4}, {6, 1}, {1, 4}}

Signal coordinates = {{1, 4}, {5, 4}, {6, 1}, {6, 7}, {7, 4}}

Signal vector = {Y[{1, 4}], Y[{5, 4}], Y[{6, 1}], Y[{6, 7}], Y[{7, 4}]}

Output coordinates (raw) = {{1, 4}, {7, 4}, {6, 7}, {5, 4}, {6, 1}}

Output coordinates = {{1, 4}, {5, 4}, {6, 1}, {6, 7}, {7, 4}}

Output Vector = {Y[{1, 4}], Y[{5, 4}], Y[{6, 1}], Y[{6, 7}], Y[{7, 4}]}

Duplicate-output coordinates = {1, 1, 1, 1, 1}

Duplicate-output flag = {False, False, False, False, False}

System variables = {Y[{1, 4}], Y[{5, 4}], Y[{6, 1}], Y[{6, 7}], Y[{7, 4}]}
```

System equations = 
$$\begin{aligned} &Y[\{7, 4\}] == Y[\{5, 4\}] - Y[\{6, 1\}] + Y[\{6, 7\}] \\ &Y[\{6, 7\}] == a\,Y[\{1, 4\}] \\ &Y[\{5, 4\}] == \frac{Y[\{1, 4\}]}{z} \\ &Y[\{6, 1\}] == H\,Y[\{1, 4\}] \end{aligned}$$

```
No floating ports.
```

## Y

Y is a reserved symbol in *SchematicSolver*.

Y is a symbol that represents the transform of signals at nodes of a system.

See also: `DiscreteSystemEquations`, `DiscreteSystemResponse`, `DiscreteSystemSignals`, `DiscreteSystemTransferFunction`, `ContinuousSystemEquations`, `ContinuousSystemResponse`, `ContinuousSystemSignals`, `ContinuousSystemTransferFunction`

## `z`

`z` is a reserved symbol in *SchematicSolver*.

`z` is a symbol that represents the z-transform variable.

See also: `DiscreteSystemEquations`, `DiscreteSystemResponse`, `DiscreteSystemSignals`, `DiscreteSystemTransferFunction`, `DiscreteSystemDisplayForm`, `DiscreteSystemFrequencyResponse`, `DiscreteSystemMagnitudeResponsePlot`, `DiscreteSystemProcessingSISO`

## $VersionSchematicSolverSchematicAnalysis

`$VersionSchematicSolverSchematicAnalysis` is a variable that contains information about the package version and release date.

```
Needs["SchematicSolver`"]

$VersionSchematicSolverSchematicAnalysis

2.0 (February 03, 2004. 19:18)
```

## 14.6. Utilities

### `CheckElementSyntax`

`CheckElementSyntax` checks the syntax of element specification.

*flag* = `CheckElementSyntax`[*elementSpec*]

*elementSpec* is an element specification.

*flag* is `False` if *elementSpec* is not a list specifying a schematic element. Otherwise, it is `True`.

See also: `CheckSchematicSyntax`, `DrawElement`, `ShowSchematic`

### Example

```
Needs["SchematicSolver`"]
```

Here is the element specification:

```
elementSpec = {"Adder", {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}}
```

```
{Adder, {{5, 4}, {6, 1}, {7, 4}, {6, 7}}, {1, -1, 2, 1}}
```

```
CheckElementSyntax[elementSpec]
```

```
True
```

## `CheckSchematicSyntax`

`CheckSchematicSyntax` checks the syntax of schematic specification.

*flag* = `CheckSchematicSyntax`[*schematicSpec*]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

*flag* is `False` if *schematicSpec* is not a list specifying a schematic. Otherwise, it is `True`.

See also: `CheckElementSyntax`, `DrawElement`, `ShowSchematic`

## dBMagnitudePlot

dBMagnitudePlot is an option for DiscreteSystemMagnitudeResponsePlot, SequenceDiscreteFourierTransformMagnitudePlot, and SequenceFourierTransformMagnitudePlot.

| | |
|---|---|
| dBMagnitudePlot→True | plots magnitude in decibels: $20\log_{10}(x)$. |
| dBMagnitudePlot→False | plots magnitude in linear scale. |

dBMagnitudePlot→True is default for DiscreteSystemMagnitudeResponsePlot.

dBMagnitudePlot→False is default for SequenceDiscreteFourierTransformMagnitudePlot and SequenceFourierTransformMagnitudePlot.

You can reset the default using

SetOptions[*function*, *option*→*value*]. For example,

SetOptions[SequenceFourierTransformMagnitudePlot, dBMagnitudePlot→True].

## `DiscreteSystemDisplayForm`

`DiscreteSystemDisplayForm` displays transfer function of a discrete system in the traditional form.

`DiscreteSystemDisplayForm`[*H*, *complexVariableName*]

`DiscreteSystemDisplayForm`[*H*] defaults to `DiscreteSystemDisplayForm`[*H*, `z`]

*H* is the transfer function.

*complexVariableName* is a symbol that represents the complex variable.

`z` is a reserved symbol for the complex variable.

`DiscreteSystemDisplayForm`[*H*, *z*] displays the transfer function *H* of a discrete system in terms of $z^{-1}$.

See also: `DiscreteSystemTransferFunction`

### Example

Here is the transfer function of a discrete system:

```
H = (a + 0.8 + z) / (z + a * z + 0.9);
DiscreteSystemDisplayForm[H]
```

$$\frac{1 + (0.8 + a)\ z^{-1}}{1 + a + 0.9\ z^{-1}}$$

## DiscreteSystemFrequencyResponse

`DiscreteSystemFrequencyResponse` computes and plots the magnitude and the phase characteristics of a discrete system.

---

{*magnitudeResponse*, *phaseResponse*} = `DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*, *digitalFrequencyName*, {*f1*, *f2*}]

`DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*, *digitalFrequencyName*] defaults to `DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*, *digitalFrequencyName*, {0, 1/2}]

`DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*] defaults to `DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*, `f`, {0, 1/2}]

`DiscreteSystemFrequencyResponse`[*H*] defaults to `DiscreteSystemFrequencyResponse`[*H*, `z`, `f`, {0, 1/2}]

`DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*, {*f1*, *f2*}] defaults to `DiscreteSystemFrequencyResponse`[*H*, *complexVariableName*, `f`, {*f1*, *f2*}]

`DiscreteSystemFrequencyResponse`[*H*, {*f1*, *f2*}] defaults to `DiscreteSystemFrequencyResponse`[*H*, `z`, `f`, {*f1*, *f2*}]

---

*H* is the transfer function.

*complexVariableName* is a symbol that represents the complex variable.

*digitalFrequencyName* is a symbol that represents the digital frequency.

{*f1*, *f2*} is the plot range of digital frequencies.

*magnitudeResponse* is an expression that is the magnitude response in terms of *digitalFrequencyName*.

*phaseResponse* is an expression that is the phase response in terms of *digitalFrequencyName*.

`z` is a reserved symbol for the complex variable.

`f` is a reserved symbol for the digital frequency.

See also: `DiscreteSystemMagnitudeResponsePlot`

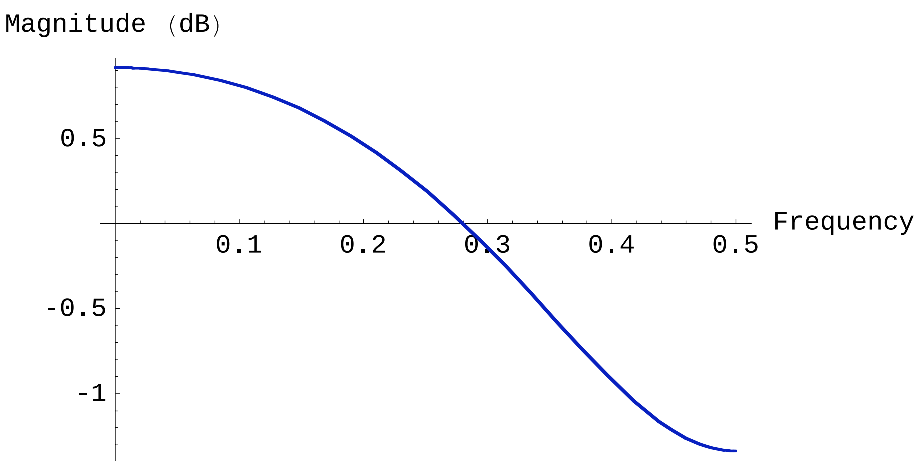
### Example

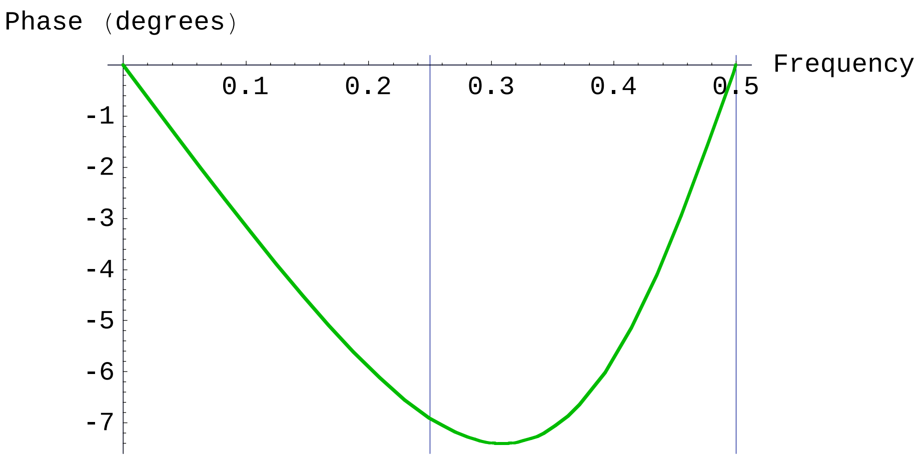
Here is the transfer function of a discrete system:

```
H = (1 / 4 + z) / (z + 1 / 8)
```

$$\frac{\frac{1}{4}+z}{\frac{1}{8}+z}$$

```
{magnitude, phase} = DiscreteSystemFrequencyResponse[H];
```





**magnitude // TraditionalForm**

$$2\left|\frac{1+4\,e^{2 i f \pi}}{1+8\,e^{2 i f \pi}}\right|$$

**phase // TraditionalForm**

$$\operatorname{Arg}(1.\,(\cos(2\,f\,\pi)+i\sin(2\,f\,\pi))+0.25)-\operatorname{Arg}(1.\,(\cos(2\,f\,\pi)+i\sin(2\,f\,\pi))+0.125)$$

## DiscreteSystemMagnitudeResponsePlot

`DiscreteSystemMagnitudeResponsePlot` plots the magnitude characteristics of a discrete system.

`DiscreteSystemMagnitudeResponsePlot`[*H*, *complexVariable*, {*f*1,*f*2}, *opts*]

`DiscreteSystemMagnitudeResponsePlot`[*H*, *complexVariable*, {*f*1,*f*2}] defaults to `DiscreteSystemMagnitudeResponsePlot`[*H*, *complexVariable*, {*f*1,*f*2}, `dBMagnitudePlot→True`, *optsPlot*]

`DiscreteSystemMagnitudeResponsePlot`[*H*, *complexVariable*] defaults to `DiscreteSystemMagnitudeResponsePlot`[*H*, *complexVariable*, {0,1/2}]

`DiscreteSystemMagnitudeResponsePlot`[*H*] defaults to `DiscreteSystemMagnitudeResponsePlot`[*H*, `z`, {0,1/2}]

`DiscreteSystemMagnitudeResponsePlot`[*H*, {*f*1,*f*2}] defaults to `DiscreteSystemMagnitudeResponsePlot`[*H*, `z`, {*f*1,*f*2}]

*H* is the transfer function of a discrete system.

*complexVariable* is a symbol that represents the complex variable.

{*f*1,*f*2} is the plot range of digital frequencies.

*opts* are options; *opts* can contain any `Plot` options (see the *Mathematica* `Plot` function for details).

*optsPlot* are the default `Plot` options. See the *Mathematica* `Plot` function for details.

`dBMagnitudePlot→True` plots magnitude in decibels, $20\log_{10}(x)$.

`dBMagnitudePlot→False` plots magnitude in linear scale.

`z` is a reserved symbol for the complex variable.

`Options[DiscreteSystemMagnitudeResponsePlot]` gives a list of the current default settings for all options. You can reset the default using

`SetOptions`[*function*, *option*→*value*]. For example,

`SetOptions[DiscreteSystemMagnitudeResponsePlot, dBMagnitudePlot→False]`.

See also: `DiscreteSystemFrequencyResponse`, `Plot`

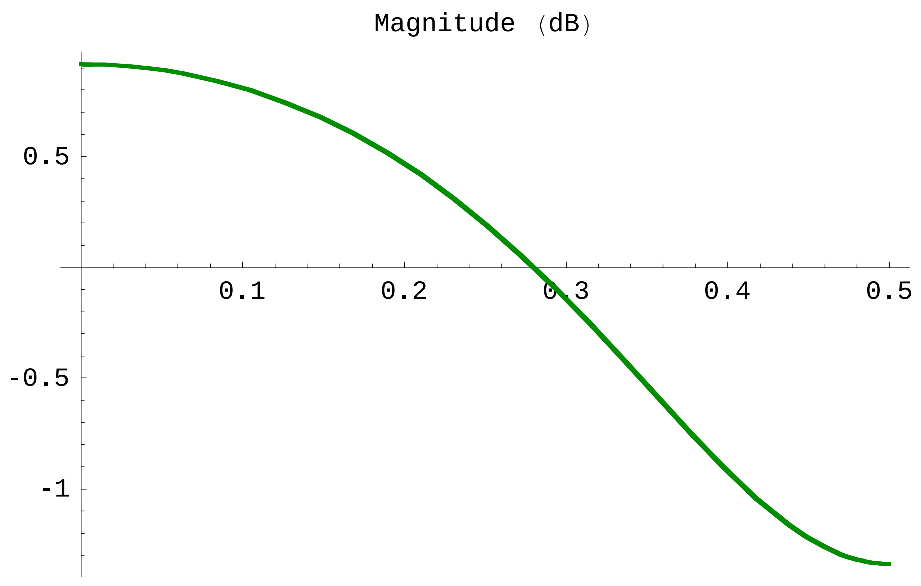
### Example

Here is the transfer function of a discrete system:

```
H = (1 / 4 + z) / (z + 1 / 8)
```

$$\frac{\frac{1}{4}+z}{\frac{1}{8}+z}$$

```
DiscreteSystemMagnitudeResponsePlot[H, PlotLabel → "Magnitude (dB)"]
```



## DiscreteSystemProcessingSISO

`DiscreteSystemProcessingSISO` processes a list of data samples, for a given transfer function, with a single-input single-output Transposed Direct Form 2 IIR discrete system.

{*outputDataList*, *finalConditions*} = `DiscreteSystemProcessingSISO`[*inputDataList*, *transferFunction*, *complexVariable*, *initialConditions*]

`DiscreteSystemProcessingSISO`[*inputDataList*, *transferFunction*, *complexVariable*] defaults to `DiscreteSystemProcessingSISO`[*inputDataList*, *transferFunction*, *complexVariable*, `{0,0,...}`]

`DiscreteSystemProcessingSISO`[*inputDataList*, *transferFunction*] defaults to `DiscreteSystemProcessingSISO`[*inputDataList*, *transferFunction*, `z`, `{0,0,...}`]

*inputDataList* is a list of data samples to be processed.

*transferFunction* is the transfer function.

*complexVariable* is a symbol that represents the complex variable.

*initialConditions* is a list of initial states. If omitted, zero initial conditions are assumed.

*outputDataList* is a list of processed data samples.

*finalConditions* is a list of final states.

`z` is a reserved symbol for the complex variable.

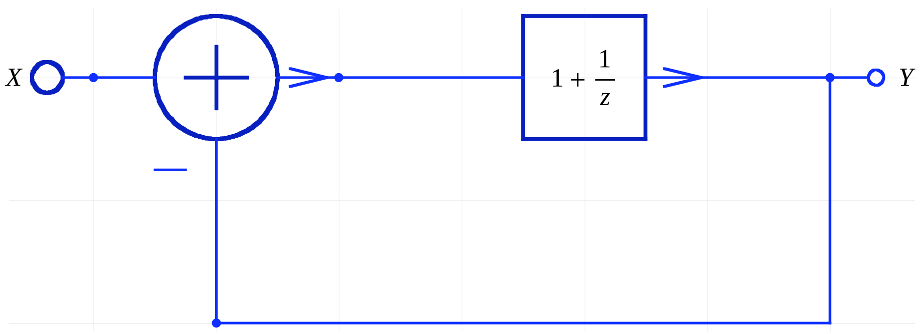
`DiscreteSystemProcessingSISO` implements the causal Transposed Direct Form 2 IIR SISO discrete system. It can process symbolic samples.

See also: `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

### Example

Consider the SISO discrete system

```
feedbackSystem = {
   {"Input", {0, 0}, X}, {"Output", {6, 0}, Y},
   {"Block", {{2, 0}, {6, 0}}, 1 + 1 / z},
   {"Adder", {{0, 0}, {1, -2}, {2, 0}, {1, 1}}, {1, -1, 2, 0}},
   {"Line", {{6, 0}, {6, -2}, {1, -2}}} };
ShowSchematic[%, Frame → False];
```



Here is the transfer function:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[feedbackSystem];

H = tfMatrix // First;
DiscreteSystemDisplayForm[H]
```

$$\frac{1 + z^{-1}}{2 + z^{-1}}$$

Assume the following data samples:

```
dataSamples = {1, 1, 1, -1, -1, -1, 0, x, 0}

{1, 1, 1, -1, -1, -1, 0, x, 0}
```

Process the data samples, assuming zero initial conditions, with

```
{outDataList, finalCond} = DiscreteSystemProcessingSISO[dataSamples, H];
outDataList // Simplify
```

$$\left\{\frac{1}{2}, \frac{3}{4}, \frac{5}{8}, -\frac{5}{16}, -\frac{27}{32}, -\frac{37}{64}, -\frac{27}{128}, \frac{27}{256} + \frac{x}{2}, -\frac{27}{512} + \frac{x}{4}\right\}$$

## TranslateSchematic

`TranslateSchematic` translates schematic in horizontal and vertical direction.

*translatedSchematicSpec* = `TranslateSchematic`[*schematicSpec*, {*Xshift*, *Yshift*}]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

{*Xshift*, *Yshift*} is a pair of numbers that specify translation in horizontal and vertical direction, respectively.

*translatedSchematicSpec* is the translated *schematicSpec* obtained by adding {*Xshift*, *Yshift*} to each coordinate of the schematic elements.

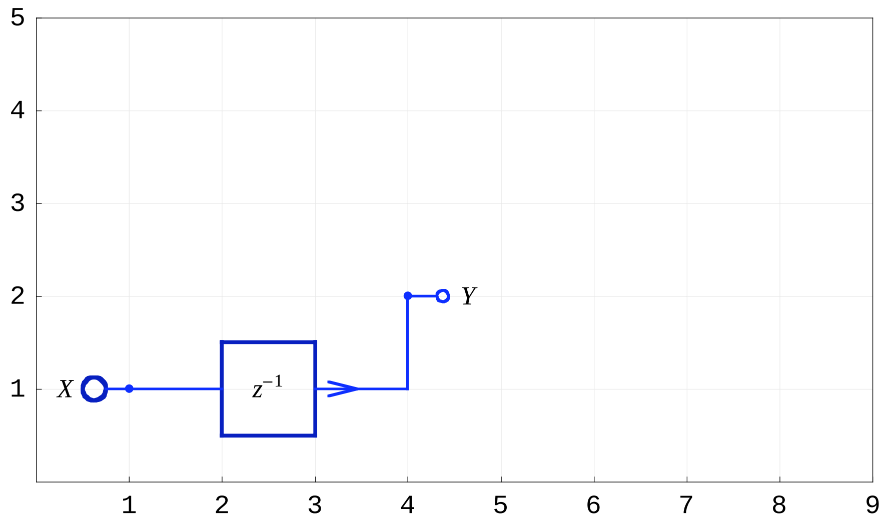
`TranslateSchematic` returns {} for unexpected arguments.

### Example

Here is the schematic specification of a discrete system:

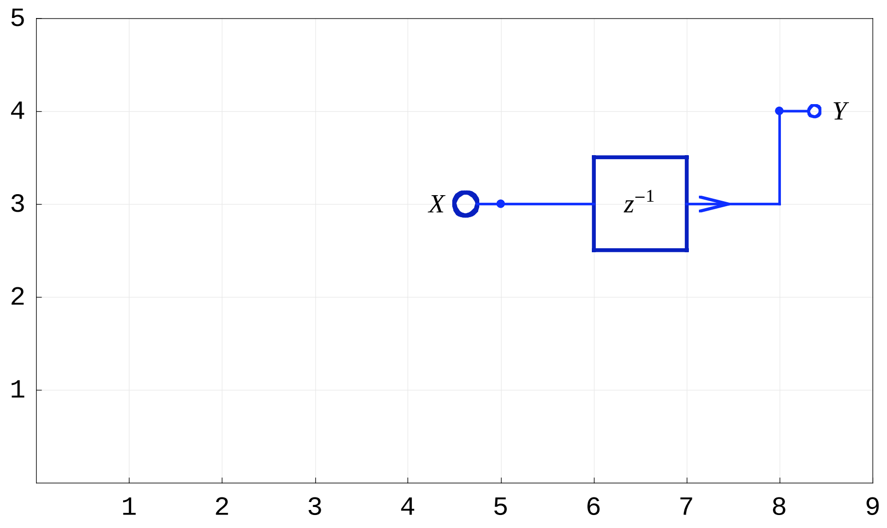
```
schematicSpec = {{"Input", {1, 1}, X}, {"Output", {4, 2}, Y},
  {"Delay", {{1, 1}, {4, 2}}, 1}}
ShowSchematic[%, PlotRange → {{0, 9}, {0, 5}}];
```

```
{{Input, {1, 1}, X}, {Output, {4, 2}, Y}, {Delay, {{1, 1}, {4, 2}}, 1}}
```



```
translatedSpec = TranslateSchematic[schematicSpec , {4, 2}]
ShowSchematic[%, PlotRange → {{0, 9}, {0, 5}}];
```

```
{{Input, {5, 3}, X}, {Output, {8, 4}, Y}, {Delay, {{5, 3}, {8, 4}}, 1}}
```



## $VersionSchematicSolverSchematicUtilities

`$VersionSchematicSolverSchematicUtilities` is a variable that contains information about the package version and release date.

```
Needs["SchematicSolver`"]
```

```
$VersionSchematicSolverSchematicUtilities
```

```
2.0 (February 04, 2004. 21:50)
```

## 14.7. Implementing Discrete Systems

### `AdjustSchematicCoordinates`

`AdjustSchematicCoordinates` returns schematic specification with non-negative element coordinates.

*adjustedSpec* = `AdjustSchematicCoordinates`[*schematicSpec*]

*schematicSpec* is a schematic specification that represents a system; it is a list of element specifications.

*adjustedSpec* is the schematic specification with non-negative element coordinates; it is required for generating implementation code.

See also: `DiscreteSystemImplementation`, `DiscreteSystemImplementationModule`, `DiscreteSystemSimulation`

## DemultiplexSequence

DemultiplexSequence converts a matrix of samples into a list of sequences.

{*sequenceA*, *sequenceB*, *sequenceC*, ..., *sequenceW*} = DemultiplexSequence[*sequenceMIMO*]

*sequenceMIMO* is a matrix of the form {{a[0], b[0], c[0], ..., w[0]}, {a[1], b[1], c[1], ..., w[1]}, ..., {a[K-1], b[K-1], c[K-1], ..., w[K-1]}}.

*sequenceA* is a K-by-1 matrix of the form {{a[0]}, {a[1]}, {a[2]}, ..., {a[K-1]}},

*sequenceB* is a K-by-1 matrix of the form {{b[0]}, {b[1]}, {b[2]}, ..., {b[K-1]}},

*sequenceC* is a K-by-1 matrix of the form {{c[0]}, {c[1]}, {c[2]}, ..., {c[K-1]}},

...

*sequenceW* is a K-by-1 matrix of the form {{w[0]}, {w[1]}, {w[2]}, ..., {w[K-1]}}.

DemultiplexSequence can be used to extract individual discrete signals from a multiplex sequence. Typically, *sequenceMIMO* is returned by DiscreteSystemImplementationProcessing or DiscreteSystemSimulation.

See also: MultiplexSequence, MultiplexDataList, ListToSequence, SequenceToList, SequencePlot, DiscreteSystemImplementationProcessing, DiscreteSystemSimulation

## `DiscreteSystemImplementation`

`DiscreteSystemImplementation` creates a *Mathematica* function that implements a system.

*codeString* = `DiscreteSystemImplementation`[*schematicSpec*, "*procedureName*"]

`DiscreteSystemImplementation`[*schematicSpec*] defaults to `DiscreteSystemImplementation`[*schematicSpec*, "`implementationProcedure`"]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

*procedureName* is the name of the function that implements the system.

*codeString* is a string of the code of the *Mathematica* function that implements the system.

Supported elements are Adder, Arrow, Delay (unit delay), Function, Input, Line, Modulator, Multiplier, Node, Output, Polyline, and Text.

*procedureName* is composed of letters and digits and should begin with a letter. *procedureName* should be enclosed within double quotation marks.

See also: `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`, `Discrete-SystemImplementationEquations`, `ValidImplementationModuleNameQ`

## `DiscreteSystemImplementationEquations`

`DiscreteSystemImplementationEquations` sets up equations for implementing a system.

{*inputVector*, *initialConditions*, *systemParameters*, *implementationEquations*, *outputVector*, *finalConditions*} = `DiscreteSystemImplementationEquations`[*schematicSpec*, *signalName*]

`DiscreteSystemImplementationEquations`[*schematicSpec*] defaults to `DiscreteSystemImplementationEquations`[*schematicSpec*, `Y`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

*signalName* is a symbol that represents signals at nodes of the system.

*inputVector* is a list of variables that represent inputs to the system.

*initialConditions* is a list of symbols that specify the initial state of the system.

*systemParameters* is a list of symbols that denote system parameters.

*implementationEquations* is a list of equations that represent the software implementation of the system.

*outputVector* is a list of symbols that represent outputs from the system.

*finalConditions* is a list of symbols that specify the final state of the system.

Supported elements are Adder, Arrow, Delay (unit delay), Function, Input, Line, Modulator, Multiplier, Node, Output, Polyline, and Text.

`DiscreteSystemImplementationEquations` returns {{}, {}, {}, {}, {}, {}} for systems that cannot be implemented.

See also: `DiscreteSystemImplementationModule`, `DiscreteSystemImplementation`, `Discrete-SystemImplementationSummary`, `DiscreteSystemSimulation`

## DiscreteSystemImplementationModule

DiscreteSystemImplementationModule creates a *Mathematica* function that implements the system, for the given implementation equations, and returns the function code as a string.

*codeString* = DiscreteSystemImplementationModule[*implementationEqnsObject*, "*procedureName*"]

DiscreteSystemImplementationModule[*implementationEqnsObject*] defaults to DiscreteSystemImplementationModule[*implementationEqnsObject*, "implementationProcedure"]

*implementationEqnsObject* is the list returned by DiscreteSystemImplementationEquations.

*procedureName* is the name of the function that implements the system.

*codeString* is a string of the code of the *Mathematica* function that implements the system.

Supported elements are Adder, Arrow, Delay (unit delay), Function, Input, Line, Modulator, Multiplier, Node, Output, Polyline, and Text.

*procedureName* is composed of letters and digits and should begin with a letter. *procedureName* should be enclosed within double quotation marks.

DiscreteSystemImplementationModule returns an empty string "" if the code for the implementation procedure cannot be generated.

See also: DiscreteSystemImplementationEquations, DiscreteSystemImplementationProcessing, DiscreteSystemSimulation, ValidImplementationModuleNameQ, DiscreteSystemImplementation

## `DiscreteSystemImplementationProcessing`

`DiscreteSystemImplementationProcessing` processes a data sequence with an existing implementation procedure.

{*outputSequence*, *finalConditions*} =
`DiscreteSystemImplementationProcessing`[*inputSequence*, *initialConditions*, *systemParameters*, *procedureName*]

*inputSequence* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, {a[2], b[2], c[2], ...}, ...}.

a[0], a[1], a[2], ... are samples to the first input;

b[0], b[1], b[2], ... are samples to the second input;

c[0], c[1], c[2], ... are samples to the third input; and so on.

{a[k], b[k], c[k], ...} are k-th samples to all inputs.

*initialConditions* is a list of initial states.

*systemParameters* is a list of system parameters.

*procedureName* is a symbol that represents the name of the function that implements the system.

*outputSequence* is a matrix of the form

{{u[0], v[0], w[0], ...}, {u[1], v[1], w[1], ...}, {u[2], v[2], w[2], ...}, ...}.

u[0], u[1], u[2], ... are samples from the first output,

v[0], v[1], v[2], ... are samples from the second output,

w[0], w[1], w[2], ... are samples from the third output, and so on.

{u[k], v[k], w[k], ...} are k-th samples from all outputs.

*finalConditions* is a list of final states.

The function named *procedureName* should exist prior to calling `DiscreteSystemImplementationProcessing`. That function can be created by `DiscreteSystemImplementation` or `DiscreteSystemImplementationModule`.

The arguments *inputSequence*, *initialConditions*, and *systemParameters* should match the corresponding arguments of the function named *procedureName*.

Use `DiscreteSystemImplementationSummary` to identify inputs, initial state, parameters, outputs, and final state.

Use `DiscreteSystemImplementationEquations` to extract the list of system parameters.

`DiscreteSystemImplementationProcessing` returns {{},{}} in the case of unexpected arguments.

See also: `DiscreteSystemImplementation`, `DiscreteSystemImplementationModule`, `DiscreteSystemImplementationSummary`, `DiscreteSystemImplementationEquations`, `MultiplexSequence`, `MultiplexDataList`, `DemultiplexSequence`, `SequencePlot`, `ListToSequence`, `SequenceToList`, `DiscreteSystemSimulation`

## DiscreteSystemImplementationSummary

`DiscreteSystemImplementationSummary` prints a summary of system implementation.

`DiscreteSystemImplementationSummary`[*schematicSpec*, *options*]

`DiscreteSystemImplementationSummary`[*schematicSpec*] defaults to `DiscreteSystemImplementationSummary`[*schematicSpec*, `Verbose→False`]

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

The summary identifies inputs, initial state, parameters, implementation equations, outputs, and final state.

`DiscreteSystemImplementationSummary`[*schematicSpec*, `Verbose→True`] prints summary with the implementation equations included.

Supported elements are Adder, Arrow, Delay (unit delay), Function, Input, Line, Modulator, Multiplier, Node, Output, Polyline, and Text.

See also: `DiscreteSystemSimulation`, `DiscreteSystemImplementationProcessing`, `Discrete-SystemImplementation`, `DiscreteSystemImplementationEquations`

## `DiscreteSystemSimulation`

`DiscreteSystemSimulation` simulates a system with zero initial conditions.

---

*outputSequence* = `DiscreteSystemSimulation`[*schematicSpec*, *inputSequence*]

`DiscreteSystemSimulation`[*schematicSpec*] assumes the unit impulse sequence for all inputs.

---

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

*inputSequence* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, {a[2], b[2], c[2], ...}, ...}.

a[0], a[1], a[2], ... are samples to the first input,

b[0], b[1], b[2], ... are samples to the second input,

c[0], c[1], c[2], ... are samples to the third input, and so on.

{a[k], b[k], c[k], ...} are k-th samples to all inputs.

*outputSequence* is a matrix of the form

{{u[0], v[0], w[0], ...}, {u[1], v[1], w[1], ...}, {u[2], v[2], w[2], ...}, ...}.

u[0], u[1], u[2], ... are samples from the first output,

v[0], v[1], v[2], ... are samples from the second output,

w[0], w[1], w[2], ... are samples from the third output, and so on.

{u[k], v[k], w[k], ...} are k-th samples from all outputs.

The first input corresponds to the first Input element in *schematicSpec*, the second input corresponds to the second Input element in *schematicSpec*, and so on. The same convention applies to the numbering of outputs.

Supported elements are Adder, Arrow, Delay (unit delay), Function, Input, Line, Modulator, Multiplier, Node, Output, Polyline, and Text.

`DiscreteSystemSimulation` returns {} in the case of unexpected arguments.

See also: `SequencePlot`, `MultiplexSequence`, `MultiplexDataList`, `DemultiplexSequence`, `ListToSequence`, `SequenceToList`, `DiscreteSystemImplementationSummary`, `Discrete-SystemImplementationProcessing`

## `DownsampleSequence`

`DownsampleSequence` downsamples a data sequence.

---

*downSequence* = `DownsampleSequence`[*dataSequence*, *n*, *d*]

`DownsampleSequence`[*dataSequence*, *n*] defaults to `DownsampleSequence`[*dataSequence*, *n*, 0]

---

*dataSequence* is a matrix of data samples.

*n* is the amount of downsampling.

*d* is the number of samples to drop prior to downsampling.

*downSequence* is the downsampled sequence.

`DownsampleSequence` takes every *n*-th row of *dataSequence* starting from the first row.

`DownsampleSequence` returns {} in the case of unexpected arguments.

See also: `UpsampleSequence`, `MultiplexSequence`, `MultiplexDataList`, `DemultiplexSequence`, `ListToSequence`, `SequenceToList`, `MultirateDownsampleSequence`, `SequencePlot`

## FirstSampleIndex

FirstSampleIndex is an option for SequencePlot that specifies the index of the first sample.

FirstSampleIndex → *i*

The option value, *i*, is a number.

FirstSampleIndex→0 is default.

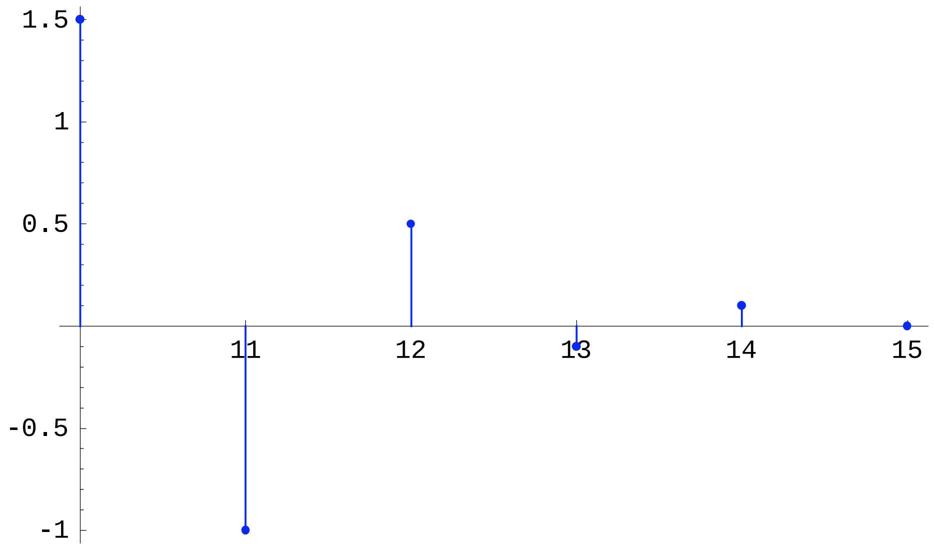
You can reset the default using SetOptions[*function*, *option*→*value*].

For example, SetOptions[SequencePlot, FirstSampleIndex→1].

See also: SequencePlot

## Examples

```
Needs["SchematicSolver`"]

mySeq = {{1.5}, {-1}, {0.5}, {-0.1}, {0.1}, {0}};

SequencePlot[mySeq, FirstSampleIndex → 10];
```



## `ListToSequence`

`ListToSequence` converts a list of data samples into the data sequence.

---

*dataSequence* = `ListToSequence`[*dataList*]

---

*dataList* is a list of K samples of the form {x[0], x[1], ..., x[k], ..., x[K-1]}, where k denotes the sample index.

*dataSequence* is a K-by-1 matrix of samples of the form {{x[0]}, {x[1]}, ..., {x[k]}, ..., {x[K-1]}}.

Typically, you can use `ListToSequence` to convert lists of measured data, or lists of samples generated by other programs, into the format required by *SchematicSolver*. For example, `DiscreteSystemProcessingSISO` returns a list of data, so you should use `ListToSequence` to plot the returned data with `SequencePlot`.

`ListToSequence` returns {} in the case of unexpected arguments.

See also: `SequenceToList`, `MultiplexDataList`, `DiscreteSystemProcessingSISO`, `SequencePlot`

## MultiplexDataList

`MultiplexDataList` converts a set of lists of data samples into a matrix of samples.

*sequenceMIMO* = `MultiplexDataList`[*dataListA*, *dataListB*, ..., *dataListW*]

*dataListA* is a list of the form {a[0], a[1], ..., a[K-1]},

*dataListB* is a list of the form {b[0], b[1], ..., b[K-1]},

...

*dataListW* is a list of the form {w[0], w[1], ..., w[K-1]}.

*sequenceMIMO* is a matrix of the form {{a[0], b[0], ..., w[0]}, {a[1], b[1], ..., w[1]}, ..., {a[K-1], b[K-1], ..., w[K-1]}}.

All arguments passed to `MultiplexDataList` should be lists of the same length.

`MultiplexDataList` can be used to combine lists of data samples into an input sequence passed to `DiscreteSystemImplementationProcessing` or `DiscreteSystemSimulation`.

`MultiplexDataList` returns {} in the case of unexpected arguments.

See also: `MultiplexSequence`, `DemultiplexSequence`, `ListToSequence`, `SequenceToList`, `SequencePlot`, `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

## `MultiplexSequence`

`MultiplexSequence` converts a set of sequences into a matrix of samples.

*sequenceMIMO* = `MultiplexSequence`[*sequenceA*, *sequenceB*, ..., *sequenceW*]

*sequenceA* is an N-by-K matrix of the form

{{a[1,1], a[1,2], ..., a[1,K]}, {a[2,1], a[2,2], ..., a[2,K]}, ..., {a[N,1], a[N,2], ..., a[N,K]}},

*sequenceB* is an N-by-L matrix of the form

{{b[1,1], b[1,2], ..., b[1,L]}, {b[2,1], b[2,2], ..., b[2,L]}, ..., {b[N,1], b[N,2], ..., b[N,L]}},

...

*sequenceW* is an N-by-M matrix of the form

{{w[1,1], w[1,2], ..., w[1,M]}, {w[2,1], w[2,2], ..., w[2,M]}, ..., {w[N,1], w[N,2], ..., w[N,M]}}.

*sequenceMIMO* is an N-by-(K+L+...+M) matrix of the form

{{a[1,1], a[1,2], ..., a[1,K], b[1,1], b[1,2], ..., b[1,L], ..., w[1,1], w[1,2], ..., w[1,M]},

{a[2,1], a[2,2], ..., a[2,K], b[2,1], b[2,2], ..., b[2,L], ..., w[2,1], w[2,2], ..., w[2,M]},

...

{a[N,1], a[N,2], ..., a[N,K], b[N,1], b[N,2], ..., b[N,L], ..., w[N,1], w[N,2], ..., w[N,M]}}.

All arguments passed to `MultiplexSequence` should be matrices of the same number of rows.

`MultiplexSequence` can be used to combine sequences into an input sequence passed to `DiscreteSystemImplementationProcessing` or `DiscreteSystemSimulation`.

`MultiplexSequence` returns {} in the case of unexpected arguments.

See also: `DemultiplexSequence`, `MultiplexDataList`, `ListToSequence`, `SequenceToList`, `SequencePlot`, `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

## MultirateDownsampleSequence

`MultirateDownsampleSequence` downsamples a data sequence for multirate processing. Converts a sequence that represents a single discrete signal into a multiplex sequence.

*downSequence* = `MultirateDownsampleSequence`[*dataSequence*, *n*]

*dataSequence* is an m-by-1 matrix of data samples that represents one discrete signal; it is of the form

{{x[0]}, {x[1]}, ..., {x[m-1]}}

where m is the number of samples.

*n* is the amount of downsampling.

*downSequence* is the downsampled sequence. It is a K-by-n matrix of the form

{{x[0], x[1], ..., x[n-1]}, {x[n], x[1+n], ..., x[2n-1]}, ..., {x[(K-1)*n], x[1+(K-1)*n], ..., x[K*n-1]}}

where

K = `IntegerPart`[m/n].

`MultirateDownsampleSequence` takes every *n*-th row of *dataSequence* starting from the first row.

`MultirateDownsampleSequence` returns {} in the case of unexpected arguments.

See also: `DownsampleSequence`, `UpsampleSequence`, `MultiplexSequence`, `DemultiplexSe-quence`, `SequencePlot`

## `NormalizedSpectrum`

`NormalizedSpectrum` is an option for `SequenceDiscreteFourierTransformMagnitudePlot` and `SequenceFourierTransformMagnitudePlot`.

`NormalizedSpectrum→True` plots the normalized spectrum.

`NormalizedSpectrum→False` plots the spectrum without normalization.

The option value of `True` plots the normalized spectrum DFT/*NumberOfSamples* or DTFT/*NumberOfSamples*, where DFT denotes spectral components of Discrete Fourier Transform (a list of complex quantities), and DTFT denotes Discrete-Time Fourier Transform (an expression in terms of the digital frequency).

`NormalizedSpectrum→True` is default.

You can reset the default using `SetOptions`[*function*, *option*→*value*].

For example, `SetOptions`[`SequenceFourierTransformMagnitudePlot`, `NormalizedSpectrum→False`].

## Power2

`Power2` squared value.

*y* = `Power2`[*x*]

*x* is an expression.

*y* = *x*^2

`Power2` can be used to specify the value of the Function element.

See also: Function element in the *SchematicSolver*'s help

### Examples

```
Needs["SchematicSolver`"]
```

```
Power2[1 / 4]
```

$$\frac{1}{16}$$

```
Power2[myCoeff]
```

$\text{myCoeff}^2$

## previousSample

`previousSample` is a reserved symbol in *SchematicSolver*.

---

`previousSample` is a wrapper that denotes the delayed sample.

---

See also: `DiscreteSystemImplementationEquations`

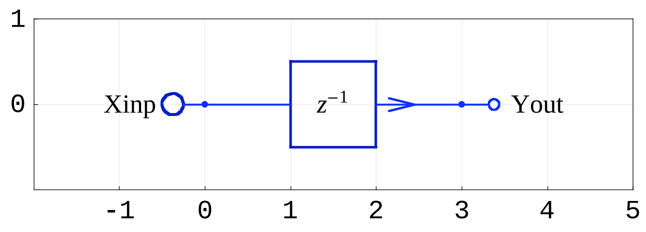
### Examples

```
Needs["SchematicSolver`"]
```

```
schematicSpecification =
 {{"Input", {0, 0}, Xinp}, {"Delay", {{0, 0}, {3, 0}}, 1}, {"Output", {3, 0}, Yout}}
```

```
{{Input, {0, 0}, Xinp}, {Delay, {{0, 0}, {3, 0}}, 1}, {Output, {3, 0}, Yout}}
```

```
ShowSchematic[schematicSpecification, PlotRange → {{-2, 5}, {-1, 1}}];
```



```
DiscreteSystemImplementationEquations[schematicSpecification] // ColumnForm
```

```
{Y[{0, 0}]}
{Y[{3, 0}]}
{}
{Y[{0, 0}] == Xinp, Y[{3, 0}] == previousSample[Y[{0, 0}]]}
{Y[{3, 0}]}
{Y[{0, 0}]}
```

## SequenceDiscreteFourierTransform

`SequenceDiscreteFourierTransform` computes Discrete Fourier Transform (DFT) of a matrix of samples.

*dft* = `SequenceDiscreteFourierTransform`[*sampleMatrix*]

*sampleMatrix* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, {a[2], b[2], c[2], ...}, ...}.

Each column represents a discrete signal.

a[0], a[1], a[2], ... represent samples of the first signal,

b[0], b[1], b[2], ... represent samples of the second signal,

c[0], c[1], c[2], ... represent samples of the third signal, and so on.

*dft* is a matrix of DFT components. Each column of *dft* represents spectral components of a discrete signal.

`SequenceDiscreteFourierTransform` computes DFT for each column of *sampleMatrix* by using `Fourier`[*dataList*, `FourierParameters`→`{1,-1}`].

DFT of a sample list $\{x_0, x_1, \ldots, x_n, \ldots, x_{N-1}\}$ is defined by the list of spectral components $\{X_0, X_1, \ldots, X_k, \ldots, X_{N-1}\}$, where $X_k = \sum_{n=0}^{N-1} x_n\, w^{-k\,n}$, $w = e^{i\,\frac{2\pi}{N}}$.

