Scilab tutorial oriented toward the Practice of Discrete-Time Signal Processing

Alexandre Trilla, Xavier Sevillano

Departament de Tecnologies Mèdia
LA SALLE – UNIVERSITAT RAMON LLULL
Quatre Camins 2, 08022 Barcelona (Spain)
atrilla@salle.url.edu, xavis@salle.url.edu

2010

Abstract

In this tutorial, Scilab is used for signal processing. The several tools needed for completing the Practice of Discrete-Time Signal Processing are described hereunder. Keep it for reference and use it at your convenience.

1 About Scilab

Scilab is an open source, numerical computational package and a high-level, numerically oriented programming language. Scilab provides an interpreted programming environment, with matrices as the main data type. By utilising matrix-based computation, dynamic typing, and automatic memory management, many numerical problems may be expressed in a reduced number of code lines, as compared to similar solutions using traditional programming languages.

Scilab binaries for GNU/Linux, Windows and Mac OS X platforms can be downloaded directly from the Scilab homepage:

http://www.scilab.org

The homepage already detects the computer's platform and shows a "Download Scilab" banner to ease the download process. Then the installation instructions must be followed. They are available in the "Support / Documentation / Tutorials" section (Introduction to Scilab) or in the README files bundled with the downloaded tarball.

Scilab was created in 1990 by researchers from Institut National de Recherche en Informatique et en Automatique (INRIA) and 'Ecole Nationale des Ponts et Chauss'ees (ENPC). The Scilab Consortium was formed in May 2003 to broaden contributions and promote Scilab as worldwide reference software in academia and industry. In July 2008, the Scilab Consortium joined the Digiteo Foundation. Scilab is in continuous development; it is an ongoing project in the Google Summer of Code.

2 Scilab environment

Once Scilab is launched, it provides a Command Line Interface on a console to introduce the commands, and interprets them interactively after each carriage return. The syntax interpreted by Scilab and some specific signal processing functions are presented with a set of examples.

2.1 Variables

Variables are defined by case-sensitive alphanumeric identifiers (underscores are also allowed). For example, to represent an integer number like $a = 1$ in Scilab:

```
--> a = 1
 a =
    1.
```

or a more general complex number like $a = 1 + 3j$:

```
--> a = 1 + 3 * %i
 a =
    1.  + 3.i
```

Note that Scilab uses the mathematical notation of i to refer to $\sqrt{-1}$ instead of the common j used in the Theory. The `%` character before `i` is used to escape the alphanumeric character of 'i' that would generally refer to a (non-predefined) variable.

2.2 Functions

Some useful functions for operating with a complex number `x` are:

- The absolute value/magnitude `abs(x)`
- The phase angle `angle(x)`
- The square root `sqrt(x)`
- The real part `real(x)`
- The imaginary part `imag(x)`
- The complex conjugate `conj(x)`
- The decimal logarithm `log10(x)`
- The natural logarithm `log(x)`
- The element-wise exponential `exp(x)`, i.e. e^x

Regarding these functions, note that number $a = 1+3j$ may also be defined in polar form:

```
--> a = abs(1 + 3 * %i) * exp(%i * angle(1 + 3 * %i))
 a =
    1.  + 3.i
```

despite the console still displays it in binomial form. Note that some functions may include other functions, like the phase angle function described above.

Additionally, some useful trigonometric functions are:

- The sine `sin(x)`
- The cosine `cos(x)`
- The sine inverse `asin(x)`
- The cosine inverse `acos(x)`
- The tangent `tan(x)`
- The inverse tangent `atan(x)`
- The hyperbolic sine `sinh(x)`
- The hyperbolic cosine `cosh(x)`
- The hyperbolic tangent `tanh(x)`

And some useful signal processing functions are:

- The convolution `convol(h,x)`
- The Discrete Fourier Transform `fft(x)`
- The group delay `group(·)`

In case of doubt regarding a function and/or its usage, the command `help` is of use to access a Help Browser.

There are some other functions in Scilab that enable interfering with the program flow in order to perform some specific tasks. For example, see function `pause`. The pause is extremely useful for debugging purposes; it will be handy as the complexity of the practice exercises grows ⌣̈

2.2.1 Creation of functions

Scilab enables the execution of self-produced functions, stored in a "∗.sci" file. These files contain the sort of commands seen above, in addition to control flow commands like conditional executions (`if..else..end`), iterative executions (`for..end`), etc., pretty much like the majority of programming languages. Check the Help Browser for further information.

