TransMan*

[Version 2.0 – *12. 01. 2004*]

Simulink block library for the demonstration of transient management methods

Budapest University of Technology and Economics, Budapest, Hungary
Department of Measurement and Information Systems

http://www.mit.bme.hu/

The latest version of this document can be found at:
http://www.mit.bme.hu/~khazy/publist.html

* *Funded, in part, by the DARPA ITO's Software-Enabled Control Program under AFRL contract F33615-99-C-3611.*

Table of contents

1. Introduction

2. Blocks in alphabetical order

3. Generic notes

The **TransMan** block library uses it own special data structures because of execution speed advantages, transparency, and to provide flexible variable size vectors and matrixes. The defined data structures can be found in Table 3.1.

Type name	Description
Vector	Variable length vector with real elements
CpxVector	Variable length vector with complex elements
Matrix	Variable sized matrix with real elements
CpxMatrix	Variable sized matrix with complex elements
FilterParam	Structure containing a filter specification
ResParam	Structure containing resonator parameters, see Chapter 0
TransferFn	Transfer function
StateMtcs	State-Space Matrices
QDFilterParam	Parameter matrices of a quadratic nonlinear filter
LatticeParam	Parameter vectors of a lattice filter (reflection and tapping consts)

Table 3.1: Data structures in TransMan

All structures have an "empty" value, which may have special meaning during operation. Structures that are not initialized have this "empty" value by default. The structures are stored in a non-MATLAB compatible format, but they can be converted to a MATLAB compatible format in a way described in the next sub-sections.

3.1. Filter specification

`FilterParam(`<filter class>`, `<filter type>`, `<filter order>`, `<frequencies>`, `<[R_P R_S]>`),`

where the parameters are equivalent to the corresponding filter designer command parameters of MATLAB. The terms R_P and R_S are ripple parameters, and their meaning and availability in the function of filter type is detailed in Table 3.1.1.

R_P : Pass-band ripple

R_S : Stop-band ripple

Filter class spec.	Filter class	R_P	R_S
'butter' or 0	Butterworth	-	-
'bessel' or 1	Bessel	-	-
'cheby1' or 2	Chebyshev I.	✓	-
'cheby2' or 3	Chebyshev II.	✓	-
'ellip' or 4	Elliptic	✓	✓

Table 3.1.1: Filter class specification

The description of the filter types can be found in Table 3.1.2, where ω is the corresponding cut-off frequency, and x is parameter that is not required.

Filter type spec.	Filter type	frequencies
'lowpass' or 0	Low-pass	[ω_0 x]
'highpass' or 1	High-pass	[ω_0 x]
'bandpass' or 2	Band-pass	[ω_1 ω_2]
'bandstop' or 3	Band-stop	[ω_1 ω_2]

Table 3.1.2: Availability of filter frequency parameters

Examples:

1. 4th order, low-pass Butterworth filter with the cut-off frequency of 0.2:
   `FilterParam('butter', 0, 4, 0.2);`
2. 8th order band-stop Chebyshev type I. filter, with cut-off frequencies 0.25 and 0.4:
   `FilterParam(2, 3, 8, [0.25 0.4], 0.2);`

3.2. Resonator parameters

`ResParam(z, w, r, d)`

The description of parameters can be found in .

3.3. Transfer function parameters

`TransferFn(`<numerator>, <denominator>`)`

Transfer function, specified by the numerator and denominator coefficients in the form of row vectors.

3.4. State-space matrices

`StateMtcs(A, B, C, D)`

The <A, B, C, D> matrixes of the state-space description of the system.

3.5. Quadratic filter parameters

`QDFilterParam(m, M)`

The description of parameters can be found in Chapter 4.4.6.

3.6. Lattice filter parameters

`LatticeParam(k, v)`

Specifies the reflection and tapping coefficients of a lattice filter. See Chapter 4.4.5.

4. Detailed description of the blocks

Some block parameters are identical for all blocks; therefore, it is reasonable to specify their effects on the operation of blocks here.

Sample time

The sample time parameter allows setting the sample time of the blocks, which is primarily used by the solver. If the sample time is set to –1 the block inherits the sample time from its neighbors.

Trigger specification

Most of the blocks have trigger inputs, which can be used to start specific activities inside the blocks by connecting trigger signals to them. The trigger condition is configurable. The possible trigger conditions are:

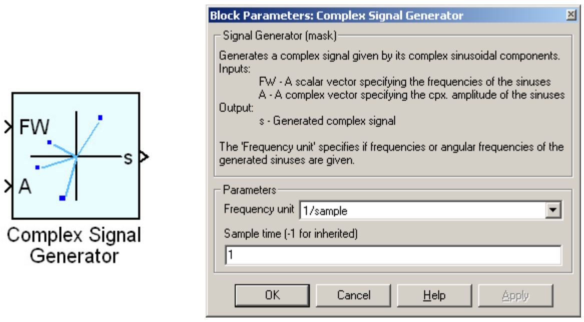
Trigger specification	Trigger condition
′high (H : above 0.0)′	Triggers if the trigger signal is high, i.e. above 0.0
′low(L : below or eq 0.0)′	Triggers if the trigger signal is low, i.e. below 0.0
′rising edge (LH)′	Triggers if the trigger signal has a rising edge, i.e. it goes from low to high
′falling edge (HL)′	Triggers if the trigger signal has a falling edge, i.e. it goes from high to low
′any edge (invert)′	Triggers if the trigger signal has any type of edge, i.e. it goes from high to low or from low to high

Table 4.1: Trigger conditions

The values above zero are treated as ′high′ the others as ′low′.

4.1. Signal sources

4.1.1. Complex Signal Generator



Signal Generator (mask)

Generates a complex signal given by its complex sinusoidal components.
Inputs:
FW - A scalar vector specifying the frequencies of the sinuses
A - A complex vector specifying the cpx. amplitude of the sinuses
Output:
s - Generated complex signal

The 'Frequency unit' specifies if frequencies or angular frequencies of the generated sinuses are given.

Parameters
Frequency unit: 1/sample
Sample time (-1 for inherited): 1

Fig. 4.1.1: The Complex Signal Generator block

The block generates a signal composed of multiple sinuses.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*FW*	Vector	A scalar vector specifying the frequencies of the sinuses
	A	CpxVector	A complex vector specifying the complex amplitude of the sinuses
Output	*s*	double (C)	Complex signal

Table 4.1.1: The interface of the Complex Signal Generator

Parameters:

⇩ Parameter	Value	Description
Frequency unit	1/sample	The frequency is specified in relative frequency
	radians/sample	The frequency is specified as angular frequency

Table 4.1.2: The parameters of the Complex Signal Generator

Operation:

The *Frequency unit* specifies if frequencies or angular frequencies of the generated sinuses are given. The output is defined by the equation given (Table 4.1.3) as:

Frequency unit ⇨	1/sample	radians/sample
s=	$\sum_{i=1}^{N} A(i)e^{2\pi j \cdot FW(i)T}$	$\sum_{i=1}^{N} A(i)e^{j \cdot FW(i)T}$

Table 4.1.3: The output relation of the Complex Signal Generator

where $N = \min[\text{length}(A), \text{length}(FW)]$.

If the vectors *A* and *FW* have different sizes, then the number of the generated sinus components will be the minimum of the vector sizes.

Remark:

1. If *FW*, or *A* inputs are not connected, or zero length, then the output 0.

Example:

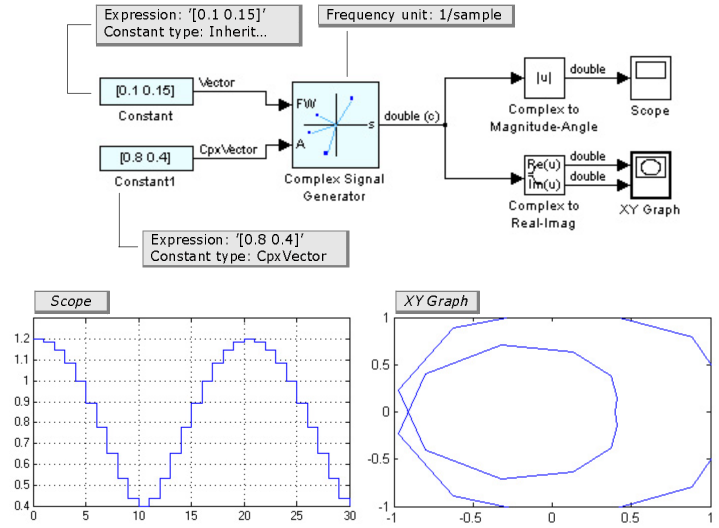


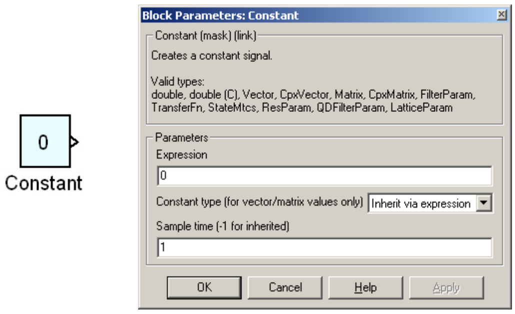
Fig. 4.1.2: Example for the Complex Signal Generator (`.\Examples\Simple\ex_csgen.mdl`)

Fig. 4.1.2 shows a signal consisting of two sinusoid components generated by the complex signal generator block. The real component of the generated signal is plotted and the movement on the complex plane are shown also. The specified parameters are:

$$A_1 = 0.8 \quad f_1 = 0.1$$
$$A_2 = 0.4 \quad f_2 = 0.15$$

See Chapter 4.1.2 (p. 10.), describing the *Constant* block.

4.1.2. Constant



Constant (mask) (link)

Creates a constant signal.

Valid types:
double, double (C), Vector, CpxVector, Matrix, CpxMatrix, FilterParam, TransferFn, StateMtcs, ResParam, QDFilterParam, LatticeParam

Parameters

Expression: 0

Constant type (for vector/matrix values only): Inherit via expression

Sample time (-1 for inherited): 1

Fig. 4.1.3: The Constant block

This block defines constant signals in **TransMan**. The *Expression* parameter specifies the constant output value. By using the *Expression* parameter it is possible to generate the internal data structures used by **TransMan** as defined in Chapters from 3.1 to 3.6. Vector and matrix specification automatically defines *Vector*, *CpxVector*, *Matrix* and *CpxMatrix* structures and signals (see Fig. 4.1.2 as example). This behavior is different to the Simulink constant block, which creates buses of vectors and matrices instead.

Parameters:

⇩ *Constant type*	Can convert expression of type…
Vector	double
CpxVector	double, double (C), Vector
Matrix	double, Vector
CpxMatrix	double, double (C), Vector, CpxVector, Matrix

Table 4.1.4: The parameter of the Constant Block and the forced types

Some types can be forced to be created by the *Constant type* parameter. The description of the (convertible) values and the corresponding types can be seen in Table 4.1.4.

The output can be any **TransMan** structure with the following example ways of specifications (see Table 4.1.5)

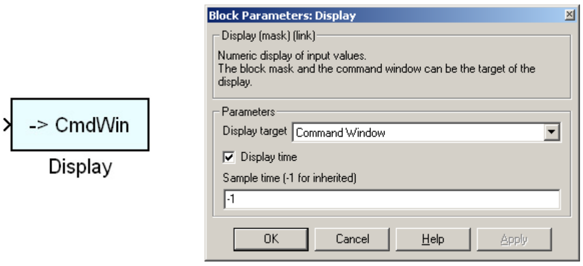
⇩ Type of *Expression*	Specification examples
Double	'0'
double (C)	'2+5i'
Vector	'[2 4]' (row vector), '[7; 2]' (column vector)
CpxVector	'[2+5i 5.4]'
Matrix	'[1 2; 3 4]'
CpxMatrix	'complex([4 2.3; 0.2 1.3])'
FilterParam	'FilterParam(0, 0, 4, 0.2)'
ResParam	'ResParam(z, w, r, d)'
TransferFn	'TransferFn(n, d)'
StateMtcs	'StateMtcs(A, B, C, D)'
QDFilterParam	'QDFilterParam(m, M)'
LatticeParam	'LatticeParam(k, v)'

Table 4.1.5: Constant types and their specification (examples)

Example: Fig. 4.1.2, p. 9.

4.2. Signal sinks

4.2.1. Display



Display (mask) (link)

Numeric display of input values.
The block mask and the command window can be the target of the display.

Parameters

Display target: Command Window

☑ Display time

Sample time (-1 for inherited)

-1

Fig. 4.2.1: The Display block

The block displays the values of the input in the command window or on the block mask.

