ahkab Documentation

Release 0.18

Giuseppe Venturini

July 12, 2015

Contents

1 General help pages **3**

1.1 Why *ahkab* . . . 3
1.2 Installing ahkab . . . 5
1.3 Command line help . . . 7
1.4 Netlist Syntax . . . 8

2 Getting started: examples and tutorials **27**

2.1 Simulating from Python . . . 27
2.2 Simulating from the command line . . . 43

3 Help pages on particular elements **65**

3.1 Mutual Inductors . . . 65

4 Module reference **69**

4.1 The ahkab core module . . . 69
4.2 ahkab.ac . . . 77
4.3 ahkab.bfpss . . . 79
4.4 ahkab.circuit . . . 80
4.5 ahkab.constants . . . 91
4.6 ahkab.csvlib . . . 92
4.7 ahkab.dc_analysis . . . 94
4.8 ahkab.dc_guess . . . 100
4.9 ahkab.devices . . . 101
4.10 ahkab.diode . . . 110
4.11 ahkab.ekv . . . 111
4.12 ahkab.fourier . . . 115
4.13 ahkab.gear . . . 118
4.14 ahkab.implicit_euler . . . 119
4.15 ahkab.mosq . . . 121
4.16 ahkab.netlist_parser . . . 126
4.17 ahkab.options . . . 134
4.18 ahkab.plotting . . . 137
4.19 ahkab.printing . . . 138
4.20 ahkab.pss . . . 142
4.21 ahkab.pz . . . 142
4.22 ahkab.results . . . 144
4.23 ahkab.shooting . . . 152

4.24 ahkab.switch 153
4.25 ahkab.symbolic 156
4.26 ahkab.testing 158
4.27 ahkab.ticker 164
4.28 ahkab.time_functions 165
4.29 ahkab.transient 170
4.30 ahkab.trap 173
4.31 ahkab.utilities 174

5 License **181**
5.1 GNU General Public License 181

6 Indices and tables **187**

Python Module Index **189**

Release 0.18

Date July 12, 2015

Ahkab (pronounced “uh, cab”) is an open-source SPICE-like interactive circuit simulator.

CHAPTER 1

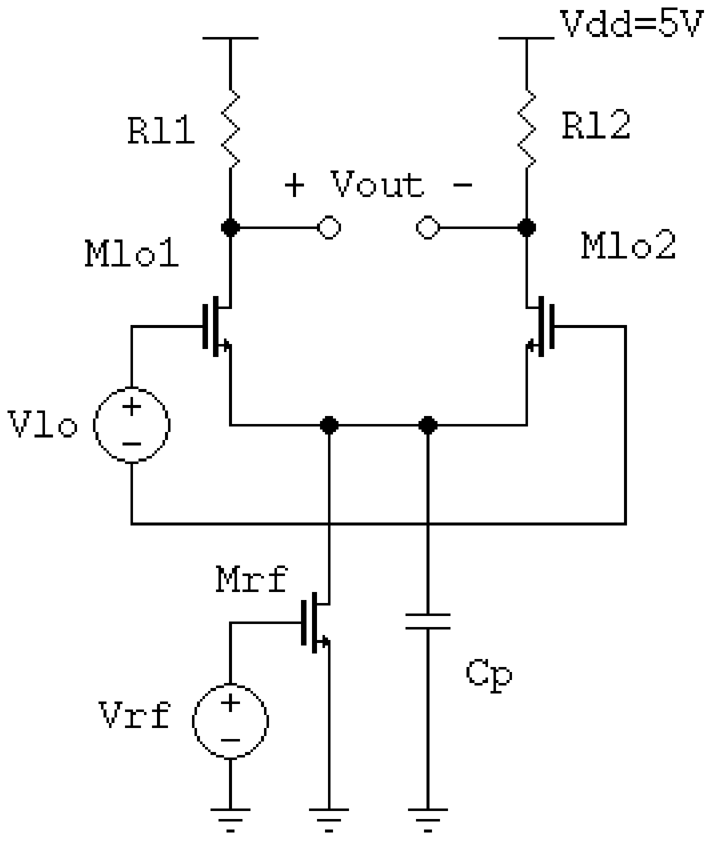
General help pages

1.1 Why *ahkab*

A rather long winded explanation as to why this project exists today, you may skip directly to *Installing ahkab*.

1.1.1 Do circuit simulators dim your wit?

A young engineer begins to design a well-known, basic circuit block. He knows his stuff and he's prepared: he did his homework on pen and paper first and now he wants to take care of schematic entry, checking that the circuit indeed works as expected, in case introducing slight adjustments as needed.



He draws the circuit and, without even thinking about it, he clicks *Simulate*.

The results are surprising to him, something he has not taken into account has influenced his simulation and the numbers are slightly off. Or maybe the schematic was drawn in a rush?

He starts fiddling with the design parameters. Make some transistors bigger. Now, make a few others smaller. *Shouldn't this fix it?* - he wonders. *Maybe*, but the aspect ratios that were just changed almost at random were initially selected to reach multiple results at the same time.

Now, the design does not meet, not one, but multiple specs. He removes parts that are secondary for the current simulation. Then proceeds to change the simulation itself...

What's left of a good idea

If a fellow design engineer were to look at his screen now, he'd see a horrifying circuit: aspect ratios all over the place, cheap hacks to make up for the "secondary" parts that were removed in order to simplify the circuit, quantities that make no sense in the physical world.

After going down the road of compulsive circuit simulation, little is left of the initial, promising design and our young engineer feels lost, frustrated and he's metaphorically about to hitting his head against the keyboard of his workstation.

Where's the fun of doing circuit design this way?

In the end

We have seen the above before... and that's not how our beloved microelectronic work should be.

Circuit simulators are just a tool. An insidious one at it, as we may naively use them instead of our gray matter, blindly trusting our models or giving in to the temptation of manual, broken "optimization" fiddling, rather than as a verification tool, area where they truly excel.

It doesn't help much that often neither it is straightforward to debug a simulation, nor it is clear what exactly the simulator is doing.

Which brings us to:

1.1.2 What we try to do here with `ahkab`

The `ahkab` circuit simulator is an experiment.

We have no expectation that our proof-of-concept, sometimes buggy, small circuit simulation tool will be replacing the mainstream circuit simulators: they are mainstream for good reasons and they very much deserve the praise and money we pay. It would be foolish to think otherwise.

1. Peek under the hood

But we do still think we have our own place: what we wish to do with `ahkab` is to allow the user, the designer, to peek behind the veil and see more clearly what goes on with his simulations.

For this reason, `ahkab` supports operations such as printing out all equation matrices.

It is also written in a scripted, interpreted language (Python) that, while requiring us to sacrifice raw speed, should make it relatively easy to see what's going on behind the hood.

And the algorithms are there for you to see, inspect and, if need be, correct: too often scientific papers about software come with no available implementation or the source is not distributed: with `ahkab` all code is available under a copy-left license, allowing you to benefit of the code, modify it and giving others the same freedom.

2. Experiment

With `ahkab`, you are welcome to implement an algorithm you have read about in a paper, if you are so inclined: we believe no lecture, no matter how in-depth, will provide an insight in a circuit simulation algorithm as deep as rolling up your sleeves and trying to code it up yourself. (*please remember talk to us well in advance if you expect us to include your work!*)

3. Have fun doing electronics!

All in all, we hope this little project helps you understand better what will goes on in your circuit when you implement it, when you simulate it and especially *we wish you have fun while doing so!*

1.2 Installing ahkab

1.2.1 Requirements

The program requires:

- the **Python 2** or **Python 3** interpreter (at least v.2.6 for Python2, at least v.3.3 for Python3),
- **numpy>=1.7.0**,
- **scipy>=0.14.0**,
- **sympy>=0.7.6**,
- and **tabulate>0.7.3**.

Strongly recommended:

- **matplotlib>=1.1.1**,
- **nose** for running the test suite.

Please try to use an up-to-date version of the libraries instead of the bare minimum required.

All platform that are supported by the dependencies are also platforms supported by `ahkab`, although the author only runs *UNIX variants. If you run into any problem, please report it in the issue manager.

Numpy and Scipy are needed for all the numeric computations. On a Debian system, Python, Numpy and Scipy may be installed running:

```
# aptitude install python python-numpy python-scipy
```

The symbolic analysis capabilities rely on the amazing sympy. Any version of sympy will do if you are interested only in numeric simulations, but, if you run symbolic simulations, *sympy version 0.7.6 or higher* is needed.

```
# aptitude install python-sympy
```

Plotting requires matplotlib:

```
# aptitude install python-matplotlib
```

1.2.2 Install

The source code for the project is hosted on GitHub and releases can be found on PyPI.

To install `ahkab`, you can have two options: using `pip` or using `distutils`.

Install with pip

If you use `pip`, which boundled in your Python installation, the source code is downloaded from you off the Python Package Index (PyPI).

You may:

- Issue `pip install ahkab`, which may require administrative access depending on what permissions your user has on your Python installation.
- To avoid having to supply admin credentials, you may use `pip` according to "the user scheme", issuing `pip install ahkab --user`.

Install with distutils

Installing manually through `distutils` requires that you download the source code, untar and move to the root directory of the package.

For which you should first either:

- download a release tarball containing the source code,
- or check out the latest code as explained on GitHub.

Then, you will need to install the module manually with the `distutils` script `setup.py` provided, you can cho

- to install for all users: `python setup.py install`
- or only your own: `python setup.py install --user`
- or to install to a different prefix: `python setup.py install --prefix=~/.local`

The Python documentation for installing with `distutils` will clear up any remaining doubt: for version 2 of the language, for version 3.

1.2.3 Thanks

Many thanks to the developers of the above libraries, their effort made this project possible. :)

1.3 Command line help

The `ahkab` simulator has a command line interface that allows for quick simulation of netlist decks, without the need to load the Python interpreter explicitely.

Several switches are available, to set the input and output files and to override some built-in options.

Notice that options set on the command line always take precedence on any netlist option or any value set in *`ahkab.options`*.

1.3.1 Usage:

```
ahkab [options] <filename>
```

The filename is the netlist to be open. Use - (a dash) to read from stdin.

1.3.2 Options:

--version	show program's version number and exit
-h, --help	show this help message and exit
-v VERBOSE, --verbose=VERBOSE	Verbose level: from 0 (almost silent) to 5 (debug)
-p, --print	Print the parsed circuit
-o OUTFILE, --outfile=OUTFILE	Data output file. Defaults to stdout.
--dc-guess=DC_GUESS	Guess to be used to start a OP or DC analysis: none or guess. Defaults to guess.
-t METHOD, --tran-method=METHOD	Method to be used in transient analysis: implicit_euler, trap, gear2, gear3, gear4, gear5 or gear6. Defaults to TRAP.
--t-fixed-step	Disables the step control in transient analysis.
--v-absolute-tolerance=VEA	Voltage absolute tolerance. Default: 1e-06 V
--v-relative-tolerance=VER	Voltage relative tolerance. Default: 0.001
--i-absolute-tolerance=IEA	Current absolute tolerance. Default: 1e-09 A
--i-relative-tolerance=IER	Current relative tolerance. Default: 0.001
--h-min=HMIN	Minimum time step. Default: 1e-20
--dc-max-nr=DC_MAX_NR_ITER	Maximum number of NR iterations for DC and OP analyses. Default: 10000
--t-max-nr=TRANSIENT_MAX_NR_ITER	Maximum number of NR iterations for each time step during transient analysis. Default: 20
--t-max-time=TRANSIENT_MAX_TIME_ITER	Maximum number of time iterations during transient analysis. Setting it to 0 (zero) disables the limit. Default: 0

--s-max-nr=SHOOTING_MAX_NR_ITER Maximum number of NR iterations during shooting analysis. Setting it to 0 (zero) disables the limit. Default: 10000

--gmin=GMIN The minimum conductance to ground. Inserted when requested. Default: 1e-12

--cmin=CMIN The minimum capacitance to ground. Default: 1e-18

1.4 Netlist Syntax

This document describes the syntax to be used to describe a circuit and its relative analyses.

Table of Contents

- *Netlist Syntax*
  - *The netlist file*
  - *Title*
  - *Elements*
    - *Linear elements*
      - *Resistors*
      - *Capacitors*
      - *Inductors*
      - *Mutual Inductors*
      - *Voltage-controlled switch*
    - *Independent sources*
      - *Voltage source*
      - *Current source*
    - *Dependent sources*
      - *Voltage-Controlled Voltage Source (VCVS)*
      - *Voltage-Controlled Current Source (VCCS)*
      - *Current-Controlled Current Source (CCCS)*
    - *Non-linear elements*
      - *Diode*
      - *MOS Transistors*
      - *User-defined elements*
    - *Subcircuit calls*
  - *Time functions*
    - *Sinusoidal waveform*
    - *Exponential source*
    - *Pulsed source*
    - *Single-Frequency Frequency Modulation (SFFM)*
    - *Amplitude Modulation (AM)*
  - *Device models*
    - *Rudimentary EKV 3.0 MOS model*
    - *Square-law MOS model*
    - *DIODE model*
    - *TANH(x)-shaped switch model*
  - *Analyses*
    - *Operating point (.OP)*
    - *DC analysis (.DC)*
    - *Transient analysis (.TRAN)*
    - *AC analysis (.AC)*
    - *Periodic Steady State (.PSS)*
    - *Pole-Zero analysis (.PZ)*
    - *Symbolic small-signal (.SYMBOLIC)*
  - *Post-processing*
    - *.Plot*
    - *.Four*
    - *.FFT*
  - *Other directives*
    - *End*
    - *Ends*
    - *Ic*
    - *Include*
    - *Subckt*

1.4.1 The netlist file

Circuits are described in text files called *netlists* (or sometimes 'decks').

Each line in a netlist file falls in one of these categories:

- The title.
- A element declaration.
- A analysis declaration.
- A directive declaration (e.g. `.ic` or `.end`).
- A comment. Comments start with `*`.
- A continuation line. Continuation lines start with `+`.
- Blank line (ignored).

1.4.2 Title

The title is a special type of comment and it is **always the first line in the file**. *Do not put any other directive here, it will be silently ignored.*

1.4.3 Elements

In general, an element is declared with the following general syntax:

```
<K><description_string> <n1> <n2> [value] [<option>=<value>] [...]
...
```

Where:

- `<K>` is a character, a unique identifier for each type of element (e.g. R for resistor).
- `<description_string>` is a string without spaces (e.g. `1`).
- `<n1>`, a string, is the node of the circuit to which the anode of the element is connected.
- `<n2>`, a string, is the node of the circuit to which the cathode of the element is connected.
- `[value]` if supported, is the 'value' of the element, in mks (e.g. `R1 1 0 500k`)
- `<option>=<value>` are the parameters of the element.

Nodes may have any label, without spaces, except the *reference node* (GND) which has to be `0`.

Linear elements

Resistors

General syntax:

```
R<string> n1 n2 <value>
```

- `n1` and `n2` are the element nodes.
- `value` is the element resistance. It may any non-zero value (negative values are supported too).

Example:

```
R1 1 0 1k
RAb_ input output 1.2e6
```

Capacitors

General syntax:

`C<string> n1 n2 <value> [ic=<value>]`

- `n1` and `n2` are the element nodes.
- `value` is the capacitance in Farads.
- `ic=<value>` is an optional attribute that can be set to provide an initial value voltage value for a transient simulation. See also the discussion of the `UIC` parameter in TRAN simulations.

Example:

```
C1 1 0 1u
Cfeedback out+ in- 1e6
```

Inductors

General syntax:

`L<string> n1 n2 <value> [ic=<float>]`

- `n1` and `n2` are the element nodes.
- `value` is the inductance in Henry.
- `ic=<value>` is an optional attribute that can be set to provide an initial value for a transient simulation. See also the discussion of the `UIC` parameter in TRAN simulations.

Example:

```
L1 1 0 1u
Lchoke inA inB 1e6
```

Mutual Inductors

General syntax:

Either:

`K<string> <inductor1> <inductor2> <value>`

or

`K<string> <inductor1> <inductor2> k=<value>`

- `<inductor1>` and `<inductor2>` are the coupled inductors. They need to be specified before the coupling can be inserted.
- `value` is the coupling factor, `k`. It is a needs to be less than 1.

Example:

```
L1 1 0 1u
L2 3 4 5u
K1 L1 L2 0.6
```

Voltage-controlled switch

General syntax:

`S<string> n1 n2 ns1 ns2 <model_id>`

- `n1` and `n2` are the nodes corresponding to the output port, where the switch opens and closes the connection.
- `ns1` and `ns2` are the nodes corresponding to the driving port, where the voltage setting the switch status is read.
- `model_id` is the model describing the switch operation. Notice that even if an ideal switch is a (piece-wise) linear element, its model implementation may not be, depending on the implementation details of the transition region.

Independent sources

Voltage source

General syntax:

`v<string> n1 n2 [type=vdc vdc=float] [type=vac vac=float] [type=....]`

Where the third type (if added) is one of: `sin`, `pulse`, `exp`, `sffm`, `am`.

Current source

General syntax:

`i<string> n1 n2 [type=idc idc=float] [type=iac iac=float] [type=....]`

The declaration of the time variant part is the same as for voltage sources, except that `vo` becomes `io`, `va` becomes `ia` and so on.

Dependent sources

Voltage-Controlled Voltage Source (VCVS)

General syntax:

`E<string> n+ n- ns+ ns- <value>`

- `n+` and `n-` are the nodes corresponding to the output port, where the voltage is forced.
- `ns+` and `ns-` are the nodes corresponding to the driving port, where the voltage is read.

- `value` is the proportionality factor, i.e.: `V(n+) - V(n-) = value*[V(sn+) - V(sn-)]`.

Voltage-Controlled Current Source (VCCS)

General syntax:

`G<string> n+ n- ns+ ns- <value>`

- `n+` and `n-` are the nodes corresponding to the output port, where the current is forced.
- `ns+` and `ns-` are the nodes corresponding to the driving port, where the voltage is read.
- `value` is the proportionality factor, i.e.: `I(n+,n-) = value*[V(sn+) - V(sn-)]`.

Current-Controlled Current Source (CCCS)

General syntax:

`F<string> n+ n- <voltage_source> <value>`

- `n+` and `n-` are the nodes corresponding to the output port, where the current is forced.
- `voltage_source` is the ID of a voltage source whose current controls the dependent current source. It must exist in the circuit. Note that netlists are case-insensitive, i.e. `Va` is the same as `vA`.
- `value` is the proportionality factor, i.e.: $I(n+, n-) = value * I[< voltage_source >]$.

Non-linear elements

The simulator has a few non-linear components built-in. Others may easily be added as external modules.

Diode

General syntax:

`D<string> n1 n2 <model_id> [<AREA=float> <T=float> <IC=float> <OFF=boolean>]`

Parameters:

- `n1`: anode.
- `n2`: cathode.
- `<model_id>`: the ID of the diode model.
- `AREA`: The area of the PN junction.
- `T`: the temperature of operation, if different from the circuit temperature.
- `IC`: initial condition statement (voltage).
- `OFF`: Consider the diode to be initially off in transient analyses.

MOS Transistors

General syntax:

`M<string> nd ng ns nb <model_id> w=<float> l=<float>`

A MOS device declaration requires:

- `nd`: the drain node,
- `ng`: the gate node,
- `ns`: the source node,
- `nb`: the bulk node.
- `<model_id>`: is a string that links this device to a `.model` declaration in the netlist. The model is actually responsible of the operation of the device.
- `w`: gate width, in meters.
- `l`: gate length, in meters.

User-defined elements

General syntax:

`Y<X> <n1> <n2> module=<module_name> type=<type> [<param1>=<value1> ...]`

Ahkab can parse user-defined elements. In order for this to work, you should write a Python module that supplies the element class. The simulator will attempt to load the module `<module_name>` and it will then look for a class named `<type>` within.

See `netlist_parser.parse_elem_user_defined()` for further information.

Subcircuit calls

General syntax:

`X<string> name=<subckt_label> [<subckt_node1>=<node_a> <subckt_node2>=<node_b> ...  ]`

Insert a subcircuit, connected as specified.

All nodes in the subcircuit specification must be connected to a circuit node. The call can be placed before or after the corresponding .subckt directive.

1.4.4 Time functions

Time functions may be used in conjunction with an independent source to define its time-dependent behavior.

This is typically done adding a `type=...` section in the element declaration, such as:

```
V1 1 2 vdc=10m type=sin VO=10m VA=1.2 FREQ=500k TD=1n THETA=0
```

Sinusoidal waveform

A damped sinusoidal time function.

It may be described with the syntax:

```
type=sin <VO> <VA> <FREQ> <TD> <THETA> <PHASE>
```

or with the more verbose variant:

```
type=sin VO=<float> VA=<float> FREQ=<float> TD=<float> THETA=<float> PHASE=<float>
```

Mathematically described by:

- When $t < td$:

$$V(t) = VO$$

- When $t \geq td$:

$$V(t) = VO + VA \cdot \exp[-THETA \cdot (t - TD)] \cdot \sin[2\pi FREQ(t - TD) + (PHASE/360)]$$

Where:

- VO is the offset voltage in Volt.
- VA is the amplitude in Volt.
- $FREQ$ is the frequency in Hertz.
- TD is the delay in seconds.
- $THETA$ is the damping factor per second.
- $PHASE$ is the phase in degrees.

Exponential source

An exponential waveform may be described with one of the following syntaxes:

```
type=EXP <V1> <V2> <TD1> <TAU1> [<TD2> <TAU2>]
```

```
type=exp v1=<float> v2=float td1=float tau1=<float> td2=<float> tau2=<float>
```

Example:

```
VIN input 0 type=vdc vdc=0 type=exp 4 1 2n 30n 60n 40n
```

Mathematically, it is described by the equations:

- $0 \leq t < TD1$:

$$f(t) = V1$$

- $TD1 < t < TD2$

$$f(t) = V1 + (V2 - V1) \cdot \left[1 - \exp\left(-\frac{t - TD1}{TAU1}\right)\right]$$

- $t > TD2$

$$f(t) = V1 + (V2 - V1) \cdot \left[1 - \exp\left(-\frac{t - TD1}{TAU1}\right)\right] + (V1 - V2) \cdot \left[1 - \exp\left(-\frac{t - TD2}{TAU2}\right)\right]$$

Parameters:

Parameter	Meaning	Default value	Units
V1	initial value		V or A
V2	pulsed value		V or A
TD1	rise delay time	0.0	s
TAU1	rise time constant		s
TD2	fall delay time	Infinity	s
TAU2	fall time constant	Infinity	s

Pulsed source

A square wave.

```
type=pulse v1=<float> v2=<float> td=<float> tr=<float> tf=<float> pw=<float> per=<float
```

or:

```
PULSE <V1> <V2> <TD> <TR> <TF> <PW> <PER>
```

Parameters:

Parameter	Meaning	Default value	Units
V1	first value		V or A
V2	second value		V or A
TD	delay time	0.0	s
TR	rise time		s
TF	fall time		s
PW	pulse width		s
PER	periodicity interval		s

Single-Frequency Frequency Modulation (SFFM)

A SFFM wave.

It may be described with any of the following syntaxes:

```
TYPE=sffm <VO> <VA> <FC> <MDI> <FS> [<TD>]
```

or

```
type=sffm vo=<float> v=<float> f=<float> md=<float> f=<float> +
[td=<float>]
```

Mathematically, it is described by the equations:

- $0 \leq t \leq t_D$:

$$f(t) = V_O$$

- $t > t_D$

$$f(t) = V_O + V_A \cdot \sin\left[2\pi f_C (t - t_D) + MDI \sin\left[2\pi f_S (t - t_D)\right]\right]$$

Parameters:

Parameter	Meaning	Default value	Units
VO	offset		V or A
VA	amplitude		V or A
FC	carrier frequency		Hz
MDI	modulation index		
FS	signal frequency		HZ
TD	time delay	0.0	s

Amplitude Modulation (AM)

An AM waveform.

It may be described with any of the following syntaxes:

```
TYPE=AM <SA> <OC> <FM> <FC> [<TD>]
```

or

```
type=am sa=<float> oc=<float> fm=<float> fc=<float> [td=<float>]
```

Mathematically, it is described by the equations:

- $0 \leq t \leq t_D$:

$$f(t) = O$$

- $t > t_D$

$$f(t) = SA \cdot \left[OC + \sin\left[2\pi f_m (t - t_D)\right]\right] \cdot \sin\left[2\pi f_c (t - t_D)\right]$$

Parameters:

Parameter	Meaning	Default value	Units
SA	amplitude		V or A
FC	carrier frequency		Hz
FM	modulation frequency		Hz
OC	offset constant		
TD	time delay	0.0	s

1.4.5 Device models

Rudimentary EKV 3.0 MOS model

General syntax:

```
.model ekv <model_id> TYPE=<n/p> [TNOM=<float> COX=<float>
GAMMA=<float> NSUB=<float> PHI=<float> VTO=<float> KP=<float>
TOX=<float> VFB=<float> U0=<float> TCV=<float> BEX=<float>]
```

The EKV model was developed by Matthias Bucher, Christophe Lallement, Christian Enz, Fabien Théodoloz, François Krummenacher at the Electronics Laboratories, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland.

It is described here:

- rev. 2.6 - http://legwww.epfl.ch/ekv/pdf/ekv_v262.pdf
- rev. 3.0 - http://www.nsti.org/publications/MSM/2002/pdf/346.pdf

The authors are in no way responsible for any bug that may be present in my implementation. :)

The model is missing:

- channel length modulation,
- complex mobility reduction,
- RSCE transcapacitances,
- the quasistatic modeling.

It does identify weak, moderate and strong inversion zones, it is fully symmetrical, it treats N and P devices equally.

Square-law MOS model

General syntax:

```
.model mosq <model_id> TYPE=<n/p> [TNOM=<float> COX=<float>
GAMMA=<float> NSUB=<float> PHI=<float> VTO=<float> KP=<float>
TOX=<float> VFB=<float> U0=<float> TCV=<float> BEX=<float>]
```

This is a square-law MOS model without velocity saturation (and second order effects like punch-through and such).

DIODE model

General syntax:

```
.model diode <model_id> [IS=<float> N=<float> ISR=<float> NR=<float>
RS=<float> CJ0=<float> M=<float> VJ=<float> FC=<float> CP=<float>
TT=<float> BV=<float> IBV=<float> KF=<float> AF=<float> FFE=<float>
TEMP=<float> XTI=<float> EG=<float> TBV=<float> TRS=<float>
TTT1=<float> TTT2=<float> TM1=<float> TM2=<float>]
```

The diode model implements the Shockley diode equation. Currently the capacitance modeling part is missing.

The most important parameters are:

Parameter	*Default value*	*Description*
IS	1e-14 A	Specific current
N	1.0	Emission coefficient
ISR	0.0 A	Recombination current
NR	2.0	Recombination coefficient
RS	0.0 ohm	Series resistance per unit area

Please refer to the SPICE documentation and the `diode.py` file for the others.

TANH(x)-shaped switch model

General syntax:

There are two possible syntax:

```
.model SW <model_id> VT=<float> VH=<float> RON=<float> ROFF=<float>
```

```
.model SW <model_id> VON=<float> VOFF=<float> RON=<float>
ROFF=<float>
```

This model implements a voltage-controlled switch where the transition is modeled with $tanh(x)$.

Hysteresis is supported through the parameter `VH`. When set, the two thresholds become `VT+VH` and `VT-VH` (distance `2*VH`!).

When `VON` and `VOFF` are specified instead of `VT` and `VH`, the latter two are set from the former according to the relationships:

- `VT = (VON-VOFF)/2 + VOFF`
- `VH = 1e-3*VT`

Parameters and default values:

Parameter	*Default value*	*Description*	*Restrictions*
VT	0 V	Threshold voltage	
VH	0 V	Hysteresis voltage	Must be positive
RON	1 ohm	ON-state resistance	Must be non-zero
ROFF	1/gmin	OFF-state resistance	Must be non-zero

1.4.6 Analyses

Operating point (.OP)

General syntax:

```
.op [guess=<ic_label>]
```

This analysis tries to find a DC solution through a pseudo Newton Rhapson (NR) iteration method. Notice that a non-linear circuit may have zero, a discrete number or infinite OPs.

Which one is found depends on the circuit and on the initial guess supplied to the method. The program has a built in method that tries to generate a “smart” initial guess to speed up convergence. When that fails, or is disabled from command line (see –help), the initial guess is set to all zeros.

The user may supply a better guess, if known. This can be done adding a .ic directive somewhere in the netlist file and setting `guess=<ic_label>` where `<ic_label>` matches the .ic’s `name=<ic_label>`.

The `t = 0` value is automatically added as DC value to every time-variant independent source without a explicit DC value.

DC analysis (.DC)

General syntax:

```
.DC src=<src_name> start=<float> stop=<float> step=<float>
type=<lin/log>
```

Performs a DC sweep (repeated OP analysis with the value of a voltage or current source changing at every iteration).

Parameters:

- **`src`: the id of the source to be swept (V12, Ibias...).** Only independent current and voltage sources.
- `start` and `stop`: sweep start and stop values.
- `type`: either `lin` or `log`
- **step: sets the value of the source from an iteration (k) to the next $(k+1)$:**
  - if `type=log`, $S(k+1) = S(k) \cdot step$
  - if `type=lin`, $S(k+1) = S(k) + step$

Transient analysis (.TRAN)

General syntax:

```
.TRAN TSTEP=<float> TSTOP=<float> [TSTART=<float> UIC=0/1/2/3
[IC_LABEL=<string>] METHOD=<string>]
```

Performs a transient analysis from `tstart` (which defaults to 0) to `tstop`, using the step provided as initial step and the method specified (if any, otherwise defaults to implicit Euler).

Parameters:

- `tstart`: the starting point, defaults to zero.
- `tstep`: this is the initial step. By default, the program will try to adjust it to keep the estimate error within bounds.
- `tstop`: Stop time.
- `UIC` (Use Initial Conditions): This is used to specify the state of the circuit at time `t = tstart`. Available values are `0`, `1`, `2` or `3`.
- `uic=0`: all node voltages and currents through v/h/e/sources will be assumed to be zero at `t = tstart`
- `uic=1`: the status at ‘t = tstart is the last result from a OP analysis.
- `uic=2`: the status at t=tstart is the last result from a OP analysis on which are set the values of currents through inductors and voltages on capacitors specified in their ic. This is done very roughly, checking is recommended.
- `uic=3`: Load a user supplied ic. This requires a `.ic` directive somewhere in the netlist and a `.ic`‘s name and `ic_label` must match.

- method: the integration method to be used in transient analysis. Built-in methods are: `implicit_euler`, `trap`, `gear2`, `gear3`, `gear4`, `gear5` and `gear6`. Defaults to `trap`. May be overridden by the value specified on the command line with the option: `-t METHOD` or `--tran-method=METHOD`.

High order methods are slower per iteration, but they often can afford a longer step with comparable error, hence they are actually faster in many cases.

If a transient analysis stops because of a step size too small, use a low order method (ie/trap) and set `--t-max-nr` to a high value (eg 1000).

AC analysis (.AC)

General syntax:

Either:

```
.AC <lin/log> <npoints> <start> <stop>
```

or:

```
.AC start=<float> stop=<float> nsteps=<integer> sweep_type=<lin/log>
```

Performs an AC analysis.

If the circuit is non-linear, a successful Operating Point (OP) is needed to linearize the circuit.

The sweep type is by default (and currently unchangeable) logarithmic.

Parameters:

- `start`: the starting frequency of the sweep, in Hz.
- `stop`: the final angular frequency, in Hz.
- `nsteps`: the number of steps to be executed.
- `sweep_type`: a parameter that can be set to `LOG` or `LIN` (the default), selecting a logarithmic or a linear frequency sweep.

Examples:

```
.ac lin 1 320 320
```

```
.ac sweep_type=lin start=320 stop=320 nsteps=1
```

Periodic Steady State (.PSS)

```
.PSS period=<float> [points=<int> step=<float> method=<string>
autonomous=<bool>]
```

This analysis tries to find the periodic steady state (PSS) solution of the circuit.

Parameters:

- `period`: the period of the solution. To be specified only in not autonomous circuits (which are somehow clocked).
- `points`: How many time points to use to discretize the solution. If `step` is set, this is automatically computed.

- `step`: Time step on the period. If `points` is set, this is automatically computed.
- `method`: the PSS algorithm to be employed. Options are: `shooting` (default) and `brute-force`.
- `autonomous`: self-explanatory boolean. If set to `True`, currently the simulator halts, because autonomous circuits are not supported, yet.

Pole-Zero analysis (.PZ)

The PZ analysis computes the poles (and optionally the zeros) of a circuit.

General syntax:

It can be specified with any of the following equivalent syntaxes:

```
.PZ [OUTPUT=<V(node1,node2)> SOURCE=<string> ZEROS=<bool>
SHIFT=<float>]
```

or

```
.PZ [V(<node1>,<node2>) <SOURCE> <ZEROS=1> <SHIFT=0>]
```

Internally, it is implemented through the modification-decomposition (MD) method, which is based on finding the eigenvalues of the Time Constant Matrix (TCM).

All the following parameters are optional and only needed for zero calculation.

Parameters:

- `output`: the circuit output voltage, in the form of `<V(node1,node2)>`. Notice the lack of space in between nodes and comma.
- `source`: the `part_id` of the input source.
- `zeros`: boolean, calculate the zeros as well. If `output` and `source` are set, then this is automatically set to 1 (true).
- `shift` initial frequency shift for calculation of the singularities. Optional. In a network that has zeros in the origin, this may be set to some non-zero value since the beginning.

Symbolic small-signal (.SYMBOLIC)

Performs a small-signal analysis of the circuit, optionally including AC elements.

General syntax:

```
.symbolic [tf=<source_id> ac=<boolean>]
```

- `tf`: If the source ID is specified, the transfer functions from the source to each of the variables in the circuit are calculated. From them, low-frequency gain, poles and zeros are extracted.
- `ac`: If set to `True`, capacitors and inductors will be included. Defaults to `False`, to speed up the solutions.

In the results, the imaginary unit is shown as `I`, the angular frequency as `w`.

We rely on the `Sympy` library for the low-level symbolic computations. The library is under active development and might have trouble (or take a long time) with medium-big or tricky netlists. Improvements are on their way, in the meanwhile, consider simplifying complex netlists, if solving is an issue.

1.4.7 Post-processing

.Plot

Plot the results from simulation to video.

General syntax:

`.plot <simulation_type> [variable1 variable2 ...  ]`

Parameters:

- `simulation_type`: which simulation will have the data plotted. Currently the available options are `tran`, `pss`, `ac` and `dc`.
- `variable1`, `variable2`: the signals to be plotted.

They may be:

- a voltage, syntax `V(<node>)`, to plot the voltage at the specified node, or `V(<node2>, <node1>)`, to plot the difference of the node voltages. E.g. `V(in)` or `V(2,1)`.
- a current, syntax `I(<source name>)`, e.g. `I(V2)` or `I(Vsupply)`

Plotting is possible only if `matplotlib` is available.

.Four

Perform a Fourier analysis over the latest transient data.

General syntax:

```
.FOUR <freq> var1 <var2 var3 ...>
```

The Fourier analysis is performed over the interval which is decided as follows:

- The data should be taken from the end of the simulation, so that if there is any build-up or stabilization process, the Fourier analysis is not affected (or less affected) by it.
- At least 1 period of the fundamental has to be used.
- Not more than 50% of the total simulation time should be used, if possible.
- Respecting the above, as much data as possible should be used, as it leads to more accurate results.

An algorithm selects the data for the Fourier transform from the data from the last transient analysis, then the data are re-sampled with a fixed time step, using a quadratic interpolation scheme.

A rectangular window is employed and the Fourier components are calculated using 10 frequency bins, ie 0, f, $2f \dots 9f$.

This post-processing function prints its results to the standard output.

Parameters:

- `freq`: the fundamental frequency, in Hz.
- `var1`, `var2` ... : the signals to execute the FOUR analysis on. Each signal is treated independently.

  They may be:

  - a voltage, syntax `V(<node>)`, e.g. `V(in)` or `V(2,1)`.
  - a current, syntax `I(<source name>)`, e.g. `I(V2)` or `I(Vsupply)`

Example:

```
.FOUR 100K V(n1) I(V2)
```

.FFT

FFT analysis of the time evolution of a variable.

General syntax:

```
 .FFT <variable> [START=<float> STOP=<float> NP=<int>
+ FORMAT=<string> WINDOW=<string> ALFA=<float>
+ FREQ=<float> FMIN=<float> FMAX=<float>]
```

This post-processing analysis is a more flexible and complete version of the .FOUR statement.

The analysis uses a variable, user-selectable amount of time data, re-sampled with a fixed time step using quadratic interpolation, with a customizable windowing applied.

The time interval is specified through the `start` and `stop` parameters, if they are not set, all the available data is used. For compatibility, the simulator accepts as synonyms of `start` and `stop` the parameters `from` and `to`.

The function behaves differently whether the parameter `freq` is specified or not:

- If the fundamental frequency `freq` (f in the following) is specified, the analysis will perform an harmonic analysis, much like a `.FOUR` statement, considering only the DC component and the harmonics of f from the first up to the 9th (ie f, $2f$, $3f$ … $9f$).
- If `freq` is left unspecified, a standard FFT analysis is performed, starting from $f = 0$, to a frequency $f_{max} = 1/(2T_{TOT}n_p)$, where T_{TOT} is the total length of the considered data in seconds and n_p is the number of points in the FTT, set through the `np` parameter to this analysis.

The output data is printed to a file having a file name identical to the output file as specified with the `-o` switch at the invocation of the simulator, with an extension `.lis` appended.

Parameters:

- `variable`: the identifier of a variable. Eg. `'V(n1)'` or `'I(VS)'`.
- `freq`: The fundamental frequency, in Hertz. If it is specified, the output will be limited to the harmonics of this frequency. The Total Harmonic Distortion (THD) evaluation will also be enabled.
- `start`: The first time instant to be considered for the transient analysis. If unspecified, it will be the beginning of the transient simulation.
- `from`: Alternative specification of the `start` parameter.
- `stop`: Last time instant to be considered for the FFT analysis. If unspecified, it will be the end time of the transient simulation.
- `to`: Alternative specification of the `stop` parameter.

- `np`: A power of two that specifies how many points should be used when computing the FFT. If it is set to a value that is not a power of 2, it will be rounded up to the nearest power of 2. It defaults to 1024.
- `window`: The windowing type. The following values are available:
  - 'RECT' for a rectangular window, equivalent to no window at all.
  - 'BART', for a Bartlett window.
  - 'HANN', for a Hanning window.
  - 'HAMM' for a Hamming window.
  - 'BLACK' for a Blackman window.
  - 'HARRIS' for a Blackman-Harris window.
  - 'GAUSS' for a Gaussian window.
  - 'KAISER' for a Kaiser-Bessel window.

  The default is the rectangular window.
- `alpha`: The
  $sigma$ for a Gaussian window or the $beta$ for a Kaiser window. Defaults to 3 and is ignored if a window different from Gaussian or Kaiser is selected.
- `fmin`: Suppress all data below this frequency, expressed in Hz. The suppressed data is neither returned nor used to compute the THD (if it is computed at all). The DC component is always preserved. Defaults to: return and use all data.
- `fmax`: The dual to `fmin`, discard data above `fmax` and also do not use it if computing the THD. Expressed in Hz, defaults to infinity.

Example:

```
.FFT V(n1,n2) NP=1024 START=0.2u STOP=1.5u WINDOW=HANN
```

1.4.8 Other directives

End

General syntax:

`.end`

Force the parser to stop reading the netlist. Everything after this line is disregarded.

Ends

General syntax:

`.ends`

Closes a subcircuit block.

Ic

Set an Initial Condition for circuit analysis.

General syntax:

`.ic name=<ic_label> [v(<node>)=<value> i(<element_name>)=<value> ... ]`

This allows the specification of a state of a circuit. Every node voltage or current (through appropriate elements) may be specified. If not set, it will be set to `0`. Notice that setting an inappropriate or inconsistent IC will create convergence problems.

Example:

```
.ic name=oscillate1 V(1)=10 V(nOUT)=2 I(VTEST)=5m
```

To use an IC directive in a transient analysis, set '`UIC=3`' and '`IC_LABEL=<ic_label>`'.

Include

General syntax:

`.include <filename>`

Include a file. It's equivalent to copy & paste the contents of the file to the bottom of the netlist.

Subckt

General syntax:

`.subckt <subckt_label> [node1 node2 ... ]`

Subcircuits are netlist block that may be called anywhere in the circuit using a subckt call. They can have other `.subckt` calls within - but beware of recursively calling the same subcircuit!

They can hold other directives, but the placement of the directive doesn't change its meaning (i.e. if you add an `.op` line in the subcircuit or outside of it it's the same).

They can't be nested and have to be ended by a `.ends` directive.

CHAPTER 2

Getting started: examples and tutorials

ahkab can be used from Python as a module and from the shell through its Command Line Interface (CLI).

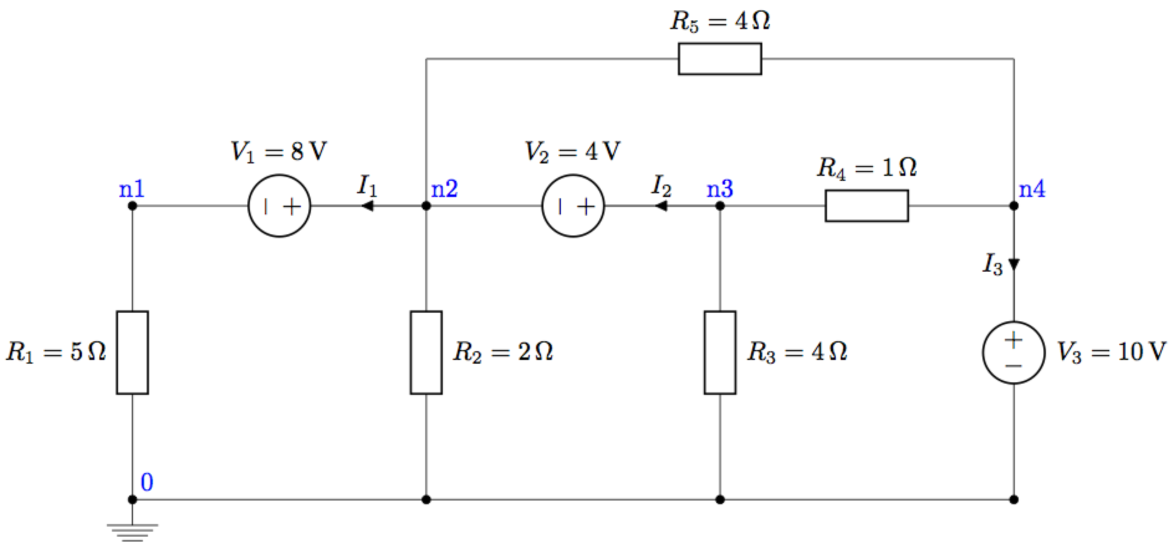
2.1 Simulating from Python

2.1.1 A first OP example

Performing numeric Operating Point (OP) simulations with ahkab of linear networks – like those in undergrad network theory textbooks– is really straight-forward.

The circuit

In this brief text, we will describe the following circuit:



Circuit description

First of all, we need the ahkab library to be imported:

```python
import ahkab
```

Next, we create a new circuit object. The only parameter you need is the name, which in the following is 'Simple Example Circuit'.

```
mycir = ahkab.Circuit('Simple Example Circuit')
```

Now's time to add the elements. The circuit object we just created (`mycir`) offers plenty of convenience methods to add elements to your circuit instances.

The general structure of the methods signature is:

```
add_<element_type>(part_id, node1, node2, value)
```

For example, we have an `add_resistor` method, with signature:

```
add_resistor(part_id, n1, n2, value)
```

Our circuit is described by:

```
mycir.add_resistor('R1', 'n1', mycir.gnd, value=5)
mycir.add_vsource('V1', 'n2', 'n1', dc_value=8)
mycir.add_resistor('R2', 'n2', mycir.gnd, value=2)
mycir.add_vsource('V2', 'n3', 'n2', dc_value=4)
mycir.add_resistor('R3', 'n3', mycir.gnd, value=4)
mycir.add_resistor('R4', 'n3', 'n4', value=1)
mycir.add_vsource('V3', 'n4', mycir.gnd, dc_value=10)
mycir.add_resistor('R5', 'n2', 'n4', value=4)
```

Defining the simulation and running it

Next, we need the OP simulation object. And then we start the simulation.

```
opa = ahkab.new_op()
r = ahkab.run(mycir, opa)['op']
```

The simulation takes a few milliseconds.

Inspecting the results

Simply printing the results with `print` formats the simulation results in a nice table:

```
print r
```

```
OP simulation results for 'Simple Example Circuit'.
Run on 2015-05-09 12:38:54, data file /var/folders/9y/nry2qj0962l38pqk3_8_8ndm0000gn/T/t
Variable    Units      Value         Error      %
----------  --------  ----------  ------------  ---
VN1         V         -3.86364     3.86369e-12    0
VN2         V          4.13636    -4.13614e-12    0
VN3         V          8.13636    -8.13571e-12    0
VN4         V         10          -1.00027e-11    0
I(V1)       A         -0.772727    0              0
I(V2)       A         -0.170455    0              0
I(V3)       A         -3.32955     0              0
```

The op_solution class also provides an easy to use dictionary-like interface to access your data, which is described in the documentation.

Conclusions

We hope this example, while basic, allows you to get more confident with using ahkab, especially to simulate linear networks.