In order to create a personalised function, Scilab provides an Editor application, but any editor will do. A sample Scilab function "examplefunc.sci" follows:

```
// Comments
function [r1] = examplefunc(p1)
    ...
    (code)
    ...
    r1 = 1; // Return
endfunction
```

where the `examplefunc` receives parameter `p1` and returns `r1`.

Finally, in order to run the function, Scilab must be aware of its existence through the `exec(''examplefunc.sci'')` command. The file must be accessible in the working directory path `pwd`, which can be reached with the change directory `cd` command, like on a Unix box.

2.3 Matrices

In the end, every variable refers to a matrix. In the former examples, number a was stored in a 1×1 matrix. Row vectors ($1 \times N$ matrices) may also be defined in Scilab with the square brackets:

```
--> rv = [ 1 2 3 4 ]
 rv =
    1.   2.   3.   4.
```

where N is the length of the vector ($N = 4$ in this example). For vectors, there exists a function called `length(x)` that yields N.

In order to obtain a column vector ($N \times 1$ matrix), the rows are separated with the semicolon:

```
--> cv = [ 1; 2; 3; 4 ]
 cv =
    1.
    2.
    3.
    4.
```

In order to access and use the content of a vector, the brackets are used as dereference operators, bearing in mind that vectors are indexed following [1 2 ... N-1]:

```
--> cv(3)
 ans =
    3.
```

Note that when the result of a command is not assigned to a variable, Scilab assigns its to the default "answer" (`ans`) variable. `ans` contains the last unassigned evaluated expression.

Matrices are defined using both the row vector and column vector syntax:

```
--> m = [ 11 12 13; 21 22 23 ]
 m =
    11.   12.   13.
    21.   22.   23.
```

The numbers contained in this matrix example also indicate their index, so that:

```
--> m(1,2)
 ans =
    12.
```

The colon mark is of use to indicate all the elements in a row or column:

```
--> m(1,:)
 ans =
    11.   12.   13.
```

where all the elements in the first row are displayed.

More enhanced dereferencing would include sequences. For example:

```
--> 1:3
 ans =
```

```
    1.  2.  3.
```

indicates a sequence from 1 to 3, with a default increment of 1 unit, which result would be equal to $-- >$ `1:1:3`. Alternatively,

```
--> 1:2:3
 ans =
    1.  3.
```

indicates a sequence from 1 to 3, with an increment of 2 units, thus resulting in 1 and 3. If this trick is applied on dereferencing a matrix variable:

```
--> m(1,1:2:3)
 ans =
    11.  13.
```

Increments may also be negative, thus resulting in decreasing sequences:

```
--> 3:-2:1
 ans =
    3.  1.
```

In order to retrieve the size of a matrix, the function `size(x)` is of use:

```
--> size(m)
 ans =
    2.  3.
```

where the first number indicates the number of rows and the second the number of columns.

Operation among elements (scalars, vectors and matrices) include the dot/scalar product (*), the matrix addition (+), the matrix subtraction (–), the matrix multiplication (*), the matrix left division (\), the matrix right division (/), the transpose (') and the power (ˆ). Term-by-term multiplication (.*) and division (./) are also allowed.

Finally, to list the variables used in a session, the `whos` function displays their names, types, sizes and number of bytes. And in order to prevent the console from being too verbose, by appending a a semicolon at the end of a command, the result is just not displayed.

2.4 Signals

In general, signals are represented by vectors, i.e. arrays of (not yet quantised) samples. Inherently, the temporal difference that exists between two consecutive samples (positions i and $i + 1$ in the vector) equals to one sampling period.

For practical purposes, the signals generally refer to audio signals, and the WAV format is of use for their portability. Therefore, in order to load a WAV sound file into a `y` variable: `y = wavread("file.wav");`. More

information may be obtained from the WAV metadata through: `[ y, fs, bit ] = wavread("file.wav");`, where `fs` keeps the sampling frequency and `bit` keeps the number of bits used to encode each sample.

After the samples of a sound file are loaded into a variable, they can be then reproduced by the computer's soundcard through the `sound(y,fs)` command. And in order to save the signal in a WAV file, the command `wavwrite(y,fs,bit,"file.wav")` is of use. Note that Scilab always autoscales the signal in order to use the maximum dynamic range available, therefore there is no need to multiply the signal by a constant in order to raise the volume.

