PyLTSpice

Release 3.0

Nuno Brum

Oct 11, 2023

CONTENTS:

CHAPTER

ONE

PYTHON MODULES

PyLTSpice contains several modules for working with netlists, simulations and simulation data.

1.1 Reading and Manipulating Netlists

PyLTSpice has the ability to read and manipulate netlist files using the *SpiceEditor* class.

The SpiceEditor class inherits almost all functions from *SpiceCircuit* class. The SpiceCircuit class allows manipulation of circuit elements such as resistors, capacitors, transistors and so on, as well as subcircuit parameters.

The SpiceEditor extends these funcionalities and adds all the functions needed to read and write netlists to the disk, as well as defining top level simulation directives, such as .TRAN and .AC for example.

The rationale for this division is to allow to manipulate not only elements that exist at the top level, but also to manipulate elements that exit inside of sub-circuits. In Example 2 there is small example where the value of a component inside a subcircuit is changed.

When all the changes are made, the write_netlist(<filename>) needs to be called in order for the updates to be registered.

Do not update the original file. It is always safer to keep the original file unchanged. This helps when debuging problems, and also allows the script to revert to the initial condition by using the reset_netlist() method.

Example 1: Setting parameters inside a flat netlist.

```
#read netlist
import PyLTSpice
net = PyLTSpice.SpiceEditor("Batch_Test.net")  # Loading the Netlist

net.set_parameters(res=0, cap=100e-6)  # Updating parameters res and cap
net.set_component_value('R2', '2k')    # Updating the value of R2 to 2k
net.set_component_value('R1', 4000)    # Updating the value of R1 to 4k
net.set_element_model('V3', "SINE(0 1 3k 0 0 0)")  # changing the behaviour of V3

# add instructions
net.add_instructions(
    "; Simulation settings",
    ".param run = 0"
)

net.write_netlist("Batch_Test_Modified.net")  # writes the modified netlist to the␣
↪indicated file
```

Example 2: Updating components inside a subcircuit

```
#read netlist
import PyLTSpice
net = PyLTSpice.SpiceEditor("Example2.net")

net.set_component_value('R1', 1000)      # Sets the value of R1 to 1k
net.set_component_value('XU1:Ra', '1k')  # Sets the value of Ra inside of XU1 to 1k
net.write_netlist("Batch_Test_Modified.net")
```

See the class documentation for more details :

- `PyLTSpice.SpiceEditor`
- `PyLTSpice.SpiceCircuit`

1.2 Running Simulations

The class `PyLTSpice.SimRunner` allows launching LTSpice simulations from a Python script, thus allowing to overcome the 3 dimensions STEP limitation on LTSpice, update resistor values, or component models. It also allows to simulate several simulations in parallel, thus speeding up the time a set of simulations would take.

The class `PyLTSpice.SpiceEditor` described in *Reading and Manipulating Netlists* allows to modify the netlist.

The code snipped below will simulate a circuit with two different diode models, set the simulation temperature to 80 degrees, and update the values of R1 and R2 to 3.3k.

```
from PyLTSpice import SimRunner, SpiceEditor, LTspice

runner = SimRunner(output_folder='./temp_batch3', simulator=LTspice)  # Configures the␣
↪simulator to use and output
# folder

netlist = SpiceEditor("Batch_Test.asc")  # Open the Spice Model, and creates the .net
# set default arguments
netlist.set_parameters(res=0, cap=100e-6)
netlist.set_component_value('R2', '2k')  # Modifying the value of a resistor
netlist.set_component_value('R1', '4k')
netlist.set_element_model('V3', "SINE(0 1 3k 0 0 0)")  # Modifying the
netlist.set_component_value('XU1:C2', 20e-12)  # modifying a
# define simulation
netlist.add_instructions(
        "; Simulation settings",
        ".param run = 0"
)

for opamp in ('AD712', 'AD820'):
    netlist.set_element_model('XU1', opamp)
    for supply_voltage in (5, 10, 15):
        netlist.set_component_value('V1', supply_voltage)
        netlist.set_component_value('V2', -supply_voltage)
        # overriding he automatic netlist naming
        run_netlist_file = "{}_{}_{}.net".format(netlist.netlist_file.name, opamp,␣
↪supply_voltage)
```

(continues on next page)

(continued from previous page)

```
        raw, log = runner.run_now(netlist, run_filename=run_netlist_file)
        # Process here the simulation results
```

In this example we are are using the SpiceEditor instantiation 'netlist' by passing an .asc file. When receiving an .asc file, it will use LTSpice to create the corresponding .net file and read it into memory.

Follows a series of function calls to 'netlist' that update the netlist in memory. The method `set_parameters(<param_name>, <param_value>)` updates the values of `.PARAM definitions`, the method `set_component_value(<element_id>, <element_value>)` will update values of R, L and C elements, in this example we are updating only resistors. The method `set_element_model()` sets the values of a voltage source 'V3' and so on.

