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]
```

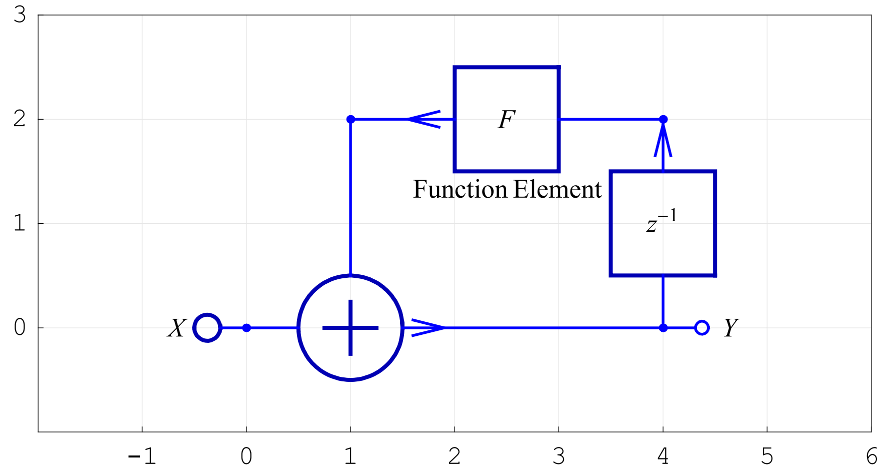
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[2]:= Needs["SchematicSolver`"];
```

Here is an example system:

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

```
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:

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

```
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.

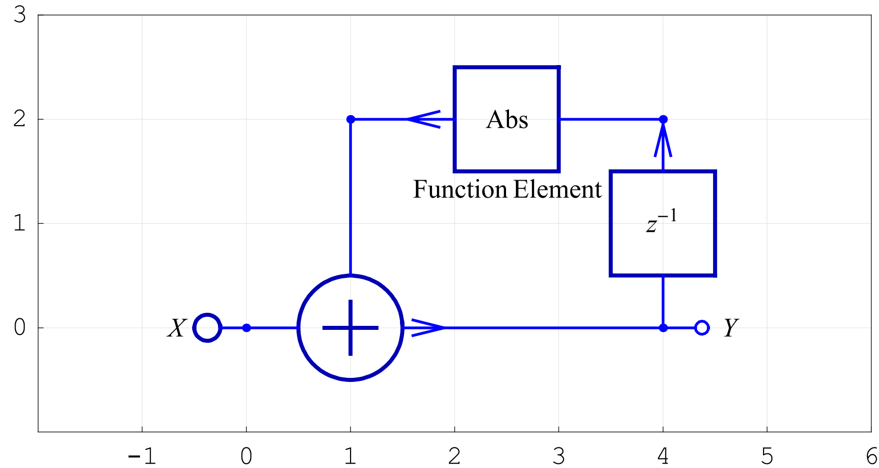
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[21]:= Needs["SchematicSolver`"];
```

Here is an example system:

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

```
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]];
```

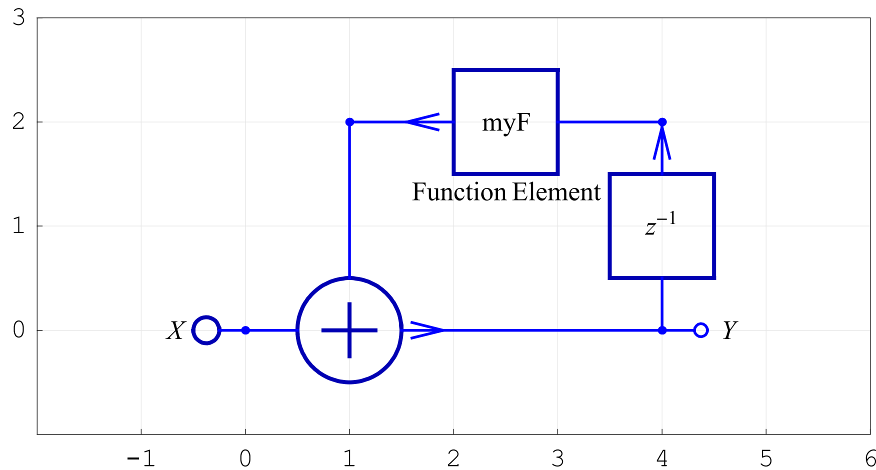
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[32]:= Needs["SchematicSolver`"];
```

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

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:

