Analog System Compiler

Cyril Collineau

Version 1.0

Table of Contents

Introduction

The complexity and cost of building electronic systems make it essential to adopt rigorous methodologies. Simulation plays a crucial role in the design process, allowing for modeling and analysis of circuit behavior. It provides an opportunity to optimize performance, detect errors, and validate the design before advancing to the production phase.

Numerous free open-source electronic simulators are available, with the most popular being the Spice clones (**pspice**, **ngspice**, **hspice**, etc.) and their many frontends (**PySpice**, etc.), **Qucs** (Quite Universal Circuit Simulator), and **Xyce**/**Spectre**.

Initially, these simulators contained component models whose descriptions in the source code could not be modified. As a result, it was impossible to add new components other than by combining existing models (for instance, through Spice's **.SUBCKT** directive, which allows a sub-circuit to be included in the main circuit).

However, over time, as needs evolved, new features were introduced to allow the integration of components whose behavior is defined by the users themselves. These components are described by symbolic equations using standard mathematical functions through language extensions such as **Verilog-AMS**[1] or, as shown in the example below, **VHDL-AMS**[2]:

```
library ieee;
use work.electrical_system.all;
use ieee.math_real.all;
entity resistor is
    generic (resistance : real);
    port (terminal n1, n2: electrical);
end resistor;

architecture analog of resistor is
quantity r_u across n1 to n2;
quantity r_i through n1 to n2;
begin
    r_u == resistance * r_i
end architecture one;
```

To support these extensions, **Xyce**, **Qucs** and some other simulators incorporate **ADMS** (Automatic Device Model Synthesizer) [3], which is a tool able to translates a **Verilog-A** circuit description into another programming language, typically *C* or *C++*, that conforms to the simulator interface.

The integration of symbolic computing capabilities into simulation tools opens up new perspectives in terms of flexibility and modeling power. Among other benefits, it enables what is known as "compact modeling" [4].

However, most of these simulators employ resolution methods that transform the system of equations into matrix form for solving. This approach is inherently complex, requiring the evaluation of input equations, the organization of unknown variable coefficients into a matrix, and the subsequent solving of this matrix. Moreover, this process must be repeated at each calculation step, as the coefficients may vary throughout the simulation.

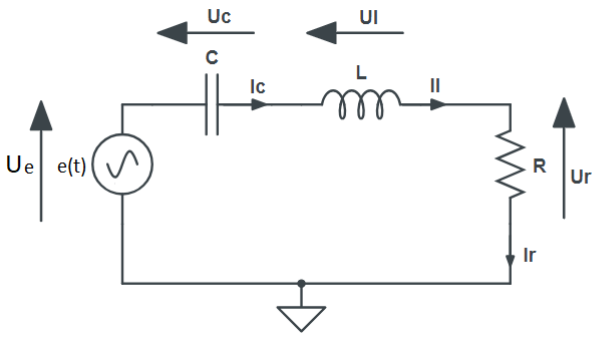
This document presents *ASysC*, an open-source tool that generates Python code directly from circuit descriptions written in a symbolic language. Unlike traditional circuit simulation methods that rely on matrix-based computations, ASysC introduces a new paradigm by preserving the entire process—from circuit description to final solution—in symbolic form.

This tool belongs to the **CAS** (**Computer Algebra System**) family [5]. Such tools are commonly used by mathematicians and researchers, with the most well-known being **Maxima**, **Axiom**, **Maple** and **Mathematica**.

With a **CAS** tool, for example, it is possible to determine how the result of an equation changes by assigning numerical values to symbolic variables.

This type of tool is well-suited for defining user components, as it natively operates with symbolic equations. It can seamlessly integrate these components to construct analog circuits and, furthermore, solve the corresponding systems of equations that describe them.

To understand the simulation process, let's first review some fundamentals and examine a classic **R**,**L**,**C** electronic circuit as an introductory example:



When formulating the equations of an electronic system, there are two sets of equations:

- the physical relations that describe the behavior of each component,
- the interconnection relations that describe how the components are connected to each other are known as Kirchhoff's laws: the junction law and the loop law.

The physical relationships linking current and voltage for every component except for the generators are:

$$\begin{cases} U_r = R \cdot I_r \\ U_l = L \cdot \frac{dI_l}{dt} \\ I_c = C \cdot \frac{dU_c}{dt} \\ U_e = E(t) \end{cases}$$

The Kirchhoff's loop rule in our circuit gives:

$$U_r + U_l + U_c = U_e$$

The Kirchhoff's junction rule gives:

$$\begin{cases} I_r = I_l \\ I_l = I_c \\ I_c = I_e \end{cases}$$

For frequency analysis, it's convenient to apply the Laplace transformation [6], using the complex variable $p = j \cdot \omega$. In the following, we'll assume that $E(t)$ is equal to the Dirac function $\delta(t)$, which is constant and equal to 1 in the frequency domain. The previous equations can then be expressed as :

$$\begin{cases} U_r = R \cdot I_r \\ U_l = p \cdot L \cdot I_l \\ I_c = p \cdot C \cdot U_c \\ U_e = 1 \\ U_r + U_l + U_c = U_e \\ I_r = I_l \\ I_l = I_c \\ I_c = I_e \end{cases}$$

The matrix representation of this system gives:

$$\begin{pmatrix} 1 & & & -R & & & \\ & 1 & & & -p \cdot L & & \\ & & 1 & & & -\frac{1}{p \cdot C} & \\ & & & 1 & & & \\ & & & & 1 & -1 & \\ & & & & & 1 & -1 \\ 1 & 1 & 1 & -1 & & & \end{pmatrix} \times \begin{pmatrix} U_r \\ U_l \\ U_c \\ U_e \\ I_r \\ I_l \\ I_c \\ I_v \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

In an electronic circuit, nodes are rarely connected to a large number of branches. As a result, in most cases, we obtain matrices composed mainly of zeros, known as sparse matrices [7]. The example shown above is a good illustration of this.

Although the matrix representation is widely used, it presents two problems:

- Firstly, the matrix representation may consumes a lot of memory, since it is mainly made up of zeros.
- Secondly, a calculation over a given frequency interval requires the matrix to be reconstructed with all coefficients and then solved for each simulation step, i.e. for each iteration of the value of ω.