Don't forget to report any bugs you may run into – *it happens!* – and have fun doing electronics! :)

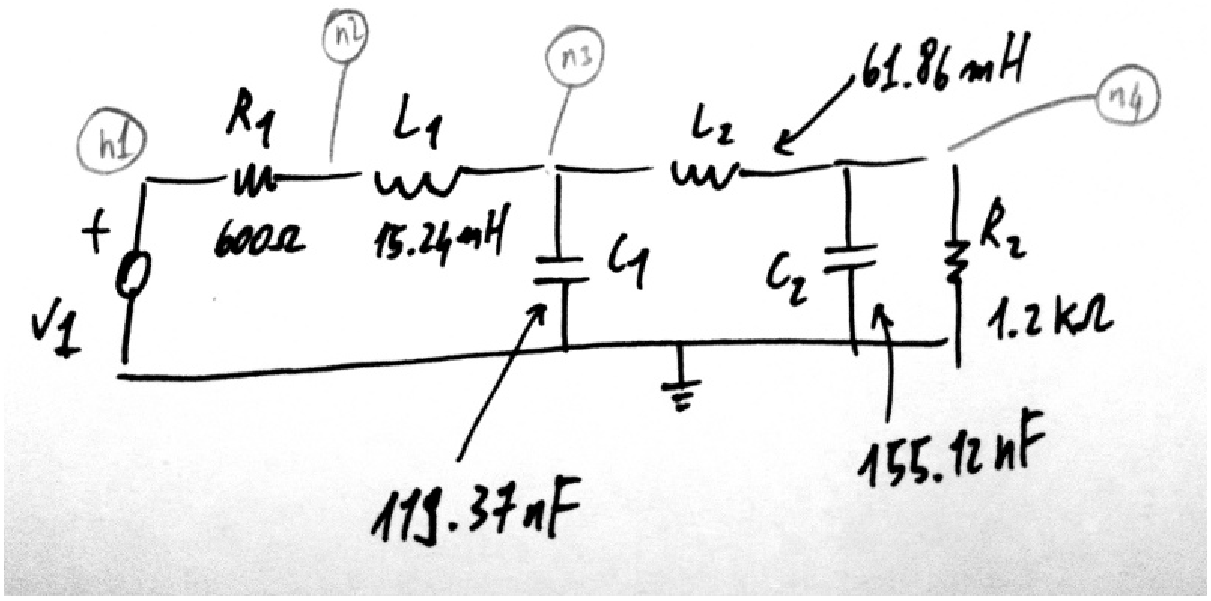
2.1.2 AC and TRAN tutorial

While the simulation below might me done from the command line with an netlist file, interacting with the simulator inside a Python program gives the user the ability to employ all the extreme flexibility and power of the Python language.

This page gives an beginners tutorial showing how, especially illustrating AC and TRAN simulations. Please refer to the doc pages for ahkab and *ahkab.circuit* for more.

Tutorial

Let's say we would like to simulate the AC characteristics and the step response of a Butterworth low pass filter, such as this:



This example is example 7.4 in from Hercules G. Dimopoulos, *Analog Electronic Filters: Theory, Design and Synthesis*, Springer.

The code to describe the circuit is the following:

First import the modules and create a new circuit:

```
import ahkab
from ahkab import circuit, printing, time_functions

mycircuit = circuit.Circuit(title="Butterworth Example circuit")
```

Elements are to be connected to *nodes*. There is one special node, the reference (gnd):

```
import ahkab
from ahkab import circuit, printing, time_functions
mycircuit = circuit.Circuit(title="Butterworth Example circuit")

gnd = mycircuit.get_ground_node()
```

and ordinary nodes.

Ordinary nodes can be defined as:

```
# setup
import ahkab
from ahkab import circuit, printing, time_functions
mycircuit = circuit.Circuit(title="Butterworth Example circuit")
# now we can define the nodes
# 1. using arbitrary strings to describe the nodes
# eg:
n1 = 'n1'
# 2. using the alternative syntax:
n1 = mycircuit.create_node('n1')
# the helper function create_node() will check that this is not a
# node name that was used somewhere else in your circuit
```

Then you can use the nodes you have defined to add your elements to the circuit. The circuit instance provides convenient helper functions.

The passives in example 7.4 can be added as:

```
import ahkab
from ahkab import circuit, printing, time_functions
mycircuit = circuit.Circuit(title="Butterworth Example circuit")

gnd = mycircuit.get_ground_node()

mycircuit.add_resistor("R1", n1="n1", n2="n2", value=600)
mycircuit.add_inductor("L1", n1="n2", n2="n3", value=15.24e-3)
mycircuit.add_capacitor("C1", n1="n3", n2=gnd, value=119.37e-9)
mycircuit.add_inductor("L2", n1="n3", n2="n4", value=61.86e-3)
mycircuit.add_capacitor("C2", n1="n4", n2=gnd, value=155.12e-9)
mycircuit.add_resistor("R2", n1="n4", n2=gnd, value=1.2e3)
```

Next, we want to add the voltage source V1.

- First, we define a pulse function to provide the time-variable characteristics of V1, to be used in the transient simulation:

```
voltage_step = time_functions.pulse(v1=0, v2=1, td=500e-9, tr=1e-12, pw=1, tf=1e-12, per
```

- Then we add a voltage source named V1 to the circuit, with the time-function we have just built:

```
mycircuit.add_vsource("V1", n1="n1", n2=gnd, dc_value=5, ac_value=1, function=voltage_st
```

Putting all together:

```
voltage_step = time_functions.pulse(v1=0, v2=1, td=500e-9, tr=1e-12, pw=1, tf=1e-12, per
mycircuit.add_vsource("V1", n1="n1", n2=gnd, dc_value=5, ac_value=1, function=voltage_st
```

We can now check that the circuit is defined as we intended, generating a netlist.

```
print mycircuit
```

If you invoke python now, you should get an output like this:

```
* TITLE: Butterworth Example circuit
R1 n1 n2 600
L1 n2 n3 0.01524
C1 n3 0 1.1937e-07
L2 n3 n4 0.06186
C2 n4 0 1.5512e-07
R2 n4 0 1200.0
V1 n1 0 type=vdc vdc=5 vac=1 arg=0 type=pulse v1=0 v2=1 td=5e-07 per=2 tr=1e-12 tf=1e-12
```

Next, we need to define the analyses to be carried out:

```
op_analysis = ahkab.new_op()
ac_analysis = ahkab.new_ac(start=1e3, stop=1e5, points=100)
tran_analysis = ahkab.new_tran(tstart=0, tstop=1.2e-3, tstep=1e-6, x0=None)
```

Next, we run the simulation:

```
r = ahkab.run(mycircuit, an_list=[op_analysis, ac_analysis, tran_analysis])
```

Save the script to a file and start python in interactive model with:

```
python -i script.py
```

All results were saved in a variable ‘r’. Let’s take a look at the OP results:

```
>>> r
`{'ac': <results.ac_solution instance at 0xb57e4ec>,
'op': <results.op_solution instance at 0xb57e4cc>,
'tran': <results.tran_solution instance at 0xb57e4fc>}`

>>> r['op'].results
{'VN4': 3.3333333333333335, 'VN3': 3.3333333333333335, 'VN2': 3.3333333333333335,
'I(L1)': 0.0027777777777777779, 'I(V1)': -0.0027777777777777779, 'I(L2)': 0.002777777777
```

You can get all the available variables calling the keys() method:

```
>>> r['op'].keys()
['VN1', 'VN2', 'VN3', 'VN4', 'I(L1)', 'I(L2)', 'I(V1)']
>>> r['op']['VN4']
3.3333333333333335
```

Then you can access the data through the dictionary interface, eg:

```
>>> "The DC output voltage is %s %s" % (r['op']['VN4'] , r['op'].units['VN4'])
'The DC output voltage is 3.33333333333 V'
```

A similar interface is available for the AC simulation results:

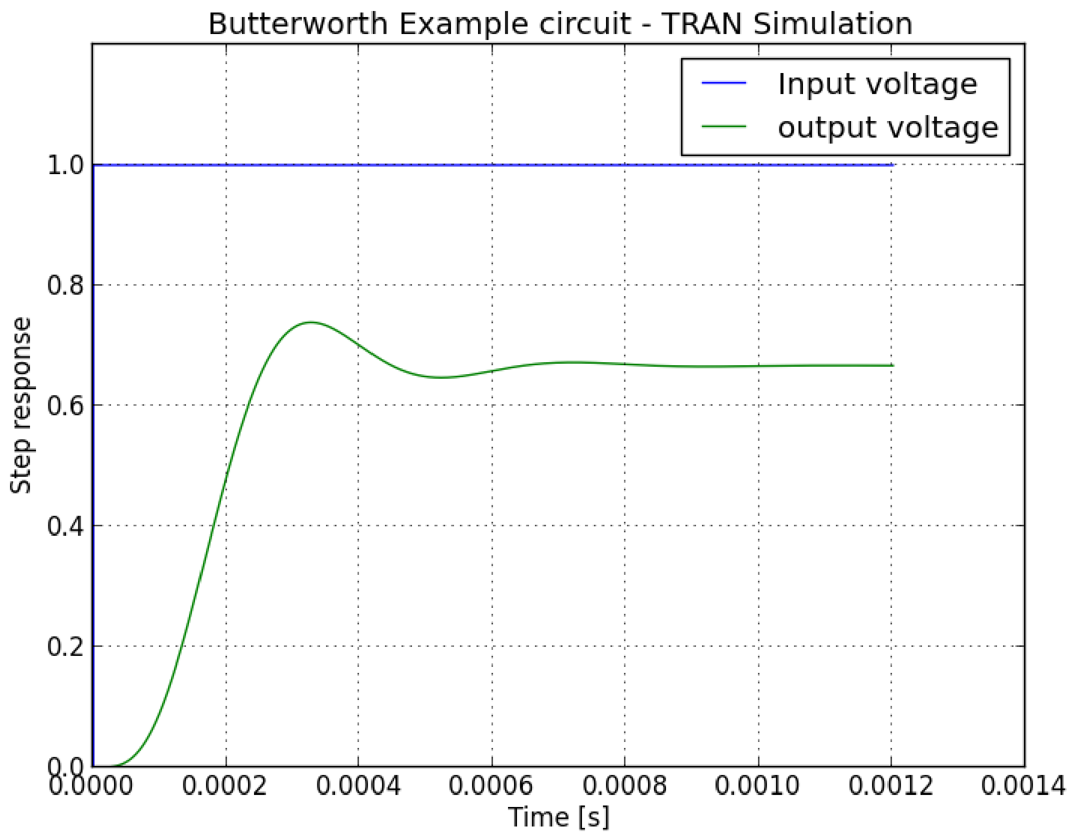
```
>>> print(r['ac'])
<AC simulation results for Butterworth Example circuit (netlist None).
LOG sweep, from 1000 Hz to 100000 Hz, 100 points. Run on 2011-12-19 17:24:29>
>>> r['ac'].keys()
['#w', '|Vn1|', 'arg(Vn1)', '|Vn2|', 'arg(Vn2)', '|Vn3|', 'arg(Vn3)', '|Vn4|',
'arg(Vn4)', '|I(L1)|', 'arg(I(L1))', '|I(L2)|', 'arg(I(L2))', '|I(V1)|', 'arg(I(V1))']
```

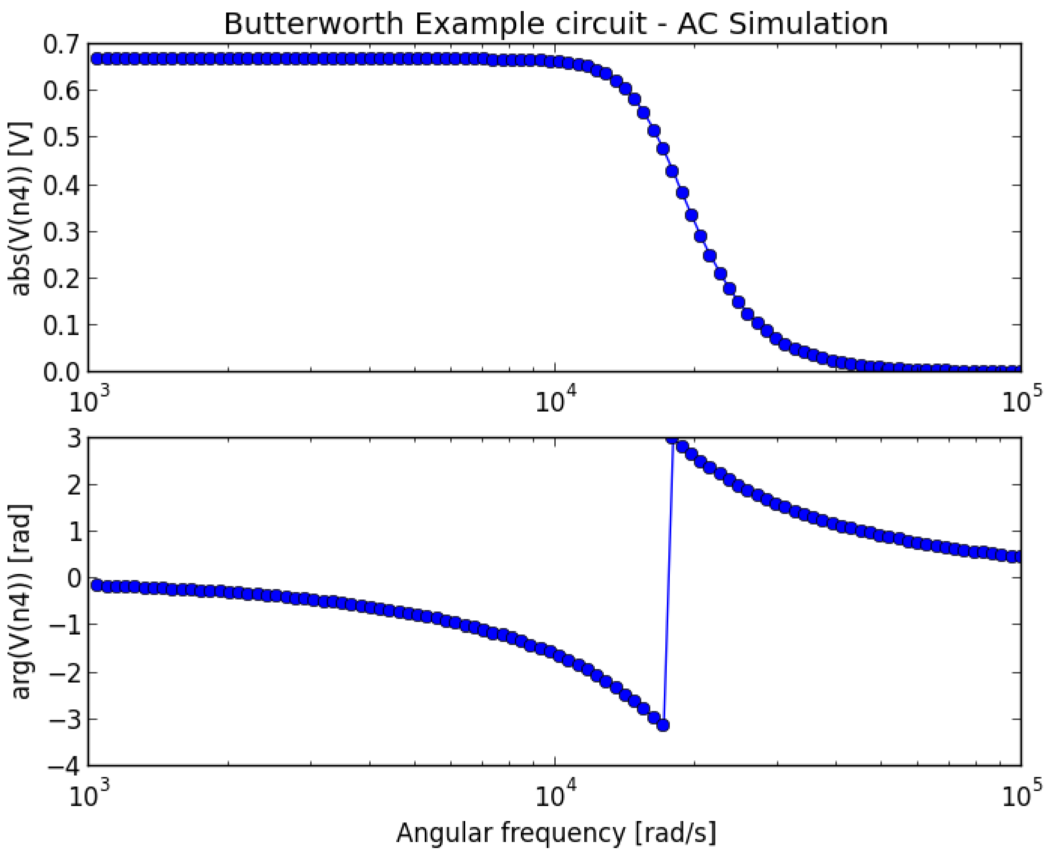
And a similar approach can be used to access the TRAN data set.

The data can be plotted through matplotlib, for example:

```
import pylab as plt
import numpy as np

fig = plt.figure()
plt.title(mycircuit.title + " - TRAN Simulation")
plt.plot(r['tran']['T'], r['tran']['VN1'], label="Input voltage")
plt.hold(True)
plt.plot(r['tran']['T'], r['tran']['VN4'], label="output voltage")
plt.legend()
plt.hold(False)
plt.grid(True)
plt.ylim([0,1.2])
plt.ylabel('Step response')
plt.xlabel('Time [s]')
fig.savefig('tran_plot.png')

fig = plt.figure()
plt.subplot(211)
plt.semilogx(r['ac']['w'], np.abs(r['ac']['Vn4']), 'o-')
plt.ylabel('abs(V(n4)) [V]')
plt.title(mycircuit.title + " - AC Simulation")
plt.subplot(212)
plt.grid(True)
plt.semilogx(r['ac']['w'], np.angle(r['ac']['Vn4']), 'o-')
plt.xlabel('Angular frequency [rad/s]')
plt.ylabel('arg(V(n4)) [rad]')
fig.savefig('ac_plot.png')
plt.show()
```

The previous code generates the following plots:


Butterworth Example circuit - TRAN Simulation
Input voltage
output voltage
Step response
Time [s]
0.0
0.2
0.4
0.6
0.8
1.0
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0.0012
0.0014


Butterworth Example circuit - AC Simulation
abs(V(n4)) [V]
arg(V(n4)) [rad]
Angular frequency [rad/s]
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
−4
−3
−2
−1
0
1
2
3
10^3
10^4
10^5

It is also possible to extract attenuation in pass-band (0-2kHz) and stop-band (6.5kHz and up).

The problem is that the voltages/currents we are looking for may not have been evaluated by ahkab at the desired points. This can be easily overcome with interpolation through scipy.

Here is a snippet of code to evaluate the attenuation is pass-band and stop band in the example:

```
import numpy as np
import scipy, scipy.interpolate

# Normalize the output to the low frequency value and convert to array
norm_out = np.abs(r['ac']['Vn4'])/np.abs(r['ac']['Vn4']).max()
# Convert to dB
norm_out_db = 20*np.log10(norm_out)
# Convert angular frequencies to Hz and convert matrix to array
frequencies = r['ac']['w']/2/np.pi
# call scipy to interpolate
norm_out_db_interpolated = scipy.interpolate.interp1d(frequencies, norm_out_db)

print "Maximum attenuation in the pass band (0-%g Hz) is %g dB" % \
(2e3, -1.0*norm_out_db_interpolated(2e3))
print "Minimum attenuation in the stop band (%g Hz - Inf) is %g dB" % \
(6.5e3, -1.0*norm_out_db_interpolated(6.5e3))
```

You should see the following output:

```
Maximum attenuation in the pass band (0-2000 Hz) is 0.351373 dB
Minimum attenuation in the stop band (6500 Hz - Inf) is 30.2088 dB
```

Download the python file.

2.1.3 Pole-Zero example

Giuseppe Venturini, Thu May 7, 2015

In this short example we will simulate a simple RLC circuit with the ahkab simulator.

In particular, we consider a series resonant RLC circuit. If you need to refresh your knowledge on 2nd filters, you may take a look at this page.

The plan: what we'll do

0. A brief analysis of the circuit

This should be done with pen and paper, we'll just mention the results. The circuit is pretty simple, feel free to skip if you find it boring.

1. How to describe the circuit with ahkab

We'll show this:

- from Python,
- and briefly with a netlist deck.

2. Pole-zero analysis

- We will extract poles and zeros.
- We'll use them to build input-output transfer function, which we evaluate.

3. AC analysis

- We will run an AC analysis to evaluate numerically the transfer function.

4. Symbolic analysis

- We'll finally run a symbolic analysis as well.
- Once we have the results, we'll substitute for the real circuit values and verify both AC and PZ analysis.

5. Conclusions

We will check that the three PZ, AC and Symbolic analysis match!

The circuit

The circuit we simulate is a very simple one:

0. Theory

Once one proves that the current flowing in the only circuit branch in the Laplace domain is given by:

$$I(s) = \frac{1}{L} \cdot \frac{s}{s^2 + 2\alpha \cdot s + \omega_0^2}$$

Where:

- s is the Laplace variable, $s = \sigma + j\omega$:
  - j is the imaginary unit,
  - ω is the angular frequency (units rad/s).
- α is known as the *Neper frequency* and it is given by $R/(2L)$,
- ω_0 is the *(undamped) resonance frequency, equal to $(\sqrt{LC})^{-1}$.

It's easy to show that the pass-band transfer function we consider in our circuit, V_{OUT}/V_{IN}, has the expression:

$$H(s) = \frac{V_{OUT}}{V_{IN}}(s) = k_0 \cdot \frac{s}{s^2 + 2\alpha \cdot s + \omega_0^2}$$

Where the coeffiecient k_0 has value $k_0 = R/L$.

Solving for poles and zeros, we get:

- One zero:
  - z_0, located in the origin.
- Two poles, p_0 and p_1:
  - $p_{0,1} = -\alpha \pm \sqrt{\alpha^2 - \omega_0^2}$

1. Describe the circuit with ahkab

Let's call `ahkab` and describe the circuit above.

First we need to import `ahkab`:

```
# libraries we need
import ahkab
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
print "We're using ahkab %s" % ahkab.__version__
```

```
We're using ahkab 0.16
```

Then we create a new circuit object titled 'RLC bandpass', which we name `bpf` from Band-Pass Filter:

```
bpf = ahkab.Circuit('RLC bandpass')
```

A circuit is made of, internally, components and nodes. For now, our `bpf` circuit is empty and really of not much use.

We wish to define our nodes, our components, specifying their connection to the appropriate nodes and inform the circuit instance about the what we did.

It sounds complicated, but it is actually very simple, also thanks to the convenience functions `add_*()` in the `Circuit` instances (circuit documentation).

We now add the inductor `L1`, the capacitor `C1`, the resistor `R1` and the input source `V1`:

```
bpf = ahkab.Circuit('RLC bandpass')
bpf.add_inductor('L1', 'in', 'n1', 1e-6)
bpf.add_capacitor('C1', 'n1', 'out', 2.2e-12)
bpf.add_resistor('R1', 'out', bpf.gnd, 13)
# we also give V1 an AC value since we wish to run an AC simulation
# in the following
bpf.add_vsource('V1', 'in', bpf.gnd, dc_value=1, ac_value=1)
```

Notice that:

- the nodes to which they get connected (`'in'`, `'n1'`, `'out'`...) are nothing but strings. If you prefer handles, you can call the `create_node()` method of the circuit instance `bpf` (create_node documentation).
- Using the convenience methods `add_*`, the nodes are not explicitly added to the circuit, but they are in fact automatically taken care of behind the hood.

Now we have successfully defined our circuit object `bpf`.

Let's see what's in there and generate a netlist:

```
print(bpf)
```

```
* RLC bandpass
L1 in n1 1e-06
C1 n1 out 2.2e-12
R1 out 0 13
V1 in 0 type=vdc value=1 vac=1
```

The above text defines the same circuit in netlist form. It has the advantage that it's a very concise piece of text and that the syntax resembles (not perfectly yet) that of simulators such as SPICE.

If you prefer to run `ahkab` from the command line, be sure to check the Netlist syntax doc page and to add the simulation statements, which are missing above.

2. PZ analysis

The analysis is set up easily by calling `ahkab.new_pz()`. Its signature is:

```
ahkab.new_pz(input_source=None, output_port=None, shift=0.0, MNA=None,
             outfile=None, x0=u'op', verbose=0)
```

And you can find the documentation for ahkab.new_pz here.

We will set:

- Input source and output port, to enable the extraction of the zeros.
  - the input source is `V1`,
  - the output port is defined between the output node `out` and ground node (`bpf.gnd`).
- We need no linearisation, since the circuit is linear. Therefore we set `x0` to `None`.
- I inserted a non-zero shift in the initial calculation frequency below. You may want to fiddle a bit with this value, the algorithm internally tries to kick the working frequency away from the exact location of the zeros, since we expect a zero in the origin, we help the simulation find the zero quickly by shifting away the initial working point.

```
pza = ahkab.new_pz('V1', ('out', bpf.gnd), x0=None, shift=1e3)
r = ahkab.run(bpf, pza)['pz']
```

The results are in the `pz_solution` object `r`. It has an interface that works like a dictionary.

Eg. you can do:

```
r.keys()
```

```
[u'p0', u'p1', u'z0']
```

Check out the documentation on pz_solution for more.

Let's see what we got:

```
print('Singularities:')
for x, _ in r:
    print "* %s = %+g %+gj Hz" % (x, np.real(r[x]), np.imag(r[x]))
```

```
Singularities:
* p0 = -1.03451e+06 -1.07297e+08j Hz
* p1 = -1.03451e+06 +1.07297e+08j Hz
* z0 = -1.44751e-13 +0j Hz
```

Note that the results are frequencies expressed in Hz (and *not* angular frequencies in rad/s).

Graphically, we can see better where the singularities are located:

```
figure(figsize=figsize)
# plot o's for zeros and x's for poles
for x, v in r:
    plot(np.real(v), np.imag(v), 'bo'*(x[0]=='z')+'rx'*(x[0]=='p'))
# set axis limits and print some thin axes
xm = 1e6
xlim(-xm*10., xm*10.)
plot(xlim(), [0,0], 'k', alpha=.5, lw=.5)
plot([0,0], ylim(), 'k', alpha=.5, lw=.5)
# plot the distance from the origin of p0 and p1
plot([np.real(r['p0']), 0], [np.imag(r['p0']), 0], 'k--', alpha=.5)
plot([np.real(r['p1']), 0], [np.imag(r['p1']), 0], 'k--', alpha=.5)
# print the distance between p0 and p1
plot([np.real(r['p1']), np.real(r['p0'])],
     [np.imag(r['p1']), np.imag(r['p0'])],
     'k-', alpha=.5, lw=.5)
# label the singularities
text(np.real(r['p1']), np.imag(r['p1'])*1.1, '$p_1$', ha='center',
     fontsize=20)
text(.4e6, .4e7, '$z_0$', ha='center', fontsize=20)
text(np.real(r['p0']), np.imag(r['p0'])*1.2, '$p_0$', ha='center',
     va='bottom', fontsize=20)
xlabel('Real [Hz]'); ylabel('Imag [Hz]'); title('Singularities');
```

As expected, we got two complex conjugate poles and a zero in the origin.

The resonance frequency

Let's check that indeed the (undamped) resonance frequency f_0 has the expected value from the theory.

It should be:

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

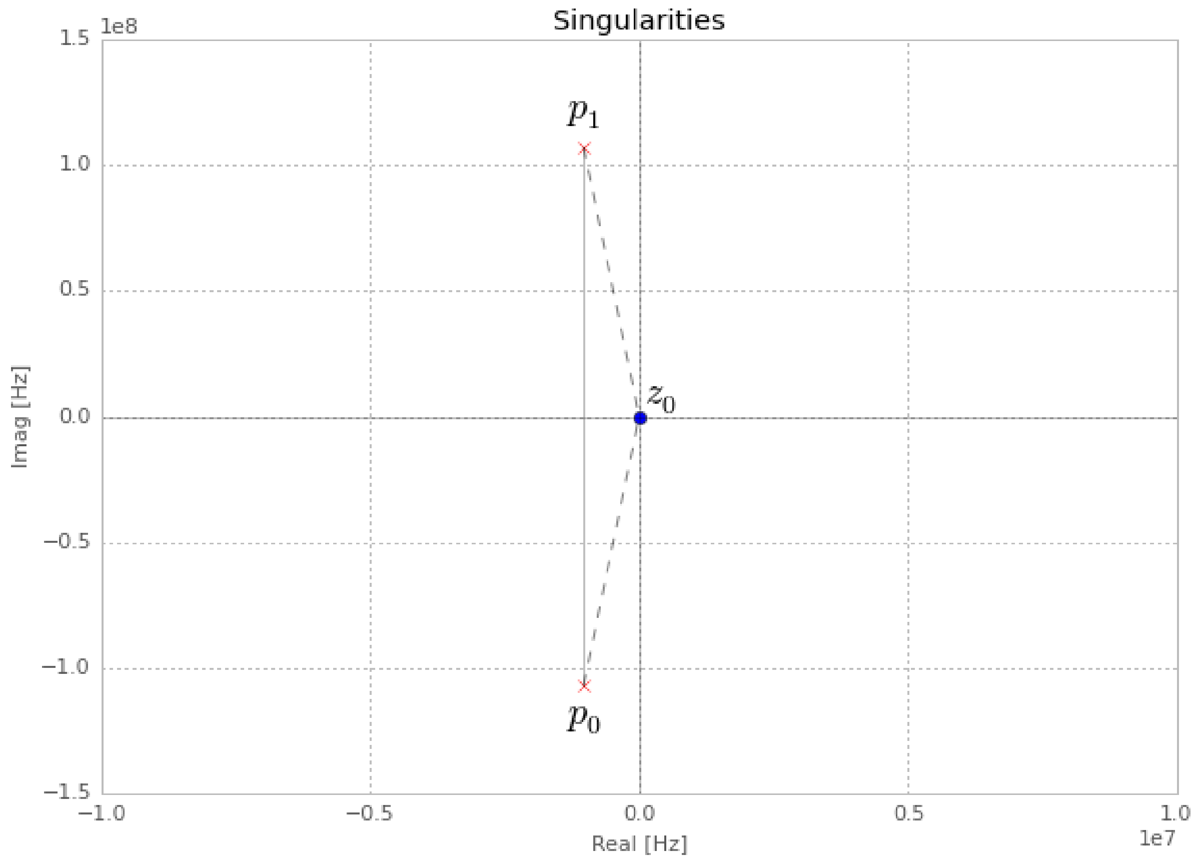
Since we have little damping, f_0 is very close to the damped resonant frequency in our circuit, given by the absolute value of the imaginary part of either $p_0$ or $p_1$.

In fact, the damped resonant frequency f_d is given by:

$$f_d = \frac{1}{2\pi}\sqrt{\alpha^2 - w_0^2}$$

Since this is an example and we have Python at our fingertips, we'll compensate for the frequency pulling due to the damping anyway. That way, the example is analytically correct.

```
C = 2.2e-12
L = 1e-6
f0 = 1./(2*np.pi*np.sqrt(L*C))
print 'Resonance frequency from analytic calculations: %g Hz' %f0
```



```
Resonance frequency from analytic calculations: 1.07302e+08 Hz
```

```
alpha = (-r['p0']-r['p1'])/2
a1 = np.real(abs(r['p0'] - r['p1']))/2
f0 = np.sqrt(a1**2 - alpha**2)
f0 = np.real_if_close(f0)
print 'Resonance frequency from PZ analysis: %g Hz' %f0
```

```
Resonance frequency from PZ analysis: 1.07292e+08 Hz
```

That's alright.

3. AC analysis

Let's perform an AC analysis:

```
aca = ahkab.new_ac(start=1e8, stop=5e9, points=5e2, x0=None)
rac = ahkab.run(bpf, aca)['ac']
```

Next, we use sympy to assemble the transfer functions from the singularities we got from the PZ analysis.

```
import sympy
sympy.init_printing()
```

```
from sympy.abc import w
from sympy import I
p0, p1, z0 = sympy.symbols('p0, p1, z0')
```

```
k = 13/1e-6 # constant term, can be calculated to be R/L
H = 13/1e-6*(I*w + z0*6.28)/(I*w +p0*6.28)/(I*w + p1*6.28)
Hl = sympy.lambdify(w, H.subs({p0:r['p0'], z0:abs(r['z0']), p1:r['p1']}))
```

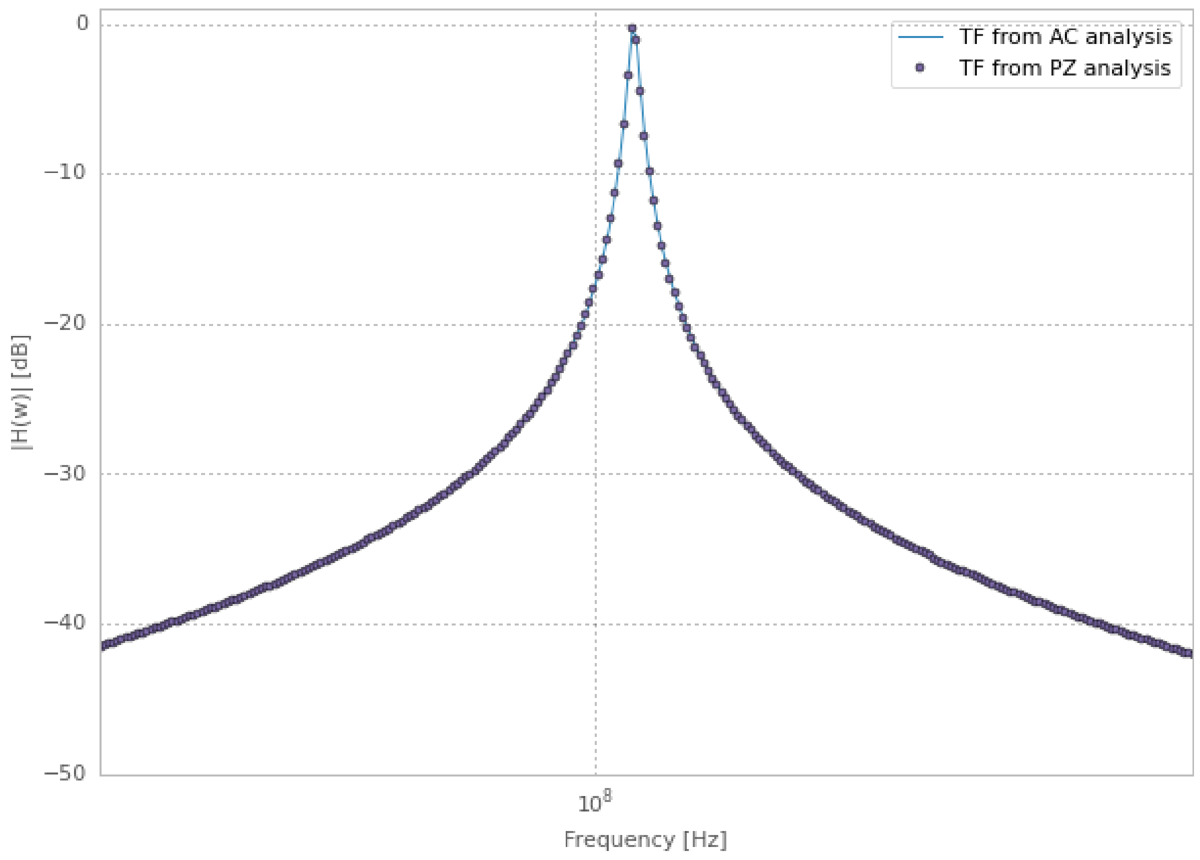
We need a function to evaluate the absolute value of a transfer function in decibels.

Here it is:

```
def dB20(x):
    return 20*np.log10(x)
```

Next we can plot $|H(\omega)|$ in dB and inspect the results visually.

```
figure(figsize=figsize)
semilogx(rac.get_x()/2/np.pi, dB20(abs(rac['vout'])),
         label='TF from AC analysis')
semilogx(rac.get_x()/2/np.pi, dB20(abs(Hl(rac.get_x()))), 'o', ms=4,
         label='TF from PZ analysis')
legend(); xlabel('Frequency [Hz]'); ylabel('|H(w)| [dB]');
xlim(4e7, 3e8); ylim(-50, 1);
```



4. Symbolic analysis

Next, we setup and run a symbolic analysis.

We set the input source to be `'V1'`, in this way, `ahkab` will calculate all transfer functions, together with low-frequency gain, poles and zeros, with respect to *every* variable in the circuit.

It is done very similarly to the previous cases:

```
symba = ahkab.new_symbolic(source='V1')
rs, tfs = ahkab.run(bpf, symba)['symbolic']
```

Notice how to the `'symbolic'` key corresponds a tuple of two objects: the symbolic results and the TF object that was derived from it.

Let's inspect their contents:

```
print(rs)
```

```
Symbolic simulation results for 'RLC bandpass' (netlist None).
Run on 2015-05-07 04:24:42.
I[L1]     = C1*V1*s/(C1*L1*s**2 + C1*R1*s + 1.0)
I[V1]     = -C1*V1*s/(C1*L1*s**2 + C1*R1*s + 1.0)
VIN   = V1
VN1   = V1*(C1*R1*s + 1.0)/(C1*L1*s**2 + C1*R1*s + 1.0)
VOUT      = C1*R1*V1*s/(C1*L1*s**2 + C1*R1*s + 1.0)
```

```
print tfs
```

```
Symbolic transfer function results for 'RLC bandpass' (netlist None).
Run on 2015-05-07 04:24:42.
I[L1]/V1:
    gain:   C1*s/(C1*L1*s**2 + C1*R1*s + 1.0)
    gain0:  0
    poles:
        0.5*(-C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
        -0.5*(C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
    zeros:
        0
I[V1]/V1:
    gain:   -C1*s/(C1*L1*s**2 + C1*R1*s + 1.0)
    gain0:  0
    poles:
        0.5*(-C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
        -0.5*(C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
    zeros:
        0
VIN/V1:
    gain:   1
VN1/V1:
    gain:   (C1*R1*s + 1.0)/(C1*L1*s**2 + C1*R1*s + 1.0)
    gain0:  1.00000000000000
    poles:
        0.5*(-C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
        -0.5*(C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
    zeros:
        -1/(C1*R1)
VOUT/V1:
    gain:   C1*R1*s/(C1*L1*s**2 + C1*R1*s + 1.0)
    gain0:  0
    poles:
        0.5*(-C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
        -0.5*(C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)
    zeros:
        0
```

In particular, to our transfer function corresponds:

```
tfs['VOUT/V1']
```

```
{u'gain': C1*R1*s/(C1*L1*s**2 + C1*R1*s + 1.0),
 u'gain0': 0,
 u'poles': [0.5*(-C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1),
  -0.5*(C1*R1 + sqrt(C1*(C1*R1**2 - 4.0*L1)))/(C1*L1)],
 u'zeros': [0]}
```

It's easy to show the above entries are a different formulation that corresponds to the theoretical results we introduced at the beginning of this example.

We'll do it graphically. First of all, let's isolate out TF:

```
Hs = tfs['VOUT/V1']['gain']
Hs
```

$$\frac{C_1 R_1 s}{C_1 L_1 s^2 + C_1 R_1 s + 1.0}$$

We wish to substitute the correct circuit values to R1, L1 and C1 to be able to evaluate numerically the results.

In order to do so, the symbolic_solution class in the results module has a method named as_symbols that takes a string of space-separed symbol names and returns the sympy symbols associated with them (symbolic_solution.as_symbols documentation).

```
s, C1, R1, L1 = rs.as_symbols('s C1 R1 L1')
HS = sympy.lambdify(w, Hs.subs({s:I*w, C1:2.2e-12, R1:13., L1:1e-6}))
```

Did we get the same results, let's sat within a 1dB accuracy?

```
np.allclose(dB20(abs(HS(rac.get_x()))), dB20(abs(H1(rac.get_x()))), atol=1)
```

```
True
```

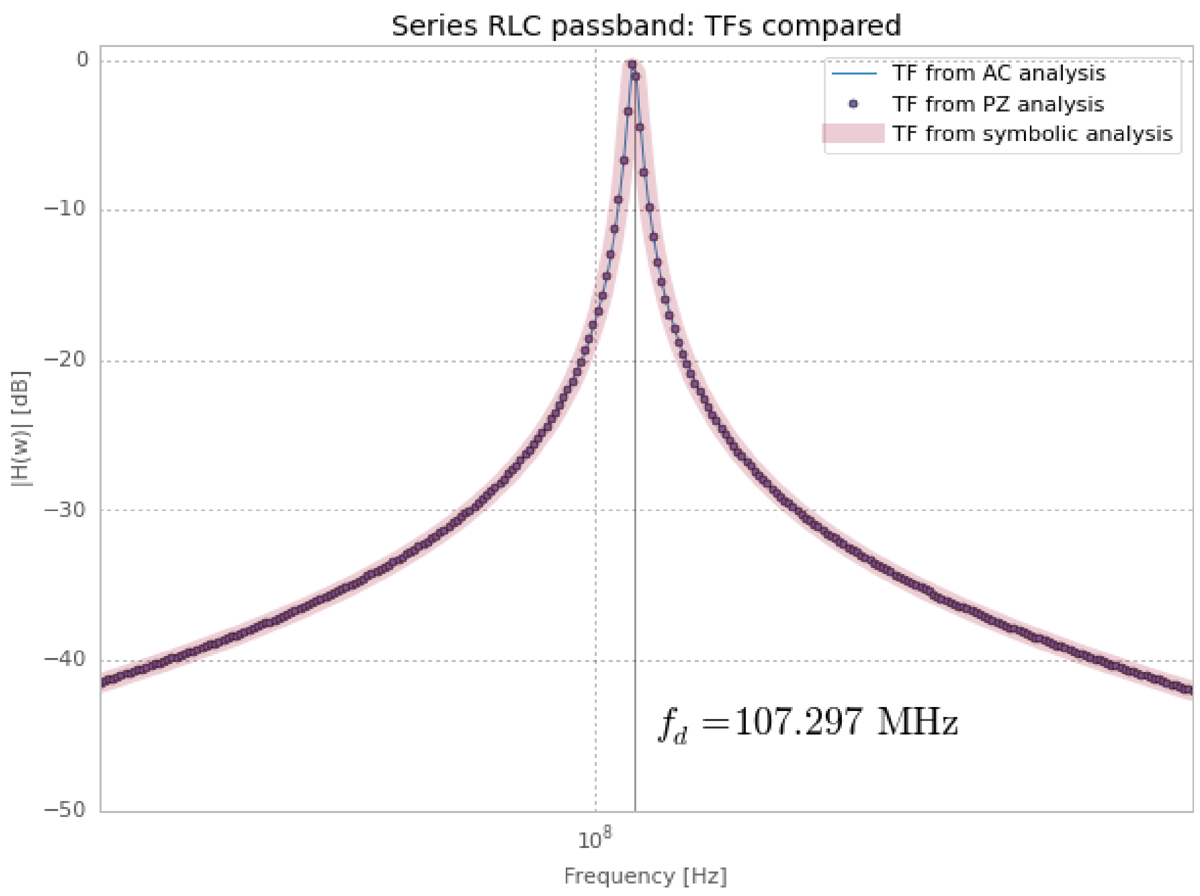
Good.

5. Conclusions

Let's take a look at PZ, AC and symbolic results together:

```
figure(figsize=figsize);  title('Series RLC passband: TFs compared')
semilogx(rac.get_x()/2/np.pi, dB20(abs(rac['vout'])),
         label='TF from AC analysis')
semilogx(rac.get_x()/2/np.pi, dB20(abs(H1(rac.get_x()))), 'o', ms=4,
         label='TF from PZ analysis')
semilogx(rac.get_x()/2/np.pi, dB20(abs(HS(rac.get_x()))), '-', lw=10,
         alpha=.2, label='TF from symbolic analysis')
vlines(1.07297e+08, *gca().get_ylim(), alpha=.4)
text(7e8/2/np.pi, -45, '$f_d = 107.297\\, \\mathrm{MHz}$', fontsize=20)
legend(); xlabel('Frequency [Hz]'); ylabel('|H(w)| [dB]');
xlim(4e7, 3e8); ylim(-50, 1);
```

I hope this example helped show how to use ahkab and in particular how to perform PZ, AC and symbolic analysis. If it also cleared up some doubts, great!



Please remember this is an experimental simulator and you may find bug... it's getting better but we're not really ready for prime time yet: please report any and all bugs you may encounter on the issue tracker.

This document was written with Jupiter running with a Python kernel (project formerly named IPython). You can find it here: Jupyter/IPython and you may access the whole notebook, which will allow you to download and modify this example.

Have fun!

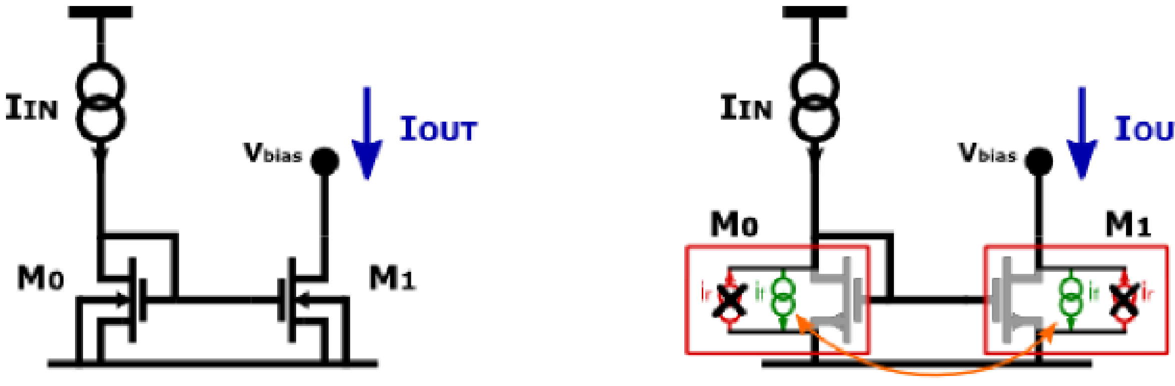
2.2 Simulating from the command line

2.2.1 Operating Point examples

Introduction

This page shows the operating point simulation of some circuits.

A simple 1:1 current mirror



A 1:1 current mirror (fig. 1a) is employed to provide a copy of an input current to a load. The two currents being resp. I_{IN} and I_{OUT}. A good current mirror complies with a mirroring ratio - equal to one here - has a low input resistance - it's a good current sink - and a high output resistance - it's a good current source or equi.v it's a low dependence of I_{out} on the output voltage. Please remember, "good" is a qualifier that is only meaningful in relation to the rest of the circuit where the mirror is employed.

According to the EKV model, the drain currents of `M0` and `M1` are composed by two components: the forward current (i_f) and the reverse current (i_r), see fig. 1b. The former is a function of the pinch-off voltage and the source voltage, referred to the bulk voltage. The latter is a function of the pinch-off voltage and the drain voltage, again referred to the bulk voltage. [Channel-length modulation is neglected]

The short between gate and drain of `M0` ensures that there can't be channel at the drain side of `M0`. The reverse current is hence null. For the correct operation of the mirror (ie I_{OUT} equal to I_{IN}), the channel has to be pinched off on the drain side of `M1` as well, because otherwise the output voltage would control the output current.

Since `M0` and `M1` share gate node, source node and bulk node, their forward currents (i_f) are equal. In addition each of them has a null reverse current.

The total drain current of an MOS transistor is according to the EKV model:

$$I_d = I_s(i_f - i_r)$$

Where I_s is known as specific current and it is a scaling factor, in particular it is proportional to the W/L ratio of the device.

In this example and in the following, all devices are matched (ie equally dimensioned).