Interface:

⇩ Connector	⇩ Name	Type	Description
Input	-	<Any valid **TransMan** type>	Line to display

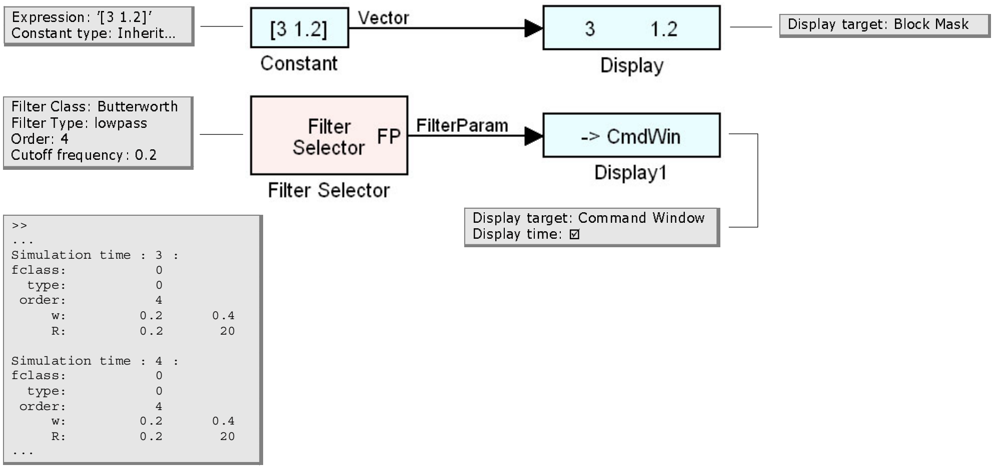
Table 4.2.1: The interface of the Display block

Parameters:

⇩ Parameter	Values	Description
Display target	Command Window	Display in Command Window
	Block Mask	Display on block mask
Display time	☐	Do not display simulation time
	☑	Display simulation time

Table 4.2.2: The parameters of the Display block

Example:



Expression: '[3 1.2]'
Constant type: Inherit...

Display target: Block Mask

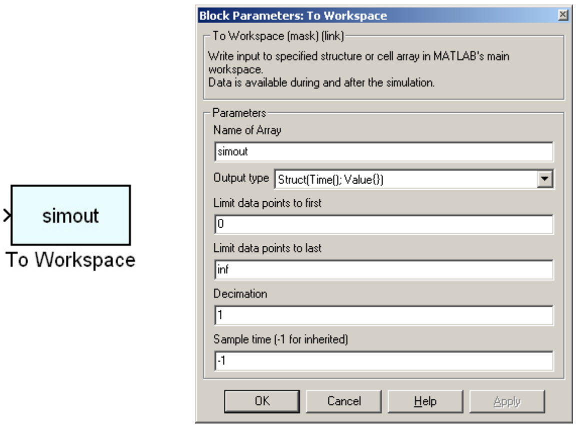
Filter Class: Butterworth
Filter Type: lowpass
Order: 4
Cutoff frequency: 0.2

Display target: Command Window
Display time: ☑

```
>>
...
Simulation time : 3 :
fclass:          0
  type:          0
 order:          4
     w:        0.2       0.4
     R:        0.2        20

Simulation time : 4 :
fclass:          0
  type:          0
 order:          4
     w:        0.2       0.4
     R:        0.2        20
...
```

Fig. 4.2.2: Example for the Display block (`.\Examples\Simple\ex_disp.mdl`)

4.2.2. *To Workspace*



Block Parameters: To Workspace

To Workspace (mask) (link)

Write input to specified structure or cell array in MATLAB's main workspace.
Data is available during and after the simulation.

Parameters

Name of Array: simout

Output type: Struct(Time(); Value{})

Limit data points to first: 0

Limit data points to last: inf

Decimation: 1

Sample time (-1 for inherited): -1

Fig. 4.2.3: The To Workspace block

Write input to a specified structure or cell array in MATLAB's main workspace. Data is available during the simulation.

Interface:

⇩ Connector	⇩ Label	Type	Description
Input	-	<Any valid **TransMan** type>	Line to store

Table 4.2.3: The interface of the To Workspace block

Parameters:

⇩ Parameter	Value(s)	Description
Name of Array	<Valid MATLAB id>	The name of the variable

Output type	Struct(Time(); Value{})	Structure of array of time and cell array of values:
		`Structure of fields:`
		`Time: [N×1 double]`
		`Value: {N×1 cell}`

	Struct()(Time; Value)	Array of structures of time and value:
		`N×1 struct array of fields:`
		`Time`
		`Value`

	Value{}	Cell array of values: `{N×1 cell}`
Limit data point to first	[0..<Stop time>]	The first step of the storing interval
Limit data point to last	[0..<Stop time>]	The last step of the storing interval
Decimation	[1..]	Decimation

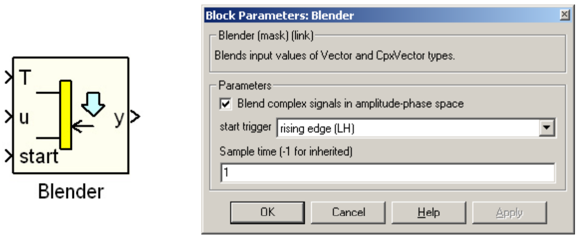
Table 4.2.4: The parameters of the To Workspace block

Remark:

1. With empty input structure the block creates empty cell values. Empty vector and matrix inputs yield empty matrices.

4.3. Signal operations

4.3.1. Blender



Blender (mask) (link)

Blends input values of Vector and CpxVector types.

Parameters

☑ Blend complex signals in amplitude-phase space

start trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.3.1: The Blender block

The *Blender* blends the original incoming signal to the new incoming signal using linear interpolation in *T* steps. The blending process starts on the *start* trigger. The *complex signals in amplitude-phase space* checkbox specifies how the blending process handles complex numbers. The default behavior is to blend the real and imaginary components, if the checkbox is selected the amplitude and phase of the complex numbers are interpolated linearly.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*T*	Double	The numer of blending steps (1..)
	u	Vector/CpxVector	The signal to be blended
	start	Double	Trigger
Output	*y*	Vector/CpxVector	Blended signal

Table 4.3.1: The interface of the Blender block

Parameters:

⇩ Parameter	Values	Description
Blend complex signals in amplitude-phase space	☐	real-imaginary blending
	☑	Amplitude-phase blending

Table 4.3.2: The parameters of the Blender block

Operation:

The *Blender* block is capable to blend vectors. The *Blender* block stores its actual output; therefore, input *u* has no effect on its operation except when the *start* input triggers, when *T* and *u* is read in, and the blending process starts from the original output to the new output specified by *u*. In *T* steps the output reaches the value specified by *u*.

Remarks:

1. If input *u* is not connected the output is a 0 length vector.
2. If input $T < 1$ *T* is set to 1 internally.
3. During blending the block is not sensitive to new *start* triggers.

At the start of simulation the initial value of *u* is automatically loaded into the block.

Example of blending complex signals:

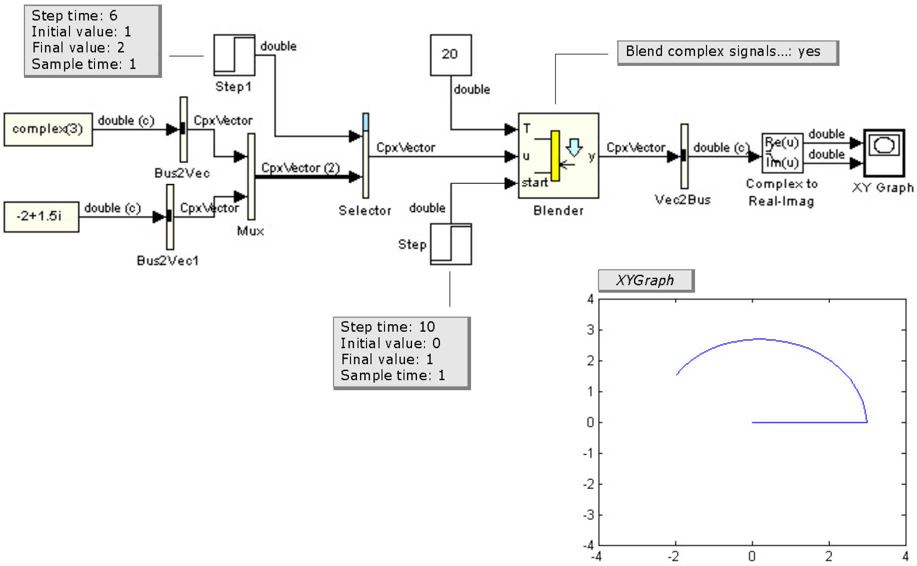
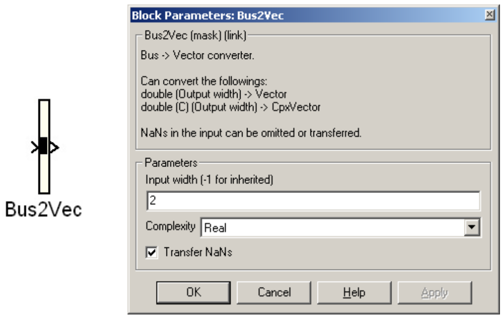


Fig. 4.3.2: Example for blending complex signals (`.\Examples\Simple\ex_blender.mdl`)

4.3.2. Bus-Vector Converter (Bus2Vec)



Bus -> Vector converter.

Can convert the followings:
double (Output width) -> Vector
double (C) (Output width) -> CpxVector

NaNs in the input can be omitted or transferred.

Input width (-1 for inherited): 2
Complexity: Real
☑ Transfer NaNs

Fig. 4.3.3: The Bus-Vector Converter block

The *Bus-Vector Converter* block converts real or complex buses to *Vector* or *CpxVector* structure signal.

Interface:

Complexity ⇨	Real	Complex
Input type	double(<*Input width par.*>)	double(<*Input width par.*>) (C)
Output type	Vector	CpxVector

Table 4.3.3: The interface of the Bus-Vector Converter block

Parameters:

⇩ Parameter	Values	Description
Input width	[1..]	Specifies the width of the input
	-1	The width of the input is inherited from the connecting block
Complexity		See Table 4.3.3
Transfer NaNs	☐	Omit NaN values
	☑	Include NaN values

Table 4.3.4: The parameters of the Bus-Vector Converter block

The *Transfer NaNs* parameter specifies the conversion of NaN (Not-a-Number) value transformation during the conversion. The default behavior is omitting the NaNs, and forming a shorter vector. If the *Transfer NaNs* parameter is selected the NaNs are converted, and the output vector will contain the NaN values.

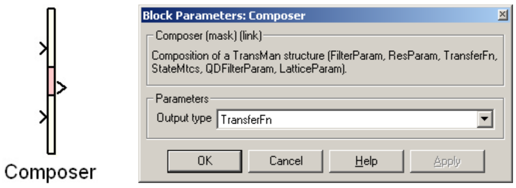
Remarks:

1. The block creates a column vector.
2. With unconnected input the block outputs an empty vector.

For examples see Fig. 4.3.2 (p. 14.).

See also Chapter 4.3.9. (p. 23.), which details the *Vec2Bus* block.

4.3.3. Composer



Composition of a TransMan structure (FilterParam, ResParam, TransferFn, StateMtcs, QDFilterParam, LatticeParam).

Output type: TransferFn

Fig. 4.3.4: The Composer block

The *Composer* block constructs **TransMan** structures from its elements. The actual meaning of the parameter and the corresponding input types are given in Table 4.3.5 and Table 4.3.6.

Output type ⇨		FilterParam	ResParam	TransferFn	StateMtcs	QDFilterParam	LatticeParam
Inputs	*1*	double	CpxVector	Vector	Matrix	Vector	CpxVector
	2	double	CpxVector	Vector	Matrix	Matrix	Vector
	3	double	Vector	-	Matrix	-	-
	4	double (2)	Double	-	Matrix	-	-
	5	double (2)	-	-	-	-	-
Output type		FilterParam	ResParam	TransferFn	StateMtcs	QDFilterParam	LatticeParam

Table 4.3.5: Input and output types of the Composer block

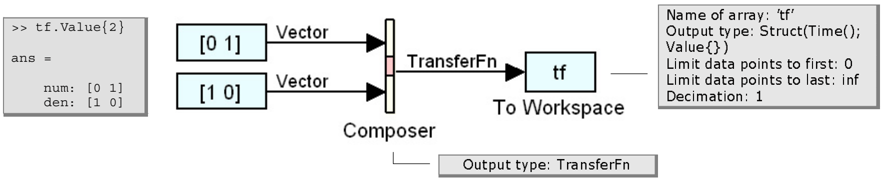
		⇩ Description					
Output type ⇨		FilterParam	ResParam	TransferFn	StateMtcs	QDFilterParam	LatticeParam
Inputs	*1*	class	z vec.	numerator	A mtx.	m, linear tap. v.	k, refl. coeff.
	2	type	w vec.	denominator	B mtx.	M, quadratic tr.	v, tapping ce.
	3	order	r vec.	-	C mtx.	-	-
	4	cutoff freq.s	D	-	D mtx.	-	-
	5	Rp, Rs	-	-	-	-	-

Table 4.3.6: The input description of the Composer block

Remarks:

1. If an input is empty the output becomes an empty structure.
2. The block does not check the validity of the constructed structures.

