DiaBloS

Release 0.9.11b

mnrojas2

Jan 19, 2024

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Introduction

DiaBloS (Diagram Block Simulator) is a block programming tool, focused on the simulation of systems represented as ordinary differential equations.

This tool is presented as an open source alternative independent of software or libraries external to Python, since it only relies on packages already provided by Python, or available through pip.

This manual is separated into two parts: A novice's guide, which explains how to use the library, mainly focused on the interface, and a programmer's guide, focused on explaining the hierarchy of functions and the most important algorithms, with the idea of making it easier for other people to contribute to the project.

1.1.1 Capabilities of the DiaBloS package

Python by itself does not provide a library to visually simulate dynamic system models, nor does it offer a visual programming alternative to paid software, such as Simulink from Mathworks.

There are Python libraries created by the open source community that allow modeling interconnected dynamic systems, such as SimuPy, as well as others that provide an interface to analyze models created and used by external programs, such as PySimulator. However, none of these can produce these dynamic models visually, requiring direct programming, or the use of external software to create these systems (like OpenModellica Simulator), and then simulate in a Python environment.

DiaBloS provides a library that does not depend on external software for the simulation of dynamic systems, and provides the necessary tools to be able to create dynamic systems, simulate and obtain graphs of their behavior over time, without requiring major programming.

The package currently offers the following functionalities:

1. Interconnect functions to create more complex systems.
2. Integrate and derivate signals.
3. Produce step responses, such as ramp responses, for systems of ordinary difference (differential) equations.
4. Generate signals with Gaussian distribution noise.
5. Create feedback systems.
6. Plot signals, both traditionally and dynamically (in active simulation time).
7. Export signal data in .npz format.
8. Save and load created block systems in files formated as .dat.
9. Vector data handling.

10. Route multiple signals into a single output vector and vice versa (multiplexing).

1.1.2 Target audience for the DiaBloS package

DiaBloS is intended for mathematicians, physicists and practicing engineers. It is a tool designed to help students understand processes associated with modeling dynamic systems, observe their physical behavior over time, as well as develop solutions for control systems, visually, without the need to have a full understanding of how to code these systems first.

1.1.3 System requirements

This library requires Python 3.9.7 or later

1.1.4 Background information and Reference Material

Nowadays, more and more software tools based on visual diagrammatic programming are being used. These tools have the advantage of facilitating the user the implementation of complex system models, allowing to visualize the interactions between subcomponents of the system. Examples of these tools can be found from industry-adopted systems such as LabView from National Instruments or Simulink from Mathworks. There are also tools for the development of computational thinking and STEM skills that employ programming programs based on block diagrams such as Lego EV3. However, there are no open libraries for the development of this type of tools, nor is there a detailed exposition of the methodologies for processing block diagrams and executing the models they represent, since in general the existing software uses proprietary methodologies. For this reason, the contribution of the present work is the approach of the main algorithms for the processing of block diagrams and the development of an open software library for Python that is made available to the community to be used both as a computational tool and as a basis for the development of other graphical modeling systems based on block diagrams, as well as an educational tool that can be useful in courses of modeling and simulation of dynamic systems.

1.2 Getting Started

1.2.1 Python requirements

This library has been tested in Python 3.9.7.

Before downloading and/or installing this library, make sure you have the following packages:

```
- pygame (2.1.2 or later)
- numpy (1.22.3 or later)
- tqdm (4.64.0 or later)
- pyqtgraph (0.12.3 or later)
```

To install the packages you can use pip (recommended):

```
pip install pygame numpy matplotlib tk tqdm pyqtgraph
```

You will also need these packages, but they should come with Python 3.9.7 by default:

```
- os
- importlib
- json
- tkinter (tk)
- functools
- copy
- time
```

1.2.2 Installing DiaBloS

1. Download from repository
2. Unpack the .zip wherever you want

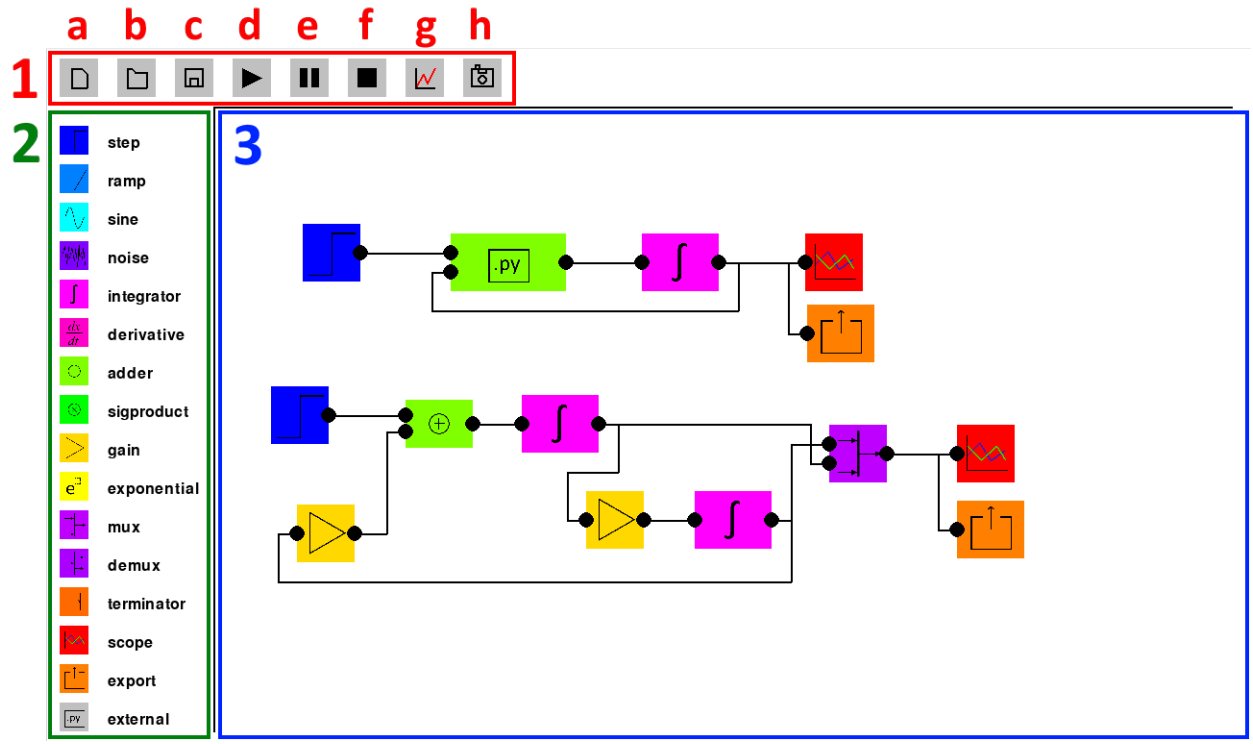
1.2.3 Loading DiaBloS

Run 'diablos_main.py'

1.3 Using DiaBloS: User Guide

1.3.1 Interface

After loading the package, a window similar like the following figure will show



ID	UI Element
1	Bar function
1.a	New button
1.b	Load button
1.c	Save button
1.d	Play simulation button
1.e	Pause simulation button
1.f	Stop simulation button
1.g	Plot graph button
1.h	Screen capture button
2	Blocks Menu
3	Canvas

1. **How to remove all elements in the canvas.**
   To remove all elements, press the NEW icon (1a).

2. **How to add blocks.**
   To add a block, drag and drop any block from the Block Menu (2) to the canvas (3).

3. **How to remove blocks.**
   To remove a block, drag and drop it outside the canvas. Also you can select it with LMB, then press the DELETE key.

4. **How to add lines.**
   To add a line, click on two block ports to generate a link between both.

   It's important to note that a link between ports only can be created if one port is labeled as input (ports located at the left side of a block) and the other is labeled as output (ports located at the left side of a block).

   Also, if an input port has already a link with another port you cannot create another link. However this restriction does not apply to output ports.

   Each port in the blocks has a unique identification. The input ports are always on the left side of the blocks, while the output ports are on the right side of the blocks. Between ports of the same type, they are differentiated according to their position from top to bottom starting from zero. e.g, a 3-input Adder block has inputs identified as $i = 0, 1, 2$ and a single output identified as $o = 0$.

5. **How to remove lines.**
   To remove a line select it with LMB, then press the DELETE key.

   When blocks are removed, the associated lines are also removed, in order to free connections that no longer make logical sense (input or output port does not exist).

6. **How to change color of the line.**
   A particular feature of the lines is that you can change their color. To do this, select the line and then press the UP_ARROW or DOWN_ARROW keys continuously until you find the color to choose.

7. **How to change parameters.**
   If the block contains editable function parameters, you can open a window to modify them by pressing RMB on the block.

   It is important to be careful to enter the parameters in the correct formats. These can be strings, boolean (as text), or floats (ints are converted to floats).

8. **How to change port numbers.**
   If a block allows changing the number of ports, a window can be opened with CTRL + RMB, with one or more entries to change the number of inputs and outputs, written as ints.

9. **How to load/save files.**
   The saving format of these files is .dat. It can be opened with any text editor to look and edit its data as wanted.

   To save a file, just click on the SAVE icon (1c), where a window will open giving the options of saving location and file name.

   To load a file, just click on the LOAD icon (1b), which will open a window giving the options to locate the file by folder and file name.

10. **How to run simulation.**
    To run the simulation, first press the PLAY icon (1d). A window will appear, to set the simulation time, the sampling time, as well as settings for the signal plotting: the size of the window in dynamic mode, and activate or deactivate the dynamic mode. Then press OK and the simulation will start and continue running until the sampling time is reached or stopped by pressing the STOP icon (1f).

    The process can be paused by pressing the PAUSE icon (1e). To restart it back, just press the PAUSE icon a second time.

11. **How to plot data.**
    To plot the curves of a simulation it is necessary to add Scope blocks and connect them to the output signals to be observed.