`SequenceDiscreteFourierTransform` returns {} in the case of unexpected arguments.

See also: `SequenceDiscreteFourierTransformMagnitudePlot`, `Fourier`, `SequenceFourier-Transform`, `SequenceFourierTransformMagnitudePlot`

## `SequenceDiscreteFourierTransformMagnitudePlot`

`SequenceDiscreteFourierTransformMagnitudePlot` plots Discrete Fourier Transform (DFT) magnitude of a matrix of samples.

`SequenceDiscreteFourierTransformMagnitudePlot`[*sampleMatrix*, *opts*]

`SequenceDiscreteFourierTransformMagnitudePlot`[*sampleMatrix*] defaults to `SequenceDiscreteFourierTransformMagnitudePlot`[*sampleMatrix*, `dBMagnitudePlot`→`False`, `NormalizedSpectrum`→`True`, `StemPlot`→`True`, `PlotJoined`→`False`, `SequencePointSize`→0.01, `SequenceLineThickness`→0.003, `SequenceSamplingFrequency`→1, *optsGraphics*]

*sampleMatrix* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, {a[2], b[2], c[2], ...}, ...}.

Each column represents a discrete signal.

a[0], a[1], a[2], ... represent samples of the first signal,

b[0], b[1], b[2], ... represent samples of the second signal,

c[0], c[1], c[2], ... represent samples of the third signal, and so on.

*opts* are options; *opts* can contain any `Graphics` options; see the *Mathematica* `Graphics` function for details.

`dBMagnitudePlot`→`True` plots magnitude in decibels.

`NormalizedSpectrum`→`False` plots magnitude spectrum without normalization.

`StemPlot`→`False` does not plot stems that represent spectral components.

`PlotJoined`→`True` joins the dots that represent spectral components.

`SequencePointSize`→*p* sets the relative size of dots that represent spectral components; *p* is a number from 0 to 1.

`SequenceLineThickness`→*t* sets the relative thickness of stems that represent spectral components; *t* is a number from 0 to 1.

`SequenceSamplingFrequency`→*Fsamp* sets the sampling frequency to *Fsamp*; *Fsamp* is a positive number.

*optsGraphics* are the default `Graphics` options. See the *Mathematica* `Graphics` function for details.

`Options`[`SequenceDiscreteFourierTransformMagnitudePlot`] gives a list of the current default settings for all options.

You can reset the default using `SetOptions`[*function*, *option*→*value*]. For example, `SetOptions`[`SequenceDiscreteFourierTransformMagnitudePlot`, `PlotJoined`→`True`].

You can plot spectral components versus the continuous-time frequency if you specify the sampling frequency: `SequenceSamplingFrequency`→*Fsamp*. In that case, the intercomponent interval (the frequency resolution) equals *Fsamp*/*K*, where *K* is the number of data samples.

You can plot spectral components versus the digital frequency if you specify the unit sampling frequency: `SequenceSamplingFrequency`→1.

`SequenceDiscreteFourierTransformMagnitudePlot` plots DFT magnitude as a discrete function of frequency.

`SequenceDiscreteFourierTransformMagnitudePlot` plots DFT magnitude of column 1 in brown, column 2 in green, column 3 in blue, and cycles through the three colors for columns 4, 5, etc.

See also: `SequenceDiscreteFourierTransform`, `NormalizedSpectrum`, `SequenceFourierTrans-form`, `SequenceFourierTransformMagnitudePlot`, `Fourier`, `SequencePlot`

## SequenceFourierTransform

`SequenceFourierTransform` computes Discrete-Time Fourier Transform (DTFT) of a matrix of samples.

*sft* = `SequenceFourierTransform`[*sampleMatrix*, *digitalFrequency*, *opts*]

`SequenceFourierTransform`[*sampleMatrix*, *digitalFrequency*] defaults to `SequenceFourierTransform`[*sampleMatrix*, *digitalFrequency*, `SequenceSamplingFrequency`→1]

`SequenceFourierTransform`[*sampleMatrix*] defaults to `SequenceFourierTransform`[*sampleMatrix*, `f`, `SequenceSamplingFrequency`→1]

*sampleMatrix* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, {a[2], b[2], c[2], ...}, ...}.

Each column represents a discrete signal.

a[0], a[1], a[2], ... represent samples of the first signal,

b[0], b[1], b[2], ... represent samples of the second signal,

c[0], c[1], c[2], ... represent samples of the third signal, and so on.

*digitalFrequency* is a symbol that denotes the digital frequency.

*opts* are options.

*sft* is a list of the form {ft[1], ft[2], ft[3], ...}, where ft[1] is DTFT of the first signal, ft[2] is DTFT of the second signal, ft[3] is DTFT of the third signal, and so on.

`f` is a reserved symbol for the digital frequency.

`SequenceFourierTransform` computes DTFT, in terms of *digitalFrequency*, for each column of *sampleMatrix*.

`SequenceSamplingFrequency`→*Fsamp* sets the sampling frequency to *Fsamp*. In that case, *digitalFrequency* represents the continuous-time frequency.

You can reset the default using `SetOptions`[*function*, *option*→*value*]. For example, `SetOptions`[`SequenceFourierTransform`, `SequenceSamplingFrequency`→8000].

`SequenceFourierTransform` returns {} in the case of unexpected arguments.

`SequenceFourierTransform` computes DTFT of a sample list $\{x_0, x_1, \ldots, x_n, \ldots, x_{N-1}\}$ according to the formula $X(f) = \sum_{n=0}^{N-1} x_n\, w^{-n}$, where $w = e^{i\, 2\, \pi\, f\, \frac{1}{F}}$, and $F$ is the sampling frequency. It is assumed that the index of the first sample is equal to zero.

See also: `SequenceFourierTransformMagnitudePlot`, `SequenceDiscreteFourierTransform`, `SequenceDiscreteFourierTransformMagnitudePlot`

### Examples

```
Needs["SchematicSolver`"]
```

```
seq = UnitSymbolicSequence[6, x, 0]
```

```
{{x0}, {x1}, {x2}, {x3}, {x4}, {x5}}
```

```
SequenceFourierTransform[seq]
```

$\{x0 + e^{-2 i f \pi} x1 + e^{-4 i f \pi} x2 + e^{-6 i f \pi} x3 + e^{-8 i f \pi} x4 + e^{-10 i f \pi} x5\}$

```
SequenceFourierTransform[seq, SequenceSamplingFrequency → Fsamp]
```

$\left\{x0 + e^{-\frac{2 i f \pi}{Fsamp}} x1 + e^{-\frac{4 i f \pi}{Fsamp}} x2 + e^{-\frac{6 i f \pi}{Fsamp}} x3 + e^{-\frac{8 i f \pi}{Fsamp}} x4 + e^{-\frac{10 i f \pi}{Fsamp}} x5\right\}$

## SequenceFourierTransformMagnitudePlot

SequenceFourierTransformMagnitudePlot plots Discrete-Time Fourier Transform (DTFT) magnitude of a matrix of samples.

SequenceFourierTransformMagnitudePlot[*sampleMatrix*, {*f1*, *f2*}, *opts*]

SequenceFourierTransformMagnitudePlot[*sampleMatrix*, {*f1*, *f2*}] defaults to SequenceFourierTransformMagnitudePlot[*sampleMatrix*, {*f1*, *f2*}, dBMagnitudePlot→False, NormalizedSpectrum→True, SequenceSamplingFrequency→1, *optsPlot*]

SequenceFourierTransformMagnitudePlot[*sampleMatrix*, *opts*] defaults to SequenceFourierTransformMagnitudePlot[*sampleMatrix*, {-1/2, 1/2}, *opts*]

SequenceFourierTransformMagnitudePlot[*sampleMatrix*] defaults to SequenceFourierTransformMagnitudePlot[*sampleMatrix*, {-1/2, 1/2}]

*sampleMatrix* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, {a[2], b[2], c[2], ...}, ...}.

Each column represents a discrete signal.

a[0], a[1], a[2], ... represent samples of the first signal,

b[0], b[1], b[2], ... represent samples of the second signal,

c[0], c[1], c[2], ... represent samples of the third signal, and so on.

*f1* is a number that specifies the lower frequency.

*f2* is a number that specifies the upper frequency.

*opts* are options; *opts* can contain any Plot options; see the *Mathematica* Plot function for details.

dBMagnitudePlot→True plots magnitude in decibels.

NormalizedSpectrum→False plots the magnitude spectrum without normalization.

SequenceSamplingFrequency→*Fsamp* sets the sampling frequency to *Fsamp*; *Fsamp* is a positive number.

*optsPlot* are the default Plot options. See the *Mathematica* Plot function for details.

Options[SequenceFourierTransformMagnitudePlot] gives a list of the current default settings for all options.

You can reset the default using SetOptions[*function*, *option*→*value*]. For example, SetOptions[SequenceFourierTransformMagnitudePlot, SequenceSamplingFrequency→8000].

You can plot DTFT magnitude versus the continuous-time frequency if you specify the sampling frequency: SequenceSamplingFrequency→*Fsamp*.

You can plot DTFT magnitude versus the digital frequency if you specify the unit sampling frequency: SequenceSamplingFrequency→1.

SequenceFourierTransformMagnitudePlot plots DTFT magnitude as a continuous function of frequency, for each column of *sampleMatrix*, over the frequency range from *f1* to *f2*.

SequenceFourierTransformMagnitudePlot plots DTFT magnitude of column 1 in brown, column 2 in green, column 3 in blue, and cycles through the three colors for columns 4, 5, 6, etc.

See also: `SequenceFourierTransform`, `NormalizedSpectrum`, `SequenceDiscreteFourierTransform`, `SequenceDiscreteFourierTransformMagnitudePlot`, `SequencePlot`

## SequenceLineThickness

`SequenceLineThickness` is an option for `SequencePlot` and `SequenceDiscreteFourierTransform-MagnitudePlot` that specifies the relative thickness of stems which represent samples.

---

`SequenceLineThickness` → *t*

---

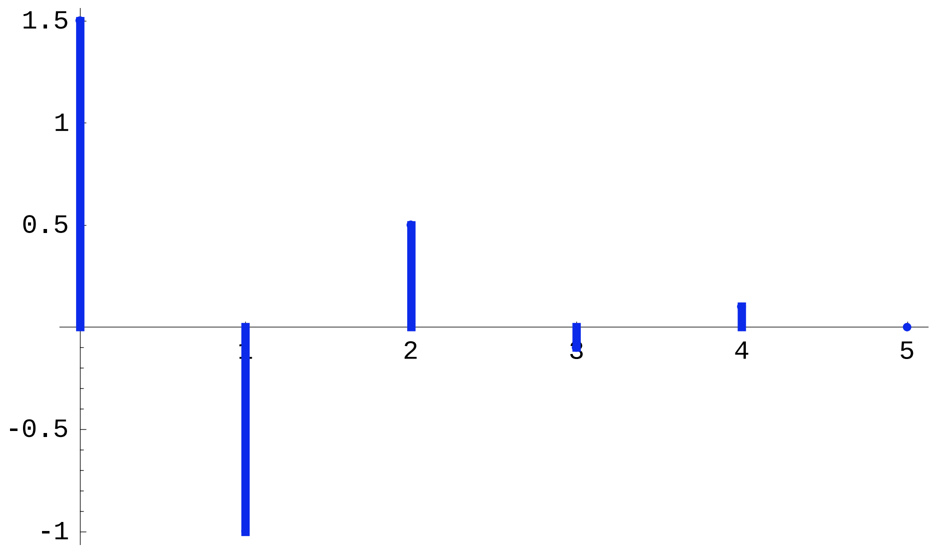
*t* is a number from 0 to 1.

`SequenceLineThickness`→0.003 is default.

You can reset the default using `SetOptions`[*function*, *option*→*value*]. For example, `SetOptions`[`SequencePlot`, `SequenceLineThickness`→0.001].

### Examples

```
Needs["SchematicSolver`"]

mySeq = {{1.5}, {-1}, {0.5}, {-0.1}, {0.1}, {0}};
SequencePlot[%, SequenceLineThickness → 0.01];
```



## SequencePlot

SequencePlot plots a matrix of samples.

SequencePlot[*sampleMatrix*, *opts*]

SequencePlot[*sampleMatrix*] defaults to SequencePlot[*sampleMatrix*, FirstSampleIndex→0, PlotJoined→False, SequenceLineThickness→0.003, SequencePointSize→0.01, SequenceSamplingFrequency→1, StemPlot→True, *optsGraphics*]

*sampleMatrix* is a matrix of the form

{{a[0], b[0], c[0], ...}, {a[1], b[1], c[1], ...}, ..., {a[k], b[k], c[k], ...}, ...}.

Each column represents a discrete signal.

a[0], a[1], ..., a[k], ... represent samples of the first signal,

b[0], b[1], ..., b[k], ... represent samples of the second signal,

c[0], c[1], ..., c[k], ... represent samples of the third signal, and so on.

*opts* are options; *opts* can contain any Graphics options; see the *Mathematica* Graphics function for details.

FirstSampleIndex→*i* sets the first sample index to *i*; *i* is a number.

PlotJoined→True joins the plot points that represent samples. This way you can draw envelopes of discrete signals.

SequenceLineThickness→*t* sets the relative thickness of stems that represent samples; *t* is a number from 0 to 1.

SequencePointSize→*p* sets the relative size of dots that represent samples; *p* is a number from 0 to 1.

SequenceSamplingFrequency→*Fsamp* sets the sampling frequency to *Fsamp*; *Fsamp* is a positive number.

StemPlot→False does not plot stems that represent samples.

*optsGraphics* are the default Graphics options. See the *Mathematica* Graphics function for details.

Options[SequencePlot] gives a list of the current default settings for all options.

You can reset the default using SetOptions[*function*, *option*→*value*]. For example, SetOptions[SequencePlot, PlotJoined→True].

You can plot samples versus time if you specify the sampling frequency: SequenceSamplingFrequency→*Fsamp*. In that case, the intersample interval equals 1/*Fsamp*. The first sample is plotted at the time point *i*/*Fsamp*, where *i* is the index of the first sample specified with FirstSampleIndex→*i*.

You can plot samples versus the sample index if you specify the unit sampling frequency: SequenceSamplingFrequency→1. The index of the first sample can be an arbitrary integer *i* specified with FirstSampleIndex→*i*.

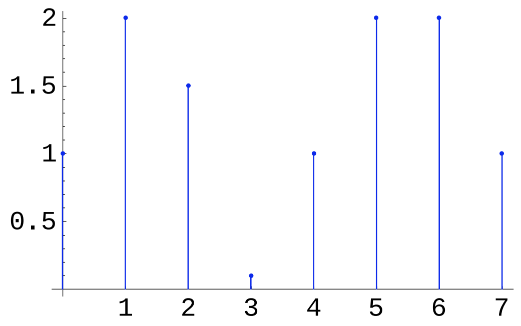
SequencePlot plots column 1 in blue, column 2 in red, column 3 in green, column 4 in black, column 5 in cyan, column 6 in magenta, column 7 in gray, column 8 in yellow, and cycles through the eight colors for columns 9, 10, etc.

See also: MultiplexSequence, MultiplexDataList, DemultiplexSequence, ListToSequence, SequenceToList, SequenceDiscreteFourierTransformMagnitudePlot, SequenceFourierTransformMagnitudePlot

## Examples

```
Needs["SchematicSolver`"]
```

```
SequencePlot[{{1}, {2}, {1.5}, {0.1}, {1}, {2}, {2}, {1}}];
```



## SequencePointSize

`SequencePointSize` is an option for `SequencePlot` and `SequenceDiscreteFourierTransformMagni-tudePlot` that specifies the relative size of dots which represent samples.

`SequencePointSize` → *p*

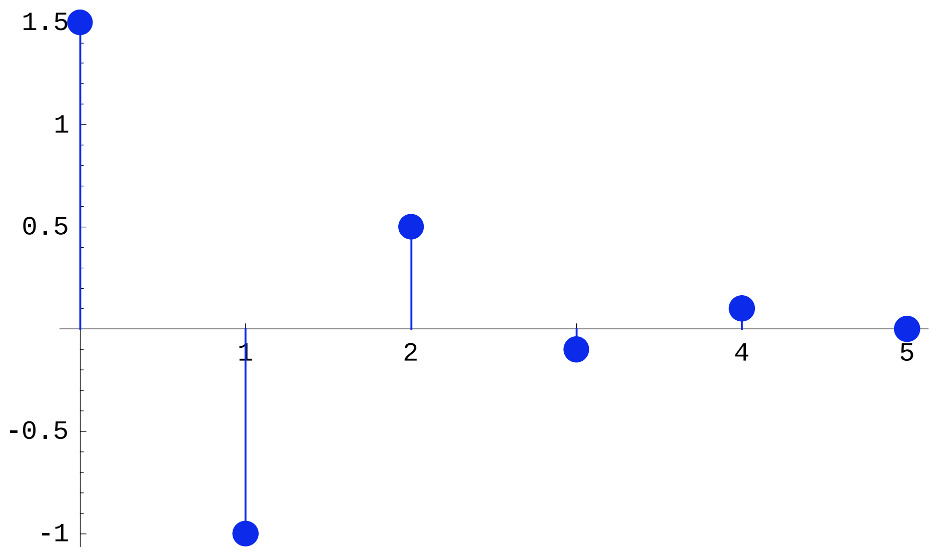
*p* is a number from 0 to 1.

`SequencePointSize`→0.01 is default.

You can reset the default using `SetOptions`[*function*, *option*→*value*]. For example, `SetOptions`[`SequencePlot`, `SequencePointSize`→0.001].

## Examples

```
Needs["SchematicSolver`"]

mySeq = {{1.5}, {-1}, {0.5}, {-0.1}, {0.1}, {0}};
SequencePlot[%, SequencePointSize → 0.03];
```



## SequenceSamplingFrequency

`SequenceSamplingFrequency` is an option for `SequenceFourierTransform`, `SequenceFourierTransformMagnitudePlot`, `SequenceDiscreteFourierTransformMagnitudePlot`, and `SequencePlot`, that specifies the sampling frequency.

`SequenceSamplingFrequency` → *`Fsamp`*

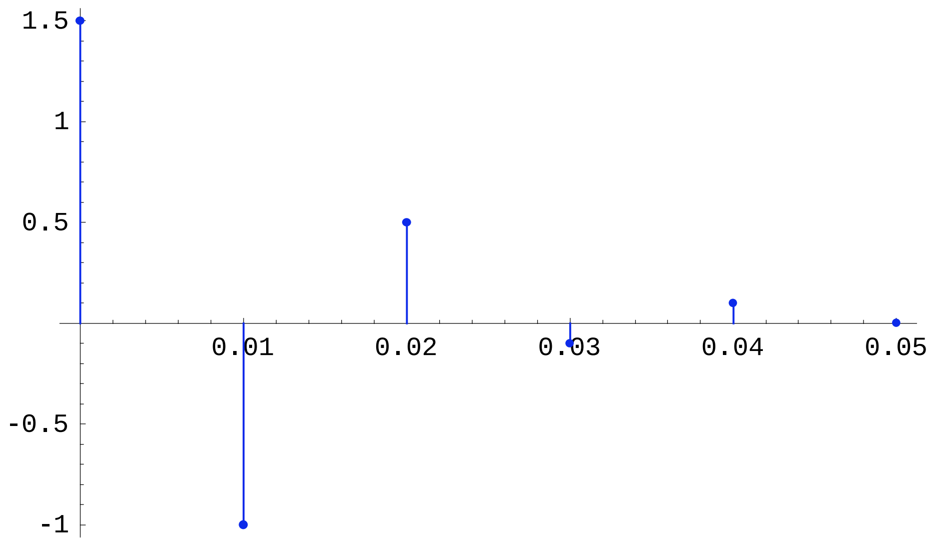
*`Fsamp`* is a positive number. It can be a symbol for `SequenceFourierTransform`.

`SequenceSamplingFrequency`→1 is default.

You can reset the default using `SetOptions`[*function*, *option*→*value*]. For example, `SetOptions`[`SequencePlot`, `SequenceSamplingFrequency`→8000].

## Examples

```
Needs["SchematicSolver`"]

mySeq = {{1.5}, {-1}, {0.5}, {-0.1}, {0.1}, {0}};
SequencePlot[%, SequenceSamplingFrequency → 100];
```



## `SequenceToList`

`SequenceToList` converts a data sequence into the list of data samples.

*dataList* = `SequenceToList`[*dataSequence*]

*dataSequence* is a K-by-1 matrix of samples of the form {{x[0]}, {x[1]}, {x[2]}, ..., {x[K-1]}}.

*dataList* is a list of K samples of the form {x[0], x[1], x[2], ..., x[K-1]}.

`SequenceToList` returns {} in the case of unexpected arguments.

See also: `ListToSequence`, `MultiplexDataList`, `ListPlot`

## StemPlot

StemPlot is an option for SequencePlot and SequenceDiscreteFourierTransformMagnitudePlot that specifies the appearance of stems.

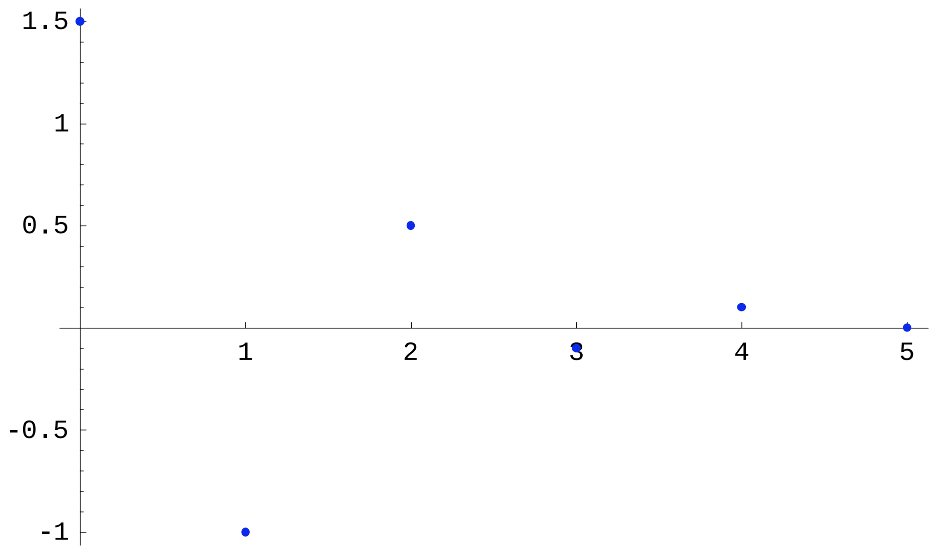
| | |
|---|---|
| StemPlot→True | plots stems. |
| StemPlot→False | does not plot stems. |

StemPlot→True is default.

You can reset the default using SetOptions[*function*, *option*→*value*]. For example, SetOptions[SequencePlot, StemPlot→False].

## Examples

```
Needs["SchematicSolver`"]

mySeq = {{1.5}, {-1}, {0.5}, {-0.1}, {0.1}, {0}};
SequencePlot[%, StemPlot → False];
```



## UndefinedSymbolQ

`UndefinedSymbolQ` tests whether a symbol has a value or a definition.

*undfsymb* = `UndefinedSymbolQ`[*symb*]

*symb* is a symbol.

*undfsymb* equals *symb* if *symb* is a symbol without a value or a definition; otherwise *undfsymb* is `False`.

### Examples

```
Needs["SchematicSolver`"]
```

```
UndefinedSymbolQ[myNewSymbol]
```

```
myNewSymbol
```

```
UndefinedSymbolQ[#] & /@ {Pi, 1.2, Sqrt, Solve, a + b, True}
```

```
{False, False, False, False, False, False}
```

```
myVar = 12; UndefinedSymbolQ[myVar]
```

```
False
```

```
Clear[myVar]; UndefinedSymbolQ[myVar]
```

```
myVar
```

## UnitExponentialSequence

`UnitExponentialSequence` generates a unit exponential sequence.

*seq* = `UnitExponentialSequence`[*n*, *expCoeff*, *expBase*]

`UnitExponentialSequence`[*n*, *expCoeff*] defaults to `UnitExponentialSequence`[*n*, *expCoeff*, 2]

`UnitExponentialSequence`[*n*] defaults to `UnitExponentialSequence`[*n*, –1, 2]

`UnitExponentialSequence`[] defaults to `UnitExponentialSequence`[8, –1, 2]

*n* is the number of samples.

*expCoeff* is the exponent coefficient.

*expBase* is the base.

*seq* is an *n*-by-1 matrix of the form {{a[0]}, {a[1]}, ..., {a[k]}, ..., {a[*n*-1]}}.

a[k] = *expBase*^(*expCoeff**k), k = 0, 1, 2, ..., (*n*-1).

`UnitExponentialSequence` returns {} in the case of unexpected arguments.

See also: `UnitImpulseSequence`, `UnitNoiseSequence`, `UnitRampSequence`, `UnitSineSequence`, `UnitStepSequence`, `UnitSymbolicSequence`, `MultiplexSequence`, `SequencePlot`, `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]
```

```
UnitExponentialSequence[4, c, b]
```

$\{\{1\}, \{b^c\}, \{b^{2c}\}, \{b^{3c}\}\}$

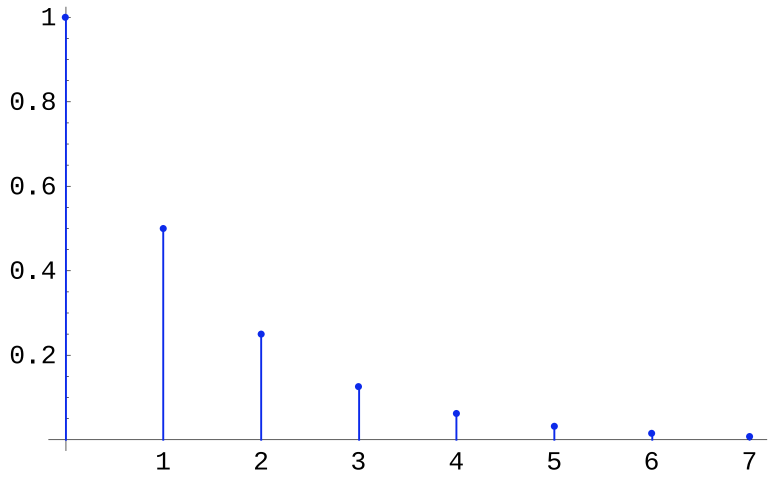
```
UnitExponentialSequence[4, c]
```

$\{\{1\}, \{2^c\}, \{2^{2c}\}, \{2^{3c}\}\}$

```
UnitExponentialSequence[]
```

$\left\{\{1\}, \left\{\frac{1}{2}\right\}, \left\{\frac{1}{4}\right\}, \left\{\frac{1}{8}\right\}, \left\{\frac{1}{16}\right\}, \left\{\frac{1}{32}\right\}, \left\{\frac{1}{64}\right\}, \left\{\frac{1}{128}\right\}\right\}$

```
UnitExponentialSequence[] // SequencePlot;
```



## UnitImpulseSequence

`UnitImpulseSequence` generates a unit impulse sequence.

*seq* = `UnitImpulseSequence`[*n*, *d*]

`UnitImpulseSequence`[*n*] defaults to `UnitImpulseSequence`[*n*, 0]

`UnitImpulseSequence`[] defaults to `UnitImpulseSequence`[8, 0]

*n* is the number of samples.

*d* is the delay, i.e., number of leading zero-samples; $d < n$.

*seq* is an *n*-by-1 matrix of the form {{a[0]}, {a[1]}, ..., {a[k]}, ..., {a[*n*-1]}}.

a[k] = 1 for k = *d*, a[k] = 0 otherwise.

`UnitImpulseSequence` returns {} in the case of unexpected arguments.

See also: `UnitExponentialSequence`, `UnitNoiseSequence`, `UnitRampSequence`, `UnitSineSequence`, `UnitStepSequence`, `UnitSymbolicSequence`, `MultiplexSequence`, `SequencePlot`, `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]
```

```
UnitImpulseSequence[6, 2]
```

```
{{0}, {0}, {1}, {0}, {0}, {0}}
```

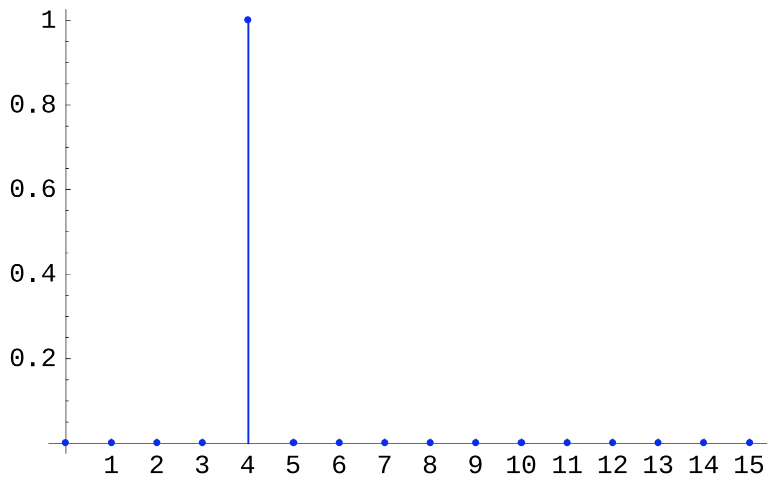
```
UnitImpulseSequence[6]
```

```
{{1}, {0}, {0}, {0}, {0}, {0}}
```

```
UnitImpulseSequence[]
```

```
{{1}, {0}, {0}, {0}, {0}, {0}, {0}, {0}}
```

```
UnitImpulseSequence[16, 4] // SequencePlot;
```



## UnitNoiseSequence

UnitNoiseSequence generates a unit noise sequence.

*seq* = UnitNoiseSequence[*n*, *d*]

UnitNoiseSequence[*n*] defaults to UnitNoiseSequence[*n*, 0]

UnitNoiseSequence[] defaults to UnitNoiseSequence[8, 0]

*n* is the number of samples.

*d* is the delay, i.e., number of leading zero-samples; $d < n$.

*seq* is an *n*-by-1 matrix of the form {{a[0]}, {a[1]}, ..., {a[k]}, ..., {a[*n*-1]}}.

–1 < a[k] < 1 for k ≥ *d*, a[k] = 0 otherwise. a[k] is a uniformly distributed random number.

UnitNoiseSequence returns {} in the case of unexpected arguments.

See also: UnitExponentialSequence, UnitImpulseSequence, UnitRampSequence, UnitSineSequence, UnitStepSequence, UnitSymbolicSequence, MultiplexSequence, SequencePlot, DiscreteSystemImplementationProcessing, DiscreteSystemSimulation

## Examples

```
Needs["SchematicSolver`"]
```

```
UnitNoiseSequence[6, 2]
```

```
{{0}, {0}, {0.834476}, {-0.242757}, {-0.894751}, {0.475183}}
```

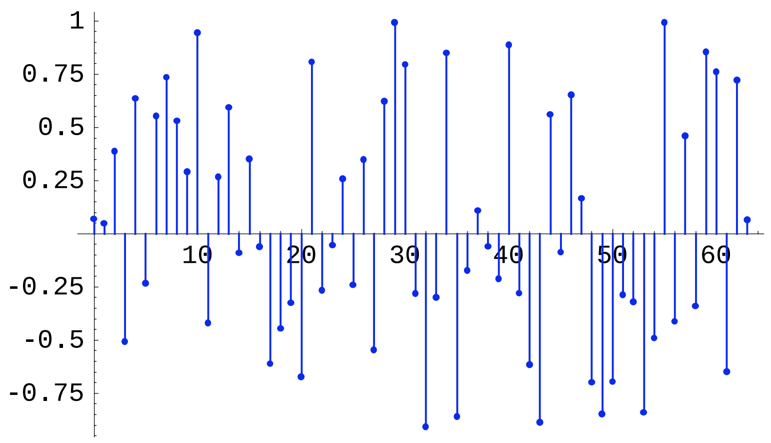
```
UnitNoiseSequence[6]
```

```
{{0.606943}, {0.155995}, {-0.736524}, {0.697687}, {0.0348474}, {0.226997}}
```

```
UnitNoiseSequence[]
```

```
{{-0.672455}, {0.205681}, {-0.644137},
 {-0.32274}, {-0.388033}, {-0.419805}, {0.820997}, {0.728346}}
```

```
UnitNoiseSequence[64] // SequencePlot;
```



## UnitRampSequence

UnitRampSequence generates a unit ramp sequence.

*seq* = UnitRampSequence[*n*]

UnitRampSequence[] defaults to UnitRampSequence[8]

*n* is the number of samples.

*seq* is an *n*-by-1 matrix of the form {{0}, {1}, {2}, (3), ..., {*n*-1}}.

UnitRampSequence returns {} in the case of unexpected arguments.

See also: UnitExponentialSequence, UnitImpulseSequence, UnitNoiseSequence, UnitSineSequence, UnitStepSequence, UnitSymbolicSequence, MultiplexSequence, SequencePlot, DiscreteSystemImplementationProcessing, DiscreteSystemSimulation

## Examples

```
Needs["SchematicSolver`"]
```

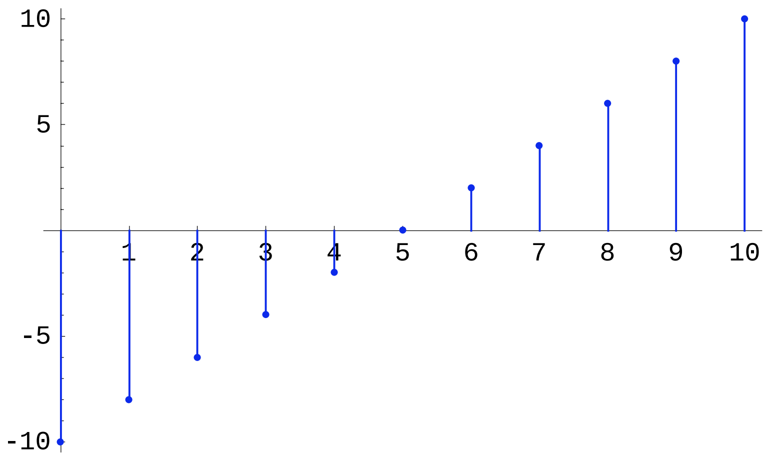
```
UnitRampSequence[6]
```

```
{{0}, {1}, {2}, {3}, {4}, {5}}
```

```
UnitRampSequence[]
```

```
{{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}}
```

```
(-10 + 2 * UnitRampSequence[11]) // SequencePlot;
```



## UnitSineSequence

UnitSineSequence generates a unit sine sequence.

*seq* = UnitSineSequence[*n*, *sineFrequency*, *sinePhase*]

UnitSineSequence[*n*, *sineFrequency*] defaults to UnitSineSequence[*n*, *sineFrequency*, 0]

UnitSineSequence[*n*] defaults to UnitSineSequence[*n*, 1/8, 0]

UnitSineSequence[] defaults to UnitSineSequence[8, 1/8, 0]

*n* is the number of samples.

*sineFrequency* is the digital frequency.

*sinePhase* is the phase.

*seq* is an *n*-by-1 matrix of the form {{a[0]}, {a[1]}, ..., {a[k]}, ..., {a[*n*-1]}}.

a[k] = sin(k*2*Pi**sineFrequency* + *sinePhase*),  k = 0, 1, 2, ..., (*n*-1).

UnitSineSequence returns {} in the case of unexpected arguments.

See also: UnitExponentialSequence, UnitImpulseSequence, UnitNoiseSequence, UnitRampSequence, UnitStepSequence, UnitSymbolicSequence, MultiplexSequence, SequencePlot, DiscreteSystemImplementationProcessing, DiscreteSystemSimulation

### Examples

```
Needs["SchematicSolver`"]
```

Continuous-time sine signal of amplitude 0.15, of phase $\pi/2$, and frequency 1200 Hz, is sampled at 8 kHz. Generate a sequence of 32 samples of the signal.

The required sinusoidal sequence is created with

```
0.15 * UnitSineSequence[32, 1200 / 8000, π / 2]
```

```
{{0.15}, {0.0881678}, {-0.0463525}, {-0.142658}, {-0.121353}, {0}, {0.121353},
 {0.142658}, {0.0463525}, {-0.0881678}, {-0.15}, {-0.0881678}, {0.0463525},
 {0.142658}, {0.121353}, {0}, {-0.121353}, {-0.142658}, {-0.0463525},
 {0.0881678}, {0.15}, {0.0881678}, {-0.0463525}, {-0.142658}, {-0.121353}, {0},
 {0.121353}, {0.142658}, {0.0463525}, {-0.0881678}, {-0.15}, {-0.0881678}}
```

Sinusoidal sequence can be symbolic:

```
a * UnitSineSequence[6, f, ϕ]
```

```
{{a Sin[ϕ]}, {a Sin[2 f π + ϕ]}, {a Sin[4 f π + ϕ]},
 {a Sin[6 f π + ϕ]}, {a Sin[8 f π + ϕ]}, {a Sin[10 f π + ϕ]}}
```

$$a * \text{UnitSineSequence}\left[6, \frac{\omega}{2\pi}, \phi\right]$$

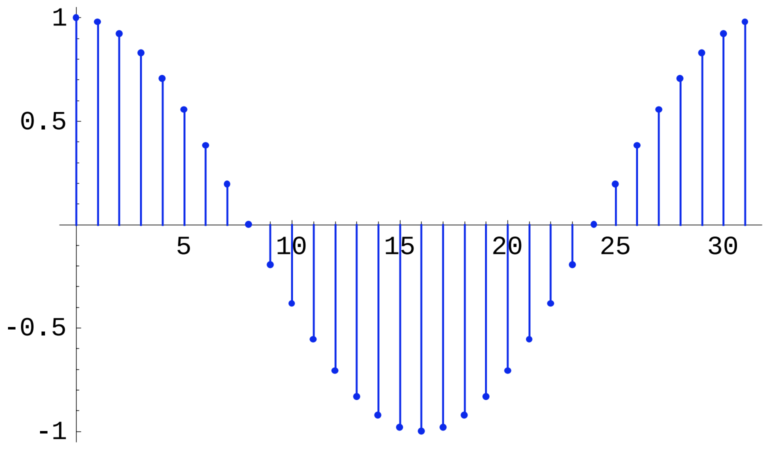
```
{{a Sin[ϕ]}, {a Sin[ϕ + ω]}, {a Sin[ϕ + 2 ω]},
 {a Sin[ϕ + 3 ω]}, {a Sin[ϕ + 4 ω]}, {a Sin[ϕ + 5 ω]}}
```

Here is the default sinusoidal sequence:

```
UnitSineSequence[]
```

$\left\{\{0\}, \left\{\frac{1}{\sqrt{2}}\right\}, \{1\}, \left\{\frac{1}{\sqrt{2}}\right\}, \{0\}, \left\{-\frac{1}{\sqrt{2}}\right\}, \{-1\}, \left\{-\frac{1}{\sqrt{2}}\right\}\right\}$

```
UnitSineSequence[32, 1 / 32, Pi / 2] // SequencePlot;
```



## UnitStepSequence

`UnitStepSequence` generates a unit step sequence.

*seq* = `UnitStepSequence`[*n*, *d*]

`UnitStepSequence`[*n*] defaults to `UnitStepSequence`[*n*, 0]

`UnitStepSequence`[] defaults to `UnitStepSequence`[8, 0]

*n* is the number of samples.

*d* is the delay, i.e., number of leading zero-samples, $d < n$.

*seq* is an *n*-by-1 matrix of the form {{a[0]}, {a[1]}, ..., {a[k]}, ..., {a[*n*-1]}}.

a[k] = 1 for $k \geq d$,  a[k] = 0 otherwise.

`UnitStepSequence` returns {} in the case of unexpected arguments.

See also: `UnitExponentialSequence`, `UnitImpulseSequence`, `UnitNoiseSequence`, `UnitRampSequence`, `UnitSineSequence`, `UnitSymbolicSequence`, `MultiplexSequence`, `SequencePlot`, `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]
```

```
UnitStepSequence[]
```

```
{{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}}
```

```
UnitStepSequence[6, 2]
```

```
{{0}, {0}, {1}, {1}, {1}, {1}}
```

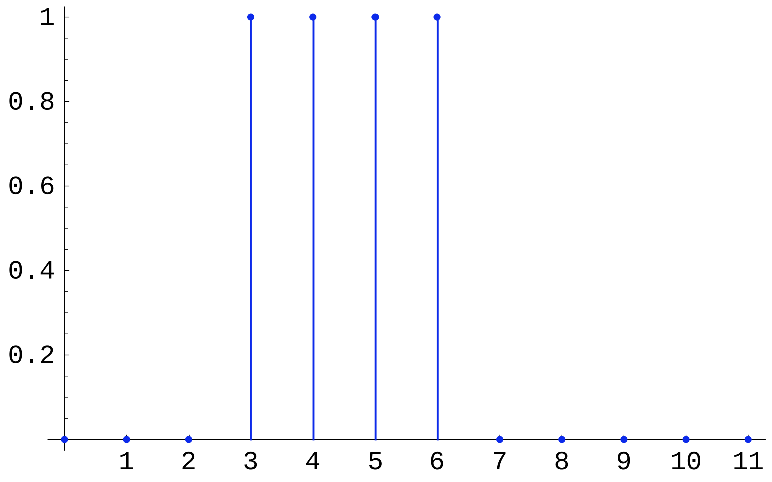
You can use `UnitStepSequence` to generate discrete pulse sequences:

```
numberOfSamples = 12; pulseDelay = 3; pulseWidth = 4;
```

```
pulseSeq = UnitStepSequence[numberOfSamples, pulseDelay] -
  UnitStepSequence[numberOfSamples, pulseDelay + pulseWidth]
```

```
{{0}, {0}, {0}, {1}, {1}, {1}, {1}, {0}, {0}, {0}, {0}, {0}}
```

```
pulseSeq // SequencePlot;
```



## UnitSymbolicSequence

`UnitSymbolicSequence` generates a sequence of symbolic samples.

*seq* = `UnitSymbolicSequence`[*n*, *symb*, *initialIndex*]

`UnitSymbolicSequence`[*n*, *symb*] defaults to `UnitSymbolicSequence`[*n*, *symb*, 1]

*n* is the number of samples.

*symb* is a symbol that is used to represent samples.

*initialIndex* is the initial sample index. It should be a non-negative integer.

*seq* is an *n*-by-1 matrix of symbolic samples.

`UnitSymbolicSequence` returns {} in the case of unexpected arguments.

See also: `UnitExponentialSequence`, `UnitImpulseSequence`, `UnitNoiseSequence`, `UnitRampSequence`, `UnitSineSequence`, `UnitStepSequence`, `MultiplexSequence`, `SequencePlot`, `DiscreteSystemImplementationProcessing`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]
```

```
UnitSymbolicSequence[8, x]
```

```
{{x1}, {x2}, {x3}, {x4}, {x5}, {x6}, {x7}, {x8}}
```

```
UnitSymbolicSequence[4, a, 0]
```

```
{{a0}, {a1}, {a2}, {a3}}
```

You can use `UnitSymbolicSequence` to generate a list of parameters, e.g., which might be the parameters of a system:

```
UnitSymbolicSequence[4, p] // SequenceToList
```

```
{p1, p2, p3, p4}
```

## `UpsampleSequence`

`UpsampleSequence` upsamples a data sequence.

*upSequence* = `UpsampleSequence`[*dataSequence*, *n*]

*dataSequence* is a matrix of data samples.

*n* is the amount of upsampling.

*upSequence* is the upsampled sequence.

`UpsampleSequence` inserts *n*-1 rows after each row of *dataSequence*. The inserted rows are of the form {0, 0, ..., 0}.

`UpsampleSequence` returns {} in the case of unexpected arguments.

See also: `DownsampleSequence`, `MultiplexSequence`, `DemultiplexSequence`, `MultirateDown-sampleSequence`, `SequencePlot`

## `ValidImplementationModuleNameQ`

`ValidImplementationModuleNameQ` tests whether a string is a valid function name.

*flag* = `ValidImplementationModuleNameQ`["*name*"]

*name* is a symbol; it should be enclosed within double quotation marks.

*flag* is `True` if *name* is a valid function name. Otherwise, *flag* is `False`.

A valid function name is composed of letters and digits and should begin with a letter.