For instance, if we want to display 200 points, corresponding to 200 simulation steps, we would need to reconstruct and solve the matrix 200 times. While there are optimizations available to speed up the resolution of these matrices, such as LU-decomposition [8], the process remains computationally intensive.

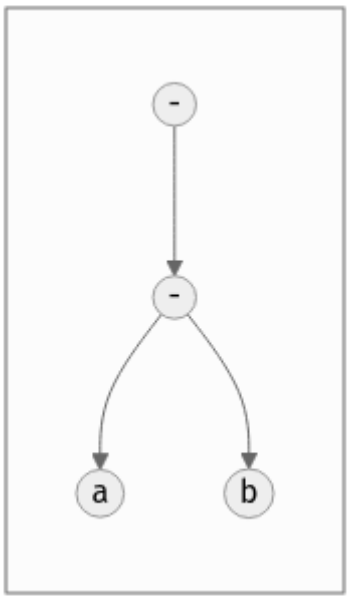
The approach presented in this document proposes an alternative method, where the matrix resolution is carried out directly in symbolic form in order to easily re-evaluate the expressions obtained as a function of ω at each simulation step.

ASysC, the Analog System Compiler

ASysC is a tool that has slowly evolved over the years to the point where it can be used to simulate electronic systems. Strictly speaking, *ASysC* does not simulate the electronic system, but it is capable of assembling the equations of components declared in a netlist into a global system of equations, which it then solves symbolically. Once this stage has been completed, it will generate *Python* code, which will then be used to simulate the circuit itself. Unlike other simulators mentioned in the introduction, *ASysC* does not solve systems in matrix form, thus avoiding excessive memory usage.

How does a CAS work?

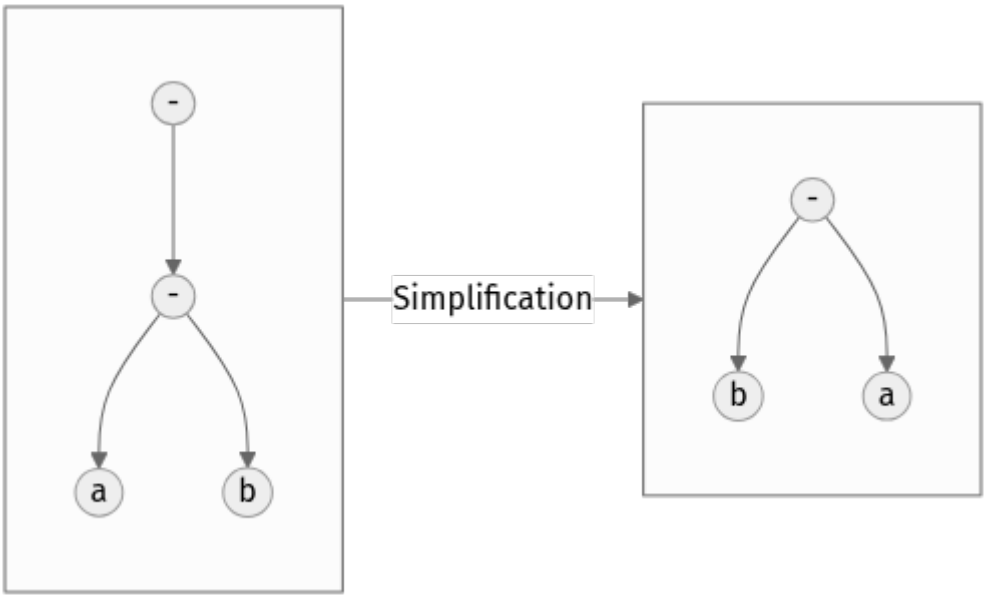
In a CAS, mathematical expressions are stored in tree form. For example, the expression $-(a-b)$ is stored internally as follows:



A CAS contains a set of rules that allow you to perform transformations on the trees.

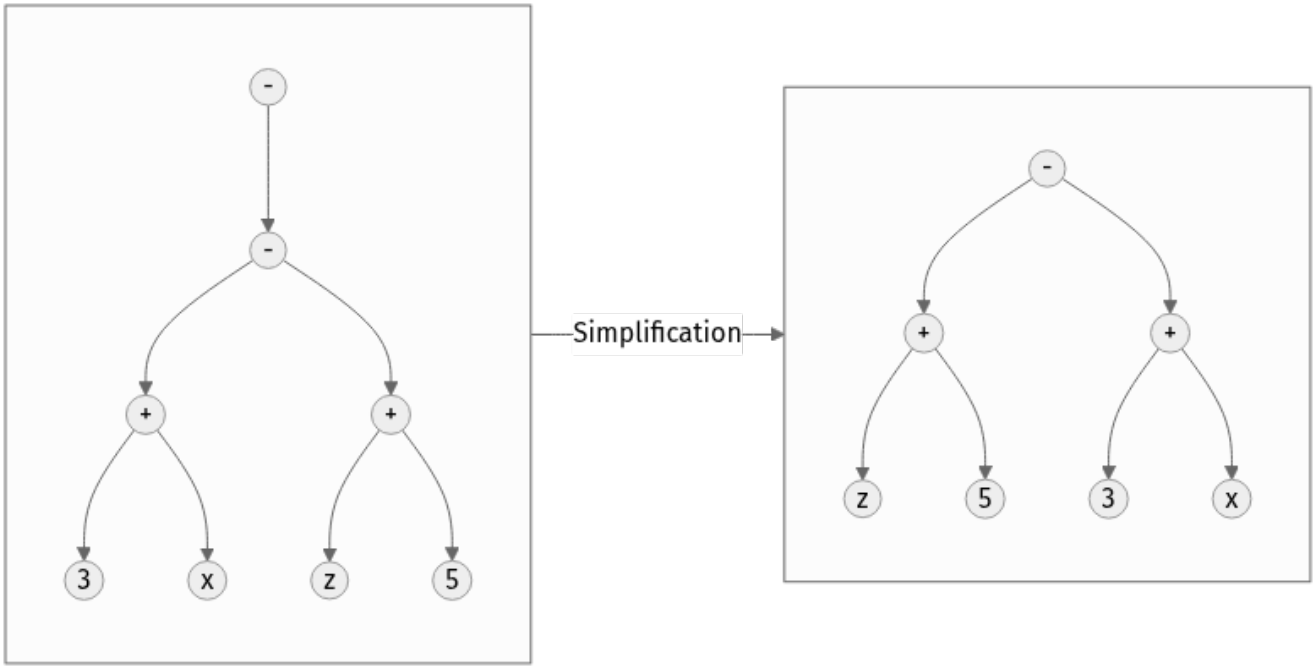
For example, there is a simplification rule that transforms the expression $-(a-b)$ into $b-a$. This rule is simply written $-(a-b) := (b-a)$.