The following sections present common instructions to deal with signals in Scilab.

2.4.1 Generation of sinusoids

The equivalent sampling process (in a Continuous-to-Discrete conversion) of a sinusoid signal may be generated with the sine and cosine functions. The following example supposes that the real signal $x(t)$ is sampled at $f_s = 11025\,Hz$.

$$x(t) = A\,\sin(\omega t + \phi) = 2\,\sin\left(2\pi\,10\,t + \frac{\pi}{2}\right)$$

Note that $x(t)$ has an amplitude peak of 2 units, a frequency of 10 Hz and a phase of $\frac{\pi}{2}$ radians. Considering that the discrete-time representation of $x(t)$, therefore $x[n]$, is given for $t = nT = \frac{n}{f_s}$, it results:

$$x[n] = x(t)\Big|_{t = nT} = \sin\left(2\pi\,10\,T\,n + \tfrac{\pi}{2}\right)$$

In order to obtain $x[n]$ in Scilab:

1. Generate an index vector `n`, i.e. a row vector, like `n = 1:1000`. Note that the final sampled signal will thus contain 1000 samples
2. Compute $x[n]$ like `x = 2 * sin ( 2 * %pi * 10 * n / 11025 + %pi / 2);`

In order to visualise the discretised signal, the function `plot(x)` is of use. If applied directly on the `x` vector variable, its sample indices are taken for the plot abscissa, see Figure 1. Otherwise, $x(t_n)$ may also be plotted vs. its temporal indices through `plot(n/11025,x)`.

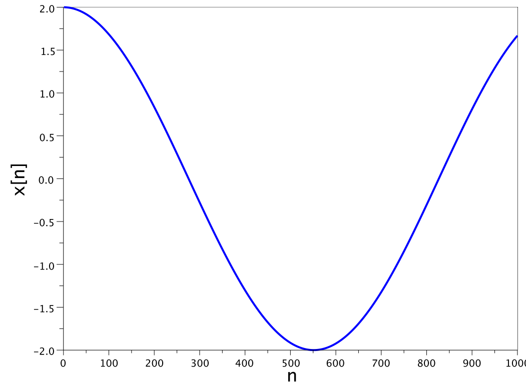

Figure 1: Plot of $x[n]$ vs. its sample indices.

2.4.2 Generation of signal delays

In order to generate a signal that is composed of several delays of a base signal, the function `delay(x,T,w,fs)` is of use, where:

- `x` Base signal
- `T` Delay vector $T = [t_1 t_2 ... t_n]$ (in seconds)
- `w` Weight vector (the amplitude weight of the base signal corresponding to each delay $w = [w_1 w_2 ... w_n]$)
- `fs` Sampling frequency

Hence, the `delay` function produces a signal $y(t)$ according to

$$y(t) = \sum_{k=1}^{n} w[k]\, x(t - T[k])$$

which implies that $T[k]$ is always positive (a delay always goes after the base signal, it is like a kind of echo).

For example, taking the former sampled signal $x[n]$ shown in Figure 1 as the base signal `x` (this signal lasts 90.70ms), with `y = delay(x,1e-1*[0 2 5],[1 0.5 -2],11025);` a new signal $y[n]$ would be obtained as a sampled version of $y(t) = x(t) + 0.5\, x(t - 200ms) - 2\, x(t - 500ms)$, shown in Figure 2.

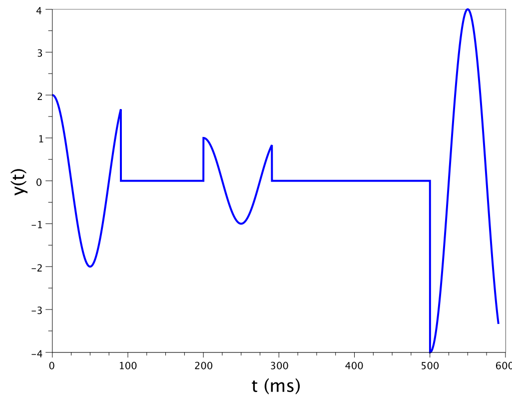

Figure 2: Example delay of $x[n]$.