See also: `DiscreteSystemImplementation`, `DiscreteSystemImplementationModule`

### Examples

```
Needs["SchematicSolver`"]
```

```
ValidImplementationModuleNameQ["myProc"]
```

```
True
```

Function name should not contain spaces:

```
ValidImplementationModuleNameQ["my Proc"]
```

```
ValidImplementationModuleNameQ::invname :
 "my Proc" is not a well-formed module name.
```

```
False
```

Function name should not begin with a number:

```
ValidImplementationModuleNameQ["1stProc"]
```

```
ValidImplementationModuleNameQ::invname :
 "1stProc" is not a well-formed module name.
```

```
False
```

## $VersionSchematicSolverSchematicImplementation

`$VersionSchematicSolverSchematicImplementation` is a variable that contains information about the implementation package version and release date.

```
Needs["SchematicSolver`"]

$VersionSchematicSolverSchematicImplementation

2.0 (February 04, 2004. 21:30)
```

## 14.8. Album of Schematics

### General Format of Album Schematics

*SchematicSolver*`s Album is a collection of functions that generate schematic specifications of systems frequently found in practice. These functions can help you automate creation of complex schematics. General format for calling the functions is

{*schematicSpec*, *inpCoords*, *outCoords*} = *SchematicAlbumFunction*[*params*, {*x0*, *y0*}, *options*]

*SchematicAlbumFunction* is the name of the function, e.g., `DirectFormFIRFilterSchematic`.

*params* is a list of parameters.

{*x0*, *y0*} are the coordinates of the schematic offset. Typically, {*x0*, *y0*} are the coordinates of a system input.

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

*inpCoords* is a list of input coordinates.

*outCoords* is a list of output coordinates.

*options* are the function options, e.g., `DelayElementValue`.

As a rule, *SchematicAlbumFunction* generates a schematic specification that does not contain any Input elements and Output elements.

## DelayElementValue

DelayElementValue is an option for DirectFormFIRFilterSchematic, DoubleDelayDirectFormFIR-FilterSchematic, HalfbandDirectFormFIRFilterSchematic, HilbertTransformerDirectForm-FIRSchematic, TransposedDirectForm2IIRBiquadSchematic, and TransposedDirectForm2IIR-FilterSchematic, that sets the value of the Delay element.

DelayElementValue → *d*

*d* is the amount of delay; it can be an integer or a symbol.

DelayElementValue→1 is default.

You can reset the default using SetOptions[*function*, *option*→*value*]. For example, SetOptions[DirectFormFIR-FilterSchematic, DelayElementValue→2].

## DirectFormFIRFilterSchematic

`DirectFormFIRFilterSchematic` creates schematic specification for Direct Form FIR filter of an arbitrary order and arbitrary parameters.

---

{*filterSpec*, *inpCoords*, *outCoords*} = `DirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}, *options*]

`DirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}] defaults to `DirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}, `DelayElementValue`→1]

`DirectFormFIRFilterSchematic`[*params*] defaults to `DirectFormFIRFilterSchematic`[*params*, {0, 0}]

`DirectFormFIRFilterSchematic`[] defaults to `DirectFormFIRFilterSchematic`[{1, –1}, {0, 0}]

---

*params* is a list of one or more parameters of the form {a[0], a[1], a[2],..., a[K]} where K is the filter order.

{*x0*, *y0*} are numeric coordinates of the filter input.

*filterSpec* is a schematic specification that represents the filter; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This filter has one input so *inpCoords* is of the form {{*xIn*, *yIn*}}.

*outCoords* is a list of output coordinates. This filter has one output so *outCoords* is of the form {{*xOut*, *yOut*}}.

`DelayElementValue`→*d* sets the Delay element value to *d*.

`DirectFormFIRFilterSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

### Examples

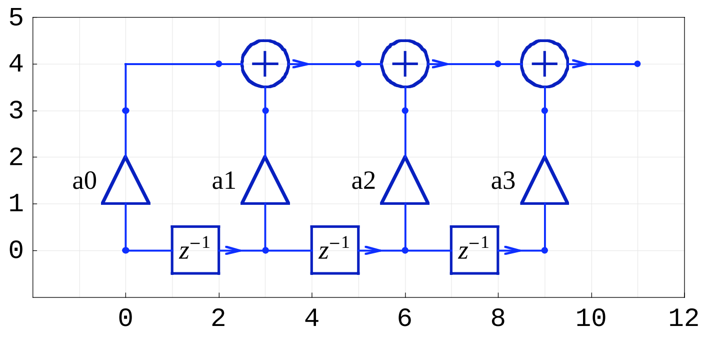
```
Needs["SchematicSolver`"]
```

```
{schematicSpecification, inputCoordinates, outputCoordinates} =
  DirectFormFIRFilterSchematic[{a0, a1, a2, a3}];
```

```
schematicSpecification
```

```
{{Multiplier, {{0, 0}, {0, 3}}, a0}, {Line, {{0, 3}, {0, 4}, {2, 4}}},
 {Delay, {{0, 0}, {3, 0}}, 1}, {Multiplier, {{3, 0}, {3, 3}}, a1},
 {Adder, {{2, 4}, {3, 3}, {5, 4}, {3, 5}}, {1, 1, 2, 0}},
 {Delay, {{3, 0}, {6, 0}}, 1}, {Multiplier, {{6, 0}, {6, 3}}, a2},
 {Adder, {{5, 4}, {6, 3}, {8, 4}, {6, 5}}, {1, 1, 2, 0}},
 {Delay, {{6, 0}, {9, 0}}, 1}, {Multiplier, {{9, 0}, {9, 3}}, a3},
 {Adder, {{8, 4}, {9, 3}, {11, 4}, {9, 5}}, {1, 1, 2, 0}}}
```

```
ShowSchematic[schematicSpecification, PlotRange → {{-2, 12}, {-1, 5}}];
```



```
inputCoordinates
```

```
{{0, 0}}
```

```
outputCoordinates
```

```
{{11, 4}}
```

## DoubleDelayDirectFormFIRFilterSchematic

`DoubleDelayDirectFormFIRFilterSchematic` creates schematic specification for Double Delay Direct Form FIR filter of an arbitrary order and arbitrary parameters.

{*filterSpec*, *inpCoords*, *outCoords*} = `DoubleDelayDirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}, *options*]

`DoubleDelayDirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}] defaults to `DoubleDelayDirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}, `DelayElementValue`→1]

`DoubleDelayDirectFormFIRFilterSchematic`[*params*] defaults to `DoubleDelayDirectFormFIRFilterSchematic`[*params*, {0, 0}]

`DoubleDelayDirectFormFIRFilterSchematic`[] defaults to `DoubleDelayDirectFormFIRFilterSchematic`[{1, –1}, {0, 0}]

*params* is a list of one or more parameters of the form {a[0], a[1], a[2],..., a[K]} where 2K is the filter order.

{*x0*, *y0*} are numeric coordinates of the filter input.

*filterSpec* is a schematic specification that represents the filter; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This filter has one input so *inpCoords* is of the form {{*xIn*, *yIn*}}.

*outCoords* is a list of output coordinates. This filter has one output so *outCoords* is of the form {{*xOut*, *yOut*}}.

`DelayElementValue`→*d* sets the Delay element value to *d*.

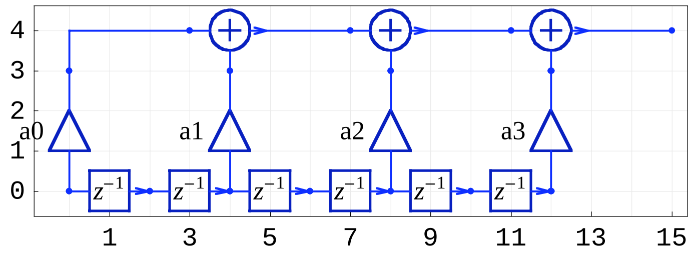
`DoubleDelayDirectFormFIRFilterSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]

{schematicSpecification, inputCoordinates, outputCoordinates} =
  DoubleDelayDirectFormFIRFilterSchematic[{a0, a1, a2, a3}];

schematicSpecification

{{Multiplier, {{0, 0}, {0, 3}}, a0},
 {Line, {{0, 3}, {0, 4}, {3, 4}}}, {Delay, {{0, 0}, {2, 0}}, 1},
 {Delay, {{2, 0}, {4, 0}}, 1}, {Multiplier, {{4, 0}, {4, 3}}, a1},
 {Adder, {{3, 4}, {4, 3}, {7, 4}, {2, 5}}, {1, 1, 2, 0}}, {Delay, {{4, 0}, {6, 0}}, 1},
 {Delay, {{6, 0}, {8, 0}}, 1}, {Multiplier, {{8, 0}, {8, 3}}, a2},
 {Adder, {{7, 4}, {8, 3}, {11, 4}, {6, 5}}, {1, 1, 2, 0}}, {Delay, {{8, 0}, {10, 0}}, 1},
 {Delay, {{10, 0}, {12, 0}}, 1}, {Multiplier, {{12, 0}, {12, 3}}, a3},
 {Adder, {{11, 4}, {12, 3}, {15, 4}, {10, 5}}, {1, 1, 2, 0}}}
```

```
ShowSchematic[schematicSpecification];
```



```
inputCoordinates
```

```
{{0, 0}}
```

```
outputCoordinates
```

```
{{15, 4}}
```

## HalfbandDirectFormFIRFilterSchematic

`HalfbandDirectFormFIRFilterSchematic` creates schematic specification for Halfband Direct Form FIR filter of an arbitrary order and arbitrary parameters.

---

{*filterSpec*, *inpCoords*, *outCoords*} = `HalfbandDirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}, *options*]

`HalfbandDirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}] defaults to `HalfbandDirectFormFIRFilterSchematic`[*params*, {*x0*, *y0*}, `DelayElementValue`→1]

`HalfbandDirectFormFIRFilterSchematic`[*params*] defaults to `HalfbandDirectFormFIRFilterSchematic`[*params*, {0, 0}]

`HalfbandDirectFormFIRFilterSchematic`[] defaults to `HalfbandDirectFormFIRFilterSchematic`[{1, –1}, {0, 0}]

---

*params* is a list of one or more parameters of the form {a[0], a[1], a[2], ..., a[K]} where 2K is the filter order.

{*x0*, *y0*} are numeric coordinates of the filter input.

*filterSpec* is a schematic specification that represents the filter; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This filter has one input so *inpCoords* is of the form {{*xIn*, *yIn*}}.

*outCoords* is a list of output coordinates. This filter has two outputs so *outCoords* is of the form {{*x1Out*, *y1Out*}, {*x2Out*, *y2Out*}}.

`DelayElementValue`→*d* sets the Delay element value to *d*.

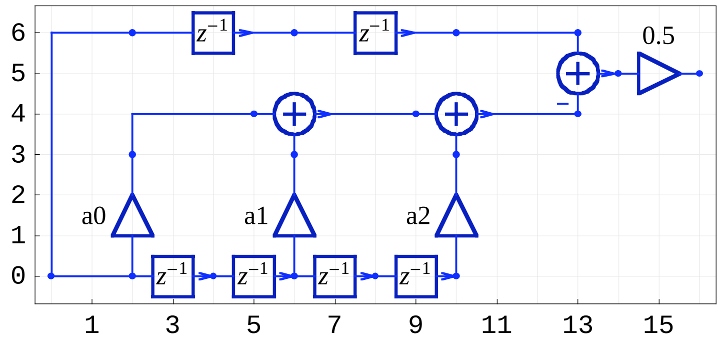
`HalfbandDirectFormFIRFilterSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]

{schematicSpecification, inputCoordinates, outputCoordinates} =
  HalfbandDirectFormFIRFilterSchematic[{a0, a1, a2}];

schematicSpecification

{{Line, {{0, 0}, {2, 0}}}, {Line, {{0, 0}, {0, 6}, {2, 6}}},
 {Multiplier, {{2, 0}, {2, 3}}, a0}, {Line, {{2, 3}, {2, 4}, {5, 4}}},
 {Delay, {{2, 0}, {4, 0}}, 1}, {Delay, {{4, 0}, {6, 0}}, 1},
 {Delay, {{2, 6}, {6, 6}}, 1}, {Multiplier, {{6, 0}, {6, 3}}, a1},
 {Adder, {{5, 4}, {6, 3}, {9, 4}, {6, 5}}, {1, 1, 2, 0}},
 {Delay, {{6, 0}, {8, 0}}, 1}, {Delay, {{8, 0}, {10, 0}}, 1},
 {Delay, {{6, 6}, {10, 6}}, 1}, {Multiplier, {{10, 0}, {10, 3}}, a2},
 {Adder, {{9, 4}, {10, 3}, {13, 4}, {10, 5}}, {1, 1, 2, 0}}, {Line, {{13, 6}, {10, 6}}},
 {Adder, {{12, 5}, {13, 4}, {14, 5}, {13, 6}}, {0, -1, 2, 1}},
 {Multiplier, {{14, 5}, {16, 5}}, 0.5}}
```

```
ShowSchematic[schematicSpecification];
```



```
inputCoordinates
```

```
{{0, 0}}
```

```
outputCoordinates
```

```
{{16, 5}}
```

## HighSpeedIIR3FIRHalfbandFilterSchematic

`HighSpeedIIR3FIRHalfbandFilterSchematic` creates schematic specification for

High Speed 3rd-order IIR – FIR Halfband Filter of an arbitrary order and arbitrary parameters.

{*filterSpec*, *inpCoords*, *outCoords*} =

`HighSpeedIIR3FIRHalfbandFilterSchematic`[*params*, {*x0*,*y0*}]

`HighSpeedIIR3FIRHalfbandFilterSchematic`[*params*] defaults to `HighSpeedIIR3FIRHalfbandFilterSchematic`[*params*, {0,0}]

`HighSpeedIIR3FIRHalfbandFilterSchematic`[] defaults to `HighSpeedIIR3FIRHalfbandFilterSchematic`[{1/2,1/2,1}, {0,0}]

*params* is a list of three or more parameters of the form {*b*, *k0*, *k1*, *k2*, ...} where *b* is the coefficient of the IIR half-band filter, *k0* is the normalization coefficient, and *k1*, *k2*, ..., are the coefficients of the FIR filter.

{*x0*,*y0*} are numeric coordinates of the filter input.

*filterSpec* is a schematic specification that represents the filter; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This filter has one input so *inpCoords* is of the form {{*xIn*,*yIn*}}.

*outCoords* is a list of output coordinates. This filter has two outputs so *outCoords* is of the form {{*x1Out*,*y1Out*}, {*x2Out*,*y2Out*}}.

`HighSpeedIIR3FIRHalfbandFilterSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

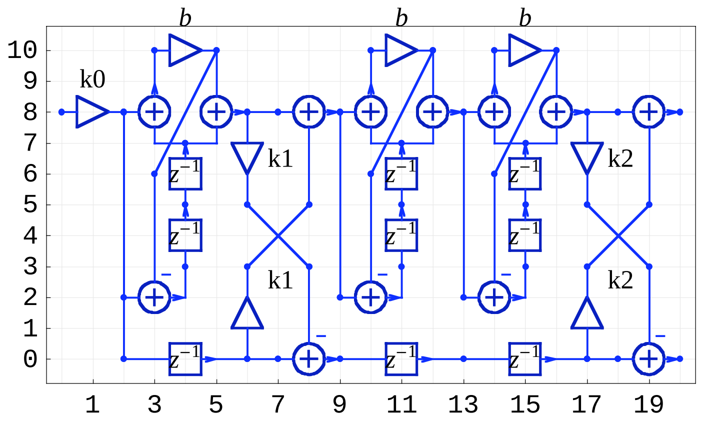
### Examples

```
Needs["SchematicSolver`"]

{schematicSpecification, inputCoordinates, outputCoordinates} =
  HighSpeedIIR3FIRHalfbandFilterSchematic[{b, k0, k1, k2}];
```

**schematicSpecification**

```
{{Multiplier, {{6, 0}, {6, 3}}, k1, , TextOffset → {-1, 1}},
 {Multiplier, {{6, 8}, {6, 5}}, k1},
 {Line, {{2, 8}, {2, 0}}}, {Multiplier, {{0, 8}, {2, 8}}, k0},
 {Adder, {{7, 8}, {8, 5}, {9, 8}, {8, 9}}, {1, 1, 2, 0}},
 {Adder, {{7, 0}, {8, -1}, {9, 0}, {8, 3}}, {1, 0, 2, -1}},
 {Line, {{6, 8}, {7, 8}}}, {Line, {{6, 0}, {7, 0}}}, {Line, {{6, 3}, {8, 5}}},
 {Line, {{6, 5}, {8, 3}}}, {Adder, {{2, 8}, {4, 7}, {4, 8}, {3, 10}}, {1, 1, 0, 2}},
 {Adder, {{4, 8}, {4, 7}, {6, 8}, {5, 10}}, {0, 1, 2, 1}},
 {Multiplier, {{3, 10}, {5, 10}}, b}, {Line, {{3, 6}, {5, 10}}},
 {Delay, {{4, 5}, {4, 7}}, 1}, {Delay, {{4, 3}, {4, 5}}, 1},
 {Adder, {{2, 2}, {4, 0}, {4, 3}, {3, 6}}, {1, 0, 2, -1}},
 {Line, {{2, 8}, {2, 2}}}, {Delay, {{2, 0}, {6, 0}}, 1},
 {Multiplier, {{17, 0}, {17, 3}}, k2, , TextOffset → {-1, 1}},
 {Multiplier, {{17, 8}, {17, 5}}, k2, },
 {Adder, {{18, 8}, {19, 5}, {20, 8}, {19, 9}}, {1, 1, 2, 0}},
 {Adder, {{18, 0}, {19, -1}, {20, 0}, {19, 3}}, {1, 0, 2, -1}},
 {Line, {{17, 8}, {18, 8}}}, {Line, {{17, 0}, {18, 0}}}, {Line, {{17, 3}, {19, 5}}},
 {Line, {{17, 5}, {19, 3}}}, {Adder, {{9, 8}, {11, 7}, {11, 8}, {10, 10}}, {1, 1, 0, 2}},
 {Adder, {{11, 8}, {11, 7}, {13, 8}, {12, 10}}, {0, 1, 2, 1}},
 {Multiplier, {{10, 10}, {12, 10}}, b}, {Line, {{10, 6}, {12, 10}}},
 {Delay, {{11, 5}, {11, 7}}, 1}, {Delay, {{11, 3}, {11, 5}}, 1},
 {Adder, {{9, 2}, {11, 0}, {11, 3}, {10, 6}}, {1, 0, 2, -1}}, {Line, {{9, 8}, {9, 2}}},
 {Adder, {{13, 8}, {15, 7}, {15, 8}, {14, 10}}, {1, 1, 0, 2}},
 {Adder, {{15, 8}, {15, 7}, {17, 8}, {16, 10}}, {0, 1, 2, 1}},
 {Multiplier, {{14, 10}, {16, 10}}, b}, {Line, {{14, 6}, {16, 10}}},
 {Delay, {{15, 5}, {15, 7}}, 1}, {Delay, {{15, 3}, {15, 5}}, 1},
 {Adder, {{13, 2}, {15, 0}, {15, 3}, {14, 6}}, {1, 0, 2, -1}}, {Line, {{13, 8}, {13, 2}}},
 {Delay, {{9, 0}, {13, 0}}, 1}, {Delay, {{13, 0}, {17, 0}}, 1}}
```

**ShowSchematic**[**schematicSpecification**];



**inputCoordinates**

```
{{0, 8}}
```

**outputCoordinates**

```
{{20, 8}, {20, 0}}
```

## HilbertTransformerDirectFormFIRSchematic

`HilbertTransformerDirectFormFIRSchematic` creates schematic specification for the Hilbert Transformer with Direct Form FIR filter of an arbitrary order and arbitrary parameters.

{*schematicSpec*, *inpCoords*, *outCoords*} = `HilbertTransformerDirectFormFIRSchematic`[*params*, {*x0*, *y0*}, *options*]

`HilbertTransformerDirectFormFIRSchematic`[*params*, {*x0*, *y0*}] defaults to `HilbertTransformerDirectFormFIRSchematic`[*params*, {*x0*, *y0*}, `DelayElementValue`→1]

`HilbertTransformerDirectFormFIRSchematic`[*params*] defaults to `HilbertTransformerDirectFormFIRSchematic`[*params*, {0, 0}]

`HilbertTransformerDirectFormFIRSchematic`[] defaults to `HilbertTransformerDirectFormFIRSchematic`[{1, –1}, {0, 0}]

*params* is a list of one or more parameters of the form {a[0], a[1], a[2], ...}.

{*x0*, *y0*} are numeric coordinates of the system input.

*schematicSpec* is a schematic specification that represents the system; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This system has one input so *inpCoords* is of the form {{*xIn*, *yIn*}}.

*outCoords* is a list of output coordinates. This system has two outputs so *outCoords* is of the form {{*x1Out*, *y1Out*}, {*x2Out*, *y2Out*}}.

`DelayElementValue`→*d* sets the Delay element value to *d*.

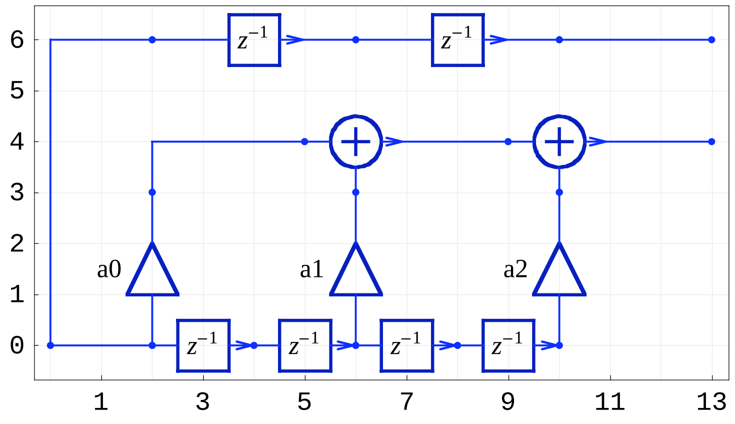
`HilbertTransformerDirectFormFIRSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

## Examples

```
Needs["SchematicSolver`"]

{schematicSpecification, inputCoordinates, outputCoordinates} =
  HilbertTransformerDirectFormFIRSchematic[{a0, a1, a2}];

schematicSpecification

{{Line, {{0, 0}, {2, 0}}}, {Line, {{0, 0}, {0, 6}, {2, 6}}},
 {Multiplier, {{2, 0}, {2, 3}}, a0}, {Line, {{2, 3}, {2, 4}, {5, 4}}},
 {Delay, {{2, 0}, {4, 0}}, 1}, {Delay, {{4, 0}, {6, 0}}, 1},
 {Delay, {{2, 6}, {6, 6}}, 1}, {Multiplier, {{6, 0}, {6, 3}}, a1},
 {Adder, {{5, 4}, {6, 3}, {9, 4}, {6, 5}}, {1, 1, 2, 0}},
 {Delay, {{6, 0}, {8, 0}}, 1}, {Delay, {{8, 0}, {10, 0}}, 1},
 {Delay, {{6, 6}, {10, 6}}, 1}, {Multiplier, {{10, 0}, {10, 3}}, a2},
 {Adder, {{9, 4}, {10, 3}, {13, 4}, {10, 5}}, {1, 1, 2, 0}}, {Line, {{13, 6}, {10, 6}}}}
```

```
ShowSchematic[schematicSpecification];
```



```
inputCoordinates
```

```
{{0, 0}}
```

```
outputCoordinates
```

```
{{13, 4}, {13, 6}}
```

## TestDiscreteLinearSISOAlbumSchematic

`TestDiscreteLinearSISOAlbumSchematic` tests a discrete linear SISO album schematic.

`TestDiscreteLinearSISOAlbumSchematic`[*albumObject*, *options*]

*albumObject* is a list returned by a function that creates schematic specification, such as `DirectFormFIRFilter-Schematic`.

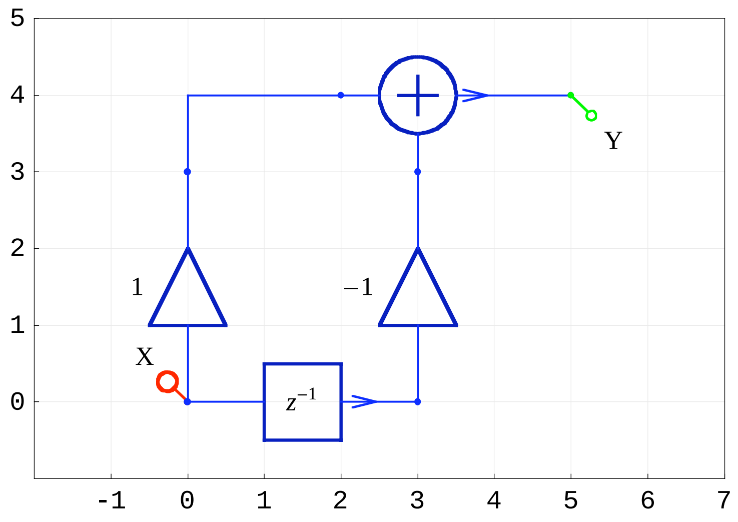
Any `ShowSchematic` option can be given to `TestDiscreteLinearSISOAlbumSchematic`.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementa-tion`, `DiscreteSystemSimulation`

### Examples

```
Needs["SchematicSolver`"]

TestDiscreteLinearSISOAlbumSchematic[
 DirectFormFIRFilterSchematic[], PlotRange → {{-2, 7}, {-1, 5}}]
```

```
{{{Multiplier, {{0, 0}, {0, 3}}, 1}, {Line, {{0, 3}, {0, 4}, {2, 4}}},
  {Delay, {{0, 0}, {3, 0}}, 1}, {Multiplier, {{3, 0}, {3, 3}}, -1},
  {Adder, {{2, 4}, {3, 3}, {5, 4}, {3, 5}}, {1, 1, 2, 0}}}, {{0, 0}}, {{5, 4}}}
```



$\{\{\{-\frac{1-z}{z}\}\}, \{Y[\{0, 0\}]\}, \{Y[\{5, 4\}]\}\}$

$1 - z^{-1}$

The above example illustrates the test that is performed by `TestDiscreteLinearSISOAlbumSchematic`. First, the schematic specification returned by the album function `DirectFormFIRFilterSchematic` is displayed. Next, `TestDiscreteLinearSISOAlbumSchematic` adds an Input element (drawn in red) and an Output element (drawn in green), and computes the transfer function. Finally, the transfer function is displayed in the traditional form in terms of $z^{-1}$.

## TransposedDirectForm2IIRBiquadSchematic

`TransposedDirectForm2IIRBiquadSchematic` creates schematic specification for Transposed Direct Form 2 IIR Biquad of arbitrary parameters.

{*filterSpec*, *inpCoords*, *outCoords*} = `TransposedDirectForm2IIRBiquadSchematic`[{*num*, *den*}, {*x0*, *y0*}, *options*]

`TransposedDirectForm2IIRBiquadSchematic`[{*num*, *den*}, {*x0*, *y0*}] defaults to `TransposedDirectForm2IIRBiquadSchematic`[{*num*, *den*}, {*x0*, *y0*}, `DelayElementValue`→1]

`TransposedDirectForm2IIRBiquadSchematic`[{*num*, *den*}] defaults to `TransposedDirectForm2IIRBiquadSchematic`[{*num*, *den*}, {0, 0}]

`TransposedDirectForm2IIRBiquadSchematic`[] defaults to `TransposedDirectForm2IIRBiquadSchematic`[{{1, 1}, {0.9}}, {0, 0}]

*num* is a list of numerator parameters {b0, b1, b2}.

*den* is a list of denominator parameters {a1, a2}.

{*x0*, *y0*} are numeric coordinates of the filter input.

*filterSpec* is a schematic specification that represents the filter; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This filter has one input so *inpCoords* is of the form {{*xIn*, *yIn*}}.

*outCoords* is a list of output coordinates. This filter has one output so *outCoords* is of the form {{*xOut*, *yOut*}}.

`DelayElementValue`→*d* sets the Delay element value to *d*.

`TransposedDirectForm2IIRBiquadSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

### Examples

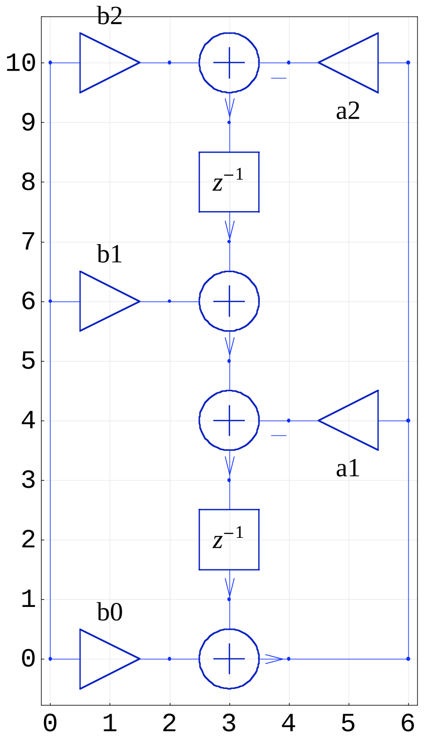
```
Needs["SchematicSolver`"]
```

```
{schematicSpecification, inputCoordinates, outputCoordinates} =
  TransposedDirectForm2IIRBiquadSchematic[{{b0, b1, b2}, {a1, a2}}];
```

```
schematicSpecification
```

```
{{Line, {{0, 0}, {0, 6}}}, {Line, {{0, 6}, {0, 10}}},
 {Line, {{6, 0}, {4, 0}}}, {Line, {{6, 0}, {6, 4}}}, {Line, {{6, 4}, {6, 10}}},
 {Adder, {{2, 0}, {3, -1}, {4, 0}, {3, 1}}, {1, 0, 2, 1}},
 {Adder, {{2, 4}, {3, 3}, {4, 4}, {3, 5}}, {0, 2, -1, 1}},
 {Adder, {{2, 6}, {3, 5}, {4, 6}, {3, 7}}, {1, 2, 0, 1}},
 {Adder, {{2, 10}, {3, 9}, {4, 10}, {3, 11}}, {1, 2, -1, 0}}, {Delay, {{3, 3}, {3, 1}}, 1},
 {Delay, {{3, 9}, {3, 7}}, 1}, {Multiplier, {{0, 0}, {2, 0}}, b0},
 {Multiplier, {{0, 6}, {2, 6}}, b1}, {Multiplier, {{0, 10}, {2, 10}}, b2},
 {Multiplier, {{6, 4}, {4, 4}}, a1}, {Multiplier, {{6, 10}, {4, 10}}, a2}}
```

```
ShowSchematic[schematicSpecification];
```



```
inputCoordinates
```

```
{{0, 0}}
```

```
outputCoordinates
```

```
{{6, 0}}
```

## TransposedDirectForm2IIRFilterSchematic

`TransposedDirectForm2IIRFilterSchematic` creates schematic specification for Transposed Direct Form 2 IIR filter of an arbitrary order and arbitrary parameters.

{*filterSpec*, *inpCoords*, *outCoords*} = `TransposedDirectForm2IIRFilterSchematic`[{*num*, *den*}, {*x0*, *y0*}, *options*]

`TransposedDirectForm2IIRFilterSchematic`[{*num*, *den*}, {*x0*, *y0*}] defaults to `TransposedDirectForm2IIRFilterSchematic`[{*num*, *den*}, {*x0*, *y0*}, `DelayElementValue`→1]

`TransposedDirectForm2IIRFilterSchematic`[{*num*, *den*}] defaults to `TransposedDirectForm2IIRFilterSchematic`[{*num*, *den*}, {0, 0}]

`TransposedDirectForm2IIRFilterSchematic`[] defaults to `TransposedDirectForm2IIRFilterSchematic`[{{1, 1}, {0. 9}}, {0, 0}]

*num* is a list of numerator parameters {b[0], b[1], b[2], ..., b[K]}.

*den* is a list of denominator parameters {a[1], a[2], ..., a[K]} where K is the filter order.

{*x0*, *y0*} are numeric coordinates of the filter input.

*filterSpec* is a schematic specification that represents the filter; it is a list of element specifications.

*inpCoords* is a list of input coordinates. This filter has one input so *inpCoords* is of the form {{*xIn*, *yIn*}}.

*outCoords* is a list of output coordinates. This filter has one output so *outCoords* is of the form {{*xOut*, *yOut*}}.

`DelayElementValue`→*d* sets the Delay element value to *d*.

`TransposedDirectForm2IIRFilterSchematic` returns {} in the case of unexpected arguments.

See also: `ShowSchematic`, `DiscreteSystemTransferFunction`, `DiscreteSystemImplementation`, `DiscreteSystemSimulation`

### Examples

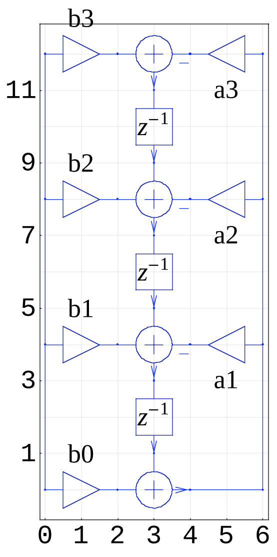
```
Needs["SchematicSolver`"]
```

```
{schematicSpecification, inputCoordinates, outputCoordinates} =
  TransposedDirectForm2IIRFilterSchematic[{{b0, b1, b2, b3}, {a1, a2, a3}}];
```

```
schematicSpecification
```

```
{{Line, {{6, 0}, {4, 0}}}, {Adder, {{2, 0}, {3, -1}, {4, 0}, {3, 1}}, {1, 0, 2, 1}},
 {Multiplier, {{0, 0}, {2, 0}}, b0}, {Line, {{0, 0}, {0, 4}}},
 {Line, {{6, 0}, {6, 4}}}, {Adder, {{2, 4}, {3, 3}, {4, 4}, {3, 5}}, {1, 2, -1, 1}},
 {Delay, {{3, 3}, {3, 1}}, 1}, {Multiplier, {{0, 4}, {2, 4}}, b1},
 {Multiplier, {{6, 4}, {4, 4}}, a1}, {Line, {{0, 4}, {0, 8}}},
 {Line, {{6, 4}, {6, 8}}}, {Adder, {{2, 8}, {3, 7}, {4, 8}, {3, 9}}, {1, 2, -1, 1}},
 {Delay, {{3, 7}, {3, 5}}, 1}, {Multiplier, {{0, 8}, {2, 8}}, b2},
 {Multiplier, {{6, 8}, {4, 8}}, a2}, {Line, {{0, 8}, {0, 12}}},
 {Line, {{6, 8}, {6, 12}}}, {Adder, {{2, 12}, {3, 11}, {4, 12}, {3, 13}}, {1, 2, -1, 0}},
 {Delay, {{3, 11}, {3, 9}}, 1}, {Multiplier, {{0, 12}, {2, 12}}, b3},
 {Multiplier, {{6, 12}, {4, 12}}, a3}}
```

```
ShowSchematic[schematicSpecification];
```



```
inputCoordinates
```

```
{{0, 0}}
```

```
outputCoordinates
```

```
{{6, 0}}
```

## $VersionSchematicSolverSchematicAlbum

`$VersionSchematicSolverSchematicAlbum` is a variable that contains information about the album package version and release date.

```
Needs["SchematicSolver`"]

$VersionSchematicSolverSchematicAlbum

2.0 (January 29, 2004. 20:50)
```

## 14.9. Figures in *SchematicSolver*

### Introduction

*SchematicSolver* has many figures that illustrate solving and implementing systems. These figures are stored in variables, as schematic specifications, and are displayed with the `ShowSchematic` function.

`ShowSchematic`[*schematicFigure*, *options*]

*schematicFigure* is the name of the variable, e.g., `SchematicSolverFigureHilbertTransformerIdeal`.

*options* are the `ShowSchematic` options.

Some variables, such as `DrawElementPlotStyleDefault`, store typical element plot styles.

```
Needs["SchematicSolver`"]
```

We specify some options to better present the figures:

```
SetOptions[InputNotebook[],
  ImageSize → {350, 250},
  ImageMargins → {{0, 0}, {0, 0}}];

SetOptions[ShowSchematic,
  ElementScale → 1, FontSize → Automatic,
  Frame → True, GridLines → Automatic,
  PlotRange → All];

SetOptions[DrawElement,
  ElementSize → {1, 1},
  PlotStyle → {
    {RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
    {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
  ShowArrowTail → True,
  ShowNodes → False, TextOffset → Automatic,
  TextStyle → {FontFamily → Times, FontSize → 10}];
```

## DrawElementPlotStyleDefault

DrawElementPlotStyleDefault is the default option value for PlotStyle in DrawElement.

```
DrawElementPlotStyleDefault

{{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
 {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}}
```

## DrawElementPlotStyleLight

DrawElementPlotStyleLight is an option value for PlotStyle in DrawElement. This value is suitable for large schematics.

```
DrawElementPlotStyleLight

{{RGBColor[0, 0, 0.7], Thickness[0.001], PointSize[0.006]},
 {RGBColor[0, 0, 1], Thickness[0.001], PointSize[0.005]}}
```

## DrawElementPlotStyleMedium

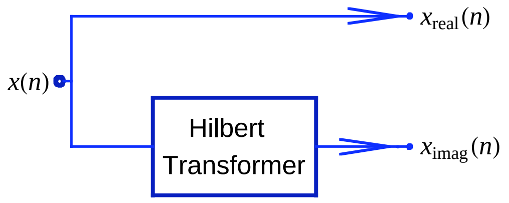
DrawElementPlotStyleMedium is an option value for PlotStyle in DrawElement.

```
DrawElementPlotStyleMedium

{{RGBColor[0, 0, 0.7], Thickness[0.0025], PointSize[0.008]},
 {RGBColor[0, 0, 1], Thickness[0.0015], PointSize[0.007]}}
```

## SchematicSolverFigureHilbertTransformerIdeal

`SchematicSolverFigureHilbertTransformerIdeal` is a schematic specification that illustrates a system with the ideal Hilbert Transformer.

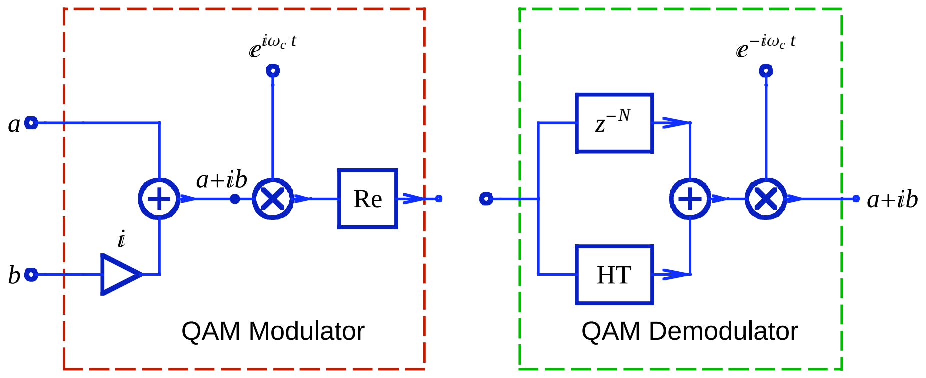
```
Needs["SchematicSolver`"]

ShowSchematic[SchematicSolverFigureHilbertTransformerIdeal,
  Frame → False, GridLines → None, PlotRange → {{-3, 25}, All}];
```



## SchematicSolverFigureHilbertTransformerQAM

`SchematicSolverFigureHilbertTransformerQAM` is a schematic specification that illustrates the QAM system with the Hilbert Transformer.

```
Needs["SchematicSolver`"]

ShowSchematic[SchematicSolverFigureHilbertTransformerQAM,
  Frame → False, GridLines → None];
```

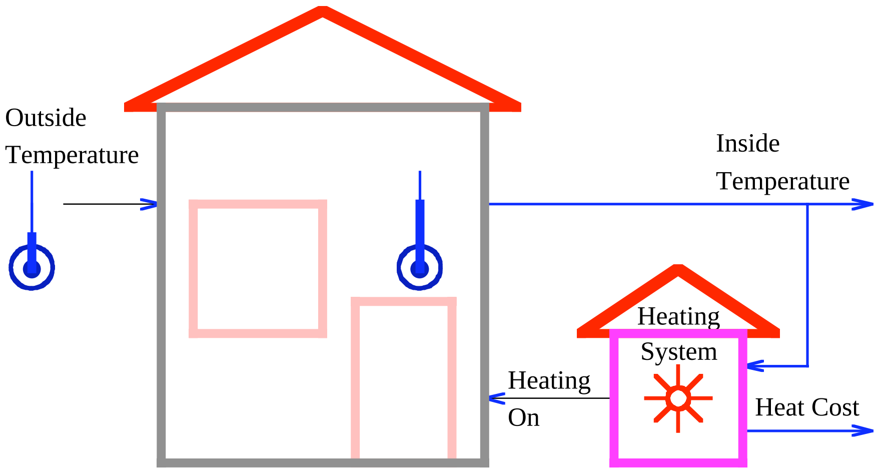


## SchematicSolverFigureImplementationExamplesHouseHeating

`SchematicSolverFigureImplementationExamplesHouseHeating` is a schematic specification that illustrates a house heating system.

```
Needs["SchematicSolver`"]
```

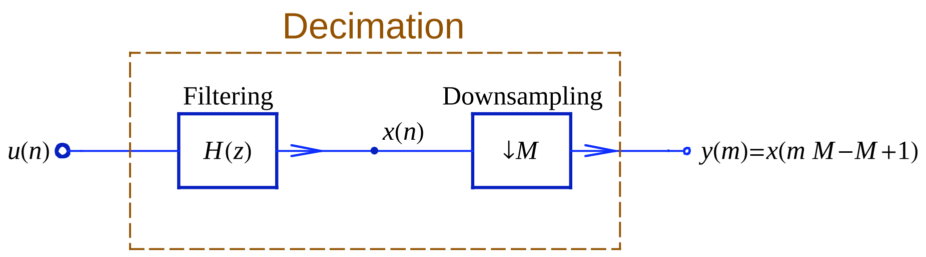
```
ShowSchematic[SchematicSolverFigureImplementationExamplesHouseHeating,
 GridLines → None, Frame → False]
```



## SchematicSolverFigureMultirateDecimation

`SchematicSolverFigureMultirateDecimation` is a schematic specification that illustrates the decimation system.

```
Needs["SchematicSolver`"]

ShowSchematic[SchematicSolverFigureMultirateDecimation,
  GridLines → None, Frame → False];
```

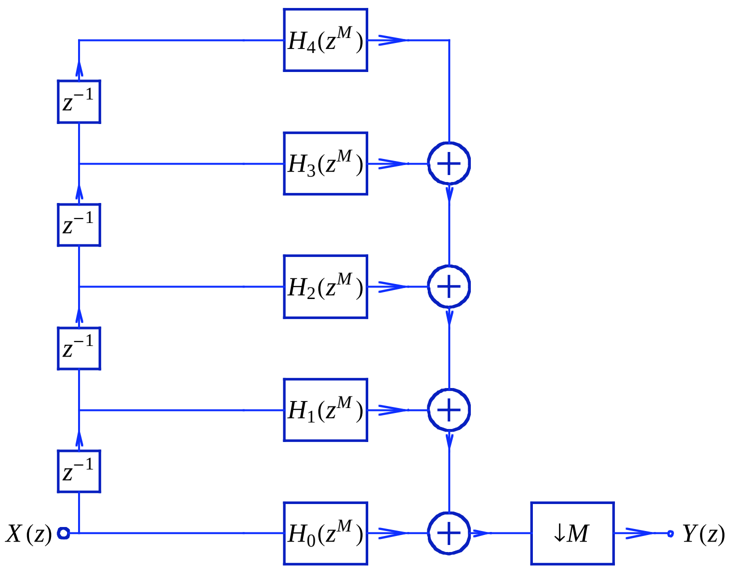


## SchematicSolverFigureMultirateDownsamplingClassic

`SchematicSolverFigureMultirateDownsamplingClassic` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]
```

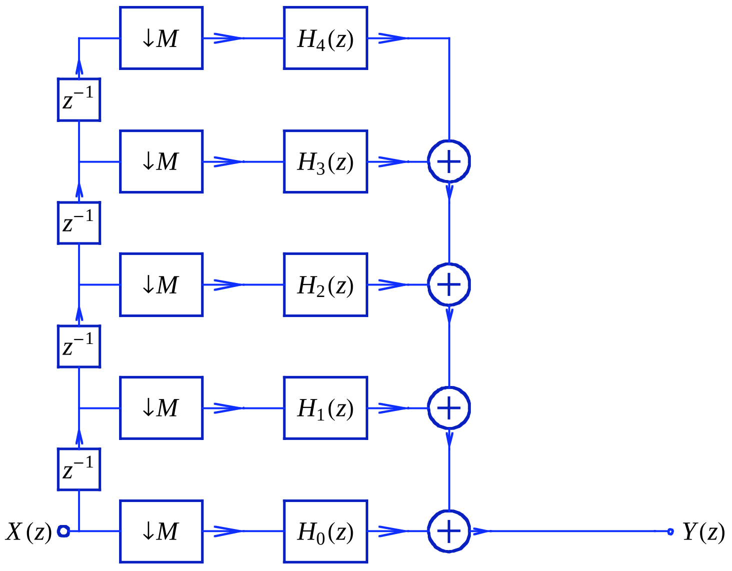
```
SchematicSolverFigureMultirateDownsamplingClassic;
 ShowSchematic[%, GridLines → None, Frame → False];
```



## SchematicSolverFigureMultirateDownsamplingEfficient

`SchematicSolverFigureMultirateDownsamplingEfficient` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]

SchematicSolverFigureMultirateDownsamplingEfficient;
 ShowSchematic[%, GridLines → None, Frame → False];
```



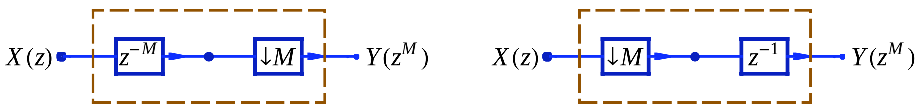
## SchematicSolverFigureMultirateDownsamplingIdentity

`SchematicSolverFigureMultirateDownsamplingIdentity` is a schematic specification that illustrates the downsampling identity.

```
Needs["SchematicSolver`"]
```

```
ShowSchematic[SchematicSolverFigureMultirateDownsamplingIdentity,
  GridLines → None, Frame → False];
```

Downsampling Identity

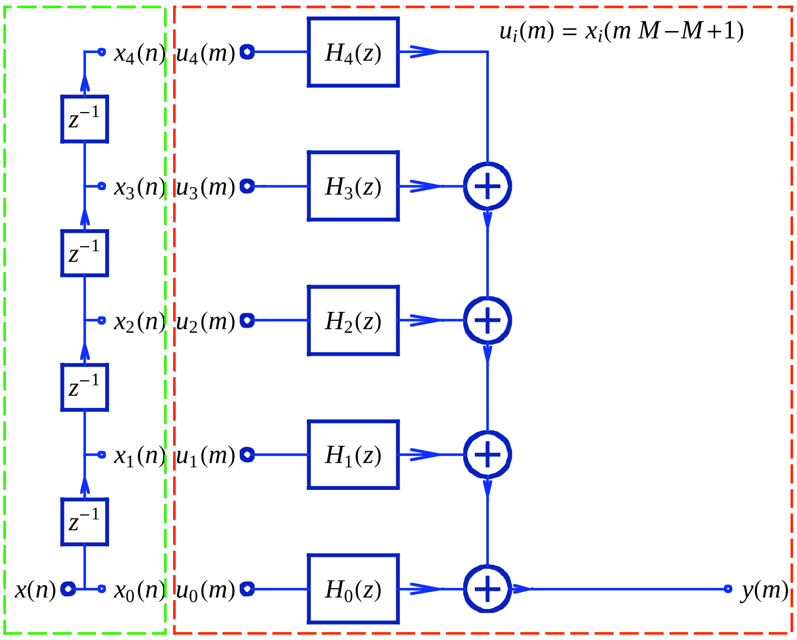


## SchematicSolverFigureMultirateDownsamplingImplemented

`SchematicSolverFigureMultirateDownsamplingImplemented` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigureMultirateDownsamplingImplemented;
 ShowSchematic[%, GridLines → None, Frame → False];
```

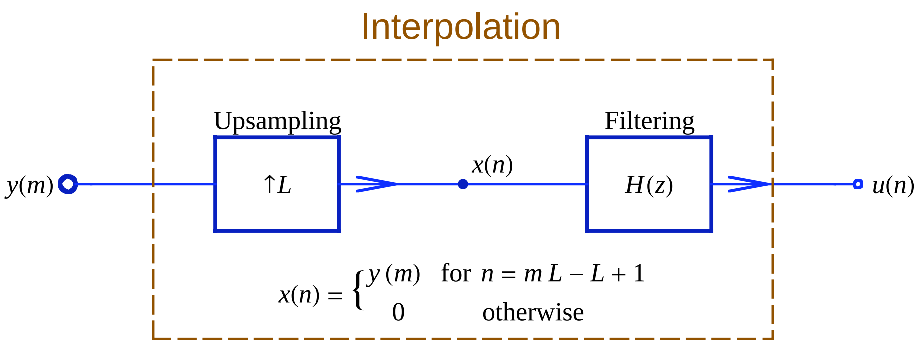


## SchematicSolverFigureMultirateInterpolation

`SchematicSolverFigureMultirateInterpolation` is a schematic specification that illustrates the interpolation system.

