The LinearSystems library for continuous and discrete control systems

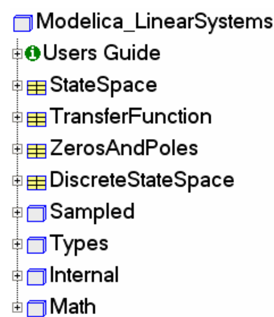
Martin Otter
German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Germany

Abstract

The free Modelica LinearSystems library provides basic data structures for linear control systems as well as operations on them, and includes blocks that allow quick switching between a continuous and a discrete representation of a multi-rate controller. The advantage is that fast simulations of the overall system can be performed when the modeling detail of the continuous controller is sufficient. It is planned to include this library in the Modelica standard library.

1 Introduction

Library *LinearSystems* is a free Modelica package providing different representations of linear,



time invariant differential and difference equation systems, as well as typical operations on these system descriptions. In the right figure a screen-shot of the first hierarchical level of the library is shown. *StateSpace*, *TransferFunction*, *ZerosAndPoles*, and *DiscreteStateSpace* are records that provide basic data structures for linear control systems. Every record contains a set of utility functions that operate on the corresponding data structure. Especially, operations are provided to transform the respective data structures in to each other.

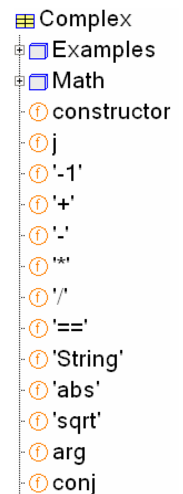
Sublibrary *Math* contains the basic data structures *Complex* and *Polynomial* that are utilized from the linear system descriptions.

Sublibrary *Sampled* contains a library of input/output blocks to conveniently model and simulate sampled data systems where it is convenient to quickly switch between a continuous and a discrete block representation.

2 Sublibrary Math

This sublibrary provides the basic data structures *Complex* and *Polynomial* that are utilized and needed from the linear systems data structures to be discussed in the next section.

In the Modelica Association there is currently a proposal to introduce operator overloading into Modelica 3.0. The data structures in the LinearSystems library are organized so that their usage becomes convenient once operator overloading is available.



The basic approach will be explained at hand of the record Math.Complex. Its content is displayed on the right side. The record contains the basic definition of a complex number consisting of a real (re) and an imaginary (im) part:

```
record Complex
  Real re;
  Real im;
  // function definitions
end Complex;
```

Additionally, a set of functions is present that operates on this record. Especially, function constructor(..) constructs an instance of this record and all functions 'xx' provide basic operations such as addition and multiplication. Note, the apostrophe is part of the function name. In the operator overloading proposal, predefined function names are defined, such as '+' that will be automatically invoked when a Complex number is used in an arithmetic expression.

The Complex number record can be utilized in *current Modelica environments*, such as Dymola (Dynasim 2006), in the following way:

```
import Modelica.Utilities.Streams;
import LinearSystems.Math.Complex;
Complex c1=Complex(re=2, im=3) "= 2 + 3j";
Complex c2=Complex(3,4)        "= 3 + 4j";
```

```
algorithm
  c3 := Complex.'+'(c1, c2) "= c1 + c2";
  Streams.print("c3 = " +
                Complex.'String'(c3));
  Streams.print("c3 = " +
               Complex.'String'(c3,"i"));
```

In the declaration, the two record instances c1 and c2 form c3. The value of record c3 is printed with the `Complex.'String'` functions in different forms resulting in the following print-out:

```
c3 = 5 + 7j
c3 = 5 + 7i
```

Once operator overloading is available, the above statements can be simply written as:

```
c3 := c1 + c2;
Streams.print("c3 = " + String(c3));
Streams.print("c3 = " + String(c3,"i"));
```

It is then also possible to write:

```
Complex j = Complex.j();
Complex c4 = -2 + 5*j;
Complex c5 = c1*c2 / ( c3 + c4);
```

In record Math.Complex the basic operations for complex numbers are provided.

In a similar way all other data structures of the LinearSystems library are defined. They can all be at once used in current Modelica environments and they will be conveniently usable once operator overloading is available in Modelica and in Modelica tools.

Record Math.*Polynomial* defines a data structure for polynomials with real-valued coefficients and provides the most important operations on polynomials. Its content is displayed on the right side. A Polynomial is constructed by the command

```
Polynomial(coeff.Vector)
```

- Polynomial
  - Examples
  - constructor
  - fromReal
  - fromComplex
  - fromZeros
  - base
  - fitting
  - degree
  - '-1'
  - '+'
  - '-'
  - '*'
  - '/'
  - '^'
  - '=='
  - 'String'
  - plot
  - derivative
  - integral
  - evaluate
  - evaluateComplex
  - derivativeValue
  - integralValue
  - roots

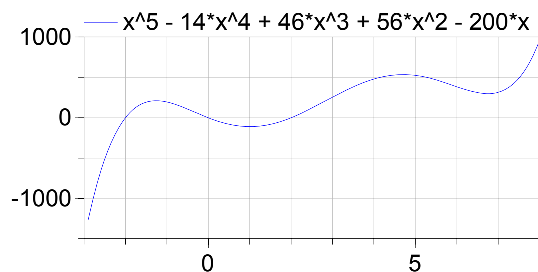
where the input argument provides the polynomial coefficients in descending order. For example, the polynomial $y = 2*x^2 + 3*x + 1$ is defined as

```
Polynomial({2,3,1})
```