    The Scope block contains a single parameter called 'labels' which is used to name the signal(s) to be plotted. If this parameter is not changed, the observed signals will be named by default as 'Scope-<n>', where 'n' corresponds to the location of the variable within the input vector to the Scope block.

In addition, dynamic plotting (plotting the data while the simulation is running), can be enabled or disabled. To do this, when starting a simulation (by pressing the PLAY icon (1d)), there is an option that allows enabling this feature as another that allows changing the size of the moving window that will show the plotted values over time.

If the simulation is finished, the graph with all the data can be seen by pressing the PLOT icon (1g). If dynamic plotting has been performed, first close the first window with the resulting graph and then reopen it by pressing the PLOT button.

12. **How to export data.**
    To export data, the process is similar to plotting.

    First an EXPORT block must be added, which must be connected to the output of the block from which the signal is wanted to be saved.

    The labels can be renamed to identify each of the vectors. Otherwise they will be called by default as 'ExportData-<n>', where 'n' corresponds to the location of the variable within the input vector to the Export block.

13. **How to load user-made functions.**
    DiaBloS allows the loading of external functions, created by the user.

    To load these type of functions, an External block must be added, where the only parameter to modify is the name of the file, that contains the user-made function, located in the 'usermodels/' folder.

    If the upload is correct, the block will update its name at the bottom, the ports and the color in the canvas. If something went wrong, the program will indicate that the function name does not exist or something wrong was found during the process.

    After loading the file, the ports and parameters info will be loaded into the block, making the latter available for editing in the same way as the default program blocks. Also, for these type of blocks, there's an option to reset the parameter values back to the original ones.

    More details about how to create these types of functions are available in *Creating new functions* section from developer's guide.

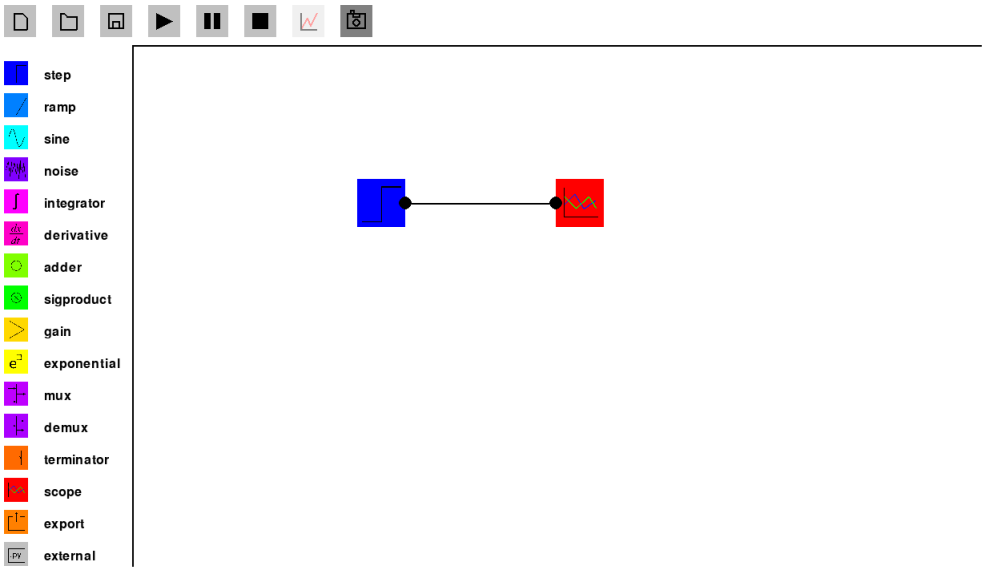
14. **How to take a capture of the canvas.**
    Press the CAPTURE icon (1h) to take a capture of the screen. These get saved in the 'captures/' folder.

15. **Some shortcuts**

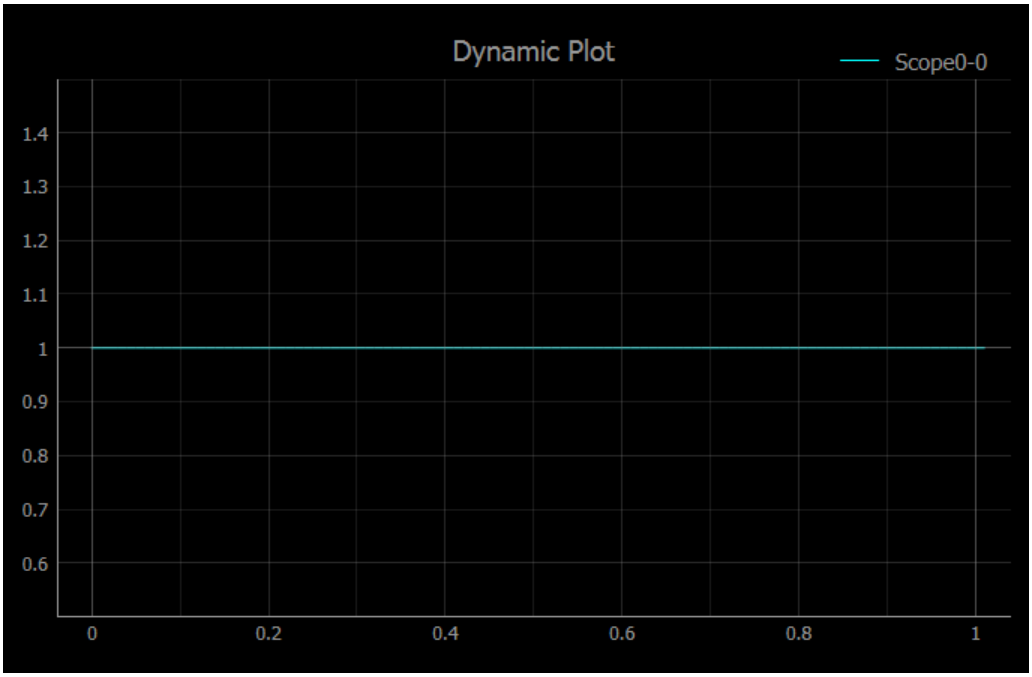
```
Ctrl + N: New
Ctrl + A: Load
Ctrl + S: Save
Ctrl + E: Play Simulation
Ctrl + P: Take Capture
```

1.3.2 First Experience

1. Load the interface.
2. Press the OPEN icon.
3. Go to examples/ and open basic_example.dat.
4. You will see something like the following picture:



1. Select the blue block (Step) and open the parameters' menu pressing RMB over the block.
2. Change the "value" parameter from "1.0" to "2.5" and change the "delay" parameter to "5.0" seconds, then press OK.
3. Select the red block (Scope) and open the parameters' menu pressing RMB over the block.
4. Change the "labels" parameter from "default" to "step", then press OK.
5. Press the PLAY icon to open the simulation pop-up window.
6. Change the "Simulation time" parameter to "10.0" (seconds).
7. Set "Dynamic Plot" as ON, then press OK.
8. Wait until the simulation is done.
9. Close the plot window.
10. Press the PLOT icon to open the plot window to observe the complete graph.
11. You will see something like the following picture:


Dynamic Plot
Scope0-0
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0
0.2
0.4
0.6
0.8
1

1.4 Using DiaBloS: Some Practical Examples

This package provides some examples, as a way to demonstrate some of the capabilities that this program has. These examples are contained in '.dat' files, located in the 'Examples' folder, and are executed as any file saved by the user.

1.4.1 Sine integration

Description

This example shows the process for integrating a sinusoidal signal, using the RK45 method and then comparing the result with the mathematically correct curve.

Demonstration

The math expression for the process is the following

$$y(t) = \int_0^t A\ \sin(\omega\, t + \phi_0) dt$$

calculating the integral rigorously, we arrive at the following expression:

$$y(t) = 1 + A\ \cos(\omega\, t + \phi_0)$$

rewriting $\cos(\theta)$ as $\sin(\theta + \pi/2)$:

$$y(t) = 1 + A\ \sin(\omega\, t + \phi_0 + \pi/2)$$

if $A = 1, \omega = 1$ y $\phi_0 = 0$, the resulting expression for $y(t)$ is:

$$y(t) = 1 + \sin(t + pi/2)$$

This is exemplified as the addition of a step of amplitude 1 and a sinusoidal starting at $\phi_0 = \pi/2$ at time $t_0 = 0$. Then an example comparing the integration process and the exact curve can be done.

Graph composition

Graph 1:

1) A Step up block with amplitude 1 and no delay.
2) A Sine block with an initial angle of $\pi/4$ to form $\cos(t)$.
3) An Adder block to form $y(t)$ defined above with the previous two blocks.
4) A Scope block to observe the result of the operation.

Graph 2:

1) A Sine block to form $\sin(t)$, the base function that will be integrated.
2) An Integrator block using the RK45 method and initial condition set in zero.
3) A Scope block to observe the result of the operation.

1.4.2 Vectorial integration

Description

This example shows a integration with the RK45 method, but the inputs and outputs are vectors instead of scalar values.

Demonstration

The Step block has support for vector outputs of the type:

```
1. Vectorial: [a, b, c, d].
2. Matrix: [[a, b], [c, d]]
3. 3D Matrix: [[[a, b], [c, d]], [[e, f], [g, h]]]
```

A graph is formed representing a simple feedback system, consisting of an integrator connected in feedback. The input values are defined by two vector sources which are added together. This value is used for a feedback system represented by the following transfer function:

$$y(t) = e^{-t} * u(t) \leftrightarrow \frac{Y(s)}{U(s)} = \frac{1}{s+1}$$

It should be noted that for this case, the initial conditions of the Integrator block must be of the same dimensions as its input. Since for this case it will be a vector of two elements, the initial conditions must be defined as $[0.0, 0.0]$, if you want to start at zero for both.