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

```
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:

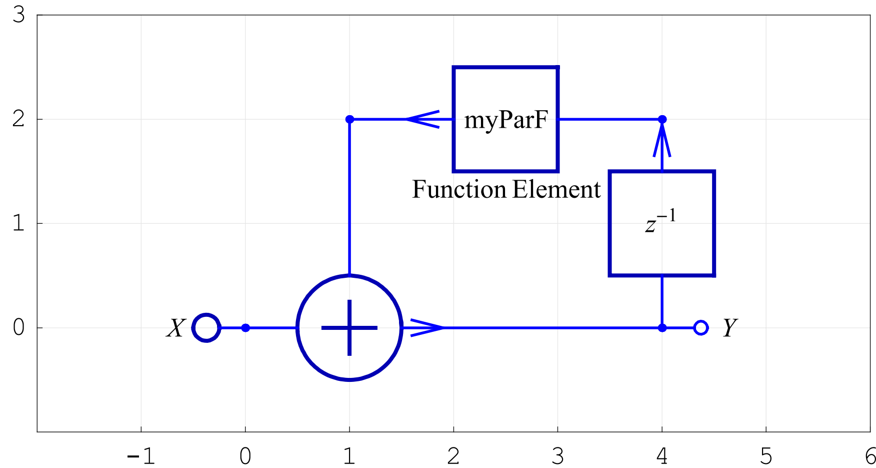
```
In[43]:= Clear[p];
         myParF[x_] := p * Abs[x];
```

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`):

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

```
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=

$$\begin{pmatrix} 1 \\ p+\frac{1}{2} \\ p\left|p+\frac{1}{2}\right|+\frac{1}{4} \\ p\left|p\left|p+\frac{1}{2}\right|+\frac{1}{4}\right|+\frac{1}{8} \\ p\left|p\left|p\left|p+\frac{1}{2}\right|+\frac{1}{4}\right|+\frac{1}{8}\right|+\frac{1}{16} \\ p\left|p\left|p\left|p\left|p+\frac{1}{2}\right|+\frac{1}{4}\right|+\frac{1}{8}\right|+\frac{1}{16}\right|+\frac{1}{32} \\ p\left|p\left|p\left|p\left|p\left|p+\frac{1}{2}\right|+\frac{1}{4}\right|+\frac{1}{8}\right|+\frac{1}{16}\right|+\frac{1}{32}\right|+\frac{1}{64} \\ p\left|p\left|p\left|p\left|p\left|p\left|p+\frac{1}{2}\right|+\frac{1}{4}\right|+\frac{1}{8}\right|+\frac{1}{16}\right|+\frac{1}{32}\right|+\frac{1}{64}\right|+\frac{1}{128} \end{pmatrix}$$

```
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.

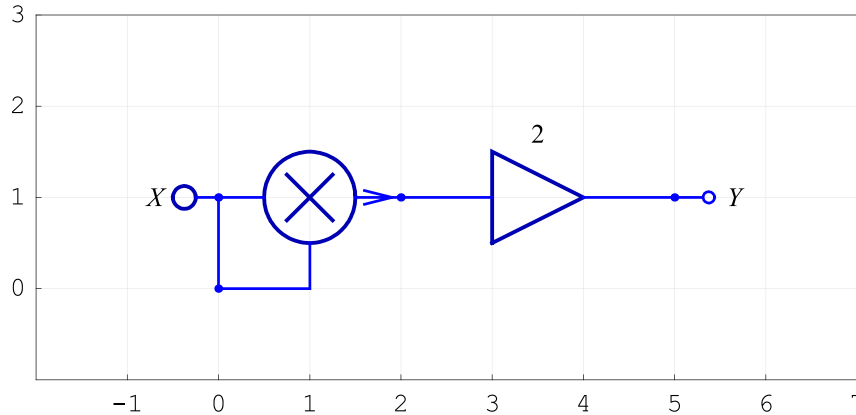
This section assumes that you have already loaded *SchematicSolver*. Otherwise, you can load the package with

```
In[57]:= Needs["SchematicSolver`"];
```

Here is an example modulator system:

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

```
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:

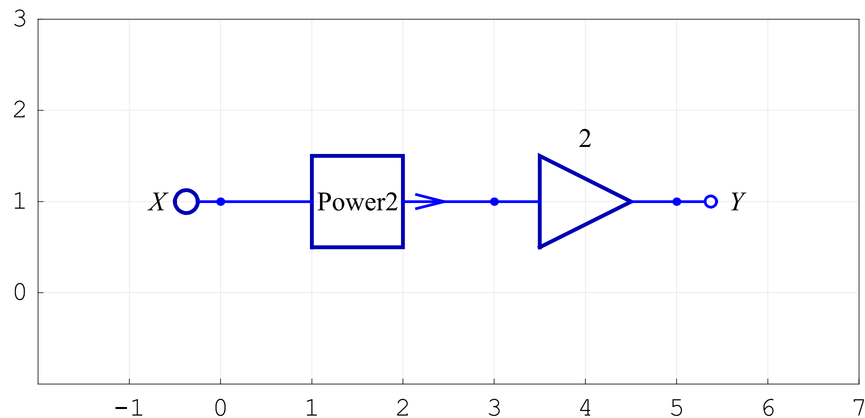
```
In[68]:= outSamples // TrigReduce

Out[68]= {1 - Cos[4 Fx n π]}
```

©2003-2004 Prof Miroslav Lutovac & Prof Dejan Tosic. All rights reserved.

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
```