Besides arithmetic operations, functions are provided to compute the zeros of a polynomial (via the eigen values of the companion matrix), to differentiate, to integrate and to evaluate the function value, its derivative and its integral. Find below a typical example from the scripting environment of Dymola:

```
import LinearSystems.Math.Polynomial;
p = Polynomial({6, 4, -3})
Polynomial.'String'(p)
  // = "-6*x^2 + 4*x - 3"

int_p = Polynomial.integral(p)
Polynomial.'String'(int_p)
  // = "-2*x^3 + 2*x^2 - 3*x"

der_p = Polynomial.derivative(p)
Polynomial.'String'(der_p)
  // = "-12*x + 4"

Polynomial.evaluate(der_p,1)
  // = -8

r = Polynomial.roots(p, printRoots=true)
  // = 0.333333 + 0.62361j
  // = 0.333333 - 0.62361j
```

With function *fitting*, a polynomial can be determined that approximates given table values. Finally with function *plot*, the interesting range of x is automatically determined (via calculating the roots of the polynomial and of its derivative) and plotted. The plotting is currently performed with the plot function in Dymola. Once a Modelica built-in plot function is available, this will be adapted. A typical plot is shown in the next figure:



3 Linear System Descriptions

At the top level of the LinearSystems library, data structures are provided as Modelica records defining different representations of linear, time invariant differential and difference equation systems. In the record definitions, functions are provided that operate on the corresponding data structure. Currently, the following linear system representations are available:

3.1 Record StateSpace

This record defines a multi-input, multi-output linear time-invariant differential equation system in state space form:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t)$$
$$\mathbf{y}(t) = \mathbf{C} \cdot \mathbf{x}(t) + \mathbf{D} \cdot \mathbf{u}(t)$$

The data part of the record contains the constant matrices A,B,C,D in the following definition:

```
record StateSpace
  Real A[:,:];
  Real B[size(A,1),:];
  Real C[:,size(A,1)];
  Real D[size(C,1), size(B,2)];
  // function definitions
end StateSpace;
```

StateSpace
- Examples
- constructor
- fromModel
- fromFile
- fromTransferFunction
- fromReal
- '-1'
- '+'
- '-'
- '*'
- '=='
- plotEigenValues
- invariantZeros

The content of the StateSpace record definition is displayed on the right side. The *fromXXX* functions transform from another representation to a StateSpace record. Especially fromModel(..) linearizes a Modelcia model and provides the linear system as output argument. fromFile(..) reads a StateSpace description from file and fromTransferFunction(..) transform a transfer function description into a StateSpace form.

The basic arithmetic operations are interpreted as series and parallel connection of StateSpace systems. Function invariantZeros(..) computes the invariant zeros. For single-input, single-output systems these are the zeros of the transfer function. Function plotEigenValues(..) computes and plots the eigenvalues of the system.

3.2 Record TansferFunction

TransferFunction
- Examples
- constructor
- fromReal
- fromPolynomials
- fromZerosAndPoles
- LaplaceVariable
- '-1'
- '+'
- '-'
- '*'
- '/'
- '^'
- '=='
- 'String'
- numeratorDegree
- denominatorDegree
- evaluate
- zerosAndPoles
- plotBode

This record defines the transfer function between the input signal u and the output signal y by the coefficients of the numerator and denominator polynomials n(s) and d(s) respectively:

$$y = \frac{n(s)}{d(s)} \cdot u$$

The order of the numerator polynomial can be larger as the order of the denominator polynomial (in such a case, the transfer function can not be transformed to a StateSpace system, but other operations are possible). For example, the transfer function

$$y = \frac{2 \cdot s + 3}{4 \cdot s^2 + 5 \cdot s + 6} \cdot u$$

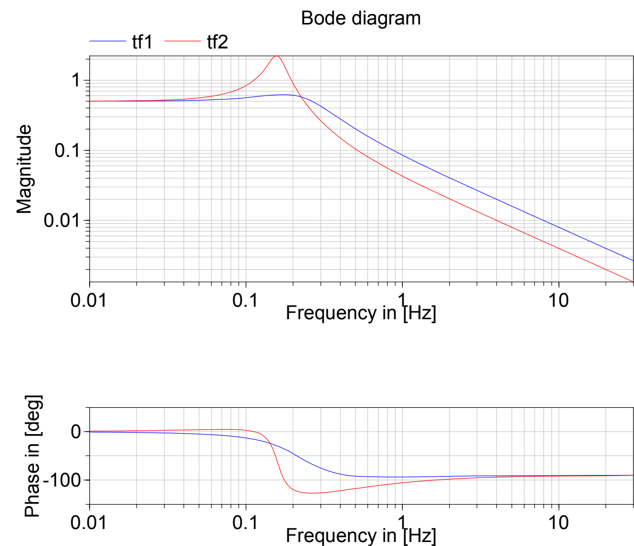
is defined in the following way:

```
import TF=LinearSystems.TransferFunction;
import Modelica.Utilities.Streams;
TF tf(n={2,3}, d={4,5,6});
print("y = " + TF.'String'(tf) + "* u")
```

