ACME Documentation

Release 0.1.1

Martin Holters

August 30, 2016

Contents

ACME is a Julia package for the simulation of electrical circuits, focusing on audio effect circuits. It allows to programmatically describe a circuit in terms of elements and connections between them and then automatically derive a model for the circuit. The model can then be run on varying input data.

ACME is based on the method described in M. Holters, U. Zölzer, "A Generalized Method for the Derivation of Non-Linear State-Space Models from Circuit Schematics".

Contents:

CHAPTER 1

Getting Started

1.1 Installation

If you have not done so already, download and install Julia. (Any version starting with 0.3 should be fine.)

To install ACME, start Julia and run:

```
Pkg.add("ACME")
```

This will download ACME and all of its dependencies.

1.2 First Steps

We will demonstrate ACME by modeling a simple diode clipper. The first step is to load ACME:

```
using ACME
```

Now we create all the necessary circuit elements:

```
j_in = voltagesource()
r1 = resistor(1e3)
c1 = capacitor(47e-9)
d1 = diode(is=1e-15)
d2 = diode(is=1.8e-15)
j_out = voltageprobe()
```

Specifying a `voltagesource()` sets up a voltage source as an input, i.e. the voltage it sources will be specified when running the model. Alternatively, one can instantiate a constant voltage source for say 9V with `voltagesource(9)`. The `resistor` and `capacitor` calls take the resistance in ohm and the capacitance in farad, respectively, as arguments. For the `diode`, one may specify the saturation current `is` as done here and/or the emission coefficient η. Finally, desired outputs are denoted by adding probes to the circuit; in this case a `voltageprobe()` will provide voltage as output.

Next we need a `Circuit` instance to keep track of how the elements connect to each other:

```
circ = Circuit()
```

Connections can be specified by naming element pins that are connected:

```
connect!(circ, j_in["+"], r1[1])
```

This connects the positive output of the input voltage source with pin 1 of the resistor. Alternatively, one can introduce named nets to which element pins connect. This may increase readability for nets with many connected elements, like supply voltages. Here, we use it for the ground net where we connect the negative side of the input voltage:

```
connect!(circ, j_in["-"], :gnd)
```

One can also connect multiple pins at once:

```
connect!(circ, r1[2], c1[1], d1["+"], d2["-"], j_out["+"])
connect!(circ, :gnd, c1[2], d1["-"], d2["+"], j_out["-"])
```

Now that all connections have been set up, we need to turn the circuit description into a model. This could hardly be any easier:

```
model = DiscreteModel(circ, 1./44100)
```

The second argument specifies the sampling interval, the reciprocal of the sampling rate, here assumed to be the typical 44100 Hz.

Now we can process some input data. It has to be provided as a matrix with one row per input (just one in the example) and one column per sample. So for a sinusoid at 1 kHz lasting one second, we do:

```
y = run!(model, sin(2π*1000/44100*(0:44099).'))
```

The output `y` now likewise is a matrix with one row for the one probe we have added to the circuit and one column per sample.

More interesting circuits can be found in the examples located at `Pkg.dir("ACME/examples")`.

In the likely event that you would like to process real audio data, take a look at the WAV package for reading writing WAV files.

Note that the solver used to solve the non-linear equation when running the model saves solutions to use as starting points in the future. Model execution will therefore become faster after an initial learning phase. Nevertheless, ACME is at present more geared towards computing all the model matrices than to actually running the model. More complex circuits may run intolerably slow or fail to run altogether.

CHAPTER 2

User Guide

2.1 Element Creation

All circuit elements are created by calling corresponding functions; see the *Element Reference* for details.

2.2 Circuit Description

Circuits are described using `Circuit` instances, created with `Circuit()`. Once a `Circuit` and elements have been created, the elements can be added to the circuit using the `add!` method:

```
r = resistor(1e3)
c = capacitor(22e-9)
circ = Circuit()
add!(circ, r)
add!(circ, c)
```

Multiple elements can be added also be at once; the last two lines could have been replaced with `add!(circ, r, c)`.