2.4.3 Generation of white Gaussian noise

White Gaussian noise is a random signal, produced by a random number generator with a Gaussian distribution (with mean 0 and variance 1). Since this noise is white, it contains frequencies along the whole spectrum. In order to generate a sequence of N samples of white Gaussian noise, the `rand(1:N,‘‘normal’’)` function is of use.

The average power of the generated white Gaussian noise, determined by a Gaussian random variable X, by definition is given by its second-order moment, i.e. its variance:

$$E(X^2) = \int_{-\infty}^{\infty} x^2 p_X(x)\, dx = \sigma_X^2$$

where E is the mathematical expectation operator, and $p_X(x)$ is the normal probability density function.

Therefore, scaling the average power of the white Gaussian noise by a factor p, implies scaling the specific values x of the Gaussian random variable X by a factor $\sqrt{p}$.

2.4.4 Filter design

This tutorial presents a design technique for linear-phase Finite Impulse Response (FIR) filters, based on directly approximating their desired/ideal frequency responses by windowing their corresponding impulse responses.

To this purpose, the function `[h,H,ft] = h_filter(G,N,fc1,fc2,fs)` is of use, where:

- `G` Gain of the filter's bandpass
- `N` Number of points of the impulse response. The greater this value, the more ideal the filter is, and thus the better it adjusts to the ideal frequency specifications
- `fc1, fc2` Filter's bandpass described by its ideal cutoff frequencies (-3dB), where:

$$f_{c1} < f_{c2}$$

$$f_{c1}, f_{c2} \in [0, \frac{f_s}{2}]$$

- `fs` Sampling frequency, expressed in Hertz
- `h` Impulse response of the filter
- `H` Frequency response of the filter (4096 points)
- `ft` Frequency test points of `H`. $H[k]$ corresponds to the frequency response of the filter at frequency $f[k]$ Hz.

For example, `[h,H,ft] = h_filter(1,20,0,1000,16000)` creates a low-pass filter with a cutoff frequency of 1 KHz, a unitary gain and a 20 points long impulse response, working with a sampling frequency of 16000 Hz.

The function `plot(ft,abs(H))` represents graphically the absolute value of the gain of the filter versus its frequency response, see Figure 3. By clicking "Edit/Figure properties" the graphical format of the figure may be changed at will.

Similarly, `plot(h)` represents graphically the corresponding impulse response of the filter, see Figure 4.

A comprehensive analysis of the design technique implemented in the `h_filter` function is presented in [Trilla and Sevillano, 2010].

2.4.5 Filter usage

Once a filter is designed, it may then be used with the `filter(num,den,x)` function, where:

- `num` Numerator of the transfer function of the filter
- `den` Denominator of the transfer function of the filter

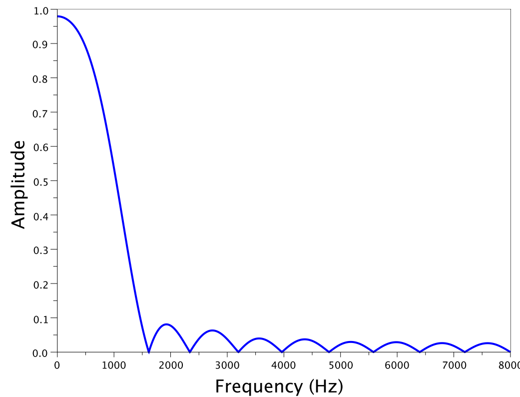

Figure 3: Example magnitude frequency response $|H(e^{j\omega})|$.

- `x` Input signal

In case the designed filter has a FIR (e.g. produced with the `h_filter` function), then `den = 1`, and the coefficients of the impulse response directly define the numerator of the transfer function, therefore `filter(h,1,x)`.

It should be allowed to use the `convol` function (convolution) to attain the same result as with the `filter` function, but for practical purposes the latter is preferred (it is implemented with the Direct Form II Transposed instead of the Fast Fourier Transform, refer to [Oppenheim and Schafer, 2009] for further details).

3 Xcos

Xcos is a Scilab toolbox for the modelling and simulation of dynamical systems. Xcos is particularly useful for modelling systems where continuous-time and discrete-time components are interconnected.

Xcos provides a Graphical User Interface to construct complex dynamical systems using a block diagram editor. These systems can in turn be reused following a modular structure.