It transforms the input tree into a simplified output tree:



Parameters **a** and **b** can be either numbers, variables or expressions themselves formed by sub-trees.

For example, the expression $-((3+x)-(z+5))$ is recognized as being of the form $-(a-b)$, and is therefore transformed by the rule in question into $(z+5)-(3+x)$, which visually translates into the following transformation:



This example illustrates a simple rule, but sophisticated transformations are possible with more complex rules that interact with each other recursively. This set of rules can be used to solve equations, linear systems, Taylor series calculations and more.

CAS rules overview

Introduction

ASysC is built on a very simple system of rules, which can be used to perform a whole range of mathematical operations, such as :

- simplification of expressions,
- solving one- or two-degree equations,
- solving systems,
- matrix operations,
- determinant calculations,
- Taylor series decompositions.

To go even further and enable it to solve electronic circuits, *ASysC* also integrates the rules needed to apply Kirchhoff's laws, as well as fundamental transformations in electronics, such as derivatives and integrals transformations for frequency and transient analysis, and linearization of non-linear functions.

For system solving, *ASysC* contains rules that support several system solving algorithms:

- Gauss-Jordan elimination method,
- Resolution by Cramer's method, also known as the determinant method.

Although both methods are implemented, it is the latter that will be preferred for solving systems of equations, as the results produced are compatible with the equations governing switch models (either voltage is zero, or current is zero, which presents a physical singularity). If we use the Gauss Pivot method, the structure of the equations obtained is different, and we may end up with divisions by zero when evaluating these equations. This is the case, for example, with a circuit containing an ideal diode (either zero voltage or zero current). On the other hand, solving a system with Cramer's method has the disadvantage of being much slower than with the Gauss-Jordan method.

Rule definition files

A large number of rules are available, some generic, others specialized for electronic circuits. These rules are contained in a set of files in text format (***.rule**)

units.rule	rules on units,
components.rule	definition of analog components,
ic.rule	mixed components, such as the **NE555**.
electricity.rule	transformations for frequency and time analysis, as well as Kirchhoff's laws,
includes.rule	input files for rule definitions,

logic.rule	some logic components: AND, OR, XOR gates and D and SR flip-flops,
matrix_vector.rule	rules for operations on vectors and matrices,
main.rule	basic algebraic rules,
symbols.rule	definition of operator symbols,
derivatives.rule	derivative transformations,
tests.rule	non-regression tests

Structure of a rule

Let's take a look at some concrete examples of how a **CAS** works. As a first example, let's use it as a simple calculator:

```
cd lightcas/bin
./asysc
************************************
*** ASysC Console                ***
*** (C) Cyril Collineau 2006-2025 ***
************************************
Type “help” for help.
> 2*2
4
```

Second example more complex, variable assignment:

```
> b:=a+a
2*a
> a:=2
2
> b
4
```

We can see here that b initially accepts the value $2 \times a$ without knowing the value of a. This is the fundamental difference between a **CAS** and a calculator. It's when we specify that a is equal to 2, that b displays the value 4.

In *ASysC*, a rule is always defined with the assignment operator **:=**.

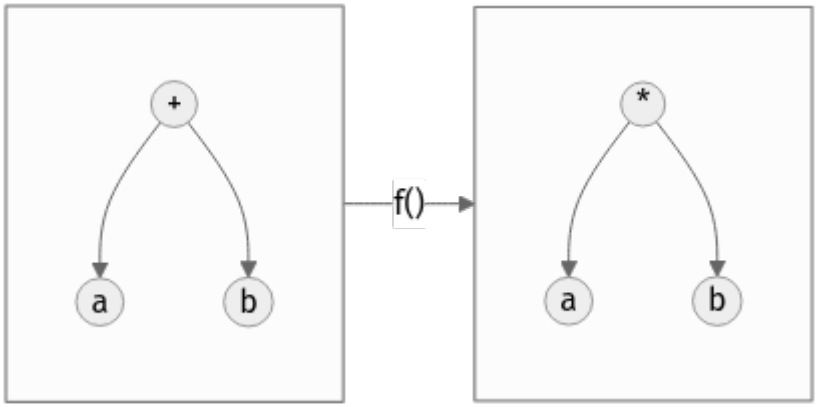
Third example, the declaration of a simple function with two parameters, an addition:

```
> my_sum(a,b) := a + b
> my_sum(2,3)
5
```

Let's go a step further and declare a symbolic transformation, like the slightly absurd example below, which associates multiplication with addition via the '**f**' transformation. Here, the parameter is no longer a variable but a tree made up of the + operator and two parameters **a** and **b** :

```
> f(a+b):=a*b
a*b
> f(2+4)
8
```

This feature as strange as it may seem enables *ASysC* to perform all kinds of transformations on trees, by performing tree pattern recognition followed by tree substitution.



For common calculations, pre-configured rules are available.