The last statement prints the following string to the output window:

```
y = (2*s + 3) / (4*s^2 + 5*s + 6) * u
```

Besides arithmetic operations on transfer functions and constructing them optionally also from polynomials and from zeros and poles, the zeros and poles can be computed and a Bode plot can be constructed, as shown in the next figure:



The function call `plotBode(tf1, legend="tf1")` automatically selects an appropriate frequency intervall and uses the selected legend. The useful frequency range is estimated such that the phase angle of the plot of *one* (numerator or denominator) zero is in the range:

$$\frac{\varphi_{min}}{n} \leq |\text{phase angle}| \leq \frac{\pi}{2} - \frac{\varphi_{min}}{n}$$

where n is the number of (numerator or denominator) zeros. Note, the phase angle of one zero for a frequency of 0 up to infinity is in the range:

$$0 \leq |\text{phase angle}| \leq \frac{\pi}{2}$$

Therefore, the frequency range is estimated such that the essential part of the phase angle (defined by φ_{min}) is present in the Bode plot (default is 10^{-4} rad).

3.3 Record ZerosAndPoles

This record defines the transfer function between the input signal u and the output signal by its zeros, poles and a gain:

$$y = k \cdot \frac{\prod (s - z_i)}{\prod (s - p_j)} \cdot u$$

where the zeros and poles are defined by two Complex vectors of coefficients z_i and p_j. The elements of the two Complex vectors must either be real numbers or conjugate complex pairs (in order that their product results in a polynomial with Real coefficients).

ZerosAndPoles
- Examples
- constructor
- fromReal
- fromTransferFunction
- fromFactorization
- filter
- LaplaceVariable
- '*'
- 'String'
- numeratorDegree
- denominatorDegree
- evaluate
- zerosAndPoles
- plotBode

A description with zeros and poles is problematic: For example, a small change in the imaginary part of a conjugate complex pole pair, leads no longer to a transfer function with real coefficients. If the same zero or pole is present twice or more, then a diagonal state space form is no longer possible. This means that the structure is very sensitive if zeros or poles are close together. Performing arithmetic operations on such a description therefore leads easily to transfer functions with non-real coefficients. For this and other reasons, the constructor transforms this data structure and stores it internally in the record as first and second order polynomials with real coefficients:

$$y = k \cdot \frac{\prod (s + n_{1i}) \cdot \prod (s^2 + n_{2j} \cdot s + n_{3j})}{\prod (s + d_{1k}) \cdot \prod (s^2 + d_{2l} \cdot s + d_{3l})} \cdot u$$

All functions operate on this data structure. It is therefore guaranteed that an operation results again in a transfer function with real coefficients. It is possible to construct a ZerosAndPoles record directly with these coefficients by using function fromFactorized(..).

This data structure is especially useful in applications where first and second order polynomials are naturally occurring, e.g., as for filters: With function filter(..) this factorized form is directly generated from the desired low and high pass filters of type CriticalDamping, Bessel, Butterworth or Chebyshev. The filter options can be seen in Figure 1, a screenshot of the input menu of this function.

Besides type of filter and whether it is a low or high pass filter, the order of the filter and the cut-off frequency can be defined. With option "normalized, filters can be defined in normalized (default) and non-normalized form.

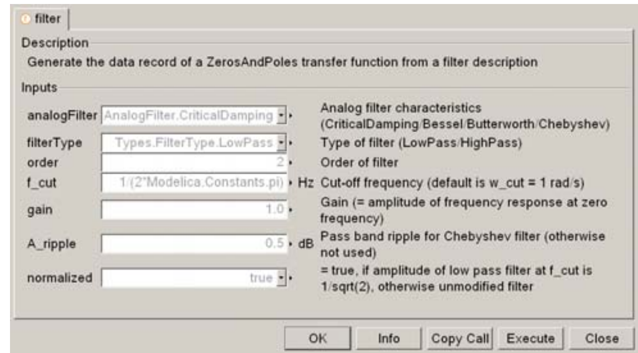


Figure 1: Input menu of function filter(..).

In the normalized form, the amplitude of the filter transfer function at the cutoff frequency is 1/sqrt(2) (= 3 dB). When trying out different filter types with different orders, the result of a comparison makes only sense if the filter is normalized.

Note, when comparing the filters of this function with other software systems, the setting of "normalized" has to be selected appropriately. For example, the signal processing toolbox of Matlab provides the filters in non-normalized form and therefore normalized = false has to be set.

In Figure 2, the magnitudes of the 4 supported low pass filters are shown in normalized form and in Figure 5 the corresponding step responses are given (generated with the LinearSystems.Sampled library).