Example:



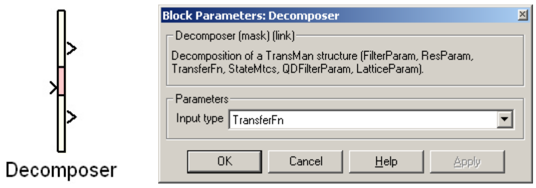
```
>> tf.Value{2}

ans =

    num: [0 1]
    den: [1 0]
```

Name of array: 'tf'
Output type: Struct(Time(); Value{})
Limit data points to first: 0
Limit data points to last: inf
Decimation: 1

Output type: TransferFn

Fig. 4.3.5: Example for Composer (`.\Examples\Simple\ex_composer.mdl`)

4.3.4. Decomposer



Block Parameters: Decomposer

Decomposer (mask) (link)

Decomposition of a TransMan structure (FilterParam, ResParam, TransferFn, StateMtcs, QDFilterParam, LatticeParam).

Parameters

Input type: TransferFn

Fig. 4.3.6: The Decomposer block

The *Decomposer* block extracts the internal data form a **TransMan** structure. The actual meaning of the parameter and the corresponding input types are given in Table 4.3.7 and Table 4.3.8.

Input type ⇨		FilterParam	ResParam	TransferFn	StateMtcs	QDFilterParam	LatticeParam
Input type		FilterParam	ResParam	TransferFn	StateMtcs	QDFilterParam	LatticeParam
Outputs	*1*	double	CpxVector	Vector	Matrix	Vector	CpxVector
	2	double	CpxVector	Vector	Matrix	Matrix	Vector
	3	double	Vector	-	Matrix	-	-
	4	double (2)	Double	-	Matrix	-	-
	5	double (2)	-	-	-	-	-

Table 4.3.7: Input and output types of the Decomposer block

		⇩ Description					
Input type ⇨		FilterParam	ResParam	TransferFn	StateMtcs	QDFilterParam	LatticeParam
Outputs	*1*	class	z vec.	numerator	A mtx.	m, linear tap. v.	k, refl. coeff.
	2	type	w vec.	denominator	B mtx.	M, quadratic tr.	v, tapping ce.
	3	order	r vec.	-	C mtx.	-	-
	4	cutoff freq.s	D	-	D mtx.	-	-
	5	Rp, Rs	-	-	-	-	-

Table 4.3.8: The output description of the Decomposer block

Remarks:

1. If the input is an empty structure the outputs are also empty.
2. The block does not check the validity of the structure.

Example:

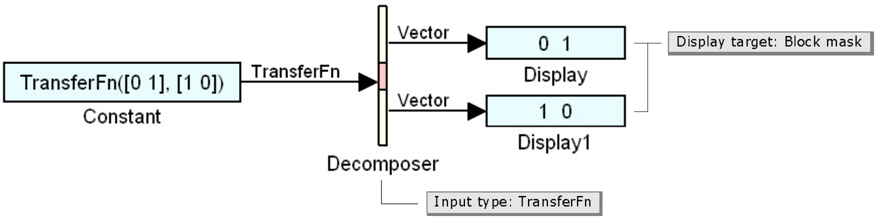
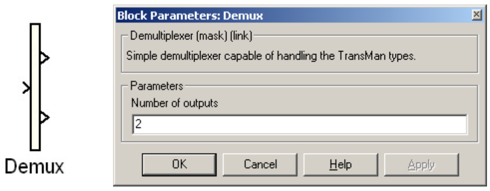


Fig. 4.3.7: Example for Decomposer (`.\Examples\Simple\ex_decomposer.mdl`)

4.3.5. DeMux



Demultiplexer (mask) (link)

Simple demultiplexer capable of handling the TransMan types.

Parameters

Number of outputs

2

Fig. 4.3.8: The Demultiplexer block

Simple demultiplexer capable of handling the valid types of **TransMan**.

Interface:

⇩ Connector	Type	Description
Input	<Any valid **TransMan** type*> (<*Number of outputs*>)	Bus input
Outputs	<Any valid **TransMan** type*>	Bus line outputs

* The input and the output have the same type

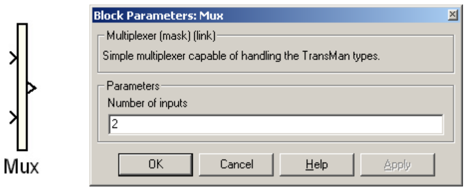
Table 4.3.9: The interface of the Demultiplexer block

Parameters:

⇩ Parameter	Values	Description
Number of outputs	[1..]	Width of the input and the number of the outputs

Table 4.3.10: The parameters of the Demultiplexer block

4.3.6. Mux



Block Parameters: Mux

Multiplexer (mask) (link)

Simple multiplexer capable of handling the TransMan types.

Parameters

Number of inputs

2

Fig. 4.3.9: The Multiplexer block

Simple multiplexer capable of handling the valid types of **TransMan**.

Interface:

⇩ Connector	Type	Description
Inputs	<Any valid **TransMan** type*>	Bus line inputs
Output	<Any valid **TransMan** type*> (<*Number of inputs*>)	Bus output

* The input and the output have the same type

Table 4.3.11: The interface of the Multiplexer block

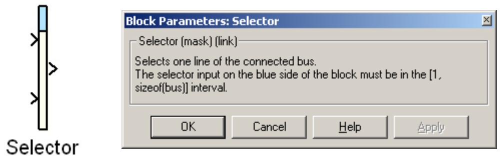
Parameters:

⇩ Parameter	Values	Description
Number of intputs	[1..]	Width of the output and the number of the inputs

Table 4.3.12: The parameters of the Multiplexer block

Example: Fig. 4.3.2, p. 14.

4.3.7. Selector



Block Parameters: Selector

Selector (mask) (link)

Selects one line of the connected bus.
The selector input on the blue side of the block must be in the [1, sizeof(bus)] interval.

Fig. 4.3.10: The Selector block

Interface:

⇩ Connector	⇩ Num	Type	Description
Inputs	*1*	double	The selected line
	2	<Any valid TransMan type*>	Input bus
Output		<Any valid TransMan type*>	Selected line

* The input and the output have the same type

Table 4.3.13: The interface of the Selector

The Selector selects a line from the input bus and puts its value onto the output. The line is selected by the input on the side of the blue field on the mask. The range of the line numbers is from 1 to the number of the lines (bus width). Values outside this range are treated as the nearest range limit.

Example:

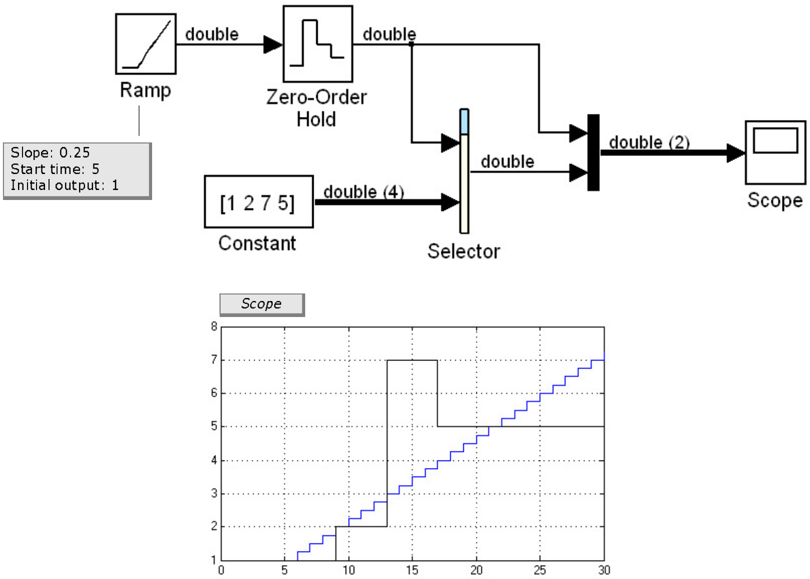
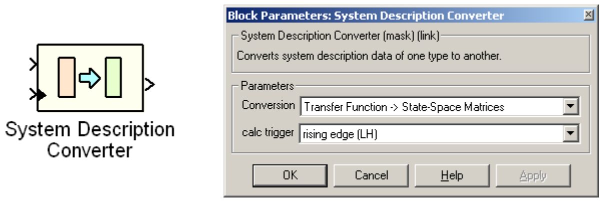


Fig. 4.3.11: Example for the Selector block (`.\Examples\Simple\ex_selector.mdl`)

4.3.8. System Description Converter



System Description Converter (mask) (link)

Converts system description data of one type to another.

Parameters

Conversion: Transfer Function -> State-Space Matrices

calc trigger: rising edge (LH)

Fig. 4.3.12: The System Description Converter block

The *System Description Converter* block converts system description data of one type to another.

Interface:

Connector name (num.) ⇨	- (1)	▶ (2)	- (1)
⇩ *Conversion par.*	⇩ Input type		⇩ Output type
Transfer Function -> State-Space Matrices	TransferFn	double	StateMtcs
State-Space Matrices -> Transfer Function	StateMtcs		TransferFn
Transfer Function -> Lattice Parameters (Basic)	TransferFn		LatticeParam
Transfer Function -> Lattice Parameters (Reversed Basic)			
Transfer Function -> Lattice Parameters (Normalized)			
Transfer Function -> Lattice Parameters (Rev.d. Norm.d.)			

Table 4.3.14: The parametrized interface of the System Description Converter block

The possible convertible structures are shown in Table 4.3.14.

Operation:

The block uses MATLAB function calls to calculate the output for some conversions. The functions used are listed in Table 4.3.15.

⇩ *Conversion par.*	⇩ MATLAB function
Transfer Function -> State-Space Matrices	tf2ss
State-Space Matrices -> Transfer Function	ss2tf
Transfer Function -> Lattice Parameters (Basic)	*
Transfer Function -> Lattice Parameters (Reversed Basic)	*
Transfer Function -> Lattice Parameters (Normalized)	*
Transfer Function -> Lattice Parameters (Rev.d. Norm.d.)	*

* Conversion functions are implemented by the block internally

Table 4.3.15: The MATLAB functions called by the System Description Converter block

Conversion algorithms (see [3] for further):

$$\mathrm{TF} = \frac{B(z)}{A(z)} \qquad N = \mathrm{length}[A(z)]$$

The computation of Schur polynomials and *LP.k*:

$$\Phi_{i-1}(z) = \frac{z^{-1}[\Phi_i(z) - k_i\Phi_i^*(z)]}{s_i} \qquad k[i] = \frac{\Phi_i(0)}{\Phi_i^*(0)} \qquad \begin{matrix}\Phi_N(z) = A(z) \\ \Phi_i^*(z) = z^i\Phi_i(z^{-1})\end{matrix}$$

where

$s_i = 1 - k[i]^2$ for basic filters, and

$s_i = \sqrt{1 - k[i]^2}$ for normalized filters

LP.v is computed by solving the following equations:

$B(z) = \sum_{i=0}^{N} v[i] \cdot \Phi_i(z)$ for non-reversed filters, and

$B^*(z) = \sum_{i=0}^{N} v[i] \cdot \Phi_i(z)$ for reversed filters, where $B^*(z)$ is the mirrored $B(z)$.

The implementation of the algorithm:

Input: num : numerator, den : denominator. Both has coeff.s in descending order of *z* in them and are row vectors. The indices goes from 0.

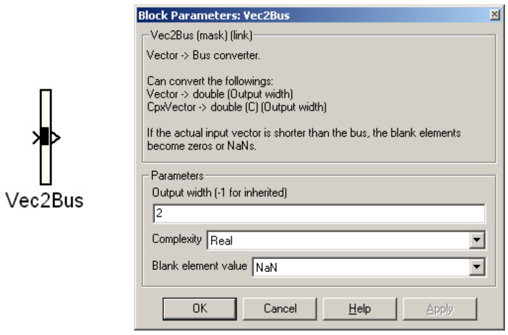
```
If TF is unconnected or invalid then exit
Set LP empty
d=mirror(den) (d[0] is the coeff. of the constant term, d[1] is for the coeff. of first order of z, etc.)
N=length(d)-1
n=num
Append n with zeros to length N+1
If Conversion==Transfer Function -> Lattice Parameters (Basic) or
        Conversion==Transfer Function -> Lattice Parameters (Normalized) then n=mirror(n)
The output k will be N length, and v will be N+1 length
Set the elements of v to zero
For i=N to 1
        v[i]=n[i]/d[i]
        n=n-d*v[i]
        k[i-1]=d[0]/d[i]
        if Conversion==Transfer Function -> Lattice Parameters (Basic) or
                Conversion==Transfer Function -> Lattice Parameters (Reversed Basic) then
                $s = 1 - (k[i-1])^2$
        else
                $s = \sqrt{1 - (k[i-1])^2}$
        $d = \frac{d - k[i-1] \cdot \mathrm{mirror}(d)}{s}$
        Shift d to left (d[0] gets the value in d[1], d[1] gets d[2], etc.)
v[0]=n[0]/d[0]
LP.k = k
LP.v = v
```

Remarks:

1. Empty output structures are created from empty input structures.
2. Invalid structure values invoke MATLAB error messages.