Therefore:

$$I_{OUT} = I_{IN}$$

If `M1` had a form factor k times bigger than `M0`, then I_{s1} would have been proportionally bigger than I_{s0} and:

$$I_{OUT} = k \cdot I_{IN}$$

Netlist

```
* 1-to-1 current mirror
m0 inc inc 0 0 nch w=10u l=3u
m1 out inc 0 0 nch w=10u l=3u

i1 0 inc type=idc idc=16u

v1 out 0 type=vdc vdc=2.5

.model ekv nch TYPE=n VTO=.4 KP=400e-6

.op
```

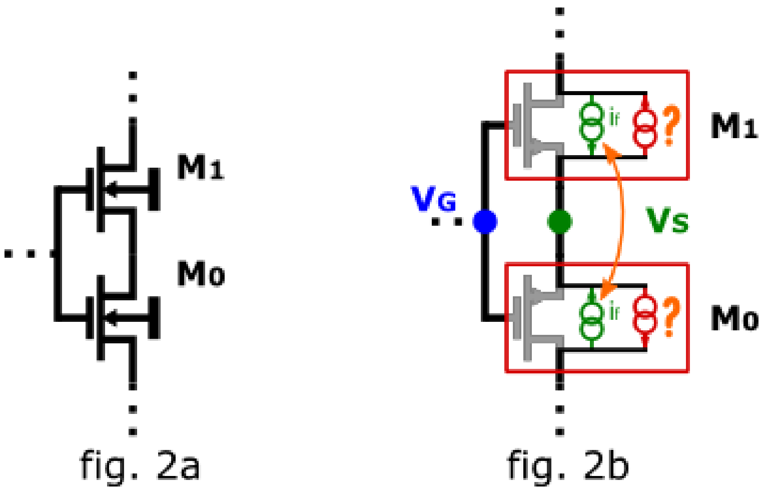
Simulation results

```
* 1-TO-1 CURRENT MIRROR
Starting op analysis:
Calculating guess: done.
Solving with Gmin:
Building Gmin matrix...
Solving...  done.
Solving without Gmin:
Solving...  done.
Difference check is within margins.
(Voltage: er=0.001, ea=1e-06, Current: er=0.001, ea=1e-09)
Solution without Gmin:
Vinc:       0.661166520045  V
Vout:                  2.5  V
I(V1):  -1.64022400455e-05  A
OP INFORMATION:
M0         N ch  MODERATE INVERSION                     SATURATION
beta  [A/V^2]:   0.000666997062349  Weff  [m]:       1e-05 (1e-05)   Leff      [m]:  2.9
Vdb       [V]:      0.661166520045   Vgb  [V]:      0.661166520045    Vsb      [V]:
VTH       [V]:                 0.4   VOD  [V]:      0.182380106934   nq:
Ids       [A]:   1.59999948649e-05  nv:              1.55302306603  Ispec      [A]:
gmg       [S]:   0.000135363138199   gms  [S]:  -0.000210508167477    rob    [Ohm]:
if:                   5.7034387414   ir:           0.00264346735244     Qf  [C/m^2]:
-------------------
M1         N ch  MODERATE INVERSION                     SATURATION
beta  [A/V^2]:   0.000683757860402  Weff  [m]:       1e-05 (1e-05)   Leff      [m]:  2.9
Vdb       [V]:                 2.5   Vgb  [V]:      0.661166520045    Vsb      [V]:
VTH       [V]:                 0.4   VOD  [V]:      0.182380106934   nq:
Ids       [A]:   1.64022352838e-05  nv:              1.55302306603  Ispec      [A]:
gmg       [S]:   0.000135367502575   gms  [S]:  -0.000210508167477    rob    [Ohm]:
if:                   5.7034387414   ir:            0.0025806785066     Qf  [C/m^2]:
-------------------
TOTAL POWER: 5.15842644346e-05 W
```

Notice how the two transistors have the same i_f and a low i_r (saturation). Channel length modulation (due to different drain voltages) explains for the discrepancy in I_{DS}

A different view of the 1:1 current mirror

Consider now the two transistors in fig. 2a.



The circuit is similar to the one in fig. 1a: the two transistors again share gate, source and bulk nodes (potential) and have a separate drain node, but this time the KCL at the source node is different: the node is not set to a fixed potential.

Nonetheless, a similar discussion can be drawn: considering fig. 2b, the forward and reverse current components have been put in evidence.

Notice how source and drain of `M0` appear to have been switched around: MOS transistors are geometrically symmetrical, the choice of drain and source labeling has no influence on the results. On the other hand, the choice made in fig. 2b allows us to state that again `M0` and `M1` have the same forward current.

Notice how the two devices can't be both in saturation: on the side on which they share the source node, if one transistor has a channel, the other needs to have one has well. In addition, at least one of the reverse currents has to be non-zero, since the total current entering the drain of `M1` (I_{TOT}) has to flow out of `M0` as well, or a net non-zero charge would be created at every instant.

Therefore both transistors can't be in saturation.

The results are:

- `M0`: i_f, $i_r = I_{TOT} + i_f$
- `M1`: i_f, $i_r = I_{TOT} - i_f$

The rest of the circuit - not shown - would set the actual value of i_f and I_{TOT}.

A 1:1/16th current mirror

A down-scaling current mirror is depicted in fig. 3a.

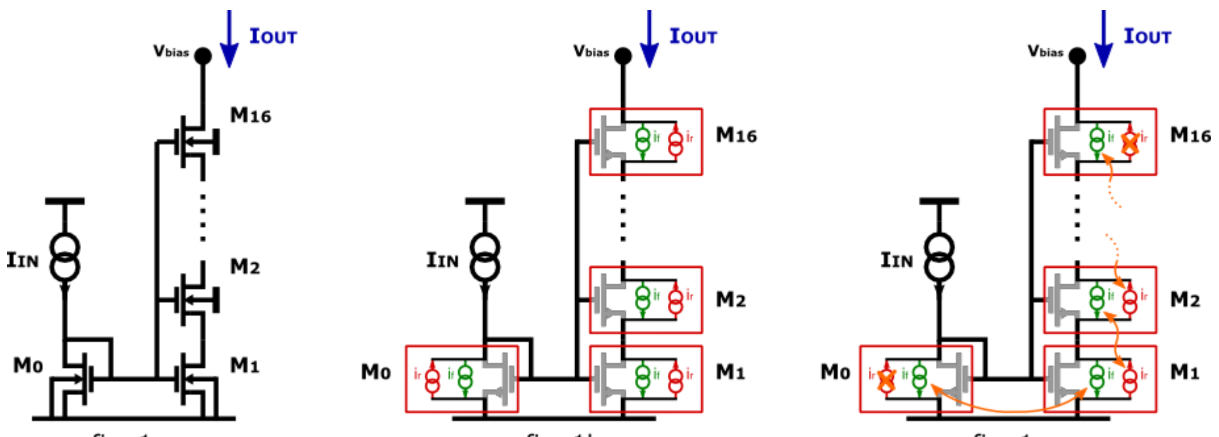


fig. 1a fig. 1b fig. 1c

Here again, the transistors share the same bulk and gate node, and, two-by-two, they also share drain/source node.

While this circuit is more complex than the previous ones, it can be analyzed in the same fashion, taking into account the results already presented:

- each neighboring transistor pair acts like a current mirror, ie same i_f/i_r,
- zero net charge can be created or destroyed at each instant.

Considering the currents, we have 17 forward currents and 17 reverse currents to be determined, for a total of 34 unknowns:

We can write:

- 15 equations of the type $i_f = i_r$ for neighboring devices,
- 1 equation for the mirror operation of the `M0`, `M1` pair,
- 1 equation setting $i_r = 0$ for `M0` (drain-gate short),
- 1 equation setting $I_s \cdot i_f = I_{IN}$ for `M0` (KCL),
- 1 equation setting $i_r = 0$ for `M16` (hp. in saturation),
- 15 equations to require that `M1`, `M2`, `M3`... `M16` have all the same drain current.

That gives a total of 34 equations.

It can be shown that the solution is:

- `M0`: $i_f = I_{IN}/I_s$, $i_r = 0$
- `M1`: $i_f = I_{IN}/I_s$, $i_r = 15/16 \cdot I_{IN}/I_s$
- `M2`: $i_f = 15/16 I_{IN}/I_s$, $i_r = 14/16 \cdot I_{IN}/I_s$
- and so on...

The general form is:

`M[n]`, for $n = 1 \ldots 16$, $i_f = (17-n)/16 \cdot I_{IN}/I_s$ and $i_r = (16-n)/16 \cdot I_{IN}/I_s$.

`M16` has $i_f = 1/16 \cdot I_{IN}/I_s$ and $i_r = 0$. Its drain current - the mirror output current - is therefore:

$$I_{OUT} = 1/16 \cdot I_{IN}.$$

Netlist

```
* 1-to-1/16th down-scaling current mirror

m0 inc inc 0 0 nch w=1u l=1u

m16 out inc n1 0  nch w=1u l=1u
m15 n1 inc n2 0 nch w=1u l=1u
m14 n2 inc n3 0 nch w=1u l=1u
m13 n3 inc n4 0 nch w=1u l=1u
m12 n4 inc n5 0 nch w=1u l=1u
m11 n5 inc n6 0 nch w=1u l=1u
m10 n6 inc n7 0 nch w=1u l=1u
m9 n7 inc n8 0 nch w=1u l=1u
m8 n8 inc n9 0 nch w=1u l=1u
m7 n9 inc n10 0 nch w=1u l=1u
m6 n10 inc n11 0 nch w=1u l=1u
m5 n11 inc n12 0 nch w=1u l=1u
m4 n12 inc n13 0 nch w=1u l=1u
m3 n13 inc n14 0 nch w=1u l=1u
m2 n14 inc n15 0 nch w=1u l=1u
m1 n15 inc 0 0 nch w=1u l=1u

i1 0 inc type=idc idc=16e-6
v1 out 0 type=vdc vdc=5

.model ekv nch TYPE=n VTO=.4 KP=400e-6

.op
```

Simulation results

```
* 1-TO-1/16TH DOWN-SCALING CURRENT MIRROR
Starting op analysis:
Calculating guess: done.
Solving with Gmin:
Building Gmin matrix...
Solving...  done.
Solving without Gmin:
Solving...  done.
Difference check is within margins.
(Voltage: er=0.001, ea=1e-06, Current: er=0.001, ea=1e-09)
Solution without Gmin:
Vinc:      0.904813615968  V
Vout:                 5.0  V
Vn1:       0.222041524366  V
Vn2:       0.183949114562  V
Vn3:       0.158276458021  V
Vn4:       0.137871535291  V
Vn5:       0.120520118975  V
Vn6:       0.105208756242  V
Vn7:      0.0913779977097  V
Vn8:      0.0786820153752  V
Vn9:       0.066890054189  V
Vn10:     0.0558393890315  V
```

```
Vn11:     0.0454103544576  V
Vn12:     0.0355120049651  V
Vn13:     0.0260733625457  V
Vn14:     0.0170378079838  V
Vn15:     0.0083593406432  V
I(V1):  -1.0327588469e-06  A
OP INFORMATION:
M0         N ch   STRONG INVERSION                      SATURATION
beta  [A/V^2]:  0.000193555471162  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.03
Vdb        [V]:      0.904813615968   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                 0.4   VOD  [V]:      0.369325572375   nq:
Ids        [A]:  1.59999863615e-05  nv:               1.51466221222  Ispec      [A]:
gmg        [S]:    8.073972779e-05   gms  [S]:  -0.000129001766756    rob    [Ohm]:
if:                  20.1828675005   ir:              0.25276934725     Qf  [C/m^2]:
-------------------
M16        N ch  MODERATE INVERSION                      SATURATION
beta  [A/V^2]:   0.000236055431981  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  8.4
Vdb        [V]:                  5.0   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                  0.4   VOD  [V]:     0.0330076658728   nq:
Ids        [A]:   1.03275797244e-06  nv:               1.51466221222  Ispec      [A]:
gmg        [S]:    1.3598388406e-05   gms  [S]:  -2.06195194629e-05    rob    [Ohm]:
if:                  1.05552698204   ir:         0.000703390112833     Qf  [C/m^2]:
-------------------
M15        N ch  MODERATE INVERSION                          LINEAR
beta  [A/V^2]:   0.000196221608214  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.01
Vdb        [V]:      0.222041524366   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                  0.4   VOD  [V]:    0.0907047995749   nq:
Ids        [A]:   1.03275775078e-06  nv:               1.51466221222  Ispec      [A]:
gmg        [S]:   9.76227671625e-06   gms  [S]:  -3.5529830659e-05    rob    [Ohm]:
if:                  2.33331263359   ir:               1.0643556238     Qf  [C/m^2]:
-------------------
M14        N ch  MODERATE INVERSION                          LINEAR
beta  [A/V^2]:   0.000197175919967  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.0
Vdb        [V]:      0.183949114562   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                  0.4   VOD  [V]:      0.129590202326   nq:
Ids        [A]:   1.03275801302e-06  nv:               1.51466221222  Ispec      [A]:
gmg        [S]:   7.48320262966e-06   gms  [S]:  -4.69804206758e-05    rob    [Ohm]:
if:                  3.60776881968   ir:              2.34495311438     Qf  [C/m^2]:
-------------------
M13        N ch  MODERATE INVERSION                          LINEAR
beta  [A/V^2]:   0.000197622982017  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.0
Vdb        [V]:      0.158276458021   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                  0.4   VOD  [V]:      0.160496767728   nq:
Ids        [A]:   1.03275814052e-06  nv:               1.51466221222  Ispec      [A]:
gmg        [S]:   6.30593158677e-06   gms  [S]:  -5.66430103458e-05    rob    [Ohm]:
if:                   4.8813573262   ir:              3.62139820283     Qf  [C/m^2]:
-------------------
M12        N ch  MODERATE INVERSION                          LINEAR
beta  [A/V^2]:   0.000197890926494  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.0
Vdb        [V]:      0.137871535291   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                  0.4   VOD  [V]:       0.18677830235   nq:
Ids        [A]:   1.03275822015e-06  nv:               1.51466221222  Ispec      [A]:
gmg        [S]:    5.5540108543e-06   gms  [S]:  -6.51632153734e-05    rob    [Ohm]:
if:                  6.15483579078   ir:              4.89658255608     Qf  [C/m^2]:
-------------------
M11        N ch  MODERATE INVERSION                          LINEAR
beta  [A/V^2]:   0.000198073131348  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.0
```

```
Vdb       [V]:      0.120520118975   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH       [V]:                 0.4   VOD  [V]:        0.2099698449   nq:
Ids       [A]:   1.03275827611e-06  nv:               1.51466221222  Ispec      [A]:
gmg       [S]:   5.01998756573e-06   gms  [S]:  -7.28717299041e-05    rob    [Ohm]:
if:                  7.42849392389   ir:              6.17139807159     Qf  [C/m^2]:
-------------------
M10       N ch  MODERATE INVERSION                           LINEAR
beta  [A/V^2]:   0.000198206852699  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.0
Vdb       [V]:      0.105208756242   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH       [V]:                 0.4   VOD  [V]:      0.230918772215   nq:
Ids       [A]:   1.03275831829e-06  nv:               1.51466221222  Ispec      [A]:
gmg       [S]:   4.61535219563e-06   gms  [S]:  -7.99649018188e-05    rob    [Ohm]:
if:                  8.70245034288   ir:              7.44620254594     Qf  [C/m^2]:
-------------------
M9        N ch  MODERATE INVERSION                           LINEAR
beta  [A/V^2]:   0.000198310138598  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.0
Vdb       [V]:     0.0913779977097   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH       [V]:                 0.4   VOD  [V]:      0.250148896904   nq:
Ids       [A]:   1.03275835162e-06  nv:               1.51466221222  Ispec      [A]:
gmg       [S]:   4.29500867285e-06   gms  [S]:  -8.65706441092e-05    rob    [Ohm]:
if:                  9.97675042417   ir:              8.72115687844     Qf  [C/m^2]:
-------------------
M8        N ch   STRONG INVERSION                          LINEAR
beta  [A/V^2]:  0.000198392895287  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.00
Vdb       [V]:    0.0786820153752   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH       [V]:                0.4   VOD  [V]:      0.268009734921   nq:
Ids       [A]:  1.03275837884e-06  nv:               1.51466221222  Ispec      [A]:
gmg       [S]:  4.03321312301e-06   gms  [S]:  -9.27778104751e-05    rob    [Ohm]:
if:                 11.2514053587   ir:              9.99633553235     Qf  [C/m^2]:
-------------------
M7        N ch   STRONG INVERSION                         LINEAR
beta  [A/V^2]:   0.00019846105755  Weff  [m]:      1e-06 (1e-06)   Leff      [m]:  1.007
Vdb       [V]:     0.066890054189   Vgb  [V]:     0.904813615968    Vsb      [V]:
VTH       [V]:                0.4   VOD  [V]:     0.284747759855   nq:
Ids       [A]:  1.03275840164e-06  nv:              1.51466221222  Ispec      [A]:
gmg       [S]:  3.81404706427e-06   gms  [S]:  -9.8651106306e-05    rob    [Ohm]:
if:                 12.5264097287   ir:             11.2717709335     Qf  [C/m^2]:
-------------------
M6        N ch   STRONG INVERSION                          LINEAR
beta  [A/V^2]:  0.000198518419499  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.00
Vdb       [V]:    0.0558393890315   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH       [V]:                0.4   VOD  [V]:      0.300544224434   nq:
Ids       [A]:   1.0327584211e-06  nv:               1.51466221222  Ispec      [A]:
gmg       [S]:  3.62706421814e-06   gms  [S]:  -0.000104239391439    rob    [Ohm]:
if:                 13.8017500679   ir:              12.5474737773     Qf  [C/m^2]:
-------------------
M5        N ch   STRONG INVERSION                          LINEAR
beta  [A/V^2]:  0.000198567531583  Weff  [m]:       1e-06 (1e-06)   Leff      [m]:  1.00
Vdb       [V]:    0.0454103544576   Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH       [V]:                0.4   VOD  [V]:      0.315536880374   nq:
Ids       [A]:  1.03275843798e-06  nv:               1.51466221222  Ispec      [A]:
gmg       [S]:   3.4650857034e-06   gms  [S]:  -0.000109580636704    rob    [Ohm]:
if:                 15.0774092523   ir:              13.8234431638     Qf  [C/m^2]:
-------------------
M4        N ch   STRONG INVERSION                        LINEAR
beta  [A/V^2]:  0.000198610178219  Weff  [m]:     1e-06 (1e-06)   Leff      [m]:  1.0069
Vdb       [V]:    0.0355120049651   Vgb  [V]:    0.904813615968    Vsb      [V]:
```

```
VTH        [V]:                 0.4   VOD  [V]:    0.329833235381   nq:
Ids        [A]:  1.03275845282e-06   nv:            1.51466221222  Ispec      [A]:
gmg        [S]:  3.32299269557e-06    gms  [S]:  -0.0001147050485    rob    [Ohm]:
if:                  16.353368823    ir:            15.0996719747     Qf  [C/m^2]:
-------------------
M3         N ch   STRONG INVERSION                         LINEAR
beta  [A/V^2]:  0.000198647649302   Weff  [m]:        1e-06 (1e-06)    Leff      [m]:  1.00
Vdb        [V]:    0.0260733625457    Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                 0.4   VOD  [V]:      0.343519048442   nq:
Ids        [A]:  1.03275846598e-06   nv:              1.51466221222  Ispec      [A]:
gmg        [S]:  3.19702338781e-06    gms  [S]:  -0.000119637126799    rob    [Ohm]:
if:                  17.6296102286    ir:             16.3761498503     Qf  [C/m^2]:
-------------------
M2         N ch   STRONG INVERSION                         LINEAR
beta  [A/V^2]:  0.000198680902931   Weff  [m]:        1e-06 (1e-06)    Leff      [m]:  1.00
Vdb        [V]:    0.0170378079838    Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                 0.4   VOD  [V]:      0.356663994983   nq:
Ids        [A]:  1.03275847775e-06   nv:              1.51466221222  Ispec      [A]:
gmg        [S]:  3.08434271957e-06    gms  [S]:  -0.000124397070523    rob    [Ohm]:
if:                  18.9061154837    ir:             17.6528648853     Qf  [C/m^2]:
-------------------
M1         N ch   STRONG INVERSION                         LINEAR
beta  [A/V^2]:  0.000198710667278   Weff  [m]:        1e-06 (1e-06)    Leff      [m]:  1.00
Vdb        [V]:    0.0083593406432    Vgb  [V]:      0.904813615968    Vsb      [V]:
VTH        [V]:                 0.4   VOD  [V]:      0.369325572375   nq:
Ids        [A]:  1.03275848844e-06   nv:              1.51466221222  Ispec      [A]:
gmg        [S]:  2.98276797184e-06    gms  [S]:  -0.000129001766756    rob    [Ohm]:
if:                  20.1828675005    ir:             18.9298046103     Qf  [C/m^2]:
-------------------
TOTAL POWER: 1.964081209e-05 W
```

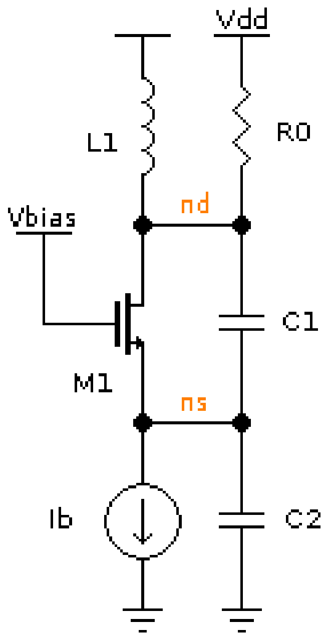
The I_{OUT}/I_{IN} scaling factor is as expected, since $I_{OUT} = 1uA$ when $I_{IN} = 16uA$. Furthermore, the results regarding the subdivision of the drain current in i_f/i_r and the mirroring of currents in neighboring devices agree with the expectations as well.

Lastly, notice how only `M0` and `M16` operate in saturation, all other transistors are in linear region.

2.2.2 Transient example

Introduction

This is an example of transient simulation, featuring the well-known Colpitts oscillator.



Bias:

$V_{dd} = 2.5V$, $V_{bias} = 2V$.

Passives:

$L_1 = 5nH$, $R_0 = 3.14k\Omega$ ($Q_L = 33$ @ 3 GHz)

$C_1 = C_2 = 1.012pF$ (resulting in $n = C_1/(C_1 + C_2) = 0.5$)

C_2 has a Q greater than 10 for every I_b less than $(w_0 C_2)^2/(2K10^2)$ = 4.8 mA.

Under such condition, the minimum transconductance required for oscillation can be calculated considering the MOS transistor an ideal voltage probe. The presence of the MOS has to be taken into account when evaluating the overall Q of the tank.

Under such hypothesis: $g_{m,min} = 1/(n(1-n)R0)$, $I_{b,min} = 1.27$ mA. In the following we use I_b = 1.3mA.

Netlist-based simulation

Netlist

```
MOS COLPITTS OSCILLATOR

vdd dd 0 type=vdc vdc=2.5

* Ql = 33 at 3GHz
l1 dd nd 5n ic=-1n
r0 nd dd 3.5k

* n = 0.5, f0 = 3GHz
c1 nd ns 1.12p ic=2.5
c2 ns 0  1.12p *ic=.01
```

```
m1 nd1 bias ns ns nmos w=200u l=1u
vtest nd nd1 type=vdc vdc=0 *read current

* Bias
vbias bias 0 type=vdc vdc=2
ib ns 0 type=idc idc=1.3m

.model ekv nmos TYPE=n VTO=.4 KP=10e-6

.op
.tran tstop=150n tstep=.1n method=trap uic=2
.plot tran v(nd)
```

The voltage generator Vtest has been added to the circuit in series with M1's drain to add the drain current to the variables.

We need to simulate the circuit for roughly $T \approx 10 Q_L / f_0$ = 110ns to approach the steady state solution. The simulation above stops at t = 150ns.

Running the simulation

Save the netlist in a file and start ahkab with:

```
$ ahkab colpitts_mos.spc -o colpitts_graph
```

The simulation takes 70s on my laptop. Set `tstop` to a lower value to make it faster.

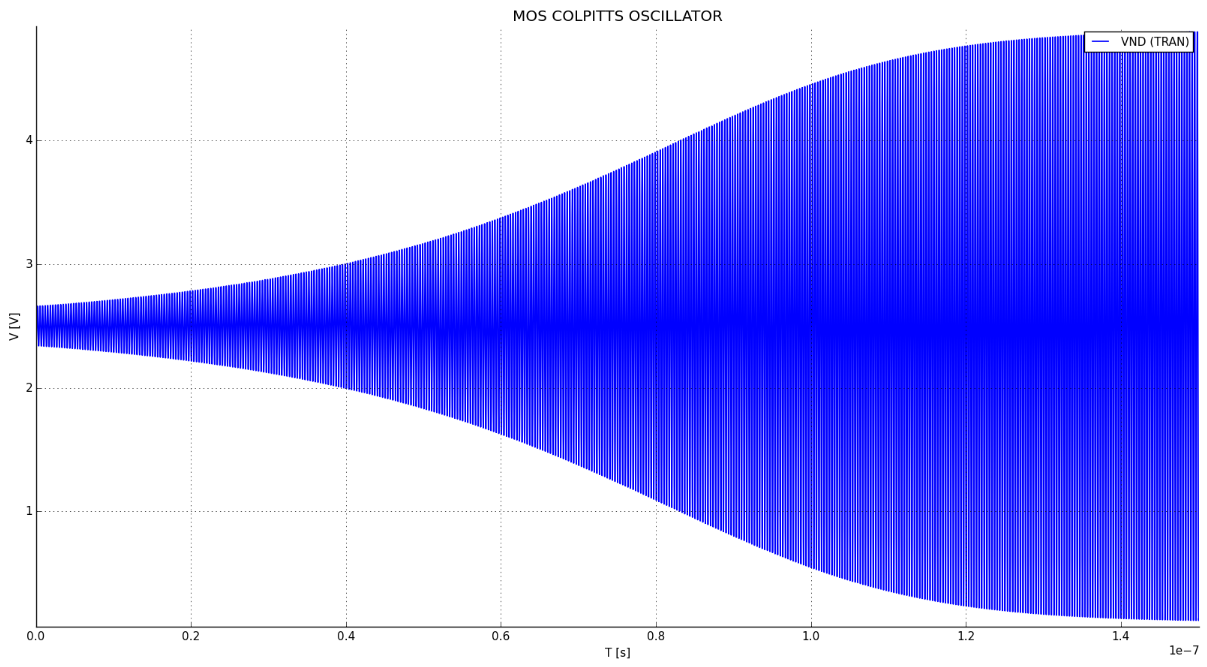
Results

Operating point (.OP) The operating point is shown in this section of the program output:

```
2015-02-23 15:28:37
ahkab v. 0.13 (c) 2006-2015 Giuseppe Venturini

Operating Point (OP) analysis

Netlist: colpitts_mos.ckt
Title: MOS colpitts oscillator
At 300.00 K
Options:
    vea = 1.000000e-06
    ver = 0.001000
    iea = 1.000000e-09
    ier = 0.001000
    gmin = 0.000000e+00

Convergence reached in 42 iterations.

========
RESULTS:
========

Variable        Value         Error    (%)       Units
----------   ---------   ------------  -------   -------
VDD            2.5       -2.5e-12      (-0.00)   V
```

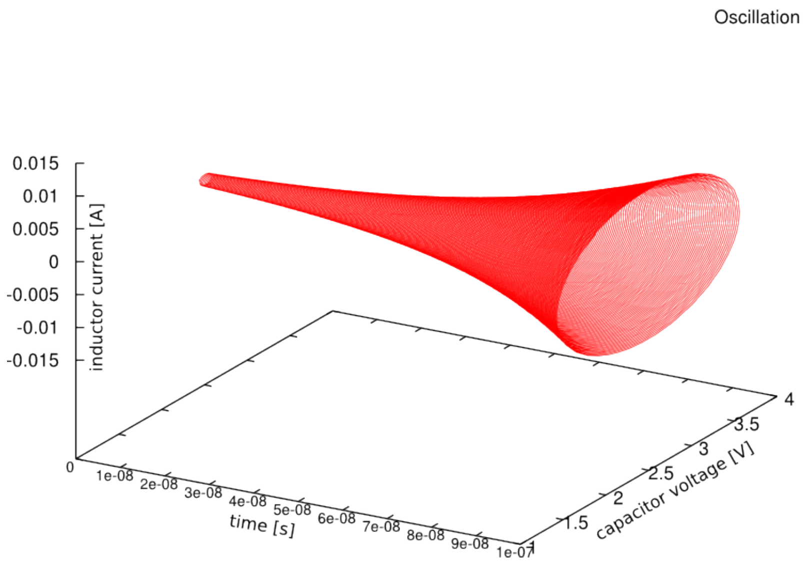
```
VND          2.5       -2.5e-12      (-0.00)  V
VNS         -0.116201   2.48567e-10  (-0.00)  V
VND1         2.5       -2.50951e-10  (-0.00)  V
VBIAS        2         -2e-12        (-0.00)  V
I(VDD)      -0.0013     0            (-0.00)  A
I(L1)        0.0013     0            (0.00)   A
I(VTEST)     0.0013     0            (0.00)   A
I(VBIAS)     0          0            (0.00)   A

========================
ELEMENTS OP INFORMATION:
========================

Part ID      V(n1-n2) [V]         Q [C]        E [J]
---------  --------------  ------------  -----------
C1                 2.6162     2.93015e-12  3.83292e-12
C2              -0.116201  -1.30145e-13  7.56149e-15

Part ID      V(n1-n2) [V]    I [A]         P [W]
---------  --------------  -------  ------------
IB              -0.116201   0.0013  -0.000151061

Part ID      (n1,n2) [Wb]    I(n1->n2) [A]      E [J]
---------  ---------------  ---------------  ---------
L1                 6.5e-12           0.0013  4.225e-15

----  --------  -----------------  ----  ----  -----------------  -----  --------  -----
m1    N ch      STRONG INVERSION               LINEAR
beta  [A/V^2]:  0.000746574470194  Weff  [m]:  0.0002 (0.0002)    Leff   [m]:      1.339
Vdb   [V]:      2.616200987        Vgb   [V]:  2.116200987        Vsb    [V]:      0.0
VTH   [V]:      0.4                VOD   [V]:  1.61188671993      nq:              1.440
Ids   [A]:      0.00129999988445   nv:         1.36453957998      Ispec  [A]:      3.849
gmg   [S]:      0.00184989038878   gms   [S]:  -0.00317451824956  rob    [Ω]:       1537
if:             475.729938341      ir:         23.4335578457      Qf     [C/m^2]:  0.001
----  --------  -----------------  ----  ----  -----------------  -----  --------  -----

Part ID      R [Ω]    V(n1,n2) [V]    I(n1->n2) [A]    P [W]
---------  -------  --------------  ---------------  -------
R0            3500               0                0        0

Part ID      V(n1,n2) [V]    I(n1->n2) [A]     P [W]
---------  --------------  ---------------  --------
VDD                   2.5          -0.0013  -0.00325
VTEST                 0             0.0013   0
VBIAS                 2             0        0
```

Transient simulation (.TRAN) The oscillation builds up quickly, as shown in this plot of V_{nd}:



From inspection, the circuit oscillates at 3.002 GHz with an oscillation amplitude of roughly 4V.

The next plot shows the oscillation starting off from the very beginning in a phase plane:



API-based simulation

As an exercise, we will show here also how to perform a similar simulation taking advantage of the Python API.

Python script

```
import ahkab
import pylab

osc = ahkab.Circuit('MOS COLPITTS OSCILLATOR')
```

```
# models need to be defined before the devices that use them
osc.add_model('ekv', 'nmos', dict(TYPE='n', VTO=.4, KP=10e-6))

osc.add_vsource('vdd', n1='dd', n2=osc.gnd, dc_value=3.3)

# Ql = 33 at 3GHz
osc.add_inductor('l1', n1='dd', n2='nd', value=5e-9, ic=-1e-9)
osc.add_resistor('r0', n1='nd', n2='dd', value=3.5e3)

# n = 0.5, f0 = 3GHz
osc.add_capacitor('c1', n1='nd', n2='ns', value=1.12e-12)
osc.add_capacitor('c2', n1='ns', n2=osc.gnd, value=1.12e-12)

osc.add_mos('m1', nd='nd1', ng='bias', ns='ns', nb='ns',
            model_label='nmos', w=600e-6, l=100e-9)
# voltage source as a current probe
osc.add_vsource('vtest', n1='nd', n2='nd1', dc_value=0)

# Bias
osc.add_vsource('vbias', n1='bias', n2=osc.gnd, dc_value=2.)
osc.add_isource('ib', n1='ns', n2=osc.gnd, dc_value=1.3e-3)

# calculate an Operating Point (OP) to initialize the transient
# analysis
op = ahkab.new_op()
res = ahkab.run(osc, op)

# modify the OP to give the circuit a little kick to start the
# oscillation
x0 = res['op'].asmatrix()
l1vdei = osc.find_vde_index('l1')
l1i = len(osc.nodes_dict) - 1 + l1vdei
x0[l1i, 0] += -1e-9

# Setup and run a transient analysis with the modified x0 as start point
tran = ahkab.new_tran(tstart=0., tstop=20e-9, tstep=.01e-9, method='trap',
                      x0=x0)
res = ahkab.run(osc, tran)['tran']

# plot the results!
pylab.subplot(211)
pylab.hold(True)
pylab.plot(res.get_x(), res['vnd'], label='ND')
pylab.plot(res.get_x(), res['vns'], label='NS')
pylab.plot(res.get_x(), res['vbias'], label='BIAS')
pylab.legend()
pylab.subplot(212)
pylab.plot(res.get_x(), res['i(vtest)'], label='I(VTEST)')
pylab.legend()
pylab.show()
```

As we have increased in the above the `W` of M1 and therefore its g_m, the oscillation will build up faster and to a higher top amplitude.

Running the simulation

To run the simulation, just save the above code to a file, for example `colp.py` and run:

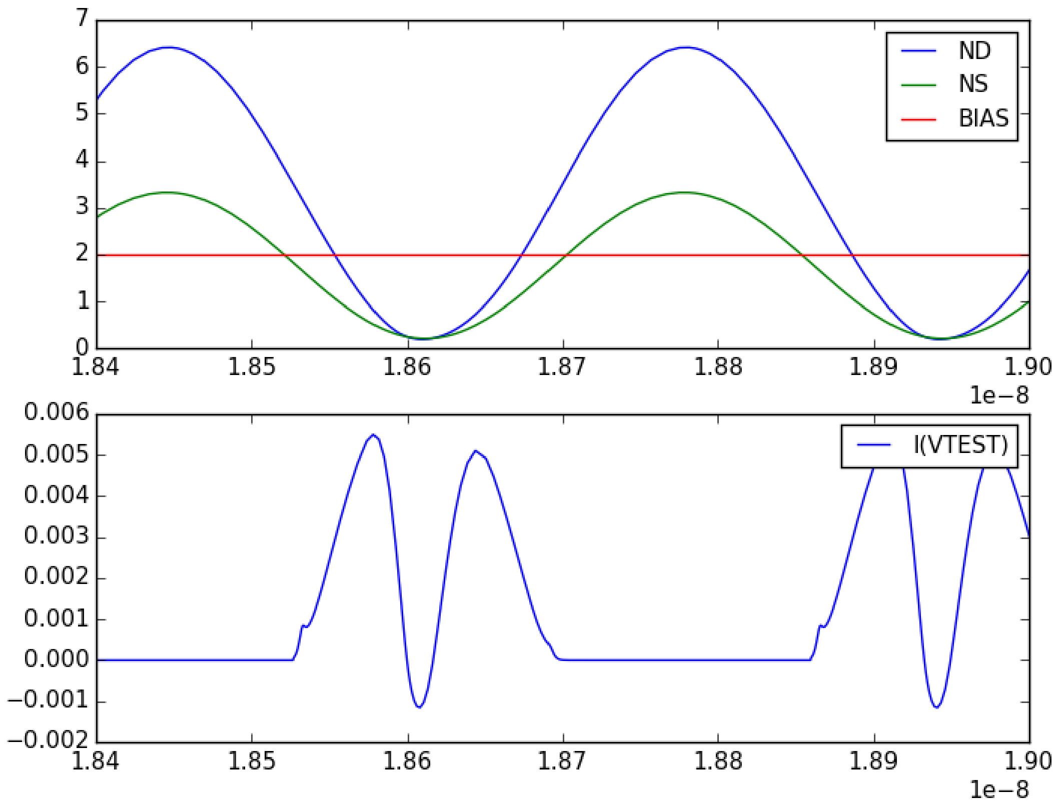
```
python colp.py
```

If `matplotlib` is available and set up correctly, a graph should pop up in a little while.

Results

The OP is not shown here, it can be printed with `res['op'].write_to_file('stdout')`, but more interesting is manipulating the raw data with `res['op'].asmatrix()`.

The following graph shows the gate, drain and source voltages of the MOS transistor, along with its drain current. M1 is on only for a fraction of each period, this happens if I_b is greater than approx. $1.5 I_{b,min}$.



It can be shown that an increase in I_b increases the oscillation amplitude. When the oscillation amplitude (at `nd`) approaches V_{dd}, a damping will appear at the middle of the current peak, because $V_{ds} = V_{nd}$ - V_{ns} will be near to zero. If the oscillation amplitude increases further V_{ds} crosses 0V and becomes negative for a small period of time. Accordingly, I_d crosses 0A and becomes negative for such period.

Of course, in any case, the average current through M1 has to be equal to I_b. In fact:

```
>> print(res['i(vtest)'].mean())
0.00119411417458
```

Which is close enough counting that it is calculated over a fractional number of periods.

During a period, M1 is always on, switching from saturation region ($Vgs > Vt$, $Vgd < Vt$) to ohmic operation (channel at both source and drain). The latter happens when I_d is maximum.

2.2.3 Symbolic examples

Introduction

A small signal symbolic analysis is requested through the `.symbolic` directive.

Its syntax is:

```
.symbolic [tf=<source_name> ac=<bool> r0s=<bool>]
```

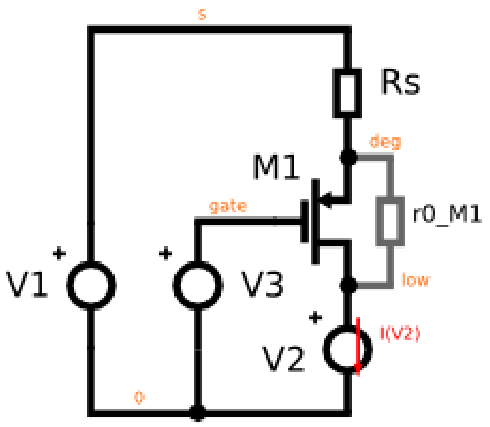
If the source is specified, all results are differentiated with respect to the source value, to obtain transfer functions.

If `ac` is set to `1`, `True` or `yes`, capacitors and inductors will be taken into account. If `r0s` is set to `True` or one of its synonyms, the output resistances of the transistors will be considered.

See also:

The symbolic analysis section in Netlist Syntax, the module *ahkab.symbolic* and the helper function *ahkab.ahkab.new_symbolic()*.

Output resistance of a degenerated MOS transistor



Let’s say we wish to check the expression of the output resistance of the circuit in the figure above, the small signal $-\mathrm{V}_2/\mathrm{I}[\mathrm{V}_2]$.

Save the following netlist to a file, in the following the name of this file is assumed to be “rd.ckt”.

```
* Output resistance of a degenerated MOS transistor
m1 low gate deg 0 pch w=1u l=1u
rs deg s 1k
v1 gate 0 type=vdc vdc=1
v3 s 0 type=vdc vdc=1
v2 low 0 type=vdc vdc=1

.model ekv pch type=p kp=10e-6 vto=-1
.symbolic tf=v2 r0s=1
```

Start ahkab with:

```
./ahkab rd.ckt
```

Results

```
* OUTPUT RESISTANCE OF A DEGENERATED MOS TRANSISTOR
Starting symbolic DC...
Building symbolic MNA, N and x...  done.
Building equations...
Performing auxiliary simplification...
Auxiliary simplification solved the problem.
Success!
[ ... very long lines omitted  ... ]
Calculating small-signal symbolic transfer functions (v2))... done.
Small-signal symbolic transfer functions:
I_[V1]/v2 = 0
    DC: 0
I_[V2]/v2 = -1.0/(R_s*gm_M1*r0_M1 + R_s + r0_M1)
    DC: -1.0/(R_s*gm_M1*r0_M1 + R_s + r0_M1)
I_[V3]/v2 = 1.0/(R_s*gm_M1*r0_M1 + R_s + r0_M1)
    DC: 1.0/(R_s*gm_M1*r0_M1 + R_s + r0_M1)
V_deg/v2 = 1.0*R_s/(R_s*gm_M1*r0_M1 + R_s + r0_M1)
    DC: 1.0*R_s/(R_s*gm_M1*r0_M1 + R_s + r0_M1)
V_gate/v2 = 0
    DC: 0
V_low/v2 = 1.00000000000000
    DC: 1.00000000000000
V_s/v2 = 0
    DC: 0
```

The simulator solves the circuit symbolically and the differentiates the results according to the transfer function requested.

We wish to know the transfer function between V_2 and $-I[V_2]$, rearranging the results above:

$$R_{out} = -\frac{dV_2}{dI[V_2]} = r_{0,\,M_1} + R_s + g_{m,\,M_1} r_{0,\,M_1} R_s$$

Resistance seen at the source of a transistor with a resistor at the drain

Let's evaluate the symmetric circuit to the previous one: this time the resistor is located at the drain (Rd) and we connect the test voltage source at the transistor source node.

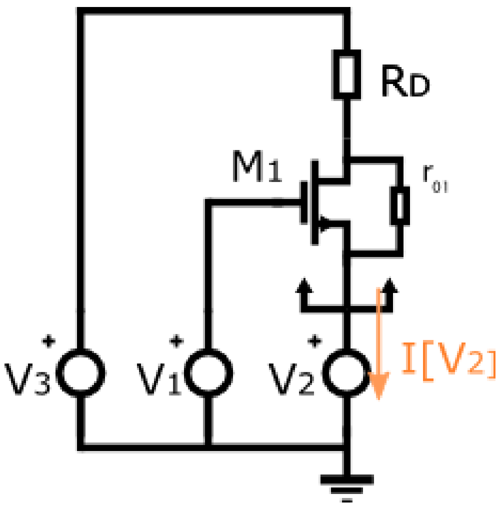
We wish to verify the famous result:

$$R_{out} = \frac{r_0 + R_D}{1 + g_m r_0}$$

Which evaluates to $1/g_m$ if:

- $g_m r_0 >> 1$,
- $r_0 >> R_D$.

The circuit we wish to simulate to extract the equivalent resistance is:



which can be described with the netlist:

```
* Resistance seen at the source of a
* transistor with a resistor at the drain
m1 low gate deg 0 pch w=1u l=1u
rd low s 1k
v1 gate 0 type=vdc vdc=1
v3 s 0 type=vdc vdc=2
v2 deg 0 type=vdc vdc=1

.model ekv pch type=p kp=10e-6 vto=-1
.symbolic tf=v2 r0s=1
```

Running ahkab just like in the example before, we get:

```
Starting symbolic AC analysis...
Building symbolic MNA, N and x...  done.
Building equations...
Solving...
Success!
Results:
I[V1]    = 0
I[V2]    = (V1*gm_m1*r0_m1 - V2*gm_m1*r0_m1 - V2 + V3)/(RD*(1 + r0_m1/RD))
I[V3]    = (-V1*gm_m1*r0_m1 + V2*(gm_m1*r0_m1 + 1) - V3)/(RD*(1 + r0_m1/RD))
Vdeg     = V2
Vgate    = V1
Vlow     = (-V1*gm_m1*r0_m1 + V2*(gm_m1*r0_m1 + 1) + V3*r0_m1/RD)/(1 + r0_m1/RD)
Vs   = V3
Calculating small-signal symbolic transfer functions (V2))... done.
Small-signal symbolic transfer functions:
I[V1]/V2 = 0
    DC: 0
I[V2]/V2 = (-gm_m1*r0_m1 - 1)/(RD + r0_m1)
    DC: (-gm_m1*r0_m1 - 1)/(RD + r0_m1)
I[V3]/V2 = (gm_m1*r0_m1 + 1)/(RD + r0_m1)
    DC: (gm_m1*r0_m1 + 1)/(RD + r0_m1)
Vdeg/V2 = 1
    DC: 1
Vgate/V2 = 0
    DC: 0
Vlow/V2 = RD*(gm_m1*r0_m1 + 1)/(RD + r0_m1)
    DC: RD*(gm_m1*r0_m1 + 1)/(RD + r0_m1)
```

```
Vs/V2 = 0
    DC: 0
```

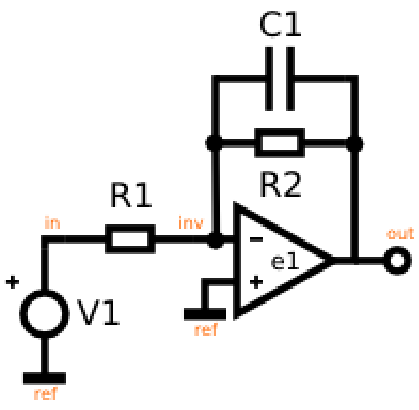
Where the transfer function we are looking for is $-V_2/I[V_2]$.

After rearranging, we get:

$$-\frac{dV_2}{dI[V_2]} = \frac{R_D + r_{0,\,M_1}}{g_{m,\,M_1} r_{0,\,M_1} + 1}$$

Just like it was expected.

Small-signal transfer function of various opamp configurations



Integrator with finite gain

The ideal integrator is the configuration shown above with no R_2 resistor. It has a transfer function equal to $T(s) = K/s$ for all frequencies. Notice the infinite zero-frequency gain.

A real integrator will have a transfer function differing from the one above because of many factors. One of which is the finite gain of every amplifier. This goes well with the simulator as it does not like “infinite quantities”.

Netlist:

```
PERFECT INTEGRATOR
v1 in 0 type=vdc vdc=1
r1 in inv 1k
e1 out 0 0 inv 1e6
c1 inv out 1p

.symbolic tf=v1 ac=1
```

If the amplifier has a gain equal to e_1, then, skipping to the results, we get:

```
I[E1]     = (C1*s*v1 + C1*e1*s*v1)/(1 + C1*R1*s + C1*R1*e1*s)
I[V1]     = -(C1*s*v1 + C1*e1*s*v1)/(1 + C1*R1*s + C1*R1*e1*s)
Vin  = v1
Vinv      = v1/(1 + C1*R1*s + C1*R1*e1*s)
Vout      = -e1*v1/(1 + C1*R1*s + C1*R1*e1*s)
Calculating symbolic transfer functions (v1)... done!
d/dv1 I[E1] = (C1*s + C1*e1*s)/(1 + C1*R1*s + C1*R1*e1*s)
    DC: 0
    P0: 1/(-C1*R1 - C1*R1*e1)
    Z0: 0
d/dv1 I[V1] = -(C1*s + C1*e1*s)/(1 + C1*R1*s + C1*R1*e1*s)
    DC: 0
    P0: 1/(-C1*R1 - C1*R1*e1)
    Z0: 0
d/dv1 Vin = 1
    DC: 1
d/dv1 Vinv = 1/(1 + C1*R1*s + C1*R1*e1*s)
    DC: 1
    P0: 1/(-C1*R1 - C1*R1*e1)
d/dv1 Vout = -e1/(1 + C1*R1*s + C1*R1*e1*s)
    DC: -e1
    P0: 1/(-C1*R1 - C1*R1*e1)
```

dV_{out}/dv_1 is what we are interested in, here: the DC gain increases proportionally to e_1 and the position of the low frequency pole moves back towards DC with e_1 increasing.