```
Needs["SchematicSolver`"]
```

```
ShowSchematic[SchematicSolverFigureMultirateInterpolation,
  GridLines → None, Frame → False];
```

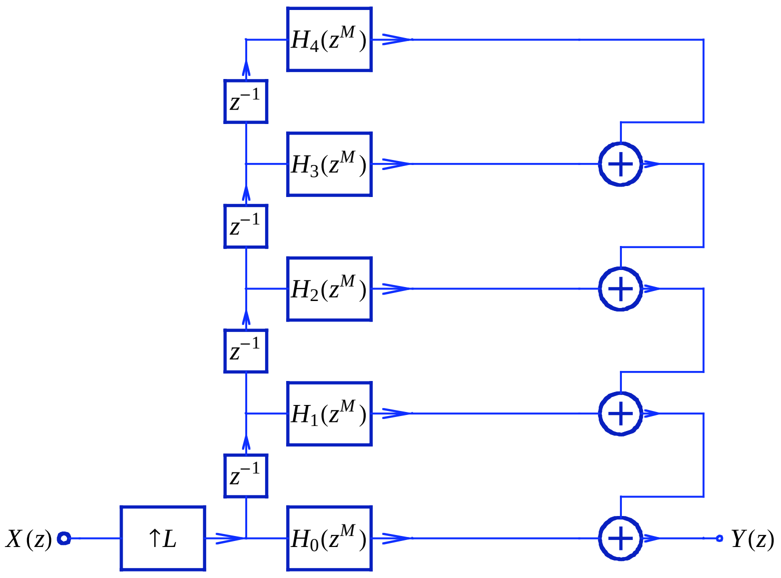

Interpolation
Upsampling
Filtering
$y(m)$
$\uparrow L$
$x(n)$
$H(z)$
$u(n)$
$x(n) = \begin{cases} y(m) & \text{for } n = mL - L + 1 \\ 0 & \text{otherwise} \end{cases}$

## SchematicSolverFigureMultirateUpsamplingClassic

`SchematicSolverFigureMultirateUpsamplingClassic` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigureMultirateUpsamplingClassic;
 ShowSchematic[%, GridLines → None, Frame → False];
```

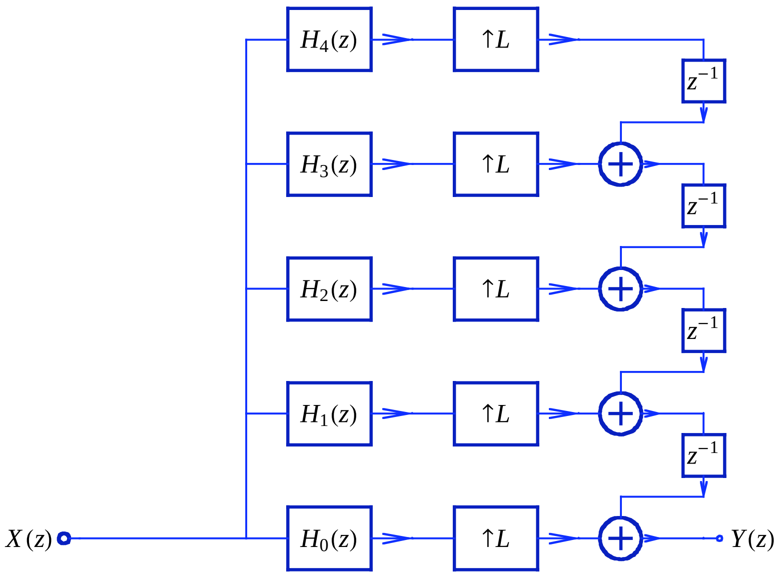


## SchematicSolverFigureMultirateUpsamplingEfficient

`SchematicSolverFigureMultirateUpsamplingEfficient` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigureMultirateUpsamplingEfficient;
 ShowSchematic[%, GridLines → None, Frame → False];
```



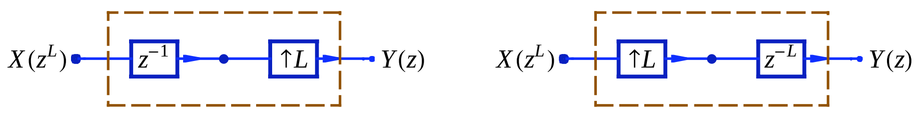
## SchematicSolverFigureMultirateUpsamplingIdentity

`SchematicSolverFigureMultirateUpsamplingIdentity` is a schematic specification that illustrates the upsampling identity.

```
Needs["SchematicSolver`"]
```

```
ShowSchematic[SchematicSolverFigureMultirateUpsamplingIdentity,
  GridLines → None, Frame → False];
```

Upsampling Identity



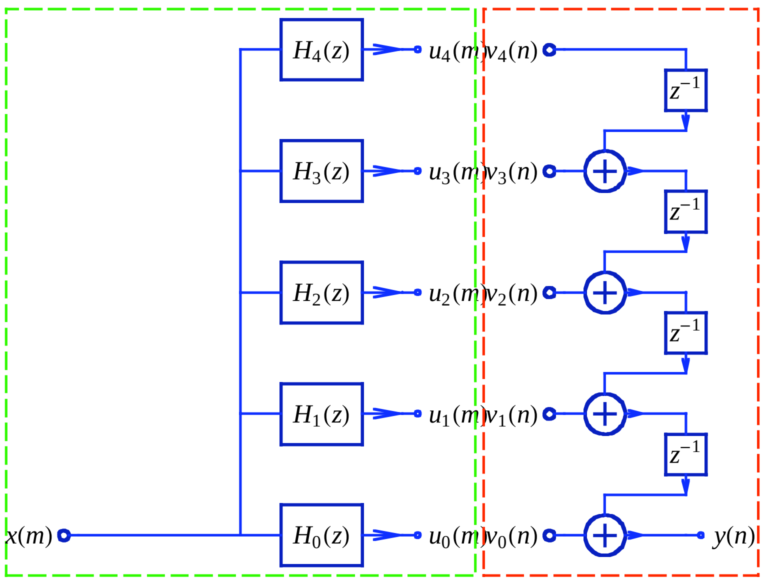
## SchematicSolverFigureMultirateUpsamplingImplemented

`SchematicSolverFigureMultirateUpsamplingImplemented` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigureMultirateUpsamplingImplemented;
 ShowSchematic[%, GridLines → None, Frame → False];
```

$$v(n) = \begin{cases} u(m) & \text{for } n = m\,L - L + 1 \\ 0 & \text{otherwise} \end{cases}$$

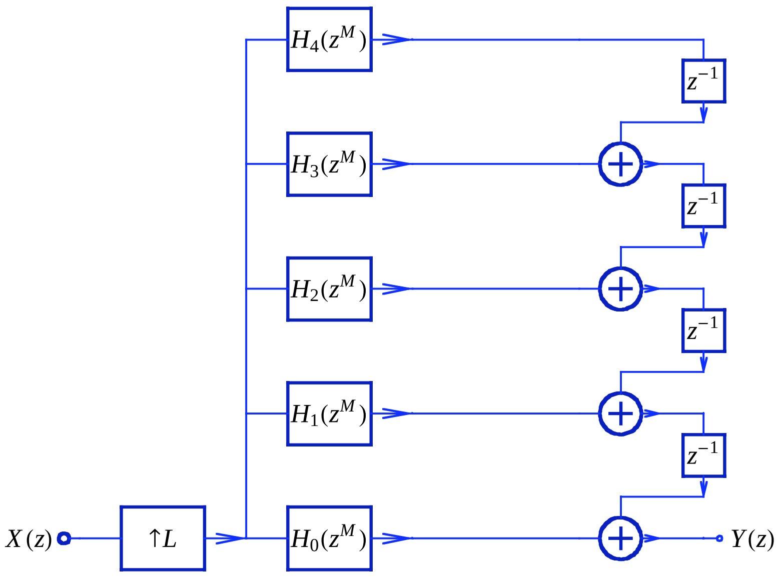


## SchematicSolverFigureMultirateUpsamplingTransposed

`SchematicSolverFigureMultirateUpsamplingTransposed` is a schematic specification that illustrates a multirate system.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigureMultirateUpsamplingTransposed;
 ShowSchematic[%, GridLines → None, Frame → False];
```

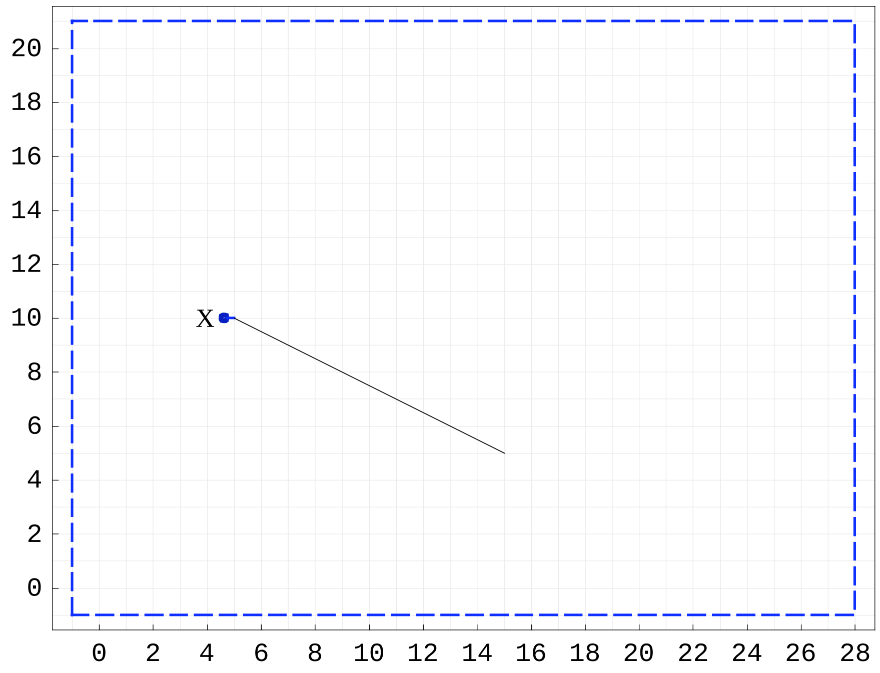


## SchematicSolverFigurePalettesDrawLine

`SchematicSolverFigurePalettesDrawLine` is a schematic specification that illustrates a drawing procedure.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigurePalettesDrawLine // ShowSchematic
```

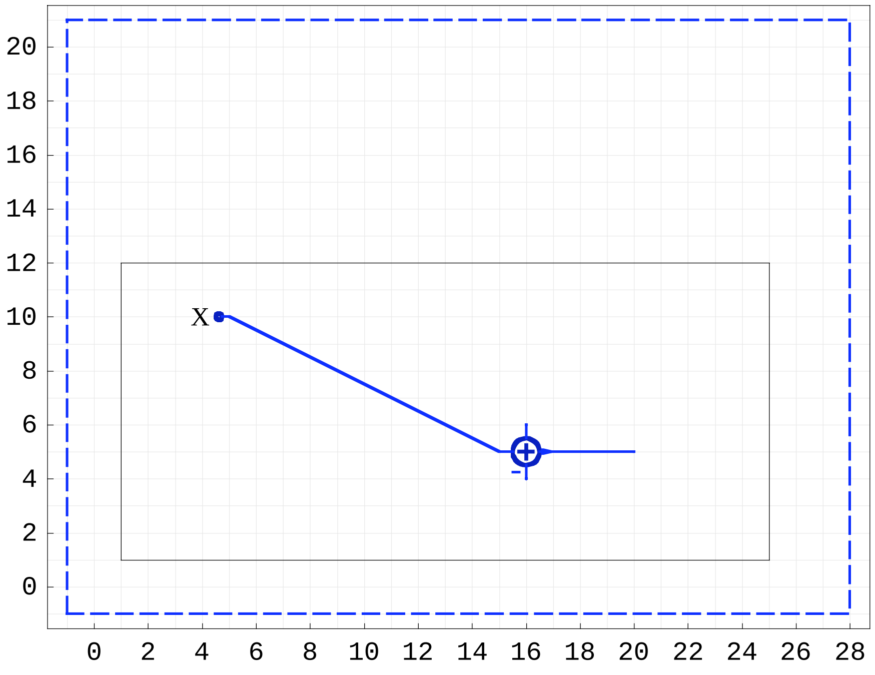


## SchematicSolverFigurePalettesDrawPolyline

`SchematicSolverFigurePalettesDrawPolyline` is a schematic specification that illustrates a drawing procedure.

```
Needs["SchematicSolver`"]
```

```
SchematicSolverFigurePalettesDrawPolyline // ShowSchematic
```

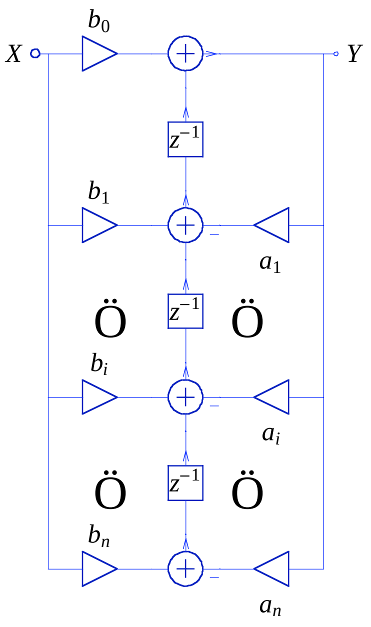


## SchematicSolverFigureProcessingTransposedDirectForm2IIR

`SchematicSolverFigureProcessingTransposedDirectForm2IIR` is a schematic specification that illustrates the transposed direct form 2 IIR realization.

```
Needs["SchematicSolver`"]
```

```
ShowSchematic[SchematicSolverFigureProcessingTransposedDirectForm2IIR,
  GridLines → None, Frame → False];
```

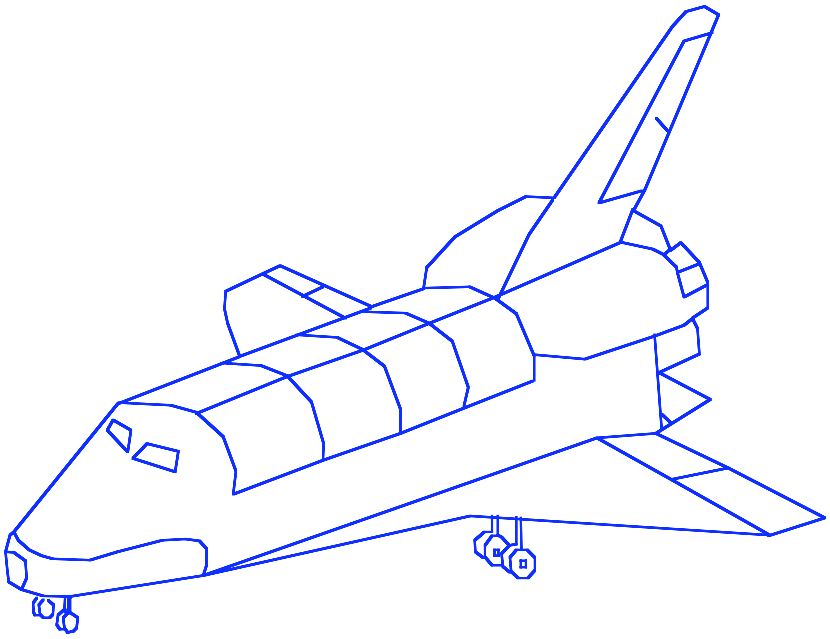


### SchematicSolverFigureShuttle

`SchematicSolverFigureShuttle` is a schematic specification that illustrates a lineart of the Space Shuttle.

```
Needs["SchematicSolver`"]
```

```
ShowSchematic[SchematicSolverFigureShuttle, GridLines → None, Frame → False];
```



### $VersionSchematicSolverSchematicFigures

`$VersionSchematicSolverSchematicFigures` is a variable that contains information about the figures package version and release date.

```
Needs["SchematicSolver`"]
```

```
$VersionSchematicSolverSchematicFigures
```

```
2.0 (January 26, 2004. 03:57)
```

# 15. Processing with *SchematicSolver*

## 15.1. Introduction

*SchematicSolver* processes discrete signals and simulate discrete systems.

*SchematicSolver* draws schematics of systems and symbolically solves systems directly from schematics:

(a) generates the equations describing a system,

(b) finds the system response,

(c) computes the system transfer function,

(d) simulates discrete system, and

(e) generates a new function that is a software implementation of the discrete system.

If the package has not already been loaded, we load it with

```
Needs["SchematicSolver`"];
```

We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[InputNotebook[], ImageSize → {350, 300}];
SetOptions[ShowSchematic, ElementScale → 1,
  FontSize → Automatic, Frame → True, GridLines → Automatic, PlotRange → All];
SetOptions[DrawElement, ElementSize → {1, 1},
  PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
    {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
  ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
  TextStyle → {FontFamily → Times, FontSize → 12}];
```

## 15.2. Drawing and Solving Systems

### Draw a System Using *SchematicSolver*

Consider a discrete system and find the system equations and the transfer function using *SchematicSolver*.

First, we represent the system as a list of discrete elements:

```
exampleSystem = {{"Input", {0, 8}, X},
   {"Output", {9, 8}, "Y1"}, {"Output", {9, 0}, "Y0"},
   {"Delay", {{2, 0}, {6, 0}}, 1}, {"Delay", {{4, 3}, {4, 5}}, 1},
   {"Delay", {{4, 5}, {4, 7}}, 1}, {"Line", {{2, 8}, {2, 2}}},
   {"Line", {{3, 6}, {5, 10}}}, {"Line", {{2, 2}, {2, 0}}},
   {"Line", {{6, 0}, {7, 0}}}, {"Line", {{6, 3}, {8, 5}}},
   {"Line", {{6, 5}, {8, 3}}}, {"Line", {{6, 8}, {7, 8}}},
   {"Adder", {{2, 2}, {4, 0}, {4, 3}, {3, 6}}, {1, 0, 2, -1}},
   {"Adder", {{2, 8}, {4, 7}, {4, 8}, {3, 10}}, {1, 1, 0, 2}},
   {"Adder", {{4, 8}, {4, 7}, {6, 8}, {5, 10}}, {0, 1, 2, 1}},
   {"Adder", {{7, 0}, {8, -1}, {9, 0}, {8, 3}}, {1, 0, 2, -1}},
   {"Adder", {{7, 8}, {8, 5}, {9, 8}, {8, 9}}, {1, 1, 2, 0}},
   {"Multiplier", {{0, 8}, {2, 8}}, k0},
   {"Multiplier", {{3, 10}, {5, 10}}, b}, {"Multiplier", {{6, 8}, {6, 5}}, k1},
   {"Multiplier", {{6, 0}, {6, 3}}, k1, "", TextOffset → {0, 1}}};
```

(In the printed code the output labels are "$Y_1$" and "$Y_0$".)

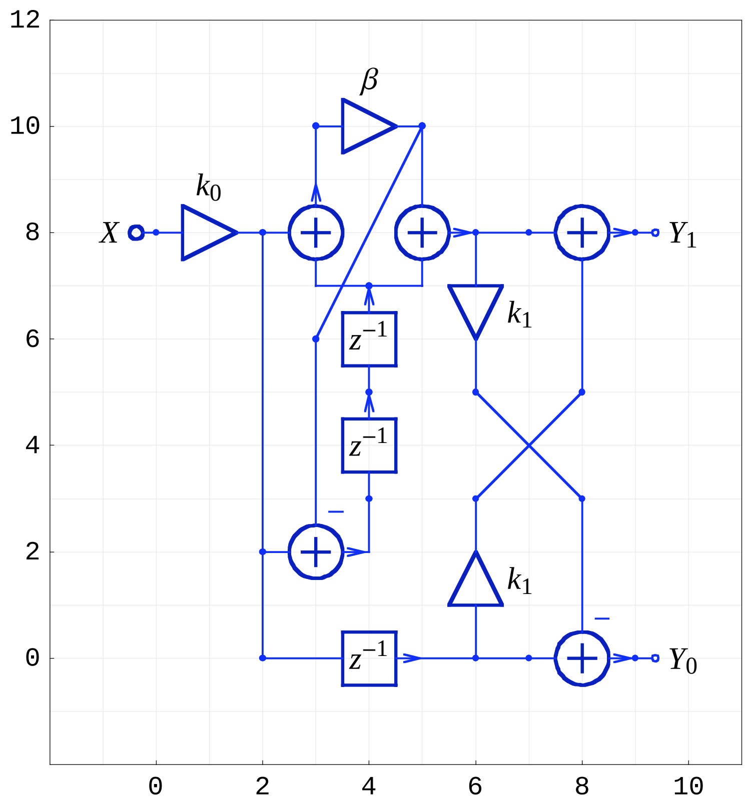
For better typesetting, the system parameters can be displayed with subscripts:

```
parameterSubstitution1 = {k0 → k₀, k1 → k₁, b → β}
```

$\{k0 \to k_0,\ k1 \to k_1,\ b \to \beta\}$

`ShowSchematic` draws the schematic of the system:

```
ShowSchematic[exampleSystem /.
   parameterSubstitution1, PlotRange → {{-2, 11}, {-2, 12}}];
```



## Set up System Equations Using *SchematicSolver*

`DiscreteSystemEquations` derives the equations that describe the system:

```
{eqns, vars} = DiscreteSystemEquations[exampleSystem];
eqns // ColumnForm
```

```
Y[{0, 8}] == X
Y[{6, 0}] == Y[{2,8}]/z
Y[{4, 5}] == Y[{4,3}]/z
Y[{4, 7}] == Y[{4,5}]/z
Y[{4, 3}] == Y[{2, 8}] - Y[{3, 6}]
Y[{3, 10}] == Y[{2, 8}] + Y[{4, 7}]
Y[{6, 8}] == Y[{3, 6}] + Y[{4, 7}]
Y[{9, 0}] == Y[{6, 0}] - Y[{6, 5}]
Y[{9, 8}] == Y[{6, 3}] + Y[{6, 8}]
Y[{2, 8}] == k0 Y[{0, 8}]
Y[{3, 6}] == b Y[{3, 10}]
Y[{6, 5}] == k1 Y[{6, 8}]
Y[{6, 3}] == k1 Y[{6, 0}]
```

`DiscreteSystemEquations` returns a list of the form {*systemEquations*, *systemVariables*}.

*systemEquations* is a list of equations describing the system.

*systemVariables* is a list of symbols that represent transforms of signals at nodes.

## Find System Response Using *SchematicSolver*

`DiscreteSystemResponse` computes the system response:

```
{resp, vars} = DiscreteSystemResponse[exampleSystem];
resp // ColumnForm
```

$$
\begin{array}{l}
Y[\{9,8\}] \to \frac{k0\ k1\ X}{z} - \frac{-k0\ X - b\ k0\ X\ z^2}{b+z^2} \\
Y[\{9,0\}] \to \frac{k0\ X}{z} - \frac{k1\ (k0\ X + b\ k0\ X\ z^2)}{b+z^2} \\
Y[\{6,5\}] \to \frac{k1\ (k0\ X + b\ k0\ X\ z^2)}{b+z^2} \\
Y[\{6,3\}] \to \frac{k0\ k1\ X}{z} \\
Y[\{4,5\}] \to -\frac{(-k0\ X + b\ k0\ X)\ z}{b+z^2} \\
Y[\{4,3\}] \to -\frac{(-k0\ X + b\ k0\ X)\ z^2}{b+z^2} \\
Y[\{3,10\}] \to \frac{k0\ X\ (1+z^2)}{b+z^2} \\
Y[\{6,8\}] \to -\frac{-k0\ X - b\ k0\ X\ z^2}{b+z^2} \\
Y[\{4,7\}] \to -\frac{-k0\ X + b\ k0\ X}{b+z^2} \\
Y[\{3,6\}] \to \frac{k0\ X\ (b+b\ z^2)}{b+z^2} \\
Y[\{6,0\}] \to \frac{k0\ X}{z} \\
Y[\{2,8\}] \to k0\ X \\
Y[\{0,8\}] \to X
\end{array}
$$

`DiscreteSystemResponse` returns a list of the form {*systemResponse*, *systemVariables*}.

*systemResponse* is a list of replacement rules describing the system response.

*systemVariables* is a list of symbols that represent transforms of signals at nodes.

## Compute Signals Using *SchematicSolver*

`DiscreteSystemSignals` computes signals at nodes:

```
{sigs, vars} = DiscreteSystemSignals[exampleSystem];
{sigs, vars} // Transpose // TableForm
```

$$\begin{array}{ll}
\frac{\mathrm{k0}\,\mathrm{k1}\,X}{z}-\frac{-\mathrm{k0}\,X-b\,\mathrm{k0}\,X\,z^2}{b+z^2} & \mathtt{Y[\{9,\ 8\}]} \\
\frac{\mathrm{k0}\,X}{z}-\frac{\mathrm{k1}\,(\mathrm{k0}\,X+b\,\mathrm{k0}\,X\,z^2)}{b+z^2} & \mathtt{Y[\{9,\ 0\}]} \\
-\frac{-\mathrm{k0}\,X-b\,\mathrm{k0}\,X\,z^2}{b+z^2} & \mathtt{Y[\{6,\ 8\}]} \\
\frac{\mathrm{k1}\,(\mathrm{k0}\,X+b\,\mathrm{k0}\,X\,z^2)}{b+z^2} & \mathtt{Y[\{6,\ 5\}]} \\
\frac{\mathrm{k0}\,\mathrm{k1}\,X}{z} & \mathtt{Y[\{6,\ 3\}]} \\
\frac{\mathrm{k0}\,X}{z} & \mathtt{Y[\{6,\ 0\}]} \\
-\frac{-\mathrm{k0}\,X+b\,\mathrm{k0}\,X}{b+z^2} & \mathtt{Y[\{4,\ 7\}]} \\
-\frac{(-\mathrm{k0}\,X+b\,\mathrm{k0}\,X)\,z}{b+z^2} & \mathtt{Y[\{4,\ 5\}]} \\
-\frac{(-\mathrm{k0}\,X+b\,\mathrm{k0}\,X)\,z^2}{b+z^2} & \mathtt{Y[\{4,\ 3\}]} \\
\frac{\mathrm{k0}\,X\,(1+z^2)}{b+z^2} & \mathtt{Y[\{3,\ 10\}]} \\
\frac{\mathrm{k0}\,X\,(b+b\,z^2)}{b+z^2} & \mathtt{Y[\{3,\ 6\}]} \\
\mathrm{k0}\,X & \mathtt{Y[\{2,\ 8\}]} \\
X & \mathtt{Y[\{0,\ 8\}]}
\end{array}$$

`DiscreteSystemSignals` returns a list of the form {*systemSignals*, *systemVariables*}.

*systemSignals* is a list of expressions representing the signals at all nodes of the system.

*systemVariables* is a list of symbols that represent transforms of signals at nodes.

## Compute Transfer Function Using *SchematicSolver*

`DiscreteSystemTransferFunction` finds the transfer function:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[exampleSystem]
```

$$\left\{\left\{\left\{-\frac{-b\,k0\,k1 - k0\,z - k0\,k1\,z^2 - b\,k0\,z^3}{z\,(b+z^2)}\right\}, \left\{-\frac{k0\,(-b + k1\,z - z^2 + b\,k1\,z^3)}{z\,(b+z^2)}\right\}\right\}, \{Y[\{0, 8\}]\}, \{Y[\{9, 8\}], Y[\{9, 0\}]\}\right\}$$

`DiscreteSystemTransferFunction` returns a list of the form {*transferFunctionMatrix*, *systemInputs*, *systemOutputs*}.

*transferFunctionMatrix* is the transfer function matrix of the system.

*systemInputs* is a list of symbols that represent the system inputs.

*systemOutputs* is a list of symbols that represent the system outputs.

The symbol *z* is reserved for the complex variable in the *z*-transform domain.

Each row of *transferFunctionMatrix* corresponds to a system output and each column corresponds to a system input:

```
tfMatrix // MatrixForm
```

$$\begin{pmatrix} -\frac{-b\,k0\,k1 - k0\,z - k0\,k1\,z^2 - b\,k0\,z^3}{z\,(b+z^2)} \\ -\frac{k0\,(-b + k1\,z - z^2 + b\,k1\,z^3)}{z\,(b+z^2)} \end{pmatrix}$$

For better typesetting, the system parameters can be displayed with subscripts:

```
tfMatrix /. {b → β, k0 → k₀, k1 → k₁} // TraditionalForm
```

$$\begin{pmatrix} -\frac{-\beta k_0 z^3 - k_0 k_1 z^2 - k_0 z - \beta k_0 k_1}{z\,(z^2+\beta)} \\ -\frac{k_0\,(\beta k_1 z^3 - z^2 + k_1 z - \beta)}{z\,(z^2+\beta)} \end{pmatrix}$$

You can extract transfer functions from `tfMatrix` with

```
tf1 = tfMatrix[[1, 1]]
```

$$-\frac{-b\,k0\,k1 - k0\,z - k0\,k1\,z^2 - b\,k0\,z^3}{z\,(b+z^2)}$$

```
tf2 = tfMatrix[[2, 1]]
```

$$-\frac{k0\,(-b + k1\,z - z^2 + b\,k1\,z^3)}{z\,(b+z^2)}$$

For some specific values

```
myValues = {b → 2 / 5, k0 → 1 / 2, k1 → 1}
```

$$\left\{b \to \frac{2}{5},\ k0 \to \frac{1}{2},\ k1 \to 1\right\}$$

the transfer function matrix becomes

```
myTFspecific = tfMatrix /. myValues // Simplify;
% // MatrixForm
```

$$\begin{pmatrix} \frac{2+5z+5z^2+2z^3}{4z+10z^3} \\ \frac{2-5z+5z^2-2z^3}{4z+10z^3} \end{pmatrix}$$

```
myTFspecific1 = myTFspecific[[1, 1]]
```

$$\frac{2+5z+5z^2+2z^3}{4z+10z^3}$$

```
myTFspecific2 = myTFspecific[[2, 1]]
```

$$\frac{2-5z+5z^2-2z^3}{4z+10z^3}$$

`DiscreteSystemDisplayForm` displays transfer functions in a traditional form

```
myTFspecific1 // DiscreteSystemDisplayForm
```

$$\frac{2+5\,z^{-1}+5\,z^{-2}+2\,z^{-3}}{10+4\,z^{-2}}$$

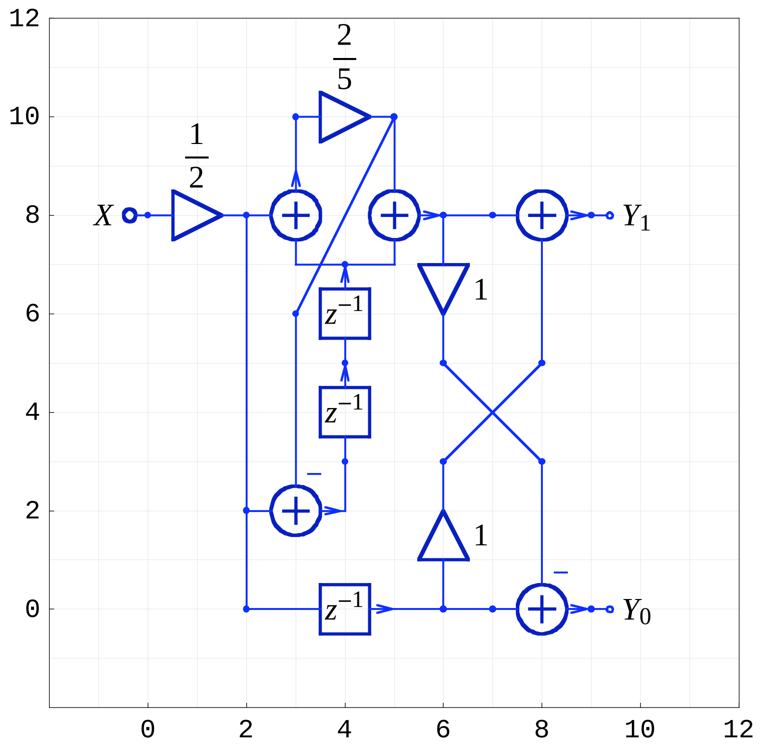
```
myTFspecific2 // DiscreteSystemDisplayForm
```

$$\frac{-2+5\,z^{-1}-5\,z^{-2}+2\,z^{-3}}{10+4\,z^{-2}}$$

## Plot Frequency Response Using *SchematicSolver*

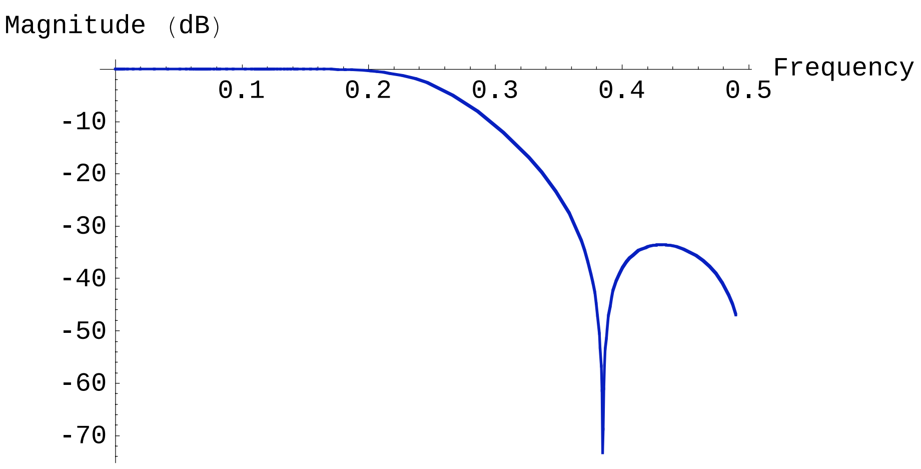
Let us redraw the schematic of the system with the specific coefficients:

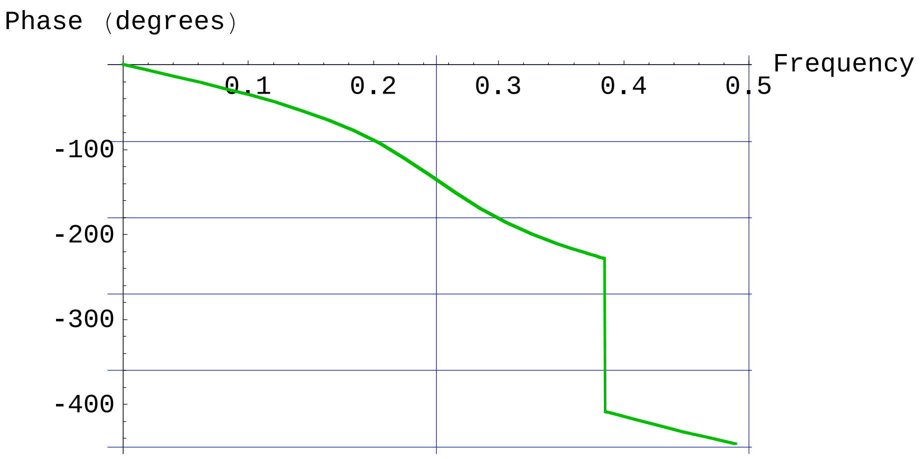
```
ShowSchematic[exampleSystem /. myValues, PlotRange → {{-2, 12}, {-2, 12}}]
```



`DiscreteSystemFrequencyResponse` plots the magnitude and the phase characteristics of a discrete system:

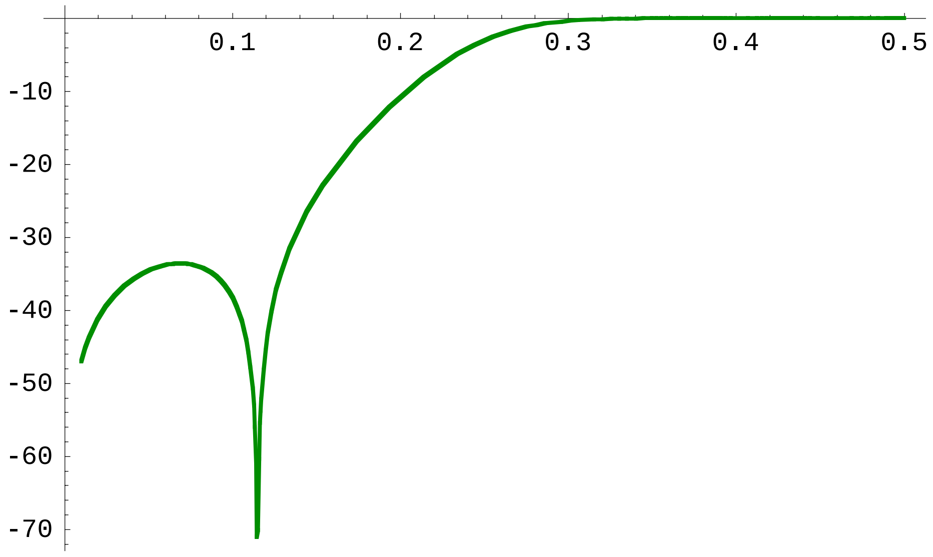
```
DiscreteSystemFrequencyResponse[myTFspecific1, {0, 0.49}];
```





`DiscreteSystemMagnitudeResponsePlot` plots the magnitude characteristic of a discrete system:

```
DiscreteSystemMagnitudeResponsePlot[myTFspecific2, {0.01, 0.5}];
```



## 15.3. Compute Impulse Response with *SchematicSolver*

Samples that are inputted to a MIMO system are represented in *SchematicSolver* as matrices that may contain several sequences. You can create the most typical inputs with the *SchematicSolver*'s functions such as `UnitImpulseSequence`:

```
impulseSeq = UnitImpulseSequence[21]
```

```
{{1}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0},
 {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}}
```

`DiscreteSystemSimulation` computes the impulse response of the system:

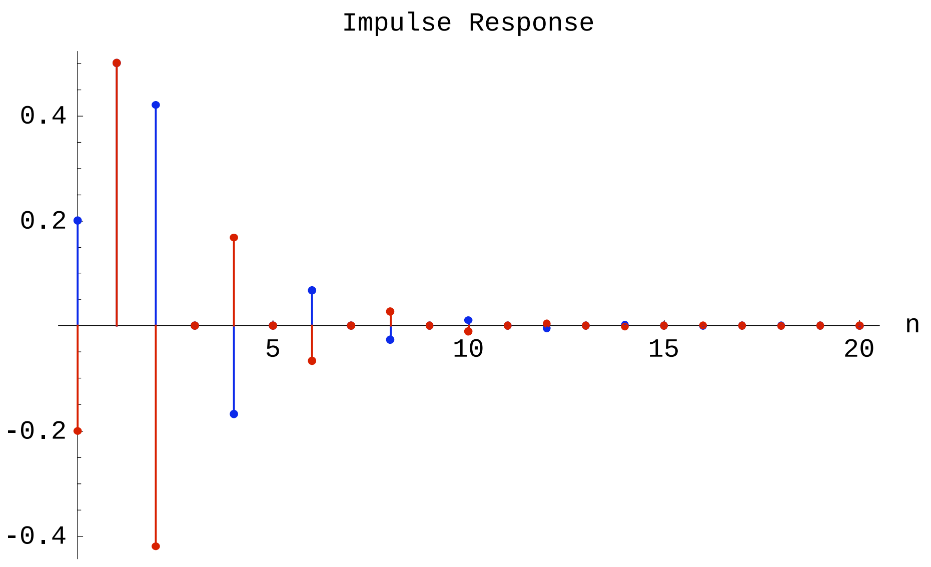
```
impulseResponseSeq = DiscreteSystemSimulation[exampleSystem /. myValues, impulseSeq]
```

$$\left\{\left\{\frac{1}{5}, -\frac{1}{5}\right\}, \left\{\frac{1}{2}, \frac{1}{2}\right\}, \left\{\frac{21}{50}, -\frac{21}{50}\right\}, \{0, 0\}, \left\{-\frac{21}{125}, \frac{21}{125}\right\}, \{0, 0\}, \left\{\frac{42}{625}, -\frac{42}{625}\right\}, \{0, 0\}, \left\{-\frac{84}{3125}, \frac{84}{3125}\right\}, \{0, 0\}, \left\{\frac{168}{15625}, -\frac{168}{15625}\right\}, \{0, 0\}, \left\{-\frac{336}{78125}, \frac{336}{78125}\right\}, \{0, 0\}, \left\{\frac{672}{390625}, -\frac{672}{390625}\right\}, \{0, 0\}, \left\{-\frac{1344}{1953125}, \frac{1344}{1953125}\right\}, \{0, 0\}, \left\{\frac{2688}{9765625}, -\frac{2688}{9765625}\right\}, \{0, 0\}, \left\{-\frac{5376}{48828125}, \frac{5376}{48828125}\right\}\right\}$$

Note that `impulseResponseSeq` is a matrix of samples.