Example: Fig. 4.4.13, p. 30.

4.3.9. Vector-Bus Converter (Vec2Bus)



Vec2Bus (mask) (link)

Vector -> Bus converter.

Can convert the followings:
Vector -> double (Output width)
CpxVector -> double (C) (Output width)

If the actual input vector is shorter than the bus, the blank elements become zeros or NaNs.

Parameters

Output width (-1 for inherited): 2

Complexity: Real

Blank element value: NaN

Fig. 4.3.13: The Vector-Bus Converter block

The *Vector-Bus Converter* block converts *Vector* or *CpxVector* structure signals to real or complex buses.

Interface:

Complexity ⇨	Real	Complex
Input type	Vector	CpxVector
Output type	double(<*Output width par.*>)	double(<*Output width par.*>) (C)

Table 4.3.16: The interface of the Vector-Bus Converter

Parameters:

⇩ Parameter	Values	Description
Output width	[1..]	Specifies the width of the output
	-1	The width of the output is inherited from the connecting block
Complexity		See Table 4.3.16
Blank element value	Zero	Zero valued blank elements
	NaN	NaN valued blank elements

Table 4.3.17: The parameters of the Vector-Bus Converter

Operation:

If the width of the input vector less than the width specified by the *Output width* parameter then the vector is padded with blank elements in the bus. The blank elements can have the value 0 or NaN.

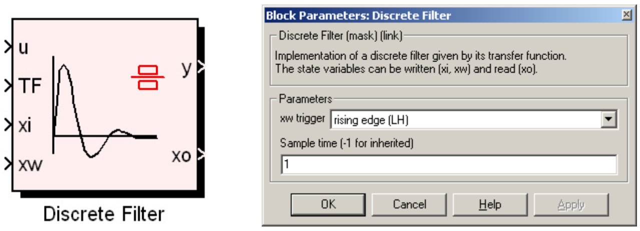
Remarks:

1. Empty or unconnected input yields a bus with the blank element value on its lines.

For examples see Fig. 4.3.2 (p. 14.).

4.4. Discrete Filters

4.4.1. Discrete Filter



Block Parameters: Discrete Filter

Discrete Filter (mask) (link)

Implementation of a discrete filter given by its transfer function.
The state variables can be written (xi, xw) and read (xo).

Parameters

xw trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.4.1: The Discrete Filter block

The block implements a discrete filter given by its transfer function.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	double	Input signal
	TF	TransferFn	Transfer function of the filter
	xi	Vector	State vector input
	xw	double	State vector write trigger
Output	*y*	double	Signal output
	xo	Vector	State vector output

Table 4.4.1: The interface of the Discrete Filter block

Operation:

The Transposed Direct II. Structure is implemented by the block (Fig. 4.4.2).

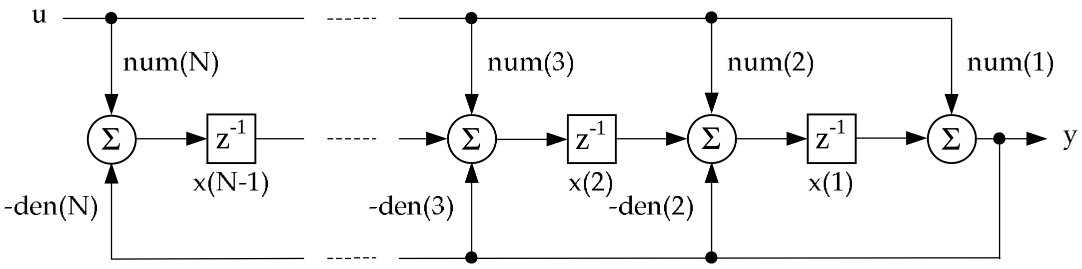


Fig. 4.4.2: The Transposed Direct II. structure (Discrete Filter)

The state vector can be set by the *xi* and the *xw* inputs and can be read from the *xo* output. If the state vector input is wider than the actual (internal) state vector then the elements with higher indices are discarded. Smaller input vectors are padded with zeros.

Remarks:

1. If the *TF* input is unconnected or empty the output will be zero.
2. The state vector stays unchanged when the *xi* input is unconnected.

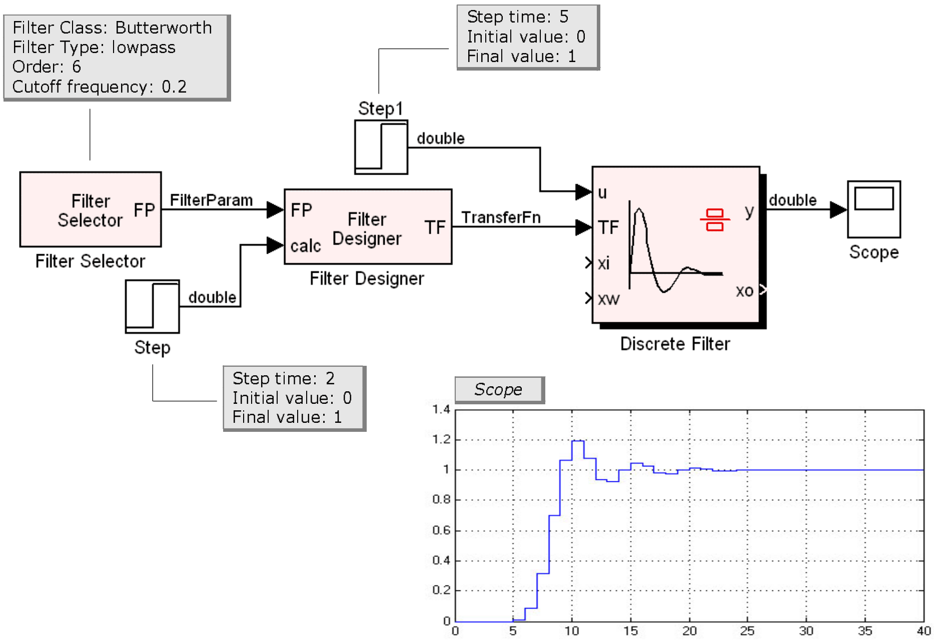


Fig. 4.4.3: Example for Discrete Filter (`.\Examples\Simple\ex_dfilter.mdl`)

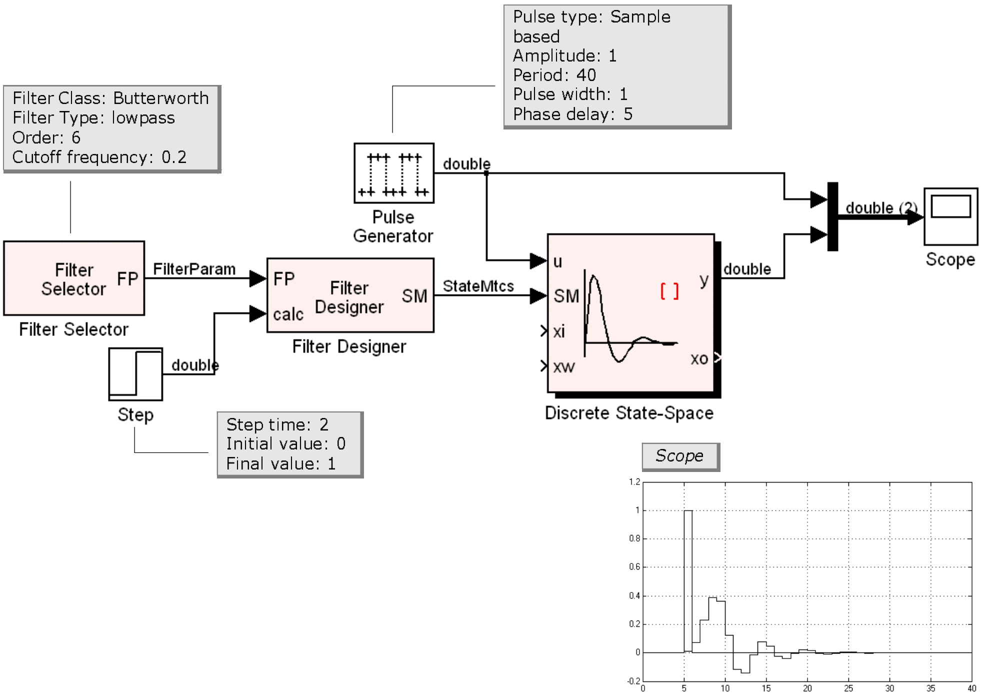
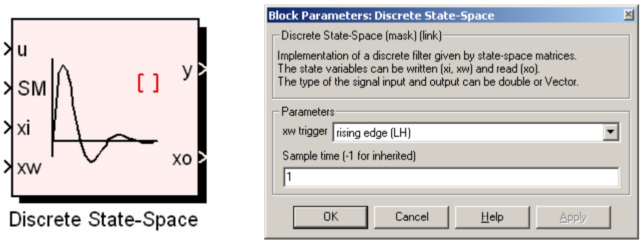


Fig. 4.4.4: Example for Discrete State-Space (`.\Examples\Simple\ex_dss.mdl`)

4.4.2. *Discrete State-Space*



Block Parameters: Discrete State-Space

Discrete State-Space (mask) (link)

Implementation of a discrete filter given by state-space matrices.
The state variables can be written (xi, xw) and read (xo).
The type of the signal input and output can be double or Vector.

Parameters

xw trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.4.5: The Discrete State-Space block

The block implements a discrete filter given by its state-space matrices.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	Vector	Input signal
	SM	StateMtcs	State-sp. matrices of the filter
	xi	Vector	State vector input
	xw	double	State vector write trigger
Output	*y*	Vector	Signal output
	xo	Vector	State vector output

Table 4.4.2: The interface of the Discrete State-Space block

Operation:

This block can simulate multiple-input-multiple-output (MIMO) systems. The z-domain description formulas of such a system are:

$$\begin{aligned} \underline{X}(z)z^{-1} &= \underline{\underline{A}}\,\underline{X}(z) + \underline{\underline{B}}\,\underline{U}(z) \\ \underline{Y}(z) &= \underline{\underline{C}}\,\underline{X}(z) + \underline{\underline{D}}\,\underline{U}(z) \end{aligned},$$

where the notations are:

$\underline{X}(z)$: State vector, $\underline{U}(z)$: Input vector, $\underline{Y}(z)$: Output vector; $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, $\underline{\underline{D}}$: The corresponding state-space matrices (given by the incoming *StateMtcs* structure).

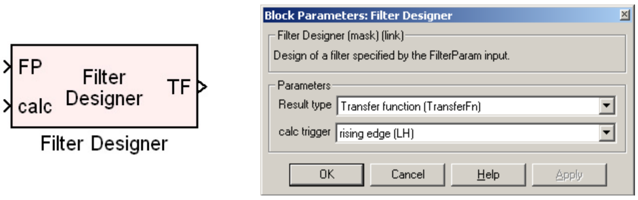
The state vector can be set by the *xi* and the *xw* inputs and can be read from the *xo* output. If the state vector input is wider than the actual (internal) state vector then the elements with higher indices are discarded. Smaller input vectors are padded with zeros.

Remarks:

1. If the *SM* input is unconnected or empty the output will be zero.
2. The state vector stays unchanged when the *xi* input is unconnected.

Example: Fig. 4.4.4 (p. 25.).

4.4.3. FilterDesigner



Filter Designer (mask) (link)

Design of a filter specified by the FilterParam input.

Parameters

Result type: Transfer function (TransferFn)

calc trigger: rising edge (LH)

Fig. 4.4.6: The Filter Designer block

The Filter Designer block designs the digital filter specified by the *FilterParam* structure incoming on input port *FP*. The output can be either the transfer function or the state-space matrices of the filter. The calculation is triggered by the *calc* input. The contents of the *FilterParam* structure are described in Chapter 3.1. (p. 5.).