Leaky integrator with finite gain

If we introduce a resistor R_2 shunting the capacitor, we get a low frequency amplifier, which approximately behaves like an amplifier with constant gain $-R_2/R_1$ before $\omega = -1/(C_1 R_2)$, then the gain decreases by 20dB/decade.

Netlist:

```
LEAKY INTEGRATOR WITH FINITE GAIN
v1 in 0 type=vdc vdc=1
r1 in inv 1k
e1 out 0 0 inv 1e6
c1 inv out 1p
r2 inv out 1k

.symbolic tf=v1 ac=1
```

From the simulation:

```
I[E1]    = (v1 + e1*v1 + C1*R2*s*v1 + C1*R2*e1*s*v1)/(R1 + R2 + R1*e1 + C1*R1*R2*s + C1*
I[V1]    = (v1 + e1*v1 + C1*R2*s*v1 + C1*R2*e1*s*v1)/(-R1 - R2 - R1*e1 - C1*R1*R2*s - C1
Vin  = v1
Vinv     = R2*v1/(R1 + R2 + R1*e1 + C1*R1*R2*s + C1*R1*R2*e1*s)
Vout     = R2*e1*v1/(-R1 - R2 - R1*e1 - C1*R1*R2*s - C1*R1*R2*e1*s)
Calculating symbolic transfer functions (v1)... done!
d/dv1 I[E1] = (1 + e1 + C1*R2*s + C1*R2*e1*s)/(R1 + R2 + R1*e1 + C1*R1*R2*s + C1*R1*R2*e
    DC: (1 + e1)/(R1 + R2 + R1*e1)
    P0: (R1 + R2 + R1*e1)/(-C1*R1*R2 - C1*R1*R2*e1)
    Z0: -1/(C1*R2)
d/dv1 I[V1] = (1 + e1 + C1*R2*s + C1*R2*e1*s)/(-R1 - R2 - R1*e1 - C1*R1*R2*s - C1*R1*R2*
    DC: (1 + e1)/(-R1 - R2 - R1*e1)
    P0: (R1 + R2 + R1*e1)/(-C1*R1*R2 - C1*R1*R2*e1)
    Z0: -1/(C1*R2)
d/dv1 Vin = 1
    DC: 1
d/dv1 Vinv = R2/(R1 + R2 + R1*e1 + C1*R1*R2*s + C1*R1*R2*e1*s)
    DC: R2/(R1 + R2 + R1*e1)
    P0: (R1 + R2 + R1*e1)/(-C1*R1*R2 - C1*R1*R2*e1)
```

```
d/dv1 Vout = R2*e1/(-R1 - R2 - R1*e1 - C1*R1*R2*s - C1*R1*R2*e1*s)
    DC: R2*e1/(-R1 - R2 - R1*e1)
    P0: (R1 + R2 + R1*e1)/(-C1*R1*R2 - C1*R1*R2*e1)
```

If e_1 is indeed very high, the circuit results are as expected.

$$\frac{dV_{out}}{dV_1} \xrightarrow[e_1 \to +\infty]{} -\frac{R_2}{R_1}\frac{1}{1+sC_1R_2}$$

Effects of finite output resistance, differential input capacitance, finite input resistance can all be simulated similarly.

CHAPTER 3

Help pages on particular elements

3.1 Mutual Inductors

3.1.1 Introduction

This page explains briefly how to use coupling between inductors.

If you are familiar with SPICE's mutual inductors, you can skip this page, they work the same way.

3.1.2 Netlist syntax

```
K<string> <inductor1> <inductor2> <float>
```

or:

```
K<string> <inductor1> <inductor2> k=<float>
```

3.1.3 API syntax

If you have a circuit instance `my_circuit` containing two inductors with IDs `'LP'` and `'LS'`, you can add a coupling of value 0.89 between them with:

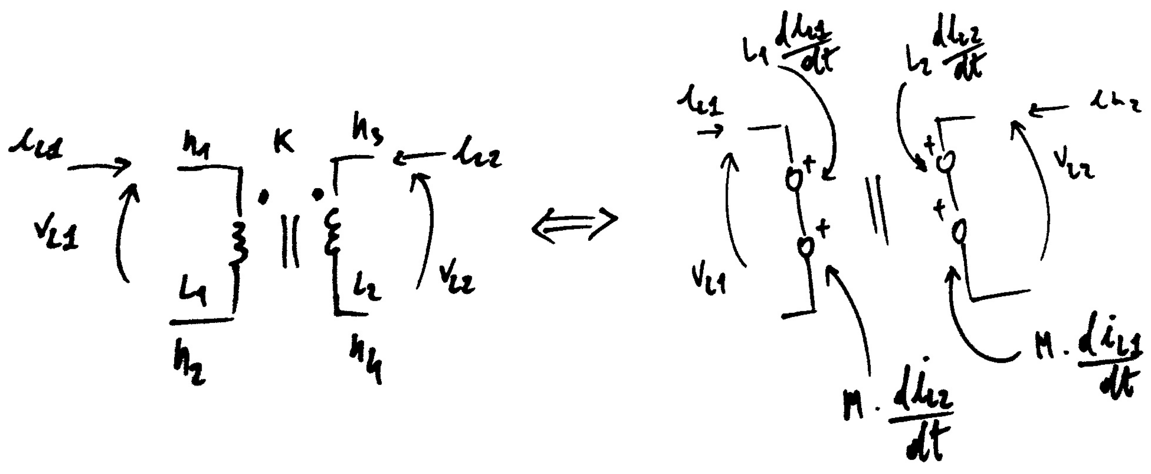
```
my_circuit.add_inductor_coupling(part_id='K1', L1='LP', L2='LS', value=.89)
```

For further information, refer to *`ahkab.circuit.Circuit.add_inductor_coupling()`*.

3.1.4 Usage and internal modeling

The coupling between two inductors is defined by the two inductors to be coupled and the value coupling factor k. *The coupling factor has to be lesser than one.*

Dot convention: for every inductor coupling, the dot is to be placed on the first node specified when the inductor was declared.



Eg. the left hand side of the figure above can be specified with the entries below:

```
L1 n1 n2 1u
L2 n3 n4 1u
K1 L1 L2 k=.2
```

Internally, the following equations are enforced (refer to the right hand side of the previous figure):

$$V_{L1} = L_1 \frac{dI(L_1)}{dt} + M \frac{dI(L_2)}{dt}$$
$$V_{L2} = L_2 \frac{dI(L_2)}{dt} + M \frac{dI(L_1)}{dt}$$

Where M is the *mutual inductance* and it is defined as:

$$M = K\sqrt{L_1 L_2}$$

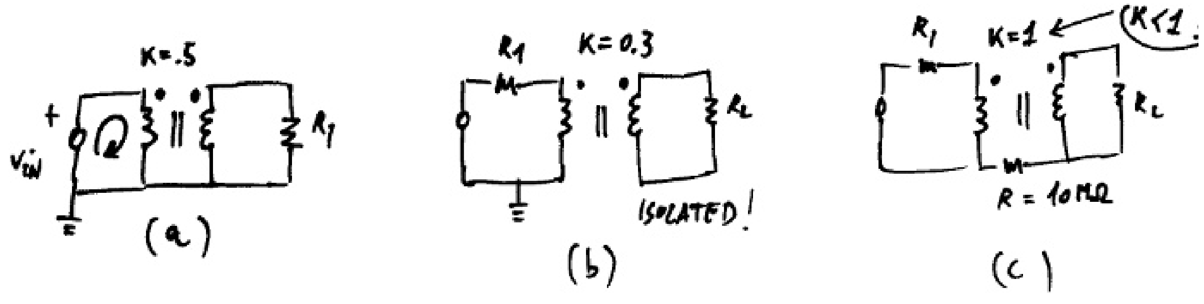
3.1.5 Ideal transformers

Ideal (perfect) transformers are not supported, but can be approximated with the following choices:

- Set $k = 0.999$ (an ideal transformer would have $k = 1$),
- Set the inductors values high enough that the primary and secondary inductances have a negligible effect on the current/voltages over the transformer. (an ideal tranformer would have "infinite" primary and secondary inductances),
- Set the ratio of the primary/secondary inductances L_1/L_2 equal to the windings ratio n_1/n_2.

3.1.6 Pathological circuits

A few pathological circuits are shown in the next figure.



Explanations:

(a) is pathological because two elements are specifying the transformer input node at the same time (think what would happen in real life...). The resulting MNA matrix is singular.

- insert a series resistor to break the loop.

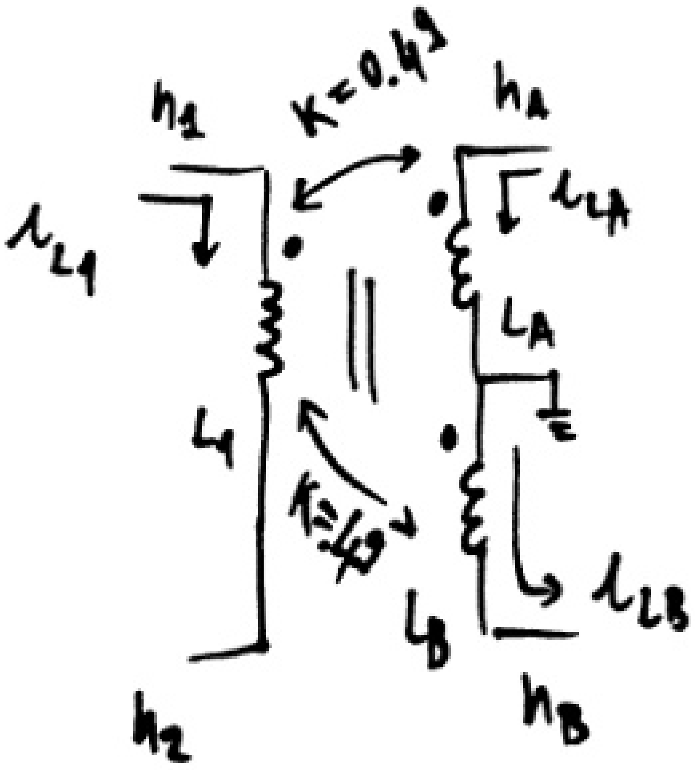
(b) corrects the issue above, but has an isolated secondary, which means that all the voltages at the secondary winding are not unequivocally defined. The resulting MNA is singular.

- join the primary and secondary with a very high isolation resistor or set the voltage of one node at the secondary with a voltage source.

(c) has $k = 1$. k has to be less than 1 or *instability ensues*.

3.1.7 Multiple coupling

It is possible to couple multiple inductors together, the following is an example of a transformer with a center tap (connected to ground in this case).



```
* Transformer with a grounded center tap:
* Primary: n1, n2
* Secondary 1: nA, 0
* Secondary 2: 0, nB

L1 n1 n2 10u
LA nA 0 5u
LB 0 nB 5u
K1 L1 LA .49
K1 L1 LB .49
```

3.1.8 Known limitations

- For the time being mutual inductors are unsupported in subcircuits.
- The inductors *have to be declared first.*

CHAPTER 4

Module reference

4.1 The ahkab core module

4.1.1 Introduction

This is the core module of the simulator. It provides helper functions to save you the need to call directly the functions in most submodules.

4.1.2 Do you have a circuit?

To run a simulation, you'll need a circuit first: a circuit can be described with a simulation deck or with a circuit object.

Define your circuit by means of a Circuit object

In a Python script, describing the circuit through the `ahkab.circuit.Circuit` interface is a very versatile a choice.

Refer to `ahkab.circuit.Circuit` for a complete description of the process and the documentation of several helper functions to assist you in this task.

You may then jump to *How to create a simulation object*.

Define your circuit by means of a netlist file

The circuit description can also be provided as a text file, also known as netlist deck, for historical reason. This file will also typically include simulation and post-processing directives, such as plotting.

The netlist should be described according to the rules in Netlist Syntax.

If you have a netlist (simulation deck) available, you have several possibilities.

The first, assuming your netlist defines some simulation would be to run it:

- you may call `ahkab` from the command line. The command line interface is described in Command line help.
- you may call `main()` directly from Python. Running the simulation through `main()` function allows to process the result in Python.

Alternatively, you may parse the netlist through `ahkab.netlist_parser.parse_circuit()`, which will return the circuit instance, all the simulations defined in the deck and all the post-processing directives as well.

You may now modify the circuit and simulation objects as you please, or create new ones, as well as run them as described in the *Run it!* section.

4.1.3 How to create a simulation object

Next, you need to have a simulation object you would like to run.

The following functions are available to quickly create a simulation object:

`new_ac`(start, stop, points[, x0, ...])	Assembles an AC analysis and returns the analysis object.
`new_dc`(start, stop, points, source[, ...])	Assembles a DC sweep analysis and returns the analysis object.
`new_op`([guess, x0, outfile, verbose])	Assembles an OP analysis and returns the analysis object.
`new_pss`(period[, x0, points, method, ...])	Assembles a Periodic Steady State (PSS) analysis and returns th
`new_pz`([input_source, output_port, shift, ...])	Assembles a Pole-Zero analysis and returns the analysis object.
`new_symbolic`([source, ac_enable, r0s, subs, ...])	Assembles a Symbolic analysis and returns the analysis object.
`new_tran`(tstart, tstop, tstep[, x0, method, ...])	Assembles a TRAN analysis and returns the analysis object.

Click on one of the above hyperlinks to be taken to the corresponding documentation section.

Note: The functions above allow you to specify an output file. This is due to two main reasons:

- Saving to a file allows you to keep a copy of the simulation results, which you can then inspect at a later time.
- Simulation results may take an uncomfortably large amount of memory. The approach we take is that we save everything to file, and only load the data to memory when the user actually accesses it.

In order for the latter to work when no output file is specified, `ahkab` stores the simulation data in a temporary file provided by your OS. When the user exits the Python interpreter (or IPython or debugger), the file is removed.

4.1.4 Run it!

Once you have a circuit and one or more simulations, it's time to run them!

The following methods are available to do so:

`run`(circ[, an_list])	Run analyses on a circuit.
`queue`(*analysis)	Queue one or more analyses to execute them subsequently with `run()`.

The `run()` function will return the results in dictionary form.

4.1.5 Extras

The core module also contains a few extra methods which were deemed important enough to be inserted here.

In particular, the `get_op_x0()` method allows the user to quickly compute an operating point to be used to specify the linearization point for a more complex analysis and `icmodified_x0()` allows the user to modify said operating point to take into account the user-specified initial conditions in the circuit description.

Lastly, `set_temperature()` can be used to quickly set the simulation temperature.

4.1.6 All methods in alphabetical order

`get_op_x0`(*circ*)
Shorthand to specify and run an OP analysis to get a linearization point.

Parameters:

circ [circuit instance] The circuit instance for which the linearization point is sought.

Returns:

x0 [an OP solution object] The linearization point.

`icmodified_x0`(*circ*, *x0*)
Modify `x0` to take into account the ICs in the circuit.

Parameters:

circ [circuit instance] The circuit instance from which the initial conditions are to be extracted.

x0 [`numpy` array] The vector to which the initial conditions are to be applied.

`main`(*filename*, *outfile=u'stdout'*, *verbose=3*)
Method to call ahkab from a Python script with a netlist file.

Parameters:

filename [string] The netlist filename.

outfile [string, optional] The outfiles base name, the suffixes shown below will be added. With the exception of the magic value `stdout` which causes ahkab to print out instead of to disk.

verbose [int, optional] the verbosity level, from 0 (silent) to 6 (debug). It defaults to 3, the same as running ahkab through its command line interface.

Filename suffixes, for each analysis:

- Alternate Current (AC): `.ac`
- Direct Current (DC): `.dc`
- Operating Point (OP): `.opinfo`
- Periodic Steady State (PSS): `.pss`
- Pole-zero Analysis (PZ): `.pz`
- TRANsient (TRAN): `.tran`
- Symbolic: `.symbolic`

Returns:

res [dict] A dictionary containing the computed results.

new_ac (*start*, *stop*, *points*, *x0=u'op'*, *sweep_type=u'LOG'*, *outfile=None*, *verbose=0*)

Assembles an AC analysis and returns the analysis object.

The analysis itself can be run with `ahkab.run(...)` or queued with `ahkab.queue(...)` and then run subsequently.

Parameters:

start [float] the start angular frequency, ω_{start}.

stop [float] the stop angular frequency, ω_{stop} (included in the sweep).

points [float] the number of points to be used the discretize the *[start, stop]* interval.

x0 [string or ndarray, optional] The linearization point for the AC analysis. If set to 'op' (default), the latest Operating point analysis will be used. Otherwise, you may supply your own linearization point in ndarray format.

sweep_type [string, optional] It can be set to either `options.ac_lin_step` (linear stepping) or `options.ac_log_step` (log10 stepping). Defaults to logarithmic stepping.

outfile [string, optional] the filename of the output file where the results will be written. '.ac' is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to `None`, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

verbose [int, optional] the verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an [dict] the analysis object (a dict)

See also:

`run()`, *`queue()`*

new_dc (*start*, *stop*, *points*, *source*, *sweep_type=u'LINEAR'*, *guess=True*, *x0=None*, *outfile=None*, *verbose=0*)

Assembles a DC sweep analysis and returns the analysis object.

The analysis itself can be run with: `ahkab.run(...)` or queued and then run subsequently.

Parameters:

start [float] the start value for the sweep.

stop [float] the stop value for the sweep (included in the sweep points).

points [int] the number of sweep points.

source [string] the `part_id` of the independent current or voltage source to be swept.

sweep_type [string, optional] can be set to either `options.dc_lin_step` (linear stepping) or `options.dc_log_step` (log10 stepping). Defaults to linear.

guess [boolean, optional] if set to `True`, the analysis will start from an initial guess, hopefully speeding up the convergence of particularly stiff circuits.

x0 [`numpy` array, optional] if the `guess` option above is not used, one can provide a starting point directly, setting `x0` to an opportunely sized `numpy` array. If both `x0` and `guess` are set, `x0` takes the precedence.

outfile [string, optional] the filename of the output file where the results will be written. '.dc' is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to `None`, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

verbose [int, optional] the verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an [dict] the analysis description

See also:

`run()`, *`queue()`*

`new_op` (*guess=None*, *x0=None*, *outfile=None*, *verbose=0*)

Assembles an OP analysis and returns the analysis object.

The analysis itself can then be run with: `ahkab.run(...)` or queued with `ahkab.queue(...)` and then run subsequently.

Parameters:

guess [boolean, optional] if set to True, the analysis will start from an initial guess, hopefully speeding up the convergence of stiff circuits.

x0 [matrix, optional] In alternative to the `guess` option above, one can provide an explicit starting point to the OP algorithm, setting x0 to an opportunely sized `numpy` array. FIXME mention help method here If both x0 and guess are set, x0 takes the precedence.

outfile [string, optional] the filename of the output file where the results will be written. `.opinfo` is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to None, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

verbose [int, optional] the verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an [dict] the analysis description

See also:

`run()`, *`queue()`*

`new_pss` (*period*, *x0=None*, *points=None*, *method=u'brute-force'*, *autonomous=False*, *outfile=None*, *verbose=0*)

Assembles a Periodic Steady State (PSS) analysis and returns the analysis object.

The analysis itself can be run with: `ahkab.run(...)` or queued with `ahkab.queue(...)` and then run subsequently.

Parameters:

period [float] the time period of the solution, in seconds. This value is required, autonomous circuits are currently unsupported.

x0 [`numpy` array, optional] the starting point solution, used at $t = 0$.

points [int, optional] the number of points to use to discretize the PSS solution. If not set, if method is 'shooting', defaults to `options.shooting_default_points`

method [string, optional] The method to be employed to attempt a PSS solution of the circuit. It can be either `ahkab.BFPSS` or `ahkab.SHOOTING`.

autonomous [bool, optional] Whether the circuit is autonomous or not. Non-autonomous circuits are currently unsupported!

mna, Tf, D [`numpy` arrays, optional] The matrices to be used to solve the circuit. They are optional, if they have already been computed, reusing them saves time.

outfile [string, optional] The filename of the output file where the results will be written. '.tran' is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to `None`, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

verbose [int, optional] The verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an [dict] the analysis object (a dict)

See also:

`run()`, *`queue()`*

`new_pz`(*input_source=None*, *output_port=None*, *shift=0.0*, *MNA=None*, *outfile=None*, *x0=u'op'*, *verbose=0*)

Assembles a Pole-Zero analysis and returns the analysis object.

The analysis itself can be run with: `ahkab.run(...)` or queued with `ahkab.queue(...)` and then run subsequently.

Parameters:

input_source [str or instance] the input source for zero calculation

output_port [tuple or single node] the output port. If it is composed of only one node, then the second node is assumed to be GND.

shift [float, optional] Perform the calculation at a shifted freq `shift`.

MNA [ndarray, optional] the numpy matrix to be used to solve the circuit. It is optional, but, if it's already been computed, reusing it will save time.

outfile [string, optional] The filename of the output file where the results will be written. '.pz' is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to `None`, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

x0 [`numpy` array or str, optional] the optional linearization point. If set to a string, it must be the result of an .OP analysis (use `'op'`) or an .IC condition defined in the netlist. It has no effect on linear circuits.

verbose [int, optional] The verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an : the analysis description object, a dict instance.

new_symbolic(*source=None*, *ac_enable=True*, *r0s=False*, *subs=None*, *outfile=None*, *verbose=0*)

Assembles a Symbolic analysis and returns the analysis object.

The analysis itself can be run with `ahkab.run(...)` or queued with `ahkab.queue(...)` and then run subsequently.

Parameters:

source [string, optional] if `source` is set, the transfer function between the current or voltage source `source` and each circuit unknown will be evaluated, with symbolic evaluation of DC gain, poles and zeros. `source` is to be set to the `part_id` of an independent current or voltage source in the circuit, eg. `'V1'` or `'Iin'`. This computation should be avoided for large circuit, as indiscriminate transfer function, gain and singularities evaluation in large circuits can result in very long run times and needs a significant amount of RAM, on top of an already resource intensive symbolic analysis. We suggest manually evaluating selected transfer functions of interest instead.

ac_enable [bool, optional] If set to `True` (default), the frequency-dependent elements will be considered, otherwise the algorithm will focus on low frequency solutions, where all capacitors are replaced with open circuits and all inductors are short circuits, usually providing a much easier circuit.

r0s [bool, optional] If set to `True`, the finite output conductances of transistors `go` (where $go = 1/r_0$) will be taken into account, otherwise they will be considered infinite (default). The finite output conductances generally introduce a significant additional complexity in large circuits, sometimes of interest to the designer, sometimes simply introducing 2nd and 3rd order effects of little-to-no interest, which would produce no significant contribution in a numerical analysis, but come at a high computation price in a symbolic analysis. A possible approach in those cases may be disabling this option and explicitly introducing additional conductances where deemed of interest.

subs [dict, optional] `subs` is a dictionary of substitutions to be performed before attempting to solve the circuit. For example, if two resistances `R1` and `R2` are to be equal, set `subs={'R2':'R1'}` and `R1` will be replaced by an instance of `R2`. This may simplify the solution (or allow finding one in reasonable time for complex circuits).

outfile [string, optional] The filename of the output file where the results will be written. '.symbolic' is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to `None`, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

verbose [int, optional] The verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an [dict] the analysis description

See also:

`run()`, *`queue()`*

new_tran(*tstart*, *tstop*, *tstep*, *x0=u'op'*, *method=u'TRAP'*, *use_step_control=True*, *outfile=None*, *verbose=0*)

Assembles a TRAN analysis and returns the analysis object.

The analysis itself can be run with `ahkab.run(...)` or queued with `ahkab.queue(...)` and then run subsequently.

Parameters:

tstart [float] the start time for the transient analysis.

tstop [float] the stop time.

tstep :float the time step. If the step control is active, this is the minimum time step value that will be allowed during simulation.

x0 [`numpy` array, optional] the optional initial conditions point, $x0 = x(t = 0)$.

method [string , optional] the differentiation method to be used. Can be set to 'IMPLICIT_EULER', 'TRAP', 'GEAR4', 'GEAR5' or 'GEAR6'. It defaults to 'TRAP'.

use_step_control [boolean, optional] Whether ste control should be enabled or not. if set to `False`, the differentiation method will use a fixed time step equal to `tstep`.

outfile [string, optional] the filename of the output file where the results will be written. '.tran' is automatically added at the end to prevent different analyses from overwriting each-other's results. If unset or set to `None`, defaults to `stdout`, if the simulator was called from the command line, otherwise, if the simulator is run from an interactive session, a temporary file will be used to store the data.

verbose [int, optional] the verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

an [dict] the analysis description

See also:

`run()`, *`queue()`*

`new_x0`(*circ*, *icdict*)

Builds an `x0` matrix from user supplied values.

Supplying a custom x0 can be useful: - To aid convergence in tough circuits. - To start a transient simulation from a particular x0

Parameters:

circ [circuit instance] The circuit

icdict [dict] a dictionary specifying the node voltages and branch currents, where appropriate, in V and A, respectively, assembled as shown in the next section. All unspecified node voltages default to `0` V and all unspecified currents default to 0.

The user-specified values are to be provided as follows:

- to specify a nodal voltage: `{'V(node)':<voltage value>}`
- to specify a branch current: `'I(<element>)':<current value>}`

Examples:

- `{'V(n1)':2.3, 'V(n2)':0.45, ...}`
- `{'I(L1)':1.03e-3, I(V4):2.3e-6, ...}`

Note: This simulator uses the normal convention, also known as the Passive sign convention.

Returns:

x0 [numpy array] The assembled x0.

process_postproc (*postproc_list*, *title*, *results*, *outfilename*)
Runs the post-processing operations, such as plotting.

Not meant for end users.

deprecated in 0.10

Parameters:

postproc_list [list,] list of post processing operations

title [string] the deck title

results: dict the results to be plotted (which may include including ones that are not needed too).

outfilename: string if the plots are saved to disk, this is the filename without extension

queue (**analysis*)
Queue one or more analyses to execute them subsequently with `run()`.

Parameters

analysis [one or more analysis descriptions.] The analyses to be queued.

Returns:

None

run (*circ*, *an_list=None*)
Run analyses on a circuit.

Parameters:

circ [circuit instance] The circuit to be simulated.

an_queue [list, optional] the list of analyses to be performed. If unset, it defaults to those queued with `queue`.

Returns:

results [dict] the results (in dict form)

See also:

`queue()`

set_temperature (*T*)
Set the simulation temperature, in Celsius.

4.2 ahkab.ac

This module contains the methods required to perform an AC analysis.

Note: Typically, the user does not need to call the functions in this module directly, instead, we recommend defining an AC analysis object through the convenience method `ahkab.ahkab.new_ac()` and then running it calling `ahkab.ahkab.run()`.

4.2.1 Overview of AC simulations

The AC simulation problem

Our AC analysis problem can be written as:

$$MNA\,x + AC(\omega)\,x + Jx + N_{ac}(\omega) = 0$$

We need:

1. the Modified Nodal Analysis matrix MNA,
2. the AC matrix, holding the frequency dependent parts,
3. J, the Jacobian matrix from the linearized non-linear elements,
4. N_{ac}, the AC sources contribution.

An Operating Point (OP) has to be computed first if there is any non-linear device in the circuit to perform the linearization.

When all the matrices are available, it is possible to solve the system for the frequency values specified by the user, providing the resulting matrix is not singular (and possibly well conditioned).

Building the AC matrix

It's easy to set up the voltage lines, since line 2 refers to node 2, etc...

A capacitor between two example nodes `n1` and `n2` introduces the following elements:

$$\begin{aligned} &(\text{KCL node n1}) \qquad +j\omega C\,V(n1) - j\omega C V(n2) + ... = ... \\ &(\text{KCL node n2}) \qquad -j\omega C\,V(n1) + j\omega C V(n2) + ... = ... \end{aligned}$$

Inductors generate, together with voltage sources, Current-Controlled Voltage sources (CCVS), Voltage-Controlled Voltage Sources (VCVS), an additional line in the MNA matrix, and hence in AC too. In fact, the current flowing through the device gets added to the unknowns vector, x.

For example, in the case of an inductors, we have:

$$(\text{KVL over n1 and n2}) \qquad V(n1) - V(n2) - j\omega L\,I(\text{inductor}) = 0$$

To understand on which line is the KVL line for an inductor, we use the *order* of the elements in `ahkab.circuit`:

- first are assembled all the voltage rows,
- then the current rows, in the same order in which the elements that introduce them are found in `ahkab.circuit.Circuit`.

Solving

For each angular frequency ω, the simulator solves the matrix equation described.

Since the equation is linear, solving is performed with a single matrix inversion and multiplication for each step.

4.2.2 Module reference

ac_analysis (*circ*, *start*, *points*, *stop*, *sweep_type=None*, *x0=None*, *mna=None*, *AC=None*, *Nac=None*, *J=None*, *outfile=u'stdout'*, *verbose=3*)

Performs an AC analysis.

Parameters:

circ [Circuit instance] The circuit to be simulated.

start [float] The start frequency for the AC analysis, in Hz.

points [float,] The number of points to be used to discretize the `[start, stop]` interval.

stop [float] The stop frequency, in Hz.

sweep_type [string, optional] Either `options.ac_log_step` (ie `'LOG'`) or `options.ac_lin_step` (ie `'LIN'`), defaults to `options.ac_log_step`, resulting in a logarithmic sweep.

x0 [OP results instance, optional] The linearization point. If not set, it will be computed running an OP analysis.

mna, AC, Nax, J [ndarrays, optional] The matrices to perform the analysis. They will be computed if not supplied.

outfile [string, optional] The name of the file where the results will be written. The suffix `'.ac'` is automatically added at the end of the string to prevent different analyses from overwriting each-other's results. Set to `'stdout'` to write to the standard output. If unset, or set to `None`, defaults to the standard output.

verbose [int, optional] The verbosity level, from 0 (silent) to 6 (debug).

Returns:

ACresult [AC solution] The AC analysis results.

Raises

- **`ValueError`** – if the parameters are out of their valid range.
- **`RuntimeError`** – if the circuit is non-linear and can't be linearized.

4.3 ahkab.bfpss

Brute-force periodic steady state analysis module

bfpss (*circ*, *period*, *step=None*, *points=None*, *autonomous=False*, *x0=None*, *mna=None*, *Tf=None*, *D=None*, *outfile=u'stdout'*, *vector_norm=<function <lambda>>*, *verbose=3*)

Performs a PSS analysis employing the 'brute-force' algorithm

The time step is constant and IE will be used as DF.

Parameters:

circ [Circuit instance] the circuit to be simulated

period [float] the period of the solution

step [float, optional] the time step between consecutive points, it will be calculated if not provided

points [int, optional] the number of points to be used to sample one period, it will be calculated if not provided

autonomous [bool, optional] Is the circuit clocked or autonomously oscillating? With the current implementation, setting `autonomous=True` will result in an exception being raised, autonomous circuits are not supported

x0 [ndarray, optional] The initial guess to be used. (Experimental, needs work.)

mna, D, Tf [ndarrays, optional] The matrices describing the circuit may be supplied to speed up the solution, if available. If not supplied, they will be automatically calculated.

vector_norm [function, optional] The norm to be employed in the convergence checks. Defaults to the Inf norm.

outfile [str, optional] the output filename. Defaults to `'stdout'`.

verbose [int, optional] Verbosity level on a scale from 0 (silent) to 6 (very verbose). The `verbose` flag is automatically set is to zero if `datafilename == 'stdout'`

Note: `step` and `points` are mutually exclusive options:

- if `step` is specified, the number of points will be automatically determined.
- if `points` is set, the step will be automatically determined.
- if none of them is set, `options.bfpss_default_points` will be used as value for `points` and `step` computed accordingly.

Returns:

sol [results.pss_solution] The simulation results

4.4 ahkab.circuit

4.4.1 Introduction

A circuit is described in the ahkab simulator by an instance of the *`Circuit`* class.

This class holds everything is needed to simulate the circuit, except the specification of the analyses to be performed.

To rewrite a netlist from a Circuit instance see the *`ahkab.printing`* module.

4.4.2 The Circuit

A circuit is derived from a list which contains all its elements.

Conceptually, every time an element is to be inserted in the circuit, two operations have to be performed:

- The element must be appended to the `Circuit` instance.
- Its connections should be ensure checking that the nodes the element refers to are indeed existing circuit nodes.

To simplify the operation of adding a component to a `Circuit`, the following convenience methods are provided to the user to add and remove most elements to the circuit:

- *Circuit.add_resistor()*
- *Circuit.add_capacitor()*
- *Circuit.add_inductor()*
- *Circuit.add_vsource()*
- *Circuit.add_isource()*
- *Circuit.add_diode()*
- *Circuit.add_mos()*
- *Circuit.add_cccs()*
- *Circuit.add_vcvs()*
- *Circuit.add_vccs()*
- *Circuit.add_user_defined()*
- *Circuit.remove_elem()*

Example:

```
mycircuit = circuit.Circuit(title="Example circuit", filename=None)
# no filename since there will be no deck associated with this circuit.
# get the ref node (gnd)
gnd = mycircuit.get_ground_node()
# add a node named n1 and a 600 ohm resistor connected between n1 and gnd
mycircuit.add_resistor(part_id="R1", n1="n1", n2=gnd, R=600)
```

Refer to the methods help for additional information.

4.4.3 Nodes

The nodes are internally stored in the following way: we assign to each node an internal ID, independetly from its external identifier used in the netlist. Those IDs are integers.

The simulator uses always the internal names. When the results are presented to the user, the internal node is not showed, the external identifier (or external node name) is printed instead.

This is done through:

```
my_circuit = Circuit()
...
[ init code ]
...
print "This is a node" + my_circuit.nodes_dict[int_node]
```

Internal only nodes

The number of internal only nodes (added automatically by the simulator) is held in `Circuit.internal_nodes`. That value shouldn't be changed by hand.

4.4.4 Device models

Non-linear elements have their operation described by specialized routines held in their module.

They are stored in `Circuit.models` (of type dict), the following methods are provided to add and remove device models to a Circuit instance.

- *`Circuit.add_model()`*
- *`Circuit.remove_model()`*

4.4.5 Reference

class `Circuit` (*title*, *filename=None*)

The circuit class.

Parameters:

title [string] The circuit title.

filename : string, optional

Deprecated since version 0.09.

If the circuit instance corresponds to a netlist file on disk, set this to the netlist filename.

`add_capacitor` (*part_id*, *n1*, *n2*, *value*, *ic=None*)

Adds a capacitor to the circuit.

The capacitor instance is added to the circuit elements and connected to the provided nodes. If the nodes are not found in the circuit, they are created and added as well.

Parameters:

part_id [string] The capacitor part_id (eg "C1"). The first letter is always C.

n1, n2 [string] The nodes to which the element is connected.

value [float] The capacitance value.

ic [float, optional] The initial condition, if any. See the simulation docs for how this affects the results.

See also:

`add_resistor()`, *`add_inductor()`*, *`add_vsource()`*, *`add_isource()`*, *`add_diode()`*, *`add_mos()`*, *`add_vcvs()`*, *`add_vccs()`*, *`add_cccs()`*, *`add_user_defined()`*, *`remove_elem()`*

`add_cccs` (*part_id*, *n1*, *n2*, *source_id*, *value*)

Adds a current-controlled current source (CCCS) to the circuit

This method takes care that its nodes are added as well.

Parameters:

part_id [string] The cccs ID (eg `'F1'`). The first letter is always `'F'`.

n1, n2 [strings] The output port nodes, where the output current is forced. Eg. "outp", "outm" or "out_a", "out_b".

source_id [string] The voltage source to be used to sense the current that drives the output. Eg. `'V1'`.

value [float] The proportionality factor between input (I_s) and output (I_o) currents. Mathematically:

$$I_o = \alpha I_s$$

See also:

ahkab.devices.FISource

add_ccvs (*part_id*, *n1*, *n2*, *source_id*, *value*)

Adds a current-controlled voltage source (CCCS) to the circuit

This method takes care that its nodes are added as well.

Parameters:

part_id [string] The cccs ID (eg `'H1'`). The first letter is always `'H'`.

n1, n2 [strings] The output port nodes, where the output current is forced. Eg. "outp", "outm" or "out_a", "out_b".

source_id [string] The voltage source to be used to sense the current that drives the output voltage. Eg. `'V1'`.

value [float] The proportionality factor between the sense current I_s flowing into the `source_id` voltage source (input) and output voltage. Mathematically:

$$Vn_1 - Vn_2 = \alpha I_s$$

See also:

ahkab.devices.EVSource, *ahkab.devices.FISource*

add_diode (*part_id*, *n1*, *n2*, *model_label*, *models=None*, *Area=None*, *T=None*, *ic=None*, *off=False*)

Adds a diode to the circuit (also takes care that the nodes are added as well).

Parameters:

part_id [string] The diode ID (eg "D1"). The first letter is always D.

n1, n2 [string] the nodes to which the element is connected. eg. `"in"` or `"out_a"`

model_label [string] The diode model identifier. The model needs to be added first, then the elements using it.

models [dict, optional] List of available model instances. If not set or `None`, the circuit models will be used (recommended).

Area [float, optional] Scaled device area (optional, defaults to 1.0)

T [float, optional] Operating temperature (no temperature dependence yet)

ic [float, optional] Initial condition (not really implemented yet)

off [bool, optional] Consider the diode to be initially off.

add_inductor(*part_id*, *n1*, *n2*, *value*, *ic=None*)

Adds an inductor to the circuit.

The inductor instance is added to the circuit elements and connected to the provided nodes. If the nodes are not found in the circuit, they are created and added as well.

Parameters:

part_id [string] The inductor part_id (eg "Lfilter"). The first letter is always L.

n1, n2 [string] The nodes to which the element is connected. Eg. `"in"` or `"out_a"`.

value [float] The inductance value.

ic [float, optional] Initial condition, see simulation types for how this affects the results.

See also:

add_resistor(), *add_capacitor()*, *add_inductor()*, *add_vsource()*, *add_isource()*, *add_diode()*, *add_mos()*, *add_vcvs()*, *add_vccs()*, *add_cccs()*, *add_user_defined()*, *remove_elem()*

add_inductor_coupling(*part_id*, *L1*, *L2*, *value*)

Add a coupling between two inductors.

Parameters:

part_id [string] The part ID for the inductor coupling device. Eg. `'K1'`, the first letter is always `'K'`.

L1 [string] The part ID of the first inductor to be coupled.

L2 [string] The part ID of the second inductor to be coupled.

value [float] The `k` value of the mutual coupling coefficient. Its value must be greater than zero and lesser or equal to``1`` or instability ensues.

add_isource(*part_id*, *n1*, *n2*, *dc_value*, *ac_value=0*, *function=None*)

Adds a current source to the circuit (also takes care that the nodes are added as well).

Parameters:

part_id [string] The current source ID (eg `"IA"` or `"I3"`). The first letter is always I.

n1, n2 [string] The nodes to which the element is connected, eg. `"in"` or `"out1"`.

dc_value [float] DC current value.

ac_value :float, optional AC current value, defaults to 0.

function [function, optional] Time function. See devices.py for built-in options.

add_model(*model_type*, *model_label*, *model_parameters*)

Add a model to the available circuit models.

Parameters:

model_type [string] the model type (eg "ekv"). Right now, the possible values are: `"mosq"`, `"ekv"`, `"diode"`, `"sw"`.

model_label [string] a unique identifier for the model being added (eg. `"nch1"`).

model_parameters: dict a dictionary holding the parameters to be supplied to the model to instantiate it.

add_mos (*part_id*, *nd*, *ng*, *ns*, *nb*, *w*, *l*, *model_label*, *models=None*, *m=1*, *n=1*)
Adds a mosfet to the circuit (also takes care that the nodes are added as well).

Parameters:

part_id [string] The mos part_id (eg "M1"). The first letter is always M.

nd [string] The drain node.

ng [string] The gate node.

ns [string] The source node.

nb [string] The bulk node.

w [float] The gate width.

l [float] The gate length.

model_label [string] The model identifier.

models [dict, optional] The circuit models.

m [int, optional] Shunt multiplier value. Defaults to 1.

n [int, optional] Series multiplier value, not always supported. Defaults to 1.

add_node (*ext_name*)
Adds the supplied node to the circuit, if needed.

When a 'normal' (not the reference) node is added, a internal name (or label) is assigned to it.

The nodes labels are stored in `Circuit.nodes_dict`, as a dictionary of pairs like `{int_node:ext_node}`.

Those internal names are integers, by definition, and are generated starting from 1, then 2, 3, 4, 5... The integer `0` is reserved for the reference node (gnd), which is required for the circuit to be non-pathological and has `ext_name=str(int_name)='0'`.

Notice that this method doesn't halt or print errors if the node is already been added previsiously. It simply returns the internal node name assigned to it.

Parameters:

ext_name [string] The unique identifier of the node.

Returns:

int_name [string] the *unique internal* ciecuit identifier of the node.

Raises TypeError if the parameter `ext_name` is not of "text" type (what that means exactly depends on which version of Python you are using.)

add_resistor (*part_id*, *n1*, *n2*, *value*)
Adds a resistor to the circuit.

The resistor instance is added to the circuit elements and connected to the provided nodes. If the nodes are not found in the circuit, they are created and added as well.

Parameters:

part_id [string] the resistor part_id (eg "R1"). The first letter is replaced by an R

n1, n2 [string] the nodes to which the resistor is connected.

value [float,] The resistance between `n1` and `n2` in Ohm.

See also:

add_resistor(), *add_capacitor()*, *add_inductor()*, *add_vsource()*, *add_isource()*, *add_diode()*, *add_mos()*, *add_vcvs()*, *add_vccs()*, *add_cccs()*, *add_user_defined()*, *remove_elem()*

add_switch(*part_id*, *n1*, *n2*, *sn1*, *sn2*, *ic*, *model_label*, *models=None*)

Adds a voltage-controlled or current-controlled switch to the circuit

This method also takes care that its nodes are added to the circuit as well, if necessary.

Notice:

- Current-controlled switches are not yet implemented. If you try to add one, you'll trigger an error. If you got a bit of time to spare, patches are welcome.
- The switches `part_id` should begin with `'S'` for voltage-controlled switches and with `'W'` for current-controlled switches.
- The actual behavior is set by the model. Make sure you supply a voltage-controlled switch model for a voltage-controlled switch and the appropriate type of model for the current-controlled switch. Mixing them up will go undetected.

Parameters:

part_id [string] the switch ID (eg `"S1"` - voltage-controlled - or `"Wa"` - current-controlled). The first letter is always `S` or `W`.

n1, n2 [string] the output port nodes, where the switch is connected. Eg. `"out1"`, `"out2"` or `"n_a"`, `"n_b"`.

sn1, sn2 [string] The input port nodes, where the input voltage is read. Eg. "inp", "inm" or "in_a", "in_b".

ic [boolean] The initial conditions for transient simulation. Not currently implemented!

model_label [string] The switch model identifier. The model needs to be added first, then the elements using it.

models [dict, optional] A dictionary assembled as (identifier:instance), containing all the available model instances. If not set or `None`, the circuit models will be used (recommended).

add_user_defined(*module*, *label*, *param_dict*)

Adds a user defined element.

In order for this to work, you should write a module that supplies the elem class.

XXX WRITE DOC

add_vccs(*part_id*, *n1*, *n2*, *sn1*, *sn2*, *value*)

Adds a voltage-controlled current source (VCCS) to the circuit

This method also takes care that its nodes are added as well.

Parameters:

part_id [string] The VCCS ID (eg `"G1"`). The first letter is always `'G'`.

n1, n2 [string] The output port nodes, where the output current is forced. Eg. "outp", "outm" or "out_a", "out_b". The passive convention is used as everywhere else in the simulator: a positive current flows into `n1` and out of `n2`.

sn1, sn2 [string] The input port nodes, where the input voltage is sensed. Eg. "inp", "inm" or "in_a", "in_b".

value [float] The proportionality factor between input and output voltages, which are related by the equality:

$$I_o = alpha * [V(inp) - V(inn)]$$

add_vcvs (*part_id*, *n1*, *n2*, *sn1*, *sn2*, *value*)

Adds a voltage-controlled voltage source (vcvs) to the circuit

This method also takes care that its nodes are added as well.

Parameters:

part_id [string] The vcvs ID (eg "E1"). The first letter is always E.

n1, n2 [string] The output port nodes, where the output voltage is forced. Eg. "outp", "outm" or "out_a", "out_b".

sn1, sn2 [string] The input port nodes, where the input voltage is read. Eg. "inp", "inm" or "in_a", "in_b".

alpha [float] The proportionality factor between input and output voltages is given by the relationship:

$$V(out_p) - V(out_n) = \alpha \cdot (V(in_p) - V(in_n))$$

add_vsource (*part_id*, *n1*, *n2*, *dc_value*, *ac_value=0*, *function=None*)

Adds a voltage source to the circuit (also takes care that the nodes are added as well).

Parameters:

part_id [string] The voltage source part_id (eg "VA"). The first letter is always V.

n1, n2 [string] The nodes to which the element is connected. Eg. `"in"` or `"out_a"`.

dc_value [float] DC voltage value

ac_value [float, optional] AC voltage value, defaults to 0.

function [function, optional] Time function. See devices.py for built-in options.

create_node (*name*)

Creates a new circuit node

If there is a node in the circuit with the same name, ValueError is raised.