3.1 Running Xcos

Xcos can be launched with the `xcos;` command in the Scilab command-line, or with the “Applications / Xcos” option, or even with an icon shortcut

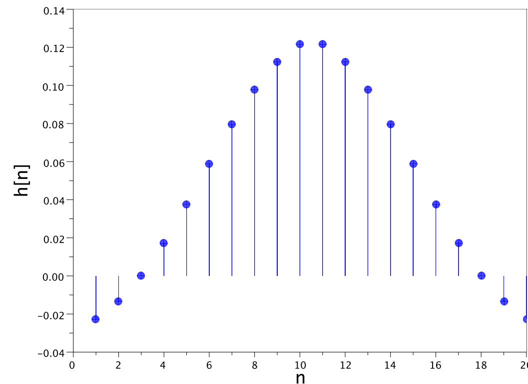

Figure 4: Example impulse response $h[n]$.

under the Scilab menus. Once Xcos is loaded, it displays the Palette browser and an empty diagram, see Figure 5.

3.2 Diagram edition

Xcos displays a graphical editor that can be used to construct block diagram models of dynamical systems. The blocks can come from various palettes provided in Xcos or can be user-defined. Editor functionalities are available through pull-down menus placed at the top of the editor window.

3.2.1 Blocks

Xcos provides many elementary blocks organised in different palettes that can be accessed using the operation "View / Palette browser". Blocks from palettes can be copied into the main Xcos diagram editor window by right-clicking (context-menu button) first on the desired block and then clicking the "Add to" option, or just by dragging and dropping them into the diagram editor.

The behaviour of a Xcos block may depend on parameters that can be modified. These parameters can be changed by double-clicking on the block. This action opens up a dialogue box showing the current values of the block parameters, and allowing the user to change them.

Often it is useful to use symbolic parameters to define block parameters. This is particularly the case if the same parameter is used in more than one

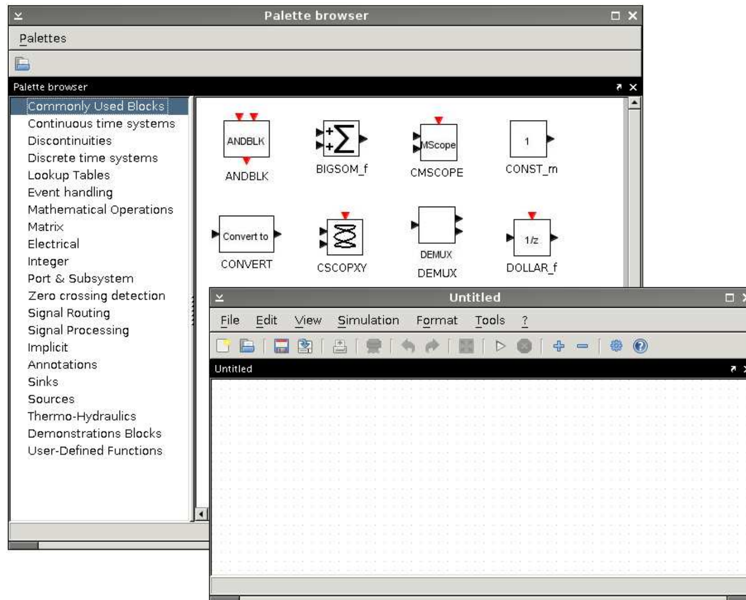

Figure 5: Xcos loaded. The Palette browser and an empty diagram are displayed.

block or if the parameter is computed as a function of other parameters. Symbolic parameters are simply Scilab variables that must be defined in the "context" of the diagram before being used in the definition of block parameters. To access the context of a diagram, use the "Simulation / Set Context" option. This opens up an editor to introduce context variables (or even a Scilab script).

Sometimes it is useful to enclose several blocks in one super-block, for its ease of interpretation (and debugging) in a higher-level diagram. In order to do so, select the blocks that are to be enclosed in a super-block and then choose the "Region to superblock" option of the context menu. The construction of a more complex hierarchy (with enhanced super-block behaviours) is beyond the scope of this tutorial, refer to [Campbell et al., 2005] for further details.

3.2.2 Links

In order to interconnect the blocks in the diagram window, their input/output ports (i.e. the arrows) have to be linked by dragging them from one to another. Splits on links can also be created by dragging the existing links to

the new ports.

Xcos diagrams contain two different types of links. The regular links transmit signals and the activation links transmit activation timing information (synchronism), i.e., system events. By default, the activation links are drawn in red and the regular links in black. And by convention, regular ports are placed on the sides of the blocks whereas activation ports are respectively on the top and at the bottom of the blocks.