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*FP*	FilterParam	Input
	Calc	double	Trigger of calculation
Output	*TF/SM*	TransferFn	Transfer function
		StateMtcs	State-space matrices

Table 4.4.3: The interface of the Filter Designer block

Parameters:

⇩ Parameter	Values	Description
		⇩ Output type
Result type	Transfer function	TransferFn
	State-Space matrices	StateMtcs

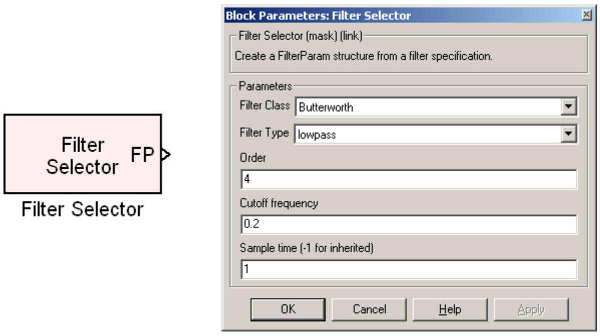
Table 4.4.4: The parameters of the Filter Designer

Remark:

1. With unconnected or empty structure input the block creates empty or unchanged output structure.

Examples: Fig. 4.4.3 (p. 25.), Fig. 4.4.4 (p. 25.).

4.4.4. FilterSelector



Block Parameters: Filter Selector

Filter Selector (mask) (link)

Create a FilterParam structure from a filter specification.

Parameters

Filter Class: Butterworth

Filter Type: lowpass

Order: 4

Cutoff frequency: 0.2

Sample time (-1 for inherited): 1

Fig. 4.4.7: The Filter Selector block

Creates a *FilterParam* structure from a filter specification. The resulted structure can be fed to a *FilterDesigner* block to design the specified filter (see Chapter 4.4.3., p. 27.). The *FilterParam* structure is described in Chapter 3.1. (p. 5.).

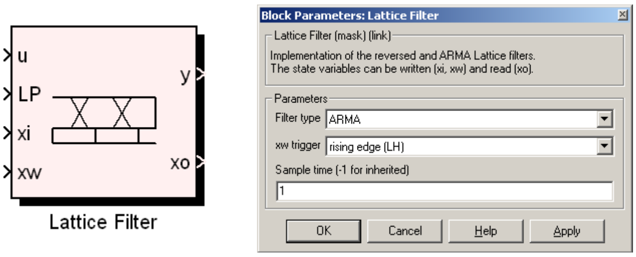
Interface:

⇩ Connector	⇩ Label	Type	Description
Output	*FP*	FilterParam	Filter specification

Table 4.4.5: The interface of the FilterSelector block

Examples: Fig. 4.4.3 (p. 25.), Fig. 4.4.4 (p. 25.).

4.4.5. Lattice Filter



Lattice Filter (mask) (link)

Implementation of the reversed and ARMA Lattice filters.
The state variables can be written (xi, xw) and read (xo).

Parameters

Filter type: ARMA

xw trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.4.8: The Lattice Filter block

The block implements Lattice filter structures.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	double	Input signal
	LP	LatticeParam	Parameters of the filter
	xi	Vector	State vector input
	xw	double	State vector write trigger
Output	*y*	double	Signal output
	xo	Vector	State vector output

Table 4.4.6: The interface of the Lattice Filter block

Parameters:

⇩ Parameter	Values	Description – Filter structure
Filter type	Basic	Fig. 4.4.11 – a
	Reversed Basic	Fig. 4.4.11 – b
	Normalized	Fig. 4.4.12 – a
	Reversed Normalized	Fig. 4.4.12 – b

Table 4.4.7: The parameters of the Lattice Filter block

The state vector can be set by the *xi* and the *xw* inputs and can be read from the *xo* output. If the state vector input is wider than the actual (internal) state vector then the elements with higher indices are discarded. Smaller input vectors are padded with zeros.

Filter architecture:

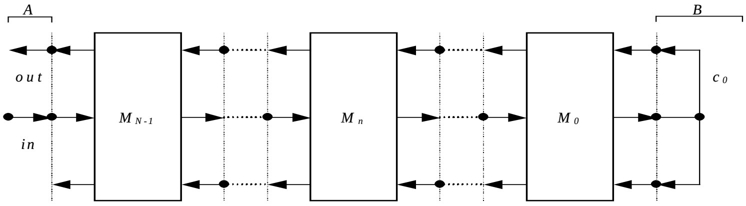


Fig. 4.4.9: The block architecture of non-reversed Lattice filters

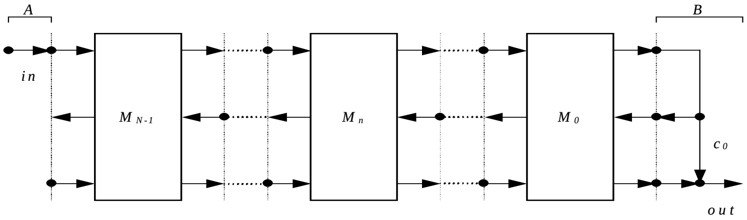
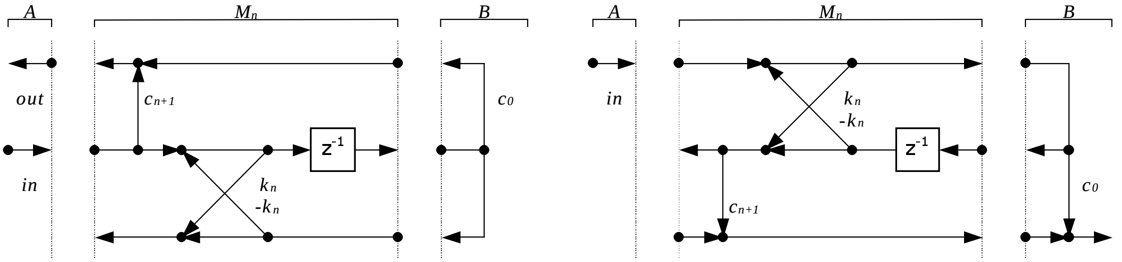


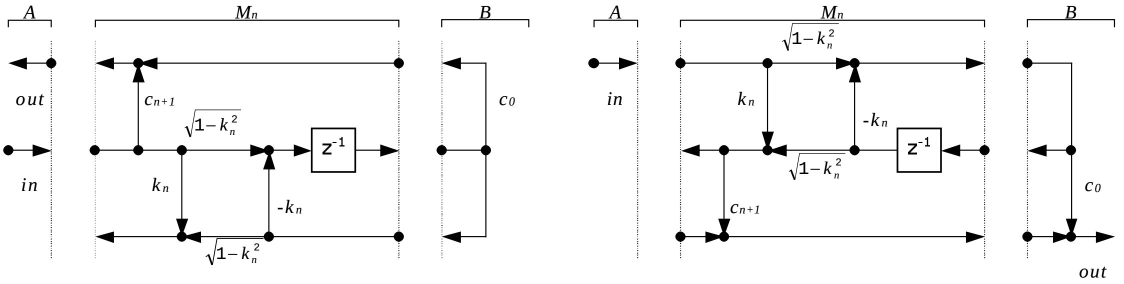
Fig. 4.4.10: The block architecture of reversed Lattice filters

Filter modules:



(a) (b)

Fig. 4.4.11: The basic (*a*) and the reversed (basic) (*b*) Lattice structures



a b

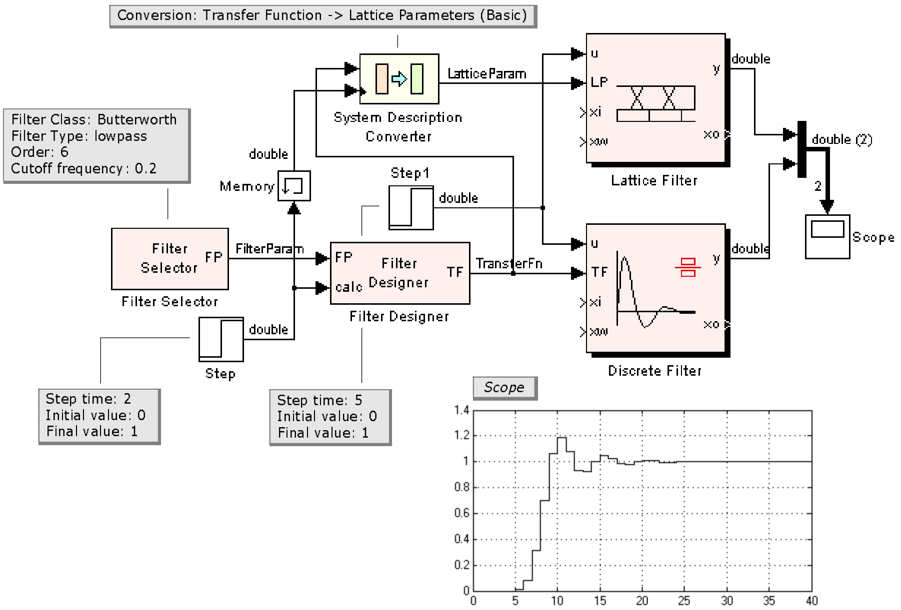
Fig. 4.4.12: The normalized (*a*) and the reversed normalized (*b*) Lattice structures

Lattice filters have a modular architecture. For non-reversed structures see Fig. 4.4.9 and for reversed structures see Fig. 4.4.10. The corresponding module structures can be seen in Fig. 4.4.11 and Fig. 4.4.12. See also Chapter 4.3.8, p. 21. for the *System Description Converter* block, and [3] for brief description of the Lattice filters and their design.

Remarks:

1. If the *LP* input is unconnected or empty the output will be zero.
2. The state vector stays unchanged when the *xi* input is unconnected.

Example:


Conversion: Transfer Function -> Lattice Parameters (Basic)

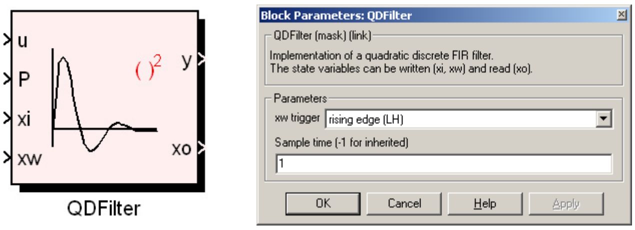
Filter Class: Butterworth
Filter Type: lowpass
Order: 6
Cutoff frequency: 0.2

Step time: 2
Initial value: 0
Final value: 1

Step time: 5
Initial value: 0
Final value: 1

Fig. 4.4.13: Example for the Lattice Filter (`.\Examples\Simple\ex_lattice.mdl`)

4.4.6. *Quadratic Discrete Filter (QDFilter)*



Block Parameters: QDFilter

QDFilter (mask) (link)

Implementation of a quadratic discrete FIR filter.
The state variables can be written (xi, xw) and read (xo).

Parameters

xw trigger: rising edge (LH)

Sample time (-1 for inherited): 1

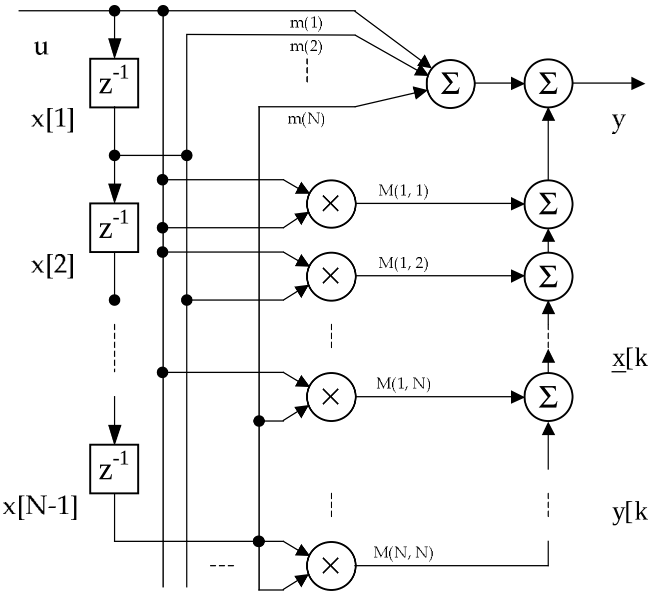
Fig. 4.4.14: The Quadratic Discrete Filter block

Implements a quadratic discrete FIR filter structure shown in Fig. 4.4.15. The *QDFilterParam* structure contains the weight matrices for the filter (see Fig. 4.4.15).