Parameters:

name [string] the _unique_ identifier of the node.

Returns:

node [string] the _unique_ identifier of the node, to be used for subsequent element declarations, for example.

Raises

- **`ValueError`** – if a new node with the given id cannot be created, for example because a node with the same name already exists in the circuit. The only exception is the ground node, which has the reserved id `'0'`, and for which this method won't raise any exception.
- **`TypeError`** – if the parameter `name` is not of "text" type (what that means exactly depends on which version of Python you are using.)

`ext_node_to_int`(*ext_node*)
This function returns the integer id associated with an external node id.

Parameters:

ext_node [string] The external node id to be converted.

Returns:

int_node [int] The internal node associated.

`find_vde`(*index*)
Finds a voltage-defined element from its MNA KVL index

Parameters:

index [int] The element index in the KVL equations.

Returns:

e [circuit element (an instance of a subclass of Component)] The element corresponding to `index`.

Raises IndexError if no element corresponds to such an index.

`find_vde_index`(*elem_or_id*, *verbose=3*)
Finds a voltage-defined element MNA index.

Parameters:

elem_or_id [string or circuit element] You may pass as first element, alternatively, either the `part_id` of the element whose index is being requested (eg. 'V1') or the element itself. Notice the `part_id` includes both the id letter (eg. 'V') and the description (eg. '1').

verbose [int] The verbosity level, from 0 (silent) to 6 (debug).

Returns:

indx [int] The index.

Raises ValueError if no such element is in the circuit.

`get_elem_by_name`(*part_id*)
Get a circuit element from its `part_id` value.

If no matching element is found, the method returns `None`. This may change in the future.

Parameters:

part_id [string] The `part_id` of the element

Returns:

elem [circuit element] Depending whether a matching element was found or not.

Raises ValueError if the element is not found.

`get_ground_node``()`
Returns the reference node, AKA GND.

`get_locked_nodes``()`
Get all nodes connected to non-linear elements.

This list is meant to be passed to `dc_solve` or `mdn_solver` to be used in `get_td` to evaluate the damping coefficient in a Newton-Rhapson iteration.

Returns:

locked_nodes [list] A list of internal nodes.

`get_nodes_number``()`
Returns the number of nodes in the circuit

`has_duplicate_elem``()`
Self-check for duplicate elements.

No circuit should ever have duplicate elements (ie elements with the same `part_id`).

Returns:

chk [boolean] The result of the check.

`int_node_to_ext``(`*int_node*`)`
This function returns the string id associated with the integer internal node id `int_node`.

Parameters:

int_node [int] The internal node id to be converted.

Returns:

ext_node [string] the string id associated with `int_node`.

`is_int_node_internal_only``(`*int_node*`)`
Check whether an internal node is an "internal only node" or not.

Parameters:

int_node [int] The internal only node to be checked.

Returns:

chk [boolean] The result of the check.

Raises TypeError if the supplied node is not an `int`. Typically this happens when the method is called with an *external* name.

`is_nonlinear``()`
Check whether the circuit is non-linear or not.

Returns:

chk [boolean] The result of the check.

`new_internal_node`()

Generate implicit internal nodes.

Some devices are made of a group of other devices, connected by “internal only” nodes, which have the prefix `'INT'` and the simulator treats specially, hiding them from the user if not explicitly asked about them.

This method generates the external names for such nodes and inserts them in the circuit.

Returns:

ext_node [string] The corresponding external node id.

`remove_elem`(*elem_or_id*)

Removes an element from the circuit and takes care that no “orphan” nodes are left.

Note: Support for removing elements is experimental.

Parameters:

elem_or_id [string or circuit element] You may pass as first element, alternatively, either the `part_id` of the element to be removed or the element itself.

The method will also take care of purging from the circuit nodes that are left orphan, ie with no elements connected.

Raises ValueError if no such element is found in the circuit.

`remove_model`(*model_label*)

Remove a model from the available models.

Parameters:

model_label [string] the unique identifier corresponding to the model being removed.

Note: This method currently silently ignores models that are not defined.

exception `CircuitError`

General circuit assembly exception.

exception `ModelError`

Model not found exception.

exception `NodeNotFoundError`

Circuit Node exception.

`is_elem_voltage_defined`(*elem*)

Check if an element needs its own KCL equation

Parameters:

elem [Component] The element to be checked.

Returns:

chk [bool] `True` if `elem` is a voltage source, an inductor, a voltage-controlled voltage source or a current-controlled voltage source. `False` otherwise.

class `subckt`(*name*, *code*, *connected_nodes_list*)

This class holds the necessary information about a circuit.

An instance of this class is returned by:

`ahkab.netlist_parser.parse_sub_declaration()`

Parameters:

name [string] The subcircuit definition label.

code [string] The netlist code that can be instantiated have a circuit instance.

connected_nodes_list [list] A list of nodes that are used in the circuit and that are meant to be connected to the external circuit.

Notice that in the current implementation, the GND node (0) is *always* global.

4.5 ahkab.constants

Constants useful for building equations and expressions describing semiconductor physics

`T` = 300
Simulation temperature in Kelvin degrees.

`Tref` = 300
Reference temperature in Kelvin degrees.

`Vth`(*T=300*)
The thermal voltage: kT/q.

Parameters:

T [float, optional] The temperature at which the thermal voltage is to be evaluated. If not specified, it defaults to `constants.Tref`.

Returns:

vth [float] The thermal voltage, kT/q.

`e` = 1.60217646e-19
The electron charge e.

`k` = 1.3806503e-23
The Boltzmann constant

`si` = <ahkab.constants.silicon instance>
Silicon class instantiated.

class `silicon`
Silicon class

Access this class as `constants.si`.

Attributes

esi: permittivity of silicon.

eox: permittivity of silicon dioxide.

`Eg`(*T=300*)
Energy gap of silicon at temperature `T`

Parameters:

T [float, optional] The temperature at which the thermal voltage is to be evaluated. If not specified, it defaults to `constants.Tref`.

Returns:

Eg [float] The energy gap, expressed in electron-volt (eV).

ni (*T=300*)

Intrinsic Silicon carrier concentration at temperature `T`

Parameters:

T [float, optional] The temperature at which the thermal voltage is to be evaluated. If not specified, it defaults to `constants.Tref`.

Returns:

ni [float] The intrinsec carrier concentration.

4.6 ahkab.csvlib

The `csvlib` module contains common routines for handling Comma Separated Values (CSV) or Tab Separated Values (TSV) files.

Functions:

1. CSV write/load:

- *write_csv()*
- *load_csv()*

2. MISC utilities

- *get_headers()*
- *write_headers()*
- *get_headers_index()*

The separator can be selected setting:

```
csvlib.SEPARATOR = '\t' # default value
```

get_headers (*filename*)

Reads the signals inside a file.

The order of the signals in the list corresponds to the order of the signals in the file.

Parameters:

filename [string] the path to the file from which the header is to be read

Returns:

headers : list of strings.

get_headers_index (*headers*, *load_headers=None*, *verbose=3*)

Creates a list of integers. Each element in the list is the COLUMN index of the signal according to the supplied headers.

Parameters:

headers [list of strings,] the signal names, as returned by `get_headers()`.

load_headers [list, optional] The headers for the data to be loaded. If not provided, all indeces will be returned.

Returns:

the header indeces : a list of int.

load_csv (*filename*, *load_headers=None*, *nsamples=None*, *skip=0*, *verbose=3*)

Reads data in CVS format from filename.

Supports:

- selective signal loading,
- loading up to a certain number of samples,
- skipping to a certain line, to allow incremental reading of big files.

Parameters:

filename [string] the path to the file to be read.

load_headers [list of strings, optional] Each one being a signal to be loaded. An empty list (or None) is interpreted as "read all signals".

nsamples [int, optional] The number of samples to be read for each signal. If `None`, read all available samples.

skip [int, optional] The index of the first sample to be read. Default: 0

Returns:

data [ndarray] The data, ordered according to the order of `load_headers` (or the order on file if `load_headers` was empty),

headers [list of strings] the names of the signals read from file,

pos [int] position of the last sample read +1, referred to the sample #0 in the file.

EOF [bool] A flag set to true is all the data in the file were read.

write_csv (*filename*, *data*, *headers*, *append=False*)

Writes data in CVS format to filename.

The headers have to be ordered according to the data order.

Parameters:

filename [string] the path to the file to be written. Use 'stdout' to write to stdout

data [ndarray] The data to be written. Notice that variables are swept across *rows*, time samples are swept along *columns*. Or equivalently: `data[variable_index, sample_number]`

headers [list of strings] the signal names, ordered so that `headers[i]` corresponds to `data[i, :]`.

append [bool, optional] If False, the file (if it exists) will be rewritten, otherwise it will be appended to.

write_headers (*filename*, *headers*)

Writes headers in CVS format to filename.

Parameters:

filename [string] the path to the file to be written. Use 'stdout' to write to stdout.

headers [a list of strings] the signal names, ordered.

4.7 ahkab.dc_analysis

This module provides the functions needed to perform OP and DC simulations.

The principal are:

- *dc_analysis()* - which performs a dc sweep,
- *op_analysis()* - which does an operation point analysis or

Notice that internally, *dc_analysis()* calls *op_analysis()*, since a DC sweep is nothing but a series of OP analyses..

The actual circuit solution is done by *mdn_solver()*, that uses a modified version of the Newton Rhapson method.

4.7.1 Module reference

build_gmin_matrix(*circ*, *gmin*, *mna_size*, *verbose*)

Build a Gmin matrix

Parameters:

circ [circuit instance] The circuit for which the matrix is built.

gmin [scalar float] The value of the minimum conductance to ground to be used.

mna_size [int] The size of the MNA matrix associated with the GMIN matrix being built.

verbose [int] The verbosity level, from 0 (silent) to 6 (debug).

Returns:

Gmin [ndarray of size (mna_size, mna_size)] The Gmin matrix itself.

build_x0_from_user_supplied_ic(*circ*, *icdict*)

Builds a vector of appropriate (reduced!) size from the values supplied in `icdict`.

Supplying a custom x0 can be useful: - To aid convergence in tough circuits, - To start a transient simulation from a particular x0.

Parameters:

circ: circuit instance The circuit the x_0 is being assembled for

icdict: dict

`icdict` is a a dictionary assembled as follows:

- to specify a nodal voltage: `{'V(node)':<voltage value>}` Eg. `{'V(n1)':2.3, 'V(n2)':0.45, ...}`. All unspecified voltages default to 0.

- to specify a branch current: `'I(<element>)':<current value>}` ie. the elements names are sorrounded by `I(...)`. Eg. `{'I(L1)':1.03e-3, I(V4):2.3e-6, ...}` All unspecified currents default to 0.

Notes: this simulator uses the standard convention.

Returns:

x0 [ndarray] The x0 matrix assembled according to `icdict`.

Raises ValueError whenever a malformed `icdict` is supplied.

dc_analysis(*circ*, *start*, *stop*, *step*, *source*, *sweep_type=u'LINEAR'*, *guess=True*, *x0=None*, *outfile=u'stdout'*, *verbose=3*)

Performs a sweep of the value of V or I of a independent source from start value to stop value using the provided step.

For every circuit generated, computes the OP. This function relays on `dc_analysis.op_analysis()` to actually solve each circuit.

Parameters:

circ [Circuit instance] The circuit instance to be simulated.

start [float] Start value of the sweep source

stop [float] Stop value of the sweep source

step [float] The step size in the sweep

source [string] The part ID of the source to be swept, eg. `'V1'`.

sweep_type [string, optional] Either options.dc_lin_step (default) or options.dc_log_step

guess [boolean, optional] op_analysis will guess to start the first NR iteration for the first point, the previsious OP is used from then on. Defaults to `True`.

outfile [string, optional] Filename of the output file. If set to `'stdout'` (default), prints to screen.

verbose [int] The verbosity level, from 0 (silent) to 6 (debug).

Returns:

rstdc [results.dc_solution instance or None] A `results.dc_solution` instance is returned, if a solution was found for at least one sweep value. or `None`, if an error occurred (eg invalid start/stop/step values) or there was no solution for any sweep value.

dc_solve(*mna*, *Ndc*, *circ*, *Ntran=None*, *Gmin=None*, *x0=None*, *time=None*, *MAXIT=None*, *locked_nodes=None*, *skip_Tt=False*, *verbose=3*)

Low-level method to perform a DC solution of the circuit

Note: Typically the user calls `dc_analysis.op_analysis()` or `dc_analysis.dc_analysis()`, which in turn will setup all matrices and call this method on their behalf.

The system we want to solve is:

$$(mna + G_{min}) \cdot x + N(t) + T(x,t) = 0$$

Where:

- mna is the reduced MNA matrix with the required KVL/KCL rows
- N is composed by a DC part, N_{dc}, and a dynamic time-dependent part $N_{tran}(t)$ and a time-dependent part $T_t(t)$.
- $T(x, t)$ is both time-dependent and non-linear with respect to the circuit solution x, and it will be built at each iteration over t and x.

Parameters:

mna [ndarray] The MNA matrix described above. It can be built calling *generate_mna_and_N()*. This matrix will contain the dynamic component due to a Differetiation Formula (DF) when this method is called from a transient analysis.

Ndc [ndarray] The DC part of N. Also this vector may be built calling *generate_mna_and_N()*.

circ [Circuit instance] The circuit instance from which `mna` and `N` were built.

Ntran [ndarray, optional] The linear time-dependent and *dynamic* part of N, if available. Notice this is typically set when a DF being applied and the method is being called from a transient analysis.

Gmin [ndarray, optional] A matrix of the same size of `mna`, containing the minimum transconductances to ground. It can be built with *build_gmin_matrix()*. If not set, no Gmin matrix is used.

x0 [ndarray or results.op_solution instance, optional] The initial guess for the Newthon-Rhapson algorithm. If not specified, the all-zeros vector will be used.

time [float scalar, optional] The time at which any matrix evaluation done by this method will be performed. Do not set for DC or OP analysis, must be set for a transisent analysis. Notice that $t = 0$ is not the same as DC!

MAXIT [int, optional] The maximum number of Newton Rhapson iterations to be performed before giving up. If unset, `options.dc_max_nr_iter` is used.

locked_nodes [list of tuples, optional] The nodes that need to have a well behaved, slowly varying voltage applied. Typically they control non-linear elements. This is generated by *ahkab.circuit.Circuit.get_locked_nodes()* and it will be generated for you if left unset. However, if you are doing many simulations of the same circuit (as it happens in a transient analysis), it's a good idea to generate it only once.

skip_Tt [boolean, optional] Do not build the $T_t(t)$ vector. Defaults to `False`.

verbose [int, optional] The verbosity level. From 0 (silent) to 6 (debug). Defaults to 3.

Returns:

x [ndarray] The solution, if found.

error [ndarray] The error associated with each solution item, if it was found.

converged [boolean] A flag set to True when convergence was detected.

tot_iterations [int] Total number of NR iterations run.

generate_mna_and_N(*circ*, *verbose=3*)

Generate the full *unreduced* MNA and N matrices required for an MNA analysis

We wish to solve the linear stationary MNA problem:

$$MNA \cdot x + N = 0$$

If `nv` is the number of nodes in the circuit, `MNA` is a square matrix composed by:

- `MNA[:nv, :]`, KCLs ordered by node, from node 0 up to node nv.

In the above submatrix, we have a voltage part: `MNA[:nv, :nv]`, where each term `MNA[i, j]` is due to the (trans-)conductances in between the nodes and a current part, `MNA[:nv, nv:]`, where each term is due to a current variable introduced by elements whose current flow is not univocally defined by the voltage applied to their port(s).

- `MNA[nv:, :]` are the KVL equations introduced by the above terms.

`N` is similarly partitioned, but it is a vector of size `(nv,)`.

Parameters:

circ [circuit instance] The circuit for which the matrices are to be computed.

verbose [int, optional] The verbosity, from 0 (silent) to 6 (debug).

Returns:

MNA, N [ndarrays] The MNA matrix and constant term vector computed as per above.

get_solve_methods()

Get all the available solving methods

We have the standard solving method and two homotopies available. The homotopies are G_{min} stepping and source stepping.

Solving methods may be enabled and disabled through the options values:

- `options.use_standard_solve_method`,
- `options.use_gmin_stepping`,
- `options.use_source_stepping`.

Returns:

standard_solving, gmin_stepping, source_stepping [dict] The dictionaries contain the options and the status of the methods.

get_td(*dx*, *locked_nodes*, *n=-1*)

Calculates the damping coefficient for the Newthon method.

The damping coefficient is choosen as the lowest between:

- the damping required for the first NR iterations, a parameter which is set through the integer `options.nr_damp_first_iters`.
- If `options.nl_voltages_lock` evaluates to `True`, the biggest damping factor that keeps the change in voltage across the locked nodes pairs less than the maximum variation allowed, set by: `(options.nl_voltages_lock_factor * Vth)`
- Unity.

Parameters:

dx [ndarray] The undamped increment returned by the NR solver.

locked_nodes [list] A vector of tuples of (internal) nodes that are a port of a non-linear component.

n [int, optional] The NR iteration counter

Note: If `n` is set to `-1` (or any negative value), `td` is independent from the iteration number and `options.nr_damp_first_iters` is ignored.

Returns:

td [float] The damping coefficient.

mdn_solver (*x*, *mna*, *circ*, *T*, *MAXIT*, *nv*, *locked_nodes*, *time=None*, *print_steps=False*, *vector_norm=<function <lambda>>*, *debug=True*)

Solves a problem like F(x) = 0 using the Newton Algorithm with a variable damping.

Where:

$$F(x) = mna * x + T + T(x)$$

- mna is the Modified Network Analysis matrix of the circuit
- $T(x)$ is the contribute of nonlinear elements to KCL
- T contains the contributions of the independent sources, time
- invariant and linear

If $x(0)$ is the initial guess, every $x(n+1)$ is given by:

$$x(n+1) = x(n) + td \cdot dx$$

Where td is a damping coefficient to avoid overflow in non-linear components and excessive oscillation in the very first iteration. Afterwards $td = 1$ To calculate td, an array of locked nodes is needed.

The convergence check is done this way:

Parameters:

x [ndarray] The initial guess. If set to `None`, it will be initialized to all zeros. Specifying a initial guess may improve the convergence time of the algorithm and determine which solution (if any) is found if there are more than one.

mna [ndarray] The Modified Network Analysis matrix of the circuit, reduced, see above.

circ [circuit instance] The circuit instance.

T [ndarray,] The T vector described above.

MAXIT [int] The maximum iterations that the method may perform.

nv [int] Number of nodes in the circuit (counting the ref, 0)

locked_nodes [list of tuples] A list of ports driving non-linear elements, generated by `circ.get_locked_nodes()`

time [float or None, optional] The value of time to be passed to non_linear _and_ time variant elements.

print_steps [boolean, optional] Show a progress indicator, very verbose. Defaults to `False`.

vector_norm [function, optional] An R^N -> R^1 function returning the norm of a vector, for convergence checking. Defaults to the maximum norm, ie $f(x) = max(|x|)$,

debug [int, optional] Debug flag that will result in an array being returned containing node-by-node convergence information.

Returns:

sol [ndarray] The solution.

err [ndarray] The remaining error.

converged [boolean] A boolean that is set to `True` whenever the method exits because of a successful convergence check. `False` whenever convergence problems where found.

N [int] The number of NR iterations performed.

convergence_by_node [list] If `debug` was set to `True`, this list has the same size of the MNA matrix and contains the information regarding which nodes fail to converge in the circuit. Ie. `if convergence_by_node[j] == False`, node `j` has a convergence problem. This may significantly help debugging non-convergent circuits.

`modify_x0_for_ic` (*circ*, *x0*)

Modifies a supplied x0.

Several circut elements allow the user to set their own Initial Conditions (IC) for either voltage or current, depending on what is appropriate for the element considered.

This method, receives a preliminary `x0` value, typically computed by an OP analysis and goes through the circuit, looking for ICs and setting them in `x0`.

Notice it is possible to require ICs that are incompatible with each other – for example supplying different ICs to two parallel caps. In that case we try to accommodate the user's requirements in a non-strict best-effort kind of way: for this reason, whenever multiple ICs are specified, it is best to visually inspect `x0` to check that what you would have expected is indeed what you got.

Parameters

circ [circuit instance] The circuit in which the ICs are specified.

x0 [ndarray or results.op_solution] The initial value to be modified

Returns:

x0p [ndarray or results.op_solution] The modified `x0`. Notice that we return the same kind of object as it was supplied. Additionally, the `results.op_solution` is a *new instance*, while the `ndarray` is simply the original array modified.

`more_solve_methods_available` (*standard_solving*, *gmin_stepping*, *source_stepping*)

Are there more solving methods available?

Parameters:

standard_solving, gmin_stepping, source_stepping [dict] The dictionaries contain the options and the status of the methods.

Returns:

rsp [boolean] The answer.

`op_analysis` (*circ*, *x0=None*, *guess=True*, *outfile=None*, *verbose=3*)

Runs an Operating Point (OP) analysis

Parameters:

circ [Circuit instance] The circuit instance on which the simulation is run

x0 [op_solution instance or ndarray, optional] The initial guess to be used to start the NR `mdn_solver()`.

guess [boolean, optional] If set to `True` (default) and `x0` is `None`, it will generate a 'smart' guess to use as `x0`.

verbose [int] The verbosity level from 0 (silent) to 6 (debug).

Returns:

A `result.op_solution` instance, if successful, `None` otherwise.

`set_next_solve_method`(*standard_solving*, *gmin_stepping*, *source_stepping*, *verbose=3*)

Select the next solving method.

We have the standard solving method and two homotopies available. The homotopies are G_{min} stepping and source stepping.

They will be selected and enabled when failures occur according to the options values:

- `options.use_standard_solve_method`,
- `options.use_gmin_stepping`,
- `options.use_source_stepping`.

The methods will be used in the order above.

The inputs to this method are three dictionaries that keep track of which method is currently enabled and which ones has failed in the past.

Parameters:

standard_solving, gmin_stepping, source_stepping [dict] The dictionaries contain the options and the status of the methods, they should be the values provided by `get_solve_methods()`.

verbose [int, optional] The verbosity level, from 0 (silent) to 6 (debug).

Returns:

standard_solving, gmin_stepping, source_stepping [dict] The updated dictionaries.

4.8 ahkab.dc_guess

This module provides the `get_dc_guess()` method, used to compute a starting point to initialize a Newton-Rhapson solver.

4.8.1 Module reference

`get_dc_guess`(*circ*, *verbose=3*)

Build a DC guess from circuit inspection.

Notice that OP analysis will call this method on the users' behalf if not instructed not to do so.

A element can suggest its guess through the `elem.dc_guess` field. If the field is not set, or not available, no information on the most likely biasing voltage is assumed.

Parameters:

circc [Circuit instance] The circuit instance the guess is being computed for.

verbose [int, optional] The verbosity level (from 0 silent to 6 debug). Defaults to 3, medium verbosity.

Returns:

dcg [ndarray or None] The DC guess, in numpy array form, or `None`, if it was not possible to compute a meaningful guess.

4.9 ahkab.devices

This module contains several basic element classes.

4.9.1 Introduction

While they may be instantiated directly by the user, notice that the main `ahkab` module provides convenience functions to instantiate and connect into a circuit instance all of the following devices.

Notice that the circuit elements are not restricted to those provided here, the user is welcome to provide his own. Please see the dedicated section below.

4.9.2 Classes defined in this module

ISource(part_id, n1, n2[, dc_value, ac_value])	An ideal current source.
VSource(part_id, n1, n2, dc_value[, ac_value])	An ideal voltage source.
Resistor(part_id, n1, n2, value)	A resistor.
Capacitor(part_id, n1, n2, value[, ic])	A capacitor.
Inductor(part_id, n1, n2, value[, ic])	An inductor.
InductorCoupling(part_id, L1, L2, K, M)	Coupling between two inductors.
EVSource(part_id, n1, n2, value, sn1, sn2)	Linear voltage-controlled voltage source
GISource(part_id, n1, n2, value, sn1, sn2)	Linear voltage controlled current source
HVSource(part_id, n1, n2, value, source_id)	Linear current-controlled voltage source
FISource(part_id, n1, n2, value, source_id)	Linear current-controlled current source

4.9.3 Defining new elements and subclassing `Component`

We recommend to subclass *ahkab.devices.Component* if you intend to define a new element.

The general form of a (possibly nonlinear) element class is described in the following.

Required attributes and methods

The class must provide:

1. Element terminals:

```
elem.n1 # the anode of the element
elem.n2 # the cathode of the element
```

Note: a positive current is a current that flows into the anode and out of the cathode. This convention is used throughout the simulator.

2. `elem.get_ports()`

This method must return a tuple of pairs of nodes.

Eg.

```
((na, nb), (nc, nd), (ne, nf), ... )
```

Each pair of nodes is used to determine a voltage that has effect on the current.

For example, the source-referred model of an nmos may provide:

```
((n_gate, n_source), (n_drain, n_source))
```

The positive terminal is the first.

From that, the calling method builds a voltage vector corresponding to the ports vector:

```
voltages_vector = ( Va-Vb, Vc-Vd, Ve-Vf, ...)
```

That's passed to:

3. `elem.i(voltages_vector, time)`

It returns the current flowing into the element if the voltages specified in the voltages_vector are applied to its ports, at the time given.

4. `elem.g(voltages_vector, port_index, time)`

similarly returns the differential transconductance between the port at position `port_index` in the `ports_vector` (see point **2** above) and the element output current, when the operating point is specified by the voltages in the `voltages_vector`.

5. `elem.is_nonlinear`

A non linear element must have a `elem.is_nonlinear` field set to True.

6. `elem.is_symbolic`

This boolean flag is used to know whether the element should be treated symbolically by the ymbolic solver or not. It is meant to be toggled by the user at will.

7. Every element should have a `get_netlist_elem_line(self, nodes_dict)` allowing the element to print a netlist entry that parses to itself.

Recommended attributes and methods

1. A non linear element may have a list/tuple of the same length of its `ports_vector` in which there are the recommended guesses for DC and OP analyses.

Eg. `Vgs` is set to `Vt0` in mosfets.

This is obviously useless for linear devices.

4.9.4 Module reference

class `Capacitor` (*part_id*, *n1*, *n2*, *value*, *ic=None*)

A capacitor.

Parameters:

part_id [string] The unique identifier of this element. The first letter should be `'C'`.

n1 [int] *Internal* node to be connected to the anode.

n2 [int] *Internal* node to be connected to the cathode.

value [float] The capacitance in Farads.

ic [float] The initial condition (IC) to be used for time-based simulations, such as TRAN analyses, when requested, expressed in Volt.

`d` (*v*, *time=0*)

`g` (*v*, *time=0*)

`get_op_info` (*ports_v*)

Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the capacitor terminals.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

`i` (*v*, *time=0*)

class `Component` (*part_id=None*, *n1=None*, *n2=None*, *is_nonlinear=False*, *is_symbolic=True*, *value=None*)

Base Component class.

This component is not meant for direct use, rather all other (simple) components are a subclass of this element.

`g` (*v*)

`get_netlist_elem_line` (*nodes_dict*)

A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

`i` (*v*)

`set_char` (*i_function=None*, *g_function=None*)

class EVSource (*part_id*, *n1*, *n2*, *value*, *sn1*, *sn2*)
Linear voltage-controlled voltage source

The source port is an open circuit, the destination port is an ideal voltage source.

Mathematically, it is equivalent to the following:

$$\begin{cases} I_s = 0 \\ Vn_1 - Vn_2 = \alpha * (Vsn_1 - Vsn_2) \end{cases}$$

Where I_s is the current at the source port and the remaining symbols are explained in the Parameters section below.

Parameters:

n1 [int] *Internal* node to be connected to the anode of the output port.

n2 [int] *Internal* node to be connected to the cathode of the output port.

value [float] Proportionality constant α between the voltages.

sn1 [int] *Internal* node to be connected to the anode of the source (sensing) port.

sn2 [int] *Internal* node to be connected to the cathode of the source (sensing) port.

get_netlist_elem_line (*nodes_dict*)
A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

class FISource (*part_id*, *n1*, *n2*, *value*, *source_id*)
Linear current-controlled current source

This element implements a current source whose current value is controlled by the current flowing in a current source, which acts as the "sensing" element.

Mathematically:

$$\begin{cases} V(sn_1) - V(sn_2) = V_S \\ I_o = \alpha \cdot I_s \end{cases}$$

Where V_s is the voltage forced at the source port by the sensing element and I_o is the current at the output port. The remaining symbols are explained in the Parameters section below.

Note: This simulator uses the passive convention: a positive current flows into the element through the anode and exits through the cathode.

Parameters:

n1 [int] *Internal* node to be connected to the anode of the output port.

n2 [int] *Internal* node to be connected to the cathode of the output port.

value [float] Proportionality constant α between the sense current and the output current.

source_id [string] `part_id` of the sensing voltage source, eg. `'V1'` or `'VSENSE'`.

`get_netlist_elem_line` (*nodes_dict*)
A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

class `GISource` (*part_id*, *n1*, *n2*, *value*, *sn1*, *sn2*)
Linear voltage controlled current source

The source port is an open circuit, the output port is an ideal current source:

$$\begin{cases} I_s = 0 \\ I_o = \alpha \cdot (V(sn_1) - V(sn_2)) \end{cases}$$

Where I_s is the current at the source port and I_o is the current at the output port. The remaining symbols are explained in the Parameters section below.

Note: This simulator uses the passive convention: a positive current flows into the element through the anode and exits through the cathode.

Parameters:

n1 [int] *Internal* node to be connected to the anode of the output port.

n2 [int] *Internal* node to be connected to the cathode of the output port.

value [float] Proportionality constant α between the sense voltage and the output current, in Ampere/Volt.

sn1 [int] *Internal* node to be connected to the anode of the source (sensing) port.

sn2 [int] *Internal* node to be connected to the cathode of the source (sensing) port.

`get_netlist_elem_line` (*nodes_dict*)
A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

class `HVSource` (*part_id*, *n1*, *n2*, *value*, *source_id*)
Linear current-controlled voltage source

The source port is an existing voltage source, used to sense the current controlling the voltage source connected to the destination port.

Mathematically, it is equivalent to the following:

$$\begin{cases} V(sn_1) - V(sn_2) = V_S \\ Vn_1 - Vn_2 = \alpha * I[V_s] \end{cases}$$

Where $I[V_s]$ is the current flowing in the source port, V_s is the voltage applied between the nodes sn_1 and sn_2. The remaining symbols are explained in the Parameters section below.

Note: This simulator uses the passive convention: a positive current flows into the element through the anode and exits through the cathode.

Parameters:

n1 [int] *Internal* node to be connected to the anode of the output port.

n2 [int] *Internal* node to be connected to the cathode of the output port.

value [float] Proportionality constant α between the sense current and the output voltage, in V/A.

source_id [string] `part_id` of the current-sensing voltage source, eg. `'V1'` or `'VSENSE'`.

`get_netlist_elem_line`(*nodes_dict*)
A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

class `ISource`(*part_id*, *n1*, *n2*, *dc_value=None*, *ac_value=0*)
An ideal current source.

Defaults to a DC current source.

To implement a time-varying source:

- set `_time_function` to an appropriate instance having a `value(self, time)` method,
- set `is_timedependent` to `True`.

Parameters:

part_id [string] The unique identifier of this element. The first letter should be `'I'`.

n1 [int] *Internal* node to be connected to the anode.

n2 [int] *Internal* node to be connected to the cathode.

dc_value [float] DC voltage in Ampere.

ac_value [complex float, optional] AC current in Ampere. Defaults to no AC characteristics, ie $I(\omega) = 0 \;\; \forall \omega > 0$.

`I`(*time=None*)
Evaluate the current forced by the current source.

If `time` is not supplied, or if it is set to `None`, or if the source is only specified for DC, returns `dc_value`.

Parameters:

time [float or None, optional] The time at which the current is evaluated, if any.

Returns:

I [float] The current, in Ampere.

Note: This simulator uses passive convention: A positive currents flows in a element into the positive node and out of the negative node

---

get_netlist_elem_line (*nodes_dict*)
A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

get_op_info (*ports_v*)
Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the current source terminals.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

class Inductor (*part_id*, *n1*, *n2*, *value*, *ic=None*)
An inductor.

Parameters:

part_id [string] The unique identifier of this element. The first letter should be `'L'`.

n1 [int] *Internal* node to be connected to the anode.

n2 [int] *Internal* node to be connected to the cathode.

value [float] The inductance in Henry.

ic [float] The initial condition (IC) to be used for time-based simulations, such as TRAN analyses, when requested, expressed in Ampere.

get_op_info (*ports_v*, *current*)
Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the inductor terminals.

current [float] The current flowing in the inductor, positive currents flow in `n1` and out of `n2`.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

class `InductorCoupling`(*part_id*, *L1*, *L2*, *K*, *M*)

Coupling between two inductors.

This element is used to simulate the coupling between two inductors, such as in the case of a transformer.

Notice that turn ratio and the inductance ratio are linked by the relationship:

$$\frac{L_1}{L_2} = \left(\frac{N_1}{N_2}\right)^2$$

Parameters:

part_id [string] The unique identifier of this element. The first letter should be `'K'`.

L1 [string] The `part_id` of the first inductor to be coupled.

L2 [string] The `part_id` of the second inductor to be coupled.

K [float] The coupling coefficient between the two windings.

M [float] The mutual inductance between the windings, it is equal to $K\sqrt(L_1L2)$, where L_1 and L_2 are the values of the two inductors `L1` and `L2`.

`get_netlist_elem_line`(*nodes_dict*)

A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

`get_other_inductor`(*Lselected*)

class `Resistor`(*part_id*, *n1*, *n2*, *value*)

A resistor.

Parameters:

part_id [string] The unique identifier of this element. The first letter should be `'R'`.

n1 [int] *Internal* node to be connected to the anode.

n2 [int] *Internal* node to be connected to the cathode.

value [float] Resistance in ohms.

`g`

`get_op_info`(*ports_v*)

Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the resistor terminals.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

`i` (*v*, *time=0*)

`value`

class `VSource` (*part_id*, *n1*, *n2*, *dc_value*, *ac_value=0*)

An ideal voltage source.

Defaults to a DC voltage source.

To implement a time-varying source:

- set `_time_function` to an appropriate instance having a `value(self, time)` method,
- set `is_timedependent` to `True`.

Parameters:

part_id [string] The unique identifier of this element. The first letter should be `'V'`.

n1 [int] *Internal* node to be connected to the anode.

n2 [int] *Internal* node to be connected to the cathode.

dc_value [float] DC voltage in Volt.

ac_value [complex float, optional] AC voltage in Volt. Defaults to no AC characteristics, ie $V(\omega) = 0 \ \ \forall \omega > 0$.

`V` (*time=None*)

Evaluate the voltage applied by the voltage source.

If `time` is not supplied, or if it is set to `None`, or if the source is only specified for DC, returns `dc_value`.

Parameters:

time [float or None, optional] The time at which the voltage is evaluated, if any.

Returns:

V [float] The voltage, in Volt.

`get_netlist_elem_line` (*nodes_dict*)

A netlist line that, parsed, evaluates to the same instance

Parameters:

nodes_dict [dict] The nodes dictionary of the circuit, so that the method can convert its internal node IDs to the corresponding external ones.

Returns:

ntlst_line [string] The netlist line.

`get_op_info` (*ports_v*, *current*)

Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the source terminals.

current [float] The current flowing in the voltage source, positive currents flow in `n1` and out of `n2`.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

4.10 ahkab.diode

This module contains a diode element and its model class.

class diode (*part_id*, *n1*, *n2*, *model*, *AREA=None*, *T=None*, *ic=None*, *off=False*)

A diode element.

Parameters:

n1, n2 [string] The diode anode and cathode.

model [model instance] The diode model providing the mathematical modeling.

ic [float] The diode initial voltage condition for transient analysis (ie $V_D = V(n_1) - V(n_2)$ at $t = 0$).

off [bool] Whether the diode should be initially assumed to be off when computing an OP.

The other are the physical parameters reported in the following table:

Parameter	*Default value*	*Description*
AREA	1.0	Area multiplier
T	circuit temp	Operating temperature

g (*op_index*, *ports_v*, *port_index*, *time=0*)

get_drive_ports (*op*)

get_netlist_elem_line (*nodes_dict*)

get_op_info (*ports_v_v*)

Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the diode terminals.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

get_output_ports ()

gstamp (*ports_v*, *time=0*, *reduced=True*)

Returns the differential (trans)conductance wrt the port specified by port_index when the element has the voltages specified in ports_v across its ports, at (simulation) time.

ports_v: a list in the form: [voltage_across_port0, voltage_across_port1, ...] port_index: an integer, 0 <= port_index < len(self.get_ports()) time: the simulation time at which the evaluation is performed. Set it to None during DC analysis.

i (*op_index*, *ports_v*, *time=0*)

istamp (*ports_v*, *time=0*, *reduced=True*)
Get the current matrix

A matrix corresponding to the current flowing in the element with the voltages applied as specified in the `ports_v` vector.

Parameters:

ports_v [list] A list in the form: [voltage_across_port0, voltage_across_port1, ...]

time: float the simulation time at which the evaluation is performed. It has no effect here. Set it to None during DC analysis.

set_temperature (*T*)
Set the operating temperature IN KELVIN degrees

class **diode_model** (*name*, *IS=None*, *N=None*, *ISR=None*, *NR=None*, *RS=None*, *CJ0=None*, *M=None*, *VJ=None*, *FC=None*, *CP=None*, *TT=None*, *BV=None*, *IBV=None*, *KF=None*, *AF=None*, *FFE=None*, *TEMP=None*, *XTI=None*, *EG=None*, *TBV=None*, *TRS=None*, *TTT1=None*, *TTT2=None*, *TM1=None*, *TM2=None*)
A diode model implementing the Shockley diode equation.

Currently the capacitance modeling part is missing.

The principal parameters are:

Parameter	*Default value*	*Description*
IS	1e-14 A	Specific current
N	1.0	Emission coefficient
ISR	0.0 A	Recombination current
NR	2.0	Recombination coefficient
RS	0.0 ohm	Series resistance per unit area

please refer to a textbook description of the Shockley diode equation or to the source file `diode.py` file for the other parameters.

get_gm (**key*)

get_i (**key*)

print_model ()

set_temperature (*T*)

4.11 ahkab.ekv

Partial implementation of the EKV 3.0 MOS transistor model

The EKV model was developed by Matthias Bucher, Christophe Lallement, Christian Enz, Fabien ThÃ©odoloz, FranÃ§ois Krummenacher at the Electronics Laboratories, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland.

The Tecnical Report upon which this implementation is based is available here:

EKV Technical Report.

This module defines two classes:

- *ekv_device*
- *ekv_mos_model*

Features:

- EKV model implementation, computation of charges, potentials, reverse and forward currents, slope factor and normalization factors.
- Calculation of trans-conductances based on the charge-driven approach.
- N/P MOS symmetry
- Rudimentary temperature effects.

The Missing Features:

- Channel length modulation,
- Reverse Short Channel Effect (RSCE),
- Complex mobility degradation,
- Transcapacitances,
- Quasi-static implementation,

Patches to implement the above are welcome!

Note: The default values in the model are suitable for a generic 500nm feature-size CMOS process.

class **ekv_device** (*part_id*, *nd*, *ng*, *ns*, *nb*, *W*, *L*, *model*, *M=1*, *N=1*)

EKV device

Parameters:

part_id [string] The element identifier, eg 'M1'

nd [int] drain node

ng [int] gate node

ns [int] source node

nb [int] bulk node

L [float] element width [m]

W [float] element length [m]

M [int] multiplier (n. of shunt devices)

N [int] series mult. (n. of series devices)

model [ekv_model instance] The corresponding instance of ekv_mos_model

Selected methods: - get_output_ports() -> (nd, ns) - get_drive_ports() -> (nd, nb), (ng, nb), (ns, nb)

INIT_IFRN_GUESS = 1

g (*op_index*, *ports_v*, *port_index*, *time=0*)

Returns the differential (trans)conductance rs the port specified by port_index when the element has the voltages specified in ports_v across its ports, at (simulation) time.

ports_v: a list in the form: [voltage_across_port0, voltage_across_port1, ...] port_index: an integer, 0 <= port_index < len(self.get_ports()) time: the simulation time at which the evaluation is performed. Set it to None during DC analysis.

get_drive_ports (*op*)

Returns a tuple of tuples of ports nodes, as: (port0, port1, port2...) Where each port is in the form: port0 = (nplus, nminus)

get_netlist_elem_line (*nodes_dict*)

get_op_info (*ports_v*)

Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The voltages applied to all the driving ports, grouped by output port.

i.e.

```
[<list of voltages for the drive ports of output port 0>,
 <list of voltages for the drive ports of output port 1>,
 ...,
 <list of voltages for the drive ports of output port N>]
```

Usually, this method returns `op_keys` and the corresponding `op_info`, two lists, one holding the labels, the other the corresponding values.

In the case of MOSFETs, the values are way too many to be shown in a linear table. For this reason, we return `None` as `op_keys`, and we return for `op_info` a list which holds both labels and values in a table-like manner, spanning the vertical and horizontal dimension.

For this reason, each MOSFET has to have its OP info printed alone, not grouped as it happens with most other elements.

Returns:

op_keys [`None`] See above for why this value is always `None`.

op_info [list of floats] The OP information ready to be passed to `printing.table()` for arranging it in a pretty table to display.

get_output_ports ()

get_value_function (*identifier*)

i (*op_index*, *ports_v*, *time=0*)

Returns the current flowing in the element with the voltages applied as specified in the ports_v vector.

ports_v: [voltage_across_port0, voltage_across_port1, ...] time: the simulation time at which the evaluation is performed.

It has no effect here. Set it to None during DC analysis.

update_status_dictionary (*ports_v*)

class **`ekv_mos_model`**(*name=None*, *TYPE=u'n'*, *TNOM=None*, *COX=None*, *GAMMA=None*, *NSUB=None*, *PHI=None*, *VTO=None*, *KP=None*, *XJ=None*, *LAMBDA=None*, *TOX=None*, *VFB=None*, *U0=None*, *TCV=None*, *BEX=None*)

`get_device_temperature`()

Returns the temperature of the device - in K.

`get_dvsmall_dismall`(*ismall*, *verbose=3*)

The Newton algorithm in get_ismall(...) requires the evaluation of the first derivative of the fixed point function:

> dv/di = 1.0/(sqrt(.25+i)-.5) * .5/sqrt(.25 + i) + 1/sqrt(.25 + i)

This is provided by this module.

`get_gmd`(*device*, *xxx_todo_changeme3*, *opdict=None*, *debug=False*)

Returns the drain-bulk transconductance or d(IDS)/d(VD-VB).

`get_gmg`(*device*, *xxx_todo_changeme4*, *opdict=None*, *debug=False*)

Returns the gate-bulk transconductance or d(IDS)/d(VG-VB).

`get_gms`(*device*, *xxx_todo_changeme2*, *opdict=None*, *debug=False*)

Returns the source-bulk transconductance or d(IDS)/d(VS-VB).

`get_ids`(*device*, *xxx_todo_changeme*, *opdict=None*, *debug=False*)

Returns: IDS, the drain-to-source current (de-normalized), qs, the (scaled) charge at the source, qr, the (scaled) charge at the drain.

`get_ip_abs_err`(*device*)

Absolute error to be enforced in the calculation of the normalized currents.

`get_ismall`(*vsmall*, *ip_abs_err*, *iguess=None*, *debug=False*)

Solves the problem: given v, find i such that:

$$v = ln(q) + 2q$$

..math:: q = sqrt(.25 + i) - .5

The Newton Method is used inside.

`get_leq_virp`(*device*, *xxx_todo_changeme1*, *Vp*, *Leff*, *ifn*)

`get_voltages`(*vd*, *vg*, *vs*)

Performs the VD <-> VS swap if needed. Returns: (VD, VG, VS) after the swap CS, an integer which equals to:

> +1 if no swap was necessary, -1 if VD and VS have been swapped.

`get_vp_nv_nq`(*VG*)

Calculates and returns: VP, the pinch-off voltage, nv, the slope factor, nq, the charge linearization factor.

`get_vsmall`(*ismall*, *verbose=3*)

Returns v according to the equations: q = sqrt(.25 + i) - .5 v = ln(q) + 2q

`ismall2qsmall`(*ismall*, *verbose=0*)

i(f,r) -> q(f,r) Convert a source/drain scaled current to the corresponding normalized charge.

print_model()
All the internal parameters of the model get printed out, for visual inspection. Notice some can be set to None (ie not available) if they were not provided in the netlist or some not provided are calculated from the others.

qsmall2ismall(*qsmall*)
q(f,r) -> i(f,r) Convert a source/drain scaled charge to the corresponding normalized current.

set_device_temperature(*T*)
Change the temperature of the device.

Correspondingly, `VTO`, `KP` and `PHI` get updated.

setup_scaling(*nq*, *device*)
Calculates and stores in self.scaling the following factors: Ut, the thermal voltage, Is, the specific current, Gs, the specific transconductance, Qs, the specific charge.

class **scaling_holder**

4.12 ahkab.fourier

This module offers the functions needed to perform a Fourier analysis of the results of a simulation.

4.12.1 Module reference

fourier(*label*, *tran_results*, *fund*)
Fourier analysis of the time evolution of a variable.

In particular, the function uses the first 10 multiples of the fundamental frequency and a rectangular window.

A variable amount of time data is used, resampled with a fixed time step. The length of the data is decided as follows:

- The data should be taken from the end of the simulation, so that if there is any build-up or stabilization process, the Fourier analysis is not affected (or less affected) by it.
- At least 1 period of the fundamental should be used.
- Not more than 50% of the total simulation time should be used, if possible.
- Respecting the above, as much data as possible should be used, as it leads to more accurate results.