Example of derivative calculation with the **DER()** function:

```
> DER(COS(3*x),x)
-(3*SIN(3*x))
```

Example of writing a function as a Taylor sequence with the **TAYLOR()** function:

```
> TAYLOR(COS(x),x,0,10)
1-2.7557319224e-07*x^10+2.48015873016e-05*x^8-0.00138888888889*x^6+0.0416666666667*x^4-0.5*x^2
```

Example of solving equations with function **SOLVE()** :

```
> SOLVE(x-2,x)
2
> SOLVE(x^2-2*x+4,x)
{1-1.73205080757*j(),1+1.73205080757*j()}
```

Example of solving systems of equations :

```
> SOLVE({x-y+1,x+y-5},{x,y})
```

```
{2,3}
```

Another interesting capability of *ASysC* is its support for lists. This expression is used to define components and circuits, as we'll see in the following chapters:

```
> my_list := {a;b;c}
```

Recursions with rules

Rules can be called recursively as follows:

```
> my_factorial( x ) := x * my_factorial( x - 1 )
my_factorial(x-1)*x
```

This capability makes it possible to perform complex transformations. But using this rule alone will result in infinite recursion. To avoid this, we need to add a specific rule for the stop condition:

```
> my_factorial( 1 ) := 1
1
```

If you look at the rules described in the files (***.rule**) located in the `lightcas/rules` directory, you'll see that the vast majority of rules are recursive.

Now try `my_factorial(6)` for example:

```
> my_factorial( 6 )
720
```

You get `720`, which is the correct result of $6!$

Compilation, installation and use

Compilation and installation

Compiling *ASysC* is normally straightforward. It requires no external dependencies other than **g++** and **make** executables. A simple invocation of the make command in the root directory suffices:

```
git clone https://github.com/analog-system-compiler/asysc.git
cd asysc
git submodule update --init
make
```

ASysC has been compiled and tested under both **Linux** and **MSYS2** environments. Once compiled under Linux, its size does not exceed 100Kb. This is not the tool you'll hesitate to remove from your hard disk because it takes up too much space!

Once the code has been compiled, the Makefile will automatically generate the *Python* files needed to simulate the circuits. This operation can sometimes take several seconds.

Before going any further, check that the **NumPy** and **Matplotlib** libraries are installed in your *Python* environment.

ASysC command-line options

The `asysc` command accepts the following options:

-i input file (***.cir**)

-o output file (***.py**). If omitted, output will be in a file with the same name as the input file but with the **.py** file extension.

-t type of analysis: **ac** or **trans**.

-c name of the circuit to be analyzed. By default: "CIRCUIT".

Example :

```
../lightcas/bin/asysc -i ac/RLC/RLC.cir -o ac/RLC/RLC.py -t ac
```

This example will create a *Python* model for frequency analysis from the input file **RLC.cir**. The process is described in more detail in the following paragraphs.

Applying *ASysC* to electronics

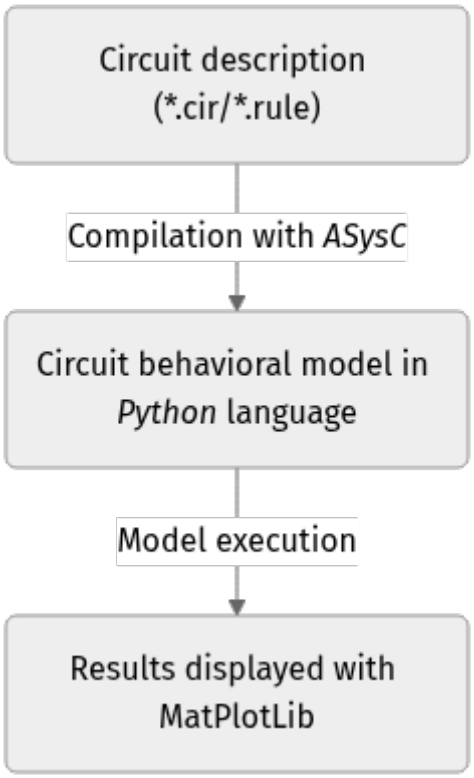
From equations to simulation

As mentioned previously, *ASysC* is not a simulator, but a tool for manipulating algebraic expressions. Used on its own, it is unable to perform a simulation. To perform this step, we'll be using the *Python* language and the popular **NumPy** and **Matplotlib** libraries. **NumPy** will be used for complex mathematical calculations, while **Matplotlib** will be used to display the results graphically. Any other graphics library can be used if desired.

To summarize, the process steps are as follows:

1. the circuit is described in text format (***.cir** or ***.rule**),
2. this description is given to the *ASysC* compiler, which analyzes the circuit, solves the equation system and creates a behavioral model in *Python*,
3. this *Python* model is then executed for simulation,
4. once the simulation is complete, *Python* displays the results graphically.

This is illustrated below:



It's best not to modify the *Python* file generated by *ASysC*, as it will be overwritten. For simulation purposes, this file cannot be used on its own; it must be supplemented by two other *Python* files:

circuit_base.py this file contains the basic classes for circuit simulation. It is common to all simulated circuits.

simulation.py this file contains the simulation and display functions for **Matplotlib**. It is specific to the circuit and simulation run. The user modifies it according to the simulation and can customize display as desired.

The great advantage of using *Python* for simulation is that all simulation data is accessible directly

in **NumPy** arrays. It is therefore possible to take advantage of the power of this library to carry out other processes, such as applying a Fourier transform following a transient analysis for instance.

Declaring a component

The fundamental idea behind *ASysC* is to consider that:

- a circuit is a function that returns a list of components,
- a component is a function that returns a list of equations.

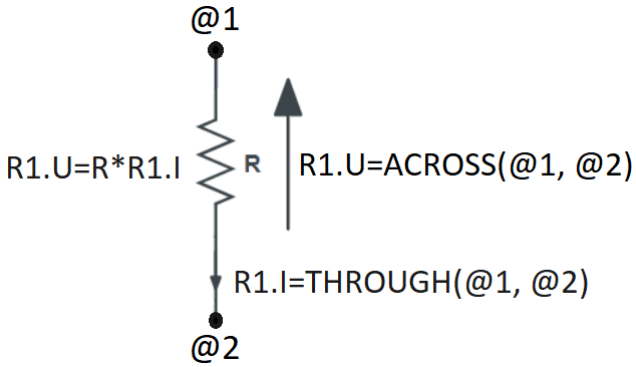
In short, all is based on functions! This incredibly simple mechanism offers almost limitless possibilities.

Hereafter, the term **rule** rather than **function** is sometimes used, which is a more appropriate term for a **CAS**, even though it's basically the same thing.