Graph Composition

1) A Step up block with amplitude $[1.5, 2.0]$ delayed in 5 seconds.
2) Another Step up block with amplitude $[1.0, 0.5]$ and no delay.
3) An Adder block to add the outputs of both blocks.
4) Another Adder block to subtract the output of the previous block with the output of the Integrator block.
5) An Integrator block using the RK45 method to obtain the integration of the Adder's output with initial conditions set in $[0.0, 0.0]$.
6) A Scope block to observe the result of the operation.

1.4.3 Gaussian noise

Description

This example shows a vectorial output (2 signals) with added noise.

Demonstration

The Noise block allows the creation of Gaussian noise for simulation effects. It only requires defining mu and $sigma$ and then adding the result to the target signal.

The graph to be shown is a system where the Step block's amplitude is $[5.0, -2.5]$, separates the vector into two independent signals, adding a Gaussian noise to each.

Graph Composition

1) A Step up block with amplitude $[5.0, -2.5]$ delayed in 2.5 seconds.
2) A Demultiplexer block to split the vector from the Step block and treat each element as an independent signal.
3) Two Noise blocks with $\mu = 0$ and $\sigma = 1$ to produce gaussian noise.

4) Two Gain block with gain of 0.5 to attenuate the signal amplitude from the noise blocks.

5) Two Adder blocks to add the noise ouput to each signal coming from the Demultiplexer.

6) A Scope block to observe the result of the operation.

Signal products

Description

This example shows element-wise products between vectors and a scalar signal and a vector.

Demonstration

The Sigproduct block is used to multiply the values of each iteration between different signals. Defining y_1 and y_2, as two output signals from two different blocks, the following signal multiplications can be observed:

1) Scalar with scalar: If $y_1 = a$ and $y_2 = b$ then $y_3 = a\,b$.

2) Scalar with vector: If $y_1 = a$ and $y_2 = [b, c]$ then $y_3 = [a\,b, a\,c]$.

3) Vector with vector: If $y_1 = [a, b]$ and $y_2 = [c, d]$ then $y_3 = [a, c, b, b, d]$.

An example graph is shown where the multiplication between three different sources is observed, based on the three types of multiplication described above.

Graph Composition

1) A Step up block with amplitude 5.0 delayed in 1 second.

2) Another Step up block with amplitude $[2.0, -3.0]$ with no delay.

3) A Step down block with amplitude $[0.75, 1.5]$ delayed in 2 seconds.

4) A Multiplexer block to append the Step blocks' outputs in one simple vector.

5) A Terminator block to finish the branch of the graph that will not be plotted.

6) Two Sigproduct blocks, one to multiply the output of the first and second Step blocks, and another to multiply the output of the second and third Step blocks.

7) Two Scope blocks to observe the results of the operations.

1.4.4 Export data

Description

This example shows how signal data can be exported in '.npz' format.

Demonstration

The Export block is used to save data of signals created during the simulation. It is enough to add this block and define the labels within the settings of this block.

In particular, the function for exporting data consists of two parts:

1) Data acquisition: During the simulation, the block will accumulate data in ordered vectors, each associated with a label. associated to a label. If labels have not been previously defined, or if they are not enough to cover all the vectors to be created, the block will vectors to be created, the system adds default names to complete the list. A matrix is then created a matrix is then created that will append the values added by each simulation loop.

2) Data conversion: At the end of the simulation, all the vectors of the Export blocks are taken (if there are more than one), and all the data is assembled. more than one), and a larger matrix is assembled, which will be exported as .npz by means of the numpy library.

After completing a simulation process, a .npz file will be created and found inside the 'saves' folder, with the same name as the savefile (by default 'data.npz'). Note that this example only exports the files. Being able to read them can be done with Python, Excel or similar.

Graph Composition

1) A Step up block with amplitude 1 and no delay.
2) A Sine block with an initial angle of $\pi/4$ to form $\cos(t)$.
3) An Adder block to form $1 + \cos(t)$ with the previous two blocks.
4) A Multiplexer block to produce a 2D vector with the Step block's output as first element and $1 + \cos(t)$ (Adder block's output) as second element.
5) An Export block to save the data from the Multiplexer block and then export it as a file in .npz format.

External source

Description
This example shows an external function implemented as a source block.

Demonstration
The Block block associates user-defined functions to give more options for graph simulation.

The only parameter needed to modify is the function name (and `.py` file) located in the `usermodels` folder. After loading this, the block acquires the data defined in the file to change, number of inputs, outputs, block type and color.

For this case, it is important to define the block inputs as 0 and the block type as 0 (source).

Details on how to create such functions can be found in *Usermodel functions*.

Graph Composition

1) An External block linked to the external usermodel function `my_function_src`.
2) Two Scope blocks to observe the outputs of the External block.

1.4.5 External Z-process

Description
This example shows an external function implemented as a Z-process block.

Demonstration
The Block block associates user-defined functions to give more options for graph simulation.

The only parameter needed to modify is the function name (and `.py` file) located in the `usermodels` folder. After loading this, the block acquires the data defined in the file to change, number of inputs, outputs, block type and color.

For this case, it is important to define the block type as 2 (z-process).

Details on how to create such functions can be found in *Usermodel functions*.

Graph Composition

1) A Step up block with amplitude 1 and no delay.
2) An External block linked to the external usermodel function `my_function_pcs`.
3) A Scope block to observe the result of the operation.

1.4.6 External integrator (N-process)

Description

This example shows an external function implemented as a N-process block. In this case, an integrator using the same RK45 method already implemented in the Integrator block.

The Block block associates user-defined functions to give more options for graph simulation.

The only parameter needed to modify is the function name (and `.py` file) located in the `usermodels` folder. After loading this, the block acquires the data defined in the file to change, number of inputs, outputs, block type and color.

For this case, it is important to define the block type as 1 (n-process).

Details on how to create such functions can be found in *Usermodel functions*, details on how the RK45 integration method works, see *Graph simulation algorithm*.

Graph Composition

1) A Step up block with amplitude 1 and no delay.
2) An External block linked to the external usermodel function `external_rk45`.
3) A Scope block to observe the result of the operation.

1.4.7 External derivator (Z-process)

Description

This example shows an external function implemented as a Z-process block. In this case a variable step-size derivator (direct feedthrough function).

Demonstration

The Block block associates user-defined functions to give more options for graph simulation.

The only parameter needed to modify is the function name (and `.py` file) located in the `usermodels` folder. After loading this, the block acquires the data defined in the file to change, number of inputs, outputs, block type and color.

For this case, it is important to define the block type as 2 (z-process).

Details on how to create such functions can be found in *Usermodel functions*.

Graph Composition

1) A Ramp block with slope 1 and no delay.
2) An External block linked to the external usermodel function `external_derivative`.
3) A Scope block to observe the result of the operation.

1.4.8 Convergent ODE system

Description

This example shows the same convergent ODE system implemented in three different ways.

Demonstration

A particular ordinary differential equation is used as an example:

$$\ddot{y} + 0.4\,\dot{y} + y = u$$

if $x_1 = y$ and $x_2 = \dot{y}$ this ODE can be represented in vector form as:

$$X' = f(X, U)$$
$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_1 - 0.4\,x_2 + u \end{bmatrix}$$

and in the same time, it can be converted to a matrix system of the type $X' = A\,X + B\,U$.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -0.4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

So three instances of this problem are created to simulate:

1) Using an external function, where value U and vector $X = [x_1, x_2]$ are received, to deliver $\dot{X} = f(X, U)$.
2) Using gain and adder blocks to form the matrix notation ($X' = A, X + B, U$) before integrating it.
3) Using the non-vector system definition, first by calculating $ddoty$, then integrate it to find $doty$ and then integrate once again to find y.

Graph Composition

Graph 1:

1) A Step up block with amplitude 1 and no delay.
2) An External block linked to the external user model function `ode_system_conv`.
3) An Integrator block using the RK45 method to obtain the integration of the previous operation's result.
4) A Scope block to observe the output of the Integrator block.
5) An Export block to save the data from the Integrator block and then export it as a file in .npz format.

Graph 2:

1) A Step up block with amplitude 1 and no delay.
2) A Gain block to multiply the output of the Step block with the vector $B = [0.0, 1.0]$ producing BU.
3) A Gain block to multiply the output vector of the Integrator block with the matrix $A = [[0.0, 1.0], [-1.0, -0.4]]$ producing AX.
4) An Adder block to add the output of both Gain blocks, producing $AX + BU$.
5) An Integrator block using the RK45 method to obtain X from the Adder block's output, and initial conditions set in $[0.0, 0.0]$.
6) A Scope block to observe the output of the Integrator block.
7) An Export block to save the data from the Integrator block and then export it as a file in .npz format.

Graph 3:

1) A Step up block with amplitude 1 and no delay.
2) An Integrator block that integrates the value of the Adder block's output to obtain x_2.
3) A Gain block to multiply x_2 by -0.4 and be used in the Adder block as future input.

4) Another Integrator block that integrates x_2 to get x_1.

5) Another Gain block used to multiply x_1 by -1 and be used in the Adder block as future input.

6) An Adder block that adds the result of both Gain blocks and the Step block's output to get $\dot{x}_2$.

7) A Multiplexer block to produce a vector with the output values of the Integrator blocks.

8) A Scope block to observe the output of the Multiplexer block.

9) An Export block to save the data from the Multiplexer block and then export it as a file in .npz format.

1.4.9 Critical ODE system

Description

This example shows the same critical ODE system implemented in three different ways, compared to the exact curve.

Demonstration

A particular ordinary differential equation system is used vector form as an example:

$$X' = f(X, U)$$
$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_1 + u \end{bmatrix}$$

and in the same time, it can be converted to a matrix system of the type $X' = A\,X + B\,U$.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

Knowing that A is the rotation matrix when $\theta = 90°$, and setting $u = 1$, the equations can be rewritten as:

$$x_1 = 1 - \cos(t)$$
$$x_2 = \sin(t)$$

So four instances of this problem are created to simulate:

1) Using an external function as source, the exact value of the curves.