Parameters:

label [str or tuple of str] The identifier of a variable. Eg. `'Vn1'` or `'I(VS)'`. If `r` is your `tran_solution` object, calling `r.keys()` will give you all the possible variable names for your result set. If a tuple of two identifiers is provided, the difference of the two, in the form `label[0]-label[1]`, will be used.

tran_results [tran_solution instance] The TRAN results containing the time data for the `'label'` variable.

fund [float] The fundamental frequency, in Hertz.

Returns:

f [ndarray of floats] The frequencies correspoding to the `F` array below.

F [ndarray of complex data] The result of the Fourier transform, including DC.

THD [float] The total harmonic distortion. This value, for a meaningful case, should be in the range (0, 1).

spicefft (*label*, *tran_results*, *freq=None*, ***args*)

FFT analysis of the time evolution of a variable.

This function is a much more flexible and complete version of the *ahkab.fourier.fourier()* function.

The function uses a variable amount of time data, resampled with a fixed time step. The time interval is specified through the `start` and `stop` parameters, if they are not set, all the available data is used.

The function behaves differently whether the parameter `freq` is specified or not:

- If the fundamental frequency `freq` (f in the following) is specified, the function will perform an harmonic analysis, considering only the DC component and the harmonics of f up to the 9th (ie f, $2f$, $3f \ldots 9f$).
- If `freq` is left unspecified, a standard FFT analysis is performed, starting from $f = 0$, to a frequency $f_{max} = 1/(2T_{TOT}n_p)$, where T_{TOT} is the total length of the considered data in seconds and n_p is the number of points in the FTT, set through the `np` parameter to this function.

Parameters:

label [str, or tuple of str] The identifier of a variable. Eg. `'Vn1'` or `'I(VS)'`. If `r` is your `tran_solution` object, calling `r.keys()` will give you all the possible variable names for your result set. If a tuple of two identifiers is provided, the difference of the two, in the form `label[0]-label[1]`, will be used.

tran_results [tran_solution instance] The TRAN results containing the time data for the `'label'` variable.

freq [float, optional] The fundamental frequency, in Hertz. If it is specified, the output will be limited to the harmonics of this frequency. The THD evaluation will also be enabled.

start [float, optional] The first time instant to be considered for the transient analysis. If unspecified, it will be the beginning of the transient simulation.

from [float, optional] Alternative specification of the `start` parameter.

stop [float, optional] Last time instant to be considered for the FFT analysis. If unspecified, it will be the end time of the transient simulation.

to [float, optional] Alternative specification of the `stop` parameter.

np [integer] A power of two that specifies how many points should be used when computing the FFT. If it is set to a value that is not a power of 2, it will be rounded up to the nearest power of 2. It defaults to 1024.

window [str, optional] The windowing type. The following values are available:

- 'RECT' for a rectangular window, equivalent to no window at all.
- 'BART', for a Bartlett window.
- 'HANN', for a Hanning window.

- 'HAMM' for a Hamming window.
- 'BLACK' for a Blackman window.
- 'HARRIS' for a Blackman-Harris window.
- 'GAUSS' for a Gaussian window.
- 'KAISER' for a Kaiser-Bessel window.

The default is the rectangular window.

alpha [float, optional] The σ for a gaussian window or the $beta$ for a Kaiser window. Defaults to 3 and is ignored if a window different from Gaussian or Kaiser is selected.

fmin [float, optional] Suppress all data below this frequency, expressed in Hz. The suppressed data is neither returned nor used to compute the THD (if it is computed at all). The DC component is always preserved. Defaults to: return and use all data.

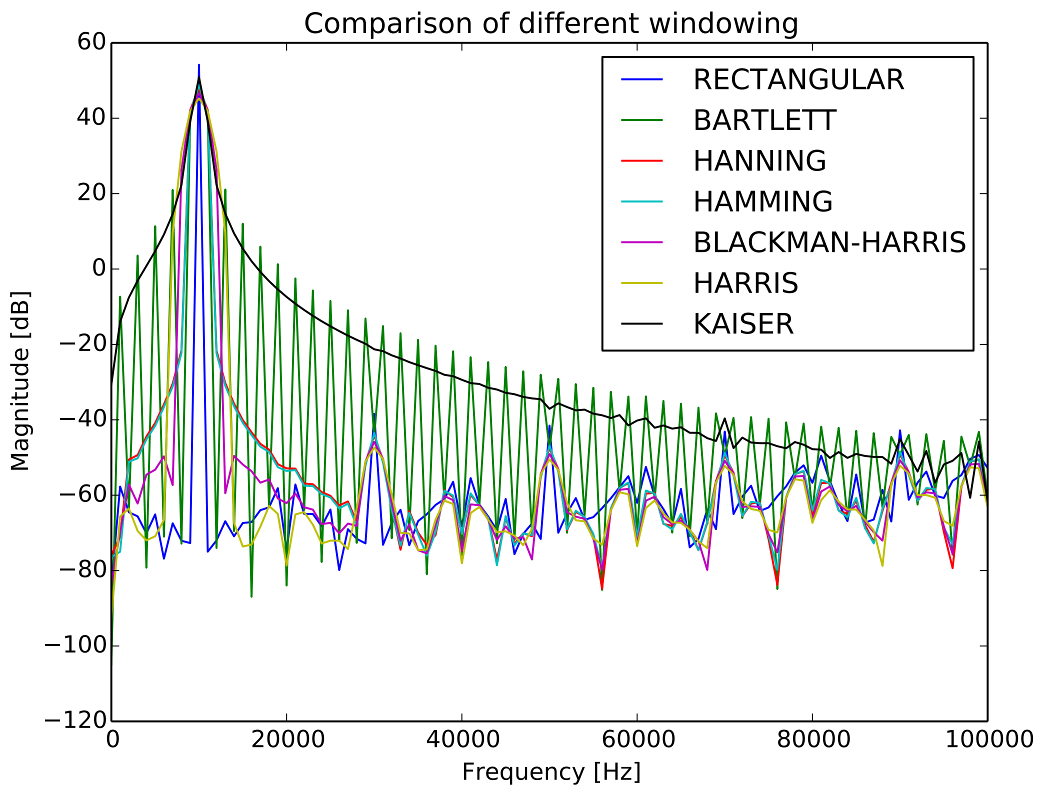
fmax [float, optional] The dual to `fmin`, discard data above `fmax` and also do not use it if computing the THD. Defaults to infinity.

Returns:

f [ndarray of floats] The frequencies, including the DC.

F [ndarray of complex data] The result of the Fourier transform, including DC.

THD [float] The total harmonic distortion, if `freq` was specified, `None` otherwise.



4.13 ahkab.gear

About the method: This is an implicit method, it means that to compute dx(n+1)/dt the value of x in (n+1) is required. We don't know it, since it's our objective). This method, as all other implicit methods, allows us to write the derivative as:

dx(n+1)/dt = x_coeff * x(n+1) + const (ii)

The get_df method returns those two vectors.

Gear's LMS interpolates the solution in a number of points equal to its order. Since it's a implicit method, one of these is x(n+1). The values x(n), x(n-1)... x(n-(order+2)) need to be supplied to the method. We can write x(t) as:

x(t) = a0 + a1*(t(n+1) - t) + a2*(t(n+1) - t)^2 + ... (i)

The equation has <order> a coefficients, which we need to determine. For this reason, we write a system of "order" equations in this way:

x(n+1) = a0 + a1*(t(n+1) - t(n+1)) + a2*(t(n+1) - t(n+1))^2 + ... x(n) = a0 + a1*(t(n+1) - t(n)) + a2*(t(n+1) - t(n))^2 + ... x(n-1) = a0 + a1*(t(n+1) - t(n-1)) + a2*(t(n+1) - t(n-1))^2 + ...

Which may be rewritten as:

z = A * a

z is the vector of known values of x A is a time dependant matrix a is a vector made of the a* coeffiecients

We don't need to explicit ALL of the a* coeffiecients. What we are really looking for is the derivative of x in t(n+1), dx(n+1)/dt in short. If we differentiate the relation (i):

dx(t)/dt = -a1 - 2*a2*(t(n+1) - t) - 3*a3*(t(n+1) - t)^2 ...

Which evaluated in t = t(n+1) gives: dx(n+1)/dt = -1 * a1

Our objective is then a1. From the previsious system we write:

a = A^-1 * z

a1 is a[1,0], which may be extracted in this way: et = [0 1 0 0 0 0 ...] (order elements) a1 = et * a = et * A^-1 * z

Because of the associative prperty of matrix multiplication, we can write:

P = et * A^-1 a1 = P[1, :] * z

But, we don't know z[0,0] = x(n+1), we can split the above relation:

a1 = P[1, 0] * x(n+1) + P[1, 1:] * z[1:, 0]

We arrived to the relation written above (ii) dx(n+1)/dt = x_coeff * x(n+1) + const = -1*a1

So: x_coeff = -1 * P[1, 0] const = -1 * P[1, 1:] * z[1:, 0]

This module uses a faster way to compute the values that doesn't require to invert the matrix. Anyway, from a theorical point of view, the above applies.

get_df (*pv_array*, *suggested_step*, *predict=False*)

The array must be built in this way: It has to be an array of arrays. Each of them has the following structure:

[time, np_matrix, np_matrix]

Hence the pv_array[k] element is made of: _ time is the time in which the solution is valid: t(n-k) _ The first np_matrix is x(n-k) _ The second is d(x(n-k))/dt Values that are not needed may be set to None and they will be disregarded.

if predict == True, it needs one more point to give a prediction of x at the suggested step.

Returns: None if the incorrect values were given, or quits. Otherwise returns an array: _ the [0] element is the np matrix of coeffiecients (Nx1) of x(n+1) _ the [1] element is the np matrix of constant terms (Nx1) of x(n+1) The derivative may be written as: d(x(n+1))/dt = ret[0]*x(n+1) + ret[1]

`get_required_values`()
This returns two python arrays built this way: [max_order_of_x, max_order_of_dx] Where: Both the values are int, or None if max_order_of_x is set to k, the df method needs all the x(n-i) values of x, where i<=k (the value the function assumed i+1 steps before the one we will ask for the derivative). The same applies to max_order_of_dx, but regards dx(n)/dt None means that NO value is required.

The first array has to be used if no prediction is required, the second are the values needed for prediction.

`has_ff`()

`is_implicit`()

4.14 ahkab.implicit_euler

This module implements the Implicit Euler (IE, aka Backward Euler, BE) and a first-order forward formula (FF) to be used for prediction.

The formula is:

$$x'_{n+1} = C_0 x_{n+1} + C_1 x_n$$

Where:

- $C_0 = 1/h$
- $C_1 = -1/h$

The backward Euler method is not only A-stable, making it suitable for the solution of stiff equations but is even L-stable.

4.14.1 Module reference

`get_df`(*pv_array*, *suggested_step*, *predict=True*)
Get the coefficients for the DF, FF and LTE calculation

Parameters:

pv_array [list] It must be an list of lists, each of them having the structure `[time, xnk, dxnk]`.

In particular, the `pv_array[k]` element of `pv_array` is composed of:

- `time`, float, which is the time at which the solution is valid: `t(n-k)`,

- `xnk`, ndarray, which is x_{n-k},
- `dxnk`, ndarray, dx_{n-k}/dt.

The length of `pv_array` has to match the value returned by *get_required_values()*.

Any values that are not needed may be set to `None`, and they will be disregarded.

suggested_step [float] The step that is (expected) to be used in the DF. It is only an expectation because it may be rejected at a later stage if there is step control enabled.

predict [boolean, optional] Whether the terms for a prediction formula are required as well or not. Defaults to `True`.

Returns:

ret [tuple] `ret` is a tuple of 5 elements, where:

- the [0] element is the coeffiecient of x_{n+1} (scalar),
- the [1] element is the matrix of constant terms of shape `(Nx1)` of x_{n+1},
- the [2] element is the coefficient of the LTE of x_{n+1} (scalar),
- the [3] element is the predicted value of x_{n+1} (matrix), only available if the `predict` parameter is set to `True`. Otherwise it's `None`.
- the [4] element is the coefficient of the LTE of the prediction (matrix), also only available if the `predict` parameter is set to `True`, otherwise, it is `None`.

Note: With the returned values, the derivative may then be written as:

$$\frac{dx_{n+1}}{dt} = \mathrm{ret}[0]\ x_{n+1} + \mathrm{ret}[1]$$

get_df_coeff(*step*)

Get the coefficients for a Backward Euler differentiation step

The first coefficient is the factor for the new point x_{n+1}, the second is the one for the previous point x_n.

If the step value is h, this method returns:

$$[1/h, -1/h]$$

Parameters:

step [float] The differentiation formula step value.

Returns:

c0, c1 [floats] The coefficients of x_{n+1} and x_n.

get_required_values()

Get what values are required by the DF and the FF

Returns

The method returns two tuples, each of them having the form:

```
[max_order_of_x, max_order_of_dx]
```

The first tuple is the one to be considered if no Forward Formula (FF) is needed, the second if the FF is also required.

Both the values in each tuple can be either of type `int` or be set to `None`.

If `max_order_of_x` is set to an arbitrary positive integer value k, the Differentiation Formula (DF) needs all the x_{n-i} values of x, where $i \leq k$ (the value x has $i+1$ steps before the one we will ask for the derivative). The same applies to `max_order_of_dx`, but it regards dx/dt instead of x.

If `max_order_of_x` or `max_order_of_dx` are set to `None`, that means that no value of x, or dx/dt, is required.

In the case at hand, where the formula is the Backward Euler (BE, aka Implicit Euler, IE), this method will return:

`((0, None), (1, None))`

`has_ff``()`
Is a forward formula for prediction available?

Returns:

`True`

`is_implicit``()`
Is this differentiation formula implicit?

`order` = 1
The order of the differentiation formula

4.15 ahkab.mosq

4.15.1 The Square Law Mos Model

This module defines two classes:

- *`mosq_device`*, the device
- `mosq_model`, the model

4.15.2 Implementation details

Assuming $V_{ds} > 0$ and a transistor type N in the following, we have the following regions implemented:

1. **No subthreshold conduction.**
    - $V_{gs} < V_T$
    - $I_D = 0$
2. **Ohmic region**
    - $V_{GS} > V_T$ and $V_{GD} > V_T$
    - $I_D = k_n W/L((V_{GS} - V_T)V_{DS} - V_{DS}^2/2)$
3. **Saturation region**

- $V_{GS} > V_T$ and $V_{DS} > V_{GS} - V_T$
- $V_{GS} < V_T$
- $I_D = 1/2k_n W/L(V_{GS} - V_T)^2 * [1 + \lambda * (V_{DS} - V_{GS} + V_T)]$

4.15.3 Module reference

class **mosq_device** (*part_id*, *nd*, *ng*, *ns*, *nb*, *W*, *L*, *model*, *M=1*, *N=1*)

Quadratic Law MOSFET device

Parameters:

part_id [string] The part ID of the model. Eg. `'M1'` or `'Mlow'`, the first letter should always be `'M'`.

nd [int] drain node

ng [int] gate node

ns [int] source node

nb [int] bulk node

L [float] element width [m]

W [float] element length [m]

model [mosq_mos_model instance] the model for the device

M [int, optional] shunt multiplier (n. of shunt devices)

N [int, optional] series multiplier (n. of series devices)

get_drive_ports (*op*)

Get the circuit ports that drive the device.

Returns:

tp : a tuple of tuples of nodes, each node being a drive port of the device.

Eg. `tp` might be defined as:

```
tp = (port0, port1, port2...)
```

Where each port in the tuple is of the form:

```
port0 = (nplus, nminus)
```

In the case of a MOSQ device, the method returns:

```
((nd, nb), (ng, nb), (ns, nb))
```

Where:

- `nd` is the internal identifier of the drain node,
- `ng` is the internal identifier of the gate node,
- `ns` is the internal identifier of the source node.
- `nb` is the internal identifier of the bulk node,

get_mc_requirements ()

get_netlist_elem_line(*nodes_dict*)
Get the element netlist entry

get_op_info(*ports_v*)
Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The voltages applied to all the driving ports, grouped by output port.

i.e.

```
[<list of voltages for the drive ports of output port 0>,
 <list of voltages for the drive ports of output port 1>,
 ...,
 <list of voltages for the drive ports of output port N>]
```

Usually, this method returns `op_keys` and the corresponding `op_info`, two lists, one holding the labels, the other the corresponding values.

In the case of MOSFETs, the values are way too many to be shown in a linear table. For this reason, we return `None` as `op_keys`, and we return for `op_info` a list which holds both labels and values in a table-like manner, spanning the vertical and horizontal dimension.

For this reason, each MOSFET has to have its OP info printed alone, not grouped as it happens with most other elements.

Returns:

op_keys [`None`] See above for why this value is always `None`.

op_info [list of floats] The OP information ready to be passed to `printing.table()` for arranging it in a pretty table to display.

get_output_ports()
Get the circuit ports where the device injects current.

Returns:

ports : a tuple of tuples of nodes, such as as:

```
(port0, port1, port2...)
```

Where each port in the tuple is itself a tuple, made of two nodes, eg.

```
port0 = (nplus, nminus)
```

In the case of a MOS device, the method returns:

```
((nd, ns),)
```

Where:

- `nd` is the internal identifier of the drain node,
- `ns` is the internal identifier of the source node.

get_value_function(*identifier*)

gstamp(*ports_v*, *time=0*, *reduced=True*)
Get the transconductance stamp matrix

Parameters:

ports_v [sequence] a sequence of the form: `[voltage_across_port0, voltage_across_port1, ...]`

time [float, optional] the simulation time at which the evaluation is performed. Set it to `None` during DC analysis. Defaults to 0.

reduced [bool, optional] Whether the returned matrix should be in reduced form or not. Defaults to `True`, corresponding to reduced form.

Returns:

indices [sequence of sequences] The indices corresponding to the stamp matrix.

stamp [ndarray] The stamp matrix.

istamp (*ports_v*, *time=0*, *reduced=True*)
Get the current stamp matrix

A stamp matrix corresponding to the current flowing in the element with the voltages applied as specified in the `ports_v` vector.

Parameters:

ports_v [list] A list in the form: `[voltage_across_port0, voltage_across_port1, ...]`

time: float the simulation time at which the evaluation is performed. It has no effect here. Set it to `None` during DC analysis.

setup_mc (*status*, *mckey*)

update_status_dictionary (*ports_v*)
Update the status dictionary

The status dictionary may be accessed at `elem.opdict` and contains several pieces of information that may be of interest regarding the biasing of the MOS device.

class mosq_mos_model (*name=None*, *TYPE=u'n'*, *TNOM=None*, *COX=None*, *GAMMA=None*, *NSUB=None*, *PHI=None*, *VTO=None*, *KP=None*, *LAMBDA=None*, *AKP=None*, *AVT=None*, *TOX=None*, *VFB=None*, *U0=None*, *TCV=None*, *BEX=None*)

device_check (*adev*)
Performs sanity check on the device parameters.

get_VT (*voltages*, *device*)
Get the threshold voltage

get_device_temperature ()
Returns the temperature of the device - in K.

get_gm (**key*)
Get the gate-source transconductance

Mathematically:

$$g_{ms} = \frac{dI_{DS}}{d(VG - VS)}$$

Often this is referred to as just g_m.

Parameters:

device [object] The device object holding the device parameters as attributes.

voltages [tuple] A tuple containing the voltages applied to the driving ports. In this case, the tuple is `(vds, vgs, vbs)`.

Returns:

gmb [float] The gate-source transconductace.

get_gmb(**key*)

Get the bulk-source transconductance

Mathematically:

$$g_{mb} = \frac{dI_{DS}}{d(VS - VB)}$$

Parameters:

device [object] The device object holding the device parameters as attributes.

voltages [tuple] A tuple containing the voltages applied to the driving ports. In this case, the tuple is `(vds, vgs, vbs)`.

Returns:

gmb [float] The source-bulk transconductace.

get_gmd(**key*)

Get the drain-source transconductance

Mathematically:

$$g_{md} = \frac{dI_{DS}}{d(VD - VS)}$$

Parameters:

device [object] The device object holding the device parameters as attributes.

voltages [tuple] A tuple containing the voltages applied to the driving ports. In this case, the tuple is `(vds, vgs, vbs)`.

Returns:

gmb [float] The drain-source transconductace.

get_ids(**key*)

Get the drain-source current

Parameters:

device [object] The device object holding the device parameters as attributes.

voltages [tuple] A tuple containing the voltages applied to the driving ports. In this case, the tuple is `(vds, vgs, vbs)`.

Returns:

ids [float] The drain-source current

get_svt_skp(*device*, *debug=False*)

get_voltages (*vds*, *vgs*, *vbs*)
Performs the D <-> S swap if needed.

Returns:

voltages [tuple] A tuple containing `(VDS, VGS, VBS)` after the swap

CS [int] `CS` is an integer which equals to: * +1 if no swap was necessary, * -1 if VD and VS have been swapped.

print_model ()
Print out the model

All the internal parameters of the model get printed out, for visual inspection. Notice some can be set to `None` (ie not available) if they were not provided and some of those not provided are calculated from the others.

set_device_temperature (*T*)
Change the temperature of the device.

Correspondingly, `VTO`, `KP` and `PHI` get updated.

4.16 ahkab.netlist_parser

Parse spice-like netlist files and generate circuits instances.

The syntax is explained in Netlist Syntax and it's based on [1] whenever possible.

4.16.1 Introduction

This module has one main circuit that is expected to be useful to the end user: `parse_circuit()`, which encapsulates parsing a netlist file and returns the circuit, the simulation objects and the post-processing directives (such as plotting instructions).

Additionally, the module provides utility functions related to parsing, among which the end user may be interested in the `convert()` function, which allows converting from SPICE-like representations of floats, booleans and strings to their Python representations.

The last type of functions in the module are utility functions to go through the netlist files and remove comments.

Except for the aforementioned functions, the rest seem to be more suitable for developers than end users.

4.16.2 Overview

Function for parsing

`parse_circuit`(filename[, ...])	Parse a SPICE-like netlist
`main_netlist_parser`(circ, netlist_lines, ...)	
`parse_elem_resistor`(line, circ)	Parses a resistor from the line supplied, adds its nodes t
`parse_elem_capacitor`(line, circ)	Parses a capacitor from the line supplied, adds its nodes

[1] http://newton.ex.ac.uk/teaching/CDHW/Electronics2/userguide/

Table 4.4 – continued from

parse_elem_inductor(line, circ)	Parses a inductor from the line supplied, adds its nodes
parse_elem_inductor_coupling(line, circ[, ...])	Parses a inductor coupling from the line supplied, return
parse_elem_vsource(line, circ)	Parses a voltage source from the line supplied, adds its
parse_elem_isource(line, circ)	Parses a current source from the line supplied, adds its
parse_elem_diode(line, circ[, models])	Parses a diode from the line supplied, adds its nodes to
parse_elem_mos(line, circ, models)	Parses a MOS transistor from the line supplied, adds its
parse_elem_vcvs(line, circ)	Parses a voltage-controlled voltage source (VCVS) from
parse_elem_vccs(line, circ)	Parses a voltage-controlled current source (VCCS) from
parse_elem_cccs(line, circ)	Parses a current-controlled current source (CCCS) from
parse_elem_ccvs(line, circ)	Parses a current-controlled voltage source (CCVS) from
parse_elem_switch(line, circ[, models])	Parses a switch device from the line supplied, adds its n
parse_elem_user_defined(line, circ)	Parses a user defined element.
parse_models(models_lines)	
parse_time_function(ftype, line_elements, stype)	Parses a time function of type ftype from the line_eleme
parse_postproc(circ, postproc_direc)	
parse_ics(directives)	
parse_analysis(circ, directives)	Parses the analyses.
parse_single_analysis(line)	Parses an analysis
parse_temp_directive(line)	Parses a TEMP directive:
parse_param_value_from_string(astr[, rtype, ...])	Search the string for a `<param>=<value>` couple an
parse_ic_directive(line)	Parses an ic directive and assembles a dictionary accord
parse_sub_declaration(subckt_lines)	Returns a circuit.subckt instance that holds the subckt in
parse_sub_instance(line, circ, subckts_dict)	Parses a subckt call/instance.
parse_include_directive(line, netlist_wd)	.include <filename> [*comments]

Utility functions for conversions

convert(astr, rtype[, raise_exception])	Convert a string to a different representation
convert_units(string_value)	Converts a value conforming to SPICE's syntax to `float`.
convert_boolean(value)	Converts the following strings to a boolean:

Utility functions for file/txt handling

join_lines(fp, line)	Read the lines coming up in the file.
is_valid_value_param_string(astr)	Has the string a form like `<param_name>=<value>`?
get_next_file_and_close_current(file_list, ...)	

4.16.3 Module reference

exception `NetlistParseError`

Netlist parsing exception.

`convert` (*astr*, *rtype*, *raise_exception=False*)

Convert a string to a different representation

Parameters:

astr [str] The string to be converted.

rtype [type] One among `float`, if a `float` sould be parsed from `astr`, `bool`, for parsing a boolean or `str` to get back a string (no parsing).

raise_exception [boolean, optional] Set this flag to `True` if you wish for this function to raise `ValueError` if parsing fails.

Returns:

ret [object] The parsed data.

`convert_boolean`(*value*)

Converts the following strings to a boolean: yes, 1, true to True no, false, 0 to False

raises NetlistParserException

Returns: boolean

`convert_units`(*string_value*)

Converts a value conforming to SPICE's syntax to `float`.

Quote from the SPICE3 manual:

> A number field may be an integer field (eg 12, -44), a floating point field (3.14159), either an integer or a floating point number followed by an integer exponent (1e-14, 2.65e3), or either an integer or a floating point number followed by one of the following scale factors:
>
> T = 1e12, G = 1e9, Meg = 1e6, K = 1e3, mil = 25.4x1e-6, m = 1e-3, u = 1e-6, n = 1e-9, p = 1e-12, f = 1e-15
>
> **Raises ValueError** if the supplied string can't be interpreted according

to the above.

Returns:

num [float] A float representation of `string_value`.

`get_next_file_and_close_current`(*file_list*, *file_index*)

`is_valid_value_param_string`(*astr*)

Has the string a form like `<param_name>=<value>`?

Note: No spaces.

Returns:

ans [a boolean] The answer to the above question.

`join_lines`(*fp*, *line*)

Read the lines coming up in the file. Each line that starts with '+' is added to the previous line (line continuation rule). When a line not starting with '+' is found, the file is rolled back and the line is returned.

`main_netlist_parser`(*circ*, *netlist_lines*, *subckts_dict*, *models*)

`parse_analysis`(*circ*, *directives*)

Parses the analyses.

Parameters:

circ: circuit class instance The circuit description

directives: list of tuples The list should be assembled as `(line, line_number)`.

Both of them are returned by `parse_circuit()`

Returns:

a list of the analyses

`parse_circuit` (*filename*, *read_netlist_from_stdin=False*)

Parse a SPICE-like netlist

Directives are collected in lists and returned too, except for subcircuits, those are added to circuit.subckts_dict.

Returns:

(circuit_instance, analyses, plotting directives)

`parse_elem_capacitor` (*line*, *circ*)

Parses a capacitor from the line supplied, adds its nodes to the circuit instance circ and returns a list holding the capacitor element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit to which the capacitor is to be connected.

Returns:

elements_list [list] A list containing a *`ahkab.devices.Capacitor`* element.

`parse_elem_cccs` (*line*, *circ*)

Parses a current-controlled current source (CCCS) from the line supplied, adds its nodes to the circuit instance and returns a list holding the CCCS element.

Syntax:

```
FX N+ N- VNAME VALUE
```

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the CCCS is to be inserted.

Returns:

elements_list [list] A list containing a *`ahkab.devices.FISource`* element.

`parse_elem_ccvs` (*line*, *circ*)

Parses a current-controlled voltage source (CCVS) from the line supplied, adds its nodes to the circuit instance and returns a list holding the CCVS element.

CCVS syntax:

```
HXXX N1 N2 VNAME VALUE
```

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the CCVS is to be inserted.

Returns:

elements_list [list] A list containing a `ahkab.devices.HVSource` element.

parse_elem_diode(*line*, *circ*, *models=None*)

Parses a diode from the line supplied, adds its nodes to the circuit instance and returns a list holding the diode element.

Diode syntax:

```
DX N+ N- <MODEL_LABEL> <AREA=xxx>
```

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the diode will be inserted.

Returns:

elements_list [list] A list containing a `ahkab.diode.Diode` element.

parse_elem_inductor(*line*, *circ*)

Parses a inductor from the line supplied, adds its nodes to the circuit instance circ and returns a list holding the inductor element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit to which the inductor is to be connected.

Returns:

elements_list [list] A list containing a `ahkab.devices.Inductor` element.

parse_elem_inductor_coupling(*line*, *circ*, *elements=[]*)

Parses a inductor coupling from the line supplied, returns a list holding the inductor coupling element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit to which the inductor coupling is to be connected.

Returns:

elements_list [list] A list containing a `ahkab.devices.InductorCoupling` element.

parse_elem_isource(*line*, *circ*)

Parses a current source from the line supplied, adds its nodes to the circuit instance and returns a list holding the current source element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the current source is to be inserted.

Returns:

elements_list [list] A list containing a `ahkab.devices.ISource` element.

`parse_elem_mos` (*line*, *circ*, *models*)

Parses a MOS transistor from the line supplied, adds its nodes to the circuit instance and returns a list holding the element.

MOS syntax:

:: MX ND NG NS KP=xxx Vt=xxx W=xxx L=xxx type=n/p <LAMBDA=xxx>

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit to which the element will be added.

Returns:

elements_list [list] A list containing a MOS element.

`parse_elem_resistor` (*line*, *circ*)

Parses a resistor from the line supplied, adds its nodes to the circuit instance circ and returns a list holding the resistor element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit instance to which the resistor is to be connected.

Returns:

elements_list [list] A list containing a `ahkab.devices.Resistor` element.

`parse_elem_switch` (*line*, *circ*, *models=None*)

Parses a switch device from the line supplied, adds its nodes to the circuit instance and returns a list holding the switch element.

General syntax:

```
SW1 n1 n2 ns1 ns2 model_label
```

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the switch is to be connected.

models [dict, optional] The currently defined models.

Returns:

elements_list [list] A list containing a `ahkab.switch.switch_device` element.

`parse_elem_user_defined` (*line*, *circ*)

Parses a user defined element.

In order for this to work, you should write a module that supplies the elem class.

Syntax: Y<X> <n1> <n2> module=<module_name> type=<type> [<param1>=<value1> ...]

This method will attempt to load the module <module_name> and it will then look for a class named <type>.

An object will be instatiated with the following arguments: n1, n2, param_dict, get_int_id_func, convert_units_func Where: n1: is the anode of the element n2: is the cathode param_dict: is a dictionary, its elements are {param1:value1, ...} get_int_id_func, convert_units_func are two function that may be used in the __init__ method, if needed. get_int_id_func: a function that gives back the internal name of a node convert_units_func: utility function to convert eg 1p -> 1e-12

See ideal_oscillators.py for a reference implementation. **Parameters:**

line [string] The netlist line.

circ [circuit instance.] The circuit to which the element will be added.

Returns:

elements_list [list] A list containing a `ahkab.devices.HVSource` element.

Parameters: line: the line circ: the circuit instance.

Returns: [userdef_elem]

`parse_elem_vccs`(*line*, *circ*)

Parses a voltage-controlled current source (VCCS) from the line supplied, adds its nodes to the circuit instance and returns a list holding the VCCS element.

Syntax:

```
GX N+ N- NC+ NC- VALUE
```

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the VCCS is to be inserted.

Returns:

elements_list [list] A list containing a `ahkab.devices.GISource` element.

`parse_elem_vcvs`(*line*, *circ*)

Parses a voltage-controlled voltage source (VCVS) from the line supplied, adds its nodes to the circuit instance circ and returns a list holding the VCVS element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the VCVS is to be inserted.

Returns:

elements_list [list] A list containing a `ahkab.devices.EVSource` element.

`parse_elem_vsource`(*line*, *circ*)

Parses a voltage source from the line supplied, adds its nodes to the circuit instance and returns a list holding the element.

Parameters:

line [string] The netlist line.

circ [circuit instance] The circuit in which the voltage source is to be inserted.

Returns:

elements_list [list] A list containing a `ahkab.devices.VSource` element.

`parse_ic_directive`(*line*)
  Parses an ic directive and assembles a dictionary accordingly.

`parse_ics`(*directives*)

`parse_include_directive`(*line*, *netlist_wd*)
  .include <filename> [*comments]

`parse_models`(*models_lines*)

`parse_param_value_from_string`(*astr*, *rtype=<type 'float'>*, *raise_exception=False*)
  Search the string for a `<param>=<value>` couple and returns a list.

  Parameters:

  astr [str] The string to be converted.

  rtype [type] One among `float`, if a `float` sould be parsed from `astr`, `bool`, for parsing a boolean or `str` to get back a string (no parsing).

  raise_exception [boolean, optional] Set this flag to `True` if you wish for this function to raise `ValueError` if parsing fails.

  Returns:

  ret [object] The parsed data. If the conversion fails and `raise_exception` is not set, a `string` is returned.

  - If `rtype` is `float` (the type), its default value, the method will attempt converting `astr` to a float. If the conversion fails, a string is returned.
  - If set `rtype` to `str` (again, the type), a string will always be returned, as if the conversion failed.

  This prevents `'0'` (str) being detected as `float` and converted to 0.0, ending up being a new node instead of the reference.

  Notice that in `<param>=<value>` there is no space before or after the equal sign.

  Returns:

  alist `[[param, value]]` where `param` is a string and `value` is parsed as described.

`parse_postproc`(*circ*, *postproc_direc*)

`parse_single_analysis`(*line*)
  Parses an analysis

  Parameters:

  line [str] The netlist line from which an analysis statement is to be parsed.

  Returns:

  an [dict] A dictionary with its parameters as keys.

  Raises NetlistParseError if the analysis is not parsed correctly.

`parse_sub_declaration`(*subckt_lines*)
  Returns a circuit.subckt instance that holds the subckt information, ready to be instantiated/called.

parse_sub_instance(*line*, *circ*, *subckts_dict*, *models=None*)
Parses a subckt call/instance.

1.Gets name and nodes connections

2.Looks in subckts_dict for a matching subckts_dict[name]

3.Builds a circuit wrapper

4.Calls main_netlist_parser() on the subcircuit code (with the wrapped circuit)

Returns: a elements list

parse_temp_directive(*line*)
Parses a TEMP directive:

The syntax is:

```
    .TEMP <VALUE>
```

parse_time_function(*ftype*, *line_elements*, *stype*)
Parses a time function of type ftype from the line_elements supplied.

Parameters:

ftype [str] One among `"pulse"`, `"exp"`, `"sin"`, `"sffm"` or `"am"`.

line_elements [list of strings] The tokens describing the time function. The list mustn't hold the `"type=<ftype>"` element

stype [str] Set this to "current" for current sources, "voltage" for voltage sources

See *ahkab.time_functions.pulse*, *ahkab.time_functions.sin*, *ahkab.time_functions.exp*, *ahkab.time_functions.sffm* and *ahkab.time_functions.am* for more.

Returns:

time_function [object] A time-function instance

4.17 ahkab.options

This module contains options and configuration switches the user may tune to meet his needs.

The default values are sensible options for the general case.

ac_max_nr_iter = 20
Maximum number of NR iterations for AC analyses.

ac_phase_in_deg = False
Use degrees instead of rads in AC phase results.

bfpss_default_points = 100
Default number of points for a BFPSS analysis.

bfpss_max_nr_iter = 10000
Maximum number of NR iterations for BFPSS analyses.

cache_len = 67108864
Cache size to be used in *ahkab.utilities.memoize()*, defaults to 512MB

`cli` = False
A boolean to differentiate command line execution from module import When cli is False, no printing and no weird stdout stuff.

`cmin` = 1e-18
Minimum capacitance to ground.

`dc_max_guess_effort` = 250000
Do not perform an init DC guess if its effort is higher than this value.

`dc_max_nr_iter` = 10000
Maximum allowed NR iterations during a DC analysis.

`dc_sweep_skip_allowed` = True
Can we skip troublesome points during DC sweeps?

`dc_use_guess` = True
Enable guessing to init the NR solver during a DC analysis.

`default_tran_method` = u'TRAP'
The default differentiation method for transient analyses.

`dense_matrix_limit` = 400
Dense matrix limit: if the dimensions of the square MNA matrix are bigger, use sparse matrices.

`encoding` = u'utf8'
Encoding of the netlist files.

`gmin` = 1e-12
Minimum conductance to ground.

`hmin` = 1e-20
Minimum allowed discretization step for time.

`iea` = 1e-09
Current absolute tolerance.

`ier` = 0.001
Current relative tolerance.

`nl_voltages_lock` = True
In all NR iterations, lock the nodes controlling non-linear elements. See also *`ahkab.dc_analysis.get_td()`*.

`nl_voltages_lock_factor` = 4
Non-linear nodes lock factor: if we allow the voltage on controlling ports to change too much, we may have current/voltage overflows. Think about the diode characteristic. So we allow them to change of `nl_voltages_lock_factor` $\cdot V_{th}$ at most and damp all variables accordingly.

`nr_damp_first_iters` = False
Should we damp artificially the first NR iterations? See also *`ahkab.dc_analysis.get_td()`*.

`plotting_display_figsize` = (12.94, 8)
Default size for plots showed to the user, in inches.

`plotting_lw` = 1.25
Plotting line width.

`plotting_outtype` = u'png'
Format to be used when writing plots to disk.

`plotting_save_figsize` = (20, 10)
Default size for plots saved to disk.

`plotting_show_plots` = False
Should plots be shown to the user? This variable is set to `True` automatically if a screen is detected in Unix systems.

Notice that by default ahkab both shows plots *and* saves them to disk.

`plotting_style` = u'-o'
Matplotlib line plot style: see matplotlib's doc.

`plotting_wait_after_plot` = True
Wait for the user to close the plot? If set to `False`, plots are created and immediately destroyed.

`print_int_nodes` = True
Should we show to the user results pertaining to nodes introduced by components or by the simulator?

`print_precision` = 8
When printing out to the user, how many decimal digits to show at maximum.

`print_suppress` = False
When printing out to the user, whether we can suppress trailing zeros.

`pz_max` = 1000000000000.0
Maximum considered angular frequency in rad/s for PZ analyses.

`shooting_default_points` = 100
Default number of points for a shooting analysis.

`shooting_max_nr_iter` = 10000
Maximum number of NR iterations for shooting analyses.

`symb_formulate_with_gs` = False
Formulate the equations with conductances and at the last moment swap resistor symbols back in. It seems to make sympy play nicer. Sometimes.

`symb_sympy_manual_solver` = False
Enable the manual solver: solve the circuit equations one at a time as you might do "manually".

`transient_aposteriori_step_threshold` = 0.9
Step change threshold: we do not want to redo the iteraction if the aposteriori check suggests a step that is very close to the one we already used. A value of 0.9 seems to be a good idea.

`transient_max_nr_iter` = 20
Maximum number of NR iterations for transient analyses.

`transient_max_time_iter` = 0
Maximum number of time iterations for transient analyses Notice the default (0) means no limit is enforced.

`transient_no_step_control` = False
Disable all step control in transient analyses.

`transient_prediction_as_x0` = True
: In a transisent analysis, if a prediction value is avalilable, use it as first guess for `x(n+1)`, otherwise `x(n)` is used.

`transient_use_aposteriori_step_control` = True
: Use aposteriori step control?

`use_gmin_stepping` = True
: Whether the gmin-settping homothopy can be used.

`use_source_stepping` = True
: Whether the source-stepping homothopy can be used.

`use_standard_solve_method` = True
: Whether the standard solving method can be used.

`vea` = 1e-06
: Voltage absolute tolerance.

`ver` = 0.001
: Voltage relative tolerance.

4.18 ahkab.plotting

This module offers the functions needed to plot the results of a simulation.

It is only functional if matplotlib is installed.

4.18.1 Module reference

`plot_results`(*title*, *y2y1_list*, *results*, *outfilename=None*)

Plot the results.

Parameters:

title [string] The plot title

y2y1_list [list] A list of tuples. Each tuple has to be in the format `(y2, y1)`. Each member of the tuple has to be a valid identifier. You can check the possible voltage and current identifiers in the result set calling `res.keys()`, where `res` is a solution object.

result [solution object or derivate] The results to be plotted.

outfilename [string, optional] The filename of the output file. If left unset, the plot will not be written to disk. The format is set through `options.plotting_outtype`.

Returns:

`None`.

`save_figure`(*filename*, *fig=None*)

Apply the figure options for saving and then save the supplied figure to `filename`.

The format of the output figure is set by `options.plotting_outtype`.

Parameters:

filename [string] The output filename.

fig [figure object, optional] The figure to be saved.

Returns:

`None`.

show_plots()
See the fruit of your work!

4.19 ahkab.printing

This is the printing module of the simulator. Using its functions, the output will be somewhat uniform.

The functions defined in this module can be divided in the following groups:

- *Informative functions*: functions to print information, errors and warnings to the user during command-line execution.
- *Printing netlist lines*: functions to print conformingly to the netlist syntax, often to show information to the user for debugging purposes.
- *Convenience functions*: functions to abstract low level issues such as Unicode handling of text and number printing formats,
- *Tabular formatting of data*: functions to format and display data into tables, which we provide straight from the `tabulate` module.
- *Printing analysis results* in a consistent fashion.

4.19.1 Informative functions

print_general_error(description[, ...])	Prints an error message to `stderr`
print_info_line(msg_relevance_tuple, verbose)	Conditionally print out a message
print_parse_error(nline, line[, print_to_stdout])	Prints a parsing error to `stderr`
print_warning(description[, print_to_stdout])	Prints a warning message to `stderr`

4.19.2 Printing netlist lines

print_analysis(an)	Prints an analysis to `stdout` in the netlist syntax

4.19.3 Printing analysis results

print_fourier(label, f, F, THD[, outfile])	Print the results of a Fourier postprocess
print_spicefft(label, f, F[, THD, uformat, ...])	Print the results of an FFT postprocess
print_symbolic_equations(eq_list)	Print symbolic equations for visual inspection
print_symbolic_results(x)	Print out symbolic results
print_symbolic_transfer_functions(x)	Print symbolic transfer functions

Convenience functions

`open_utf8`(filename)	Get a file handle wrapped in a UTF-8 writer
`printoptions`(*args, **kwds)	A context manager for `numpy.set_printoptions`

Tabular formatting of data

`table`(data, *args, **argsd)	Format a fixed width table for pretty printing

4.19.4 All functions in alphabetical order

`open_utf8` (*filename*)

Get a file handle wrapped in a UTF-8 writer

The file is opened in `w` mode.

Parameters:

filename [string] The file name, just like you would pass to Python's built-in `open()` method.

Returns:

fp [codecs.UTF8Writer object] The wrapped file pointer.

`print_analysis` (*an*)

Prints an analysis to `stdout` in the netlist syntax

Parameters:

an [dict] An analysis description in dictionary format.

`print_fourier` (*label*, *f*, *F*, *THD*, *outfile=u'stdout'*)

Print the results of a Fourier postprocess

Parameters:

label [str, or tuple of str] The identifier of a variable. Eg. `'Vn1'` or `'I(VS)'`. If a tuple of two identifiers is provided, it will be interpreted as the difference of the two, in the form `label[0]-label[1]`.

f [ndarray of floats] The frequencies, including the DC.

F [ndarray of complex data] The result of the Fourier transform.

THD [float, optional] The total harmonic distortion, if `freq` was specified in the FFT analysis. Defaults to `None`.

outfile [str, optional] The file name to print to. Defaults to `'stdout'`, for the standard output.

`print_general_error` (*description*, *print_to_stdout=False*)

Prints an error message to `stderr`

Parameters:

description [str] The error description.

print_to_stdout [bool, optional] When set to `True`, printing to `stdout` instead of `stderr`. Defaults to `False`.

`print_info_line` (*msg_relevance_tuple*, *verbose*, *print_nl=True*)
Conditionally print out a message

Parameters:

msg_relevance_tuple [sequence] A tuple or list made of `msg` and `importance`, where `msg` is a string, containing the information to be displayed to the user, and `importance`, an integer, is its importance level. Zero corresponds to the highest possible importance level, which is always printed out by the simple algorithm discussed below.

verbose [int] The verbosity level of the program execution. Admissible levels are in the 0-6 range.

print_nl [boolean, optional] Whether a new line character should be appended or not to the string `msg` described above, if it's printed out. Defaults to `True`.

Algorithm selecting when to print:

The message `msg` is printed out if the verbosity level is greater or equal than its importance.

`print_parse_error` (*nline*, *line*, *print_to_stdout=False*)
Prints a parsing error to `stderr`

Parameters:

nline [int] The number of the line on which the error occurred.

line [str] The line of the file with the error.

print_to_stdout [bool, optional] When set to `True`, printing to `stdout` instead of `stderr`. Defaults to `False`.

`print_result_check` (*badvars*, *verbose=2*)
Prints out the results of an OP check

It assumes one set of results is calculated with G_{min}, the other without.

Parameters:

badvars [list] The list returned by `results.op_solution.gmin_check()`.

verbose [int, optional] The verbosity level, from 0 (silent) to 6.

`print_spicefft` (*label*, *f*, *F*, *THD=None*, *uformat=u'NORM'*, *window=None*, *outfile=u'stdout'*)
Print the results of an FFT postprocess

Parameters:

label [str, or tuple of string] The identifier of a variable. Eg. `'Vn1'` or `'I(VS)'`. If a tuple of two identifiers is provided, it will be interpreted as the difference of the two, in the form `label[0]-label[1]`.

f [ndarray of floats] The frequencies, including the DC.

F [ndarray of complex data] The result of the Fourier transform.