Thus, a component will be defined as a rule comprising a list of equations, as follows:

```
my_component( node1, node2, ..., noden, parameter1, parameter2, ..., parameteren ) := {

    equation1;
    equation2;
    ...
}
```

Here's a concrete example for a resistor **R1** connected to nodes **node_p** and **node_n** :



The resistor component declaration contains three equations:

- an equation for defining the potential at nodes **node_p** and **node_n** using the **ACROSS()** function. This function will be used to determine the voltage in the circuit by applying the Kirchhoff's loop rule.
- an equation to define the current between nodes **node_p** and **node_n**. With the **THROUGH()** function. This function will be used to determine the current in the circuit by applying the Kirchhoff's junction rule.
- an equation describing the physical relationship between current and voltage. In the case of a resistor, we have $U = R \cdot I$.

This gives the following declaration for the resistor:

```
NAME.CR(node_p, node_n, R) := {

    NAME.U = ACROSS(node_p, node_n);
    NAME.I = THROUGH(node_p,node_n);
    NAME.U = R*NAME.I

};
```

This representation is quite similar to the *VHDL-AMS* language, in which the keywords **accross** and **through** are used. In the same way as for resistors, the declaration of an inductance uses the **DER()** function to declare a derivative:

```
NAME.CL(node_p, node_n, L) := {

    NAME.U = ACROSS(node_p, node_n);
    NAME.I = THROUGH(node_p, node_n);
    NAME.U = L * DER(NAME.I, t)

};
```

And for the declaration of a capacitor :

```
NAME.CC(node_p, node_n, C) := {

    NAME.U = ACROSS(node_p, node_n);
    NAME.I = THROUGH(node_p, node_n);
    NAME.I = C * DER(NAME.U, t)

};
```

Note that "**.**" is a hierarchical operator that will propagate the instance name throughout the component's internal equations, to avoid ending up with variables with identical names when all equations are extracted for system resolution.

Thus, **NAME** will be replaced by **R1** and the instantiation **R1.CR** will replace the voltage **NAME.U** by **R1.U** and the current **NAME.I** by **R1.I**.

NOTE

1. Prefixing nodes with the @ character is not mandatory. This is used to identify the nodes in the parameter list, to provide better readability. Nothing prevents you from noting your nodes "my_node1", "my_node2", etc.
2. The semicolon is used as a list separator **{ ; ; }** and therefore MUST not appear at the end of the last line of your circuit declaration.

Instantiating a component

A component instantiation is written in the following form:

```
<Instance name>.<Component name>( <node list>, <parameter list>)
```

For example, a resistance of type **CR** named **R1**, connected between nodes **@3** and **@4** with a value of **1K** will be instantiated as follows:

```
R1.CR( @3, @4, 1K)
```

Quantities

ASysC also manages quantities for better readability. The quantities supported are:

p	pico
n	nano
u	micro
m	milli
k or K	kilo
M	mega
G	giga

So if you write **1G** for example, it will correspond to the value **1000000000**.

Frequency and time analysis

If you are looking forward to see what *ASysC* is capable of and want to pause before continuing the article, you can run all the examples with the following command typed in the root directory:

```
make run
```

This command starts the simulation of all the examples contained in the project and display the results. Now that you have been able to appreciate the performance of *ASysC* let's go into detail about the different analysis supported.

Frequency analysis

Let's start with the simulation of our first circuit and begin in a first step by a frequency analysis of the circuit (so-called «AC» analysis). The transformation of the system of equations to perform a frequency analysis is done by calling the **SOLVE_AC()** function:

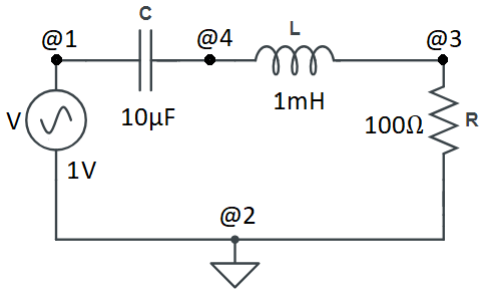
```
CIRCUIT := SOLVE_AC( RLC )
```

This function will start the system resolution by preapplying the Laplace transform. The converted circuit is then used for frequency analysis. It is called «CIRCUIT» because it is the default variable used by *ASysC* to recognize the circuit that will be converted into a *Python* model. This can be changed by a specific option in the command line.

NOTE	The bias point calculation is not yet implemented in the current version of *ASysC*.

Example with a conventional R, L, C circuit

Let's return to our initial example. Our circuit comprises four components: a voltage source **CV**, a resistor **CR**, an inductor **CL**, a capacitor **CC**, whose names are **V**, **R**, **L** and **C** respectively. Some component names are preceded by the letter '**C**' to avoid conflicts with instance names. Nodes are identified as **@1**, **@2**, **@3** and **@4**. The circuit is shown below:



The netlist description of this circuit is:

```
`include <components.rule>
```

```
RLC() := {
    V.CV(@1, @2, 1);
    R.CR(@2, @3, 100);
    L.CL(@3, @4, 1m);
    C.CC(@4, @1, 10u)
};

CIRCUIT := SOLVE_AC( RLC )
```

Components are declared in a list assigned to the **RLC()** rule, which is then given as a parameter to function **SOLVE_AC()**.

The `include directive is used to include the definitions of components that are not included by default, unlike the basic rules. It is entirely possible to write your own component library and include it in your project.

Generating *Python* code

Having seen how to declare a circuit and instantiate components, let's move on to the most interesting part: generating a *Python* model of our circuit.