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	double	Input signal
	P	QDFilterParam	Weight matrices of the filter
	xi	Vector	State vector input
	xw	double	State vector write trigger
Output	*y*	double	Signal output
	xo	Vector	State vector output

Table 4.4.8: The interface of the Quadratic Discrete Filter block



$$\underline{x}[k+1]=\begin{bmatrix}0 & 0 & \cdots & 0 & 0\\ 1 & 0 & \cdots & 0 & 0\\ 0 & 1 & \cdots & 0 & 0\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ 0 & 0 & \cdots & 1 & 0\end{bmatrix}\cdot\underline{x}[k]+\begin{bmatrix}1\\ 0\\ \vdots\\ 0\\ 0\end{bmatrix}\cdot u[k]$$

$$y[k]=\underline{m}\cdot\begin{bmatrix}u[k]\\ \underline{x}[k]\end{bmatrix}+\underline{\underline{M}}\cdot\left(\begin{bmatrix}u[k]\\ \underline{x}[k]\end{bmatrix}\cdot\begin{bmatrix}u[k]\\ \underline{x}[k]\end{bmatrix}^T\right)$$

Fig. 4.4.15: The structure and the formulas of the Quadratic Discrete Filter block

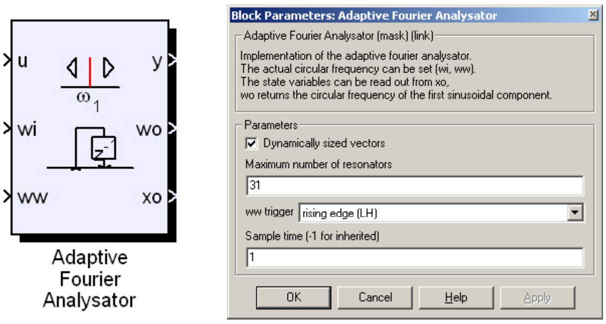
The state vector can be set by the *xi* and the *xw* inputs and can be read from the *xo* output. If the state vector input is wider than the actual (internal) state vector then the elements with higher indices are discarded. Smaller input vectors are padded with zeros.

Remarks:

1. If the *P* input is unconnected or empty the output will be zero.
2. The state vector stays unchanged when the *xi* input is unconnected.

4.5. Resonators

4.5.1. Adaptive Fourier Analysator



Block Parameters: Adaptive Fourier Analysator

Adaptive Fourier Analysator (mask) (link)

Implementation of the adaptive fourier analysator.
The actual circular frequency can be set (wi, ww).
The state variables can be read out from xo,
wo returns the circular frequency of the first sinusoidal component.

Parameters

☑ Dynamically sized vectors

Maximum number of resonators: 31

ww trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.5.1: The Adaptive Fourier Analysator block

The Adaptive Fourier Analysator decomposes the input signal to its sinusoidal components (harmonics) while tracking and finding the circular frequency of the first harmonic.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	double (C)	Signal input
	wi	double	Circ. frequency of the first harmonic
	ww	double	Write trigger for *wi*
Output	*Y*	double (C)	Signal output
	wo	double	The circular frequency of the first harmonic (ω)
	xo	Vector	State vector output

Table 4.5.1: The interface of the Adaptive Fourier Analysator block

Parameters:

⇩ Parameter	Values	Description
Dynamically sized vectors	☐	The width of the spectrum is determined by the *Maximum number of resonators* parameter
	☑	The width of the spectrum can change during the operation
Maximum number of resonators	[1..] (uneven)	The maximal width of the spectrum (and the state vector)

Table 4.5.2: The parameters of the Filter Designer

The algorithm:

Notation:

N	The number of resonators
N_M	The maximal number of resonators
$\underline{x}$	Component (state) vector
$\underline{c}$	Resonator coefficients
ang(.)	Function returning the angle of a complex number
k	Actual step (discrete time)
ω	The circular frequency of the first harmonic

Computation of the output:

$$y[k] = \underline{x}[k]^T \cdot \underline{c}[k]$$

Computation of the state transition:

$$L[k] = \left\lceil \frac{\pi}{\omega[k-1]} \right\rceil - 1 \qquad N[k] = 2L[k] + 1 \qquad r[k] = \frac{1}{N[k]}$$

$$x_{N..N_M}[k] = 0 \qquad c_{N..N_M}[k] = 1$$

$$\underline{x}[k+1] = \underline{x}[k] + (u[k] - y[k]) \cdot r[k] \cdot \underline{\bar{c}}[k]$$

$$\omega[k] = \omega[k-1] + r[k] \cdot [\mathrm{ang}(x_2[k+1]) - \mathrm{ang}(x_2[k])]$$

$$c_{2i}[k+1] = c_{2i}[k] \cdot e^{ij\omega} \qquad c_{2i+1}[k+1] = c_{2i+1}[k] \cdot e^{-ij\omega}$$

Remark:

1. The initial circular frequency for the first harmonic is set to $\omega = 0.52\pi$ (unless one is specified by *wi* and *ww*).

Example:

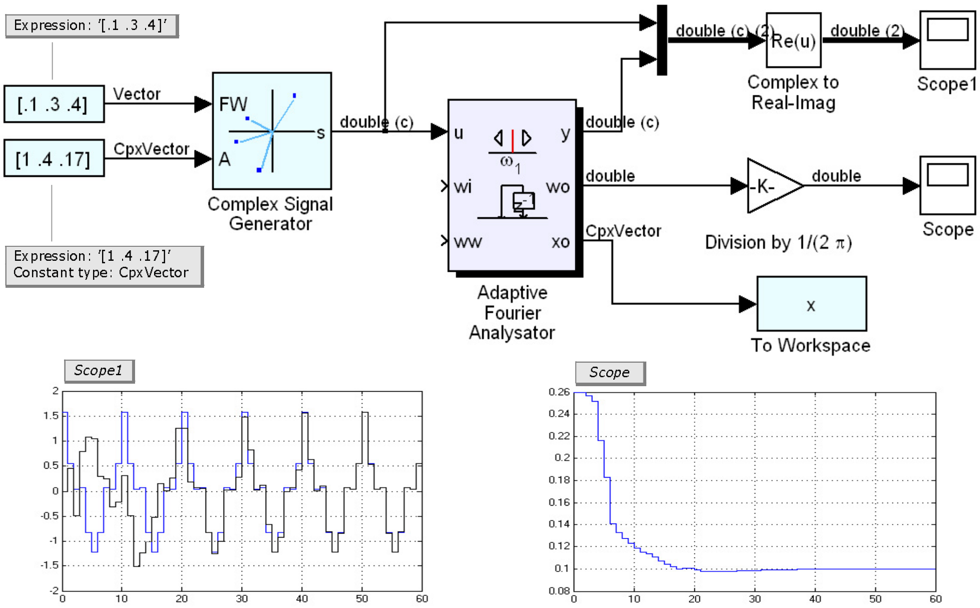
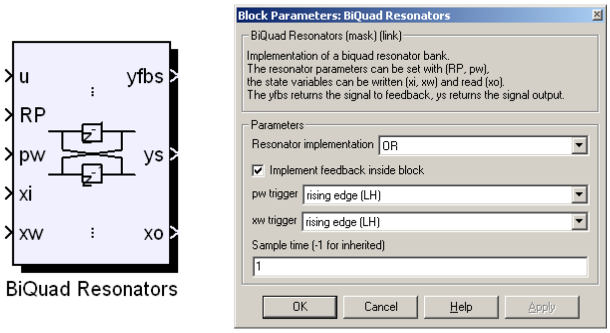


Fig. 4.5.2: Example for the Adaptive Fourier Analysator (`.\Examples\Simple\ex_afa.mdl`)

4.5.2. BiQuad Resonators



BiQuad Resonators (mask) (link)

Implementation of a biquad resonator bank.
The resonator parameters can be set with (RP, pw),
the state variables can be written (xi, xw) and read (xo).
The yfbs returns the signal to feedback, ys returns the signal output.

Parameters

Resonator implementation: OR

☑ Implement feedback inside block

pw trigger: rising edge (LH)

xw trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.5.3: The BiQuad Resonators block

The *BiQuad Resonators* block implements a resonator bank of biquad structures. The two supported structures are the orthogonal and wave-digital. The resonator coefficients are given by the incoming *ResParam* structure (see Fig. 4.5.5, Fig. 4.5.6 and Chapter 4.5.6., p. 41.: *tf2resparam* block).

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	double	Signal input
	RP	ResParam	Resonator parameters
	pw	double	Parameter input trigger
	xi	Vector	State vector input
	xw	double	State vector input trigger
Output	*yfbs*	double	Feedback line
	ys	double	Signal output
	xo	Vector	State vector output

Table 4.5.3: The interface of the BiQuad Resonators block

Parameters:

⇩ Parameter	Values	Description
Dynamically sized vectors	WD	Wave-digital structure (see Fig. 4.5.5)
	OR	Orthogonal structure (see Fig. 4.5.6)
Maximum number of resonators	☐	The *yfbs* has to be fed back to the input
	☑	The feedback of *yfbs* is implemented inside

Table 4.5.4: The parameters of the BiQuad Resonators block

Operation and structure diagrams:

The block diagram of the filter is shown in Fig. 4.5.4, where $R_1..R_N$ denote the resonators. The implementation of the resonators depends on the *Resonator implementation* parameter that is the value *WD* selects the wave-digital (see Fig. 4.5.5) and the value *OR* selects the orthogonal (Fig. 4.5.6) structure. The feedback marked with the dashed line in Fig. 4.5.4 is implemented when the *Implement feedback inside block* parameter is set.

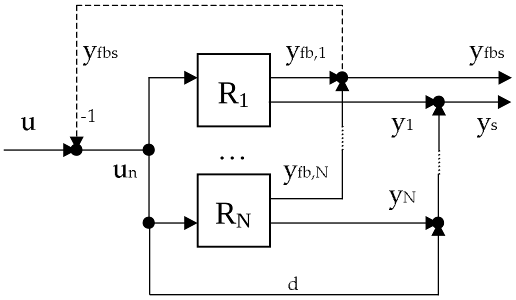
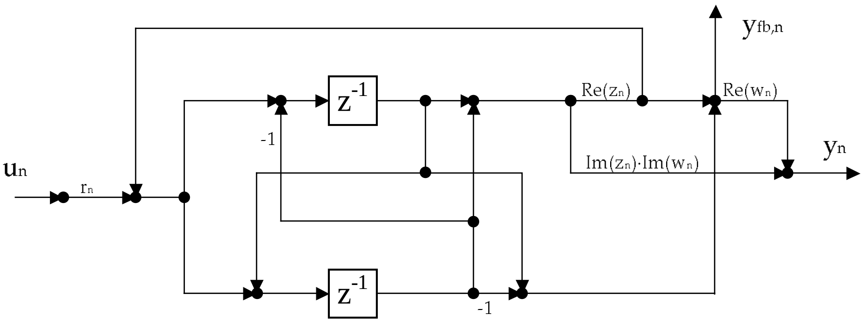


Fig. 4.5.4: The structure of the BiQuad filter

for complex z_n-s:



for real z_n-s:

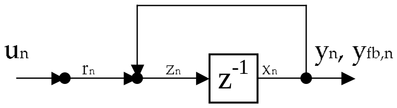
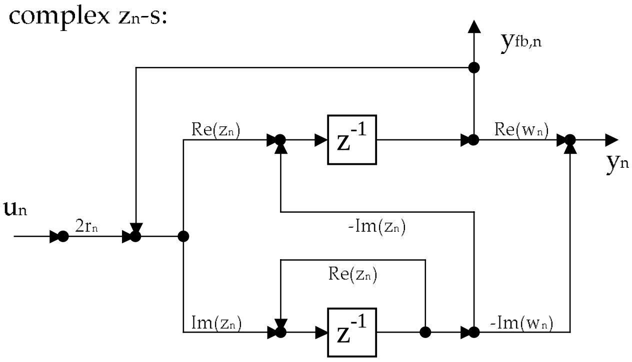


Fig. 4.5.5: The wave-digital resonator structure (one of the resonators in the BiQuad filter)

for complex z_n-s:



for real z_n-s:

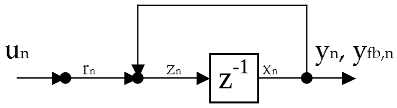


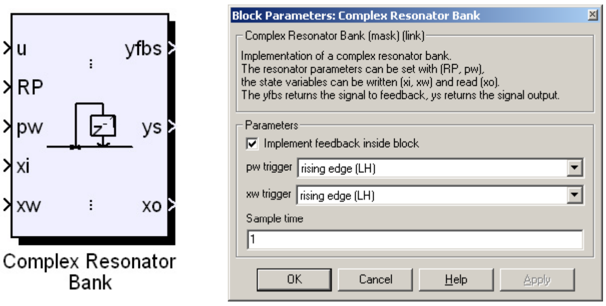
Fig. 4.5.6: The Orthogonal resonator structure (one of the resonators in the BiQuad filter)

Remarks:

1. If the input *RP* is unconnected or the value is empty or invalid then the output becomes zero and the state vector stays unchanged.
2. If the size of the structure on the *RP* input is not equal to that of the stored internal structure then new states are created with zero initial values or some resonators are deleted depending on whether the new size is greater or less than the old.

Example: Fig. 4.5.9, p. 37.

4.5.3. Complex Resonator Bank



Block Parameters: Complex Resonator Bank

Complex Resonator Bank (mask) (link)

Implementation of a complex resonator bank.
The resonator parameters can be set with (RP, pw),
the state variables can be written (xi, xw) and read (xo).
The yfbs returns the signal to feedback, ys returns the signal output.