THD [float, optional] The total harmonic distortion, if `freq` was specified in the FFT analysis. Defaults to `None`.

uformat [str, optional] The parameter format selects whether normalized or unnormalized magnitudes are printed. It is to be set to 'NORM' (default value) for normalized magnitude, to 'UNORM' for unnormalized.

window [str, optional] The window employed in the FFT analisys. Defaults to rectangular.

outfile [str] The file name to print to. Defaults to `'stdout'`, for the standard output.

print_symbolic_equations(*eq_list*)
Print symbolic equations for visual inspection

Parameters:

eq_list [list] The list of equations to be printed. This is what `sympy` will be asked to solve, typically.

print_symbolic_results(*x*)
Print out symbolic results

Parameters:

x [dict] A dictionary composed of elements like `{v:expr}`, where `v` is a circuit variable and `expr` is the `sympy` expression corresponding to it, as found by the solver.

print_symbolic_transfer_functions(*x*)
Print symbolic transfer functions

Parameters:

x [dict] A dictionary of dictionaries. Each top level dictionary is a symbol : symbolic transfer function pair, eg. `{vo/vin:<tf>}`. Each transfer function (`<tf>`) is itself a dictionary, having as keys the following strings: `'gain'`, corresponding to the complete symbolic TF expression, `'gain0'`, corresponding to the DC gain and `'poles'` and `'zeros'`, corresponding to lists of symbolic expressions of the singularities.

print_warning(*description*, *print_to_stdout=False*)
Prints a warning message to `stderr`

Parameters:

description [str] The warning message.

print_to_stdout [bool, optional] When set to `True`, printing to `stdout` instead of `stderr`. Defaults to `False`.

printoptions(**args*, ***kwds*)
A context manager for `numpy.set_printoptions`

table(*data*, **args*, ***argsd*)
Format a fixed width table for pretty printing

No data processing is done here, instead we call tabulate's `tabulate.tabulate()`, passing all arguments unmodified.

Parameters:

data [list-of-lists or a dictionary of iterables or a 2D NumPy array (or more).] The tabular data.

The remaining arguments, not documented here, are:

headers [sequence, optional] An explicit list of column headers.

tablefmt [str, optional] Table formatting specification.

floatfmt [str, optional] Floats formatting specification.

numalign [str, optional] Alignment flag for numbers.

stralign [str, optional] Alignment specification for strings, eg. "right".

missingval [str, optional] Element for the missing values.

4.20 ahkab.pss

Periodic Steady State (PSS) analysis module.

This module is an interface to a generic PSS analysis, which will be set up automatically for you according to the algorithm selected.

`pss_analysis`(**largs*, ***args*)

Perform a PSS analysis.

The only required argument is `method` (string), which selects the algorithm to be used.

Two algorithms are a shooting PSS, selected with the `"shooting"` switch and brute-force PSS, selected by the `"brute-force"` switch. Any other value for the `method` parameter will result in a `ValueError` exception being raised.

The rest of the arguments will be passed to the algorithm implementing the analysis.

For the shooting algorithm see *`ahkab.shooting`*, for the brute-force algorithm see *`ahkab.bfpss`*.

Returns:

sol [PSS solution object (`results.pss_solution`)] The solution.

4.21 ahkab.pz

This module offers the functions needed to perform a numeric pole-zero extraction.

Currently, this module implements the MD algorithm, more may be added in the future.

A description of the algorithm is found in the following references:

> Haley, S.B., "The generalized eigenproblem: pole-zero computation," *Proceedings of the IEEE*, vol.76, no.2, pp.103,120, Feb 1988

and:

> Raghuram, R.; Divekar, D.; Wang, P., "Implementation of pole-zero analysis in SPICE based on the MD method," Circuits and Systems, 1991., *Proceedings of the 34th Midwest Symposium on*, pp.380, 383 vol.1, 14-17 May 1991

Frequency sweeping – or shifting – is performed with a random frequency kick, currently, hoping not to kick so hard that we end up on the negative side. A bisection method would be better and hopefully will be implemented soon.

4.21.1 Overview

Two main methods are available in this module:

- *`calculate_singularities()`*, which computes both zeros and poles,

- `calculate_poles()`, which only computes the poles.

Currently this module uses dense matrices.

4.21.2 Reference

`calculate_poles`(*mc*, *MNA=None*, *x0=None*, *outfile=None*, *verbose=0*)

Calculate the circuit poles.

Parameters:

mc [circuit instance] The circuit to be analyzed.

MNA [ndarray, optional] The Modified Nodal Analysis matrix, if available. In case the circuit is non-linear, MNA should include the contributes of the non-linear elements (ie the Jacobian J).

x0 [ndarray or op_solution, optional] The linearization point. Only needed for non-linear circuits.

outfile [str or None, optional] The data filename.

verbose [int, optional] Verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

pz_sol [pz_solution instance] The PZ solution, with no zeros.

`calculate_singularities`(*mc*, *input_source=None*, *output_port=None*, *MNA=None*, *x0=None*, *shift=0*, *outfile=None*, *verbose=0*)

Calculate poles and zeros.

By default, only poles are calculated, as they need no information other than the circuit description.

To activate zeros calculation, it is necessary:

- to specify an input source (`input_source`),
- to specify an output port (`output_port`).

Parameters:

mc [circuit instance] The circuit to be analyzed.

input_source [string or element, optional] If zeros are to be calculated, set this to the input surce.

output_port [external node (ref. to gnd) or tuple of external nodes, opt] If zeros are to be calculated, set this to the output nodes.

MNA [ndarray, optional] The Modified Nodal Analysis matrix, if available. In case the circuit is non-linear, MNA should include the contributes of the non-linear elements (ie the Jacobian J).

x0 [ndarray or op_solution, optional] The linearization point. Only needed for non-linear circuits.

shift [float, optional] Shift frequency at which the algorithm should be run.

outfile [str or None, optional] The data filename.

verbose [int, optional] Verbosity level, from 0 (silent, default) to 6 (debug).

Returns:

pz_sol [pz_solution instance] The PZ solution

4.22 ahkab.results

This module provides classes for easy, dictionary-like access to simulation results.

Simulation results are typically returned upon successful simulation of a circuit and the user is not expected to use their constructor, but rather to use the methods they provide to access their data set.

4.22.1 Overview of the data interface

The solution classes define special methods according to their simulation type but they all subclass `solution`, which provides the shared data interface.

The interface allows for accessing the values as:

```
>>> ac_sol.keys()
['f', 'Vn1', 'Vn2', 'I(V1)', 'I(L1)', 'I(L2)']
```

Where `ac_sol` is a generic example instance of `ac_solution`.

Checking with the `in` construct:

```
>>> 'Vn1' in ac_sol
True
```

Access any variable in the solution object:

```
>>> ac_sol['f']
array([ 6098.38572827,  6102.08394991,  6105.78441425,  6109.48712265,
        6113.19207648,  6116.89927708,  6120.60872583,  6124.32042408,

        [... omissis ...]

        6463.83880528,  6467.75864729,  6471.68086639])
```

Iterate over the results:

```
>>> for var in ac_sol:
...     # do something with ac_sol[var]
...     pass
```

Convenience methods are available to identify and access the independent, swept variable, when it is available:

```
>>> ac_sol.get_xlabel()
'f'
>>> ac_sol.get_x()
array([ 6098.38572827,  6102.08394991,  6105.78441425,  6109.48712265,
        6113.19207648,  6116.89927708,  6120.60872583,  6124.32042408,

        [... omissis ...]

        6463.83880528,  6467.75864729,  6471.68086639])
```

4.22.2 Index of the solution classes

`ac_solution`(circ, start, stop, points, ...)	AC results
`dc_solution`(circ, start, stop, sweepvar, ...)	DC results
`op_solution`(x, error, circ, outfile[, ...])	OP results
`pss_solution`(circ, method, period, outfile)	PSS results
`pz_solution`(circ, poles, zeros, outfile)	PZ results
`symbolic_solution`(results_dict, ...[, ...])	Symbolic results
`tran_solution`(circ, tstart, tstop, op, ...)	Transient results

4.22.3 Module reference

class **`ac_solution`** (*circ*, *start*, *stop*, *points*, *stype*, *op*, *outfile*)

Bases: `ahkab.results.solution`, `ahkab.results._mutable_data`

AC results

Parameters:

circ [circuit instance] the circuit instance of the simulated circuit

start [float] the AC sweep frequency start value, in Hz.

stop [float] the AC sweep frequency stop value, in Hz.

points [int] the AC sweep total points.

stype [str] the type of sweep, `"LOG"`, `"LIN"` or arb. `"POINTS"`.

op [op_solution] the linearization Operating Point used to compute the results.

outfile: str the file to write the results to. Use `"stdout"` to write to the standard output.

`add_line` (*frequency*, *x*)

`asarray` ()

Return all data as a (possibly huge) python matrix.

`get` (*name*, *default=None*)

Get a solution by variable name.

`get_x` ()

`get_xlabel` ()

`has_key` (*name*)

Determine whether the result set contains a variable.

`items` ()

`keys` ()

Get all of the results set's variables names.

`next` ()

`values` ()

Get all of the results set's variables values.

class **`case_insensitive_dict`**

Bases: `object`

A dictionary that uses case-insensitive strings as keys.

get (*name*, *default=None*)
Given the case-insensitive string key `name`, return its corresponding value.

If not found, return `default`.

has_key (*name*)
Determine whether the result set contains the variable `name`.

items ()
Get all keys and values pairs

keys ()
Get all keys

update (*adict*)
Update the dictionary contents with the mapping in the dictionary `adict`.

values ()
Get all values

class dc_solution (*circ*, *start*, *stop*, *sweepvar*, *stype*, *outfile*)
Bases: *ahkab.results.solution*, `ahkab.results._mutable_data`

DC results

Parameters:

circ [circuit instance] the simulated circuit.

start [float] the DC sweep start value.

stop [float] the DC sweep stop value.

sweepvar [str] the swept variable `part_id`.

stype [str] the type of sweep, `"LOG"`, `"LIN"` or arb. `"POINTS"`.

outfile [str] the filename of the file where the results will be written. Use `"stdout"` to write to std output.

add_op (*sweepvalue*, *op*)
A DC sweep is made of a set of OP points.

This method adds an OP solution and its corresponding sweep value to the results set.

asarray ()
Return all data.

Note: This method loads to memory a possibly huge data matrix.

get (*name*, *default=None*)
Get a solution by variable name.

get_x ()

get_xlabel ()

has_key (*name*)
Determine whether the result set contains a variable.

items ()

keys()
Get all of the results set's variables names.

next()

values()
Get all of the results set's variables values.

class op_solution(*x*, *error*, *circ*, *outfile*, *iterations=0*)
Bases: *ahkab.results.solution*, `ahkab.results._mutable_data`

OP results

Parameters:

x [ndarray] the result set

error [ndarray] the residual error after solution,

circ [circuit instance] the circuit instance of the simulated circuit

outfile: str the file to write the results to. Use "stdout" to write to std output.

iterations, int, optional The number of iterations needed for convergence, if known.

asarray()
Get all data as a `numpy` array

get(*name*, *default=None*)
Get a solution by variable name.

get_table_array()

static gmin_check(*op2*, *op1*)
Checks the differences between two sets of OP results.

It is assumed that one set of results is calculated with Gmin, the other without.

Parameters:

op1, op2: op_solution instances the results vectors, interchangeable

Returns:

test_fail_variables [list] The list of the variables that did not pass the test. They are extracted from the op_solution objects. If the check was passed, this is an empty list.

has_key(*name*)
Determine whether the result set contains a variable.

items()

keys()
Get all of the results set's variables names.

next()

print_short()
Print a short, essential representation of the OP results

values()
Get all of the results set's variables values.

write_to_file(*filename=None*)

class **`pss_solution`**(*circ*, *method*, *period*, *outfile*)

Bases: *`ahkab.results.solution`*, `ahkab.results._mutable_data`

PSS results

Parameters:

circ [circuit instance] the circuit instance of the simulated circuit.

method [str] the PSS algorithm employed.

period [float] the solution period.

outfile [str] the filename of the save file. Use "stdout" to write to the std output.

Note: Instantiating `pss_solution` creates an *empty* data set. Call *`set_results()`* to initialize its data.

`asarray`()

`get`(*name*, *default=None*)

Get a solution by variable name.

`get_x`()

`get_xlabel`()

`has_key`(*name*)

Determine whether the result set contains a variable.

`items`()

`keys`()

Get all of the results set's variables names.

`next`()

`set_results`(*t*, *x*)

Set the results in the data set

Note:

- All the data are set at the same time for a PSS results set.
- Instantiating `pss_solution` creates an empty data set.
- This method should be called as soon as the data is available.

Parameters:

t [ndarray] The time. The array should be 2D with shape `(1, N)`.

x [ndarray] The data corresponding to the variables. The array should be 2D with shape `(M, N)`, where `M` is the number of variables in the data set.

`values`()

Get all of the results set's variables values.

class **`pz_solution`**(*circ*, *poles*, *zeros*, *outfile*)

Bases: *`ahkab.results.solution`*, `ahkab.results._mutable_data`

PZ results

Parameters:

circ [circuit instance] the circuit instance of the simulated circuit.

poles [sequence] the circuit zeros

zeros [sequence] the circuit poles

outfile [str] the filename of the save file.

`asarray`()
Return all data.

Note: This method loads to memory a possibly huge data matrix.

`get`(*name*, *default=None*)

`has_key`(*name*)
Determine whether the result set contains a variable.

`items`()

`keys`()
Get all of the results set's variable's names.

`next`()

`values`()
Get all of the results set's variable's values.

class `solution`(*circ*, *outfile*)
Bases: `object`

Base class storing a set of generic simulation results.

This class is not meant to be accessed directly, rather it is subclassed by the classes for the specific simulation solutions.

Parameters:

circ [circuit instance] the circuit instance of the simulated circuit.

outfile [string] the filename of the save file

`asarray`()
Return all data.

Note: This method loads to memory a possibly huge data matrix.

`get`(*name*, *default=None*)
Get a solution by variable name.

`has_key`(*name*)
Determine whether the result set contains a variable.

`items`()

`keys`()
Get all of the results set's variables names.

`next`()

values()
Get all of the results set's variables values.

class **symbolic_solution**(*results_dict*, *substitutions*, *circ*, *outfile=None*, *tf=False*)
Bases: `object`

Symbolic results

Parameters:

results_dict [dict] the results dict returned by `sympy.solve()`,

substitutions [dict] the substitutions (dictionary) employed before solving,

circ [circuit instance] the circuit instance of the simulated circuit.

outfile [str, optional] the filename of the save file. Use `"stdout"` to write to the standard output.

tf [bool, optional] Transfer function flag: set this to `True` if this set of results corrsponds to a transfer function. Defaults to `False`.

as_symbol(*variable*)
Converts a string to the corresponding symbolic variable.

This symbol may then be used by the user as an atom to construct new expressions, modify the results expressions or it can be passed to Sympy's functions.

Parameters:

variable [string] The string that identifies the variable. Eg. `'R1'` for the variable corresponding to the resistance of the resistor `R1`. Note that the case is disregarded and that the first letter defines the type of the element (resistor, capacitor...).

Returns:

symbol [Sympy symbol] The corresponding symbol, if it exists in the result set.

Raises:

ValueError [exception] In case no such symbol is found.

as_symbols(*spacedstr*)
Convenience function to call `as_symbol()` multiple times.

Parameters:

spacedstr [string,] A string containing several symbol identifiers separated by spaces. Eg. `'R1 C2 L3'`.

Returns:

(s1, s2, ...) [tuple of Sympy symbol instances] The symbols corresponding to the identifiers in the string supplied, ordered as the identifiers in the string.

Raises:

ValueError [exception] In case any corresponding symbol is not found.

get(*name*, *default=None*)
Get the solution corresponding to a variable.

has_key(*name*)
Determine whether the result set contains a variable.

items()
Get all solutions.

keys()
Get all of the results set's variable's names.

static **load**(*filename*)
Static method to load a symbolic solution from disk.

Parameters:

filename [str] The filename corresponding to the file to load from.

Returns:

sol [symbolic solution instance] The solution instance loaded from disk.

Warning: This method employs `pickle.load`, which is to be used exclusively on trusted data. **Only load trusted simulation files!**

next()

save()
Write the results to disk.

It is necessary first to set the `filename` attribute, indicating which file to write to.

Raises:

RuntimeError [exception] If the *filename* attribute is not set.

values()
Get all of the results set's variable's values.

class **tran_solution**(*circ*, *tstart*, *tstop*, *op*, *method*, *outfile*)
Bases: *ahkab.results.solution*, `ahkab.results._mutable_data`

Transient results

Parameters:

circ [circuit instance] the circuit instance of the simulated circuit.

tstart [float] the transient simulation start time.

tstop [float] the transient simulation stop time.

op [op_solution instance] the Operating Point (OP) used to start the transient analysis.

method [str] the differentiation method employed.

outfile [str] the filename of the save file. Use "stdout" to write to the standard output.

add_line(*time*, *x*)
This method adds a solution and its corresponding time value to the results set.

asarray()
Return all data.

Note: This method loads to memory a possibly huge data matrix.

get (*name*, *default=None*)
Get a solution by variable name.

get_x ()

get_xlabel ()

has_key (*name*)
Determine whether the result set contains a variable.

items ()

keys ()
Get all of the results set's variables names.

lock ()

next ()

values ()
Get all of the results set's variables values.

4.23 ahkab.shooting

Periodic steady state analysis based on the shooting method.

shooting_analysis (*circ*, *period*, *step=None*, *x0=None*, *points=None*, *autonomous=False*, *matrices=None*, *outfile=u'stdout'*, *vector_norm=<function <lambda>>*, *verbose=3*)
Performs a periodic steady state analysis based on the algorithm described in:

Brambilla, A.; D'Amore, D., "Method for steady-state simulation of strongly nonlinear circuits in the time domain," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol.48, no.7, pp.885-889, Jul 2001.

http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=933329&isnumber=20194

The results have been computed again by me, the formulation is not exactly the same, but the idea behind the shooting algorithm is.

This method allows us to have a period with many points without having to invert a huge matrix (and being limited to the maximum matrix size).

A transient analysis is performed to initialize the solver.

We compute the change in the last point, calculating several matrices in the process. From that, with the same matrices we calculate the changes in all points, starting from 0 (which is the same as the last one), then 1, ...

Key points:

- Only non-autonomous circuits are supported.
- The time step is constant.
- Implicit Euler is used as DF.

Parameters:

circ [Circuit instance] The circuit description class.

period [float] The period of the solution.

step [float, optional] The time step between consecutive points. If not set, it will be computed from `period` and `points`.

points [int, optional] The number of points to be used. If not set, it will be computed from `period` and `step`.

autonomous [bool, optional] This parameter has to be `False`, autonomous circuits are not currently supported.

matrices [dict, optional] A dictionary that may have as keys 'MNA', 'N' and 'D', with entries set to the corresponding MNA-formulation matrices, in case they have been already computed and the user wishes to save time by reusing them. Defaults to `None` (recompute).

outfile [string, optional] The output filename. Please use `stdout` (the default) to print to the standard output.

verbose [boolean, optional] Verbosity switch (0-6). It is set to zero (print errors only) if `outfile` == 'stdout'``, as not to corrupt the data.

Notice that `step` and `points` are mutually exclusive options:

- if `step` is specified, the number of points will be automatically determined.
- if `points` is set, the step will be automatically determined.
- if none of them is set, `options.shooting_default_points` will be used as points.

Returns:

sol [PSS solution object or `None`] The solution. If the circuit can't be solve, `None` is returned instead.

4.24 ahkab.switch

Implementation of a voltage controlled switch.

This module defines two classes: switch_device, switch_model

class **`switch_device`**(*n1*, *n2*, *sn1*, *sn2*, *model*, *ic=None*, *part_id=u'S'*)

This is a general switch element.

It has the following structure:

In ASCII for those who are consulting the documentation from the Python command line:

```
sn1 o--+          +--o n1
       |          |
     +-+        \ o
     |R|         \
     +-+          +
       |          |
sn2 o--+          +--o n2
```

The behavior is set by the model supplied.

The device instance calls the following methods in the model:

- `get_i(ports_v, device)` - output current
- `get_go(ports_v, device)` - ouput conductance
- `get_gm(ports_v, device)` - output transconductance
- `get_dc_guess(self, is_on)` - guesses for OP

The device instance accesses the following attributes: `part_id` (a string), the device label.

Parameters:

n1 [str] Positive output node (+)

n2 [str] Negative output node (-)

sn1 [str] Positive input node (+)

sn2 [str] Negative input node (-)

model [model obj] An instance of (v)switch_model

ic [bool, optional] The initial conditions: `True` stands for on, `False` for off.

Selected methods:

- *`get_output_ports()`* -> (n1, n2)
- *`get_drive_ports()`* -> (n1, n2), (ns1, ns2)

g (*op_index*, *ports_v*, *port_index*, *time=0*)

Returns the differential (trans)conductance.

The transconductance is computed wrt the port specified by `port_index` when the element has the voltages specified in `ports_v` across its ports, at (simulation) `time`.

Parameters:

ports_v [list] Voltages applied to the switch. The list should be in the form: `[voltage_across_port0, voltage_across_port1, ... ]`

port_index [int] The index of the output port.

time [float] The simulation time at which the evaluation is performed. Set it to `None` during DC analysis.

Returns:

g [float] The transconductance.

get_drive_ports (*op*)

Get the ports that drive the output ports.

Parameters:

op [op solution] The OP where the drive ports are used.

Returns:

pts : tuple of tuples of ports nodes, as: `(port0, port1, port2 ... )`

Where each port is in the form: `port0 = (nplus, nminus)`

get_netlist_elem_line (*nodes_dict*)
Return a netlist line corresponding to the switch.

get_op_info (*ports_v*)
Information regarding the Operating Point (OP)

Parameters:

ports_v [list of lists] The parameter is to be set to `[[v]]`, where `v` is the voltage applied to the switch terminals.

Returns:

op_keys [list of strings] The labels corresponding to the numeric values in `op_info`.

op_info [list of floats] The values corresponding to `op_keys`.

get_output_ports ()
Get the output port.

The output port is `(n1, n2)` for the voltage-controlled switch case.

Returns:

pts [tuple of tuples of ports nodes] Such as: `(port0, port1, port2 ... )`. Where each port is in the form: `port0 = (nplus, nminus)`

get_value_function (*identifier*)

i (*op_index*, *ports_v*, *time=0*)
Returns the current flowing in the element.

The element is assumed to be biased with the voltages applied as specified in the `ports_v` vector.

Parameters:

op_index [int] The index of the output port for which the current is evaluated.

ports_v [tuple] A tuple constructed such as `(voltage_across_port0, voltage_across_port1, ... )`

time [float, optional] The simulation time at which the evaluation is performed. It is needed by time-variant elements, and it has no effect here. Set it to `None` during DC analysis.

Returns:

i [int] The output current.

update_status_dictionary (*ports_v*)
Updates an internal dictionary that can then be used to provide information to the user regarding the status of the element.

Normally, one would call `get_op_info()`.

Returns:

`None`.

class vswitch_model (*name*, *VT=None*, *VH=None*, *VON=None*, *VOFF=None*, *RON=None*, *ROFF=None*)
Voltage-controlled switch model.

```
    sn1 o--+          +--o n1
           |          |
          +-+       \ o
          |R|        \
          +-+         +
           |          |
    sn2 o--+          +--o n2
```

Note that:

- R is infinite.
- The voltage needed to close the switch is: $V(s_{n1}) - V(s_{n2}) > V_T + V_H$.
- To re-open it, one needs to satisfy the relationship: $V(s_{n1}) - V(s_{n2}) < V_T - V_H$.

The switch commutes between two statuses:

- $R_{OUT} = R_{OFF}$
- $R_{OUT} = R_{ON}$

None of which can be set to zero or infinite.

The switching characteristics are modeled with $tanh(x)$.

get_dc_guess(*is_on*)
Returns a list of two floats to be used as initial guesses for the OP analysis

get_gm(*xxx_todo_changeme2*, *dev*, *debug=False*)
Returns the source to output transconductance or d(I)/d(Vsn1-Vsn2).

get_go(*xxx_todo_changeme1*, *dev*, *debug=False*)
Returns the output conductance d(I)/d(Vn1-Vn2).

get_i(*xxx_todo_changeme*, *dev*, *debug=False*)
Returns the output current.

print_model()
All the internal parameters of the model get printed out, for visual inspection.

4.25 ahkab.symbolic

This module provides the functionality needed to perform a small-signal symbolic simulation.

The principal method is `symbolic_analysis()`, which performs the symbolic circuit solution.

Note: This module is geared towards *setting up and running* the symbolic simulation. Typically, it should be used in conjunction with `ahkab.results.symbolic_solution`, the symbolic solution class, which holds several convenience methods to extensively manipulate, simplify, post-process and analyze the simulation results, complementing the functionality offered by the Sympy module itself.

4.25.1 Reference

calculate_gains(*sol*, *xin*, *optimize=True*)
Calculate low-frequency gain and roots of a transfer function.

Parameters:

sol [dict] the circuit solution

xin [Sympy symbol] the input variable

optimize [boolean, optional] If `optimize` is set to `False`, no algebraic simplification will be attempted on the results. The default (`optimize=True`) results in `sympy.together` being called on each expression.

Returns:

gs [dict] A dictionary with as keys the strings <key>/<xin> and as values dictionaries with keys `'gain'`,`'gain0'`,`'poles'`,`'zeros'`.

generate_mna_and_N(*circ*, *opts*, *ac=False*, *subs=None*, *verbose=3*)

Generate a symbolic Modified Nodal Analysis matrix and N vector.

Only elements that have an `is_symbolic` attribute set to `True` (the default) are considered symbolically. Simply set the attribute to `False` to employ the numeric value. This allows to simplify and speed up the work of the symbolic solver.

The formulation can be performed using conductances or resistances. The choice is made setting the global `options.symb_formulate_with_gs` value to `True`. A formulation done in terms of resistors, may result in many separate $1/R$ terms in the matrices. Historically, `sympy` choked on those, because of a long-standing bug in polys. Now the issue seems to have been solved and the two computations should be symbolically equivalent albeit computationally different (as expected). The option value `options.symb_formulate_with_gs` is provided to restore the old functionality in case you use an old version of `sympy`.

Parameters:

circ [circuit instance] The circuit.

opts [dict] The options to be used for the generation of the matrices. As of now, the only supported option is `'r0s'` which can be set to either `True` or `False`, and selects whether the equivalent output resistance of the transistors should be taken into account or not.

ac [bool, optional] Flag to trigger the inclusion of frequency-dependent elements. Defaults to `False` currently (but may change).

subs [dict, optional] The substitution dictionary, composed by mappings of `<symbol>`:`<sympy expression>`.

verbose [int, optional] Verbosity flag, from `0` (silent) to `6` (very logorrhoic). Defaults to `3`.

Returns:

mna, N [Sympy matrices] The MNA matrix and the contant term of symbolic type.

subs_gs [dict of symbols] In case the formulation of the MNA is performed in terms of conducatances, this dictionary is to be used to substitute away the conducatances for the resistor symbols, after the circuit is solved but before the results are shown to the user. `sympy`‘s `sub()` can take care of that for you. If not necessary, this dictionary is empty.

Note: Setting `opts['r0s'] = True`, ie considering all the transistors output resistances, can significantly slow down – or even prevent by consuming all available memory – the solution of complex circuits with several active elements.

We recommend a combination of the following:

- setting the above option in simple circuits only,
- inserting explicitely the r_0 you wish to consider at circuit level,
- beefing up your machine with extra RAM and extra computing power,
- being patient.

---

get_roots (*expr*)
Given the transfer function `expr`, returns `poles, zeros`.

get_variables (*circ*)
Get a sympy matrix containing the circuit variables to be solved for.

Parameters:

circ [circuit instance] The circuit

Returns:

vars [sympy matrix, shape (n, 1)] The variables in a column vector.

s = s
the Laplace variable

symbolic_analysis (*circ*, *source=None*, *ac_enable=True*, *r0s=False*, *subs=None*, *outfile=None*, *verbose=3*)
Attempt a symbolic, small-signal solution of the circuit.

Parameters:

circ [circuit instance] the circuit instance to be simulated.

source [string, optional]

the `part_id` of the source to be used as input for the transfer function. If `None`, no transfer function is evaluated.

ac_enable [bool, optional] take frequency dependency into consideration (default: True).

r0s [bool, optional] take transistors' output impedance into consideration (default: False)

subs: dict, optional a dictionary of part IDs to be substituted. It makes solving the circuit easier. Eg. `subs={'R1':'R2'}` - replace the resistor R1 with R2.

outfile [string, optional] output filename - `'stdout'` means print to stdout, the default.

verbose: int, optional verbosity level 0 (silent) to 6 (painful).

Returns:

sol [symbolic solution] The solutions.

tfs [symbolic solution] The transfer functions, only if requested. Otherwise `tfs` is a `None` object.

4.26 ahkab.testing

A straight-forward framework to buid tests to ensure no regressions occur during development.

Two classes for describing tests are defined in this module:

- *`NetlistTest`*, used to run a netlist-based test,

- *APITest*, used to run an API-based test.

Every test, no matter which class is referenced internally, is univocally identified by a alphanumeric id, which will be referred to as `<test_id>` in the following.

4.26.1 Directory structure

The tests are placed in `tests/`, under a directory with the same id as the test, ie:

```
tests/<test_id>/
```

4.26.2 Running tests

The test is performed with as working directory one among the following:

- The ahkab repository root,
- `tests/`,
- `tests/<test_id>`.

this is necessary for the framework to find its way to the reference files.

More specifically a test can either be run manually through the Python interpreter:

```
python tests/<test_id>/test_<test_id>.py
```

or with the `nose` testing package:

```
nosetests tests/<test_id>/test_<test_id>.py
```

To run the whole test suite, issue:

```
nosetests tests/*/*.py
```

Please refer to the nose documentation for more info about the command `nosetests`.

4.26.3 Running your tests for the first time

The first time you run a test you defined yourself, no reference data will be available to check the test results and decide whether the test was passed or if a test fail occurred.

In this case, if you call `nose`, the test will (expectedly) fail.

Please run the test manually (see above) and the test framework will generate the reference data for you.

Please *check the generated reference data carefully!* Wrong reference defeats the whole concept of running tests!

4.26.4 Overview of a typical test based on `NetlistTest`

Each test is composed by multiple files.

Required files

The main directory must contain:

- `<test_id>.ini`, an INI configuration file containing the details of the test,
- `test_<test_id>.py`, the script executing the test,
- `<test_id>.ckt`, the main netlist file to be run.
- the reference data files for checking the pass/fail status of the test. These can be automatically generated, as it will be shown below.

With the exception of the netlist file, which is free for the test writer to define, and the data files, which clearly depend on the test at hand, the other files have a predefined structure which will be examined in more detail in the next sections.

Configuration file

Few rules are there regarding the entries in the configuration file.

They are as follows:

- The file name must be `<test_id>.ini`,
- It must be located under `tests/<test_id>/`,
- It must have a `[test]` section, containing the following entries:
  - `name`, set to the `<test_id>`, for error-checking,
  - `netlist`, set to the netlist filename, `<test_id>.ckt`, prepended with the the netlist path relative to `tests/<test_id>/` (most of the time that means just `<test_id>.ckt`)
  - `type`, a comma-separated list of analyses that will be executed during the test. Values may be `op`, `dc`, `tran`, `symbolic`... and so on.
  - One entry `<analysis>_ref` for each of the analyses listed in the `type` entry above. The value is recommended to be set to `<test_id>-ref.<analysis>` or `<test_id>-ref.<analysis>.pickle`, if you prefer to save data in Python's pickle format. Notice only trusted pickle files should ever be loaded.
  - `skip-on-travis`, set to either `0` or `1`, to flag whether this test should be run on Travis-CI or not. Torture tests, tests needing lots of CPU or memory, and long-lasting tests in general should be disabled on Travis-CI to not exceed:
    - a total build time of 50 minutes,
    - A no stdout activity time of 10 minutes.
  - `skip-on-pypy`, set to either `0` or `1`, to flag whether the test should be skipped if useing a PYPY Python implemetntation or not. In general, as PYPY supports neither `scipy` nor `matplotlib`, only symbolic-oriented tests make sense with PYPY (where it really excels!).

The contents of an example test configuration file `rtest1.ini` follow, as an example.

```
[test]
name = rtest1
netlist = rtest1.ckt
```

```
type = dc, op
dc_ref = rtest1-ref.dc
op_ref = rtest1-ref.op
skip-on-travis = 0
skip-on-pypy = 1
```

Script file

The test script file is where most of the action takes place and where the highest amount of flexibility is available.

That said, the ahkab testing framework was designed to make for extremely simple and straight-forward test scripts.

It is probably easier to introduce writing the scripts with an example.

Below is a typical script file.

```python
from ahkab.testing import NetlistTest
from ahkab import options
# add this to prevent interactive plot directives
# in the netlist from halting the test waiting for
# user input
options.plotting_show_plots = False

def myoptions():
    # optionally, set non-standard options
    sim_opts = {}
    sim_opts.update({'gmin':1e-9})
    sim_opts.update({'iea':1e-3})
    sim_opts.update({'transient_max_nr_iter':200})
    return sim_opts

def test():
    # this requires a netlist ``mytest.ckt``
    # and a configuration file ``mytest.ini``
    nt = NetlistTest('mytest', sim_opts=myoptions())
    nt.setUp()
    nt.test()
    nt.tearDown()

# It is recommended to set the docstring to a meaningful value
test.__doc__ = "My test description, printed out by nose"

if __name__ == '__main__':
    nt = NetlistTest('mytest', sim_opts=myoptions())
    nt.setUp()
    nt.test()
```

Notice how a function `test()` is defined, as that will be run by `nose`, and a `'__main__'` block is defined too, to allow running the script from the command line.

It is slightly non-standard, as *`NetlistTest.setUp()`* and *`NetlistTest.tearDown()`* are called inside `test()`, but this was found to be an acceptable compromise between complexity and following standard practices.

The script is meant to be run from the command line in case a regression is detected by `nose`, possibly with the aid of a debugger. As such, the *NetlistTest.tearDown()* function is not executed in the '__main__' block, so that the test outputs are preserved for inspection.

That said, the example file should be easy to understand and in most cases a simple:

```
:%s/mytest/<test_id>/g
```

in VIM - will suffice to generate your own script file. Just remember to save to `test_<test_id>.py`.

4.26.5 Overview of a typical test based on APITest

Required files

The main directory must contain:

- `test_<test_id>.py`, the script executing the test,
- the reference data files for checking the pass/fail status of the test. These can be automatically generated, as it will be shown below.

Script file

Again, it is probably easier to introduce the API test scripts with an example.

Below is a typical test script file:

```
import ahkab
from ahkab import ahkab, circuit, printing, devices, testing

cli = False

def test():
    """Test docstring to be printed out by nose"""

    mycircuit = circuit.Circuit(title="Butterworth Example circuit", filename=None)

    ## define nodes
    gnd = mycircuit.get_ground_node()
    n1 = mycircuit.create_node('n1')
    n2 = mycircuit.create_node('n2')
    # ...

    ## add elements
    mycircuit.add_resistor(name="R1", n1="n1", n2="n2", value=600)
    mycircuit.add_inductor(name="L1", n1="n2", n2=gnd, value=15.24e-3)
    mycircuit.add_vsource("V1", n1="n1", n2=gnd, dc_value=5, ac_value=.5)
    # ...

    if cli:
        print(mycircuit)

    ## define analyses
    op_analysis = ahkab.new_op(outfile='<test_id>')
    ac_analysis = ahkab.new_ac(start=1e3, stop=1e5, points=100, outfile='<test_id>')
    # ...
    ## create a testbench
    testbench = testing.APITest('<test_id>', mycircuit,
                                [op_analysis, ac_analysis],
                                skip_on_travis=True, skip_on_pypy=True)

    ## setup and test
    testbench.setUp()
    testbench.test()

    ## this section is recommended. If something goes wrong, you may call the
    ## test from the cli and the plots to video in the following will allow
    ## for quick inspection
    if cli:
        ## re-run the test to grab the results
        r = ahkab.run(mycircuit, an_list=[op_analysis, ac_analysis])
        ## plot and save interesting data
        fig = plt.figure()
        plt.title(mycircuit.title + " - TRAN Simulation")
        plt.plot(r['tran']['T'], r['tran']['VN1'], label="Input voltage")
        plt.hold(True)
        plt.plot(r['tran']['T'], r['tran']['VN4'], label="output voltage")
        plt.legend()
        plt.hold(False)
        plt.grid(True)
        plt.ylabel('Step response')
        plt.xlabel('Time [s]')
        fig.savefig('tran_plot.png')
    else:
        ## don't forget to tearDown the testbench when under nose!
        testbench.tearDown()

if __name__ == '__main__':
    import pylab as plt
    cli = True
    test()
    plt.show()
```

Once again, a function `test()` is defined, as that will be the entry point of `nose`, and a `'__main__'` block is defined as well, to allow running the script from the command line.

Inside `test()`, the circuit to be tested is defined, accessing the `ahkab` module directly, to set up elements, sources and analyses. Directly calling `ahkab.run()` is not necessary, `APITest.test()` will take care of that for you.

Notice how *`APITest.setUp()`* and *`APITest.tearDown()`* are called inside `test()`, as in the previous case.

The script is meant to be run from the command line in case a regression is detected by `nose`, possibly with the aid of a debugger. As such, the *`APITest.tearDown()`* function is not executed in the `'__main__'` block, so that the test outputs are preserved for inspection.

Additionally, plotting is performed if the test is directly run from the command line.

In case non-standard simulation options are necessary, they can be set as in the previous example.

4.26.6 Module reference

class **APITest** (*test_id*, *circ*, *an_list*, *er=1e-06*, *ea=1e-09*, *sim_opts=None*, *skip_on_travis=False*, *skip_on_pypy=True*)

A class to run a supplied circuit and check the results against a pre-computed reference.

Parameters:

test_id [string] The test id.

circ [circuit instance] The circuit to be tested

an_list [list of dicts] A list of the analyses to be performed

er [float, optional] Allowed relative error (applies to numeric results only).

er [float, optional] Allowed absolute error (applies to numeric results only).

sim_opts [dict, optional] A dictionary containing the options to be used for the test.

skip_on_travis [bool, optional] Should we skip the test on Travis? Set to `True` for long tests. Defaults to `False`.

skip_on_pypy [bool, optional] Should we skip the test on PYPY? Set to `True` for tests requiring libraries not supported by PYPY (eg. `scipy`, `matplotlib`). Defaults to `True`, as most numeric tests will fail.

setUp ()

Set up the testbench

tearDown ()

Remove temporary files - if needed.

class **NetlistTest** (*test_id*, *er=1e-06*, *ea=1e-09*, *sim_opts=None*, *verbose=6*)

A class to run a netlist file and check the results against a pre-computed reference.

Parameters:

test_id [string] The test id. For a netlist named `"rc_network.ckt"`, this is to be set to `"rc_network"`.

er [float, optional] Allowed relative error (applies to numeric results only).

er [float, optional] Allowed absolute error (applies to numeric results only).

sim_opts [dict, optional] A dictionary containing the options to be used for the test.

verbose [int] The verbosity level to be used in the test. From 0 (silent) to 6 (verbose). Notice higher verbosity values usually result in higher coverage. Defaults to 6.

setUp ()

Set up the testbench.

tearDown ()

Remove temporary files - if needed.

ok (*x*, *ref*, *rtol*, *atol*, *msg*)

4.27 ahkab.ticker

A progress indicator.

class ticker(*increments_for_step=10*)

This is a progress indicator class.

If activated, you shouldn't print anything to screen before calling ticker.hide().

If you wish to change the progress indicator, change self.progress to something else.

display(*enable=None*)

Print to screen the progress indicator. Call hide to hide it again.

hide(*enable=None*)

Before printing text to screen, call this to hide the progress indicator.

reset()

Reset to initial status. Doesn't hide it.

step()

After calling this function ticker.increments_for_step times the status is incremented.

4.28 ahkab.time_functions

This module contains several basic time functions.

The classes that are found in module are useful to provide a time-varying characteristic to independent sources.

Notice that the time functions are not restricted to those provided here, the user is welcome to provide his own. Implementing a custom time function is easy and common practice, as long as you are interfacing to the simulator through Python. Please see the dedicated section *Defining custom time functions* below.

4.28.1 Classes defined in this module

`pulse`(v1, v2, td, tr, pw, tf, per)	Square wave aka pulse function
`pwl`(x, y[, repeat, repeat_time, td])	Piece-Wise Linear (PWL) waveform
`sin`(vo, va, freq[, td, theta, phi])	Sine wave
`exp`(v1, v2, td1, tau1, td2, tau2)	Exponential wave
`sffm`(vo, va, fc, mdi, fs, td)	Single-Frequency FM (SFFM) waveform
`am`(sa, fc, fm, oc, td)	Amplitude Modulated (AM) waveform

4.28.2 Supplying a time function to an independent source

Providing a time-dependent characteristic to an independent source is very simple and probably best explained with an example.

Let's say we wish to define a sinusoidal voltage source with no offset, amplitude 5V and 1kHz frequency.

It is done in two steps:

- first we define the time function with the built-in class `ahkab.time_functions.sin`:

```
sin1k = time_functions.sin(vo=0, va=5, freq=1e3)
```

- Then we define the voltage source and we assign the time function to it:

```
        cir.add_vsource('V1', 'n1', cir.gnd, 1, function=mys)
```

In the example above, the sine wave is assigned to a voltage source `'V1'`, that gets added to a circuit `cir` (not shown).

4.28.3 Defining custom time functions

Defining a custom time function is easy, all you need is either:

- A function that takes a `float` (the time) and returns the function value,
- An instance with a `__call__(self, time)` method. This solution allows having internal parameters, typically set through the constructor.

In both cases, in time-based simulations, the simulator will call the object at every time step, supplying a single parameter, the simulation time (`time` in the following, of type `float`).

In turn, the simulator expects to receive as return value a `float`, corresponding to the value of the time-dependent function at the time specified by the `time` variable.

If the time-dependent function is used to define the characteristics of a voltage source (`VSource`), its return value has to be expressed in Volt. In the case of a current source (`ISource`), the return value is to be expressed in Ampere.

The standard notation applies.

As an example, we'll define a custom time-dependent voltage source, having a $\mathrm{sinc}(ft)$ characteristic. In this example, f has a value of 10kHz.

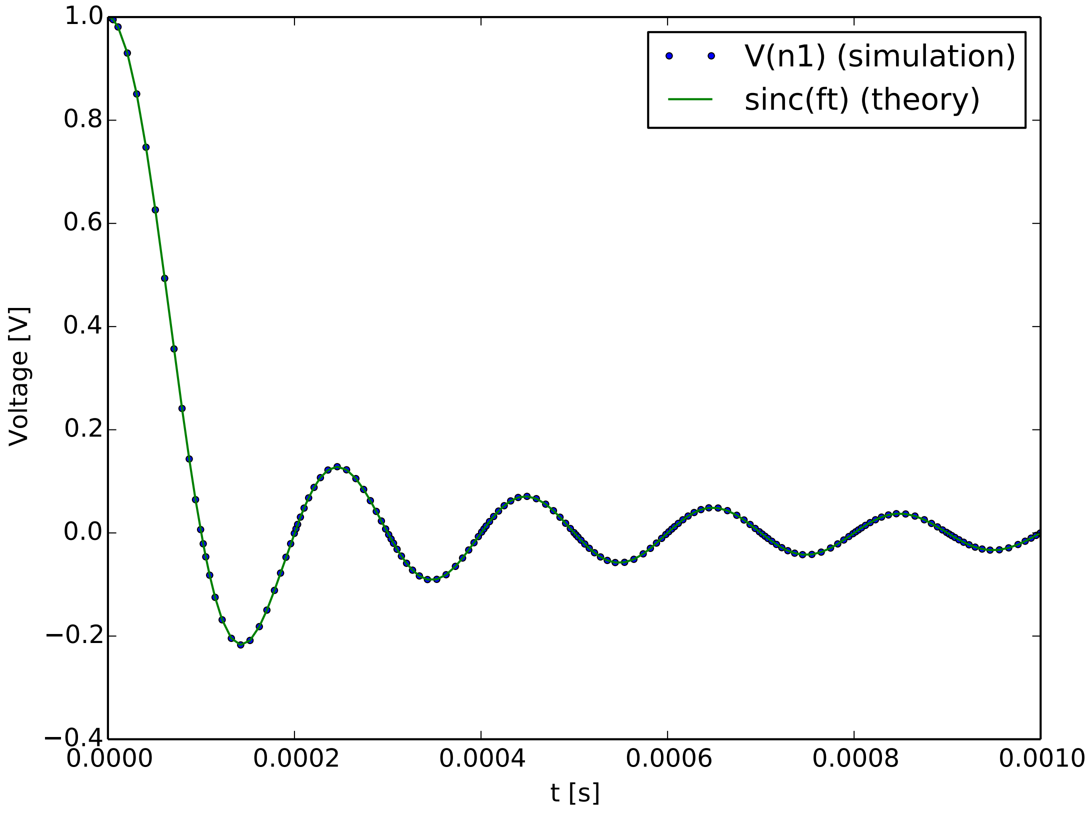
First we define the time function, in this case we'll do that through the Python `lambda` construct.

```
mys = lambda t: 1 if not t else math.sin(math.pi*1e4*t)/(math.pi*1e4*t)
```

Then, we define the circuit – a very simple one in this case – and assign our `mys` function to `V1`. In the following circuit, we simply apply the voltage from `V1` to a resistor `R1`.

```
import ahkab
cir = ahkab.Circuit('Test custom time functions')
cir.add_resistor('R1', 'n1', cir.gnd, 1e3)
cir.add_vsource('V1', 'n1', cir.gnd, 1, function=mys)
tr = ahkab.new_tran(0, 1e-3, 1e-5, x0=None)
r = ahkab.run(cir, tr)['tran']
```

Plotting `Vn1` and the expected result ($\mathrm{sinc}(ft)$) we get:



4.28.4 Module reference

class am (*sa*, *fc*, *fm*, *oc*, *td*)

Amplitude Modulated (AM) waveform

Mathematically, it is described by the equations:

- $0 \leq t \leq t_D$:

$$f(t) = O$$

- $t > t_D$

$$f(t) = SA \cdot [OC + \sin[2\pi f_m(t - t_D)]] \cdot \sin[2\pi f_c(t - t_D)]$$

Parameters:

sa [float] Signal amplitude in Volt or Ampere.

fc [float] Carrier frequency in Hertz.

fm [float] Modulation frequency in Hertz.

oc [float] Offset constant, setting the absolute magnitude of the modulation.

td [float] Time delay before the signal begins, in seconds.

class exp (*v1*, *v2*, *td1*, *tau1*, *td2*, *tau2*)

Exponential wave

Mathematically, it is described by the equations:

- $0 \leq t < TD1$:

$$f(t) = V1$$

- $TD1 < t < TD2$

$$f(t) = V1 + (V2 - V1) \cdot \left[1 - \exp\left(-\frac{t - TD1}{TAU1}\right)\right]$$

- $t > TD2$

$$f(t) = V1 + (V2 - V1) \cdot \left[1 - \exp\left(-\frac{t - TD1}{TAU1}\right)\right] + (V1 - V2) \cdot \left[1 - \exp\left(-\frac{t - TD2}{TAU2}\right)\right]$$

Parameters:

v1 [float] Initial value.

v2 [float] Pulsed value.

td1 [float] Rise delay time in seconds.

td2 [float] Fall delay time in seconds.

tau1 [float] Rise time constant in seconds.

tau2 [float] Fall time constant in seconds.

class pulse (*v1*, *v2*, *td*, *tr*, *pw*, *tf*, *per*)

Square wave aka pulse function

Parameters:

v1 [float] Square wave low value.

v2 [float] Square wave high value.

td [float] Delay time to the first ramp, in seconds. Negative values are considered as zero.

tr [float] Rise time in seconds, from the low value `v1` to the pulse high value `v2`.

tf [float] Fall time in seconds, from the pulse high value `v2` to the low value `v1`.

pw [float] Pulse width in seconds.

per [float] Periodicity interval in seconds.

class pwl (*x*, *y*, *repeat=False*, *repeat_time=0*, *td=0*)

Piece-Wise Linear (PWL) waveform

A piece-wise linear waveform is defined by a sequence of points (x_i, y_i).