2) Using an external function, where value U and vector $X = [x_1, x_2]$ are received, to deliver $\dot{X} = f(X, U)$.

3) Using gain and adder blocks to form the matrix notation ($X' = A, X + B, U$) before integrating it.

4) Using the non-vector system definition, first by calculating $\dot{x}_2$, then integrate it to find $x_2 = \dot{x}_1$ and then integrate once again to find x_1.

Graph Composition

Graph 1:

1) An External block linked to the external user model function `ode_exact_crit`.

2) 1) A Scope block to observe the output of the External block.

Graph 2:

1) A Step up block with amplitude 1 and no delay.
2) An External block linked to the external user model function `ode_system_crit`.
3) An Integrator block using the RK45 method to obtain the integration of the previous operation's result.
4) A Scope block to observe the output of the Integrator block.
5) An Export block to save the data from the Integrator block and then export it as a file in .npz format.

Graph 3:

1) A Step up block with amplitude 1 and no delay.
2) A Gain block to multiply the output of the Step block with the vector $B = [0.0, 1.0]$ producing BU.
3) A Gain block to multiply the output vector of the Integrator block with the matrix $A = [[0.0, 1.0], [-1.0, -0.4]]$ producing AX.
4) An Adder block to add the output of both Gain blocks, producing $AX + BU$.
5) An Integrator block using the RK45 method to obtain X from the Adder block's output, and initial conditions set in $[0.0, 0.0]$.
6) A Scope block to observe the output of the Integrator block.
7) An Export block to save the data from the Integrator block and then export it as a file in .npz format.

Graph 4:

1) A Step up block with amplitude 1 and no delay.
2) An Integrator block that integrates the value of the Adder block's output to obtain x_2.
3) A Gain block to multiply x_2 by -0.4 and be used in the Adder block as future input.
4) Another Integrator block that integrates x_2 to get x_1.
5) Another Gain block used to multiply x_1 by -1 and be used in the Adder block as future input.
6) An Adder block that adds the result of both Gain blocks and the Step block's output to get $\dot{x}_2$.
7) A Multiplexer block to produce a vector with the output values of the Integrator blocks.
8) A Scope block to observe the output of the Multiplexer block.
9) An Export block to save the data from the Multiplexer block and then export it as a file in .npz format.

1.4.10 Watertank control

Description

This example shows the classic watertank control problem, trying to stabilize the height of the water using a PI control.

Demonstration

According to Bernoulli's principle the equation representing the height of water in a pond is:

$$\dot{h}(t) = \frac{1}{A_e}u - \frac{A_s}{A_e}\sqrt{2g \cdot h(t)}$$

where, $A_e = 3.14[m]$ is the base area for the pond, $A_s = 3.14 \cdot 10^{-4}[m]$ is the drainage section area and $g = 9.81[m/s^2]$ is the gravitational acceleration.

To control the system, a PI controller is used to adjust the height of the water, under a reference h_{ref}:

$$u(t) = k_p\left(h_{ref} - h(t)\right) + k_i \int_0^t (h_{ref} - h(t))dt$$

The controller requires only a couple of user-adjustable constants, which for this case are: $k_p = 10.0$ and $k_i = 55.5$.

However, since the PI controller also contains an integration stage, this formula can be rewritten, while retaining its validity in the system, as follows:

$$\dot{u} = -k_p\,\dot{h}(t) + k_i\left(h_{ref} - h(t)\right)$$

Then, it is possible to rewrite the pond water height control model as a system of two equations:

$$\begin{bmatrix}\dot{h}(t)\\ \dot{u}(t)\end{bmatrix} = \begin{bmatrix}1/A_e \cdot u(t) - A_s/A_e \cdot \sqrt{2g \cdot h(t)}\\ -k_p \cdot \dot{h}(t) + k_i \cdot (r - h(t))\end{bmatrix}$$

The system of equations is modeled as a block diagram using primarily basic blocks. In addition, two External blocks are added to perform functions that are not available by default, as a way to also show implementation examples for user-defined blocks; one to perform the square root operation, while the other to saturate the input value to prevent the water level from going beyond the physical limits that are set.

Also, the system of two equations is modeled as one External block connected to a Integrator block, to compare the performance of both methods.

The reference is taken as $h_{ref1} = 1.25[m]$ during the first 5.0 seconds, and $h_{ref2} = 0.5[m]$ in the following $5.0[s]$, simulating in total 10.0 seconds at a sampling rate of $0.01[s]$.

Graph composition

Shared blocks:

1) A Step up block with amplitude 1.25.
2) Another Step up block with amplitude 0.75 delayed in 5 seconds.
3) An Adder block to subtract the output of both blocks to have an amplitude of 0.5 after 5 seconds.

Graph 1:

1) An Adder block to subtract the reference (from the previous Adder block) with the actual calculated value of h.
2) A Gain block to multiply the previously calculated subtraction by k_i.

3) A Gain block to multiply $\dot{h}$ by k_p.

4) An Adder block to subtract the results of the previous two Gain blocks to obtain $\dot{u}$.

5) An Integrator block that integrates $\dot{u}$ to obtain u.

6) A Gain block to multiply u by $1/A_e$.

7) An External block linked to the external usermodel function `sqrt_pcs`, to calculate the square root.

8) An Gain block to multiply the result of `sqrt_pcs` with A_s/A_e.

9) An Adder block to subtract u/A_e and the result of the previous Gain to obtain $\dot{h}$.

10) An Integrator block that integrates $\dot{h}$ to obtain h.

11) An External block linked to the external usermodel function `sat_pcs`, to restrict the value of h between 0 and 2.

12) An Export block to save the data from h and then export it as a file in .npz format.

13) A Scope block to observe the result of h in time.

Graph 2:

1) An External block linked to the external usermodel function `watertank_code`.

2) An Integrator block using the RK45 method to obtain the integration of the previous operation's result.

3) A Demux block to split the output vector in h and u to plot only the former element.

4) A Terminator block to finish the branch of u.

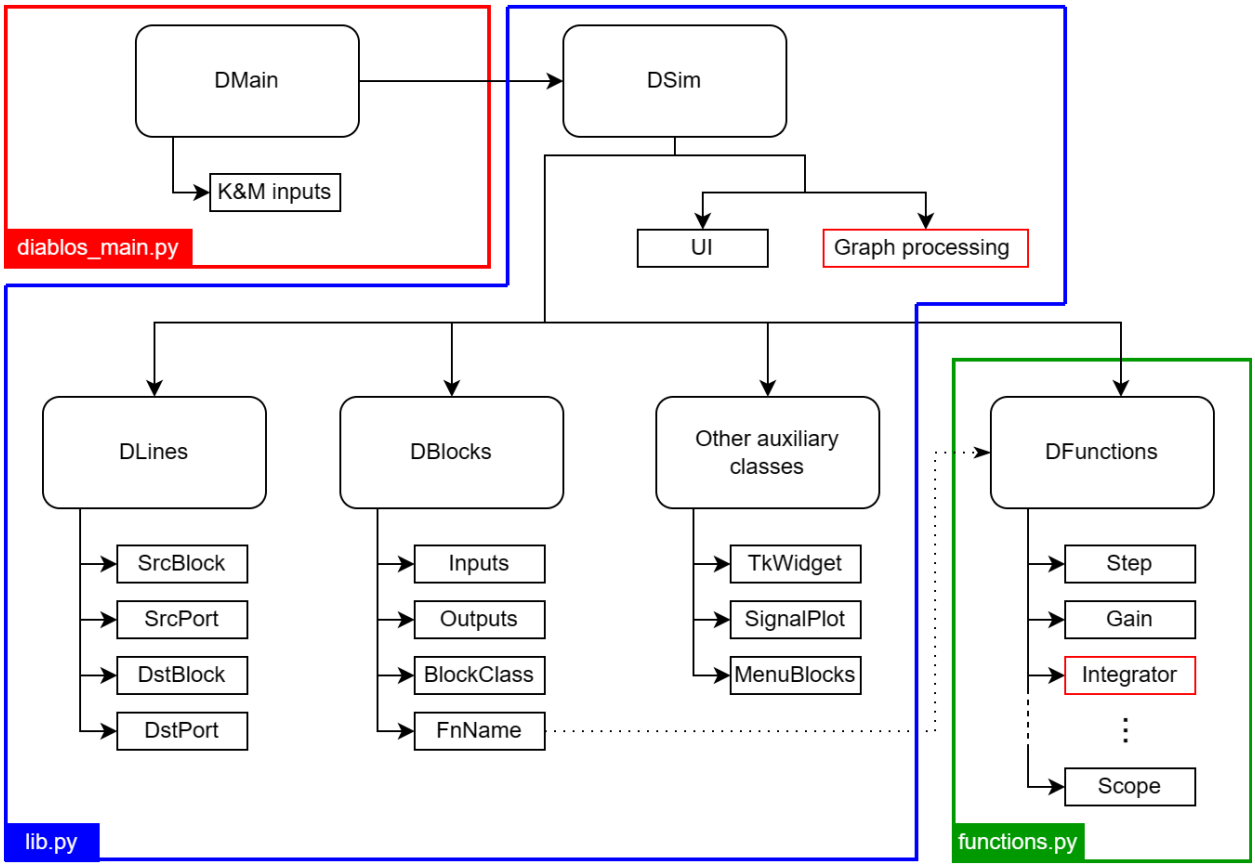
5) An Export block to save the data from h and then export it as a file in .npz format.

6) A Scope block to observe the result of h in time.

1.5 Using DiaBloS: Developer Guide

1.5.1 Software hierarchy

A hierarchical diagram of the tool's main classes and functions is presented in the following figure, and a description of its most important classes and functions of this software library is presented below:



- **DMain:**
  Executes the main whileloop of the interface from the moment the program starts. It calls the DSim class and handles the data input from the user.