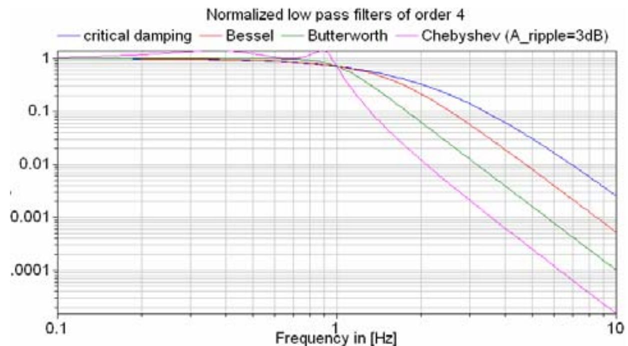


Figure 2: Magnitudes of normalized low pass filters

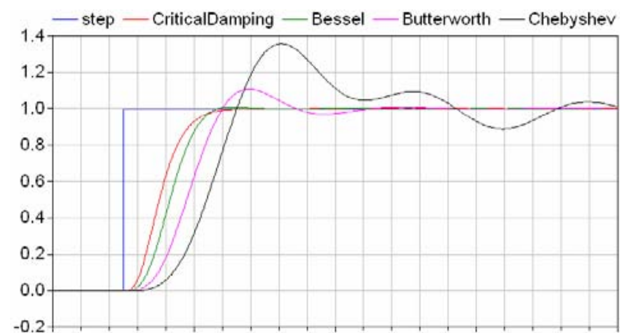


Figure 3: Step responses of norm. low pass filters

Obviously, the frequency responses give a somewhat wrong impression of the filter characteristics: Although Butterworth and Chebyshev filters have a significantly steeper magnitude as the CriticalDamp-

ing and Bessel filters, the step responses of the latter ones are much better since the settling times are shorter and no overshoot occurs. This means for example, that a CriticalDamping or a Bessel filter should be selected, if a filter is mainly used to make a non-linear inverse model realizable.

3.4 Record DiscreteStateSpace

This record defines a linear time invariant *difference equation system* in state space form. At the time of writing, this record contains only the core function fromStateSpace(..) to transform a StateSpace description in a DiscreteStateSpace form. The details of this record and of this function are discussed in section 4.3.

4 Sublibrary Sampled

4.1 Overview

The core of the LinearSystems library is sublibrary *Sampled* to model continuous and discrete multi-rate control systems. Experience shows that the combination of physical plant models that are controlled by digital controllers slow down the simulation speed significantly, if the sampling rates of the digital controllers are small compared to the step-size that could be used for the continuous plant. For example, at DLR detailed, calibrated models of robot systems are available. A stiff solver with variable step size integrates this system with step sizes in the order of 10 – 20 ms. When replacing the continuous approximation of the controllers by the actual digital controller implementation, the simulation time is **increased** by a factor of **20-30**. The reason is that the digital controllers have a sample time in the order of 1 ms and therefore the step size of the integrator is limited by 1 ms. Additionally at every sample point the integrator has to be restarted to reliably handle the discontinuous change of the actuator signal which reduces again the simulation efficiency.

For some design phases a continuous approximation of a controller might be sufficient, e.g., when tuning the controller parameters by parameter variation or multi-criteria optimization. By reducing the simulation time with a factor of, say 20-30, will then significantly reduce the optimization time.

It is also necessary to validate or fine tune the controllers with the most detailed models available. Then effects such as sampling, AD-/DA-converter quantization, resolver quantization and noise, computing time of the control algorithms, signal communication times, as well as additional filters have to be taken into account.

Practical experience at DLR shows that it is difficult to maintain the consistency between a continuous and a digital representation of a control system model. For this reason, in a master thesis project (Walther 2002) a Modelica library was developed that allows to easily switch between a continuous and a discrete representation of a controller. This library has been in use at DLR for some years. Based on the gained experience in using this library, as well as new features in Modelica and in the Modelica simulation environment Dymola (Dynasim 2006), the library was considerably restructured, and completely newly implemented. Besides (Walter 2002), the books of (Aström and Wittenmark 1997) and (Tietze and Schenk 2002) have been helpful for the design and the actual implementation.

4.2 Introductory Example

In the left part of Figure 4 below a screenshot of the Sampled library and a simple example in using it is shown in the right part of the figure. The example is a simple controlled flexible drive consisting of a motor inertia, the gear elasticity and the load inertia. The angle and the angular velocity of the motor inertia are measured and are used in the position and speed controller. The output of the speed controller is directly used as the torque driving the motor inertia. The multi-rate controller consists of all blocks in the red square together with component *sampleClock*. The latter defines the base sampling time together with defaults for the blockType (Continuous or Discrete), the methodType (the used discretization method for a discrete block) and the block initialization (none, initialize states, initialize in steady state). All Sampled blocks on the same hierarchical level as *sampleClock*, and all blocks on a lower hierarchical level, use the *sampleClock* setting as a default. Every block can override the default and can use its individual settings.

In the example of Figure 5, block *filter* is a continuous filter that filters the reference motion of the motor (here: *ramp*). For this reason, *filter* uses explicitly the blockType *Continuous* whereas all other blocks use the blockType *UseSampleClockOption*, i.e., they use the setting defined in *sampleClock*. By changing the blockType from *Continuous* to *Discrete* in block *sampleClock*, the controller is automatically transformed from a continuous to a discrete representation.