Parameters

☑ Implement feedback inside block

pw trigger: rising edge (LH)

xw trigger: rising edge (LH)

Sample time: 1

Fig. 4.5.7: The Complex Resonator Bank block

This block implements a bank of complex resonators. The resonator coefficients are given by the incoming *ResParam* structure (see Fig. 4.5.8 and Chapter 4.5.6., p. 41.: *tf2resparam* block).

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	double (C)	Signal input
	RP	ResParam	Resonator parameters
	pw	double	Parameter input trigger
	xi	CpxVector	State vector input
	xw	double	State vector input trigger
Outputs	*yfbs*	double (C)	Feedback line
	ys	double (C)	Signal output
	xo	CpxVector	State vector output

Table 4.5.5: The interface of the Complex Resonator Bank

Parameters:

⇩ Parameter	Values	Description
Implement feedback inside block	☐	The *yfbs* has to be fed back to the input
	☑	The feedback of *yfbs* is implemented inside

Table 4.5.6: The parameters of the Complex Resonator Bank

Operation:

The structure of the implemented layout is shown in Fig. 4.5.8.

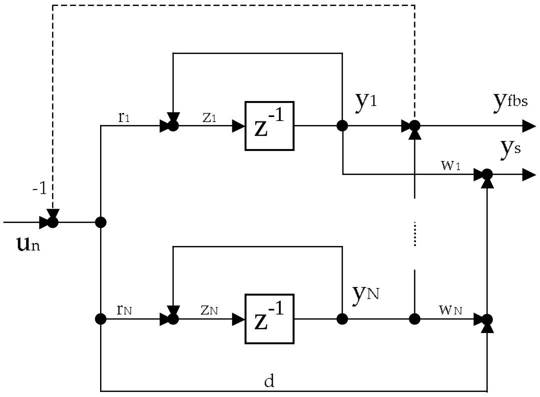


Fig. 4.5.8: The structure of the Complex Resonator Bank

The feedback marked with the dashed line in Fig. 4.5.8 is implemented when the *Implement feedback inside block* parameter is set.

Remarks:

1. If the input *RP* is unconnected or the value is empty or invalid then the output becomes zero and the state vector stays unchanged.
2. If the size of the structure on the *RP* input is not equal to that of the stored internal structure then new states are created with zero initial values or some resonators are deleted depending on whether the new size is greater or less than the old.

Example:

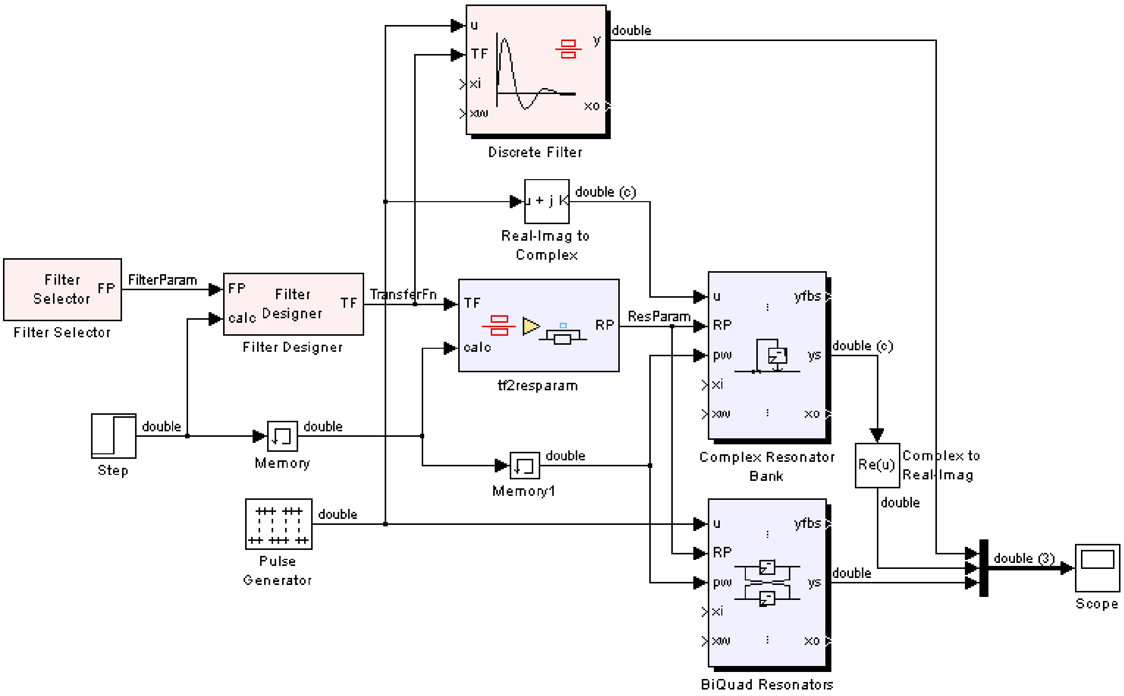
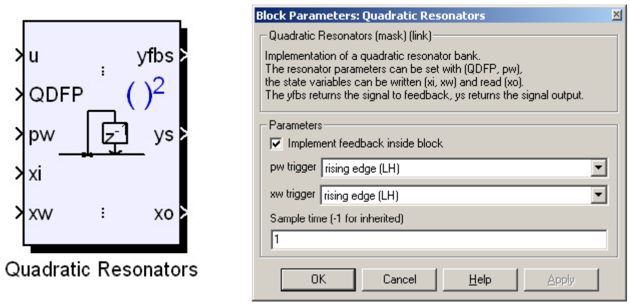


Fig. 4.5.9: Example for the Complex Resonator Bank and the BiQuad Resonators blocks
(.\Examples\Simple\ex_resbank.mdl)

4.5.4. Quadratic Resonator Bank



Quadratic Resonators (mask) (link)

Implementation of a quadratic resonator bank.
The resonator parameters can be set with (QDFP, pw),
the state variables can be written (xi, xw) and read (xo).
The yfbs returns the signal to feedback, ys returns the signal output.

Parameters

☑ Implement feedback inside block

pw trigger: rising edge (LH)

xw trigger: rising edge (LH)

Sample time (-1 for inherited): 1

Fig. 4.5.10: The Quadratic Resonators block

The block implements a resonator bank immediately connected to a weigthted tapping network. The architecture is shown in Fig. 4.5.11. The coefficients of the resonators are placed evenly on the unit circle. The tapping coefficients are given by the *QDFilterParam* structure on the *QDFP* input (see Chapter 4.4.6, p. 31. and Fig. 4.5.11).

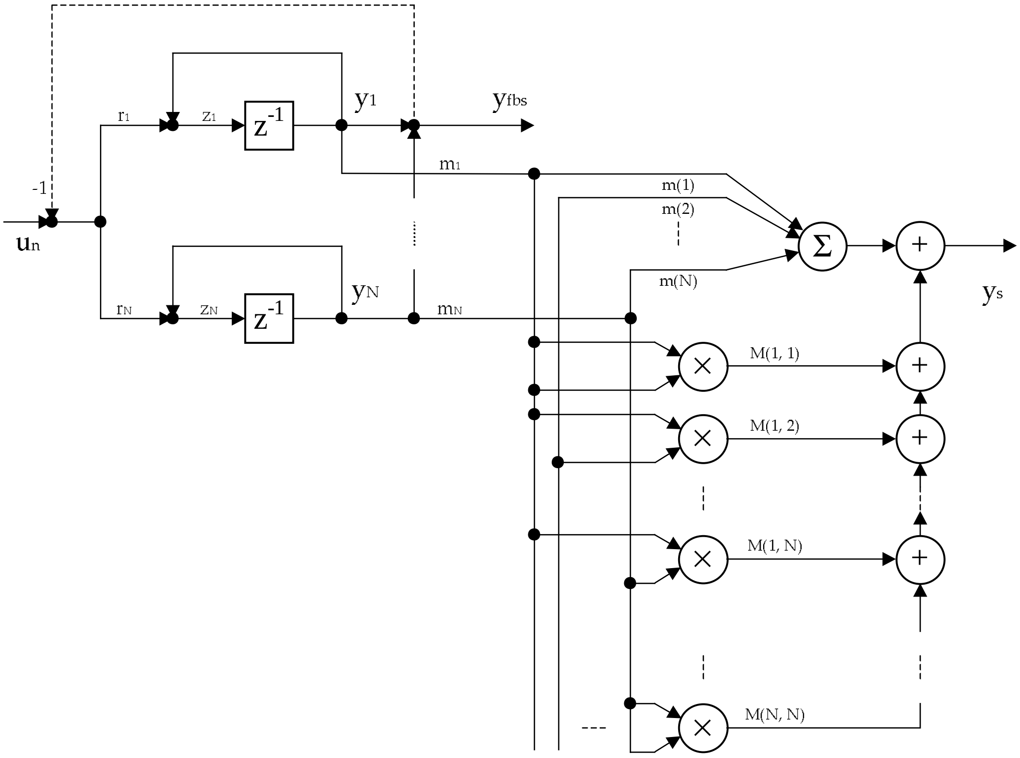


Fig. 4.5.11: The structure of the Quadratic Resonators block

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*U*	double (C)	Signal input
	QDFP	QDFilterParam	Quadratic Resonator parameters
	pw	double	Parameter input trigger
	xi	CpxVector	State vector input
	xw	double	State vector input trigger
Outputs	*yfbs*	double (C)	Feedback line
	ys	double (C)	Signal output
	xo	CpxVector	State vector output

Table 4.5.7: The interface of the Quadratic Resonators block

Parameters:

⇩ Parameter	Values	Description
Implement feedback inside block	☐	The *yfbs* has to be fed back to the input
	☑	The feedback of *yfbs* is implemented inside

Table 4.5.8: The parameters of the Quadratic Resonators block

The number of the resonators is equal to the length of the vector *m* in the *QDFilterParam* (linear tapping coefficients). The resonator positions can be expressed by:

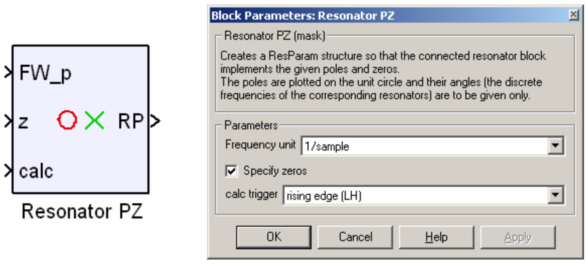
$$z_n = e^{j\frac{n-1}{N}} \qquad n:1..N$$

The feedback marked with the dashed line in Fig. 4.5.11 is implemented when the *Implement feedback inside block* parameter is set.

Remarks:

1. If the input *RP* is unconnected or the value is empty or invalid then the output becomes zero and the state vector stays unchanged.
2. If the size of the structure on the *RP* input is not equal to that of the stored internal structure then new states are created with zero initial values or some resonators are deleted depending on whether the new size is greater or less than the old.

4.5.5. Resonator PZ



Resonator PZ (mask)

Creates a ResParam structure so that the connected resonator block implements the given poles and zeros.
The poles are plotted on the unit circle and their angles (the discrete frequencies of the corresponding resonators) are to be given only.

Parameters

Frequency unit: 1/sample

☑ Specify zeros

calc trigger: rising edge (LH)

Fig. 4.5.12: The Resonator PZ block

The block computes a *ResParam* structure from the incoming pole (*FW_p*) (and zero (*z*)) specifications so that they will be implemented by the connected resonator block. The angle of the each pole is obtained from the elements of the *FW_p* vector input. The *Frequency unit* specifies if the angles are given proportionally to 1 or to 2π.

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*FW_p*	Vector	Angles of the pole set
	z	CpxVector	Zero set $^{(*)}$
	calc	double	Trigger of calculation
Output	*RP*	ResParam	Resonator parameters output

Table 4.5.9: The interface of the Resonator PZ block

Parameters:

⇩ Parameter	Values	*z* $^{(*)}$	Description
Frequency unit	1/sample		Angles proportional to the whole circle
	radians/sample		Angles proportional to one radian
Specify zeros	☐	no	No zeros specified (*z*=0 is used) $^{(*)}$
	☑	yes	Zeros specified by the *z* input $^{(*)}$

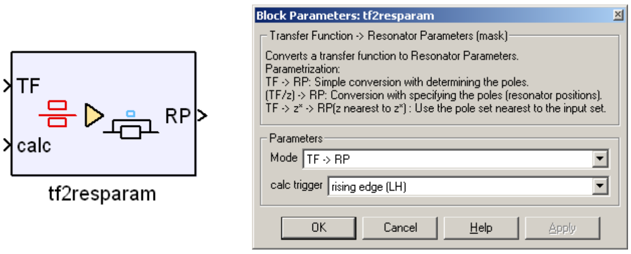
Table 4.5.10: The parameters of the Resonator PZ block

Output:

Parameter ⇨	⇩ *Frequency unit*	
⇩ *Resparam* field	1/sample	radians/sample
z	$z_m = e^{2\pi j \cdot FW(m)}$	$z_m = e^{j \cdot FW(m)}$
w	$w_m = 1$	
r	$r_m = \frac{\prod_{\substack{n=1 \\ n\neq m}}^{M}\left(1-z_m^{-1}z(n)\right)}{\prod_{\substack{n=1 \\ n\neq m}}^{N}\left(1-z_m^{-1}z_n\right)}$	
d	0	
$m = 1..N$		

Table 4.5.11: The output of the Resonator PZ block

4.5.6. *tf2resparam*



Transfer Function -> Resonator Parameters (mask)

Converts a transfer function to Resonator Parameters.
Parametrization:
TF -> RP: Simple conversion with determining the poles.
(TF/z) -> RP: Conversion with specifying the poles (resonator positions).
TF -> z* -> RP(z nearest to z*) : Use the pole set nearest to the input set.