- **DSim:**
  Class that controls and executes the main functions of the interface, such as creating block and link objects, simulating the block diagram, opening and saving files, and loading the graphical representations of the elements in the interface.

- **DBlocks:**
  Class that defines each node or block of the network as an object with a finite number of inputs and outputs, color, type of block (source, zprocess, nprocess, terminal), associated function, among others. In addition, as an object, it contains the variables associated to the executable function, independently from other blocks of the same type.

- **DLine:**
  Class that defines the link between blocks as an object with start and end, both identified as a block-port pair. It also controls the segmentation of a line for graphical effects, such as changing its color in the interface.

- **DFunctions:**
  Class that contains as methods all the default functions associated to the blocks created in the system. These

functions have no memory, since this information is contained in each block created under DBlocks, they only calculate data and provide a result.

- **Auxiliary classes (`TkWidget()`, `SignalPlot()`, `MenuBlocks()`):**
  These are classes with specific functionalities to support the use of the tool by the user. i.e.: Plot curves, change parameters, create blocks in the interface.

The creation of blocks and functions as independent elements is supported by the idea of facilitating the production process. In most cases, a block only requires the name of the function to be associated and the main parameters to obtain a functional node in the diagram, making it unique from the rest. This also allows the creation of external functions to those already available in the library, allowing the user to create more complex processes, simplifying the elaboration of diagrams, if required.

1.5.2 Graph simulation algorithm

The algorithm to simulate a block diagram, will be posted soon.

1.5.3 Data management

Communication between blocks

Data sent and received by blocks, has to be packaged inside a dictionary. This dictionary only has integer numbers as keys, requiring as many as the number of output ports a block has. e.g: A function linked with a block with two outputs would likely return a dictionary like the following:

```
return {0: np.array(data0), 1: np.array([data1,data2])}
```

The values for each key only supports numpy array variables. Although, there is no actual restriction for the format as long as the block receiving this data can process it.

There is one exception for keys. The key `'E'` is used to indicate an error happened while executing a function and the simulation must be stopped. More details about using this particular key are presented in *Creating new functions*.

Vector management

Currently DiaBloS supports vectors up to 3 D.O.F.:

```
1. Vector form: [a, b, c, d]
2. Matrix form: [[a, b], [c, d]]
3. 3D-matrix form: [[[a, b], [c, d]], [[e, f], [g, h]]]
```

The process which a string is converted to `numpy.ndarray` format is explained below:

- The `TkWidget()` class is responsible for creating windows to edit parameter values for each block. The `string_to_vector()` method converts strings identified as vectors, which start and end with `[` and `]`, respectively.
- A copy of the original string is made by removing the numeric values and spaces, to count the number of square brackets and thus determine the dimensions of the vector/matrix.
- A second copy of the original string is made, this time removing the square brackets and spaces, creating a single vector that is resized with the values resulting from the previous process, as well as checking that all values within the vector are numeric.

- If the number of elements in the vector does not correspond to the dimensions of the vector/matrix, an error is indicated and a `NaN` is returned, retaining its previous value.

1.5.4 Usermodel functions

DiaBloS supports the use of usermade functions, which can be accessed in the block diagram with the External block. Usermodel functions files are composed of two functions:

1. **Execution function:**
   Function executed when a block diagram is simulated. The equivalent to the functions implemented as methods of the DFunctions class.

   It is important that the name of this function and the `.py` file are the same, otherwise the function will not be associated to the External block.

2. **Initialization function:**
   Function containing parameters to update the External block associated with the execution function. It only contains two dictionaries:

   1. `io_data`: Dictionary containing port and process type data. I.e.: Number of input and output ports, block type, block color.
   2. `params`: Dictionary containing the default parameters needed for the execution function. I.e.: `_init_start_`.

- If auxiliary functions to the execution function are needed, they can be added to the same file.
- If a library is required to perform a process, simply import it from the file at the beginning.

Creating new functions

A generic layout for a user-made function named `my_function()` is presented below:

- Execution function `my_function()`:

```
# filename: my_function.py
"""import libraries"""

def my_function(time, inputs, params):
    """function code, either source, N_process, Z_process or drain"""
    if params['_init_start_']:
        """commands that must be executed in first execution loop only"""

        params['_init_start_'] = False
    """commands that always must be executed"""

    return {0: variable_output0, 1: variable_output1, ...} # return values that␣
↪must be sent to other blocks.
```

- Initialization function `_init_()`:

```
def _init_():
    io_data = { # parameters for the block associated with the function
        'inputs': input_value,
        'outputs': output_value,
        'b_type': block_type_value, #0: source, #1: N_process, #2: Z_process, #3:␣
```

(continues on next page)

(continued from previous page)

```
↪drain
        'color': color_string_or_rgb_triplet #(r,g,b), 'red'
    }
    params = { # default parameters defined for the function
        'parameter0': value,
        '_init_start_': True
    }
    return io_data, params
```

- Use of `except` to display messages in the command console for errors:

```
...
try:
    """normal process"""
except:
    """Commands to do if there's an error in the process. i.e: printing "ERROR" in␣
↪the terminal."""
    return {'E': True} # error happened, returns a flag to stop the block diagram's␣
↪simulation.
...
```

There are templates available in `usermodels/templates` folder for each type of block to simulate.

Tips for testing new functions

1. It is recommended to implement this function as an external-function type first, then add it as a method of the Functions class.
2. First define inputs, outputs, block type and block color in the external function file `_init_()` and implement the most simplified version of the function to add.
3. After that, create a simple graph diagram to test the new block. i.e: A Step block, connected to the external block (where the new function is implemented), connected to a Scope block.
4. If the system doesn't fail execution, add new elements to the external function being aware of not breaking the simulation stability.
5. When everything is ok, add the new finished function to the Functions class and create a new MenuBlock in `InitSim.menu_blocks_init()`, using the parameters already defined in the external function `_init_()`, defining block size and if the function allows change of inputs and/or outputs.
6. Test again the function in the simulation, this time replacing the External Block with the corresponding to the new implemented function.

1.6 Function Reference

Below is a list of the functions of this package. It should be noted that several of these functions are only auxiliary, focused on handling a particular problem in the process.

diablos_main	diablos_main.py - Module to run the simulator interface
lib.lib	lib.py - Contains all the core functions and classes for the simulation and execution of the graphs.
lib.functions	functions.py - Contains all the functions associated with the blocks in the simulation

1.6.1 diablos_main

diablos_main.py - Module to run the simulator interface

Functions

dmain()	Function that manages keyboard and mouse inputs, block positions and line connections.

diablos_main.**dmain**()

Function that manages keyboard and mouse inputs, block positions and line connections. The user-interface.

1.6.2 lib.lib

lib.py - Contains all the core functions and classes for the simulation and execution of the graphs.

Classes

Button(name, coords[, active])	Class to create and show buttons in the user interface.
DBlock(block_fn, sid, coords, color[, ...])	Class to initialize, mantain and modify function blocks.
DLine(sid, srcblock, srcport, dstblock, ...)	Class to initialize and maintain lines that connect blocks.
DSim()	Class that manages the simulation interface and main functions.
MenuBlocks(block_fn, fn_name, io_params, ...)	Class to create and show basic blocks used as a mark to generate functional blocks in the user interface.
SignalPlot(dt[, labels, xrange])	Class that manages the display of dynamic plots through the simulation.
TkWidget(name, params[, external])	Class used to create popup windows for changing data, like ports and parameters.

class lib.lib.**Button**(*name*, *coords*, *active=True*)

Class to create and show buttons in the user interface.

Parameters

- **name** (*str*) – Name of the variable associated to the button.

- **coords** (*list*) – Coordinates of the button in the canvas.
- **active** (*bool*) – Boolean that indicates the state of the function associated to the button.

draw_button(*zone*)

Purpose
Draws the button.

class lib.lib.**DBlock**(*block_fn*, *sid*, *coords*, *color*, *in_ports=1*, *out_ports=1*, *b_type=2*, *io_edit=True*, *fn_name='block'*, *params={}*, *external=False*)

Class to initialize, mantain and modify function blocks.

Parameters

- **block_fn** (*str*) – Block name, defined according to the available blocks created in DSim.
- **sid** (*int*) – Unique identification for the created block.
- **coords** (*list*) – List with tuples that contain the location and size of the block in the canvas.
- **color** (*str/triplet*) – String or triplet that defines the color of the block in the canvas.
- **in_ports** (*int*) – Number of inputs for the block.
- **out_ports** (*int*) – Number of outputs for the block.
- **b_type** (*int*) – Variable for block type identification (0: source, 1: N_process, 2: Z_process, 3: drain).
- **io_edit** (*str*) – Variable that defines if a block can change the number of its input ports and/or output ports.
- **fn_name** (*str*) – Function name, function associated to the block. That function defined in the Functions class.
- **params** (*dict*) – Dictionary with function-related parameters.
- **external** (*bool*) – Parameter that set a block with an external function (not defined in Functions class).

change_params()

Purpose
Generates a pop-up window to change modifiable parameters only.

Description
Through the use of the TkWidget class, a pop-up is created to modify parameters associated to the blocks. It should be noted that the only parameters that can be modified are those defined at the beginning (during the definition of the block in 'DSim.menu_blocks_init'), as well as those that do not start with '_' underscore. The function separates the parameters, they are shown to the user, returned to the system and all are put back together at the end.

change_port_numbers()

Purpose
Generates a pop-up window for the user to change number of input and/or output ports for the block.

Description
Through the use of the TkWidget class, a pop-up window is created containing 2 parameters that can be changed by the user: inputs and outputs. After loading them, the 'update_Block' function is executed to adjust the size of the block and the position of its ports. It should be

noted that whether the number of inputs or outputs can be changed depends on the type of block, which is differentiated within the function.