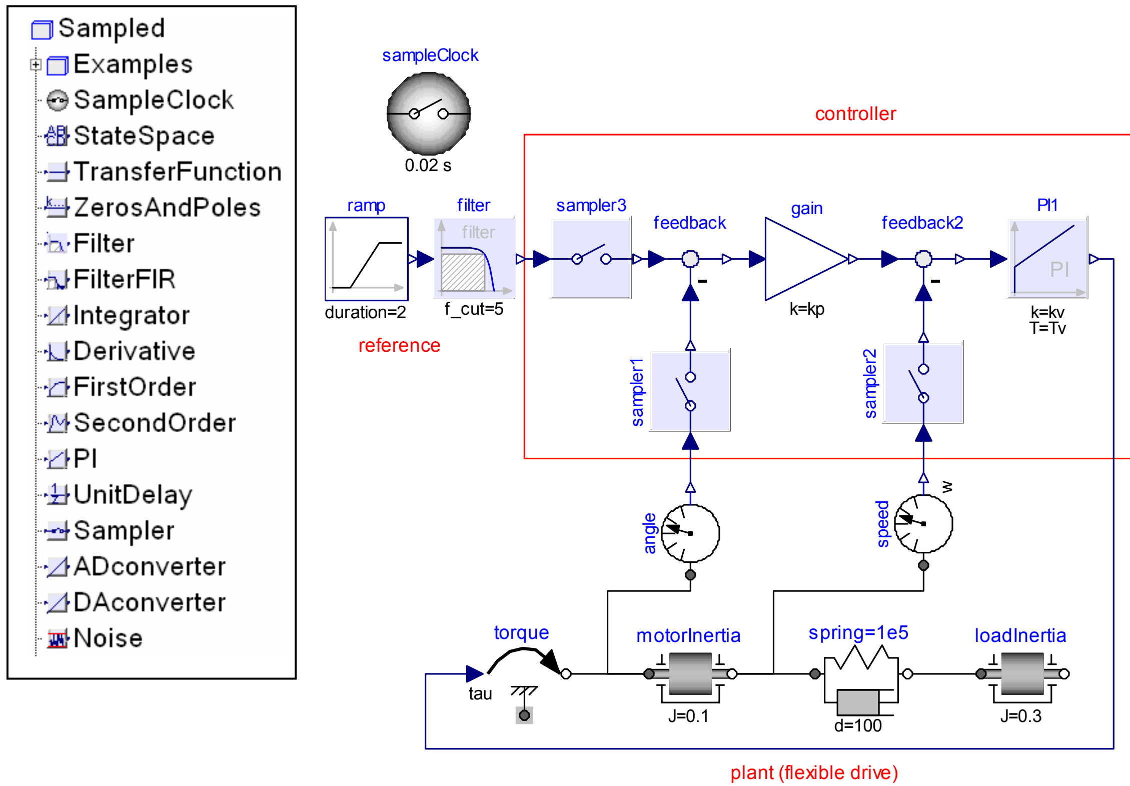


Figure 4: Blocks of Sampled library (left figure) and introductory example (right figure).

Every block of the Sampled library has a *continuous* input and a *continuous* output. Inside the respective block, the input and output signals might be sampled with the base sampling period defined in "sampleClock" or with an Integer multiple of it. For example, the PI controller in Figure 4 has the following parameter menu to define the continuous parameterization of the PI controller (i.e., gain *k* and time constant *T*):

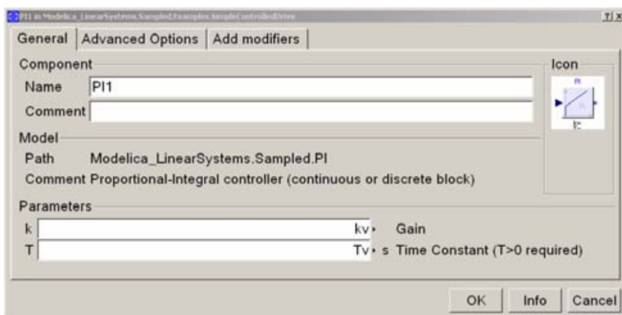


Figure 5: Parameter menu of PI controller

In the *Advanced Options* tab, see Figure 6, the remaining block settings are present, especially to define whether it is a continuous or a discrete block and in the latter case define also the discretization method. Since parameter *sampleFactor* is 1, the base sampling time of the *sampleClock* component is used. In other blocks of the controller (*Sampler1*, *Sampler2*), parameter *sampleFactor* = *5* which means that the inputs and outputs of these blocks are sampled by a sampling rate that is 5 times of the base sample time defined in the *sampleClock* component. Note, the *samplerX* blocks in Figure 4 have only an effect if the controller is discrete. For a continuous representation, these blocks just state that the output signal is identical to the input signal.

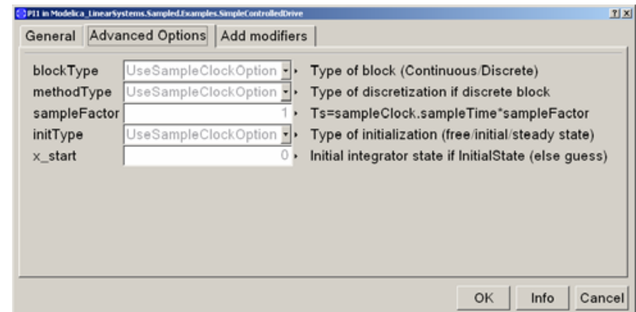


Figure 6: Advanced Options parameter menu

Finally, with parameter *initType* the type of initialization can be defined by using either the setting from the sampleClock component (default) or a local definition. The default of the sampleClock component initialization is "steady state". This means that *continuous* blocks are initialized so that the *state derivatives are zero* and *discrete* blocks are initialized so that a re-evaluation of the discrete equations gives

the *same discrete states* (provided the input did not change). Since the equations of the blocks of the Sampled library are linear, the default setting leads to linear systems of equations during initialization that can be usually solved very reliably. The "steady state" initialization for a control system or a filter has the significant advantage that unnecessary settling times after simulation start are avoided.