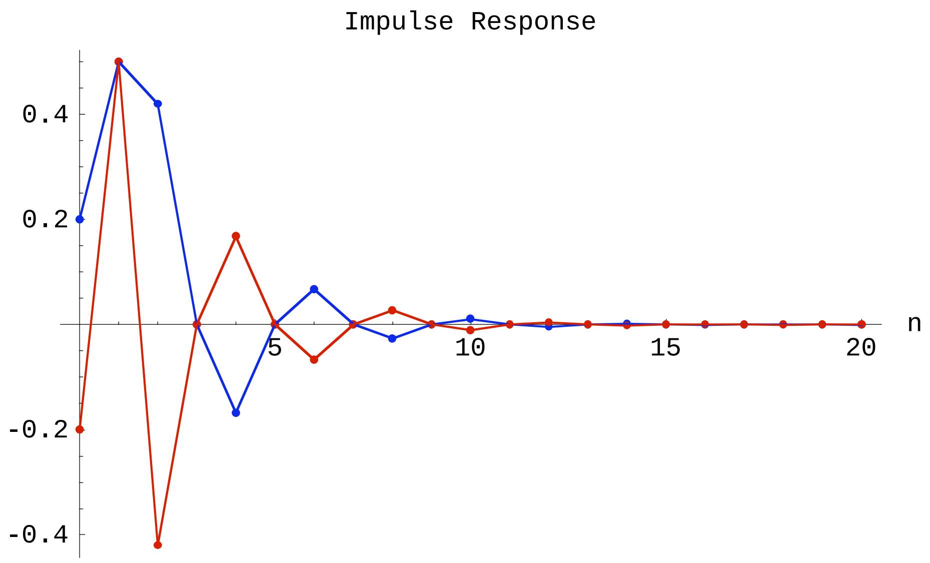
`SequencePlot` plots discrete signals represented by sequences:

```
SequencePlot[impulseResponseSeq,
  PlotLabel → "Impulse Response",
  AxesLabel → {"n", ""}];
```



You can plot the two discrete signals more clearly by setting the `SequencePlot` options to `StemPlot→False` and `PlotJoined→True`.

```
SequencePlot[impulseResponseSeq,
  PlotLabel → "Impulse Response",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



## 15.4. Symbolic Impulse Response with *SchematicSolver*

Consider the transfer function of the example system, returned by *SchematicSolver*, and assign numeric values to some parameters, but keep one parameter as a symbol:

```
myValues2 = {b → K, k0 → 1 / 2, k1 → 1}
```

$$\left\{b \to K,\ k0 \to \frac{1}{2},\ k1 \to 1\right\}$$

```
{myTFsymbolic, systemInp, systemOut} =
 DiscreteSystemTransferFunction[exampleSystem /.
   myValues2];
myTFsymbolic // MatrixForm
```

$$\begin{pmatrix} \frac{1}{z} + \frac{-K+z-z^2+K z^3}{2 z (K+z^2)} \\ -\frac{-K+z-z^2+K z^3}{2 z (K+z^2)} \end{pmatrix}$$

```
myTFsymbolic[[1, 1]] // DiscreteSystemDisplayForm
```

$$\frac{K + z^{-1} + z^{-2} + K z^{-3}}{2 + 2 K z^{-2}}$$

```
myTFsymbolic[[2, 1]] // DiscreteSystemDisplayForm
```

$$\frac{-K + z^{-1} - z^{-2} + K z^{-3}}{2 + 2 K z^{-2}}$$

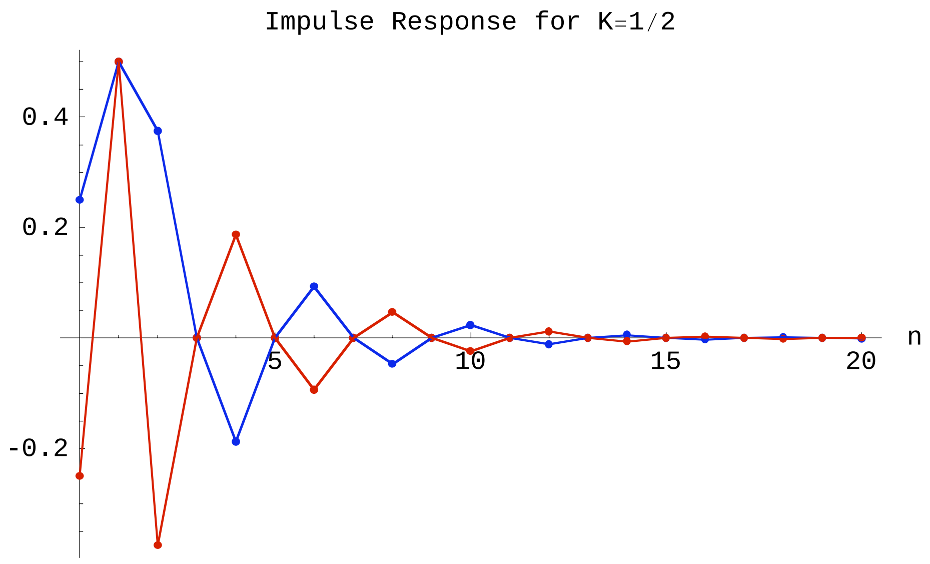
`DiscreteSystemSimulation` finds the symbolic impulse response for both outputs in terms of K:

```
impulseResponseSeq2 = DiscreteSystemSimulation[exampleSystem /. myValues2, impulseSeq]
```

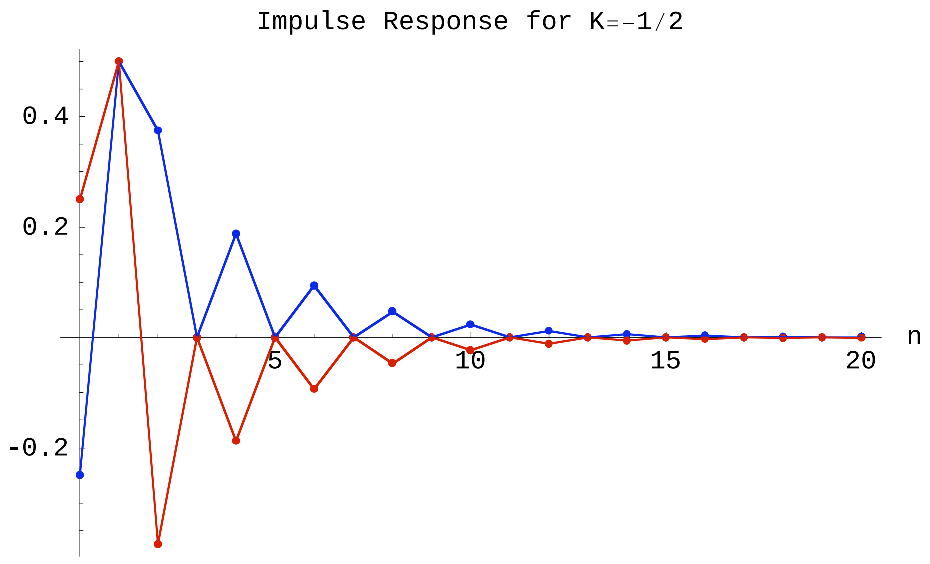
$$\begin{aligned}
&\Big\{\Big\{\frac{K}{2}, -\frac{K}{2}\Big\}, \Big\{\frac{1}{2}, \frac{1}{2}\Big\}, \Big\{\frac{1}{2} - \frac{K}{2} + \Big(\frac{1}{2} - \frac{K}{2}\Big) K, -\frac{1}{2} + \frac{K}{2} - \Big(\frac{1}{2} - \frac{K}{2}\Big) K\Big\},\\
&\{0, 0\}, \Big\{-\Big(\frac{1}{2} - \frac{K}{2}\Big) K - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^2, \Big(\frac{1}{2} - \frac{K}{2}\Big) K + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^2\Big\},\\
&\{0, 0\}, \Big\{\Big(\frac{1}{2} - \frac{K}{2}\Big) K^2 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^3, -\Big(\frac{1}{2} - \frac{K}{2}\Big) K^2 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^3\Big\},\\
&\{0, 0\}, \Big\{-\Big(\frac{1}{2} - \frac{K}{2}\Big) K^3 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^4, \Big(\frac{1}{2} - \frac{K}{2}\Big) K^3 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^4\Big\},\\
&\{0, 0\}, \Big\{\Big(\frac{1}{2} - \frac{K}{2}\Big) K^4 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^5, -\Big(\frac{1}{2} - \frac{K}{2}\Big) K^4 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^5\Big\},\\
&\{0, 0\}, \Big\{-\Big(\frac{1}{2} - \frac{K}{2}\Big) K^5 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^6, \Big(\frac{1}{2} - \frac{K}{2}\Big) K^5 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^6\Big\},\\
&\{0, 0\}, \Big\{\Big(\frac{1}{2} - \frac{K}{2}\Big) K^6 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^7, -\Big(\frac{1}{2} - \frac{K}{2}\Big) K^6 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^7\Big\},\\
&\{0, 0\}, \Big\{-\Big(\frac{1}{2} - \frac{K}{2}\Big) K^7 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^8, \Big(\frac{1}{2} - \frac{K}{2}\Big) K^7 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^8\Big\},\\
&\{0, 0\}, \Big\{\Big(\frac{1}{2} - \frac{K}{2}\Big) K^8 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^9, -\Big(\frac{1}{2} - \frac{K}{2}\Big) K^8 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^9\Big\},\\
&\{0, 0\}, \Big\{-\Big(\frac{1}{2} - \frac{K}{2}\Big) K^9 - \Big(\frac{1}{2} - \frac{K}{2}\Big) K^{10}, \Big(\frac{1}{2} - \frac{K}{2}\Big) K^9 + \Big(\frac{1}{2} - \frac{K}{2}\Big) K^{10}\Big\}\Big\}
\end{aligned}$$

For specific values of K the impulse response looks like

```
SequencePlot[impulseResponseSeq2 /. K → 1 / 2,
  PlotLabel → "Impulse Response for K=1/2",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



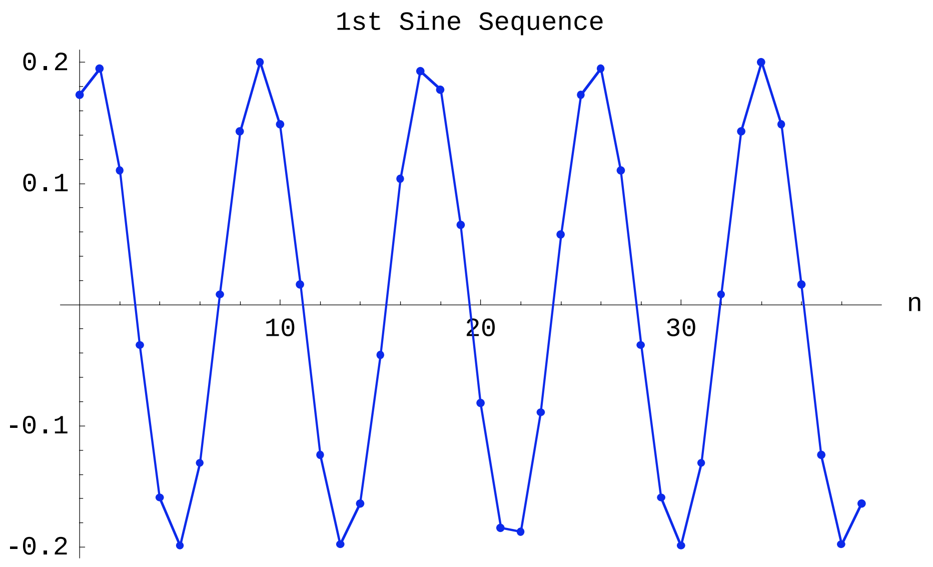
```
SequencePlot[impulseResponseSeq2 /. K → -1 / 2,
  PlotLabel → "Impulse Response for K=-1/2",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



## 15.5. Processing Signals with *SchematicSolver*

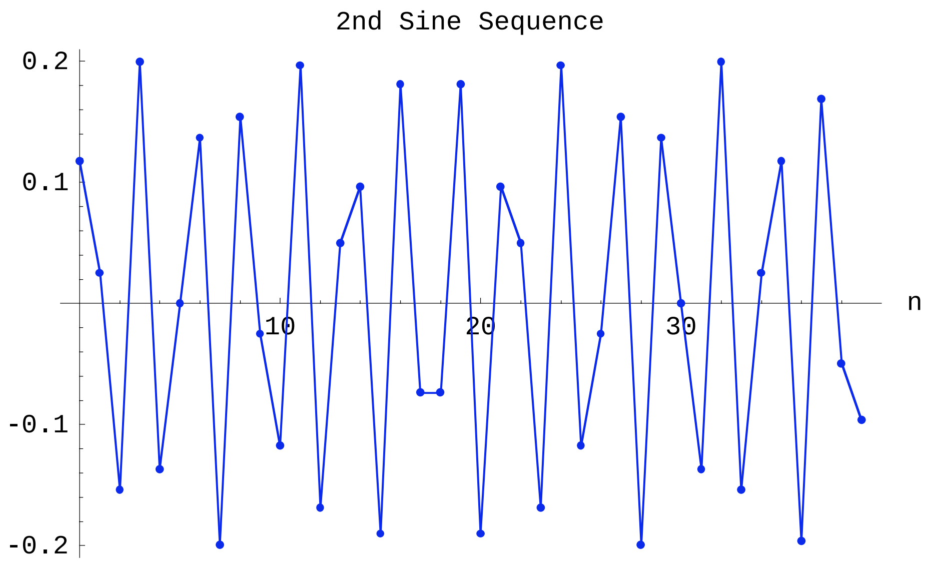
Consider two discrete sinusoidal signals in terms of the amplitude X, the digital frequency f and the phase $\phi$:

```
numberOfSamples = 40;

f1 = 12 / 100;
X1 = 1 / 5;
ϕ1 = π / 3;
inpSeq1 = X1 * UnitSineSequence[numberOfSamples, f1, ϕ1];

SequencePlot[inpSeq1,
  PlotLabel → "1st Sine Sequence",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



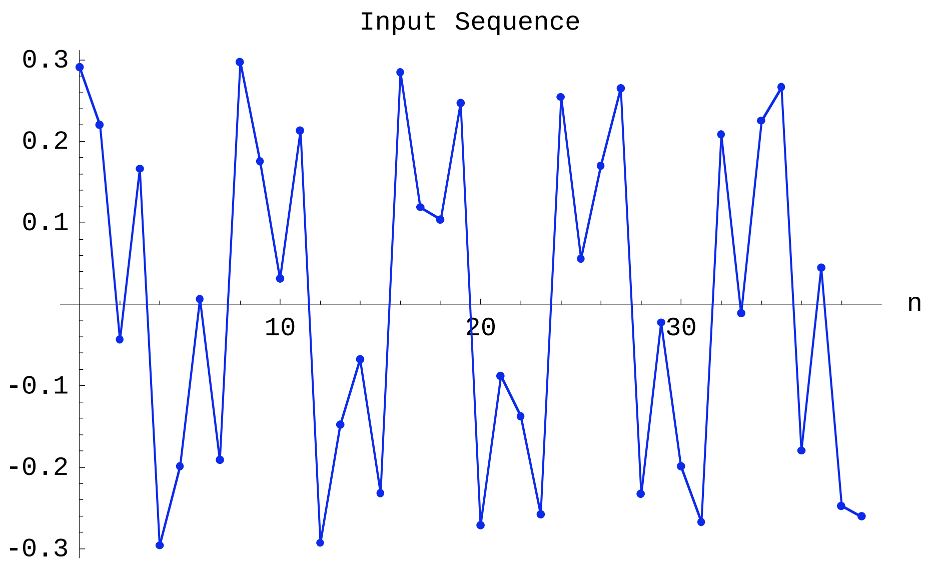
```
f2 = 38 / 100;
X2 = 1 / 5;
ϕ2 = π / 5;
inpSeq2 = X2 * UnitSineSequence[numberOfSamples, f2, ϕ2];
```

```
SequencePlot[inpSeq2,
  PlotLabel → "2nd Sine Sequence",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



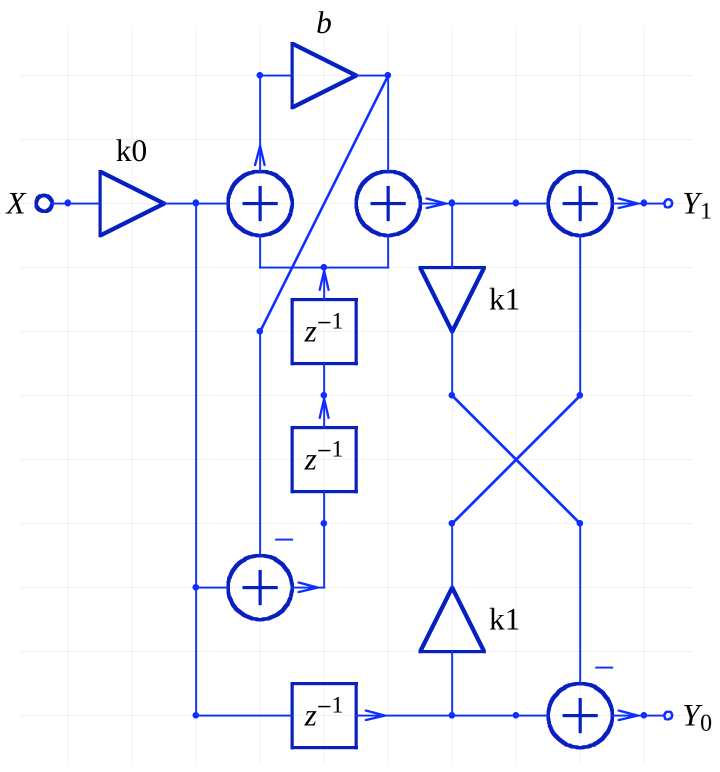
Assume that the input sequence is the sum of the two sequences:

```
inpSeq = inpSeq1 + inpSeq2;

SequencePlot[inpSeq,
  PlotLabel → "Input Sequence",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



Schematic of the example system follows:

```
ShowSchematic[exampleSystem, Frame → False];
```

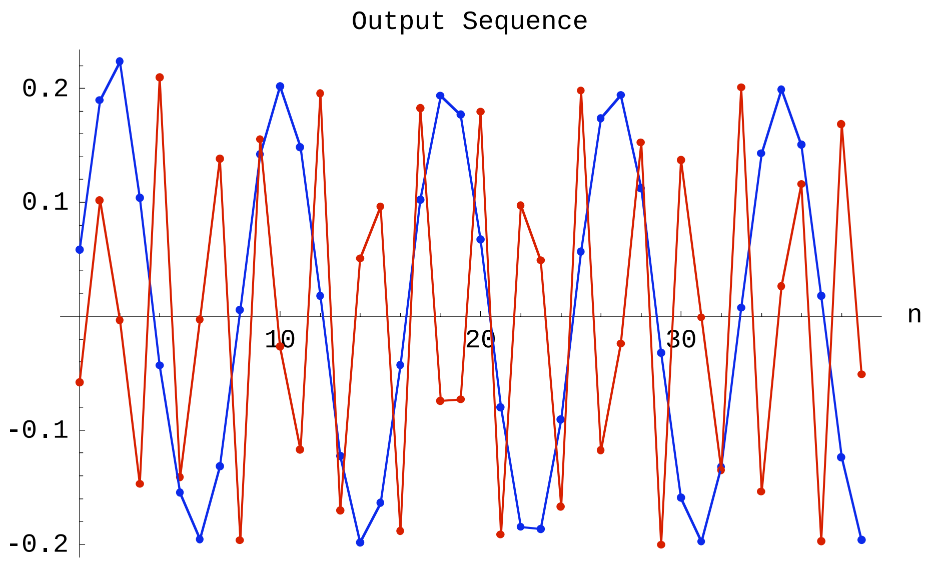


`DiscreteSystemSimulation` finds the response for the given parameter values:

```
myValues = {b → 2 / 5, k0 → 1 / 2, k1 → 1}
```

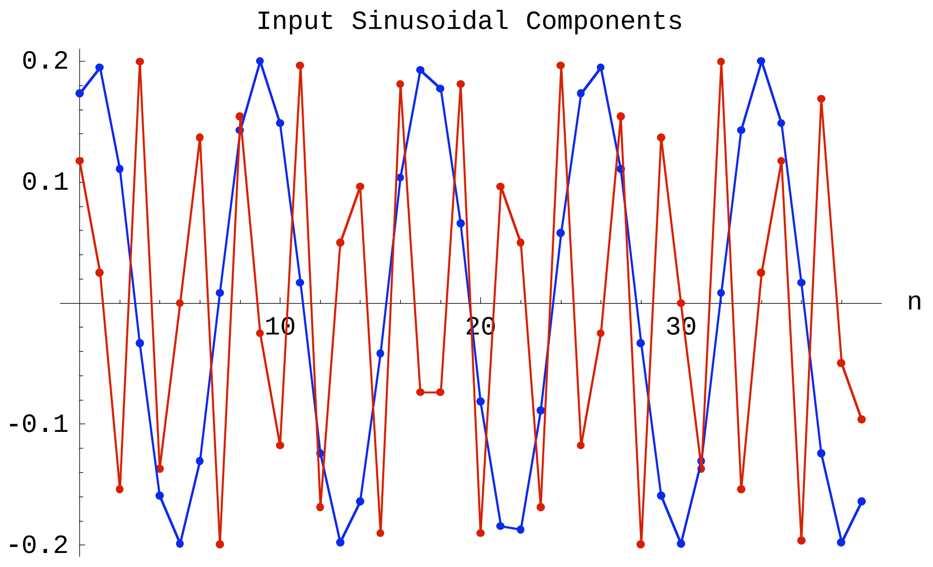
$\{b \to \frac{2}{5},\ k0 \to \frac{1}{2},\ k1 \to 1\}$

```
outSeq = DiscreteSystemSimulation[exampleSystem /. myValues, inpSeq];

SequencePlot[outSeq,
  PlotLabel → "Output Sequence",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



The two sinusoidal components of the input sequence can be plotted with `MultiplexSequence`:

```
SequencePlot[MultiplexSequence[inpSeq1, inpSeq2],
  PlotLabel → "Input Sinusoidal Components",
  StemPlot → False, PlotJoined → True,
  AxesLabel → {"n", ""}];
```



From the above plots of the input and output sequences it follows that this system (a) passes without attenuation the sequence `inpSeq1` at the output `Y1` and (b) passes without attenuation the sequence `inpSeq2` at output `Y2.`

## 15.6. Block Processing with Initial Conditions

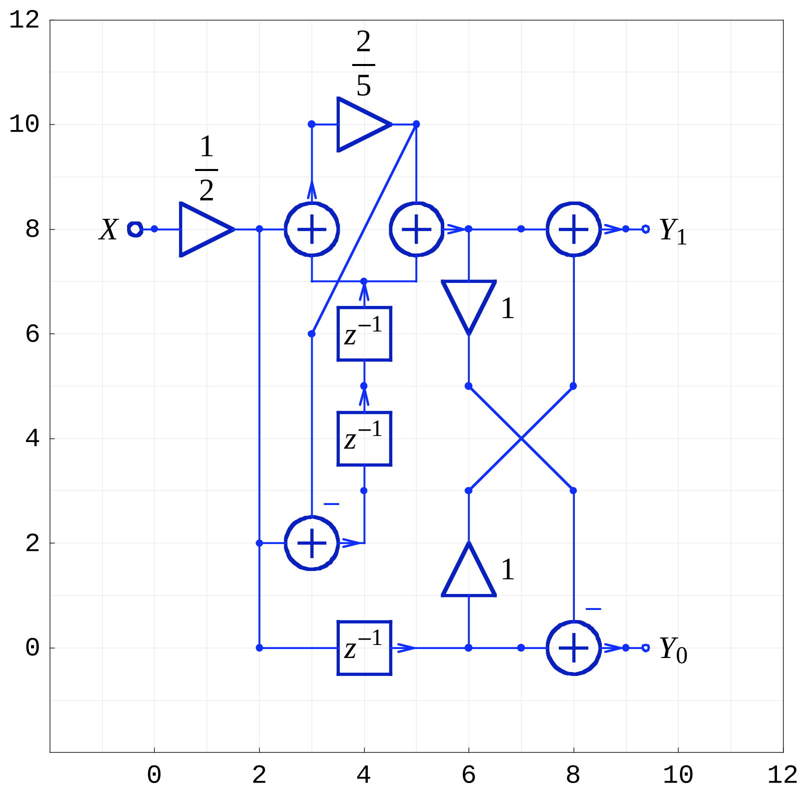
Initial conditions are important in many applications such as processing data in blocks.

Consider a system

```
myValues = {b → 2 / 5, k0 → 1 / 2, k1 → 1}
```

$\{b \to \frac{2}{5},\ k0 \to \frac{1}{2},\ k1 \to 1\}$

```
mySystem = exampleSystem /. myValues;
ShowSchematic[%, PlotRange → {{-2, 12}, {-2, 12}}]
```



`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system:

```
DiscreteSystemImplementation[mySystem, "myImplementationProcedure"];
```

```
    Implementation procedure name: myImplementationProcedure

    Implementation procedure usage:

     {{Y9p8, Y9p0}, {Y2p8, Y4p3, Y4p5}} = myImplementationProcedure[
      {Y0p8},{Y6p0, Y4p5, Y4p7},{}] is the template for calling
      the procedure.  The general template is {outputSamples,
      finalConditions} = procedureName[inputSamples, initialConditions,
      systemParameters]. See also: DiscreteSystemImplementationProcessing
```

`DiscreteSystemImplementationSummary` generates the system summary that points out the system input, initial state, parameter set, output, and final state:

```
DiscreteSystemImplementationSummary[mySystem]
   Input: {Y[{0, 8}]}
   Initial state: {Y[{6, 0}], Y[{4, 5}], Y[{4, 7}]}
   Parameter: {}
   Output: {Y[{9, 8}], Y[{9, 0}]}
   Final state: {Y[{2, 8}], Y[{4, 3}], Y[{4, 5}]}
```

The system has two initial states and two final states. The system has no parameters:

```
systemParameters = {};
```

Samples that are inputted to a MIMO system are represented in *SchematicSolver* as matrices that may contain several sequences. You can create the most typical input sequences, such as sine sequences, with

```
numberOfSamples = 2 * 50;

inpSeq = UnitSineSequence[numberOfSamples, 12 / 100] +
   UnitSineSequence[numberOfSamples, 38 / 100];
```

Suppose that we process the input sequence, `inpSeq`, in two blocks. First, we split the input sequence into two sequences:

```
inpSeq1 = Take[inpSeq, {1, numberOfSamples / 2}];
inpSeq2 = Take[inpSeq, {1 + numberOfSamples / 2, numberOfSamples}];
```

Let us process the first sequence with the generated function `myImplementationProcedure` for zero initial conditions:

```
initialConditions1 = {0, 0, 0};

{outSeq1, finalConditions1} = DiscreteSystemImplementationProcessing[inpSeq1,
   initialConditions1, systemParameters, myImplementationProcedure];
```

`finalConditions1` are the initial conditions for processing the next data block `inpSeq2.`

```
initialConditions2 = finalConditions1;

{outSeq2, finalConditions2} = DiscreteSystemImplementationProcessing[inpSeq2,
   initialConditions2, systemParameters, myImplementationProcedure];
```

The whole output sequence is obtained by joining the two sequences.

```
blockOutSeq = Join[outSeq1, outSeq2];
```

The same result should be obtained by processing the whole input sequence.

`DiscreteSystemSimulation` finds the output sequence for zero initial conditions.

```
outSeq = DiscreteSystemSimulation[mySystem, inpSeq];
```

The two output sequences, obtained by block processing and by processing the whole input sequence, should be the same:

```
(outSeq - blockOutSeq) // Simplify
```

```
{{0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}}
```

```
SameQ[outSeq, blockOutSeq]
```

```
True
```

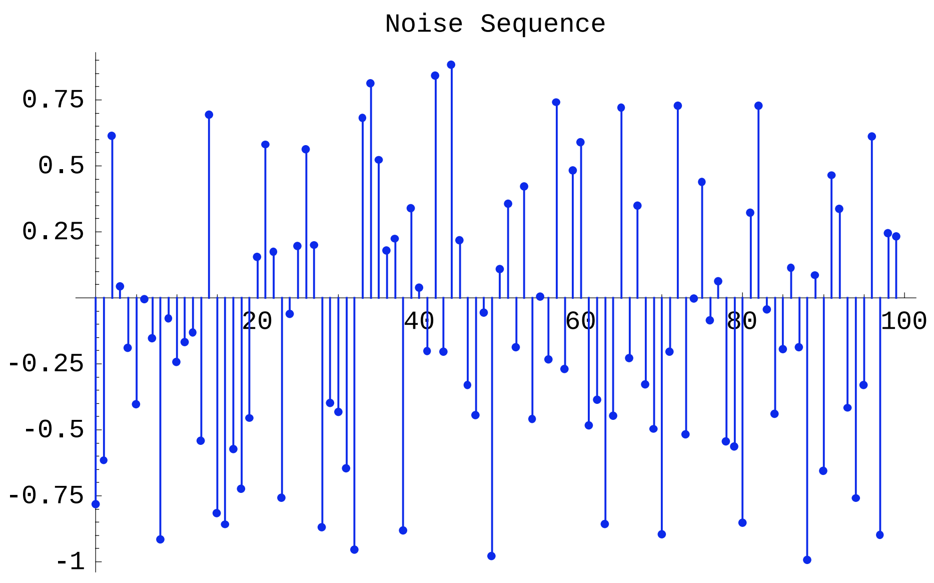
## 15.7. Processing Signals with Noise

`UnitNoiseSequence` can be used to add noise to the input signal:

```
numberOfSamples = 100;
```

```
noiseSeq = UnitNoiseSequence[numberOfSamples];
```

```
SequencePlot[noiseSeq, PlotLabel → "Noise Sequence"];
```

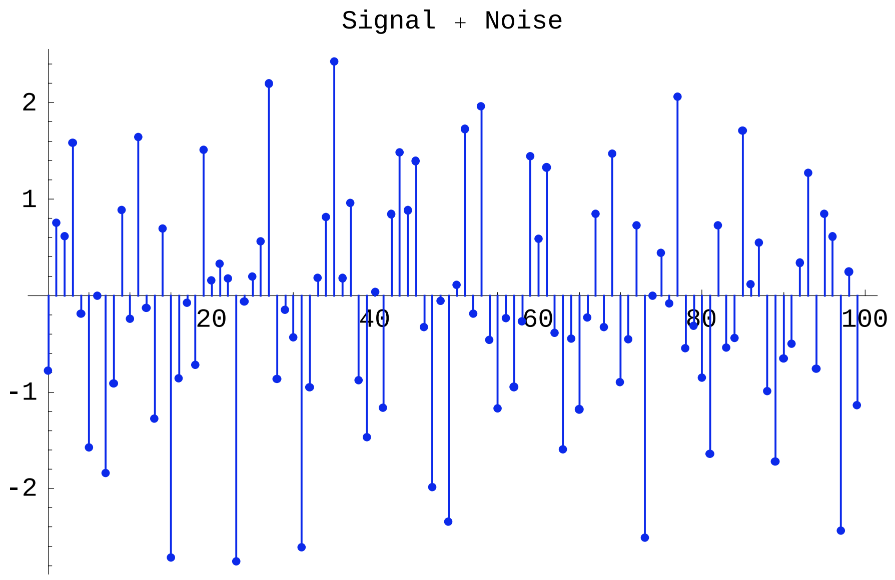


```
sineSeq = UnitSineSequence[numberOfSamples, 12 / 100] +
   UnitSineSequence[numberOfSamples, 38 / 100];
```

Here is a noisy discrete signal:

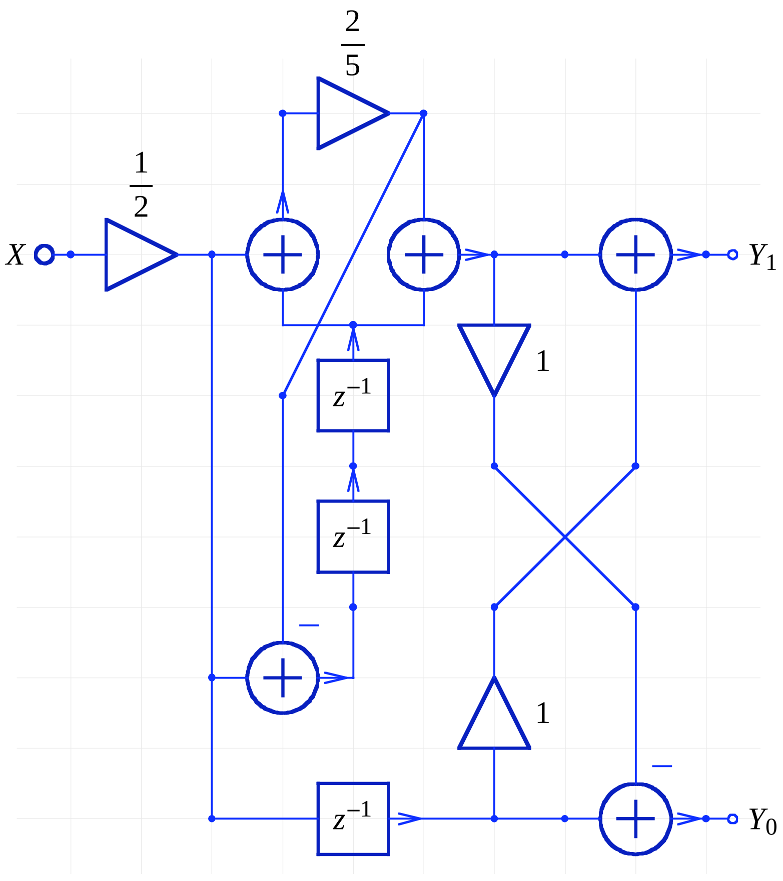
```
noisyInpSeq = sineSeq + noiseSeq;
```

```
SequencePlot[noisyInpSeq, PlotLabel → "Signal + Noise"];
```



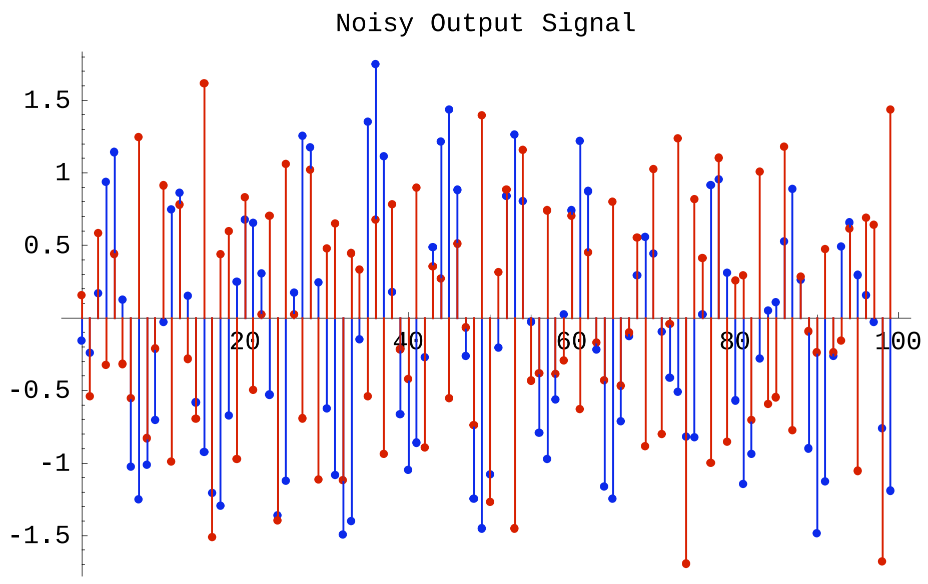
Assume that the following system is used to process `noisyInpSeq`:

```
mySystem = exampleSystem /. {b → 2 / 5, k0 → 1 / 2, k1 → 1};
ShowSchematic[%, Frame → False]
```



`DiscreteSystemSimulation` computes the output sequence:

```
noisyOutSeq = DiscreteSystemSimulation[mySystem, noisyInpSeq];
SequencePlot[noisyOutSeq, PlotLabel → "Noisy Output Signal"];
```



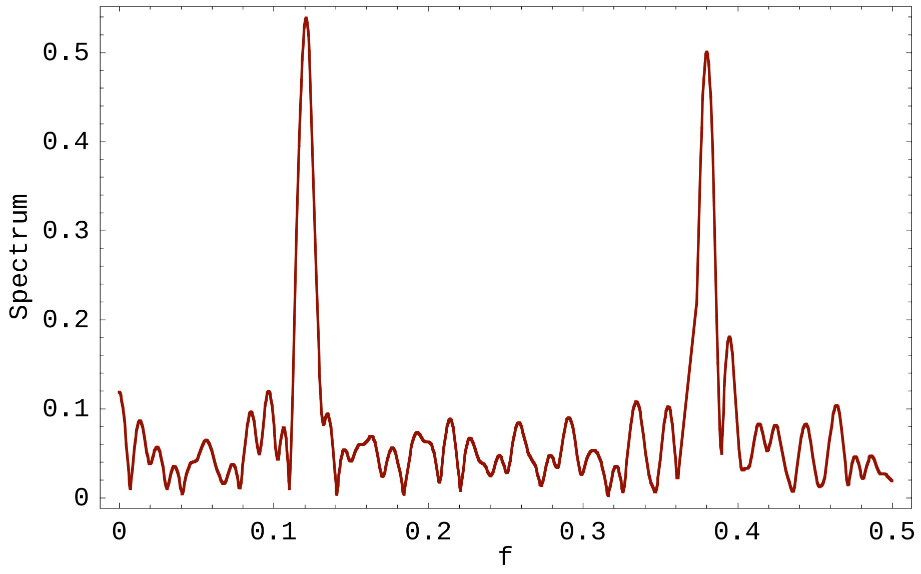
The above plot does not present enough information to examine the nature of the processed sequences, that is, to examine the system. Spectra of the discrete signals should be computed to get better insight into the characteristics of the system.

## 15.8. Signal Spectra

`SequenceFourierTransformMagnitudePlot` computes and plots the magnitude spectrum of discrete signals.

Here is the spectrum of the input sequence:

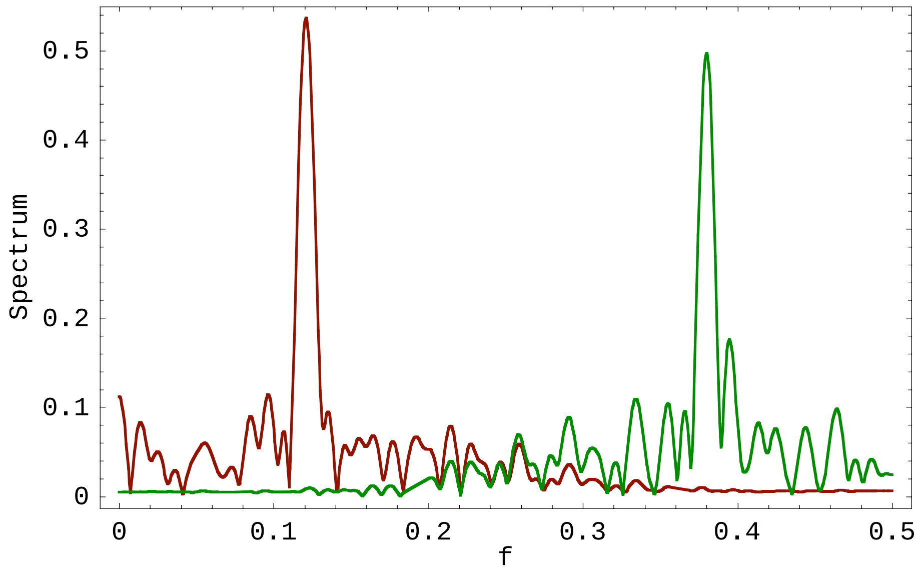
```
SequenceFourierTransformMagnitudePlot[noisyInpSeq, {0, 0.5},
 Frame → True, FrameLabel → { "f", "Spectrum"}];
```



`SequenceFourierTransformMagnitudePlot` shows two strong peaks at frequencies 0.12 and 0.38.

The system has two outputs, so `SequenceFourierTransformMagnitudePlot` shows two spectra:

```
SequenceFourierTransformMagnitudePlot[noisyOutSeq, {0, 0.5},
 Frame → True, FrameLabel -> { "f", "Spectrum"}];
```



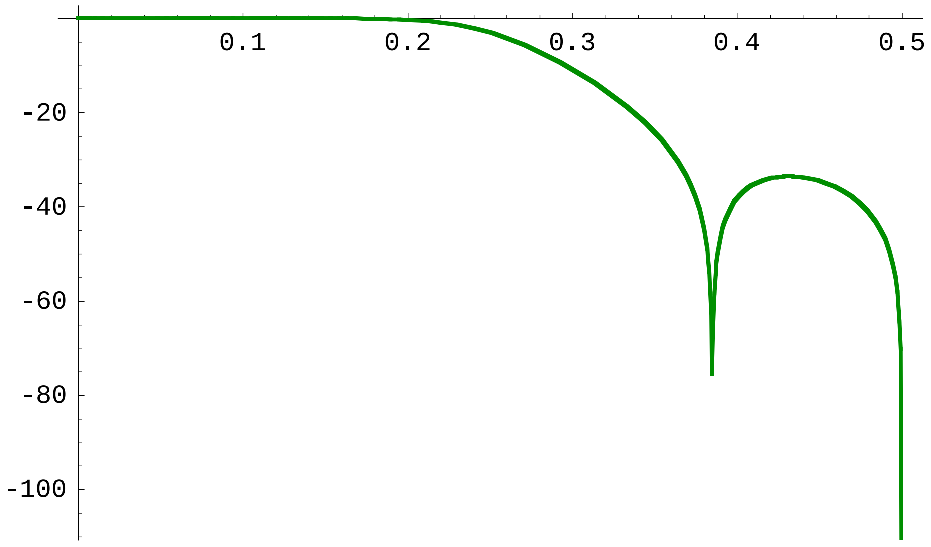
The first spectrum has a strong peak at frequency 0.12 and the second spectrum has a strong peak at frequency 0.38. This means that the system separates the signal components: (a) the high-frequency components appear at the second output, while (b) the low-frequency components appear at the first output.

The same conclusion follows from the plots of the magnitude response.

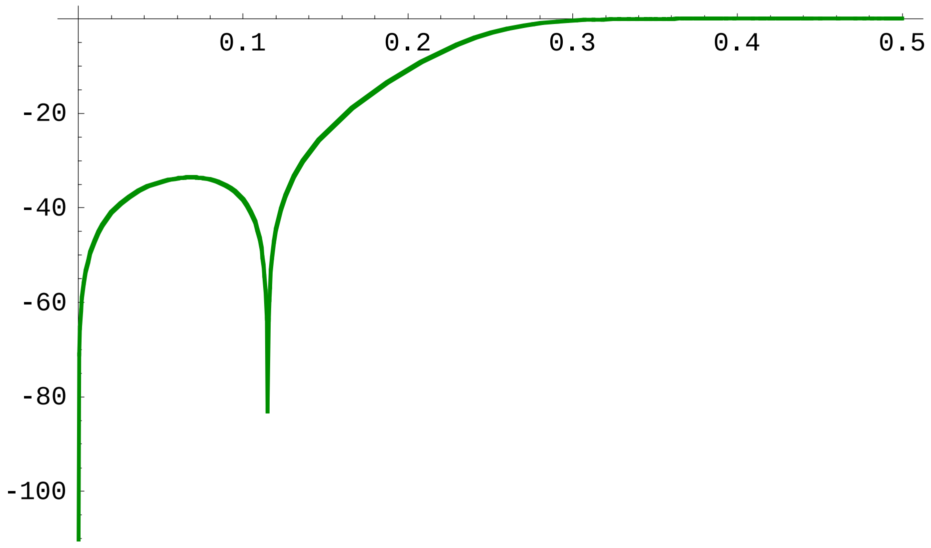
```
{{myTF1, myTF2}, systemInp, systemOut} = DiscreteSystemTransferFunction[mySystem]
```

$$\left\{\left\{\left\{-\frac{-2-5z-5z^2-2z^3}{2z\,(2+5z^2)}\right\},\left\{-\frac{-2+5z-5z^2+2z^3}{2z\,(2+5z^2)}\right\}\right\},\{Y[\{0,8\}]\},\{Y[\{9,8\}],Y[\{9,0\}]\}\right\}$$

```
DiscreteSystemMagnitudeResponsePlot[myTF1];
```



```
DiscreteSystemMagnitudeResponsePlot[myTF2];
```



## 15.9. Processing for Given Transfer Functions

Consider a SISO system generated from the *SchematicSolver*'s album, find its transfer function, and process some data samples.

`TransposedDirectForm2IIRFilterSchematic` creates the schematic of an example SISO system.