`draw_Block`(*zone*)

Purpose
Draws block and its ports.

Parameters
zone – Pygame's layer where the figure is drawn.

`draw_selected`(*zone*)

Purpose
Draws the black line indicating that the block is selected.

Parameters
zone – Pygame's layer where the figure is drawn.

`load_external_data`(*params_reset=False*)

Purpose
Loads initialization data of a function located in a external python file.

Description
Through the use of the importlib library, a .py file is imported from the 'usermodels' folder, from where the function parameters and the block parameters (input, output, block_type) are extracted, importing them to the existing block, modifying its qualities if necessary.

`loading_params`(*new_params*)

Purpose
Loads parameters from a dictionary list, converting lists in array vectors.

Parameters
new_params – Dictionary with parameters for block.

`port_collision`(*m_coords*)

Purpose
Checks if a point collides with one of the ports of a block. Returns a tuple with the port type and port id.

Parameters
m_coords (*tuple*) – Coordinates of mouse input.

`reload_external_data`()

Purpose
Reloads the external function parameters.

`relocate_Block`(*new_coords*)

Purpose
Relocates port.

Parameters
m_coords (*tuple*) – New coordinates for the block (left, top).

`resize_Block`(*new_coords*)

Purpose
Resizes block.

Parameters
m_coords (*tuple*) – New parameters for block (width, height).

saving_params()

Purpose
Saves parameters only defined at initialization.

update_Block()

Purpose
Updates location and size of the block, including its ports.

Description
This function handles what is necessary to display the block on the screen. It draws the square of the block, draws the ports and places them depending on their quantity, and even extends or reduces the size of the block if the number of ports is not enough or too many. This function always acts when the block is updated, being much more important when it changes location.

class lib.lib.**DLine**(*sid*, *srcblock*, *srcport*, *dstblock*, *dstport*, *points*, *cptr=0*)

Class to initialize and maintain lines that connect blocks.

Parameters

- **sid** (*int*) – Unique identification for the created line.
- **srcblock** (*int*) – Block from where the line starts.
- **srcport** (*int*) – Port of the block where the line starts.
- **dstblock** (*int*) – Block to where the line ends.
- **dstport** (*int*) – Port of the block where the line ends.
- **points** (*list*) – List of tuples that defines the vertex of the trajectory of the line (if it's not a straight line).
- **cptr** (*int*) – Variable used as a pointer to assign a color to the line. It depends on the 'colors' list defined in DSim.

change_color(*ptr*)

Purpose
Pointer indicating which color is chosen from the color list defined in DSim.

Parameters
ptr (*int*) – Value that adds or subtracts 1 depending of the user's input.

collision(*m_coords*)

Purpose
Checks if there is collision between a point and the line.

draw_line(*zone*)

Purpose
Draws line.

Parameters
zone – Pygame's layer where the figure is drawn.

`trajectory`(*points*)

Purpose
Generates segments to display a connection between blocks linked by a line.

Parameters
points (*list*) – List with coordinates for each vertex of the line group.

`update_line`(*block_list*)

Purpose
Updates line from size and location of blocks.

Description
The function searches in the canvas for the location of the input and output ports to which it is connected, then produces a new trajectory using the 'trajectory' function.

`class` lib.lib.**DSim**

Class that manages the simulation interface and main functions.

Parameters

- **SCREEN_WIDTH** (*int*) – The width of the window
- **SCREEN_HEIGHT** (*int*) – The height of the window
- **canvas_top_limit** (*int*) – Top limit where blocks and lines must be drawn.
- **canvas_left_limit** (*int*) – Left limit where blocks and lines must be drawn.
- **colors** (*dict*) – List of predefined colors for elements that show in the canvas.
- **fps** (*int*) – Base frames per seconds for pygame's loop.
- **l_width** (*int*) – Width of the line when a block or a line is selected.
- **ls_width** (*int*) – Space between a selected block and the line that indicates the former is selected.
- **filename** (*str*) – Name of the file that was recently loaded. By default is 'data.dat'.
- **sim_time** (*float*) – Simulation time for graph execution.
- **sim_dt** (*float*) – Simulation sampling time for graph execution.
- **plot_trange** (*int*) – Width in number of elements that must be shown when a graph is getting executed with dynamic plot enabled.

`add_block`(*block*, *m_pos=(0, 0)*)

Purpose
Function that adds a block to the interface, with a unique ID.

Description
From a visible list of MenuBlocks objects, a complete Block instance is created, which is available for editing its parameters or connecting to other blocks.

Parameters

- **block** (*BaseBlock class*) – Base-block containing the base parameters for each type of block.
- **m_pos** (*tuple*) – Coordinates (x, y) to locate the upper left corner of the future block.

Bugs
Under a wrongly configured MenuBlock, the resulting block may not have the correct qualities or parameters.

`add_line`(*srcData*, *dstData*)

Purpose
Function that adds a line to the interface, with a unique ID.

Description
Based on the existence of one or more blocks, this function creates a line between the last selected ports.

Parameters

- **srcData** (*triplet*) – Triplet containing 'block name', 'port number', 'port coordinates' of an output port (starting point for the line).
- **dstData** (*triplet*) – Triplet containing 'block name', 'port number', 'port coordinates' of an input port (finish point for the line).

`check_diagram_integrity`()

Purpose
Checks if the graph diagram doesn't have blocks with ports unconnected before the simulation execution.

Description
This function is only used to check that the network is properly connected. All ports must be connected without exception. In case something is disconnected, a warning is printed indicating where the problem is and returns to the main function indicating that it cannot continue.

Returns
0 if there are no errors, 1 if there are errors.

Return type
int

`check_global_list`()

Purpose
Checks if there are no blocks of a graph left unexecuted.

`check_line_block`(*line*, *b_del_list*)

Purpose
Checks if a line is connected to one or more removed blocks.

Parameters

- **line** – Line object.
- **b_del_list** – List of recently removed blocks.

`check_line_port`(*line*, *block*)

Purpose
Checks if there are lines left from a removed port (associated to a block).

Parameters

- **line** – Line object.

- **block** – Block object.

children_recognition(*block_name*, *children_list*)

Purpose
For a block, checks all the blocks that are connected to its outputs and sends a list with them.

Parameters

- **block_name** (*str*) – Block object name id.
- **children_list** (*list*) – List of dictionaries with blocks data that require the output of block 'block_name'.

clear_all()

Purpose
Cleans the screen from all blocks, lines and some main variables.

count_computed_global_list()

Purpose
Counts the number of already computed blocks of a graph.

count_rk45_ints()

Purpose
Checks all integrators and looks if there's at least one that use 'RK45' as integration method.

display_blocks(*zone*)

Purpose
Draws blocks defined in the main list on the screen.

Parameters
zone – A layer in a pygame canvas where the figure is drawn.

display_buttons(*zone*)

Purpose
Displays all the buttons on the screen.

Parameters
zone – Pygame's layer where the figure is drawn.

display_lines(*zone*)

Purpose
Draws lines connecting blocks in the screen.

Parameters
zone – Pygame's layer where the figure is drawn.

display_menu_blocks(*zone*)

Purpose
Draws MenuBlocks objects in the screen.

Parameters
zone – Pygame's layer where the figure is drawn.

`dynamic_pyqtPlotScope`(*step*)

Purpose
Plots the data saved in Scope blocks dynamically with pyqtgraph.

Description
This function is executed while the simulation is running, starting after all the blocks were executed in the first loop. It looks for Scope blocks, from which takes their 'labels' parameter and initializes a SignalPlot class object that uses pyqtgraph to show a graph. Then for each loop completed, it calls those Scope blocks again to get their vectors and update the graph with the new information.

`execution_failed`()

Purpose
If an error is found while executing the graph, this function stops all the processes and resets values to the state before execution.

`execution_init`()

Purpose
Initializes the graph execution.

Description
This is the first stage of the graph simulation, where variables and vectors are initialized, as well as testing to verify that everything is working properly. A previous autosave is done, as well as a block connection check and possible algebraic loops. If everything goes well, we continue with the loop stage.

`execution_init_time`()

Purpose
Creates a pop-up window to ask for graph simulation setup values.

Description
The first step in order to be able to perform a network simulation, is to have the execution data. These are mainly simulation time and sampling period, but we also ask for variables needed for the graphs.

`execution_loop`()

Purpose
Continues with the execution sequence in loop until time runs out or an special event stops it.

Description
This is the second stage of the network simulation. Here the reading of the complete graph will be done cyclically until the time is up, the user indicates that it is finished (by pressing Stop) or simply until one of the blocks gives error. At the end, the data saved in blocks like 'Scope' and 'External_data', will be exported to other libraries to perform their functions.

`export_data`()

Purpose
Exports the data saved in Export blocks in .npz format.

Description
This function is executed after the simulation has finished or stopped. It looks for export blocks, which have some vectors saved with signal outputs from previous blocks. Then it merge all vectors in one big matrix, which is exported with the time vector to a .npz file, formatted in a way it is ready for graph libraries.

`get_max_hierarchy()`

Purpose
Finds in the global execution list the max value in hierarchy.

`get_neighbors(block_name)`

Purpose
Finds all the connected blocks to "block_name".

Parameters
block_name (*str*) – Block object name id.

`get_outputs(block_name)`

Purpose
Finds all the blocks that need a "block_name" result as input.

Parameters
block_name (*str*) – Block object name id.

`main_buttons_init()`

Purpose
Creates a button list with all the basic functions available

`menu_blocks_init()`

Purpose
Function that initializes all types of blocks available in the menu.

Description
From the MenuBlocks class, base blocks are generated for the functions already defined in lib.functions.py. Then they are accumulated in a list so that they are available in the interface menu.

`open()`

Purpose
Loads blocks, lines and other data from a .dat.