4.3 Discretization Methods

The core of the Sampled library is function *DiscreteStateSpace.fromStateSpace* that transforms a linear, time invariant **differential equation** system in state space form:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{x}(t) + \mathbf{B} \cdot \mathbf{u}(t)$$
$$\mathbf{y}(t) = \mathbf{C} \cdot \mathbf{x}(t) + \mathbf{D} \cdot \mathbf{u}(t)$$

into a linear, time invariant, **difference equation** system in state space form

$$\mathbf{x}_d\left((k+1)\cdot T_s\right) = \mathbf{A}\cdot \mathbf{x}_d\left(k\cdot T_s\right) + \mathbf{B}\cdot \mathbf{u}\left(k\cdot T_s\right)$$
$$\mathbf{y}\left(k\cdot T_s\right) = \mathbf{C}\cdot \mathbf{x}_d\left(k\cdot T_s\right) + \mathbf{D}\cdot \mathbf{u}\left(k\cdot T_s\right)$$
$$\mathbf{x}\left(k\cdot T_s\right) = \mathbf{x}_d\left(k\cdot T_s\right) + \mathbf{B}_2\cdot \mathbf{u}\left(k\cdot T_s\right)$$

where

- t is the time
- T_s is the sample time
- k is the index of the actual sample instance (k = 0, 1, 2, ...)
- **u**(t) is the input vector
- **y**(t) is the output vector
- **x**(t) is the state vector of the continuous system from which the discrete block has been derived.
- $\mathbf{x}_d$(t) is the state vector of the discretized system

If the discretization method, e.g., the trapezoidal integration method, accesses actual and past values of the input **u** (e.g. **u**(Ts*k), **u**(Ts*(k-1), **u**(Ts*(k-2))), a state transformation is needed to arrive at the difference equation above where only the actual value **u**(Ts*k) is accessed.

If the original continuous state vector at the sample times shall be computed from the discrete equations, the matrices of this transformation have to be known. For simplicity and efficiency, library LinearSystems supports only the specific transformation as needed for the provided discretization methods: As a result, the state vector of the underlying continuous system can be calculated by adding the term $\mathbf{B}_2$***u** to the state vector of the discretized system. In fact, since the **discrete** state vector depends on the discretization method, it is a protected variable that cannot be accessed outside of the block. This vector is also not needed by the user of the block, because the continuous state vector is available both in the continuous and in the discrete case.

In Modelica notation, the difference equation above is implemented as:

```
when {initial(), sample(Ts,Ts)} then
  new_x_d = A * x_d + B * u;
        y = C * x_d + D * u;
      x_d = pre(new_x_d);
        x = x_d + B2 * u;
end when;
```

Since no "next value" operator is available in Modelica, an auxiliary variable "`new_x_d`" stores the value of "`x_d`" for the next sampling instant. The relationship between "`new_x_d`" and "`x_d`" is defined via equation "`x_d = pre(new_x_d)`".

The body of the when-clause is active during initialization and at the next sample instant t = Ts. Note, the when-equation is **not** active after the initialization at t = 0 (due to sample(Ts,Ts) instead of sample(0,Ts)), since the state "`x_d`" of the initialization has to be used also at t = 0. Additional equations are added for the initialization to uniquely compute vectors "`x`" and "`x_d`" at the initial time::

```
initial equation
  if init == InitialState then
     x = x_start;
  elseif init == SteadyState then
     x_d = new_x_d;
  end if;
```

If option *InitialState* is set, the discrete state vector "`x_d`" is computed such that the continuous state vector "`x = x_start`", i.e., the continuous state vector is identical to the desired start value of the continuous state.

If option *SteadyState* is set, the discrete controller is initialized in steady state (= default setting in *sampleClock*). This means that the output **y**(T_s*k), k=0,1,2,.., remains constant provided the input vector **u**(T_s*k) remains constant. Most simulation systems support steady state initialization only for the continuous part of a model. Due to the equation based nature of Modelica, where basically everything is mapped to equations, it is possible in Modelica to initialize also a mixture of continuous and discrete equations in steady state.

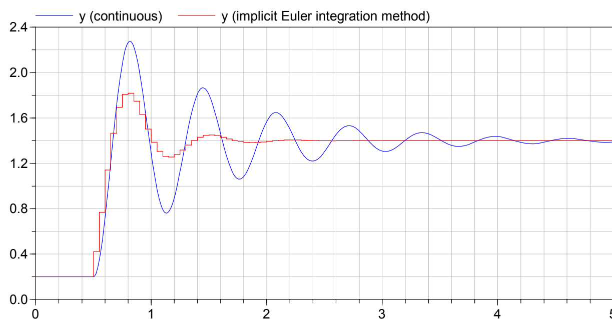
Via parameter *methodType* in the global block *sampleClock* or also individually for every block, the following discretization methods can be selected:

- explicit Euler integration method,
- implicit Euler integration method,
- trapezoidal integration method,

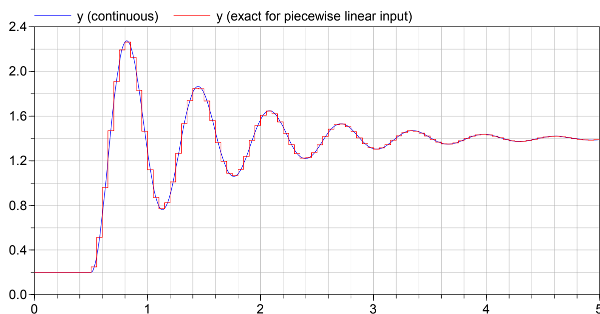
- exact solution under the assumption that the input signal is piecewise constant,
- exact solution under the assumption that the input signal is piecewise linear.