Please supply the abscissa values $\{x\}_i$ in the vector `x`, the ordinate values $\{y\}_i$ in the vector `y`, separately.

Parameters:

x [sequence-like] The abscissa values of the interpolation points.

y [sequence-like] The ordinate values of the interpolation points.

repeat [boolean, optional] Whether the waveform should be repeated after its end. If set to `True`, `repeat_time` also needs to be set to define when the repetition begins. Defaults to `False`.

repeat_time [float, optional] In case the waveform is set to be repeated, setting the `repeat` flag above, the parameter, defined in seconds, set the first time instant at which the waveform repetition happens.

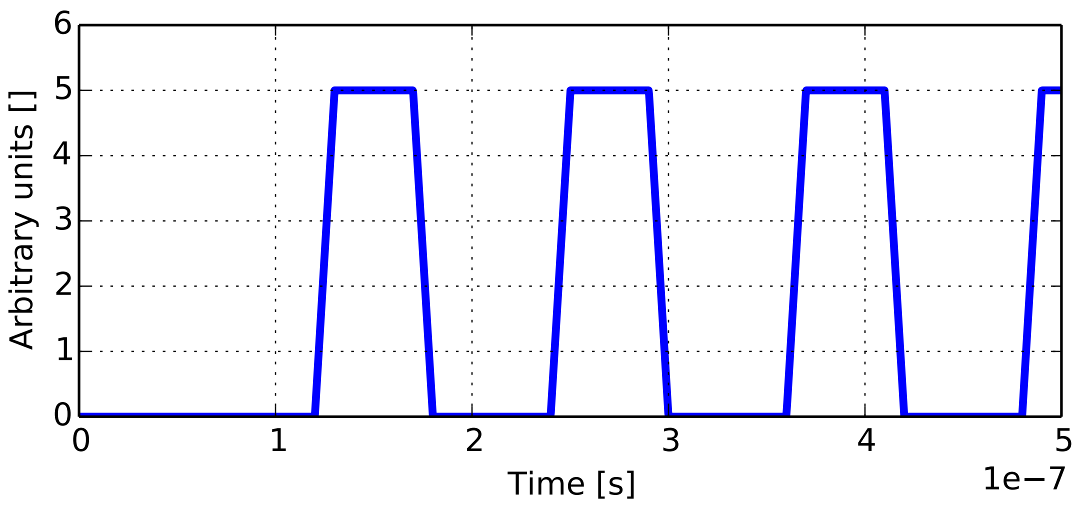
td [float, optional] Time delay before the signal begins, in seconds. Defaults to zero.

Example:

The following code:

```
import ahkab
import numpy as np
import pylab as plt
# vs = (x1, y1, x2, y2, x3, y3 ...)
vs = (60e-9, 0, 120e-9, 0, 130e-9, 5, 170e-9, 5, 180e-9, 0)
x, y = vs[::2], vs[1::2]
fun = ahkab.time_functions.pwl(x, y, repeat=1, repeat_time=60e-9, td=0)
myg = np.frompyfunc(fun, 1, 1)
t = np.linspace(0, 5e-7, 2000)
plt.plot(t, myg(t), lw=3)
plt.xlabel('Time [s]'); plt.ylabel('Arbitrary units []')
```

Produces:



class sffm(*vo*, *va*, *fc*, *mdi*, *fs*, *td*)

Single-Frequency FM (SFFM) waveform

Mathematically, it is described by the equations:

- $0 \leq t \leq t_D$:

$$f(t) = V_O$$

- $t > t_D$

$$f(t) = V_O + V_A \cdot \sin\left[2\pi f_C (t - t_D) + MDI \sin\left[2\pi f_S (t - t_D)\right]\right]$$

Parameters:

vo [float] Offset in Volt or Ampere.

va [float] Amplitude in Volt or Ampere.

fc [float] Carrier frequency in Hz.

mdi [float] Modulation index.

fs [float] Signal frequency in HZ.

td [float] Time delay before the signal begins, in seconds.

class sin (*vo*, *va*, *freq*, *td=0.0*, *theta=0.0*, *phi=0.0*)

Sine wave

Mathematically, the sine wave function is defined as:

- $t < t_d$:

$$f(t) = v_o + v_a \sin\left(\pi\phi/180\right)$$

- $t \geq t_d$:

$$f(t) = v_o + v_a \exp\left[-(t - t_d)\,\theta\right] \sin\left[2\pi f(t - t_d) + \pi\phi/180\right]$$

Parameters:

vo [float] Offset value.

va [float] Amplitude.

freq [float] Sine frequency in Hz.

td [float, optional] time delay before beginning the sinusoidal time variation, in seconds. Defaults to 0.

theta [float optional] damping factor in 1/s. Defaults to 0 (no damping).

phi [float, optional] Phase delay in degrees. Defaults to 0 (no phase delay).

Note: This implementation is consistent with the SPICE simulator, other simulators use different formulae.

4.29 ahkab.transient

This module provides the methods required to perform a transient analysis.

Our problem can be written as:

$$D \cdot dx/dt + MNA \cdot x + T_v(x) + T_t(t) + N = 0$$

We need:

1. MNA, the static Modified Nodal Analysis matrix,

2. N, constant DC term,
3. $T_v(x)$, the non-linear DC term
4. $T_t(t)$, the time variant term, time dependent-sources, to be evaluated at each time step,
5. The dynamic D matrix,
6. a differentiation method to approximate dx/dt.

`check_step` (*tstep*, *time*, *tstop*, *HMAX*)

Checks the step for several common issues and corrects them.

The following problems are checked:

- the step must be shorter than `HMAX`. In the context of a transient analysis, that usually is the time step provided by the user,
- the step must be equal or shorter than the simulation time left (ie stop time minus current time),
- the step must be longer than `options.hmin`, the minimum allowable time step. If the step goes below this value, convergence problems due to machine precision will occur. Typically when this happens, we halt the simulation.

Parameters:

tstep [float] The time step, in second, that needs to be checked.

time [float] The current simulation time.

tstop [float] The time at which the simulation ends.

HMAX [float] The maximum allowable time step.

Returns:

tstep [float] The step provided if it passes the tests, a *shortened* step otherwise.

Raises ValueError When the step is shorter than `option.hmin`.

class `dfbuffer` (*length*, *width*)

This is a LIFO buffer with a method to read it all without deleting the elements.

Newer entries are added on top of the buffer. It checks the size of the added elements, to be sure they are of the same size.

Parameters:

length [int] The length of the buffer. Samples are added at index `0`, shifting all the previous samples back to higher indices. Samples at an index equal to `length` (or higher) are discarded without notice.

width [int] The width of the buffer, every time `add()` is called, it must be to add a tuple of the same length as this parameter.

`add` (*atuple*)

Add a new data point to the buffer.

Parameters:

atuple [tuple of floats] The data point to be added. Notice that the length of the tuple must agree with the width of the buffer.

Raises ValueError if the provided tuple and the buffer width do not

match.

get_as_matrix()

get_df_vector()

Read out the contents of the buffer, without any modification

This method, in the context of a transient analysis, returns a vector suitable for a differentiation formula.

Returns:

vec [list of tuples] a list of tuples, each tuple being composed of `width` floats. In the context of a transient analysis, the list (or vector) conforms to the specification of the differentiation formulae. That is, the simulator stores in the buffer a list similar to:

```
[[time(n), x(n), dx(n)], [time(n-1), x(n-1), dx(n-1)], ...]
```

isready()

This shouldn't be used to determine if the buffer has enough points to use the df _if_ you use the step control. In that case, it holds even the points required for the FF.

generate_D(*circ*, *shape*)

Generates the D matrix

For every time t, the D matrix is used (elsewhere) to solve the following system:

$$Ddx/dt + MNAx + N + T(x) = 0$$

It's easy to set up the KCL law for the voltage unknowns, capacitors introduce stamps just like resistors do in the MNA and we know that row 1 refers to node 1, row 2 refers to node 2, and so on

Inductors generate, together with voltage sources, ccvs, vcvs, a additional line in the MNA matrix, and hence in D too.

The current flowing through the device gets added to the x vector.

In the case of an inductors, we have:

$$V(n_1) - V(n_2) - V_L = 0$$

Where:

$$V_L = LdI/dt$$

That's 0 (zero) in DC analysis, but not in transient analysis, where it needs to be differentiated.

To understand on which line does the inductor's L*dI/dt go, we use the order of the elements in *circuit*: first are all voltage lines, then the current ones in the same order of the elements that introduce them. Therefore, we need to access the circuit (*circ*).

Parameters:

circ [circuit instance] The circuit instance for which the D matrix is computed.

shape [tuple of ints] The shape of the *reduced* MNA matrix, D will be of the same shape.

Returns:

D [ndarray] The *unreduced* D matrix.

`import_custom_df_module` (*method*, *print_out*)
Imports a module that implements differentiation formula through imp.load_module Parameters: method: a string, the name of the df method module print_out: print to stdout some verbose messages

Returns: The df module or None if the module is not found.

`transient_analysis` (*circ*, *tstart*, *tstep*, *tstop*, *method=u'TRAP'*, *use_step_control=True*, *x0=None*, *mna=None*, *N=None*, *D=None*, *outfile=u'stdout'*, *return_req_dict=None*, *verbose=3*)
Performs a transient analysis of the circuit described by circ.

Parameters: circ: circuit instance to be simulated. tstart: start value. Better leave this to zero. tstep: the maximum step to be allowed during simulation or tstop: stop value for simulation method: differentiation method: 'TRAP' (default) or 'IMPLICIT_EULER' or 'GEARx' with x=1..6 use_step_control: the LTE will be calculated and the step adjusted. default: True x0: the starting point, the solution at t=tstart (defaults to None, will be set to the OP) mna, N, D: MNA matrices, defaulting to None, for big circuits, reusing matrices saves time outfile: filename, the results will be written to this file. "stdout" means print out. return_req_dict: to be documented verbose: verbosity level from 0 (silent) to 6 (very verbose).

4.30 ahkab.trap

This file implements the Trapezoidal (TRAP) Differentiation Formula (DF) and a second order prediction formula.

4.30.1 Module reference

`get_df` (*pv_array*, *suggested_step*, *predict=True*)
Get the coefficients for DF and prediction formula

Parameters:

pv_array [sequence of sequences] Each element of `pv_array` must be of the form:

```
(time, x, derivate(x))
```

In particular, the k element of *pv_array* contains the values of:

- t_{n-k} (the time),
- x_{n-k},
- dx_{n-k}/dt

evaluated k time steps before the current one, labeled $n+1$.

How many samples are necessary is given by `ahkab.trap.get_required_values()`.

Values that are not needed may be set to `None`, as they will be disregarded.

suggested_step [float] The step that will be used for the current iteration, provided the error will be deemed acceptable.

predict [bool, optional] Whether a prediction for x_n is needed as well or not. Defaults to `True`.

Returns:

ret [tuple] The return value has the form:

```
(C1, C0, x_lte_coeff, predict_x, predict_lte_coeff)
```

The derivative may be written as:

$$d(x(n+1))/dt = C1x(n+1) + C0$$

x_lte_coeff is the coefficient of the Local Truncation Error, *predict_x* is the predicted value for x and *predict_lte_coeff* is the LTE coefficient for the prediction.

Raises ValueError if the *pv_array* is malformed.

get_required_values()

Get info regarding what values are needed by the DF

Returns:

tpl [tuple of tuples] A tuple of two tuples.

- The first tuple indicates what past values of the unknown are needed for the DF.
- The second tuple indicates what past values of the unknown are needed for the prediction method.

In particular, each of the sub-tuples is built this way:

```
(max_order_of_x, max_order_of_dx)
```

Where both the values are either `int`, or `None`. If `max_order_of_x` is set to an integer value k, the DF needs all the x_{n-i} values of x, for all $0 \leq i \leq k$. In the previous text, x_{n-i} is the value the x array assumed i steps before the one we are considering for the derivative.

Similar considerations apply to `max_order_of_dx`, but regard rather dx_n/dt instead of x_n.

If any of the values is set to `None`, it is to be assume that no value is required.

The first array has to be used if no prediction is required, the second are the values needed for prediction.

has_ff()

Has the method a Forward Formula for prediction?

Returns:

doesit [bool] In this particular case, this function always returns `True`.

is_implicit()

Is this Differentiation Formula (DF) implicit?

Returns:

isit [boolean] In this case, that's `True`.

4.31 ahkab.utilities

This module holds miscellaneous utility functions.

4.31.1 Module reference

`Celsius2Kelvin`(*cel*)
Convert Celsius degrees to Kelvin

`EPS` = **2.2204460492503131e-16**
The machine epsilon, the upper bound on the relative error due to rounding in floating point arithmetic.

`Kelvin2Celsius`(*kel*)
Convert Kelvin degrees to Celsius

`check_circuit`(*circ*)
Performs some easy sanity checks.

Checks performed:

- Has the circuit more than one node?
- Has the circuit a connection to ground?
- Has the circuit more than two elements?
- Are there no two elements with the same `part_id`?

Parameters:

circ [circuit instance] The circuit to be checked.

Returns:

chk [boolean] The logical `and()` of the answer to the above questions.

msg [string] A message describing the error, if any.

`check_file`(*filename*)
Checks whether the supplied path refers to a valid file.

Parameters:

filename [string] The file name.

Returns:

chk [boolean] A value of `True` if `filename` is found and it is a file.

Raises IOError if no such file exists or if the supplied file is a directory.

`check_ground_paths`(*mna*, *circ*, *reduced_mna=True*, *verbose=3*)
Checks that every node has a DC path to ground

The path to ground might be through non-linear elements.

Note:

- This does not ensure that the circuit will have a DC solution.
- A node without DC path to ground would be rescued (likely) by GMIN so (for the time being at least) we do *not* halt the execution.
- Also, two series capacitors always fail this check (GMIN saves us)

Bottom line: if there is no DC path to ground, there is probably a mistake in the netlist. Print a warning.

Returns:

chk [boolean] A boolean set to true if there is a DC path to ground from all nodes in the circuit.

check_step_and_points (*step=None*, *points=None*, *period=None*, *default_points=100*)

Sets consistently the step size and the number of points

The calculation is done according to the given period.

Parameters:

step [scalar, optional] The discretization step.

points [int, optional] The number of points to be used in the discretization.

period [scalar, optional] The length of the interval to be discretized. Not setting this parameter will result in a `ValueError`.

default_points [int, optional] The default number of points.

Returns:

(points, step) [tuple] The adjusted number of points and step value.

class **combinations** (*L*, *k*)

This class is an iterator that returns all the k-combinations _without_repetition_ of the elements of the supplied list.

Each combination is made of a subset of the list, consisting of k elements.

Parameters:

L [list] The set from which the elements are taken.

k [int] The size of the subset, the number of elements to be taken

next ()

convergence_check (*x*, *dx*, *residuum*, *nv_minus_one*, *debug=False*)

Perform a convergence check

Parameters:

x [array-like] The results to be checked.

dx [array-like] The last increment from a Newton-Rhapson iteration, solving `F(x) = 0`.

residuum [array-like] The remaining error, ie `F(x) = residdum`

nv_minus_one [int] Number of voltage variables in x. If `nv_minus_one` is equal to `n`, it means `x[:n]` are all voltage variables.

debug [boolean, optional] Whether extra information is needed for debug purposes. Defaults to `False`.

Returns:

chk [boolean] Whether the check was passed or not. `True` means 'convergence!'.

rbn [ndarray] The convergence check results by node, if `debug` was set to `True`, else `None`.

`current_convergence_check`(*x*, *dx*, *residuum*, *debug=False*)
Perform a convergence check for current variables

Parameters:

x [array-like] The results to be checked.

dx [array-like] The last increment from a Newton-Rhapson iteration, solving `F(x) = 0`.

residuum [array-like] The remaining error, ie `F(x) = residdum`

debug [boolean, optional] Whether extra information is needed for debug purposes. Defaults to `False`.

Returns:

chk [boolean] Whether the check was passed or not. `True` means 'convergence!'.

rbn [ndarray] The convergence check results by node, if `debug` was set to `True`, else `None`.

`custom_convergence_check`(*x*, *dx*, *residuum*, *er*, *ea*, *eresiduum*, *debug=False*)
Perform a custom convergence check

Parameters:

x [array-like] The results to be checked.

dx [array-like] The last increment from a Newton-Rhapson iteration, solving `F(x) = 0`.

residuum [array-like] The remaining error, ie `F(x) = residdum`

ea [float] The value to be employed for the absolute error.

er [float] The value for the relative error to be employed.

eresiduum [float] The maximum allowed error for the residuum (left over error).

debug [boolean, optional] Whether extra information is needed for debug purposes. Defaults to `False`.

Returns:

chk [boolean] Whether the check was passed or not. `True` means 'convergence!'.

rbn [ndarray] The convergence check results by node, if `debug` was set to `True`, else `None`.

`expand_matrix`(*matrix*, *add_a_row=False*, *add_a_col=False*)
Append a row and/or a column to the given matrix

Parameters:

matrix [ndarray] The matrix to be manipulated.

add_a_row [boolean, optional] If set to `True` a row is appended to the supplied matrix.

add_a_col [boolean] If set to `True` a column is appended.

Returns:

matrix [ndarray] A reference to the same matrix supplied.

class `lin_axis_iterator`(*min*, *max*, *points*)
This iterator provides the values for a linear sweep.

Parameters:

min [float] The minimum value, also the start point of the axis.

max [float] The maximum value, also the end point of the axis.

num [int] The number of samples to generate. In general, this should be greater than 1. A value of 1 is accepted only if `min == max`, in which case, only one value is returned by the iterator: `min`.

Start and end points are always included.

Notice that, differently from numpy's `linspace()`, the values are only computed at access time, and hence the memory footprint of the iterator is low.

Raises ValueError if the number `points` is either negative or does not

respect the conditions above.

`next`()

class **`log_axis_iterator`**(*min*, *max*, *points*)

This iterator provides the values for a base-10 logarithmic sweep.

Parameters:

min [float] The minimum value, also the start point of the axis.

max [float] The maximum value, also the end point of the axis.

points [int] The number of points which will be used to discretize the `max` - `min` interval.

Notice that, differently from numpy's `logspace()`, the values are only computed at access time, and hence the memory footprint of the iterator is low.

Start and end values are always included.

`next`()

`memoize`(*f*)

Memoization decorator

Parameters:

f [function] The function to apply memoization to.

Returns:

fm [function] The function with added memoization.

Implementation:

Originally from this post, it has been modified to provide a cache of size `options.cache_len`.

Note: The size of the cache is per model instance and per function. If you have one model, shared by several elements, you probably prefer to have a big cache.

`remove_row`(*matrix*, *rrow=0*)

Removes a row from a matrix

Parameters:

matrix [ndarray] The matrix to be modified.

rrow [int or None, optional] The index of the row to be removed. If set to `None`, no row will be removed. By default the first row is removed.

Note: No size checking is done.

Returns:

matrix [ndarray] A reference to the modified matrix.

remove_row_and_col(*matrix*, *rrow=0*, *rcol=0*)
Removes a row and/or a column from a matrix

Parameters:

matrix [ndarray] The matrix to be modified.

rrow [int or None, optional] The index of the row to be removed. If set to `None`, no row will be removed. By default the first row is removed.

rcol [int or None, optional] The index of the row to be removed. If set to `None`, no row will be removed. By default the first column is removed.

Note: No size checking is done.

Returns:

matrix [ndarray] A reference to the modified matrix.

set_submatrix(*row*, *col*, *dest_matrix*, *source_matrix*)
Copies a source matrix into another matrix

row, col [ints] The coordinates of the upper left corner in the destination matrix where the source matrix will be copied.

dest_matrix [ndarray] The matrix to be copied to.

source_matrix [ndarray] The matrix to be copied from.

Returns:

dest_matrix [ndarray] A reference to the modified destination matrix.

tuplinator(*alist*)
Convert a list of lists (of lists...) to tuples

voltage_convergence_check(*x*, *dx*, *residuum*, *debug=False*)
Perform a convergence check for voltage variables

Parameters:

x [array-like] The results to be checked.

dx [array-like] The last increment from a Newton-Rhapson iteration, solving `F(x) = 0`.

residuum [array-like] The remaining error, ie `F(x) = residdum`

debug [boolean, optional] Whether extra information is needed for debug purposes. Defaults to `False`.

Returns:

chk [boolean] Whether the check was passed or not. `True` means 'convergence!'.

rbn [ndarray] The convergence check results by node, if `debug` was set to `True`, else `None`.

CHAPTER 5

License

5.1 GNU General Public License

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
```

```
You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

ahkab.ac, 77
ahkab.ahkab, 69
ahkab.bfpss, 79
ahkab.circuit, 80
ahkab.constants, 91
ahkab.csvlib, 92
ahkab.dc_analysis, 94
ahkab.dc_guess, 100
ahkab.devices, 101
ahkab.diode, 110
ahkab.ekv, 111
ahkab.fourier, 115
ahkab.gear, 118
ahkab.implicit_euler, 119
ahkab.mosq, 121
ahkab.netlist_parser, 126
ahkab.options, 134
ahkab.plotting, 137
ahkab.printing, 138
ahkab.pss, 142
ahkab.pz, 142
ahkab.results, 144
ahkab.shooting, 152
ahkab.switch, 153
ahkab.symbolic, 156
ahkab.testing, 158
ahkab.ticker, 164
ahkab.time_functions, 165
ahkab.transient, 170
ahkab.trap, 173
ahkab.utilities, 174

Index

A

ac_analysis() (in module ahkab.ac), 79
ac_max_nr_iter (in module ahkab.options), 134
ac_phase_in_deg (in module ahkab.options), 134
ac_solution (class in ahkab.results), 145
add() (dfbuffer method), 171
add_capacitor() (Circuit method), 82
add_cccs() (Circuit method), 82
add_ccvs() (Circuit method), 83
add_diode() (Circuit method), 83
add_inductor() (Circuit method), 83
add_inductor_coupling() (Circuit method), 84
add_isource() (Circuit method), 84
add_line() (ac_solution method), 145
add_line() (tran_solution method), 151
add_model() (Circuit method), 84
add_mos() (Circuit method), 84
add_node() (Circuit method), 85
add_op() (dc_solution method), 146
add_resistor() (Circuit method), 85
add_switch() (Circuit method), 86
add_user_defined() (Circuit method), 86
add_vccs() (Circuit method), 86
add_vcvs() (Circuit method), 87
add_vsource() (Circuit method), 87
ahkab.ac (module), 77
ahkab.ahkab (module), 69
ahkab.bfpss (module), 79
ahkab.circuit (module), 80
ahkab.constants (module), 91
ahkab.csvlib (module), 92
ahkab.dc_analysis (module), 94
ahkab.dc_guess (module), 100
ahkab.devices (module), 101
ahkab.diode (module), 110
ahkab.ekv (module), 111
ahkab.fourier (module), 115
ahkab.gear (module), 118
ahkab.implicit_euler (module), 119
ahkab.mosq (module), 121
ahkab.netlist_parser (module), 126
ahkab.options (module), 134
ahkab.plotting (module), 137
ahkab.printing (module), 138
ahkab.pss (module), 142
ahkab.pz (module), 142
ahkab.results (module), 144
ahkab.shooting (module), 152
ahkab.switch (module), 153
ahkab.symbolic (module), 156
ahkab.testing (module), 158
ahkab.ticker (module), 164
ahkab.time_functions (module), 165
ahkab.transient (module), 170
ahkab.trap (module), 173
ahkab.utilities (module), 174
am (class in ahkab.time_functions), 167
APITest (class in ahkab.testing), 164
as_symbol() (symbolic_solution method), 150
as_symbols() (symbolic_solution method), 150
asarray() (ac_solution method), 145
asarray() (dc_solution method), 146
asarray() (op_solution method), 147
asarray() (pss_solution method), 148
asarray() (pz_solution method), 149
asarray() (solution method), 149
asarray() (tran_solution method), 151

B

bfpss() (in module ahkab.bfpss), 79
bfpss_default_points (in module ahkab.options), 134
bfpss_max_nr_iter (in module ahkab.options), 134
build_gmin_matrix() (in module ahkab.dc_analysis), 94
build_x0_from_user_supplied_ic() (in module ahkab.dc_analysis), 94

C

cache_len (in module ahkab.options), 134
calculate_gains() (in module ahkab.symbolic), 156
calculate_poles() (in module ahkab.pz), 143
calculate_singularities() (in module ahkab.pz), 143
Capacitor (class in ahkab.devices), 103
case_insensitive_dict (class in ahkab.results), 145
Celsius2Kelvin() (in module ahkab.utilities), 175
check_circuit() (in module ahkab.utilities), 175
check_file() (in module ahkab.utilities), 175
check_ground_paths() (in module ahkab.utilities), 175
check_step() (in module ahkab.transient), 171
check_step_and_points() (in module ahkab.utilities), 176
Circuit (class in ahkab.circuit), 82
CircuitError, 90
cli (in module ahkab.options), 134
cmin (in module ahkab.options), 135
combinations (class in ahkab.utilities), 176
Component (class in ahkab.devices), 103
convergence_check() (in module ahkab.utilities), 176
convert() (in module ahkab.netlist_parser), 127
convert_boolean() (in module ahkab.netlist_parser), 128
convert_units() (in module ahkab.netlist_parser), 128
create_node() (Circuit method), 87
current_convergence_check() (in module ahkab.utilities), 176
custom_convergence_check() (in module ahkab.utilities), 177

D

d() (Capacitor method), 103
dc_analysis() (in module ahkab.dc_analysis), 95
dc_max_guess_effort (in module ahkab.options), 135
dc_max_nr_iter (in module ahkab.options), 135
dc_solution (class in ahkab.results), 146
dc_solve() (in module ahkab.dc_analysis), 95
dc_sweep_skip_allowed (in module ahkab.options), 135
dc_use_guess (in module ahkab.options), 135
default_tran_method (in module ahkab.options), 135
dense_matrix_limit (in module ahkab.options), 135
device_check() (mosq_mos_model method), 124
dfbuffer (class in ahkab.transient), 171
diode (class in ahkab.diode), 110
diode_model (class in ahkab.diode), 111
display() (ticker method), 165

E

e (in module ahkab.constants), 91
Eg() (silicon method), 91
ekv_device (class in ahkab.ekv), 112
ekv_mos_model (class in ahkab.ekv), 113
encoding (in module ahkab.options), 135
EPS (in module ahkab.utilities), 175
EVSource (class in ahkab.devices), 103
exp (class in ahkab.time_functions), 167
expand_matrix() (in module ahkab.utilities), 177
ext_node_to_int() (Circuit method), 88

F

find_vde() (Circuit method), 88
find_vde_index() (Circuit method), 88
FISource (class in ahkab.devices), 104
fourier() (in module ahkab.fourier), 115

G

g (Resistor attribute), 108
g() (Capacitor method), 103
g() (Component method), 103
g() (diode method), 110
g() (ekv_device method), 113
g() (switch_device method), 154
generate_D() (in module ahkab.transient), 172
generate_mna_and_N() (in module ahkab.dc_analysis), 96
generate_mna_and_N() (in module ahkab.symbolic), 157
get() (ac_solution method), 145
get() (case_insensitive_dict method), 146
get() (dc_solution method), 146
get() (op_solution method), 147
get() (pss_solution method), 148
get() (pz_solution method), 149
get() (solution method), 149
get() (symbolic_solution method), 150
get() (tran_solution method), 151
get_as_matrix() (dfbuffer method), 172
get_dc_guess() (in module ahkab.dc_guess), 100
get_dc_guess() (vswitch_model method), 156
get_device_temperature() (ekv_mos_model method), 114
get_device_temperature() (mosq_mos_model method), 124
get_df() (in module ahkab.gear), 118
get_df() (in module ahkab.implicit_euler), 119

get_df() (in module ahkab.trap), 173
get_df_coeff() (in module ahkab.implicit_euler), 120
get_df_vector() (dfbuffer method), 172
get_drive_ports() (diode method), 110
get_drive_ports() (ekv_device method), 113
get_drive_ports() (mosq_device method), 122
get_drive_ports() (switch_device method), 154
get_dvsmall_dismall() (ekv_mos_model method), 114
get_elem_by_name() (Circuit method), 88
get_gm() (diode_model method), 111
get_gm() (mosq_mos_model method), 124
get_gm() (vswitch_model method), 156
get_gmb() (mosq_mos_model method), 125
get_gmd() (ekv_mos_model method), 114
get_gmd() (mosq_mos_model method), 125
get_gmg() (ekv_mos_model method), 114
get_gms() (ekv_mos_model method), 114
get_go() (vswitch_model method), 156
get_ground_node() (Circuit method), 89
get_headers() (in module ahkab.csvlib), 92
get_headers_index() (in module ahkab.csvlib), 92
get_i() (diode_model method), 111
get_i() (vswitch_model method), 156
get_ids() (ekv_mos_model method), 114
get_ids() (mosq_mos_model method), 125
get_ip_abs_err() (ekv_mos_model method), 114
get_ismall() (ekv_mos_model method), 114
get_leq_virp() (ekv_mos_model method), 114
get_locked_nodes() (Circuit method), 89
get_mc_requirements() (mosq_device method), 122
get_netlist_elem_line() (Component method), 103
get_netlist_elem_line() (diode method), 110
get_netlist_elem_line() (ekv_device method), 113
get_netlist_elem_line() (EVSource method), 104
get_netlist_elem_line() (FISource method), 105
get_netlist_elem_line() (GISource method), 105
get_netlist_elem_line() (HVSource method), 106
get_netlist_elem_line() (InductorCoupling method), 108
get_netlist_elem_line() (ISource method), 107
get_netlist_elem_line() (mosq_device method), 123
get_netlist_elem_line() (switch_device method), 154
get_netlist_elem_line() (VSource method), 109
get_next_file_and_close_current() (in module ahkab.netlist_parser), 128
get_nodes_number() (Circuit method), 89
get_op_info() (Capacitor method), 103
get_op_info() (diode method), 110
get_op_info() (ekv_device method), 113
get_op_info() (Inductor method), 107
get_op_info() (ISource method), 107
get_op_info() (mosq_device method), 123
get_op_info() (Resistor method), 108
get_op_info() (switch_device method), 155
get_op_info() (VSource method), 109
get_op_x0() (in module ahkab.ahkab), 71
get_other_inductor() (InductorCoupling method), 108
get_output_ports() (diode method), 110
get_output_ports() (ekv_device method), 113
get_output_ports() (mosq_device method), 123
get_output_ports() (switch_device method), 155
get_required_values() (in module ahkab.gear), 119
get_required_values() (in module ahkab.implicit_euler), 120
get_required_values() (in module ahkab.trap), 174
get_roots() (in module ahkab.symbolic), 158
get_solve_methods() (in module ahkab.dc_analysis), 97
get_svt_skp() (mosq_mos_model method), 125
get_table_array() (op_solution method), 147
get_td() (in module ahkab.dc_analysis), 97
get_value_function() (ekv_device method), 113
get_value_function() (mosq_device method), 123
get_value_function() (switch_device method), 155
get_variables() (in module ahkab.symbolic), 158
get_voltages() (ekv_mos_model method), 114
get_voltages() (mosq_mos_model method), 125
get_vp_nv_nq() (ekv_mos_model method), 114
get_vsmall() (ekv_mos_model method), 114
get_VT() (mosq_mos_model method), 124
get_x() (ac_solution method), 145
get_x() (dc_solution method), 146
get_x() (pss_solution method), 148
get_x() (tran_solution method), 152
get_xlabel() (ac_solution method), 145
get_xlabel() (dc_solution method), 146
get_xlabel() (pss_solution method), 148
get_xlabel() (tran_solution method), 152
GISource (class in ahkab.devices), 105
gmin (in module ahkab.options), 135
gmin_check() (op_solution static method), 147
gstamp() (diode method), 110
gstamp() (mosq_device method), 123

H

has_duplicate_elem() (Circuit method), 89

has_ff() (in module ahkab.gear), 119
has_ff() (in module ahkab.implicit_euler), 121
has_ff() (in module ahkab.trap), 174
has_key() (ac_solution method), 145
has_key() (case_insensitive_dict method), 146
has_key() (dc_solution method), 146
has_key() (op_solution method), 147
has_key() (pss_solution method), 148
has_key() (pz_solution method), 149
has_key() (solution method), 149
has_key() (symbolic_solution method), 150
has_key() (tran_solution method), 152
hide() (ticker method), 165
hmin (in module ahkab.options), 135
HVSource (class in ahkab.devices), 105

I

i() (Capacitor method), 103
i() (Component method), 103
i() (diode method), 111
i() (ekv_device method), 113
I() (ISource method), 106
i() (Resistor method), 109
i() (switch_device method), 155
icmodified_x0() (in module ahkab.ahkab), 71
iea (in module ahkab.options), 135
ier (in module ahkab.options), 135
import_custom_df_module() (in module ahkab.transient), 173
Inductor (class in ahkab.devices), 107
InductorCoupling (class in ahkab.devices), 107
INIT_IFRN_GUESS (ekv_device attribute), 112
int_node_to_ext() (Circuit method), 89
is_elem_voltage_defined() (in module ahkab.circuit), 90
is_implicit() (in module ahkab.gear), 119
is_implicit() (in module ahkab.implicit_euler), 121
is_implicit() (in module ahkab.trap), 174
is_int_node_internal_only() (Circuit method), 89
is_nonlinear() (Circuit method), 89
is_valid_value_param_string() (in module ahkab.netlist_parser), 128
ismall2qsmall() (ekv_mos_model method), 114
ISource (class in ahkab.devices), 106
isready() (dfbuffer method), 172
istamp() (diode method), 111
istamp() (mosq_device method), 124
items() (ac_solution method), 145
items() (case_insensitive_dict method), 146
items() (dc_solution method), 146
items() (op_solution method), 147
items() (pss_solution method), 148
items() (pz_solution method), 149
items() (solution method), 149
items() (symbolic_solution method), 150
items() (tran_solution method), 152

J

join_lines() (in module ahkab.netlist_parser), 128

K

k (in module ahkab.constants), 91
Kelvin2Celsius() (in module ahkab.utilities), 175
keys() (ac_solution method), 145
keys() (case_insensitive_dict method), 146
keys() (dc_solution method), 147
keys() (op_solution method), 147
keys() (pss_solution method), 148
keys() (pz_solution method), 149
keys() (solution method), 149
keys() (symbolic_solution method), 151
keys() (tran_solution method), 152

L

lin_axis_iterator (class in ahkab.utilities), 177
load() (symbolic_solution static method), 151
load_csv() (in module ahkab.csvlib), 93
lock() (tran_solution method), 152
log_axis_iterator (class in ahkab.utilities), 178

M

main() (in module ahkab.ahkab), 71
main_netlist_parser() (in module ahkab.netlist_parser), 128
mdn_solver() (in module ahkab.dc_analysis), 98
memoize() (in module ahkab.utilities), 178
ModelError, 90
modify_x0_for_ic() (in module ahkab.dc_analysis), 99
more_solve_methods_available() (in module ahkab.dc_analysis), 99
mosq_device (class in ahkab.mosq), 122
mosq_mos_model (class in ahkab.mosq), 124

N

NetlistParseError, 127
NetlistTest (class in ahkab.testing), 164
new_ac() (in module ahkab.ahkab), 71
new_dc() (in module ahkab.ahkab), 72
new_internal_node() (Circuit method), 89
new_op() (in module ahkab.ahkab), 73
new_pss() (in module ahkab.ahkab), 73
new_pz() (in module ahkab.ahkab), 74

new_symbolic() (in module ahkab.ahkab), 74
new_tran() (in module ahkab.ahkab), 75
new_x0() (in module ahkab.ahkab), 76
next() (ac_solution method), 145
next() (combinations method), 176
next() (dc_solution method), 147
next() (lin_axis_iterator method), 178
next() (log_axis_iterator method), 178
next() (op_solution method), 147
next() (pss_solution method), 148
next() (pz_solution method), 149
next() (solution method), 149
next() (symbolic_solution method), 151
next() (tran_solution method), 152
ni() (silicon method), 92
nl_voltages_lock (in module ahkab.options), 135
nl_voltages_lock_factor (in module ahkab.options), 135
NodeNotFoundError, 90
nr_damp_first_iters (in module ahkab.options), 135

O

ok() (in module ahkab.testing), 164
op_analysis() (in module ahkab.dc_analysis), 99
op_solution (class in ahkab.results), 147
open_utf8() (in module ahkab.printing), 139
order (in module ahkab.implicit_euler), 121

P

parse_analysis() (in module ahkab.netlist_parser), 128
parse_circuit() (in module ahkab.netlist_parser), 129
parse_elem_capacitor() (in module ahkab.netlist_parser), 129
parse_elem_cccs() (in module ahkab.netlist_parser), 129
parse_elem_ccvs() (in module ahkab.netlist_parser), 129
parse_elem_diode() (in module ahkab.netlist_parser), 130
parse_elem_inductor() (in module ahkab.netlist_parser), 130
parse_elem_inductor_coupling() (in module ahkab.netlist_parser), 130
parse_elem_isource() (in module ahkab.netlist_parser), 130
parse_elem_mos() (in module ahkab.netlist_parser), 130
parse_elem_resistor() (in module ahkab.netlist_parser), 131
parse_elem_switch() (in module ahkab.netlist_parser), 131
parse_elem_user_defined() (in module ahkab.netlist_parser), 131
parse_elem_vccs() (in module ahkab.netlist_parser), 132
parse_elem_vcvs() (in module ahkab.netlist_parser), 132
parse_elem_vsource() (in module ahkab.netlist_parser), 132
parse_ic_directive() (in module ahkab.netlist_parser), 132
parse_ics() (in module ahkab.netlist_parser), 133
parse_include_directive() (in module ahkab.netlist_parser), 133
parse_models() (in module ahkab.netlist_parser), 133
parse_param_value_from_string() (in module ahkab.netlist_parser), 133
parse_postproc() (in module ahkab.netlist_parser), 133
parse_single_analysis() (in module ahkab.netlist_parser), 133
parse_sub_declaration() (in module ahkab.netlist_parser), 133
parse_sub_instance() (in module ahkab.netlist_parser), 133
parse_temp_directive() (in module ahkab.netlist_parser), 134
parse_time_function() (in module ahkab.netlist_parser), 134
plot_results() (in module ahkab.plotting), 137
plotting_display_figsize (in module ahkab.options), 135
plotting_lw (in module ahkab.options), 135
plotting_outtype (in module ahkab.options), 135
plotting_save_figsize (in module ahkab.options), 136
plotting_show_plots (in module ahkab.options), 136
plotting_style (in module ahkab.options), 136
plotting_wait_after_plot (in module ahkab.options), 136
print_analysis() (in module ahkab.printing), 139
print_fourier() (in module ahkab.printing), 139
print_general_error() (in module ahkab.printing), 139
print_info_line() (in module ahkab.printing), 139
print_int_nodes (in module ahkab.options), 136
print_model() (diode_model method), 111
print_model() (ekv_mos_model method), 114

print_model() (mosq_mos_model method), 126
print_model() (vswitch_model method), 156
print_parse_error() (in module ahkab.printing), 140
print_precision (in module ahkab.options), 136
print_result_check() (in module ahkab.printing), 140
print_short() (op_solution method), 147
print_spicefft() (in module ahkab.printing), 140
print_suppress (in module ahkab.options), 136
print_symbolic_equations() (in module ahkab.printing), 141
print_symbolic_results() (in module ahkab.printing), 141
print_symbolic_transfer_functions() (in module ahkab.printing), 141
print_warning() (in module ahkab.printing), 141
printoptions() (in module ahkab.printing), 141
process_postproc() (in module ahkab.ahkab), 77
pss_analysis() (in module ahkab.pss), 142
pss_solution (class in ahkab.results), 147
pulse (class in ahkab.time_functions), 168
pwl (class in ahkab.time_functions), 168
pz_max (in module ahkab.options), 136
pz_solution (class in ahkab.results), 148

Q

qsmall2ismall() (ekv_mos_model method), 115
queue() (in module ahkab.ahkab), 77

R

remove_elem() (Circuit method), 90
remove_model() (Circuit method), 90
remove_row() (in module ahkab.utilities), 178
remove_row_and_col() (in module ahkab.utilities), 179
reset() (ticker method), 165
Resistor (class in ahkab.devices), 108
run() (in module ahkab.ahkab), 77

S

s (in module ahkab.symbolic), 158
save() (symbolic_solution method), 151
save_figure() (in module ahkab.plotting), 137
scaling_holder (class in ahkab.ekv), 115
set_char() (Component method), 103
set_device_temperature() (ekv_mos_model method), 115
set_device_temperature() (mosq_mos_model method), 126
set_next_solve_method() (in module ahkab.dc_analysis), 100
set_results() (pss_solution method), 148
set_submatrix() (in module ahkab.utilities), 179
set_temperature() (diode method), 111
set_temperature() (diode_model method), 111
set_temperature() (in module ahkab.ahkab), 77
setUp() (APITest method), 164
setUp() (NetlistTest method), 164
setup_mc() (mosq_device method), 124
setup_scaling() (ekv_mos_model method), 115
sffm (class in ahkab.time_functions), 169
shooting_analysis() (in module ahkab.shooting), 152
shooting_default_points (in module ahkab.options), 136
shooting_max_nr_iter (in module ahkab.options), 136
show_plots() (in module ahkab.plotting), 138
si (in module ahkab.constants), 91
silicon (class in ahkab.constants), 91
sin (class in ahkab.time_functions), 170
solution (class in ahkab.results), 149
spicefft() (in module ahkab.fourier), 116
step() (ticker method), 165
subckt (class in ahkab.circuit), 90
switch_device (class in ahkab.switch), 153
symb_formulate_with_gs (in module ahkab.options), 136
symb_sympy_manual_solver (in module ahkab.options), 136
symbolic_analysis() (in module ahkab.symbolic), 158
symbolic_solution (class in ahkab.results), 150

T

T (in module ahkab.constants), 91
table() (in module ahkab.printing), 141
tearDown() (APITest method), 164
tearDown() (NetlistTest method), 164
ticker (class in ahkab.ticker), 164
tran_solution (class in ahkab.results), 151
transient_analysis() (in module ahkab.transient), 173
transient_aposteriori_step_threshold (in module ahkab.options), 136
transient_max_nr_iter (in module ahkab.options), 136
transient_max_time_iter (in module ahkab.options), 136
transient_no_step_control (in module ahkab.options), 136

transient_prediction_as_x0 (in module ahkab.options), 136
transient_use_aposteriori_step_control (in module ahkab.options), 137
Tref (in module ahkab.constants), 91
tuplinator() (in module ahkab.utilities), 179

U

update() (case_insensitive_dict method), 146
update_status_dictionary() (ekv_device method), 113
update_status_dictionary() (mosq_device method), 124
update_status_dictionary() (switch_device method), 155
use_gmin_stepping (in module ahkab.options), 137
use_source_stepping (in module ahkab.options), 137
use_standard_solve_method (in module ahkab.options), 137

V

V() (VSource method), 109
value (Resistor attribute), 109
values() (ac_solution method), 145
values() (case_insensitive_dict method), 146
values() (dc_solution method), 147
values() (op_solution method), 147
values() (pss_solution method), 148
values() (pz_solution method), 149
values() (solution method), 149
values() (symbolic_solution method), 151
values() (tran_solution method), 152
vea (in module ahkab.options), 137
ver (in module ahkab.options), 137
voltage_convergence_check() (in module ahkab.utilities), 179
VSource (class in ahkab.devices), 109
vswitch_model (class in ahkab.switch), 155
Vth() (in module ahkab.constants), 91

W

write_csv() (in module ahkab.csvlib), 93
write_headers() (in module ahkab.csvlib), 93
write_to_file() (op_solution method), 147