In many cases, however, explicitly calling `add!` is not necessary. All that is needed is `connect!`, which connects two (or more) element pins. The elements to which these pins belong are automatically added to the circuit if needed. The only reason to explicitly call `add!` is to control the insertion order of sources or sinks, which determines the order in which inputs have to be provided and outputs are obtained.

Pins are obtained from elements using `[]`-style indexing, i.e. `r[1]` gives the first pin of the resistor defined above. So this connects the first pin of the resistor with the first pin of the capacitor:

```
connect!(circ, r[1], c[1])
```

Further connections involving the same pins are possible and will *not* replace existing ones. So this will effectively shorten the resistor, because now both of its pins are connected to `c[1]`:

```
connect!(circ, r[2], c[1])
```

Note that not all elements have numbered pins. For elements with polarity, they may be called + and −, while a bipolar transistor has pins `base`, `collector`, and `emitter`. The pins provided by each type of element are described in the *Element Reference*. Internally, the pin designators are `Symbol`s. However, not all symbols are conveniently entered in Julia: `:base` is nice, `symbol("1")` less so. Therefore, the `[]` operation on elements also accepts integers and strings and converts them to the respective `Symbol`s. So `r[symbol("1")]` is equivalent to `r[1]` and (assuming `d` to be a diode) `d[:+]` is equivalent to `d["+"]` (but `d[+]` does not work).

In addition to pins, `connect!` also accepts `Symbol`s as input. This creates named nets which may improve readability for nets with many conneted pins:

```
connect!(c[2], :gnd)
connect!(r[2], :gnd)
```

Again, this only adds connections, keeping existing ones, so together with the above snippets, now all pins are connected to each other and to net named `gnd`. It is even possible to connect multple named nets to each other, though this will only rarely be useful.

2.3 Model Creation and Use

A `Circuit` only stores elements and information about their connections. To simulate a circuit, a model has to be derived from it. This can be as simple as:

```
model = DiscreteModel(circ, 1/44100)
```

Here, `1/44100` denotes the sampling interval, i.e. the reciprocal of the sampling rate at which the model should run. Optionally, one can specify the solver to use for solving the model's non-linear equation as a type parameter:

```
model = DiscreteModel{HomotopySolver{SimpleSolver}}(circ, 1/44100)
```

See *Solvers* for more information about the available solvers.

Once a model is created, it can be run:

```
y = run!(model, u)
```

The input `u` is matrix with one row for each of the circuit's inputs and one column for each time step to simulate. Likewise, the output `y` will be a matrix with one row for each of the circuit's outputs and one column for each simulated time step. The order of the rows will correspond to the order in which the respective input and output elements were added to the `Circuit`. To simulate a circuit without inputs, a matrix with zero rows may be passed:

```
y = run!(model, zeros(0, 100))
```

The internal state of the model (e.g. capacitor charges) is preserved accross calls to `run!`. Initially, all states are zeroed. It is also possible to set the states to a steady state (if one can be found) with:

```
steadystate!(model)
```

This is often desirable for circuits were bias voltages are only slowly obtained after turning them on.

2.4 Solvers

2.4.1 `SimpleSolver`

The `SimpleSolver` is the simplest available solver. It uses Newton iteration which features fast local convergence, but makes no guarantees about global convergence. The initial solution of the iteration is obtained by extrapolating the last solution found (or another solution provided externally) using the available Jacobians. Due to the missing global convergence, the `SimpleSolver` is rarely useful as such.

2.4.2 HomotopySolver

The `HomotopySolver` extends an existing solver (provided as a type parameter) by applying homotopy to (at least theoretically) ensure global convergence. It can be combined with the `SimpleSolver` as `HomotopySolver{SimpleSolver}` to obtain a useful Newton homtopy solver with generally good convergence properties.