The last method (exact solution for piecewise linear input) gives nearly always the best approximation to the continuous solution without frequency or phase distortion. The effect of the discretization methods and of steady state initialization is demonstrated at hand of a simple PT2 block that is initialized in steady state, where the input signal is 0.2 at the beginning and jumps to 1.2 at 0.5 s

In the next figure, the blue curve is the solution of the continuous block and the red curve is the solution of the digital block using the **implicit Euler integration method** for the block discretization. As can be seen, even for this simple block the results of the continuous and digital representation are quite different. A much smaller sample time would be needed here, in order that the solutions of the discrete and the continuous representations would be closer together.



In the next figure, the red curve is the solution of the digital block using the **rampExact** method (i.e. the exact solution under the assumption that the input signal is piecewise linear). The solutions of the continuous and of the discrete blocks are practically identical without any phase shift. A better solution cannot be expected.



Continuous models in StateSpace form are transformed into discrete systems according to the sketched discretization methods. A TransferFunction system description is implemented as state space system in controller canonical form.

The transformation of a ZerosAndPoles system is more involved in order to reduce numerical difficulties, especially for filter implementations: The basic approach is to transform the transfer function

$$y = k \cdot \frac{\prod (s + n_{1i}) \cdot \prod (s^2 + n_{2j} \cdot s + n_{3j})}{\prod (s + d_{1k}) \cdot \prod (s^2 + d_{2l} \cdot s + d_{3l})} \cdot u$$

into a connected series of first and second order systems. All these systems are implemented as small state space systems in controller canonical form that are connected together in series. The output is scaled such that the gain of every block is one (if this is possible) in order that the inputs and outputs of the blocks are in the same order of magnitude. The output of the last block is multiplied with a factor so that the original gain of the transfer function is recovered.

A simpler, alternative implementation for both the TransferFunction and the ZerosAndPoles system would have been to compute the A,B,C,D matrices of the corresponding state space system with available function calls and then use the general StateSpace model for their implementations. The severe disadvantage of this approach is that the structure of the state space system is lost for the symbolic preprocessing. If, e.g., index reduction has to be applied (e.g. since a filter is used to realize a non-linear inverse model), then the tool cannot perform the index reduction anymore. Example:

Assume, a generic first order state space system is present

$$\dot{x} = a \cdot x + b \cdot u$$
$$y = c \cdot x + d \cdot u$$

and the values of the scalars a,b,c,d are parameters that might be changed before the simulation starts. If y has to be differentiated symbolically during code generation, then

$$\dot{y} = c \cdot \dot{x} + d \cdot \dot{u}$$
$$\dot{x} = a \cdot x + b \cdot u$$

As a result, the input u needs to be differentiated too, and this might not be possible and therefore translation might fail.

On the other hand, if the first order system is defined to be a low pass filter and the state space system is generated by keeping this structure, we have:

$$\dot{x} = -b \cdot x + u$$
$$y = x$$

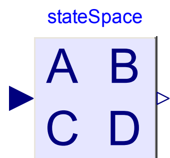
Differentiating y symbolically leads to:

$$\dot{y} = \dot{x}$$
$$\dot{x} = -b \cdot x + u$$

Therefore, in this case, the derivative of u is not needed and the tool can continue with the symbolic processing.

4.4 Block Components

A short description of the input/output blocks of the Sampled sublibrary is given in Table 1. As an example block Sampled.StateSpace is shortly described. Its icon is shown in the figure at the left. When double clicking on the block, the parameter menu of Figure 7 pops up. Here, either the name of a StateSpace record can be given, or when clicking on the "table" symbol at the right end of the input field another menu pops up in which the 4 matrices can be defined, as shown in Figure 8.



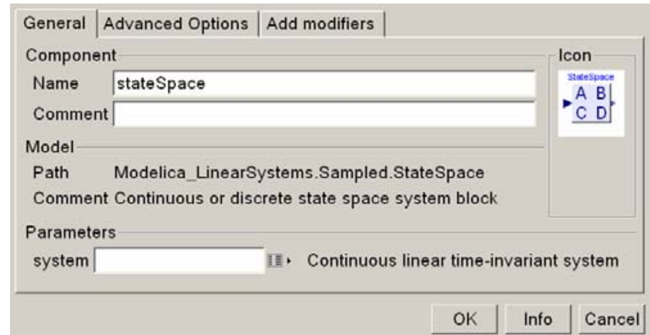


Figure 7: ParameterMenu of StateSpace block

By clicking again on the "table" symbol a matrix editor appears that allows to conveniently define a matrix.

Some of the blocks have only a pure discrete representation, such as the FilterFIR, UnitDelay, Sampler, ADconverter, DAconverter and Noise blocks. The continuous representation of these blocks is defined to be y = u, i.e., the output is identical to the input which means that these components are effectively removed.