3.2.3 Synchronism and special blocks

When two blocks are activated with the same activation source (they have the same activation times) it is said that they are synchronized. Alternatively, if the output of one is connected to the input of the other, the compiler makes sure the blocks are executed in the correct order. Synchronism is an important property. Two blocks activated by two independent clocks exactly at the same time are not synchronized. Even though their activations have identical timing, they can be activated in any order by the simulator.

There are two very special blocks in Xcos: the If-then-else block "Event handling / IFTHEL_f" and the event select "Event handling / ESELECT_f" block. These blocks are special because they are the only blocks that generate events synchronous with the incoming event that had activated them (they introduce no synchronism delay).

3.2.4 Save and load

Xcos diagrams are saved in a XML file format, following the "File / Save" option, and giving the saved diagram a "*.xcos" file extension. Text files are of great utility in software development for enabling the use of incremental (backup) copies and the use of a version control system.

Saved diagrams may be loaded at Xcos launch time through the `xcos diagram.xcos;` command, or once in Xcos, through the "File / Open" option.

3.3 Diagram simulation

To simulate a diagram, the "Simulation" menu is of use.

3.3.1 Setup

Simulation parameters can be set by the "Simulation / Setup" operation. For example, one useful parameter to adjust is the final simulation time (in

seconds), otherwise the simulation goes on for a very long period of time (100000 seconds by default). In order to do so, in the "Set Parameters" window that pops up, adjust the "Final integration time" to the number of samples that are to be simulated.

3.3.2 Start

To start a simulation, click on the "Simulation / Start" option, or the corresponding icon shortcut.

Running the simulation for a diagram leads to the opening of convenient graphics windows to display the output signal of the simulation and/or any other signals of interest set to be displayed (windows are opened and updated by the scope block, see Section 3.4.2). A simulation can be stopped using the "Simulation / Stop" option or the corresponding icon shortcut, subsequent to which the user has the option of continuing the simulation, ending the simulation, or restarting it.

3.4 Signal processing blocks and functions

Some of blocks are already available in Xcos palettes. These blocks provide elementary operations needed to construct models of many dynamical systems. The most important blocks for the Practice of Discrete-Time Signal Processing are reviewed in this section (describing their parameters and output). In case that any useful blocks are not directly available in Xcos, design instructions are given to obtain them.

3.4.1 Signals

Signal blocks are output systems, aka sources. They provide the input signals to the diagrams.

Constant Given the "Constant" amplitude value, it outputs a signal that maintains this value during the whole simulation interval. It is directly implemented in the "Sources / CONST_m" block.

Unit step Given the "Step time", the "Initial value" (0 by default) and the "Final value" (1 by default), it outputs a step function at the specified time. It is directly implemented in the "Sources / STEP_FUNCTION" block.

Unit rectangular pulse Defined by the duration of the pulse dp and the time shift ts (initial time, when the pulse begins to rise). It may be

implemented with the difference between two unit step blocks and one addition block "Mathematical Operations / SUMMATION". The first unit step is adjusted to rise at ts and the second at $ts + dp$, see Figure 6.

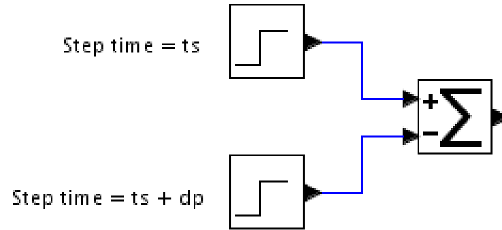

Figure 6: Xcos diagram of a unit rectangular pulse.

Unit triangular pulse Defined by the same parameters as the unit step plus the duration of the pulse. It may be implemented by firstly summing three step function blocks with an addition block "Mathematical Operations / BIGSOM_f", and then integrating the sum with one integral block "Continuous time systems / INTEGRAL_m". The first step block begins at ts and has a final value of $\alpha = \frac{2}{dp}$, the second step begins at $ts + \frac{dp}{2}$ and has a final value of -2α, and the third step begins at $ts + dp$ and has a final value of α. Also, the addition block "Inputs ports signs/gains" parameter needs to be set to `[1;1;1]` in order to define three input ports, see Figure 7.