```
{exampleSISOSchematic, {inpCoord}, {outCoord}} =
  TransposedDirectForm2IIRFilterSchematic[{{b0, b1, b2}, {a1, a2}}];
```

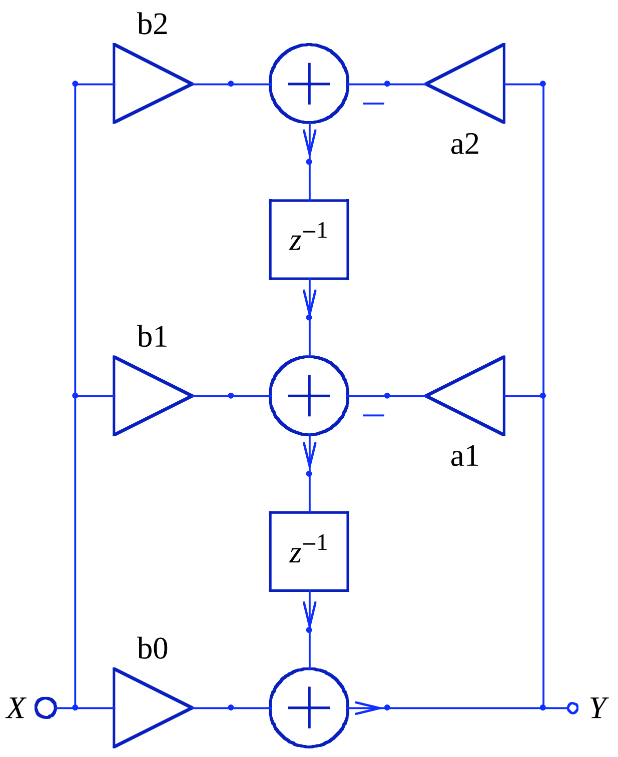
`TransposedDirectForm2IIRFilterSchematic` returns the coordinates of the system input and the system output. You can add the input element and the output element to form the system specification:

```
exampleSISOSystem = Join[exampleSISOSchematic,
  {{"Input", inpCoord, X},
   {"Output", outCoord, Y}}]
```

```
{{Line, {{6, 0}, {4, 0}}}, {Adder, {{2, 0}, {3, -1}, {4, 0}, {3, 1}}, {1, 0, 2, 1}},
 {Multiplier, {{0, 0}, {2, 0}}, b0}, {Line, {{0, 0}, {0, 4}}},
 {Line, {{6, 0}, {6, 4}}}, {Adder, {{2, 4}, {3, 3}, {4, 4}, {3, 5}}, {1, 2, -1, 1}},
 {Delay, {{3, 3}, {3, 1}}, 1}, {Multiplier, {{0, 4}, {2, 4}}, b1},
 {Multiplier, {{6, 4}, {4, 4}}, a1}, {Line, {{0, 4}, {0, 8}}},
 {Line, {{6, 4}, {6, 8}}}, {Adder, {{2, 8}, {3, 7}, {4, 8}, {3, 9}}, {1, 2, -1, 0}},
 {Delay, {{3, 7}, {3, 5}}, 1}, {Multiplier, {{0, 8}, {2, 8}}, b2},
 {Multiplier, {{6, 8}, {4, 8}}, a2}, {Input, {0, 0}, X}, {Output, {6, 0}, Y}}
```

`ShowSchematic` draws the schematic of the system:

```
ShowSchematic[exampleSISOSystem, Frame → False, GridLines → None]
```



The transfer function of the system is

```
{tf, systemInp, systemOut} = DiscreteSystemTransferFunction[exampleSISOSystem];
tf // DiscreteSystemDisplayForm
```

$$\frac{\mathrm{b0} + \mathrm{b1}\,z^{-1} + \mathrm{b2}\,z^{-2}}{1 + \mathrm{a1}\,z^{-1} + \mathrm{a2}\,z^{-2}}$$

For the specific values

```
myValues = {a1 → 0.1, a2 → 0.5, b0 → 1, b1 → 1, b2 → 1}
```

```
{a1 → 0.1, a2 → 0.5, b0 → 1, b1 → 1, b2 → 1}
```

the transfer function becomes

```
tfSpecific = tf[[1, 1]] /. myValues // Simplify
```

$$\frac{1 + z + z^2}{0.5 + 0.1\,z + z^2}$$

`DiscreteSystemProcessingSISO` simulates single-input single-output (SISO) systems, and takes up to four arguments:

`DiscreteSystemProcessingSISO[`*inputDataList*, *transferFunction*, *complexVariable*, *initialConditions*`]`

*inputDataList* is a list of data samples.

*transferFunction* is the transfer function of the system.

*complexVariable* is a symbol that represents the complex variable.

*initialConditions* is a list of initial conditions that represent the state of the system.

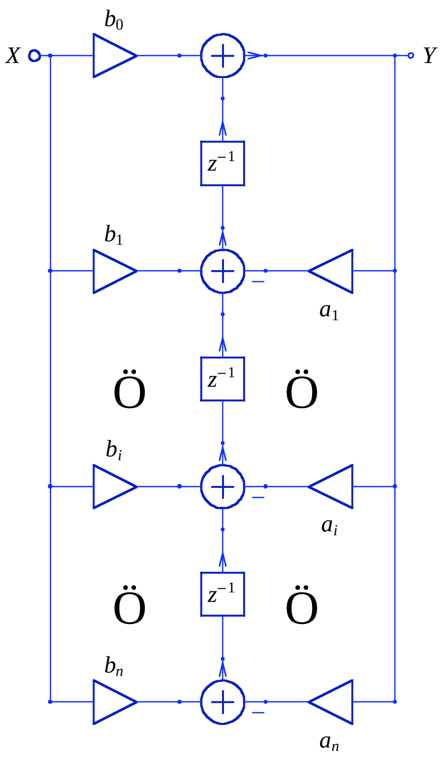
`DiscreteSystemProcessingSISO` returns a list of the form {*outputDataList*, *finalConditions*}.

*outputDataList* is a list of processed data samples.

*finalConditions* is a list of final conditions that represent the final state of the system.

`DiscreteSystemProcessingSISO` has been implemented as a transposed direct form 2 structure as shown below:

```
ShowSchematic[SchematicSolverFigureProcessingTransposedDirectForm2IIR,
  GridLines → None, Frame → False, FontSize → 9];
```



The corresponding transfer function is of the form $H(z) = \frac{b_0+b_1\, z^{-1}+\ldots+b_i\, z^{-i}+\ldots+b_n\, z^{-n}}{1+a_1\, z^{-1}+\ldots+a_i\, z^{-i}+\ldots+a_n\, z^{-n}}$.

Initial conditions are important when we process data in blocks. For example, the input signal can be represented by two or more data lists, and each list is processed individually.

Assume that a unit step signal should be processed by the transposed direct form 2 system with the known transfer function:

```
stepList = UnitStepSequence[32] // SequenceToList
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

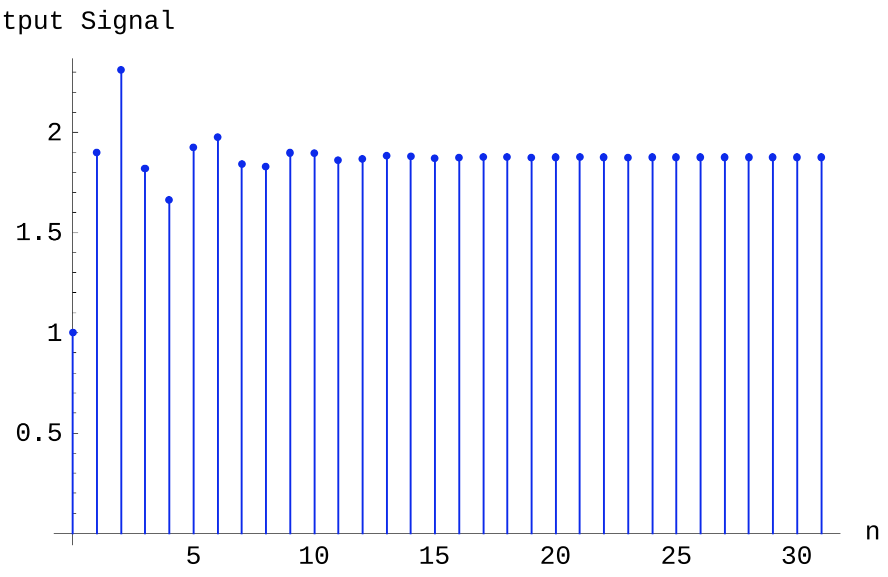
```
tfSpecific
```

$$\frac{1 + z + z^2}{0.5 + 0.1\, z + z^2}$$

The step response is computed with

```
{outputList, finalConditions} = DiscreteSystemProcessingSISO[stepList, tfSpecific];
```

```
SequencePlot[ListToSequence[outputList], AxesLabel → {"n", "Output Signal"}];
```



`DiscreteSystemProcessingSISO` works with causal systems, only.

# 16. Post-Processing with *Mathematica* Built-in Functions

## 16.1. Introduction

*Mathematica* has a rich collection of powerful functions and packages that can be efficiently used for analyzing systems.

We use *SchematicSolver* to draw schematics of systems and to symbolically compute the system transfer function directly from schematics.

We can use *Mathematica* built-in functions and standard packages for post-processing results returned by *Schematic-Solver* to

- a) compute the impulse response with the built-in function `InverseZTransform`
- b) compute the output signal by multiplying the input signal transform by the transfer function and by taking the inverse *z*-transform of the product (if the input signal is represented as a formula, and if the input signal transform can be computed with the built-in function `ZTransform`)
- c) compute the impulse response and the output signal as a list of numbers with the built-in function `ListConvolve` (if the input signal is represented as a data list)
- d) process signals with noise using `ListConvolve`
- e) plot discrete signals represented by formulas

If packages have not already been loaded, we load them with

```
Needs["Graphics`"];
Needs["SchematicSolver`"];
```

We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[InputNotebook[], ImageSize → {350, 300}];
SetOptions[ShowSchematic, ElementScale → 1,
  FontSize → Automatic, Frame → True, GridLines → Automatic, PlotRange → All];
SetOptions[DrawElement, ElementSize → {1, 1},
  PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
    {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
  ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
  TextStyle → {FontFamily → Times, FontSize → 12}];
```

## 16.2. Drawing and Solving Systems with *SchematicSolver*

### Draw a System Using *SchematicSolver*

Consider a discrete-time system and find the transfer function using *SchematicSolver*.

`TransposedDirectForm2IIRFilterSchematic` creates the schematic of an example system:

```
{exampleSchematic, {inpCoord}, {outCoord}} =
  TransposedDirectForm2IIRFilterSchematic[{{b0, b1, b2}, {a1, a2}}];
```

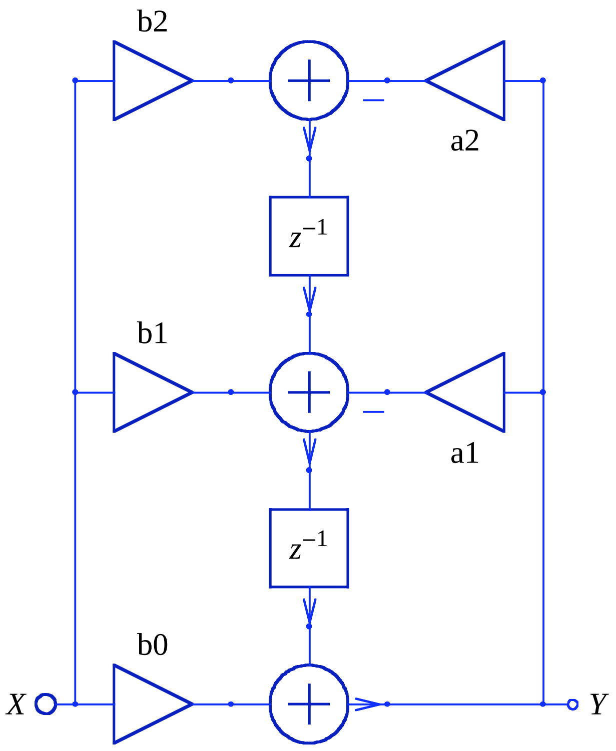
`TransposedDirectForm2IIRFilterSchematic` returns the coordinates of the system input and the system output. You can add the input element and the output element to form the system specification:

```
exampleSystem = Join[exampleSchematic,
  {{"Input", inpCoord, X},
   {"Output", outCoord, Y}}]
```

```
{{Line, {{6, 0}, {4, 0}}}, {Adder, {{2, 0}, {3, -1}, {4, 0}, {3, 1}}, {1, 0, 2, 1}},
 {Multiplier, {{0, 0}, {2, 0}}, b0}, {Line, {{0, 0}, {0, 4}}},
 {Line, {{6, 0}, {6, 4}}}, {Adder, {{2, 4}, {3, 3}, {4, 4}, {3, 5}}, {1, 2, -1, 1}},
 {Delay, {{3, 3}, {3, 1}}, 1}, {Multiplier, {{0, 4}, {2, 4}}, b1},
 {Multiplier, {{6, 4}, {4, 4}}, a1}, {Line, {{0, 4}, {0, 8}}},
 {Line, {{6, 4}, {6, 8}}}, {Adder, {{2, 8}, {3, 7}, {4, 8}, {3, 9}}, {1, 2, -1, 0}},
 {Delay, {{3, 7}, {3, 5}}, 1}, {Multiplier, {{0, 8}, {2, 8}}, b2},
 {Multiplier, {{6, 8}, {4, 8}}, a2}, {Input, {0, 0}, X}, {Output, {6, 0}, Y}}
```

`ShowSchematic` draws the schematic of the system:

```
ShowSchematic[exampleSystem, Frame → False, GridLines → None]
```



## Find Transfer Function Using *SchematicSolver*

`DiscreteSystemTransferFunction` finds the transfer function of the example system:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[exampleSystem]
```

$$\left\{\left\{\left\{-\frac{-b2 - b1\,z - b0\,z^2}{a2 + a1\,z + z^2}\right\}\right\}, \{Y[\{0, 0\}]\}, \{Y[\{6, 0\}]\}\right\}$$

The transfer function of this system is the first element of `tfMatrix`:

```
tf = tfMatrix[[1, 1]] // Together
```

$$\frac{b2 + b1\,z + b0\,z^2}{a2 + a1\,z + z^2}$$

The symbol *z* is reserved for the complex variable in the *z*-transform domain.

For the specific values

```
myValues = {a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9}
```

```
{a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9}
```

we obtain the transfer function of the system as

```
tfSpecific = tf /. myValues // Simplify
```

$$\frac{0.9 + 0.8\,z + 0.7\,z^2}{0.95 + 0.85\,z + z^2}$$

which can be displayed with `DiscreteSystemDisplayForm` in the traditional form:

```
tfSpecific // DiscreteSystemDisplayForm
```

$$\frac{0.7 + 0.8\, z^{-1} + 0.9\, z^{-2}}{1 + 0.85\, z^{-1} + 0.95\, z^{-2}}$$

## 16.3. Processing Using Built-in Functions

### Introduction

*Mathematica* is a very rich environment for the mixed symbolic-numeric computation needed in signal processing.

In symbolic processing, the signal is represented in a computer as a formula, rather than as a sequence of numbers. Thus, the value of a signal might only be known in terms of a formula, instead of a number. In a similar manner, signal-processing operators, the building blocks for systems, are maintained in symbolic form. This enables a computer to simplify, rearrange, and rewrite symbolic expressions until they take a desired form. When one of the operators is applied to a function, no evaluation takes place; the resulting function is stored in the symbolic form until it becomes convenient to compute it explicitly.

### Representing Signals and Systems by Formulas and Operators

Consider the transfer function returned by *SchematicSolver* and assign some numeric values to the parameters:

```
tfNumeric = tf /. {b0 → 1, b1 → 2, b2 → 1, a1 → 0, a2 → 1 / 2};
% // DiscreteSystemDisplayForm
```

$$\frac{2 + 4\, z^{-1} + 2\, z^{-2}}{2 + z^{-2}}$$

Impulse response can be computed with the built-in function `InverseZTransform`:

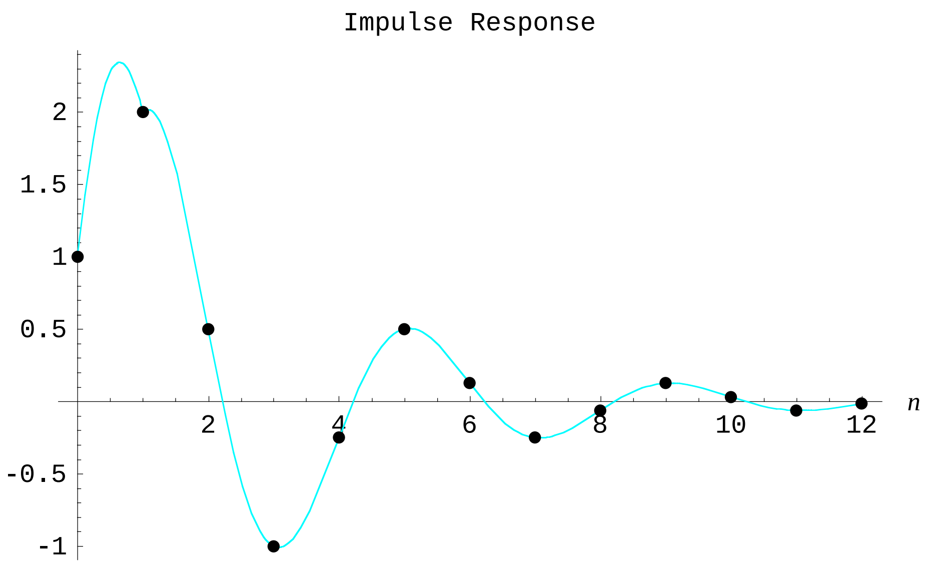
```
impulseResponse = InverseZTransform[tfNumeric, z, n] // FullSimplify
```

$$2^{-n/2} \left(2\sqrt{2}\, \mathrm{Sin}\left[\frac{n\pi}{2}\right] + \mathrm{Cos}\left[\frac{n\pi}{2}\right] (1 - 2\, \mathrm{UnitStep}[-1 + n])\right)$$

We define a function to plot discrete signals represented by formulas:

```
Clear[PlotDiscreteSignal]
PlotDiscreteSignal[y_,
  {n_Symbol, n1_Integer: 0, n2_Integer: 1}, label_: "Discrete Signal"] :=
 Module[{t}, t = Table[{n, y}, {n, n1, n2}]; DisplayTogether[
   Plot[y, {n, n1, n2}, PlotStyle → Hue[0.5],
    PlotLabel → label, AxesLabel → {n // TraditionalForm, ""}],
   ListPlot[t, PlotStyle → {PointSize[0.015]}, PlotRange → All]]]
```

```
PlotDiscreteSignal[impulseResponse, {n, 0, 12}, "Impulse Response"];
```



Here is another example:

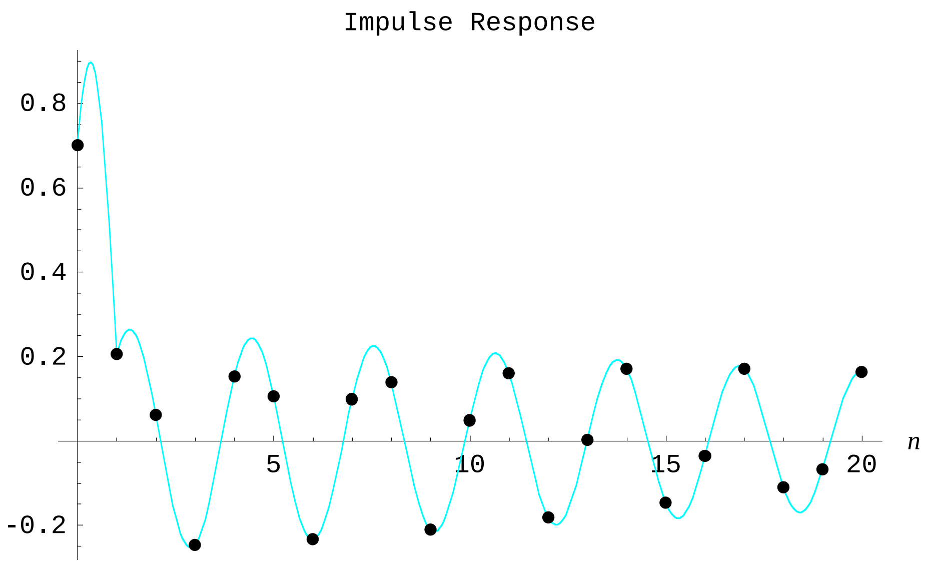
```
tfNumeric2 = tf /. {a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9};
% // DiscreteSystemDisplayForm
```

$$\frac{0.7 + 0.8\, z^{-1} + 0.9\, z^{-2}}{1 + 0.85\, z^{-1} + 0.95\, z^{-2}}$$

```
impulseResponse2 =
 InverseZTransform[tfNumeric2, z, n] // Re // ComplexExpand // FullSimplify
```

```
-0.947368 0.974679^n (-0.738889 Cos[2.02199 n] - 0.604711 Sin[2.02199 n] +
   (1. Cos[2.02199 n] + 0.484529 Sin[2.02199 n]) UnitStep[-1 + n])
```

```
PlotDiscreteSignal[impulseResponse2, {n, 0, 20}, "Impulse Response"];
```



## Processing Using ZTransform

Consider two discrete sinusoidal signals in terms of the amplitude X, the angular digital frequency w and the phase $\phi$:

```
w1 = 6 / 10;
X1 = 1 / 3;
ϕ1 = π / 3;
sineSignal1 = X1 * Sin[w1 * n + ϕ1]
```

$$\frac{1}{3}\,\mathrm{Sin}\left[\frac{3\,n}{5}+\frac{\pi}{3}\right]$$

```
w2 = 2;
X2 = 1 / 5;
ϕ2 = π / 5;
sineSignal2 = X2 * Sin[w2 * n + ϕ2]
```

$$\frac{1}{5}\,\mathrm{Sin}\left[2\,n+\frac{\pi}{5}\right]$$

We can generate an input signal as a sum of the two signals

```
inpSignal = sineSignal1 + sineSignal2
```

$$\frac{1}{5}\,\mathrm{Sin}\left[2\,n+\frac{\pi}{5}\right]+\frac{1}{3}\,\mathrm{Sin}\left[\frac{3\,n}{5}+\frac{\pi}{3}\right]$$

The transforms of the signals are

```
sineTransform1 = ZTransform[sineSignal1, n, z] // FullSimplify
```

$$-\frac{e^{\frac{3i}{5}}\,z\,\left(\sqrt{3}\,\left(z-\mathrm{Cos}\left[\frac{3}{5}\right]\right)+\mathrm{Sin}\left[\frac{3}{5}\right]\right)}{6\left(e^{\frac{3i}{5}}-z\right)\left(-1+e^{\frac{3i}{5}}\,z\right)}$$

```
sineTransform2 = ZTransform[sineSignal2, n, z] // FullSimplify
```

$$-\frac{e^{2i}\,z\left(\sqrt{10-2\sqrt{5}}\,\left(z-\mathrm{Cos}[2]\right)+\left(1+\sqrt{5}\right)\mathrm{Sin}[2]\right)}{20\left(e^{2i}-z\right)\left(-1+e^{2i}\,z\right)}$$

```
inpTransform = sineTransform1 + sineTransform2
```

$$-\frac{e^{\frac{3i}{5}}\,z\,\left(\sqrt{3}\,\left(z-\mathrm{Cos}\left[\frac{3}{5}\right]\right)+\mathrm{Sin}\left[\frac{3}{5}\right]\right)}{6\left(e^{\frac{3i}{5}}-z\right)\left(-1+e^{\frac{3i}{5}}\,z\right)}-\frac{e^{2i}\,z\left(\sqrt{10-2\sqrt{5}}\,\left(z-\mathrm{Cos}[2]\right)+\left(1+\sqrt{5}\right)\mathrm{Sin}[2]\right)}{20\left(e^{2i}-z\right)\left(-1+e^{2i}\,z\right)}$$

The corresponding output signals are computed by

(1) multiplying the input signal transform by the transfer function, and

(2) taking the inverse transform of the product.

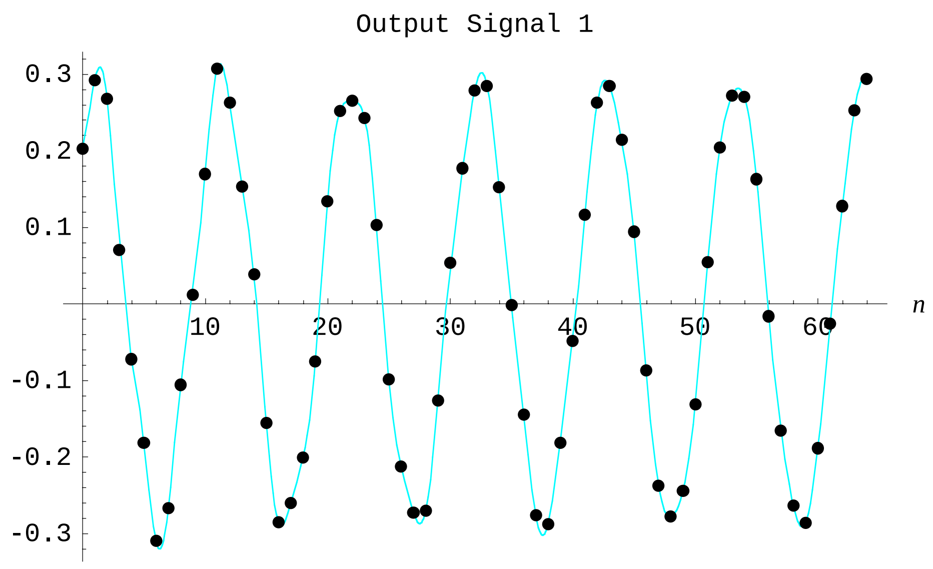
```
tfSpecific = tf /. {a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9};
% // DiscreteSystemDisplayForm
```

$$\frac{0.7+0.8\,z^{-1}+0.9\,z^{-2}}{1+0.85\,z^{-1}+0.95\,z^{-2}}$$

```
outTransform1 = sineTransform1 * tfSpecific;
outTransform2 = sineTransform2 * tfSpecific;
outTransform = inpTransform * tfSpecific;
```

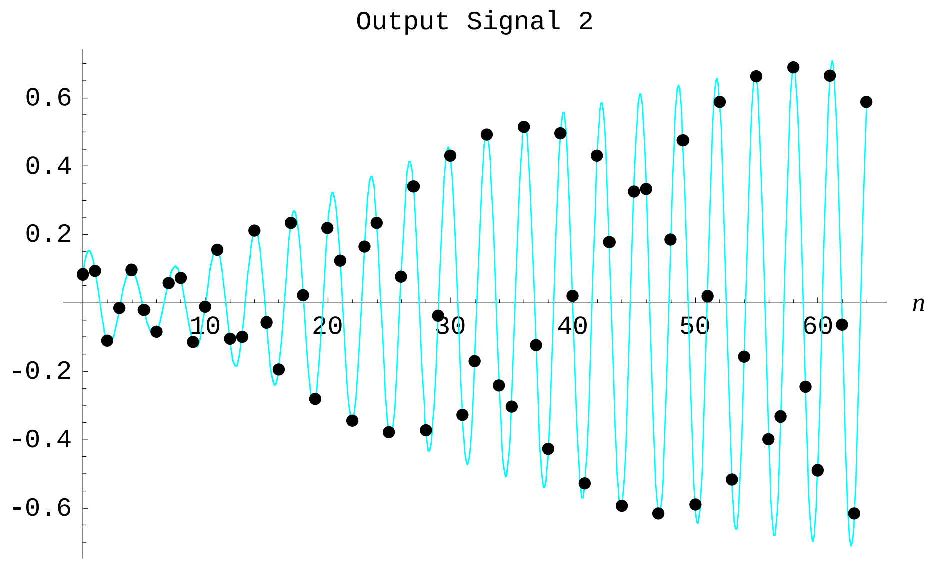
```
outSignal1 = InverseZTransform[outTransform1, z, n] // Re // ComplexExpand // Simplify
PlotDiscreteSignal[outSignal1, {n, 0, 64}, "Output Signal 1"];
```

$0.239414\,1.^n \operatorname{Cos}[0.6\,n] - 0.0373415\,0.974679^n \operatorname{Cos}[2.02199\,n] + 0.159706\,1.^n \operatorname{Sin}[0.6\,n] - 0.0134674\,0.974679^n \operatorname{Sin}[2.02199\,n]$



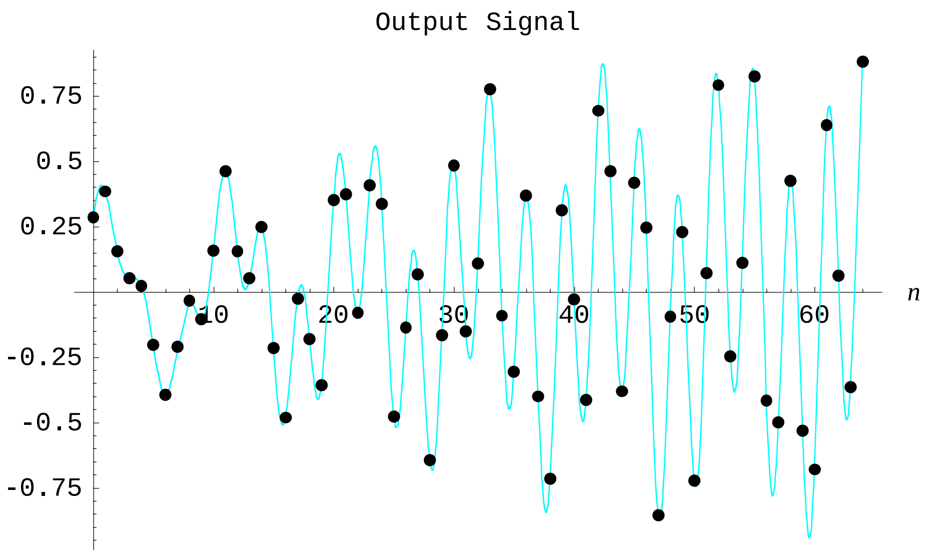
```
outSignal2 = InverseZTransform[outTransform2, z, n] // Re // ComplexExpand // Simplify
PlotDiscreteSignal[outSignal2, {n, 0, 64}, "Output Signal 2"];
```

$-0.699873\,1.^n \operatorname{Cos}[2.\,n] + 0.782163\,0.974679^n \operatorname{Cos}[2.02199\,n] + 0.292561\,1.^n \operatorname{Sin}[2.\,n] - 0.150503\,0.974679^n \operatorname{Sin}[2.02199\,n]$



```
outSignal = InverseZTransform[outTransform, z, n] // Re // ComplexExpand // Simplify
PlotDiscreteSignal[outSignal, {n, 0, 64}, "Output Signal"];
```

$0.239414\, 1.^{n} \mathrm{Cos}[0.6\, n] - 0.699873\, 1.^{n} \mathrm{Cos}[2.\, n] + 0.744821\, 0.974679^{n} \mathrm{Cos}[2.02199\, n] + 0.159706\, 1.^{n} \mathrm{Sin}[0.6\, n] + 0.292561\, 1.^{n} \mathrm{Sin}[2.\, n] - 0.16397\, 0.974679^{n} \mathrm{Sin}[2.02199\, n]$

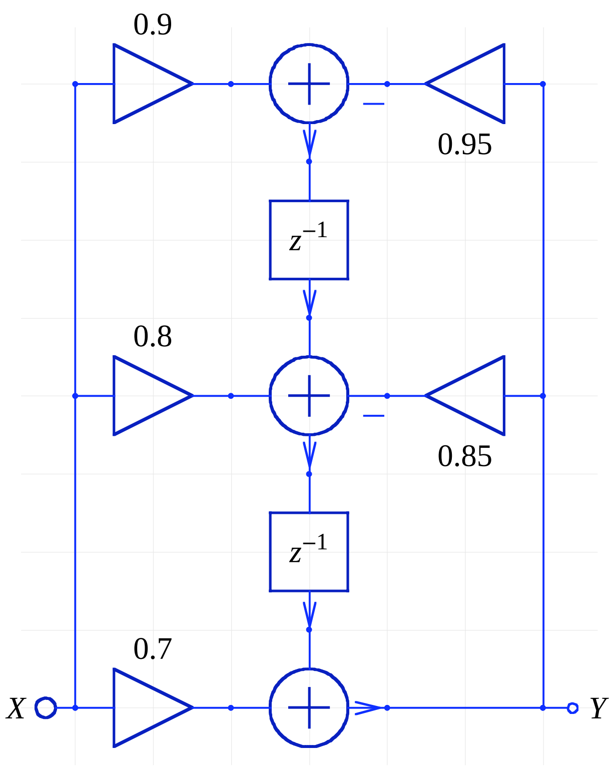


## Processing Using ListConvolve

Let us redraw the schematic of the system with the specific coefficients:

```
myValues = {a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9};
```

```
ShowSchematic[exampleSystem /. myValues, Frame → False]
```



The corresponding transfer function is

```
{tfMatrix, systemInp, systemOut} =
  DiscreteSystemTransferFunction[exampleSystem /. myValues];

tf = tfMatrix[[1, 1]];
% // DiscreteSystemDisplayForm
```

$$\frac{0.7 + 0.8\, z^{-1} + 0.9\, z^{-2}}{1. + 0.85\, z^{-1} + 0.95\, z^{-2}}$$

and its impulse response is of the form

```
impulseResponse = InverseZTransform[tf, z, n] // Re // ComplexExpand // Simplify
```

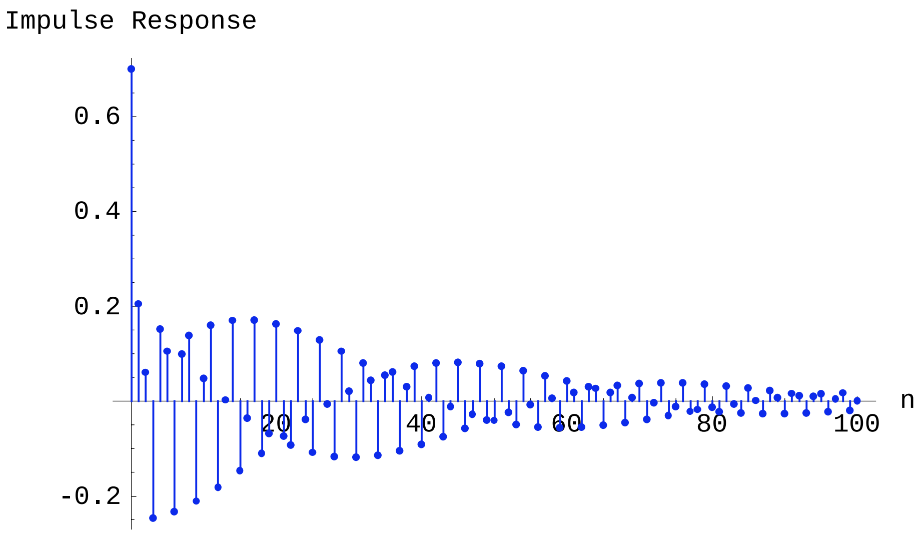
```
-0.947368 0.974679^n (-0.738889 Cos[2.02199 n] - 0.604711 Sin[2.02199 n] +
   (1. Cos[2.02199 n] + 0.484529 Sin[2.02199 n]) UnitStep[-1 + n])
```

We find the first 100 samples of the impulse response with

```
impulseResponseList = Table[impulseResponse, {n, 0, 100}];
```

*SchematicSolver*'s function `SequencePlot` plots discrete signals represented by sequences in matrix form. A list of values can be converted to a sequence with `ListToSequence`:

```
SequencePlot[ListToSequence[impulseResponseList],
  AxesLabel → {"n", "Impulse Response"}];
```



`ListConvolve[impulseResponseList, inputSignalList]` is a built-in function that gives the convolution of the list `impulseResponseList` with the list `inputSignalList`. For a given input signal as a list of numbers

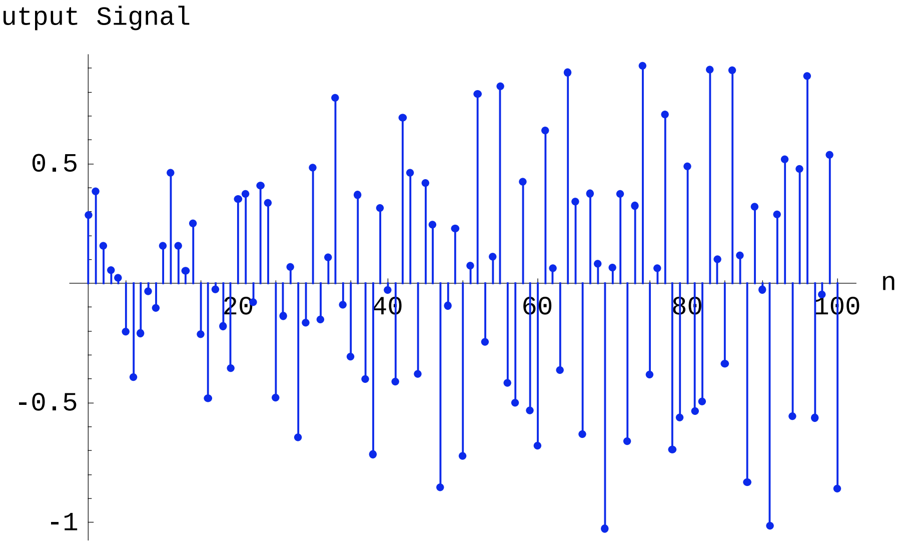
```
inpList = Table[inpSignal, {n, 0, 100}];

paddedInpList = PadLeft[inpList, Length[inpList] + Length[impulseResponseList] - 1];
```

and for the known impulse response, we can compute the output signal as a list of numbers:

```
outList = ListConvolve[impulseResponseList, paddedInpList];
```

`SequencePlot` plots the signal:

```
SequencePlot[ListToSequence[outList],
  AxesLabel→{"n", "Output Signal"}];
```



# 17. Post-Processing Using Control System Professional

## 17.1. Introduction

*Control System Professional* is a collection of *Mathematica* programs that extend *Mathematica* to solve a wide range of control system problems. Both classic and modern approaches are supported for continuous-time (analog) and discrete-time (sampled) systems.

We use *SchematicSolver* to draw schematics of systems and to symbolically compute the system transfer function directly from schematics.