Name	Description
SampleClock	Global options for blocks of Sampled library (e.g. base sample time)
StateSpace	Continuous or discrete state space system block
TransferFunction	Continuous or discrete, single input single output transfer function
ZerosAndPoles	Continuous or discrete, single input single output block described by zeros and poles
Filter	Analog low or high pass IIR-filters (CriticaDaming/Bessel/Butterworth/ Chebyshev)
FilterFIR	Discrete finite impulse response (low or high pass) filters
Integrator	Output the integral of the input signal (continuous or discrete block)
Derivative	Approximate derivative (continuous or discrete block)
FirstOrder	First order (continuous or discrete) transfer function block (= 1 pole)
SecondOrder	Second order (continuous or discrete) transfer function block (= 2 poles)
PI	Proportional-Integral controller (continuous or discrete block)
UnitDelay	Delay the input by a multiple of the base sample time if discrete block or y = u if continuous block
Sampler	Sample the input signal if discrete block or y = u if continuous block
ADconverter	Analog to digital converter (including sampler)
DAconverter	Digital to analog converter (including zero order hold)
Noise	Generates uniformly distributed noise in a given band at sample instants if discrete and y = 0 if continuous

Table 1: Blocks of the Sampled sublibrary.

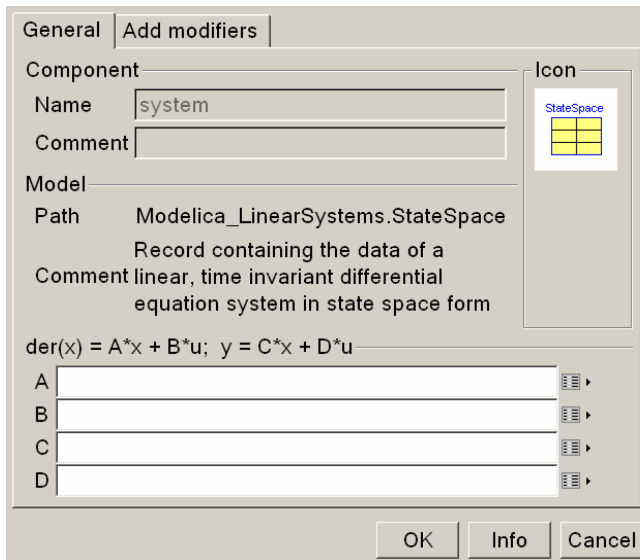


Figure 8: ParameterMenu of StateSpace record

5 Conclusion

The new, free LinearSystems library has been presented and important implementation details have been discussed. The library introduces a standard definition of the most important linear systems used for controller definitions and provides continuous and discrete block representations of every block. Some standard blocks, such as the Filter block for the most important IIR filters and the FilterFIR block for the most important FIR filters are provided.

The library is currently available in a Beta release. The available functionality is already very useful for controller simulations. For a first 1.0 release, some missing functions and components will be added and the documentation will be improved.

6 Acknowledgements

The first version of the LinearSystems.Sampled library was the library "Sampled" of Nico Walther as a result of his master thesis from the electrical engineering at the HTWK-Leipzig (supervised by Prof. Müller, HTWK, and Prof. Martin Otter, DLR). Based on the experience in using the Sampled library, new features in Modelica as well as in Dymola, the Sampled library was considerably restructured, and newly implemented. Not all blocks from the previous version are yet included.

Some functionality of LinearSystems (e.g., linearizing a Modelica model by StateSpace.fromModel) has been originally developed by Sven Erik Mattsson from Dynasim.

The Math.Complex, Math.Polynomial, and TransferFunction packages are based on proposals from Hilding Elmqvist, Dynasim, presented at the 33rd Modelica design meeting in Bielefeld (Nov. 2003) and the 37th Modelica design meeting in Lund (Jan. 2004).

Several functions of package Complex have been provided by Anton Haumer, who also performed a thorough test of the package.

The design of the records (such as Math.Complex and Math.Polynomial) has been inspired by the discussions about operator overloading at various Modelica design meetings.

Advice for implementation issues given by Hans Olsson from Dynasim, as well as advice for some numerical algorithms given by Andras Varga and Dieter Joos from DLR is appreciated.

Partial financial support of DLR for the development of this library within the European Network of Excellence HYCON (Hybrid Control: taming heterogeneity and complexity of networked embedded systems; contract number: 511368), and within the German BMBF Verbundprojekt PAPAS (Plug-And-Play Antriebs- und Steuerungskonzepte für die Produktion von Morgen; Förderkennzeichen: 02PH2060) is highly appreciated.

References

Aström K.J., Wittenmark B. (1997): *Computer Controlled Systems: Theory and Design*. Prentice Hall. 3rd edition.

Dynasim (2006). *Dymola Version 6.0*. Dynasim AB, Lund, Sweden. Homepage: http://www.dynasim.se/.

Tietze U., and Schenk C. (2002): *Halbleiter-Schaltungstechnik*. Springer Verlag, 12. Auflage, pp. 815-852.

Walther N. (2002): *Praxisgerechte Modelica-Bibliothek für Abtastregler*. Diplomarbeit, HTWK Leipzig, Fachbereich Elektro- und Informationstechnik, supervised by Prof. Müller (HTWK) and Prof. Martin Otter (DLR), 12 Nov. 2002.