Description
Starting from the .dat file, the data saved in the dictionaries are unpacked, updating the data in DSim, creating new blocks and lines, leaving the canvas and the configurations as they were saved before.

Notes
The name of the loaded file is saved in the system, in order to facilitate the saving of data in it (overwriting it).

`plot_again()`

Purpose
Plots the data saved in Scope blocks without needing to execute the simulation again.

`port_availability(dst_line)`

Purpose
Checks if an input port is free to get connected with a line to another port.

Parameters
dst_line (*str*) – The name of a Line object.

`pyqtPlotScope()`

Purpose
Plots the data saved in Scope blocks using pyqtgraph.

Description
This function is executed while the simulation has stopped. It looks for Scope blocks, from which takes their 'vec_labels' parameter to get the labels of each vector and the 'vector' parameter containing the vector (or matrix if the input for the Scope block was a vector) and initializes a SignalPlot class object that uses pyqtgraph to show a graph.

`remove_block_and_lines()`

Purpose
Function to remove blocks or lines.

Description
Removes a block or a line depending on whether it is selected or not.

Notes
Lines associated to a block being removed are also removed.

`reset_execution_data()`

Purpose
Resets the execution state for all the blocks of a graph.

`reset_memblocks()`

Purpose
Resets the "_init_start_" parameter in all blocks.

`save(`*autosave=False*`)`

Purpose
Saves blocks, lines and other data in a .dat file.

Description
Obtaining the location where the file is to be saved, all the important data of the DSim class, each one of the blocks and each one of the lines, are copied into dictionaries, which will then be loaded to the external file by means of the JSON library.

Parameters
autosave (*bool*) – Flag that defines whether the process to be performed is an autosave or not.

Notes
This function is executed automatically when you want to simulate, so as not to lose unsaved information.

`screenshot(`*zone*`)`

Purpose
Takes a capture of the screen with all elements seen on display.

Parameters
zone – Pygame's layer where the figures, lines and buttons are drawn.

`set_color(`*color*`)`

Purpose
Defines color for an element drawn in pygame.

Parameters
color (*str/(float, float, float)*) – The color in string or rgb to set.

update_blocks_data(*block_data*)

Purpose
Updates information related with all the blocks saved in a file to the current simulation.

Parameters
block_data (*dict*) – Dictionary with Block object id, parameters, variables, etc.

update_global_list(*block_name*, *h_value*, *h_assign=False*)

Purpose
Updates the global execution list.

Parameters

- **block_name** (*str*) – Block object name id.
- **h_value** (*int*) – Value in graph hierarchy.
- **h_assign** (*bool*) – Flag that defines if the block gets assigned with h_value or not.

update_lines()

Purpose
Updates lines according to the location of blocks if these changed place.

update_lines_data(*line_data*)

Purpose
Updates information related with all the lines saved in a file to the current simulation.

Parameters
line_data (*dict*) – Dictionary with Line object id, parameters, variables, etc.

update_sim_data(*data*)

Purpose
Updates information related with the main class variables saved in a file to the current simulation.

Parameters
data (*dict*) – Dictionary with DSim parameters.

class lib.lib.**MenuBlocks**(*block_fn*, *fn_name*, *io_params*, *ex_params*, *b_color*, *coords*, *external=False*)
Class to create and show basic blocks used as a mark to generate functional blocks in the user interface.

Parameters

- **block_fn** (*str*) – Block type, defined according to the available blocks created in DSim
- **fn_name** (*str*) – Function name, function associated to the block type. That function defined in the Functions class.
- **io_params** (*dict*) – Dictionary with block-related parameters (input ports, output ports, b_type, edit inputs/outputs).
- **ex_params** (*dict*) – Dictionary with function-related parameters.
- **b_color** (*str/triplet*) – String or triplet that defines the color of the block in the canvas.
- **coords** (*list*) – List with tuples that contain the location and size of the block in the canvas.

- **external** (*bool*) – Parameter that set a block with an external function (not defined in Functions class).

draw_menublock(*zone*, *pos*)

Purpose
Draws the menu block.

class lib.lib.**SignalPlot**(*dt*, *labels=['default']*, *xrange=100*)

Class that manages the display of dynamic plots through the simulation. *WARNING: It uses pyqtgraph as base (MIT license, but interacts with PyQT5 (GPL)).*

Parameters

- **dt** (*float*) – Sampling time of the system.
- **labels** (*list*) – List of names of the vectors.
- **xrange** (*int*) – Maximum number of elements to plot in axis x.

loop(*new_t*, *new_y*)

Purpose
Updates the time and scope vectors and plot them.

pltcolor(*index*, *hues=9*, *hueOff=180*, *minHue=0*, *maxHue=360*, *val=255*, *sat=255*, *alpha=255*)

Purpose
Assigns a color to a vector for plotting purposes.

sort_labels(*labels*)

Purpose
Rearranges the list if some elements are lists too.

sort_vectors(*ny*)

Purpose
Rearranges all vectors in one matrix.

class lib.lib.**TkWidget**(*name*, *params*, *external=False*)

Class used to create popup windows for changing data, like ports and parameters.

Parameters

- **name** (*str*) – Name of the source (class, block, line, element that calls this class function).
- **params** (*dict*) – Parameters of the source to display in the popup window.

create_entry_widget(*x*)

Purpose
Creates a new entry for the widget.

destroy()

Purpose
Finishes the window instance.

external_toggle()

Purpose
Creates a prompt to reset value parameters from a external block.

get_values()

Purpose
Gets values in a dictionary after the pop-up window is closed.

string_to_vector(*string*)

Purpose
Converts the string into an array vector.

Description
This function takes the resulting string and checks whether or not it corresponds to a vector. The function supports receiving up to a three-dimensional array.

1.6.3 lib.functions

functions.py - Contains all the functions associated with the blocks in the simulation

Classes

DFunctions()	Class to contain all the default functions available to work with in the simulation interface

class lib.functions.**DFunctions**

Class to contain all the default functions available to work with in the simulation interface

adder(*time*, *inputs*, *params*)

Adder function

Purpose
Function that returns the addition of two or more inputs.

Description
This is a process type function. It takes each input value and associates it with a sign (positive or negative), and then adds or subtracts them in an auxiliary variable. The function supports both scalar and vector operations.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['sign']** (*str*) – String that contains all the signs associated to each input value (or vector). It should be noted that in case of having less symbols than vectors, the function will assume that the remaining symbols will add up.
- **params['_name_']** (*str*) – Auxiliary parameter delivered by the associated block, for error identification.

Returns
The sum of all inputs.

Return type
numpy.ndarray

Examples
See example in *Gaussian noise*, *Export data* and *Convergent ODE system*.

Notes
This function returns 'Error' if the dimensions of any of the entries are not equal.

`demux`(*time*, *inputs*, *params*)

Demultiplexer function

Purpose
Function that splits an input vector into two or more.

Description
This is a process type function. It takes the input vector and splits it into several smaller equal vectors, depending on the number of outputs.

Parameters

- **`time`** (*`float`*) – Value indicating the current period in the simulation.
- **`inputs`** (*`dict`*) – Dictionary that provides one or more entries for the function (if applicable).
- **`params['output_shape']`** (*`float`*) – Value defining the number of dimensions with which each output will have.
- **`params['_name_']`** (*`str`*) – Auxiliary parameter delivered by the associated block, for error identification.
- **`params['_outputs_']`** (*`float`*) – Auxiliary parameter delivered by the associated block, for identification of available outputs.

Returns
A given number of outputs, with each output having equal dimensions.

Return type
numpy.ndarray

Examples
See example in *Gaussian noise*.

Notes
This function returns 'Error' if the number of values in the input vector is not enough to get all the outputs at the required dimensions. It also returns a 'Warning' if the vector is larger than required, truncating the values that are not taken.

`derivative`(*time*, *inputs*, *params*)

Derivative function

Purpose
Function that obtains the derivative of a signal.

Description
This is a process type function. It takes the input value and the value of the current time, then takes the difference of these with their previous and obtains the slope.

Parameters

- **`time`** – Value indicating the current period in the simulation.
- **`inputs`** (*`dict`*) – Dictionary that provides one or more entries for the function (if applicable).
- **`params['t_old']`** (*`float`*) – Previous value of the variable time.

- **params['i_old']** (*float*) – Previous value of the entry.
- **params['_init_start_']** (*bool*) – Auxiliary parameter used by the system to perform special functions in the first simulation loop.:type time: float

Returns
The slope between the previous value and the current value.

Return type
numpy.ndarray

Notes
...

exponential(*time*, *inputs*, *params*)

Exponential function

Purpose
Function that returns the value of an exponential from an input.

Description
This is a process type function. It takes the input value, and calculates the exponential of it, with scaling factors for the base as well as the exponent.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['a']** (*float*) – Scaling factor for the base of the exponential.
- **params['b']** (*float*) – Scaling factor for the exponent of the exponential.

Returns
The exponential of the input value.

Return type
numpy.ndarray

export(*time*, *inputs*, *params*)

Block to save and export block signals

Purpose
Function that accumulates values over time for later export to .npz.

Description
This is a drain type function. It takes the input value and concatenates it to a vector. If the input has more than one dimension, the function concatenates so that the saving vector has the corresponding dimensions as a function of time.

Parameters

- **time** – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['str_name']** (*str*) – String supplied by the user with the names of the input values separated by comma: ("value1,value2,value3,...")
- **params['vec_dim']** (*float*) – Value defined by the function that gets the number of dimensions of the input.

- **params['vec_labels']** (*numpy.ndarray*) – Vector produced by the function that gets the name for each element of the saving vector.
- **params['vector']** (*numpy.ndarray*) – Vector that accumulates the input values of the block.
- **params['_init_start_']** (*bool*) – Auxiliary parameter used by the system to perform special functions in the first simulation loop.
- **params['_skip_']** (*bool*) – Auxiliary parameter used by the system to indicate when not to save the input value (RK45 half steps).
- **params['_name_']** (*str*) – Auxiliary parameter delivered by the associated block, for error identification.:type time: float

Returns
A value set in zero.