Sinusoidal wave Given the "Magnitude" amplitude value, the "Frequency" and the "phase", it outputs a sinusoidal signal according to the given parameters. It is directly implemented in the "Sources / GENSIN_f" block. The frequency is set in radians/second, i.e., $\omega = 2\pi f$.

Square wave Given the "Amplitude" M parameter, for every input event it outputs M and $-M$ alternatively. It is directly implemented in the "Sources / GENSQR_f" block.

The needed activation events may be generated with an activation clock block "Sources / CLOCK_c", given the desired "Period" and "Init time". And with an adequate delay of synchronism, a rectangular pulse train may be obtained.

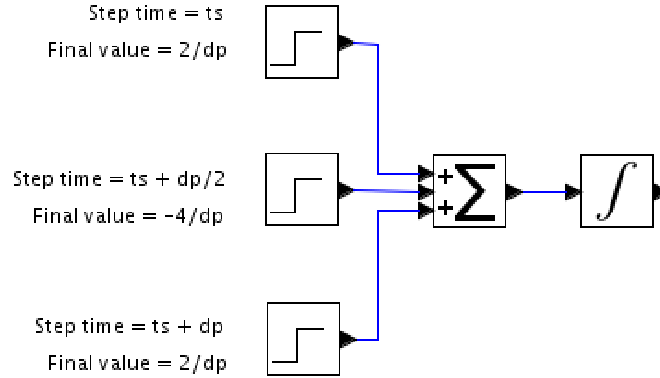

Figure 7: Xcos diagram of a unit triangular pulse.

Sawtooth wave While no activation event is present, it outputs a ramp by the rate of one amplitude unit per second. Otherwise, it sets to zero and then the cycle begins another time. It is directly implemented in the "Sources / SAWTOOTH_f" block.

Chirp Defined by a sinusoidal signal with variable frequency. It can be expressed as

$$x(t) = cos(2\pi f(t)\, t)$$

where $f(t)$ implements a linear frequency transition from f_0 to f_1 in T seconds

$$f(t) = \frac{f_1 - f_0}{T}\, t + f_0$$

The chirp may be implemented with an arbitrary mathematical expression block "User-Defined Functions / EXPRESSION", three reference constant blocks and one time function block "Sources / TIME_f", see Figure 8. The mathematical expression block has to be adjusted to 4 "number of inputs" and its "scilab expression" has to be `cos(2 * %pi * (((u2-u1)/u3)*u4+u1) * u4)`.

Noise Defined by a random number distribution (say Normal) and a power G. It may be implemented with a random generator block "Sources / RAND_m", a gain block "Mathematical Operations / GAINBLK_f" and an activation clock block "Sources / CLOCK_c" to drive the random generator, see Figure 9. The "flag" parameter of the random

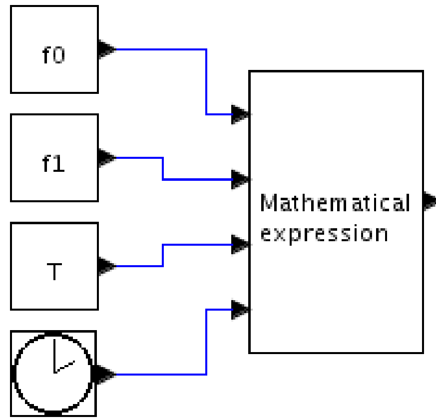

Figure 8: Xcos diagram of a chirp signal.

generator has to be set to 1 in order to obtain a Normal distribution, and the “Gain” parameter of the gain block has to be set to the squared root of the desired power gain G.

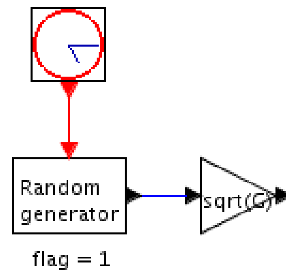

Figure 9: Xcos diagram of a Gaussian noise signal with power G.

Rectangular pulse train Defined by the duration of the rectangular pulse dp and the period of the pulse train $T > dp$ (time difference between two consecutive ascending or falling signal flanks). It may be implemented with a unit rectangular pulse block, a delay block “Continuous time systems / TIME_DELAY” and an addition block “Mathematical Operations / BIGSOM_f”, see Figure 10.