We can use *Control System Professional* for post-processing results returned by *SchematicSolver* to:

- a) find the state-space realization with the function `StateSpace`
- b) simplify and find a more convenient presentation of the continuous-time system
- c) represent the state-space realization of the MIMO system in the traditional typeset form with `TraditionalForm`
- d) display the state-space objects as the familiar state-space equations with `EquationForm`
- e) compute the discrete-time approximation of the continuous-time system
- f) discretize the state-space system representation
- g) display the state-space equations as difference equations (for the discretized system)
- h) find the zeros, poles, and gains with the function `ZeroPoleGain`
- i) represent the transfer function in factored form with the function `FactorRational`
- j) compute the output signal of the continuous system with the function `OutputResponse`
- k) compute the state response with the function `StateResponse`

If packages have not already been loaded, we load *Control System Professional* and *SchematicSolver* with

```
Needs["ControlSystems`"];
Needs["SchematicSolver`"];
```

We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[InputNotebook[], ImageSize → {350, 300}];
SetOptions[ShowSchematic, ElementScale → 1,
  FontSize → Automatic, Frame → True, GridLines → Automatic, PlotRange → All];
SetOptions[DrawElement, ElementSize → {1, 1},
  PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
    {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
  ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
  TextStyle → {FontFamily → Times, FontSize → 12}];
```

## 17.2. Drawing and Solving Systems Using *SchematicSolver*

Consider a simple integrator system, see Section 1.3 of *Control System Professional*, and find the transfer function using *SchematicSolver*.

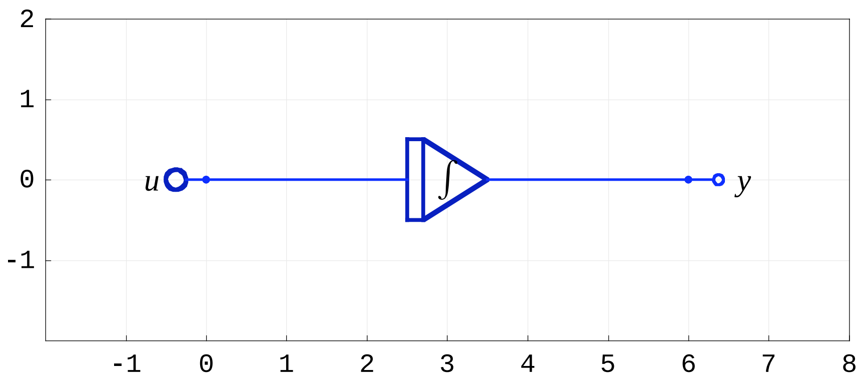
We describe the integrator system as a list of continuous-time elements that contains an input, an integrator, and an output:

```
integratorSystem = {
  {"Input", {0, 0}, u}, {"Output", {6, 0}, y},
  {"Integrator", {{0, 0}, {6, 0}}, 1}}
```

```
{{Input, {0, 0}, u}, {Output, {6, 0}, y}, {Integrator, {{0, 0}, {6, 0}}, 1}}
```

`ShowSchematic` draws the schematic:

```
ShowSchematic[integratorSystem, PlotRange → {{-2, 8}, {-2, 2}}];
```



`ContinuousSystemTransferFunction` finds the transfer function:

```
{myTF, myInp, myOut} = ContinuousSystemTransferFunction[integratorSystem]
```

$\{\{\{\frac{1}{s}\}\}, \{Y[\{0, 0\}]\}, \{Y[\{6, 0\}]\}\}$

`ContinuousSystemTransferFunction` returns a list of the form {*transferFunctionMatrix*, *systemInputs*, *systemOutputs*}. *transferFunctionMatrix* is the transfer function matrix of the system. *systemInputs* is a list of symbols that represent the system inputs. *systemOutputs* is a list of symbols that represent the system outputs.

The symbol *s* is reserved for the complex frequency in the Laplace transform domain.

## 17.3. Processing Systems Using *Control System Professional*

In *Control System Professional*, the integrator system is created in the transfer function form as follows:

```
myTFobject = TransferFunction[s, myTF]
```

$\text{TransferFunction}[s, \{\{\frac{1}{s}\}\}]$

By applying the `StateSpace` function, we find the state-space realization of the transfer function object `myTFobject`:

```
StateSpace[myTFobject]
% // TraditionalForm
```

```
StateSpace[{{0}}, {{1}}, {{1}}]
```

$$\left(\begin{array}{c|c} 0 & 1 \\ \hline 1 & 0 \end{array}\right)^{\mathcal{S}}.$$

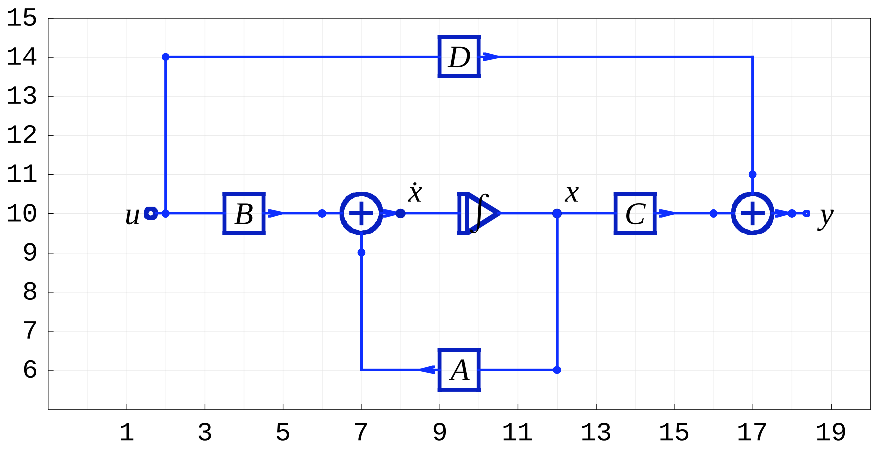
## 17.4. Drawing and Solving State-Space Models with *SchematicSolver*

The continuous-time state-space system is described by the set of equations

$$\begin{aligned} \dot{x}(t) &= A\,x(t) + B\,u(t) \\ y(t) &= C\,x(t) + D\,u(t) \end{aligned}$$

The schematic of a continuous-time state-space system is

```
myStateSpace = {
   {"Input", {2, 10}, u},
   {"Block", {{2, 10}, {6, 10}}, B},
   {"Adder", {{6, 10}, {7, 9}, {8, 10}, {7, 11}}, {1, 1, 2, 0}},
   {"Integrator", {{8, 10}, {12, 10}}, 1},
   {"Block", {{12, 10}, {16, 10}}, C},
   {"Adder", {{16, 10}, {17, 9}, {18, 10}, {17, 11}}, {1, 0, 2, 1}},
   {"Output", {18, 10}, y},
   {"Block", {{2, 14}, {17, 11}}, D},
   {"Block", {{12, 6}, {7, 9}}, A},
   {"Line", {{2, 10}, {2, 14}}},
   {"Node", {8, 10}, xp}, {"Node", {12, 10}, x},
   {"Line", {{12, 10}, {12, 6}}}
  };
ShowSchematic[% /. {xp → ẋ}, PlotRange → {{-1, 20}, {5, 15}}];
```



`ContinuousSystemTransferFunction` finds the transfer function:

```
{myTF, myInp, myOut} = ContinuousSystemTransferFunction[myStateSpace]
```

$$\left\{\left\{\left\{-\frac{B\,C - A\,D + D\,s}{A - s}\right\}\right\}, \{Y[\{2, 10\}]\}, \{Y[\{18, 10\}]\}\right\}$$

For the specific values

```
myValues = {A → 0, B → 1, C → 1, D → 0}
```

```
{A → 0, B → 1, C → 1, D → 0}
```

we obtain the transfer function of the simple integrator system

```
myTFintegrator = myTF /. myValues
```

$$\left\{\left\{\frac{1}{s}\right\}\right\}$$

## 17.5. Processing State-Space Models with *Control System Professional*

In *Control System Professional*, the system described by `myStateSpace` can represent the state-space object

```
mySSobject = StateSpace[{{A}}, {{B}}, {{C}}, {{D}}]
TraditionalForm[mySSobject]
```

```
StateSpace[{{A}}, {{B}}, {{C}}, {{D}}]
```

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)^{\mathcal{S}}_{\bullet}$$

For the specific values, the above expression simplifies to

```
TraditionalForm[mySSobject] /. myValues
```

$$\left(\begin{array}{c|c} 0 & 1 \\ \hline 1 & 0 \end{array}\right)^{\mathcal{S}}_{\bullet}$$

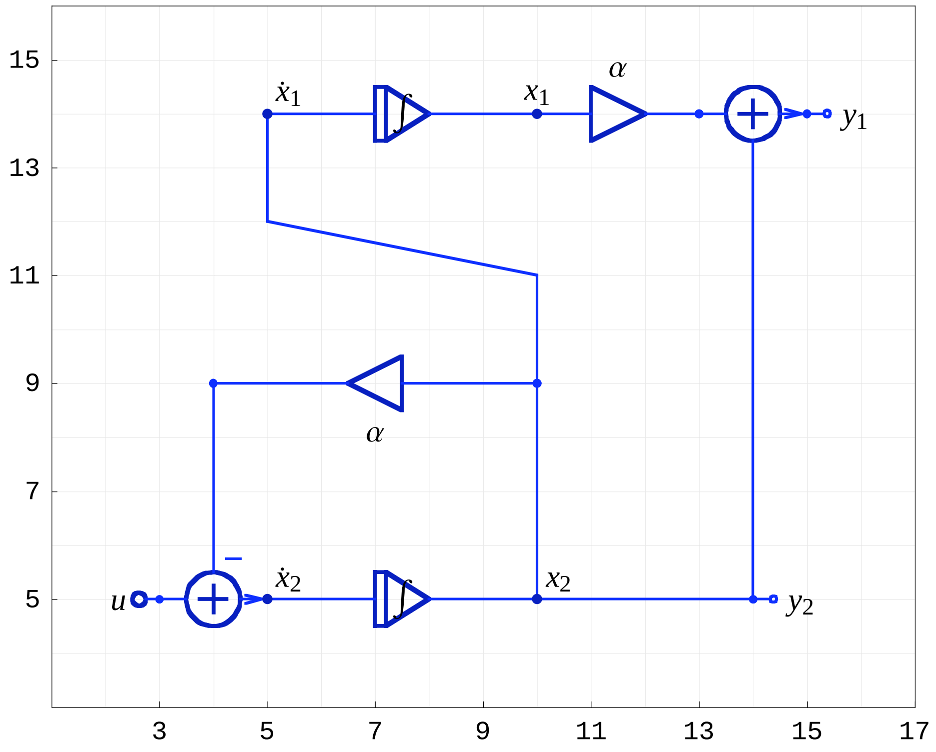
that represents the state-space model of the integrator.

## 17.6. Drawing and Solving 1-Input 2-Output Systems with *SchematicSolver*

Let us draw a single-input two-output system, see Section 1.4 of *Control System Professional*, and find its transfer function using *SchematicSolver*.

```
my1in2outSystem = {
   {"Integrator", {{5, 5}, {10, 5}}, 1},
   {"Integrator", {{5, 14}, {10, 14}}, 1},
   {"Amplifier", {{10, 9}, {4, 9}}, a1},
   {"Amplifier", {{10, 14}, {13, 14}}, a2},
   {"Adder", {{3, 5}, {4, 4}, {5, 5}, {4, 9}}, {1, 0, 2, -1}},
   {"Adder", {{13, 14}, {14, 5}, {15, 14}, {14, 15}}, {1, 1, 2, 0}},
   {"Line", {{10, 9}, {10, 5}}},
   {"Line", {{10, 9}, {10, 11}, {5, 12}, {5, 14}}},
   {"Line", {{14, 5}, {10, 5}}},
   {"Node", {5, 14}, xp1},
   {"Node", {10, 14}, x1, "", TextOffset → {0, -1}},
   {"Node", {5, 5}, xp2},
   {"Node", {10, 5}, x2},
   {"Input", {3, 5}, u},
   {"Output", {15, 14}, y1},
   {"Output", {14, 5}, y2}};
```

```
ShowSchematic[my1in2outSystem /. {a1 → α, a2 → α, x1 → x₁, x2 → x₂,
   xp1 → ẋ₁, xp2 → ẋ₂, y1 → y₁, y2 → y₂},
  PlotRange → {{1, 17}, {3, 16}}];
```



Given the values

```
myValues = {a1 → α, a2 → α}
```

```
{a1 → α, a2 → α}
```

`ContinuousSystemTransferFunction` finds the transfer function

```
{my1in2outTF, myInp, myOut} =
  ContinuousSystemTransferFunction[my1in2outSystem] /. myValues
```

$\{\{\{\frac{1}{s}\}, \{\frac{1}{s+\alpha}\}\}, \{Y[\{3, 5\}]\}, \{Y[\{15, 14\}], Y[\{14, 5\}]\}\}$

From the schematic, we see that `Y[{15,14}]` represents the first output denoted by $y_1$, `Y[{14,5}]` represents the second output denoted by $y_2$, and `Y[{3,5}]` represents input denoted by $u$. Here are the corresponding transfer functions:

```
{my1in2outTF, myOut} // Transpose // TableForm
```

| | |
|---|---|
| $\frac{1}{s}$ | Y[{15, 14}] |
| $\frac{1}{s+\alpha}$ | Y[{14, 5}] |

## 17.7. Processing 1-Input 2-Output Systems with *Control System Professional*

In *Control System Professional*, the single-input two-output system is created in the transfer function form as follows:

```
my1in2outTFobject = TransferFunction[s, my1in2outTF]
```

$$\text{TransferFunction}\left[s, \left\{\left\{\frac{1}{s}\right\}, \left\{\frac{1}{s+\alpha}\right\}\right\}\right]$$

The transfer function object can be represented in the traditional typeset form:

```
TraditionalForm[my1in2outTFobject]
```

$$\begin{pmatrix} \frac{1}{s} \\ \frac{1}{s+\alpha} \end{pmatrix}^{\mathcal{T}}$$

The transfer function object is assumed to be in the continuous-time domain and the variable *s* is used. The superscripted letter $\mathcal{T}$ distinguishes the result from a regular matrix.

The state-space realization of this system can be represented in the `TraditionalForm`:

```
my1in2outSSobject = StateSpace[my1in2outTFobject];
% // TraditionalForm
```

$$\left(\begin{array}{cc|c} 0 & 1 & 0 \\ 0 & -\alpha & 1 \\ \hline \alpha & 1 & 0 \\ 0 & 1 & 0 \end{array}\right)^{\mathcal{S}}_{\bullet}$$

The superscripted letter $\mathcal{S}$ identifies the `StateSpace` object, while the small subscripted bullet character denotes the continuous-time domain.

*Control System Professional* provides the function `EquationForm` that allows you to display the `StateSpace` objects as the familiar state-space equations:

```
EquationForm[StateSpace[my1in2outTFobject]]
```

$$\dot{x} = \begin{pmatrix} 0 & 1 \\ 0 & -\alpha \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u$$

$$y = \begin{pmatrix} \alpha & 1 \\ 0 & 1 \end{pmatrix} x$$

## 17.8. Discrete-Time Models of Continuous-Time Systems

The discrete-time approximation of the continuous-time system `my1in2outSystem` sampled with period 3 is

```
my1in2outTFdiscrete =
  ToDiscreteTime[my1in2outTFobject, Sampled → Period[3]] // Simplify;
% // TraditionalForm
```

$$\begin{pmatrix} \frac{3}{z-1} \\ \frac{1-e^{3\alpha}}{\alpha - e^{3\alpha} z \alpha} \end{pmatrix}^{\mathcal{T}}_{3}$$

Note that the `TraditionalForm` of the discretized object is displayed using the variable *z*. The subscript 3 gives the value of the sampling period.

The discretized state-space system represented in `TraditionalForm` is

```
my1in2outSSdiscrete =
  ToDiscreteTime[my1in2outSSobject, Sampled → Period[3]] // Simplify;
% // TraditionalForm
```

$$\left(\begin{array}{cc|c} 1 & \frac{1-e^{-3\alpha}}{\alpha} & \frac{3\alpha+e^{-3\alpha}-1}{\alpha^2} \\ 0 & e^{-3\alpha} & \frac{1-e^{-3\alpha}}{\alpha} \\ \hline \alpha & 1 & 0 \\ 0 & 1 & 0 \end{array}\right)_3^{\mathcal{S}}$$

For the discretized system, the state-space equations are displayed as difference rather than differential equations.

```
EquationForm[my1in2outSSdiscrete]
```

$$x(k+1) = \begin{pmatrix} 1 & \frac{1-e^{-3\alpha}}{\alpha} \\ 0 & e^{-3\alpha} \end{pmatrix} x(k) + \begin{pmatrix} \frac{3\alpha+e^{-3\alpha}-1}{\alpha^2} \\ \frac{1-e^{-3\alpha}}{\alpha} \end{pmatrix} u(k)$$

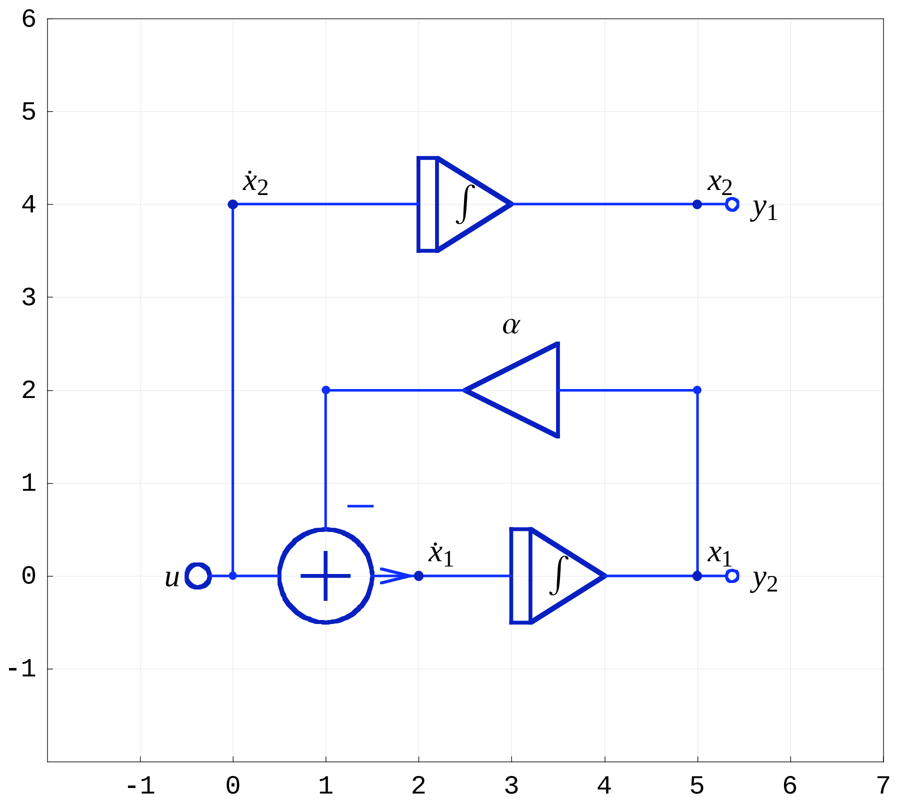
$$y(k) = \begin{pmatrix} \alpha & 1 \\ 0 & 1 \end{pmatrix} x(k)$$

## 17.9. Simplifying Realizations with *SchematicSolver*

Let us find a simpler realization of the previous system `my1in2outSystem`. Consider the following single-input two-output system:

```
my1in2outSimpler =
  {{"Input", {0, 0}, u}, {"Adder", {{0, 0}, {1, -1}, {2, 0}, {1, 2}}, {1, 0, 2, -1}},
   {"Integrator", {{2, 0}, {5, 0}}, 1},
   {"Amplifier", {{5, 2}, {1, 2}}, a, "", TextOffset → {0, 1}},
   {"Line", {{5, 0}, {5, 2}}}, {"Output", {5, 4}, y1}, {"Output", {5, 0}, y2},
   {"Integrator", {{0, 4}, {5, 4}}, 1}, {"Line", {{0, 0}, {0, 4}}}, {"Node", {2, 0}, xp1},
   {"Node", {5, 0}, x1}, {"Node", {0, 4}, xp2}, {"Node", {5, 4}, x2}};
```

```
ShowSchematic[
  my1in2outSimpler /. {a → α, x1 → x₁, x2 → x₂, xp1 → ẋ₁, xp2 → ẋ₂, y1 → y₁, y2 → y₂},
  PlotRange → {{-2, 7}, {-2, 6}}];
```



There is no realization with the number of integrators less than 2, but we reduced the number of adders and amplifiers.

`ContinuousSystemTransferFunction` finds the transfer function:

```
{my1in2outSimplerTF, myInp, myOut} =
  ContinuousSystemTransferFunction[my1in2outSimpler] /. {a → α}
```

$$\left\{\left\{\left\{\frac{1}{s}\right\}, \left\{\frac{1}{s+\alpha}\right\}\right\}, \{Y[\{0, 0\}]\}, \{Y[\{5, 4\}], Y[\{5, 0\}]\}\right\}$$

From the schematic, we see that `Y[{5,4}]` represents the first output denoted by $y_1$, `Y[{5,0}]` represents the second output denoted by $y_2$, and `Y[{0,0}]` represents input denoted by $u$. Here are the corresponding transfer functions:

```
{my1in2outSimplerTF, myOut} // Transpose // TableForm
```

| | |
|---|---|
| $\frac{1}{s}$ | Y[{5, 4}] |
| $\frac{1}{s+\alpha}$ | Y[{5, 0}] |

Verify that the transfer functions of the two systems, `my1in2outSimpler` and `my1in2outSystem`, are the same:

```
SameQ[my1in2outSimplerTF, my1in2outTF]
```

```
True
```

In *Control System Professional*, the single-input two-output system is created in the transfer function form as follows:

```
my1in2outSimplerTFobject = TransferFunction[s, my1in2outSimplerTF]
```

$$\text{TransferFunction}\left[s, \left\{\left\{\frac{1}{s}\right\}, \left\{\frac{1}{s+\alpha}\right\}\right\}\right]$$

Directly from the schematic, we derive the state-space equations of `my1in2outSimpler`:

$$\begin{aligned}\dot{x}_1 &= -\alpha\, x_1 + 0\, x_2 + 1\, u\\ \dot{x}_2 &= 0\, x_1 + 0\, x_2 + 1\, u\\ y_1 &= 0\, x_1 + 1\, x_2 + 0\, u\\ y_2 &= 1\, x_1 + 0\, x_2 + 0\, u\end{aligned}$$

or

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}-\alpha & 0\\ 0 & 0\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} + \begin{pmatrix}1\\ 1\end{pmatrix} u$$
$$\begin{pmatrix}y_1\\ y_2\end{pmatrix} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} + \begin{pmatrix}0\\ 0\end{pmatrix} u$$

The state-space realization of the system `my1in2outSimpler` obtained with the function `StateSpace` is

```
StateSpace[{{-α, 0}, {0, 0}}, {{1}, {1}}, {{0, 1}, {1, 0}}]
% // TraditionalForm
```

```
StateSpace[{{-α, 0}, {0, 0}}, {{1}, {1}}, {{0, 1}, {1, 0}}]
```

$$\left(\begin{array}{cc|c} -\alpha & 0 & 1\\ 0 & 0 & 1\\ \hline 0 & 1 & 0\\ 1 & 0 & 0\end{array}\right)^{\mathcal{S}}.$$

and the corresponding transfer function is obtained with `TransferFunction`

```
TransferFunction[s,
  StateSpace[{{-α, 0}, {0, 0}}, {{1}, {1}}, {{0, 1}, {1, 0}}]] // Simplify
```

$$\text{TransferFunction}\left[s, \left\{\left\{\frac{1}{s}\right\}, \left\{\frac{1}{s+\alpha}\right\}\right\}\right]$$

The schematic of a system, generated by *SchematicSolver*, provides a convenient way to derive the state-space equations of the system.

State-space equations are uniquely defined only if you know the schematic of the system. State-space equations cannot be uniquely determined from the transfer function, that is, by using the `StateSpace` function:

```
StateSpace[my1in2outSimplerTFobject] // TraditionalForm
```

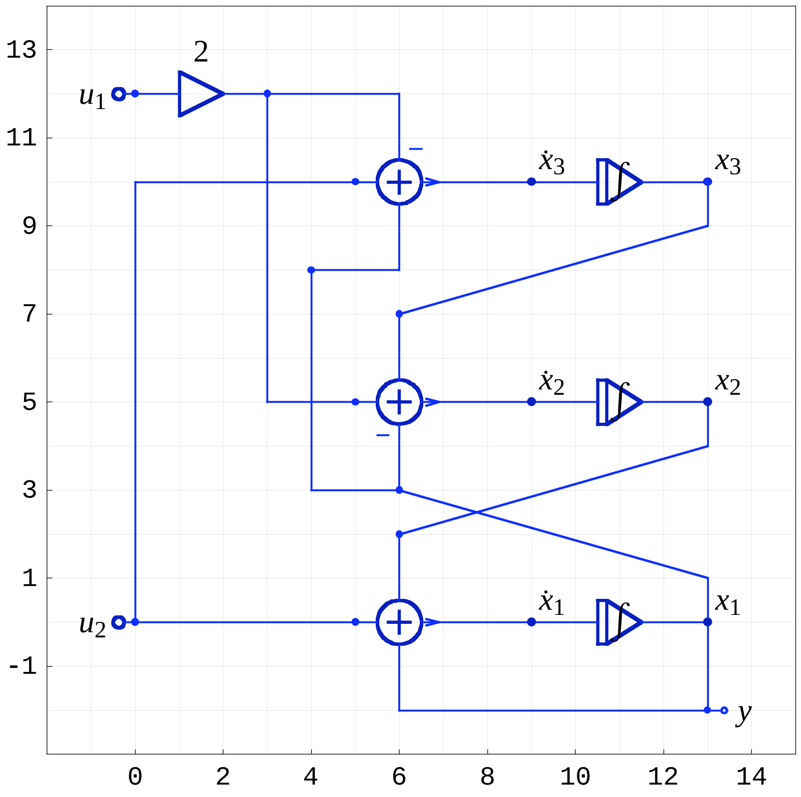
$$\left(\begin{array}{cc|c} 0 & 1 & 0\\ 0 & -\alpha & 1\\ \hline \alpha & 1 & 0\\ 0 & 1 & 0\end{array}\right)^{\mathcal{S}}.$$

The state-space representation that is obtained directly from the transfer function matrix corresponds to the controllable companion form, sometimes referred to as the controllable canonical form.

## 17.10. Drawing and Solving 2-Input 1-Output Systems with *SchematicSolver*

Let us draw a two-input single-output system, see Section 3.2 of *Control System Professional*, and find the transfer function using *SchematicSolver*.

```
my2in1outSystem = {
   {"Integrator", {{9, 10}, {13, 10}}, 1},
   {"Integrator", {{9, 5}, {13, 5}}, 1},
   {"Integrator", {{9, 0}, {13, 0}}, 1},
   {"Adder", {{5, 10}, {4, 8}, {9, 10}, {3, 12}}, {1, 1, 2, -1}},
   {"Adder", {{5, 5}, {6, 3}, {9, 5}, {6, 7}}, {1, -1, 2, 1}},
   {"Adder", {{5, 0}, {13, -2}, {9, 0}, {6, 2}}, {1, 1, 2, 1}},
   {"Input", {0, 12}, u1},
   {"Input", {0, 0}, u2},
   {"Amplifier", {{0, 12}, {3, 12}}, 2},
   {"Line", {{3, 12}, {3, 5}, {5, 5}}},
   {"Line", {{0, 0}, {0, 10}, {5, 10}}},
   {"Line", {{6, 2}, {13, 4}, {13, 5}}},
   {"Line", {{6, 3}, {13, 1}, {13, 0}}},
   {"Line", {{13, -2}, {13, 0}}},
   {"Line", {{0, 0}, {5, 0}}},
   {"Output", {13, -2}, y},
   {"Node", {13, 0}, x1},
   {"Node", {13, 5}, x2},
   {"Node", {13, 10}, x3},
   {"Node", {9, 0}, xp1},
   {"Node", {9, 5}, xp2},
   {"Node", {9, 10}, xp3},
   {"Line", {{6, 7}, {13, 9}, {13, 10}}},
   {"Line", {{6, 3}, {4, 3}, {4, 8}}}
  };
ShowSchematic[my2in1outSystem /.
   {u1 → u₁, u2 → u₂, x1 → x₁, x2 → x₂, x3 → x₃, xp1 → ẋ₁, xp2 → ẋ₂, xp3 → ẋ₃},
  PlotRange → {{-2, 15}, {-3, 14}}];
```



`ContinuousSystemTransferFunction` finds the transfer function:

```
{my2in1outTF, myInp, myOut} =
 ContinuousSystemTransferFunction[my2in1outSystem] // Simplify
```

$$\left\{\left\{\left\{\frac{2}{1+s^2}, \frac{1}{-1+s}\right\}\right\}, \{Y[\{0, 12\}], Y[\{0, 0\}]\}, \{Y[\{13, -2\}]\}\right\}$$

The transfer function matrix of the system is a 1-by-2 matrix

```
MatrixForm[my2in1outTF]
```

$$\begin{pmatrix} \frac{2}{1+s^2} & \frac{1}{-1+s} \end{pmatrix}$$

## 17.11. Processing 2-Input 1-Output Systems with *Control System Professional*

In *Control System Professional*, the two-input single-output system is created in the transfer function form as follows:

```
my2in1outTFobject = TransferFunction[s, my2in1outTF]
```

$$\text{TransferFunction}\left[s, \left\{\left\{\frac{2}{1+s^2}, \frac{1}{-1+s}\right\}\right\}\right]$$

The transfer function object can be represented in the traditional typeset form:

```
TraditionalForm[my2in1outTFobject]
```

$$\begin{pmatrix} \frac{2}{s^2+1} & \frac{1}{s-1} \end{pmatrix}^{\mathcal{T}}$$

`ZeroPoleGain` finds the zeros, poles, and gains:

```
ZeroPoleGain[my2in1outTFobject]
```

```
ZeroPoleGain[s, {{{}, {}}}, {{{-ⅈ, ⅈ}, {1}}}, {{2, 1}}]
```

Notice that there are no finite zeros in the transfer function matrix, so the corresponding list of zeros is empty.

We can use the function `FactorRational` to represent the transfer function in factored form:

```
FactorRational[my2in1outTFobject]
```

$$\text{TransferFunction}\left[s, \left\{\left\{\frac{2}{(-i+s)\,(i+s)}, \frac{1}{-1+s}\right\}\right\}\right]$$

Assume that the stimulus

```
myStimulus = Sin[8 t] * ⅇ^(-t/10)
```

$$e^{-t/10}\,\text{Sin}[8\,t]$$

is applied at both inputs, and find the output response with `OutputResponse`:

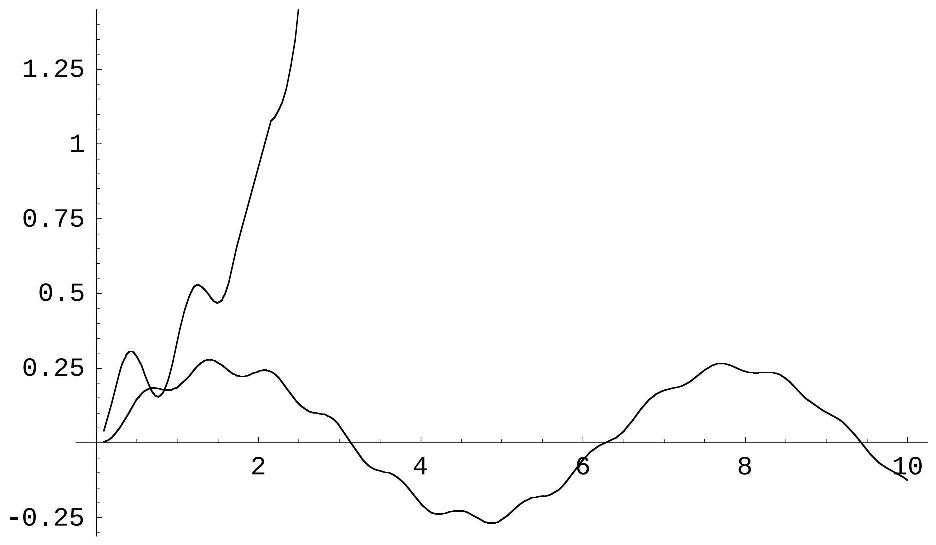
```
myOutput =
  OutputResponse[my2in1outTFobject, myStimulus, t] // ComplexExpand // Simplify;
```

```
myOutput // TraditionalForm
```

$$\begin{pmatrix} \frac{200\, e^{-t/10}\,(-160\, e^{t/10}\,\cos(t)+160\cos(8\,t)+50408\, e^{t/10}\,\sin(t)-6299\sin(8\,t))}{39703001} \\ \frac{10\, e^{-t/10}\,(-80\cos(8\,t)+80\, e^{11\,t/10}-11\sin(8\,t))}{6521} \end{pmatrix}$$

The built-in *Mathematica* function `Plot` plots both outputs:

```
Plot[Evaluate[myOutput], {t, 0.1, 10}];
```

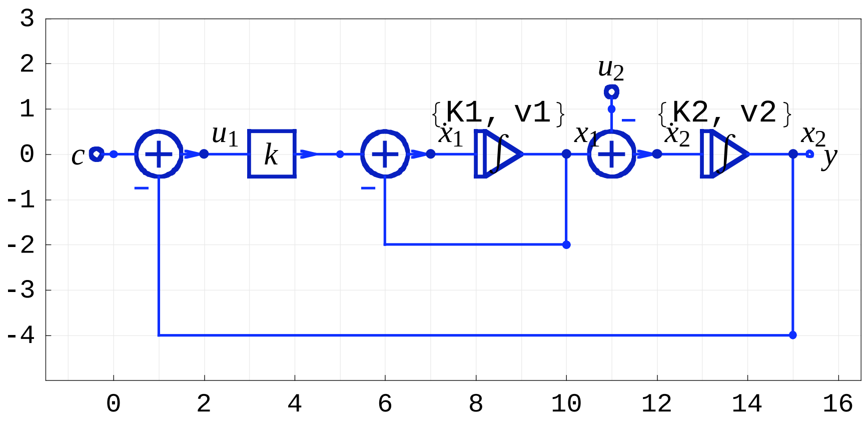


## 17.12. Deriving State-Space Equations with *SchematicSolver*

As an example, consider the simple production and inventory control model from Brogan, see *Control System Professional* Chapter 4.1.

```
BroganSystem = {
   {"Input", {0, 0}, c},
   {"Input", {11, 1}, u2, "", TextOffset → {0, -1}},
   {"Output", {15, 0}, y},
   {"Adder", {{0, 0}, {15, -4}, {2, 0}, {1, 1}}, {1, -1, 2, 0}},
   {"Adder", {{5, 0}, {10, -2}, {7, 0}, {6, 1}}, {1, -1, 2, 0}},
   {"Adder", {{10, 0}, {11, -1}, {12, 0}, {11, 1}}, {1, 0, 2, -1}},
   {"Block", {{2, 0}, {5, 0}}, k},
   {"Integrator", {{7, 0}, {10, 0}}, {K1, v1}},
   {"Integrator", {{12, 0}, {15, 0}}, {K2, v2}},
   {"Line", {{10, 0}, {10, -2}}},
   {"Line", {{15, 0}, {15, -4}}},
   {"Node", {2, 0}, u1}, {"Node", {7, 0}, xp1},
   {"Node", {12, 0}, xp2}, {"Node", {10, 0}, x1},
   {"Node", {15, 0}, x2}};
```

```
ShowSchematic[BroganSystem /. {u1 → u₁, u2 → u₂, x1 → x₁, x2 → x₂, xp1 → ẋ₁, xp2 → ẋ₂},
  PlotRange → {{-1.5, 16.5}, {-5, 3}}];
```



ContinuousSystemTransferFunction finds the transfer function:

```
{BroganSystemTF, BroganInp, BroganOut} =
 ContinuousSystemTransferFunction[BroganSystem] /. {K1 → 1, K2 → 1} // Simplify
```

$$\left\{\left\{\left\{\frac{k}{k+s+s^2}, -\frac{1+s}{k+s+s^2}\right\}\right\}, \{Y[\{0, 0\}], Y[\{11, 1\}]\}, \{Y[\{15, 0\}]\}\right\}$$

```
BroganSystemTFobject = TransferFunction[s, BroganSystemTF]
```

$$\text{TransferFunction}\left[s, \left\{\left\{\frac{k}{k+s+s^2}, -\frac{1+s}{k+s+s^2}\right\}\right\}\right]$$

The state-space realization returned by StateSpace does not correspond to BroganSystem.

```
StateSpace[BroganSystemTFobject] // TraditionalForm
StateSpace[BroganSystemTFobject, TargetForm → ObservableCompanion] //
 TraditionalForm
```

$$\left(\begin{array}{cccc|cc} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -k & 0 & -1 & 0 & 1 & 0 \\ 0 & -k & 0 & -1 & 0 & 1 \\ \hline k & -1 & 0 & -1 & 0 & 0 \end{array}\right)^{\mathcal{S}}.$$

$$\left(\begin{array}{cc|cc} 0 & -k & k & -1 \\ 1 & -1 & 0 & -1 \\ \hline 0 & 1 & 0 & 0 \end{array}\right)^{\mathcal{S}}.$$

The correct state-space equations for BroganSystem can be obtained only from the schematic.

By inspection, we derive the following state-space equations:

$$\begin{aligned} \dot{x}_1 &= -1\,x_1 - k\,x_2 + k\,c + 0\,u_2 \\ \dot{x}_2 &= 1\,x_1 + 0\,x_2 + 0\,c - 1\,u_2 \\ y &= 0\,x_1 + 1\,x_2 + 0\,c + 0\,u_2 \end{aligned}$$

or

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} -1 & -k \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} k & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} c \\ u_2 \end{pmatrix}$$

$$y = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 & 0 \end{pmatrix} \begin{pmatrix} c \\ u_2 \end{pmatrix}$$

that is equivalent to

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = A \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + B \begin{pmatrix} c \\ u_2 \end{pmatrix}$$

$$y = C \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + D \begin{pmatrix} c \\ u_2 \end{pmatrix}$$

and can be conveniently presented by

```
StateSpace[{{A}}, {{B}}, {{C}}, {{D}}] // TraditionalForm
StateSpace[{{-1, -k}, {1, 0}}, {{k, 0}, {0, -1}}, {{0, 1}}, {{0, 0}}] //
 TraditionalForm
```

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)^{\mathcal{S}}.$$

$$\left(\begin{array}{cc|cc} -1 & -k & k & 0 \\ 1 & 0 & 0 & -1 \\ \hline 0 & 1 & 0 & 0 \end{array}\right)^{\mathcal{S}}.$$

## 17.13. Step-by-Step Procedure for Deriving State-Space Equations

First, we find the signals at all nodes using `ContinuousSystemSignals`:

```
{BroganSignals, BroganVars} = ContinuousSystemSignals[BroganSystem] // Simplify;
% // Transpose // TableForm
```

$$\frac{c\,k\,K1\,K2-K2\,s\,u2+K2\,v1+s\,v2+K1\,(-K2\,u2+v2)}{k\,K1\,K2+s\,(K1+s)} \qquad Y[\{15,\ 0\}]$$

$$\frac{c\,k\,K1\,s-K1\,s\,u2-s^2\,u2+s\,v1-k\,K1\,v2}{k\,K1\,K2+s\,(K1+s)} \qquad Y[\{12,\ 0\}]$$

$$u2 \qquad Y[\{11,\ 1\}]$$

$$\frac{c\,k\,K1\,s+s\,v1+k\,K1\,(K2\,u2-v2)}{k\,K1\,K2+s\,(K1+s)} \qquad Y[\{10,\ 0\}]$$

$$\frac{c\,k\,s^2+k\,K2\,s\,u2-k\,K2\,v1-s\,v1-k\,s\,v2}{k\,K1\,K2+s\,(K1+s)} \qquad Y[\{7,\ 0\}]$$

$$\frac{k\,(c\,s\,(K1+s)+K1\,K2\,u2+K2\,s\,u2-K2\,v1-K1\,v2-s\,v2)}{k\,K1\,K2+s\,(K1+s)} \qquad Y[\{5,\ 0\}]$$

$$\frac{c\,s\,(K1+s)+K1\,K2\,u2+K2\,s\,u2-K2\,v1-K1\,v2-s\,v2}{k\,K1\,K2+s\,(K1+s)} \qquad Y[\{2,\ 0\}]$$

$$c \qquad Y[\{0,\ 0\}]$$

Next, we extract the signals at integrator inputs, `Y[{7,0}]` representing $\dot{x}_1$, and `Y[{12,0}]` representing $\dot{x}_2$:

```
X1p = BroganSignals[[5]] // Simplify
X2p = BroganSignals[[2]] // Simplify
```

$$\frac{c\,k\,s^2+k\,K2\,s\,u2-k\,K2\,v1-s\,v1-k\,s\,v2}{k\,K1\,K2+s\,(K1+s)}$$

$$\frac{c\,k\,K1\,s-K1\,s\,u2-s^2\,u2+s\,v1-k\,K1\,v2}{k\,K1\,K2+s\,(K1+s)}$$

```
Clear[matrixA, matrixB, matrixC, matrixD];

General::spell1 : Possible spelling error: new
   symbol name "matrixB" is similar to existing symbol "matrixA".

General::spell : Possible spelling error: new symbol
   name "matrixC" is similar to existing symbols {matrixA, matrixB}.

General::spell : Possible spelling error: new symbol
   name "matrixD" is similar to existing symbols {matrixA, matrixB, matrixC}.
```

The matrix *A* can be computed for zero input signals ($c \to 0$, $u2 \to 0$) and by suppressing integration ($K1 \to 0$, $K2 \to 0$). The coefficients of the matrix *A* can be computed by setting the initial conditions to 1 or 0. For example, the first coefficient represents a flow from $x_1$ to $\dot{x}_1$, and we set ($v1 \to 1$, $v2 \to 0$).

```
matrixA = s * {
     {(X1p /. {v1 → 1, v2 → 0}), (X1p /. {v1 → 0, v2 → 1})},
     {(X2p /. {v1 → 1, v2 → 0}), (X2p /. {v1 → 0, v2 → 1}) }
    } /. {K1 → 0, K2 → 0, c → 0, u2 → 0} // Simplify
% // MatrixForm

{{-1, -k}, {1, 0}}

( -1  -k )
(  1   0 )
```

The matrix *B* can be computed for zero initial conditions ($v1 \to 0$, $v2 \to 0$) and by suppressing integration ($K1 \to 0$, $K2 \to 0$). The coefficients of the matrix *B* can be computed by setting the input signals to 1 or 0. For example, the first coefficient represents a flow from *c* to $\dot{x}_1$, and we set ($c \to 1$, $u2 \to 0$).

```
matrixB = {
    {(X1p /. {c → 1, u2 → 0}), (X1p /. {c → 0, u2 → 1})},
    {(X2p /. {c → 1, u2 → 0}), (X2p /. {c → 0, u2 → 1})}
   } /. {K1 → 0, K2 → 0, v1 → 0, v2 → 0} // Simplify
% // MatrixForm

{{k, 0}, {0, -1}}

( k   0 )
( 0  -1 )
```

From the schematic, we identify that `Y[{15,0}]` represents the output signal *y*:

```
Yout = BroganSignals[[1]] // Simplify
```

$$\frac{c\ k\ K1\ K2 - K2\ s\ u2 + K2\ v1 + s\ v2 + K1\ (-K2\ u2 + v2)}{k\ K1\ K2 + s\ (K1 + s)}$$

The matrix *C* can be computed for zero input signals ($c \to 0$, $u2 \to 0$) and by suppressing integration ($K1 \to 0$, $K2 \to 0$). The coefficients of the matrix *C* can be computed by setting the initial conditions to 1 or 0. For example, the first coefficient represents a flow from $x_1$ to *y*, and we set ($v1 \to 1$, $v2 \to 0$).

```
matrixC = s * {
     {(Yout /. {v1 → 1, v2 → 0}), (Yout /. {v1 → 0, v2 → 1})}
    } /. {K1 → 0, K2 → 0, c → 0, u2 → 0} // Simplify
% // MatrixForm

{{0, 1}}

( 0  1 )
```

The matrix *D* can be computed for zero initial conditions ($v1 \to 0$, $v2 \to 0$) and by suppressing integration ($K1 \to 0$, $K2 \to 0$). The coefficients of the matrix *D* can be computed by setting the input signals to 1 or 0. For example, the first coefficient represents a flow from *c* to *y*, and we set ($c \to 1$, $u2 \to 0$).

```
matrixD = {
    {(Yout /. {c → 1, u2 → 0}), (Yout /. {c → 0, u2 → 1})}
   } /. {K1 → 0, K2 → 0, v1 → 0, v2 → 0} // Simplify
% // MatrixForm
```

```
{{0, 0}}
```

$$\begin{pmatrix} 0 & 0 \end{pmatrix}$$

We can verify that this state-space realization is the state-space of `SystemBrogan`:

```
StateSpace[matrixA, matrixB, matrixC, matrixD] // TraditionalForm
```

$$\left(\begin{array}{cc|cc} -1 & -k & k & 0 \\ 1 & 0 & 0 & -1 \\ \hline 0 & 1 & 0 & 0 \end{array}\right)^{\mathcal{S}}.$$

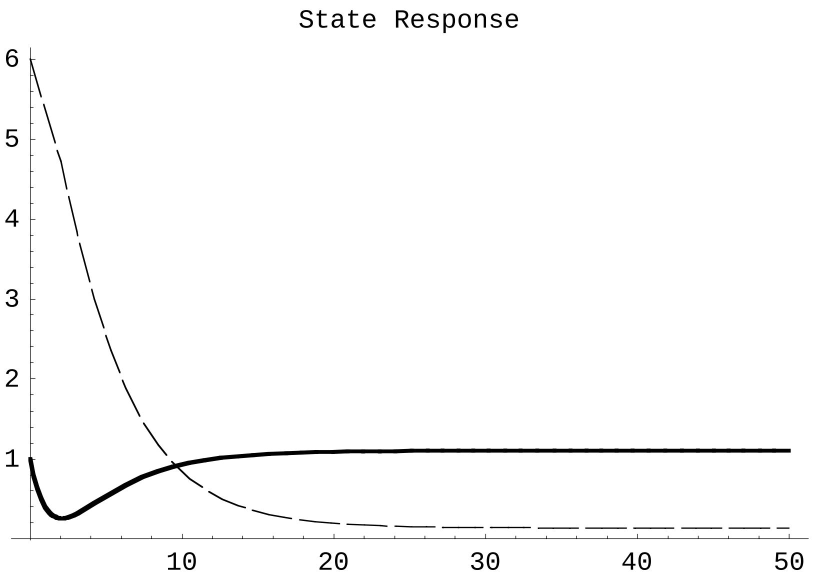
We can use *Control System Professional* to compute the state response of the system with `StateResponse`:

```
stateResponseBrogan = StateResponse[StateSpace[matrixA, matrixB] /.
    k → 3 / 16, {c, 1.1 x10}, t, InitialConditions → {x10, c}] // Simplify
```

$\{(1.1 + 2.05\, e^{-3t/4} - 2.15\, e^{-t/4})\, x10,\ 1.\, c + (-5.86667 - 2.73333\, e^{-3t/4} + 8.6\, e^{-t/4})\, x10\}$

Specifying the initial conditions and following the procedure described in *Control System Professional*, we can plot the results for particular initial values.