Parameters
Mode: TF -> RP
calc trigger: rising edge (LH)

Fig. 4.5.13: The tf2resparam block

The *tf2biquadp* block converts the incoming transfer function to resonator parameters so that the resonators implement the given transfer function. The resonator coefficients are given by the incoming *ResParam* structure (see Chapter 4.5.2. and Chapter 4.5.3.).

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*TF*	TransferFn	Transfer function to convert
	z	CpxVector	Pole set (*)
	calc	double	Trigger of calculation
Output	*RP*	ResParam	Resonator parameters output

Table 4.5.12: The interface of the tf2resparam block

Parameters:

⇩ Parameter	Values	*z* (*)	Description
Mode	TF -> RP	no	See the „Algorithm" section
	(TF/z) -> RP	yes	
	TF -> z* -> RP(z nearest to z*)	yes	

Table 4.5.13: The parameters of the tf2resparam block

Algorithm:

num_0: The numerator of the transfer function by descending orders of z.
den_0: The denominatior of the transfer function by descending orders of z.
z_0: The pole set incoming on the input *z*.

---

1. $\mathrm{M} = \mathrm{length}(\mathrm{num}_0)$, $\mathrm{N} = \mathrm{length}(\mathrm{den}_0)$
2. $\mathrm{num} : \mathrm{num}[i] = \mathrm{num}_0[\mathrm{M} - i] \quad i = 1..\mathrm{M}$, $\mathrm{den} : \mathrm{den}[i] = \mathrm{den}_0[\mathrm{N} - i] \quad i = 1..\mathrm{N}$ (reverse vector)
3. $\mathrm{M} > \mathrm{N} : \mathrm{den} = \underbrace{\begin{bmatrix} 0 & \cdots & 0 & \mathrm{den}_1 & \mathrm{den}_2 & \cdots & \mathrm{den}_\mathrm{N} \end{bmatrix}}_{\mathrm{M}}$, $\mathrm{M} = \mathrm{N}$ (padding)
4. $\mathrm{d} = \mathrm{num}[\mathrm{k}_\mathrm{n}] / \mathrm{den}[\mathrm{k}_\mathrm{d}]$, where k_n and k_d denote the indices of the first rightmost nonzero elements of the two vectors.
5. $\mathrm{num} /= \mathrm{den}[\mathrm{k}_\mathrm{d}]$, $\mathrm{den} /= \mathrm{den}[\mathrm{k}_\mathrm{d}]$ (normalization)
6. $\mathrm{mden} : \mathrm{mden}[i] = \mathrm{den}_0[\mathrm{N} - i] \quad i = 1..\mathrm{N}$ (reverse vector)
7. $\mathrm{p} = \mathrm{roots}(\mathrm{mden})$ (the roots of mden)
8. If *Mode* == (TF/z) -> RP (and $\mathrm{length}(\mathrm{z}_0) == \mathrm{N} - 1$):
   a. $\mathrm{z} = \mathrm{sort}(\mathrm{z}_0)$
   b. Normalize z to unit length

```
    c. $r[n] = \frac{\prod_{i=1}^{N-1}\left(1 - p[i]\cdot\bar{z}[n]\right)}{\prod_{i=1, i\neq n}^{N-1}\left(1 - z[i]\cdot\bar{z}[n]\right)}$
9. else:
    a. $z_a = \mathrm{roots}(den - mden)$, $z_b = \mathrm{roots}(den + mden)$
    b. $\mathrm{sort}(z_a)$, $\mathrm{sort}(z_b)$ (Sort complex roots by the descending order of the real parts and then the real roots by descending order.)
    c. Normalize $z_a$ and $z_b$ to unit length.
    d. $r_a[n] = \frac{\prod_{i=1}^{N-1}\left(1 - p[i]\cdot\bar{z}_a[n]\right)}{\prod_{i=1, i\neq n}^{N-1}\left(1 - z_a[i]\cdot\bar{z}_a[n]\right)}$, $r_b[n] = \frac{\prod_{i=1}^{N-1}\left(1 - p[i]\cdot\bar{z}_b[n]\right)}{\prod_{i=1, i\neq n}^{N-1}\left(1 - z_a[i]\cdot\bar{z}_b[n]\right)}$
    e. If Mode==TF -> z* -> RP(z nearest to z*) then:
        i. $dist_a = \sum_{i=1}^{N-1}\left[\mathrm{ang}(z_a[i]) - \mathrm{ang}(p[i])\right]^2$, $dist_b = \sum_{i=1}^{N-1}\left[\mathrm{ang}(z_b[i]) - \mathrm{ang}(p[i])\right]^2$
        ii. If $dist_a < dist_b$ then $r = r_a$, $z = z_a$ else $r = r_b$, $z = z_b$
    f. else:
        i. If $\sum_{i=1}^{N} r_a[i] > 1$ then $r = r_b$, $z = z_b$ else $r = r_a$, $z = z_a$
10. $w[n] = \frac{\sum_{i=1}^{N} num[n]\cdot z[i]^{(i-1-M)}}{\sum_{i=1}^{N} den[n]\cdot z[i]^{(i-1-N)}}$
```

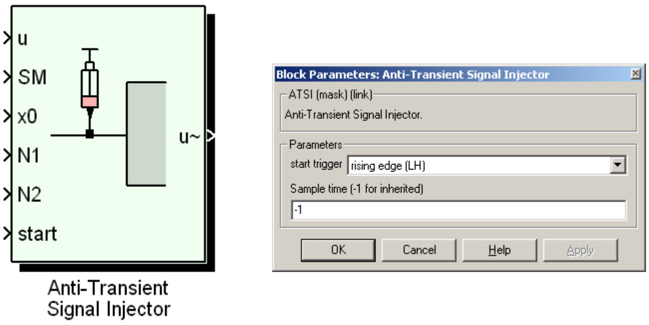
Remarks:

1. If the input structure (*TF*) is empty or unconnected then the output structure will be empty.
2. If the *z* input is empty or unconnected the block implements the *TF -> RP* mode.

Example: Fig. 4.5.9, p. 37.

ATSI (Anti-Transient Signal Injector)

4.5.7. ATSI (Anti-Transient Signal Injector)



Block Parameters: Anti-Transient Signal Injector

ATSI (mask) (link)

Anti-Transient Signal Injector.

Parameters

start trigger: rising edge (LH)

Sample time (-1 for inherited): -1

Fig. 0.1: Anti-Transient Signal Injector block

The block implements the anti-transient signal injection scheme described in [1], p. 32-35. (The paper can be downloaded in the download section of the **TransMan** homepage.)

Interface:

⇩ Connector	⇩ Label	Type	Description
Inputs	*u*	Vector	Signal input
	SM	StateMtcs (2)	State matrices of the system before and after the reconfiguration
	x0	Vector	State vector input
	N1	double	Number of samples to inject before the reconfiguration
	N2	double	Number of samples to inject after the reconfiguration
	start	double	Start trigger
Output	*u~*	Vector	Anti-transient signal

Table 0.1: The interface of the Anti-Transient Signal Injector block

Example: in Fig. 0.2 (p. 44.).

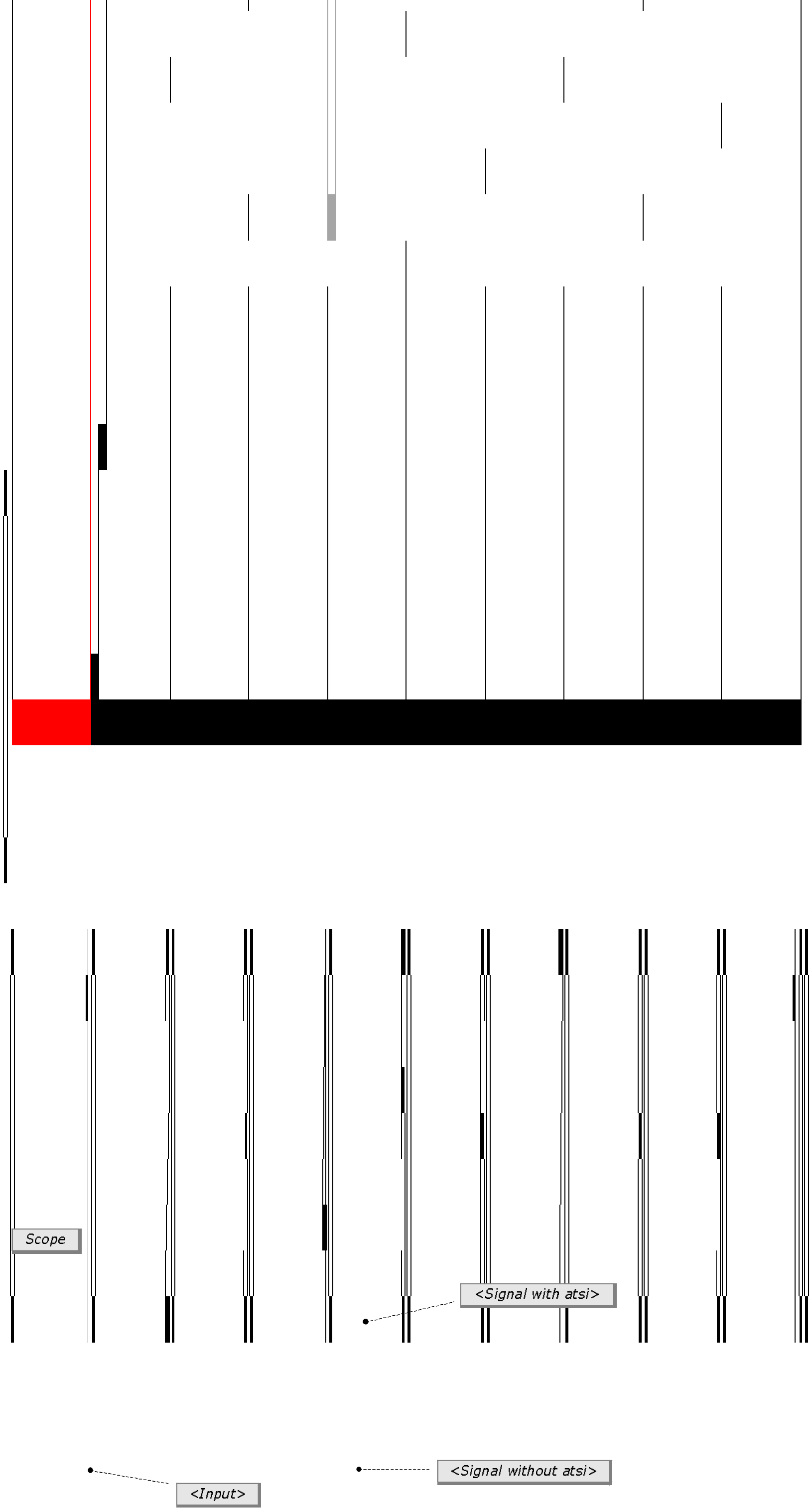


Fig. 0.2: Example for the Anti-Transient Signal Injector block (`.\Examples\Simple\ex_ATSI.mdl`)

5. Examples

6. References

[1] Gyula Simon, Tamás Kovácsházy, Gábor Péceli, "*Transient Management in Reconfigurable Control Systems*", Technical Report, Budapest, Hungary, 2002.

[2] Tamás Kovácsházy, Gábor Samu, Gábor Péceli: "*Simulink Block Library for Fast Prototyping of Reconfigurable DSP Systems*", IEEE International Symposium on Intelligent Signal Processing, Budapest, Hungary, 4-6 September, 2003., paper ID: W-79.

[3] Jin-Gyun Chung, Keshab K. Parhi: "*Pipelined Lattice and wave digital recursive filters*", Book. Kluwer Academic Publishers, Boston, Hardbound, ISBN 0-7923-9656-1, November 1995.

7. Table of figures