2.4.3 CachingSolver

The `CachingSolver` extends an existing solver (provided as a type parameter) by storing found solutions in a k-d tree to use as initial solutions in the future. Whenever the underlying solver needs more than a preset number of iterations (defaults to five), the solution will be stored. Storing new solutions is a relatively expensive operation, so until the stored solutions suffice to ensure convergence in few iterations throughout, use of a `CachingSolver` may actually slow things down.

The default solver used is a `HomotopySolver{CachingSolver{SimpleSolver}}`.

CHAPTER 3

Element Reference

3.1 Passives

resistor(*r*)
Creates a resistor obeying Ohm's law.

Parameters **r** – Resistance in Ohm

Pins: 1, 2

capacitor(*c*)
Creates a capacitor.

Parameters **c** – Capacitance in Farad

Pins: 1, 2

inductor(*l*)
Creates an inductor.

Parameters **l** – Inductance in Henri

Pins: 1, 2

inductor(*Val{:JA}; D*, *A*, *n*, *a*, α, *c*, *k*, *Ms*)
Creates a non-linear inductor based on the Jiles-Atherton model of magnetization assuming a toroidal core thin compared to its diameter.

Parameters

- **D** – Torus diameter (in meters)
- **A** – Torus cross-sectional area (in square-meters)
- **n** – Winding's number of turns
- **a** – Shape parameter of the anhysteretic magnetization curve (in Ampere-per-meter)
- α – Inter-domain coupling
- **c** – Ratio of the initial normal to the initial anhysteretic differential susceptibility
- **k** – amount of hysteresis (in Ampere-per-meter)
- **Ms** – saturation magnetization (in Ampere-per-meter)

A detailed discussion of the paramters can be found in D. C. Jiles and D. L. Atherton, "Theory of ferromagnetic hysteresis," J. Magn. Magn. Mater., vol. 61, no. 1–2, pp. 48–60, Sep. 1986 and J. H. B. Deane, "Modeling the

dynamics of nonlinear inductor circuits," IEEE Trans. Magn., vol. 30, no. 5, pp. 2795–2801, 1994, where the definition of `c` is taken from the latter.

Pins: `1`, `2`

`transformer` (*l1, l2; [coupling_coefficient=1,] [mutual_coupling=coupling_coefficient*sqrt(l1*l2)]*)
Creates a transformer with two windings having inductances.

Parameters

- **`l1`** – Primary self-inductance in Henri
- **`l2`** – Secondary self-inductance in Henri
- **`coupling_coefficient`** – Coupling coefficient (0 is not coupled, 1 is closely coupled)
- **`mutual_coupling`** – Mutual inductance in Henri; overrides `coupling_coefficient` if both are given

Pins: `1` and `2` for primary winding, `3` and `4` for secondary winding

`transformer` (*Val{:JA}; D, A, ns, a, α, c, k, Ms*)
Creates a non-linear transformer based on the Jiles-Atherton model of magnetization assuming a toroidal core thin compared to its diameter.

Parameters

- **`D`** – Torus diameter (in meters)
- **`A`** – Torus cross-sectional area (in square-meters)
- **`ns`** – Windings' number of turns as a vector with one entry per winding
- **`a`** – Shape parameter of the anhysteretic magnetization curve (in Ampere-per-meter)
- α – Inter-domain coupling
- **`c`** – Ratio of the initial normal to the initial anhysteretic differential susceptibility
- **`k`** – amount of hysteresis (in Ampere-per-meter)
- **`Ms`** – saturation magnetization (in Ampere-per-meter)

A detailed discussion of the parameters can be found in D. C. Jiles and D. L. Atherton, "Theory of ferromagnetic hysteresis," J. Magn. Magn. Mater., vol. 61, no. 1–2, pp. 48–60, Sep. 1986 and J. H. B. Deane, "Modeling the dynamics of nonlinear inductor circuits," IEEE Trans. Magn., vol. 30, no. 5, pp. 2795–2801, 1994, where the definition of `c` is taken from the latter.