For this step, we'll use the following command:

```
cd examples/
../lightcas/bin/asysc -i ac/RLC/RLC.cir -o ac/RLC/RLC.py -t ac
```

This command generates the circuit model in *Python*, as shown in the piece of code below:

```
def compute_f(self):
    self._setf(self.C_U, -(1/((0.001*self.s+1)+1e-08*self.s**2)), self.freq)
    self._setf(self.V_I, -((1e-05*self.s)/((0.001*self.s+1)+1e-08*self.s**2)), self.freq)
    self._setf(self.V_U, ((1e-08*self.s**2+0.001*self.s)+1)/((1e-08*self.s**2+0.001*self.s)+1),
self.freq)
    self._setf(self.R_U, -((0.001*self.s)/((0.001*self.s+1)+1e-08*self.s**2)), self.freq)
    self._setf(self.R_I, -((1e-05*self.s)/((0.001*self.s+1e-08*self.s**2)+1)), self.freq)
    self._setf(self.L_U, -((1e-08*self.s**2)/((1e-08*self.s**2+1)+0.001*self.s)), self.freq)
    self._setf(self.L_I, -((1e-05*self.s)/((1+1e-08*self.s**2)+0.001*self.s)), self.freq)
    self._setf(self.C_I, -((1e-05*self.s)/((0.001*self.s+1)+1e-08*self.s**2)), self.freq)
```

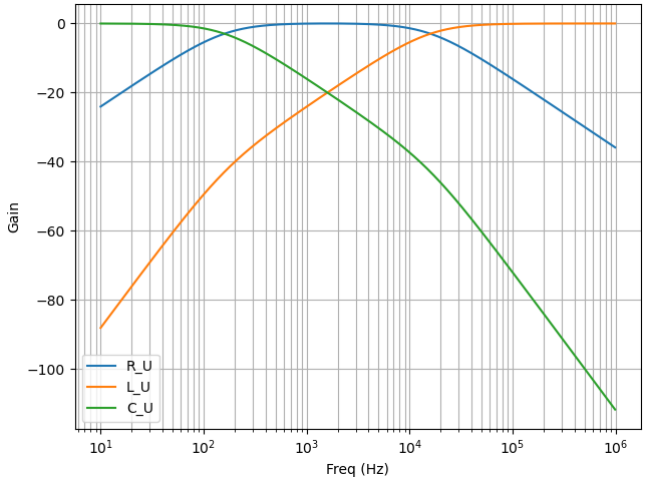
This code extract shows the classic transfer functions of an **R, L, C** circuit. To run the frequency simulation from the *Python* code, create the circuit object in the **simulation.py** file and call the **simulate_f** function with the start frequency, end frequency and desired number of samples as parameters:

```
my_circuit = circuit()
my_circuit.simulate_f(10, 1e6, 100)
```

For our example, the simulation is launched with the command :

```
cd ac/RLC/
python3 simulation.py
```

After running the simulation, the remaining task is to visualize the data stored in **NumPy** arrays using the **MatPlotLib** library. The expected result is shown graphically:



Frequency analysis with dynamic change of parameter values

In some cases, it may be useful to be able to change the value of some parameters between simulation runs without recompiling the circuit.

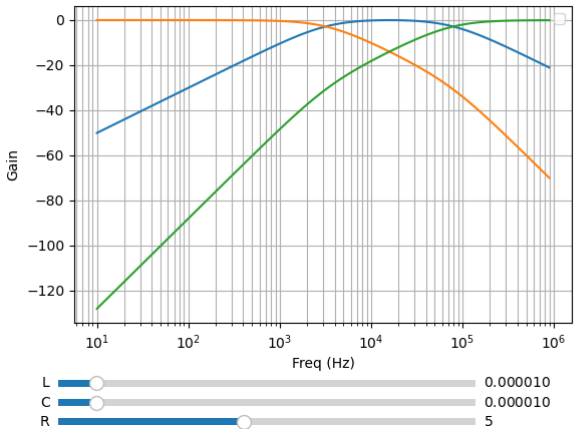
To achieve this, the **_getv()** function is used. During simulation, this function retrieves the current value of the specified variable. The updated circuit is as follows:

```
`include <components.rule>

RLC( PR, PL, PC ) := {

    V.CV (@1, @2, 1);
    R1.CR(@2, @3, PR );
    L1.CL(@3, @4, PL );
    C1.CC(@4, @1, PC )
};

CIRCUIT := SOLVE_AC( RLC( _getv( R ), _getv( L ), _getv( C ) ) )
```

The **RLC()** circuit now admits three parameters **PR**, **PL** and **PC**, whose values will take the values of the *Python* variables **self.R**, **self.L** and **self.C** through the **_getv()** function during simulation.

The user can then dynamically change the values of the **R**, **L** and **C** components using the slider functionality available in the **MatplotLib** library. For more information, please refer to the **RLC_slider** *Python* example code.

The simulation results window featuring sliders:



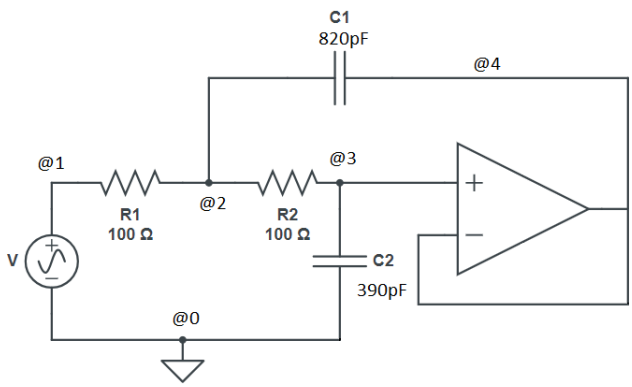
Another circuit example: a Sallen-Key type filter.

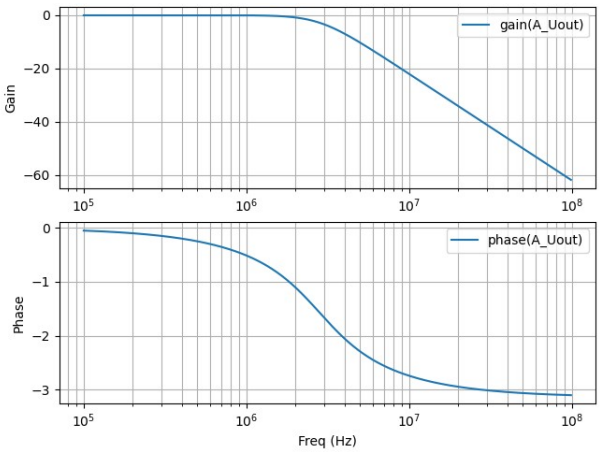
In the example below is presented the simulation of a second-order **Sallen-Key** filter [9] consisting of two RC-cells and an operational amplifier. The simulation displays gain and phase diagrams as a function of frequency.