```
Plot[Evaluate[stateResponseBrogan /. {x10 → 1, c → 6}], {t, 0, 50},
  PlotStyle → {Thickness[.005], Dashing[{.05, .01}]},
  PlotRange → All, PlotLabel → "State Response"];
```



See *Control System Professional* Chapter 4.1 for details.

# 18. Post-Processing Using *Signals and Systems Pack*

## 18.1. Introduction

*Signals and Systems Pack* is a *Mathematica* implementation of the necessary tools for working with signals and systems. Analysis techniques for both discrete and continuous signals are available.

We use *SchematicSolver* to draw schematics of systems and to symbolically compute the system transfer function directly from schematics.

We can use *Signals and Systems Pack* for post-processing results returned by *SchematicSolver* to

- a) compute the transfer function poles and zeros with the function `GetRootList`
- b) represent a single-input single-output discrete-time system by the function `DigitalFilter`
- c) compute the impulse response with the functions `EvaluateOperators` and `DigitalFilter`
- d) compute the output signal for a given input signal represented as a formula
- e) find the Discrete-Time Fourier Transform of discrete signals with the function `DiscreteTimeFourierTransform`
- f) find and plot magnitude response of discrete systems represented by poles and zeros with the function `DiscreteTimeFourierTransform`
- g) create a general report about the system for common signal analysis procedures with the function `DSPAnalyze`, (a plot with respect to the sample index, the *z*-transform with its region of convergence, a pole-zero plot, and the frequency response with the magnitude-phase plot),
- h) plot discrete signals with the function `DiscreteSignalPlot`

If packages have not already been loaded, we load *SignalProcessing* and *SchematicSolver* with

```
Needs["SignalProcessing`"];
Needs["SchematicSolver`"];
```

We shall adjust some options to obtain better appearance of the example schematics:

```
SetOptions[InputNotebook[], ImageSize → {350, 300}];

SetOptions[ShowSchematic, ElementScale → 1,
  FontSize → Automatic, Frame → True, GridLines → Automatic, PlotRange → All];

SetOptions[DrawElement, ElementSize → {1, 1},
  PlotStyle → {{RGBColor[0, 0, 0.7], Thickness[0.005], PointSize[0.012]},
    {RGBColor[0, 0, 1], Thickness[0.0035], PointSize[0.01]}},
  ShowArrowTail → True, ShowNodes → True, TextOffset → Automatic,
  TextStyle → {FontFamily → Times, FontSize → 12}];
```

## 18.2. Drawing and Solving Systems with *SchematicSolver*

### Draw Systems Using *SchematicSolver*

Consider a discrete-time system and find the transfer function using *SchematicSolver*.

`TransposedDirectForm2IIRFilterSchematic` creates the schematic of an example system:

```
{exampleSchematic, {inpCoord}, {outCoord}} =
  TransposedDirectForm2IIRFilterSchematic[{{b0, b1, b2}, {a1, a2}}];
```

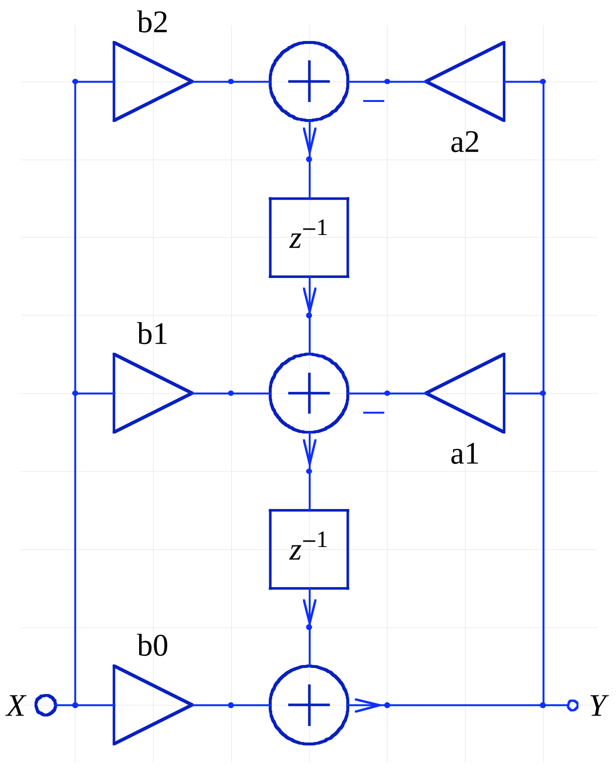
`TransposedDirectForm2IIRFilterSchematic` returns the coordinates of the system input and the system output. You can add the input element and the output element to form the system specification:

```
exampleSystem = Join[exampleSchematic,
  {{"Input", inpCoord, X},
   {"Output", outCoord, Y}}]
```

```
{{Line, {{6, 0}, {4, 0}}}, {Adder, {{2, 0}, {3, -1}, {4, 0}, {3, 1}}, {1, 0, 2, 1}},
 {Multiplier, {{0, 0}, {2, 0}}, b0}, {Line, {{0, 0}, {0, 4}}},
 {Line, {{6, 0}, {6, 4}}}, {Adder, {{2, 4}, {3, 3}, {4, 4}, {3, 5}}, {1, 2, -1, 1}},
 {Delay, {{3, 3}, {3, 1}}, 1}, {Multiplier, {{0, 4}, {2, 4}}, b1},
 {Multiplier, {{6, 4}, {4, 4}}, a1}, {Line, {{0, 4}, {0, 8}}},
 {Line, {{6, 4}, {6, 8}}}, {Adder, {{2, 8}, {3, 7}, {4, 8}, {3, 9}}, {1, 2, -1, 0}},
 {Delay, {{3, 7}, {3, 5}}, 1}, {Multiplier, {{0, 8}, {2, 8}}, b2},
 {Multiplier, {{6, 8}, {4, 8}}, a2}, {Input, {0, 0}, X}, {Output, {6, 0}, Y}}
```

`ShowSchematic` draws the schematic of the system:

```
ShowSchematic[exampleSystem, Frame → False]
```



## Find Transfer Function Using *SchematicSolver*

`DiscreteSystemTransferFunction` finds the transfer function:

```
{tfMatrix, systemInp, systemOut} = DiscreteSystemTransferFunction[exampleSystem]
```

$$\left\{\left\{\left\{-\frac{-b2 - b1\,z - b0\,z^2}{a2 + a1\,z + z^2}\right\}\right\}, \{Y[\{0, 0\}]\}, \{Y[\{6, 0\}]\}\right\}$$

The transfer function of this system is the first element of `tfMatrix`:

```
tf = tfMatrix[[1, 1]]
```

$$-\frac{-b2 - b1\,z - b0\,z^2}{a2 + a1\,z + z^2}$$

The symbol *z* is reserved for the complex variable in the *z*-transform domain.

For the specific values

```
myValues = {a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9}
```

```
{a1 → 0.85, a2 → 0.95, b0 → 0.7, b1 → 0.8, b2 → 0.9}
```

we obtain the transfer function of the system as

```
tfSpecific = tf /. myValues // Simplify
```

$$\frac{0.9 + 0.8\,z + 0.7\,z^2}{0.95 + 0.85\,z + z^2}$$

## 18.3. Processing Systems with *Signals and Systems Pack*

### Representing Systems in *Signals and Systems Pack*

Single-input single-output discrete-time system can be represented in *Signals and Systems Pack* by the function `DigitalFilter`. This function requires three arguments: (1) a list of transfer function poles, (2) a list of transfer function zeros, and (3) the time variable *n*.

```
tfPoles = GetRootList[Denominator[tfSpecific], z]
tfZeros = GetRootList[Numerator[tfSpecific], z]
tfSystem = DigitalFilter[tfPoles, tfZeros, n]
```

```
{-0.425 - 0.87714 ⅈ, -0.425 + 0.87714 ⅈ}

{-0.571429 - 0.979379 ⅈ, -0.571429 + 0.979379 ⅈ}

DigitalFilter[{-0.425 - 0.87714 ⅈ, -0.425 + 0.87714 ⅈ},
 {-0.571429 - 0.979379 ⅈ, -0.571429 + 0.979379 ⅈ}, n]
```

The symbol *z* is reserved for the complex variable in the *z* plain. $z = e^{i\,\omega}$ refers to the unit circle.

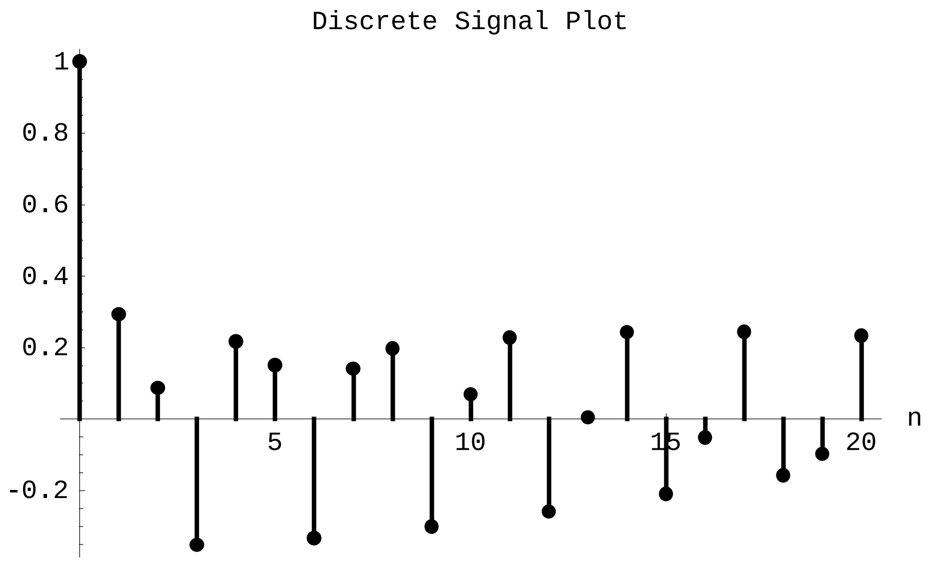
`DigitalFilter[poles,zeros,n][signal]` represents the discrete time-domain processing of `signal` by the discrete-time system. The impulse response is obtained when signal is `KroneckerDelta`. We should use `EvaluateOperators` to generate the output signal.

```
inpSignal = KroneckerDelta[n]
outSignal = EvaluateOperators[tfSystem[inpSignal]]
```

```
KroneckerDelta[n]

(-0.176692 + 0.0813263 ⅈ) (-0.425 - 0.87714 ⅈ)^n DiscreteStep[-1 + n] -
 (0.176692 + 0.0813263 ⅈ) (-0.425 + 0.87714 ⅈ)^n DiscreteStep[-1 + n] + 1. KroneckerDelta[n]
```

`DiscreteSignalPlot` plots a discrete signal:

```
DiscreteSignalPlot[outSignal, {n, 0, 20}];
```



The output signal of a system with the real coefficients and a real input signal is a real signal. We can use real part of the output signal to find a more convenient presentation of the output signal:

```
outSignalRe = ComplexExpand[Re[outSignal]]
```

$-0.353383\ 0.974679^n\ \text{Cos}[2.02199\ n]\ \text{DiscreteStep}[-1+n]+$
$1.\ \text{KroneckerDelta}[n]+0.162653\ 0.974679^n\ \text{DiscreteStep}[-1+n]\ \text{Sin}[2.02199\ n]$

## Processing in *Signals and Systems Pack*

Consider two discrete sinusoidal signals:

```
inpSignal1 = 0.2 * Sin[n / 10 + π / 3]
```

$0.2\ \text{Sin}\left[\frac{n}{10}+\frac{\pi}{3}\right]$

```
inpSignal2 = 0.2 * Sin[2.02 * n + π / 5]
```

$0.2\ \text{Sin}\left[2.02\ n+\frac{\pi}{5}\right]$

We can generate an input signal as a sum of two signals

```
inpSignal  = inpSignal1 + inpSignal2
```

$0.2\ \text{Sin}\left[2.02\ n+\frac{\pi}{5}\right]+0.2\ \text{Sin}\left[\frac{n}{10}+\frac{\pi}{3}\right]$

and apply the input signal to the system `tfSystem` to compute the output signal:

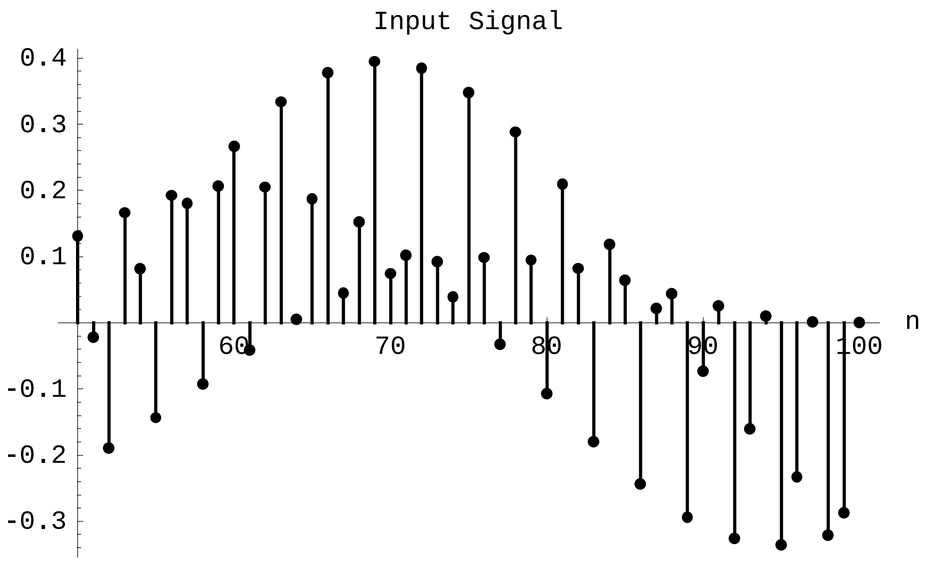
```
outSignal = EvaluateOperators[tfSystem[inpSignal]] // Re // ComplexExpand // Simplify
```

$0.290762\ \text{KroneckerDelta}[n]+\text{DiscreteStep}[-1+n]$
$(0.210874\ 1.^n\ \text{Cos}[0.1\ n]-1.24333\ 1.^n\ \text{Cos}[2.02\ n]+1.32322\ 0.974679^n\ \text{Cos}[2.02199\ n]+$
$0.124615\ 1.^n\ \text{Sin}[0.1\ n]-0.443552\ 1.^n\ \text{Sin}[2.02\ n]+0.640625\ 0.974679^n\ \text{Sin}[2.02199\ n])$

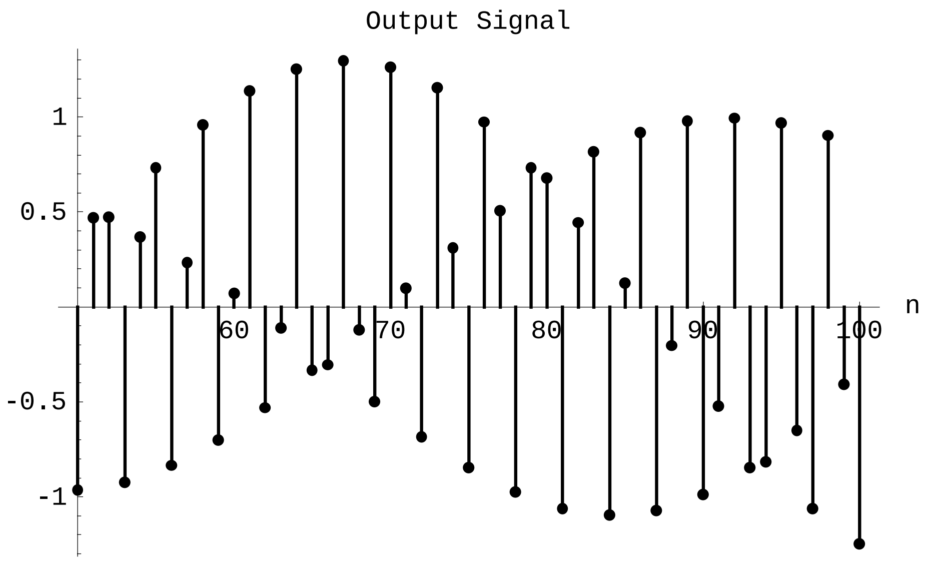
`ComplexExpand` expands an expression assuming that all variables are real. If you do not want to see complex numbers in the output signal, you could use `Re`, `ComplexExpand`, and `Simplify`.

`DiscreteSignalPlot` plots the discrete signals:

```
DiscreteSignalPlot[inpSignal, {n, 50, 100}, PlotLabel → "Input Signal"];
```



```
DiscreteSignalPlot[outSignal, {n, 50, 100}, PlotLabel → "Output Signal"];
```

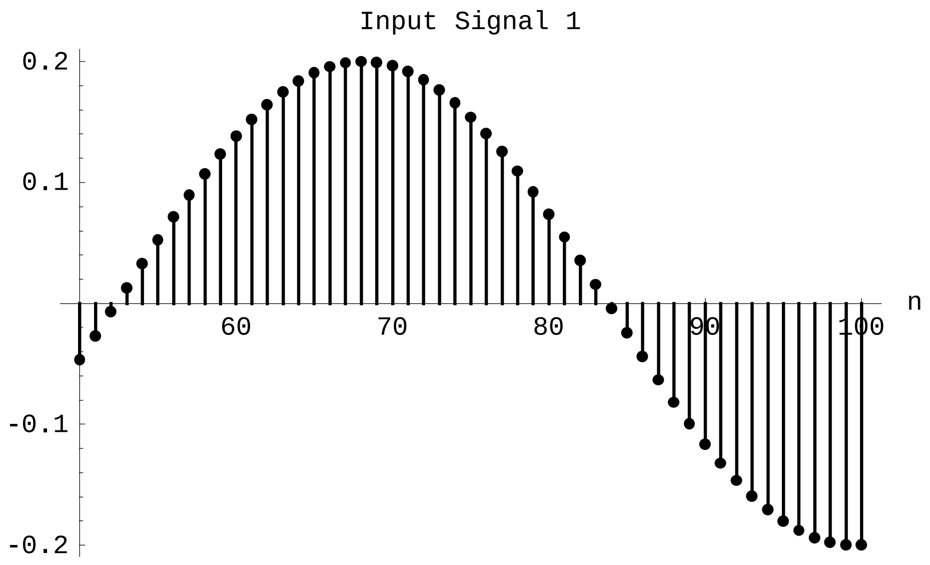


Comparing the above plots, we cannot identify how the system affects the input signal. Let us process only the first sinusoidal signal:

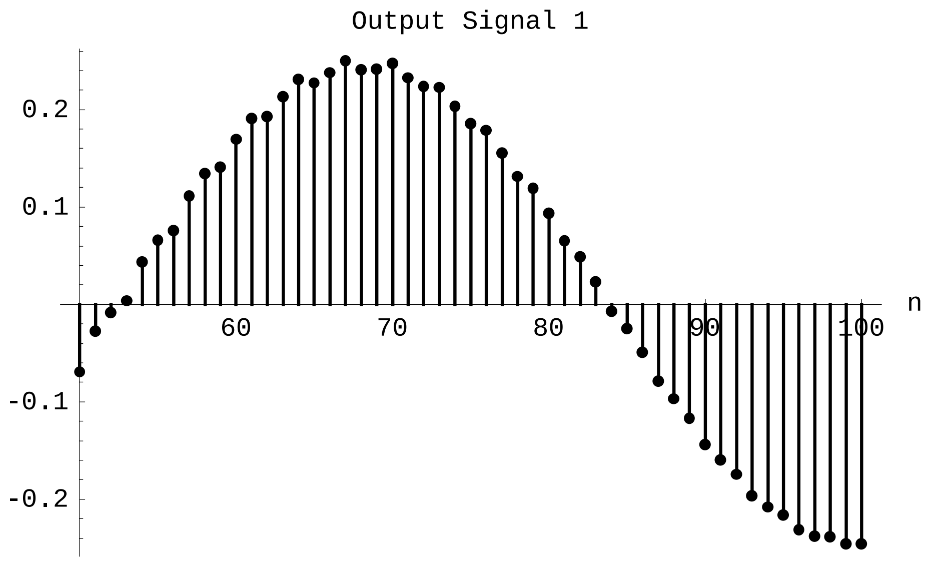
```
outSignal1 = EvaluateOperators[tfSystem[inpSignal1]] // Re // ComplexExpand // Simplify
```

```
0.173205 KroneckerDelta[n] +
 DiscreteStep[-1 + n] (0.210874 1.^n Cos[0.1 n] - 0.0376692 0.974679^n Cos[2.02199 n] +
    0.124615 1.^n Sin[0.1 n] - 0.00595517 0.974679^n Sin[2.02199 n])
```

```
DiscreteSignalPlot[inpSignal1, {n, 50, 100}, PlotLabel → "Input Signal 1"];
```



```
DiscreteSignalPlot[outSignal1, {n, 50, 100}, PlotLabel → "Output Signal 1"];
```

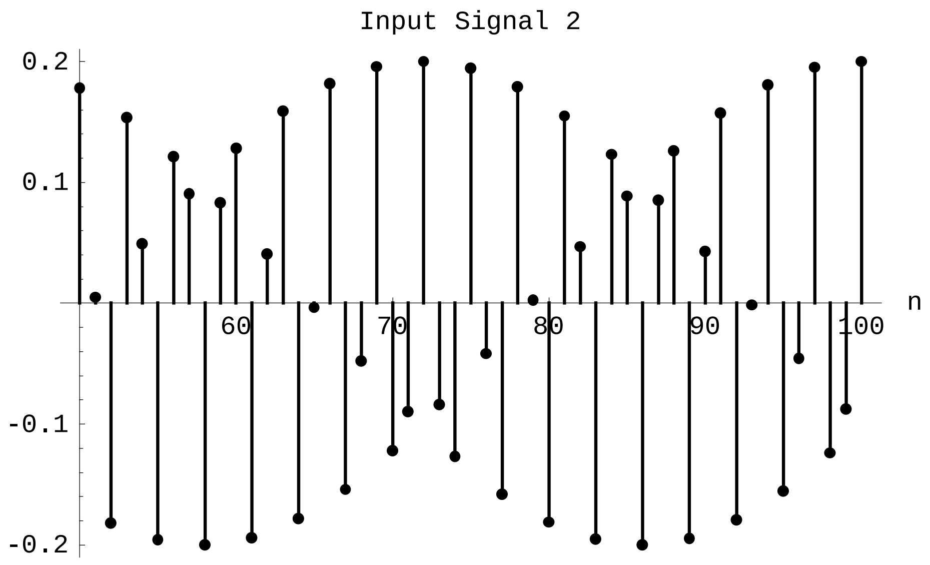


We see that the output signal is approximately the same as the input signal. Let us process only the second sinusoidal signal:

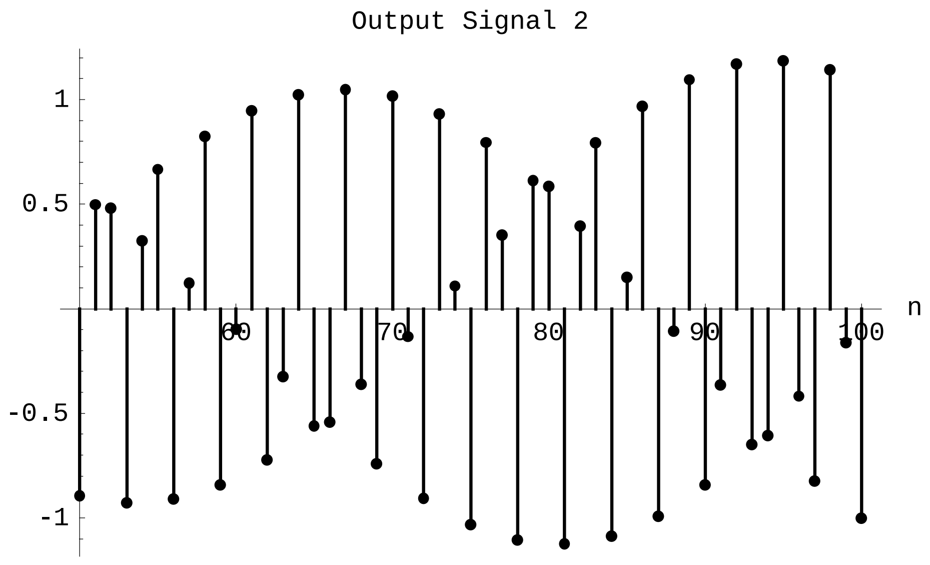
```
outSignal2 = EvaluateOperators[tfSystem[inpSignal2]] // Re // ComplexExpand // Simplify
```

```
0.117557 KroneckerDelta[n] +
 DiscreteStep[-1 + n] (-1.24333 1.^n Cos[2.02 n] + 1.36089 0.974679^n Cos[2.02199 n] -
    0.443552 1.^n Sin[2.02 n] + 0.646581 0.974679^n Sin[2.02199 n])
```

```
DiscreteSignalPlot[inpSignal2, {n, 50, 100}, PlotLabel → "Input Signal 2"];
```



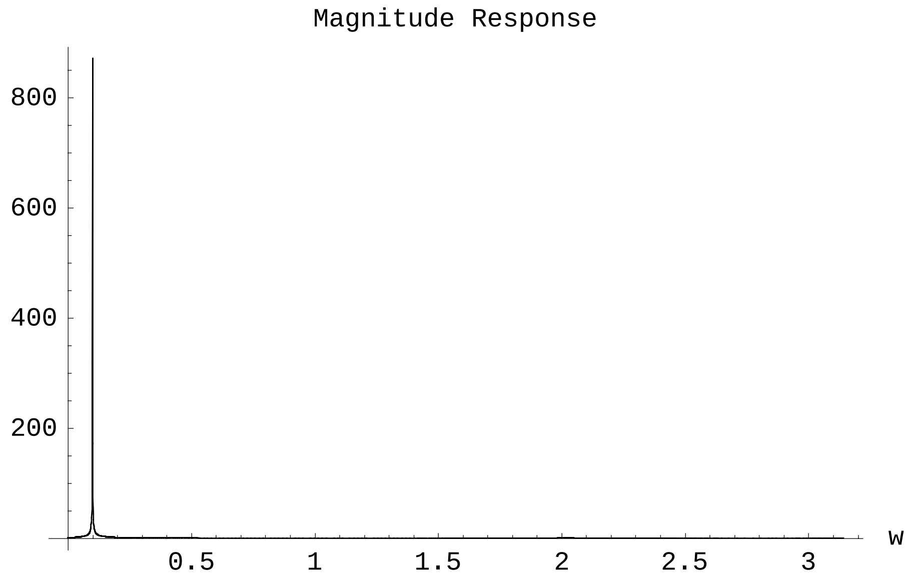
```
DiscreteSignalPlot[outSignal2, {n, 50, 100}, PlotLabel → "Output Signal 2"];
```



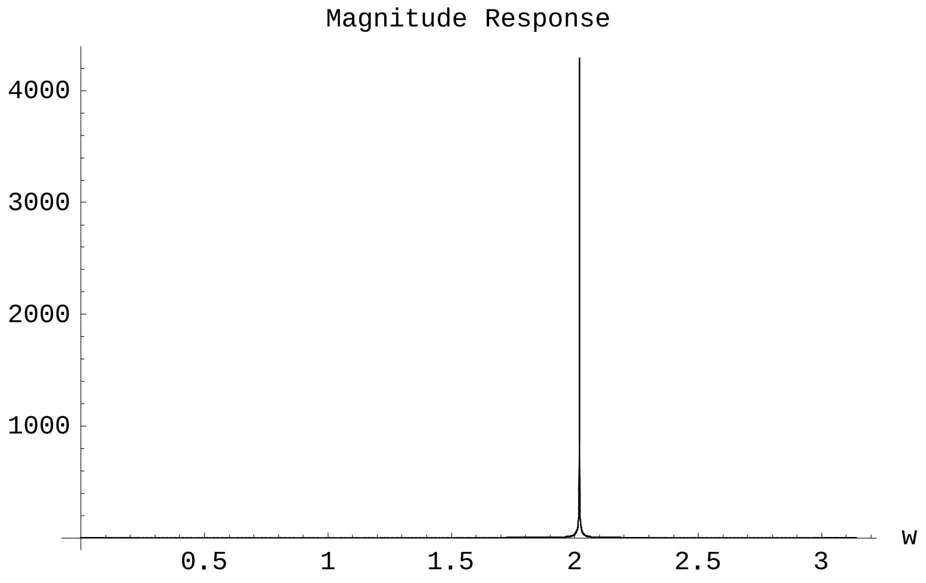
Note that the output signal has approximately five times larger amplitude compared with the input signal. In addition, the output signal is delayed with respect to the input signal.

If we apply `DiscreteTimeFourierTransform` to the output signals, we get a better insight into the processing:

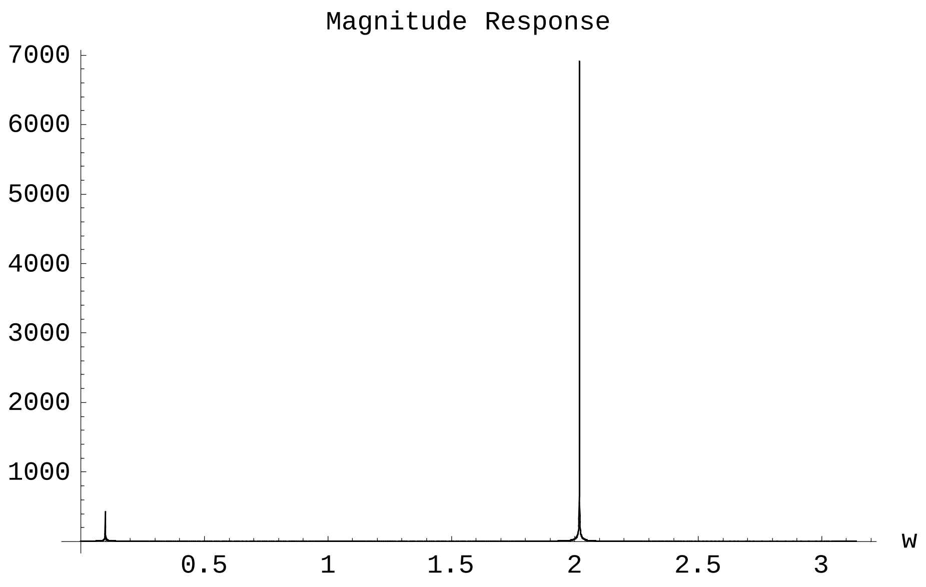
```
DiscreteTimeFourierTransform[outSignal1, n, w];
MagnitudePhasePlot[%, {w, 0, Pi}, PhaseScale → None];
```



```
DiscreteTimeFourierTransform[outSignal2, n, w];
MagnitudePhasePlot[%, {w, 0, Pi}, PhaseScale → None];
```



```
DiscreteTimeFourierTransform[outSignal, n, w];
MagnitudePhasePlot[%, {w, 0, Pi}, PhaseScale → None];
```

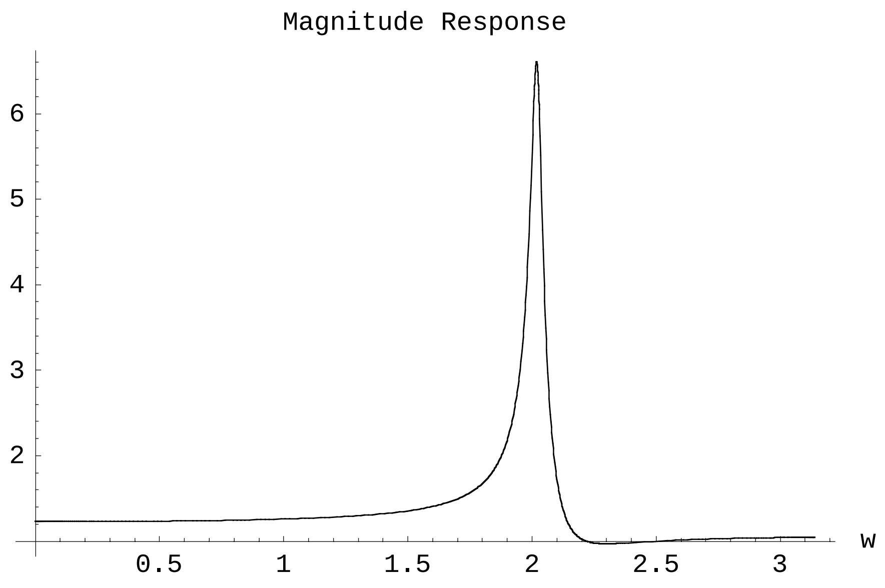


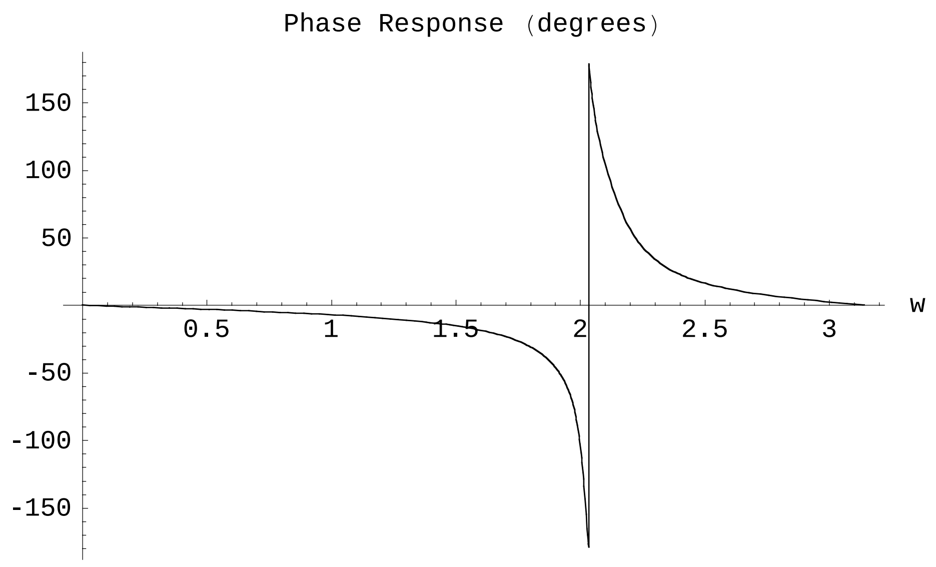
Comparing the magnitude responses of the output signals we conclude that the resultant signal consists of two sinusoidal signals; the amplitude of the sinusoidal signal with higher frequency is significantly larger.

## Plot Frequency Response using *Signals and Systems Pack*

We can apply `DiscreteTimeFourierTransform` to the system, represented by poles and zeros, to find and plot the frequency response:

```
DiscreteTimeFourierTransform[tfSystem, n, w];
MagnitudePhasePlot[%, {w, 0, π}];
```

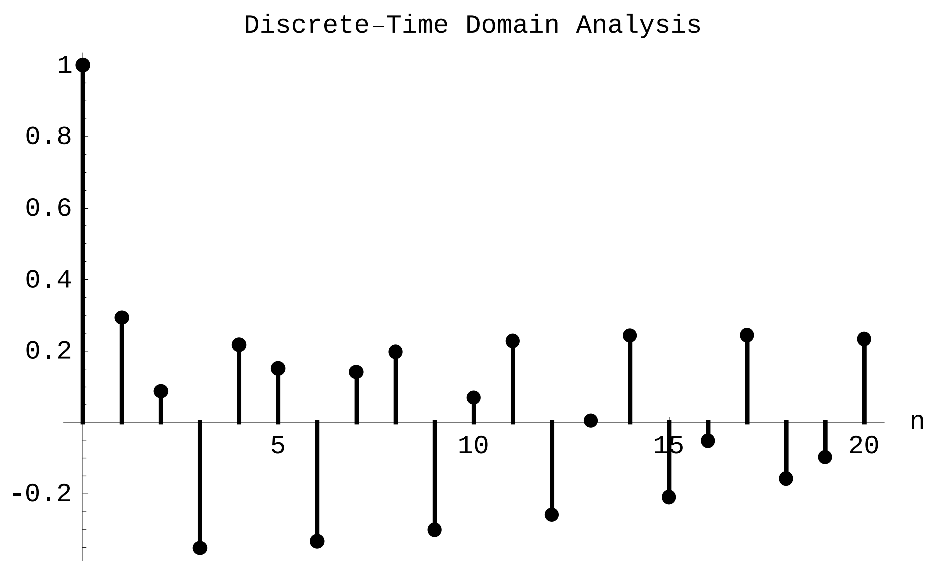




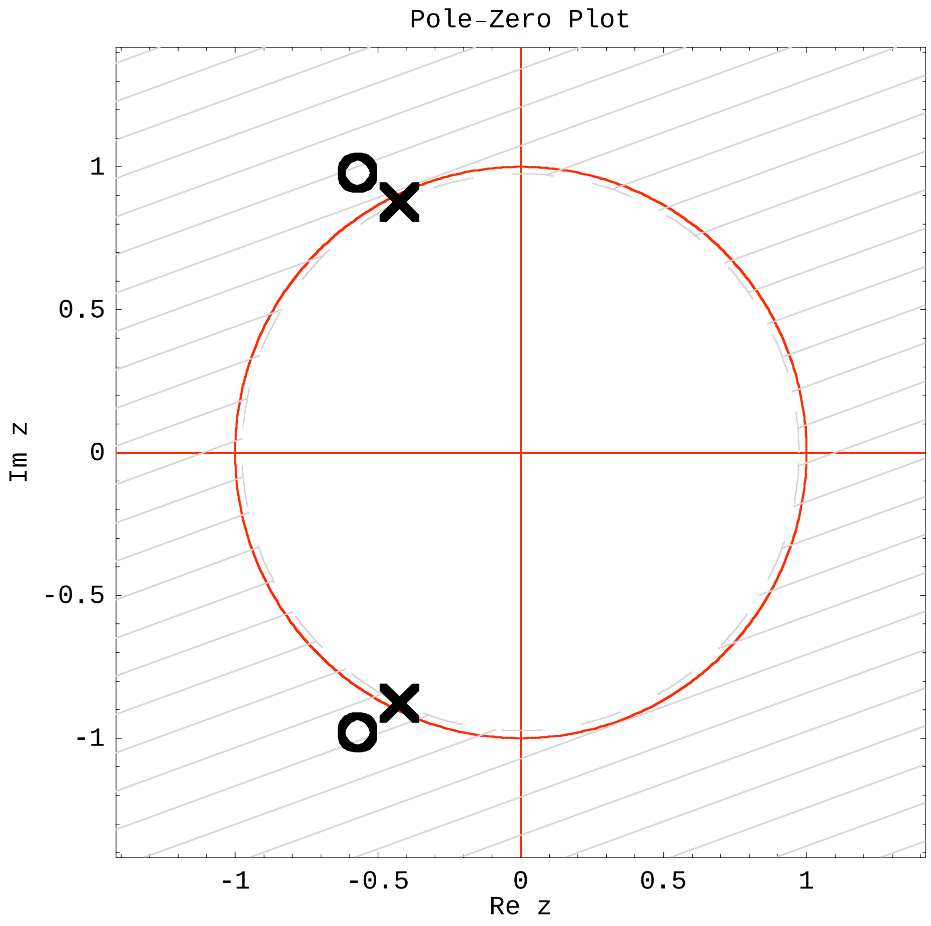
## General Report on Systems in *Signals and Systems Pack*

For common signal analysis procedures, we can use the omnibus functions `DSPAnalyze`. This function creates a general report about the system:

```
DSPAnalyze[tfSystem, {n, 0, 20}]
```

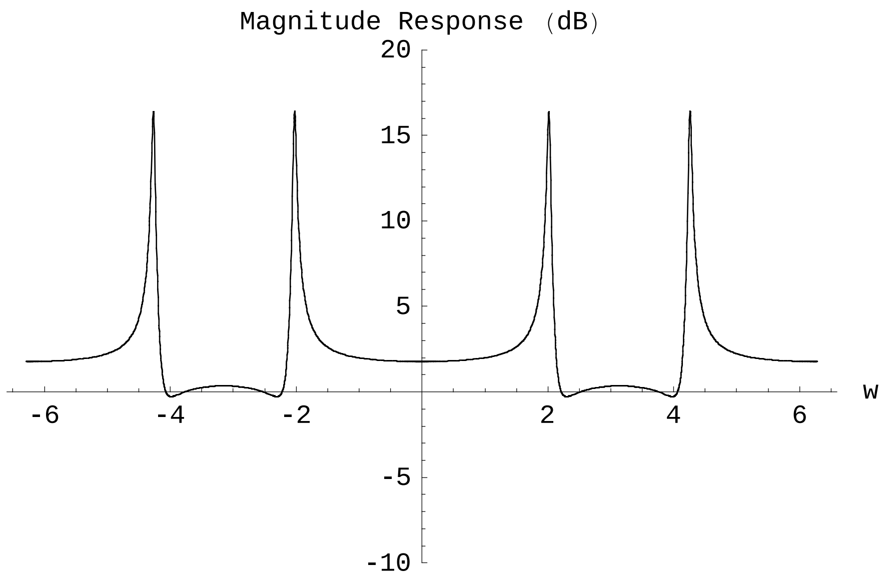


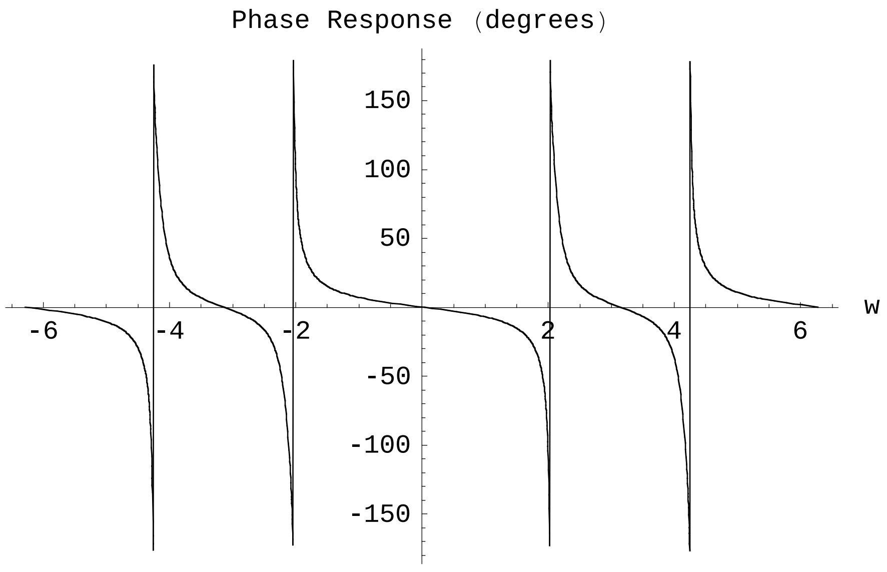
```
Given the input function:

DigitalFilter[{-0.425 - 0.87714 i, -0.425 + 0.87714 i},
 {-0.571429 - 0.979379 i, -0.571429 + 0.979379 i}, n]

The z-transform is:

(((0.571429 - 0.979379 i) + z) ((0.571429 + 0.979379 i) + z)) /
 (((0.425 - 0.87714 i) + z) ((0.425 + 0.87714 i) + z))

The region of convergence is:

0.974679 < |z| < ∞

The system is stable.
The zeroes are:   {-0.571429 - 0.979379 i, -0.571429 + 0.979379 i}

The poles are:    {-0.425 - 0.87714 i, -0.425 + 0.87714 i}
```



```
Since the sequence is stable with the default
       values assumed for any of the parameters, the
       frequency response can be computed
       directly from the z-transform.

The frequency response is:
```

$$\frac{(1.28571 + 0.\, i) + (1.14286 + 0.\, i)\, e^{i w} + e^{2 i w}}{(0.95 + 0.\, i) + (0.85 + 0.\, i)\, e^{i w} + e^{2 i w}}$$





```
{ZTransformData[(((0.571429 - 0.979379 i) + z) ((0.571429 + 0.979379 i) + z)) /
   (((0.425 - 0.87714 i) + z) ((0.425 + 0.87714 i) + z)),
  RegionOfConvergence[0.974679, ∞], TransformVariables[z]],
 DTFTData[ (1.28571 + 0. i) + (1.14286 + 0. i) e^(i w) + e^(2 i w)
           ------------------------------------------------ , TransformVariables[w]]}
            (0.95 + 0. i) + (0.85 + 0. i) e^(i w) + e^(2 i w)
```

# 19. Bibliography

[Ant1979] Antoniou, A., *Digital Filters: Analysis and Design*, McGraw-Hill, New York, NY, 1979.

[Bah1990] Baher, H., *Analog and Digital Signal processing*, John Wiley and Sons, New York, NY, 1990.

[Che1970] Chen, C. T., *Introduction to Linear System Theory*, Holt, Rinehart and Winston, Inc., New York, NY, 1970.

[CSP2002] *Control System Professional*, *Mathematica* application package, Wolfram Research, Inc., 2002.

[Dol2002] Dolecek, G., *Multirate Systems: Design and Applications*, Idea Group Publishing, Hershey, PA, 2002.

[Gly2001] Glynn, J. and Gray, T., *The Beginner's Guide to Mathematica Version 4*, Cambridge University Press, Cambridge, 2001.

[Kai1980] Kailath, T., *Linear Systems*, Prentice Hall, Englewood Cliffs, NJ, 1980.

[Lut2001] Lutovac, M. D., Tosic, D. V., and Evans, B. L., *Filter Design for Signal Processing Using MATLAB and Mathematica*, Prentice Hall, Upper Saddle River, NJ, 2001.

[Mit1998] Mitra, S. K., *Digital Signal Processing*, McGraw-Hill, New York, NY, 1998.

[Opp1992] Oppenheim, A. V., and Nawab, S. H., *Symbolic and Knowledge-based Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[SSP1995] *Signals and Systems*, *Mathematica* application package, Wolfram Research, Inc., 1995.

[Wol1999] Wolfram, S., *The Mathematica Book*, 4th ed., Cambridge University Press, Wolfram Media, Cambridge, 1999.

[Zad1969] Zadeh, L. A., and Polak, E., *System Theory*, McGraw-Hill, New York, NY, 1969.