Return type
numpy.ndarray

Examples
See example in *Export data* and *Convergent ODE system*.

Notes
If not enough labels are detected for 'vec_labels', the function adds the remaining labels using '_name_' and a number depending on the number of missing names.

gain(*time*, *inputs*, *params*)

Gain function

Purpose
Function that scales an input by a factor.

Description
This is a process type function. It returns the same input, but scaled by a user-defined factor. This input can be either scalar or vector, as well as the scaling factor.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['gain']** (*float/numpy.ndarray*) – Scaling value of the input. Can be a scalar value, or a matrix (only for vector multiplication).

Returns
The input value, scaled by the 'gain' factor.

Return type
numpy.ndarray

Examples
See example in *Gaussian noise* and *Convergent ODE system*.

integrator(*time*, *inputs*, *params*, *output_only=False*, *next_add_in_memory=True*, *dtime=0.01*)

Integrator function

Purpose
Function that integrates the input signal.

Description

This is a process type function. It takes the input signal and adds it to an internal variable, weighted by the sampling time. It allows 4 forms of integration, the most complex being the Runge Kutta 45 method.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['init_conds']** (*numpy.ndarray*) – Value that contains the initial conditions for the integrator.
- **params['method']** (*str*) – ['FWD_EULER/BWD_EULER/TUSTIN/RK45'] String that contains the method of integration to use.
- **params['dtime']** (*float*) – Auxiliary variable that contains the sampling time that the simulation is using (fixed step integration).
- **params['mem']** (*numpy.ndarray*) – Variable containing the sum of all data, from start to lapse 'time'.
- **params['mem_list']** (*numpy.ndarray*) – Vector containing the last values of 'mem'.
- **params['mem_len']** (*float*) – Variable defining the number of elements contained in 'mem_list'.
- **params['nb_loop']** (*int*) – Auxiliary variable indicating the current step of the RK45 method.
- **params['RK45_Klist']** (*numpy.ndarray*) – Auxiliary vector containing the last values of K1,K2,K3,K4 (RK45 method).
- **params['add_in_memory']** (*bool*) – Auxiliary variable indicating when the input value is added to 'mem', as well as returning an auxiliary result (method RK45).
- **params['aux']** (*numpy.ndarray*) – Auxiliary variable containing the sum of 'mem' above, with half a simulation step (method RK45)
- **params['_init_start_']** (*bool*) – Auxiliary parameter used by the system to perform special functions in the first simulation loop.
- **params['_name_']** (*str*) – Auxiliary parameter delivered by the associated block, for error identification.

Returns

The accumulated value of all inputs since step zero weighted by the sampling time.

Return type

numpy.ndarray

Examples

See example in *Sine integration* and *Convergent ODE system*.

Notes

The 'init_conds' parameter must be set by the user if the input has more than one dimension. You can define a vector value as [a,b,...], with a and b scalar values.

mux(*time*, *inputs*, *params*)

Multiplexer function

Purpose
Function that concatenates several values or vectors.

Description
This is a process type function. It concatenates each of its entries in such a way as to obtain a vector equal to the sum product of the number of entries by the number of dimensions of each one. The order of the values is given by the order of the block entries.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).

Returns
The vector with all values sorted in a single dimension ((a,1) with a>=1).:rtype: numpy.ndarray

Examples
See example in *Signal products*, *Export data* and *Convergent ODE system*.

noise(*time*, *inputs*, *params*)

Gaussian noise function

Purpose
Function returns a normal random noise.

Description
This is a source type function. It produces a gaussian random value of mean mu and variance sigma**2.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['mu']** (*float*) – Mean value of the noise.
- **params['sigma']** (*float*) – Standard deviation value of the noise.

Returns
Gaussian random value of mean mu and variance sigma**2.

Return type
numpy.ndarray

Examples
See example in *Gaussian noise*.

ramp(*time*, *inputs*, *params*)

Ramp source function

Purpose
Function that returns a value that changes linearly over time.

Description
This is a source type function, which is piecewise. The value changes linearly over time, and can increase or decrease.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['slope']** (*float*) – The value of the slope that the ramp has.
- **params['delay']** (*float*) – Indicates a point in time where the start of the ramp happens.

Returns
The value of the slope multiplied by the difference between 'time' and 'delay'.

Return type
numpy.ndarray

Examples
See example in *External derivator (Z-process)*.

scope(*time*, *inputs*, *params*)

Function to plot block signals

Purpose
Function that accumulates values over time to plot them with pyqtgraph both later and during the simulation.

Description
This is a drain type function. It takes the input value and concatenates it to a vector. If the input has more than one dimension, the function concatenates in such a way that the saving vector has the corresponding dimensions as a function of time.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['labels']** (*str*) – String supplied by the user with the names of the input values separated by comma: ("value1,value2,value3,...")
- **params['vec_dim']** (*float*) – Value defined by the function that gets the number of dimensions of the input.
- **params['vec_labels']** (*numpy.ndarray*) – Vector produced by the function that gets the name for each element of the saving vector.
- **params['vector']** (*numpy.ndarray*) – Vector that accumulates the input values of the block.
- **params['_init_start_']** (*bool*) – Auxiliary parameter used by the system to perform special functions in the first simulation loop.
- **params['_skip_']** (*bool*) – Auxiliary parameter used by the system to indicate when not to save the input value (RK45 half steps).
- **params['_name_']** (*str*) – Auxiliary parameter delivered by the associated block, for error identification.

Returns
A value set in zero.

Return type
numpy.ndarray

Examples
See example in *Sine integration*, *Signal products*, *Gaussian noise*.

Notes
If not enough labels are detected for 'vec_labels', the function adds the remaining ones using '_name_' and a number depending on the number of missing names.

`sigproduct`(*time*, *inputs*, *params*)

Element-wise product between signals

Purpose
Function that returns the multiplication by elements of two or more inputs.

Description
This is a process type function. It takes each input value and multiplies it with a base value (or vector).

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).

Returns
The multiplication of all inputs.

Return type
numpy.ndarray

Examples
See example in *Signal products*.

Notes
Unlike the sumator function, this one does not check if the inputs have the same dimensions, since there may be occasions where the result needed may be something larger.

Limitations
The function does not check that the result has the desired dimensions, so it is a job to be done by the user.

`sine`(*time*, *inputs*, *params*)

Sinusoidal source function

Purpose
Function that returns a sinusoidal in time.

Description
This is a source type function. It returns a sinusoidal with variation in the parameters of amplitude, frequency and initial angle.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['amplitude']** (*float*) – Amplitude value taken by the sinusoidal.
- **params['omega']** (*float*) – Value in rad/s (2*pi*f) of the frequency taken by the sinusoidal.

- **params['init_angle']** (*float*) – Value in radians of the angle taken by the sinusoidal at time zero.

Returns
A sinusoidal of amplitude 'amplitude', frequency 'omega' and initial angle 'init_angle'.

Return type
numpy.ndarray

Examples
See example in *Sine integration*.

step(*time*, *inputs*, *params*)

Step source function

Purpose
Function that returns a constant value over time.

Description
This is a source type function, which is piecewise. It can be used to indicate the beginning or end of a branch of a network.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).
- **params['value']** (*float/numpy.ndarray*) – The value that the function returns. It can be a scalar (float) as well as a vector ([float, ...]).
- **params['delay']** (*float*) – Indicates a point in time where the piecewise jump occurs.
- **params['type']** (*str*) – ['up'/'down'/'pulse'/'constant'] Indicates whether the jump is upward ('value'), downward (0), in pulse form or constant ('value').
- **params['pulse_start_up']** (*bool*) – Indicates whether the pulse starts upwards (True) or downwards (False).
- **params['_init_start_']** (*bool*) – Auxiliary parameter used by the system to perform special functions in the first simulation loop.
- **params['_name_']** (*str*) – Auxiliary parameter delivered by the associated block, for error identification.

Returns
The value defined in 'value' or 0.

Return type
numpy.ndarray

Examples
See example in *Vectorial integration*, *Gaussian noise* and *Signal products*.

terminator(*time*, *inputs*, *params*)

Signal terminator function

Purpose
Function that terminates with the signal.

Description
This is a drain type function. It takes any input value and does nothing with it. This function is useful for terminating signals that will not be plotted.

Parameters

- **time** (*float*) – Value indicating the current period in the simulation.
- **inputs** (*dict*) – Dictionary that provides one or more entries for the function (if applicable).

Returns
A value set in zero.

Return type
numpy.ndarray

Examples
See example in *Signal products*.

1.7 Topics for Further Improvement

1. **Support for other similar processes.**
   Right now this algorithm only works with input-output signals, but no other types of systems that can be described with graphs, like mass balance systems or electric grids or even distributed systems. So one thing that could be worked in the future are support for other graph represented systems.
2. **Use of Threading, improve and simplify some processes.**
   Right now this library works with everything under the same while loop. So anything that can affect the execution loop also affects the interface fps, so it's necessary for the future that both loops can be happening in parallel, essentially split the UI process from the graph executing process).

   Also, improve the versatility of some functions by using better methods to programming in python. e.g.: Use of *args and **kwargs to allow more specific parameters to some functions without affecting the readability and stability of others.
3. **Use of PyQT5.**
   Right now the package works with pygame only, in an acceptable way, but to make this a better option for the user, it is necessary to rework interface using PyQT5 or another similar alternative. It's important to consider this as changing the license from MIT to GPL, due to the requirements of some of these libraries.
4. **More support for special functions.**
   Add support to functions that require different clocks, or elements not implemented yet, like an option to set parameters through inputs with data from other blocks.

PYTHON MODULE INDEX

INDEX

N

O

P

R

S

T

U