The circuit description:

```
`include <components.rule>

sallen_key() :=
{
    V.VSIN   ( @1, @0, 1 );
    R1.CR    ( @1, @2, 100 );
    R2.CR    ( @2, @3, 100 );
    C1.CC    ( @2, @4, 820p );
    C2.CC    ( @3, @0, 390p );
    A.IOPAMP ( @3, @4, @4, @0 )
};

CIRCUIT := SOLVE_AC( sallen_key )
```

The circuit schematic and simulation results with gain and phase display:





Time analysis, also known as "Transient Analysis"

Transforming the system of equations to perform a transient analysis is done in the same way as above, but with a call to function **SOLVE_TRANS()**:

```
CIRCUIT := SOLVE_TRANS( RLC )
```

This directive solves the system by first performing all the necessary transformations on the non-linear and reactive elements. As previously stated, the transformed circuit must always be named “CIRCUIT”. This is the variable used by *ASysC* to convert the circuit into a *Python* model.

There are several methods for solving a non-linear system. The simplest is the Newton-Raphson method [10], also known as the “tangent” method. It involves replacing the system’s non-linear functions with their tangents, calculated at the iteration point. Once the tangents have been calculated, the system solved and the unknowns variables determined, the tangents are recalculated at the new iteration point and the process is repeated. The problem thus comes down to solving a sequence of linear systems with their solutions gradually converging to the final result.

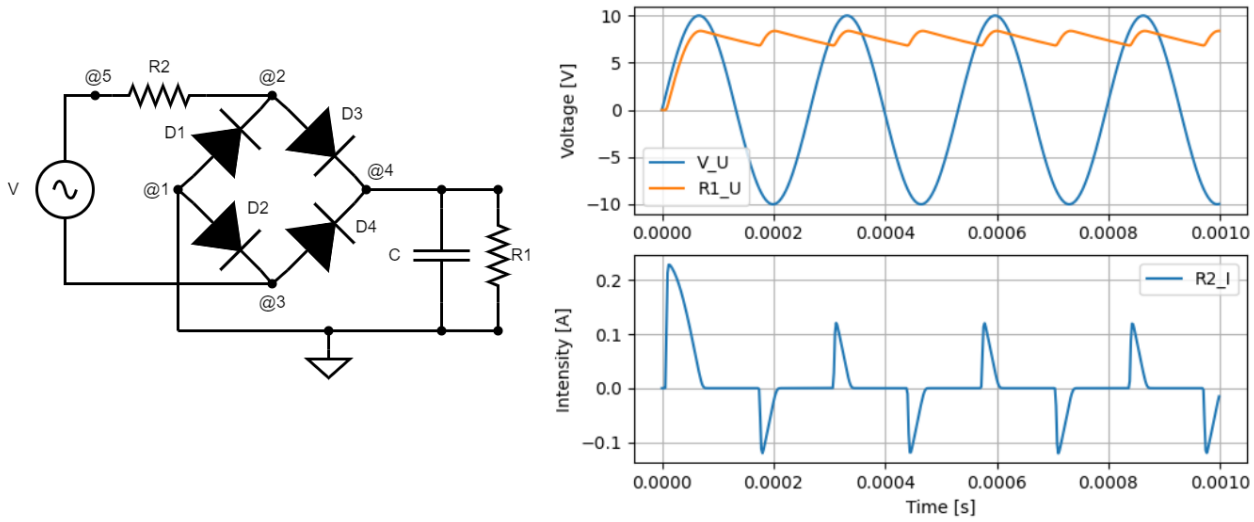
For the simulation of reactive elements such as capacitors or inductors, the trapezoidal integration algorithm is used. This provides both acceptable results and a simple implementation.

Example of a Graetz bridge simulation in transient analysis

In this example, the circuit consists of a sinusoidal source **V**, a source resistor **R2**, four diodes (**D1**, **D2**, **D3** and **D4**) and a load represented by a resistor **R1** and a capacitor **C**. The circuit description is as follows:

```
`include <components.rule>

graetz_bridge( RS ) :=
{
    V.VSIN   ( @5, @3, 10, 2*PI*600 );
    R2.CR    ( @5, @2, 1 );
    D1.DIODE ( @1, @2 );
    D2.DIODE ( @1, @3 );
    D3.DIODE ( @2, @4 );
    D4.DIODE ( @3, @4 );
    R1.CR    ( @1, @4, RS );
    C.CC     ( @1, @4, 1u )
};

CIRCUIT := SOLVE_TRANS( graetz_bridge( 500 ) )
```

The corresponding schematic and simulation result:



Similar to the frequency simulation, the time simulation is initiated from the **simulation.py** Python code by creating the circuit object and invoking the **simulate_t()** function.

This function receives as parameters the simulation time, the number of samples, the desired resolution and the maximum number of iterations for the nonlinear convergence algorithm.

```
my_circuit = circuit()
my_circuit.simulate_t(duration=1e-6, nb=500, res=0.1, max_iter=50)
```

In certain cases, it may be necessary to initialize some variables before running the simulation. To do this, the `init()` function should be applied to the element object containing the variable, as shown in the `transient/oscillator` example:

```
my_circuit.NOT1_Uin.init( 5 )
```

Compact modeling

To illustrate compact modeling, let's consider the **NE555**, a relatively complex component, as an example.

Rather than modeling its behavior with all its transistors, which would lead to an extremely long simulation, it is often more efficient to use a compact behavioral description that consists of just a few equations. This is known as "compact modeling".