Dirac comb Defined by the period T of the impulse train. Note that the Dirac delta function does not exist in reality, hence it is approximated by a triangular function of total integral 1 and a pulse duration of

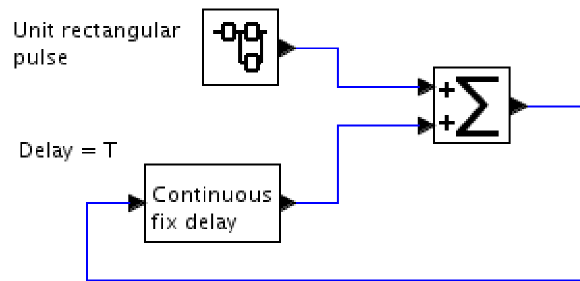

Figure 10: Xcos diagram of a rectangular pulse train.

one tenth of T. The approximation is continuous and compactly supported, although not smooth and so not a mollifier (aka approximation to the identity).

With the knowledge acquired to develop the previous basic signals, many (other) arbitrary signals may be "easily" built.

3.4.2 Basic systems

Basic system blocks enable the modification/processing of signals. Sometimes, for a certain application, more than one block may accomplish the desired task. General guidelines are given in this section. For further details, refer to the "Block help" option of the context menu for each palette block.

Addition/Subtraction Performs (positive or negative) addition of its inputs. It is implemented in the "Mathematical Operations / BIGSOM_f" block. Its "Inputs ports signs/gain" parameter permits specifying the number of ports of the block as well as its (positive or negative) port gains like $[g_1; g_2; ...; g_N]$.

Product/Division Performs the product or division of its inputs. It is implemented in the "Mathematical Operations / PRODUCT" block. Its "Number of inputs or sign vector" parameter defines which input ports are multipliers (positive sign) and which ones are dividends (negative sign).

Amplification Applies a given "Gain" to the input signal. It is implemented in the "Mathematical Operations / GAINBLK_f" block.

Integration Integrates the input signal. It is implemented in the "Continuous time systems / INTEGRAL_m" block.

Derivation Derives the input signal. It is implemented in the "Continuous time systems / DERIV" block.

Absolute value Calculates the absolute value of the input signal. It is implemented in the "Mathematical Operations / ABS_VALUE" block.

Sign Calculates the sign of the input signal value x, i.e., it outputs -1 for $x < 0$, 0 for $x = 0$ and 1 for $x > 0$. It is implemented in the "Mathematical Operations / SIGNUM" block.

Delay Given a "Delay" value, it delays the input signal accordingly. It is implemented in the "Continuous time systems / TIME_DELAY" block. Its initial value may also be set with the "initial input" parameter.

Saturation Given an "Upper limit" and a "Lower limit", it delivers an output signal value that never exceeds these limits. It is implemented in the "Discontinuities / SATURATION" block.

Mathematical expression Given the "number of inputs" (i.e., u1, u2, ...) and a "scilab expression", it permits evaluation a mathematical expression (using addition/subtraction, product/division and power raising). It is implemented in the "User-Defined Functions / EXPRESSION" block.

Filter Given the "Numerator" and the "Denominator" of the desired filter in Laplace domain (obtained with any design method), it filters the input signal. It is implemented in the "Continuous time systems / CLR" block.

Sample and Hold Each time an activation event is received, it copies the signal value of its input on its output, and holds it until a new activation event is received. It is implemented in the "Signal Processing / SAMPHOLD_m" block.

Quantisation Given a "Step" size (in amplitude units), it outputs the input signal (with continuous amplitude) with discrete amplitude values. It is implemented in the "Signal Processing / QUANT_f" block.

Finally, as a special block due to its extensive use in dynamic systems modelling, the scope block is presented.

Scope Given an activation event, it plots the present input signal value onto a graphic window display. Therefore, the scope permits the visualisation of a given signal during the simulation. It is implemented in the "Sinks / CSCOPE" for a single input signal, or in the "Sinks / CMSCOPE" for multiple input signals.

References

[Campbell et al., 2005] Campbell, S. L., Chancelier, J.-P., and Nikoukhah, R. (2005). *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, New York, NY, USA, first edition.

[Oppenheim and Schafer, 2009] Oppenheim, A. V. and Schafer, R. W. (2009). *Digital Signal Processing*. Prentice–Hall, third edition.

[Trilla and Sevillano, 2010] Trilla, A. and Sevillano, X. (2010). Filter analysis and design. (Web-Available).