Pins: `1` and `2` for primary winding, `3` and `4` for secondary winding, and so on

3.2 Independent Sources

`voltagesource` (*[v]*)
Creates a voltage source.

Parameters **`v`** – Source voltage in Volt. If omitted, the source voltage will be an input of the circuit.

Pins: + and − with `v` being measured from + to −

`currentsource` (*[i]*)
Creates a current source.

Parameters **`i`** – Source current in Ampere. If omitted, the source current will be an input of the circuit.

Pins: + and − where `i` measures the current leaving source at the + pin

3.3 Probes

`voltageprobe``()`
Creates a voltage probe, provding the measured voltage as a circuit output.

Pins: + and − with the output voltage being measured from + to −

`currentprobe``()`
Creates a current probe, provding the measured current as a circuit output.

Pins: + and − with the output current being the current entering the probe at +

3.4 Semiconductors

`diode``(;[is=1e-12,] [`η *= 1]*`)`
Creates a diode obeying Shockley's law $i = I_S \cdot (e^{v/(\eta v_T)} - 1)$ where v_T is fixed at 25 mV.

Parameters

- **`is`** – Reverse saturation current in Ampere
- η – Emission coefficient

`bjt``(`*typ; is=1e-12,* η*=1,* *isc=is,* *ise=is,* η*c=*η*,* η*e=*η*,* β*f=1000,* β*r=10*`)`
Creates a bipolar junction transistor obeying the Ebers-Moll equation

$$i_E = I_{S,E} \cdot (e^{v_E/(\eta_E v_T)} - 1) - \frac{\beta_r}{1+\beta_r} I_{S,C} \cdot (e^{v_C/(\eta_C v_T)} - 1)$$

$$i_C = -\frac{\beta_f}{1+\beta_f} I_{S,E} \cdot (e^{v_E/(\eta_E v_T)} - 1) + I_{S,C} \cdot (e^{v_C/(\eta_C v_T)} - 1)$$

where v_T is fixed at 25 mV.

Parameters

- **`typ`** – Either `:npn` or `:pnp`, depending on desired transistor type
- **`is`** – Reverse saturation current in Ampere
- η – Emission coefficient
- **`isc`** – Collector reverse saturation current in Ampere (overriding `is`)
- **`ise`** – Emitter reverse saturation current in Ampere (overriding `is`)
- η**`c`** – Collector emission coefficient (overriding η)
- η**`e`** – Emitter emission coefficient (overriding η)
- β**`f`** – Forward current gain
- β**`r`** – Reverse current gain

3.5 Integrated Circuits

opamp()
Creates an ideal operational amplifier. It enforces the voltage between the input pins to be zero without sourcing any current while sourcing arbitrary current on the output pins wihtout restricting their voltage.

Note that the opamp has two output pins, one of which will typically be connected to a ground node and has to provide the current sourced on the other output pin.

Pins: `in+` and `in-` for input, `out+` and `out-` for output

opamp(*Val{:macak}*, *gain*, *vomin*, *vomax*)
Creates a clipping operational amplifier where input and output voltage are related by

$$v_{\text{out}} = \frac{1}{2} \cdot (v_{\text{max}} + v_{\text{min}}) + \frac{1}{2} \cdot (v_{\text{max}} - v_{\text{min}}) \cdot \tanh\left(\frac{g}{\frac{1}{2} \cdot (v_{\text{max}} - v_{\text{min}})} \cdot v_{\text{in}} \right).$$

The input current is zero, the output current is arbitrary.

Note that the opamp has two output pins, one of which will typically be connected to a ground node and has to provide the current sourced on the other output pin.

Pins: `in+` and `in-` for input, `out+` and `out-` for output

B

C

D

I

O

R

T

V