As an example, the **NE555**'s compact behavioral description includes just two subcomponents:

- a **SWITCH** component
- a **SRFFC** set-reset type flip-flop

plus a few logic equations. It all fits into just a few lines, as illustrated in the netlist below:

```
NAME.NE555( @trigger, @threshold, @discharge, @output, @vcc, @gnd ) :=
{
    NAME.VCC    = ACROSS( @vcc, @gnd );
    NAME.UTRIG  = ACROSS( @trigger, @gnd );
    NAME.UTRESH = ACROSS( @threshold, @gnd );
    NAME.UOUT   = ACROSS( @output, @gnd );
    NAME.UIN1   = ACROSS( NAME.@in1, @gnd );
    NAME.UIN2   = ACROSS( NAME.@in2, @gnd );

    NAME.UIN1 = ( NAME.UTRIG < ( NAME.VCC / 3 ) );
    NAME.UIN2 = ( NAME.UTRESH < ( NAME.VCC * 2 / 3 ) );

    NAME.SW.SWITCH( @discharge, @gnd, NAME.UOUT < (NAME.VCC/2) );
    NAME.SR.SRFFC( NAME.@in1, NAME.@in2, @output, @vcc, @gnd)
};
```

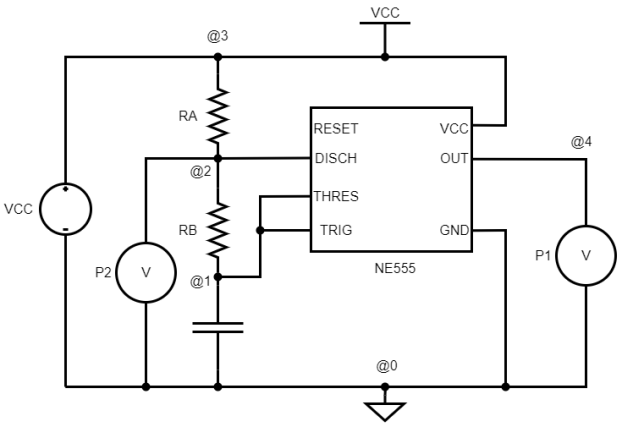
The **NE555** test circuit:

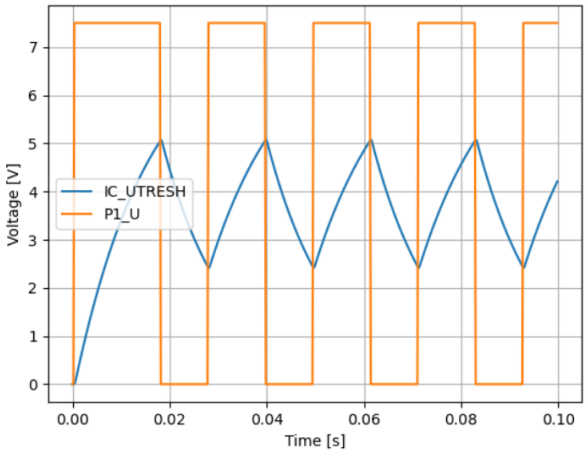
```
`include <components.rule>
`include <ic.rule>

test_NE555( R ) :=
{

  V1.CV    ( @3, @0, 7.5 );
  C1.CC    ( @1, @0, 1u );
  RB.CR    ( @2, @1, R);
  RA.CR    ( @3, @2, 2.7K );
  U.NE555  ( @1, @1, @2, @4, @3, @0  );
  P1.PROBE ( @4, @0 );
  P2.PROBE ( @2, @0 )
};
```

```
CIRCUIT := SOLVE_TRANS( test_NE555( _getv( _RV ) ) )
```

Note that the compact model contains sub-nodes `NAME.@in1` and `NAME.@in2`. As our **NE555** instance is called **U**, once instantiated, these two nodes will be called `U.@in1` and `U.@in2` respectively, and will not conflict with nodes external to the component.

Below is presented the result of a simulation with the **NE555** in an oscillator configuration:





This example shows that a compact behavioral description of a component gives a simulation result close to a real description, while reducing computation time.

The circuit shown in the example above contains probes **P1** and **P2**. These are used to visualize the voltages between two given nodes.

- **P1** is connected to the **NE555**'s **OUT** output and ground.
- **P2** and is connected to the **NE555**'s **DISH** input and ground.

The use of these probes is highly convenient, as they provide access to all the voltages within the circuit.

Conclusion

Despite its small executable size of roughly 100 kilobytes, *ASysC*, when combined with *Python* libraries like **Matplotlib** and **NumPy**, can perform both time-domain and frequency-domain simulations of components defined through algebraic expressions. These components can be structured hierarchically to form more complex systems and organized into reusable libraries.

Simulation results can be customized with **Matplotlib**, allowing them to be displayed on a single graph or on separate graphs with distinct scales.

ASysC provides an interesting alternative to more complex commercial simulators, offering significant flexibility for creation and experimentation. It is portable and compatible with both Linux and Windows.

While it may not match the performance of simulators like **Spice** and others, it still provides valuable capabilities. Since the simulation runs in *Python*, users can easily process the results, perform operations like Fourier transforms (**FFT**), and conduct measurements at specific points to identify optimal conditions. Additionally, users can run multiple simulations with different sets of random parameters, similar to a Monte Carlo approach. It also allows for direct modification of variables through *Python* code to observe how the system reacts.

Moreover, *ASysC* can be integrated with **CocoTB** [11], another *Python*-based tool designed to facilitate the testing of modules written in *VHDL* and *Verilog*. This integration allows for the simultaneous simulation of both analog and digital processes. The combination of these tools opens up exciting new possibilities, enabling the creation of advanced test benches for developers working with mixed analog/digital systems.

ASysC is still an experimental tool and lacks many essential features needed to transform it into a full professional tool.

References

- [1] Verilog-AMS: https://fr.wikipedia.org/wiki/Verilog-AMS
- [2] VHDL-AMS: https://fr.wikipedia.org/wiki/VHDL-AMS
- [3] ADMS: https://en.wikipedia.org/wiki/ADMS
- [4] Compact Modeling: https://www.amcad-engineering.com/transistor-model-extraction
- [5] Formal computation: https://en.wikipedia.org/wiki/Computer_algebra_system
- [6] Laplace transforms: https://en.wikipedia.org/wiki/Laplace_transform
- [7] Sparse matrices: https://en.wikipedia.org/wiki/Sparse_matrix
- [8] LU decomposition: https://qucs.sourceforge.net/docs/technical/technical.pdf §15.2.4
- [9] Sallen-Key filters: https://en.wikipedia.org/wiki/Sallen%E2%80%93Key_topology
- [10] Newton-Raphson: https://en.wikipedia.org/wiki/Newton%27s_method
- [11] CocoTB: https://www.cocotb.org/