Then we pass the object to the `runner.run_now()` method. By doing so, it will first write the netlist to the path indicated by the output folder indicated as parameter in the SimRunner instantiation, then will launch LTSpice to execute the simulation.

Although this works well, it is not very efficient on the processor usage. A simulation is ended before another one is started. An alternative method to this is presented in the next section.

1.2.1 Multiprocessing

For making better use of today's computer capabilities, the SimRunner can spawn several LTSpice instances each executing in parallel a simulation. This is exemplified in the modified example below.

```
from PyLTSpice import SimRunner, SpiceEditor, LTspice

def processing_data(raw_file, log_file):
    """This is the function that will process the data from simulations"""
    print("Handling the simulation data of %s, log file %s" % (raw_file, log_file))

# Configures the simulator to use and output folder. Also defines the number of parallel␣
↪simulations
runner = SimRunner(output_folder='./temp_batch3', simulator=LTspice, parallel_sims=4)

netlist = SpiceEditor("Batch_Test.asc")  # Open the Spice Model, and creates the .net
# set default arguments
netlist.set_parameters(res=0, cap=100e-6)
netlist.set_component_value('R2', '2k')  # Modifying the value of a resistor
netlist.set_component_value('R1', '4k')
netlist.set_element_model('V3', "SINE(0 1 3k 0 0 0)")  # Modifying the
netlist.set_component_value('XU1:C2', 20e-12)  # modifying a
# define simulation
netlist.add_instructions(
        "; Simulation settings",
        ".param run = 0"
)

for opamp in ('AD712', 'AD820'):
    netlist.set_element_model('XU1', opamp)
    for supply_voltage in (5, 10, 15):
        netlist.set_component_value('V1', supply_voltage)
        netlist.set_component_value('V2', -supply_voltage)
        # overriding he automatic netlist naming
```

(continues on next page)

(continued from previous page)

```
        run_netlist_file = "{}_{}_{}.net".format(netlist.netlist_file.name, opamp,␣
↪supply_voltage)
        # This will launch up to 'parallel_sims' simulations in background before waiting␣
↪for resources
        runner.run(netlist, run_filename=run_netlist_file, callback=processing_data)

# This will wait for the all the simulations launched before to complete.
runner.wait_completion()
# The timeout counter is reset everytime a simulation is finished.

# Sim Statistics
print('Successful/Total Simulations: ' + str(runner.okSim) + '/' + str(runner.runno))
```

If the `parallel_sims` parallel simulations is not given, it defaults to 4. This means that a fifth simulation will only start when one of the other 4 is finished. If `parallel_sims` needs to be adjusted according to the computer capabilities. If resources are abundant, this number can be set to a higher number. If set for example to 16, it means that the 17th simulation will wait for another one to finish before starting. Another way of bypassing this behaviour is just by setting the parameter `wait_resource=False` to False

```
runner.run(netlist, wait_resource=False)
```

Finally we see in the example the `runner.wait_completion()` method. This method will wait for the completion of all the pending jobs. The usage of `wait_completion()` is recommended if the further steps on the script require that all the simulations are done.

An alternative to `wait_completion` is to use an iterator as exemplified here:

```
runner = SimRunner(output_folder='./temp_batch3', simulator=LTspice)  # Configures the␣
↪simulator to use and output
# folder

netlist = SpiceEditor("Batch_Test.asc")  # Open the Spice Model, and creates the .net

for opamp in ('AD712', 'AD820'):
    netlist.set_element_model('XU1', opamp)
    for supply_voltage in (5, 10, 15):
        netlist.set_component_value('V1', supply_voltage)
        netlist.set_component_value('V2', -supply_voltage)
        runner.run(netlist, run_filename=run_netlist_file)

# runner.wait_completion()
for raw_file, log_file in runner:
    if raw_file:
        # process the raw file information
        print("Processed the raw file in the main thread")

print(f'Successful/Total Simulations: {runner.okSim} /{runner.runno}')
```

1.2.2 Callbacks

The methods above are alright for tasks that don't require much computational effort, or there is a small risk that the the processing fails. If this is not the case, then executing the processing of simulation results on the background thread may make sense. This not only speeds up the process, but, it also avoids crashing the program, when a simulation among hundreds fails for some reason.

For this purpose, the user can define a call back function and pass it to the `run()` function using the callback parameter. The callback function is called when the simulation has finished directly by the thread that has handling the simulation. A function callback receives two arguments. The RAW file and the LOG file names. Below is an example of a callback function.

```
def processing_data(raw_filename, log_filename):
    '''This is a call back function that just prints the filenames'''
    print("Simulation Raw file is %s. The log is %s" % (raw_filename, log_filename)
    # Other code below either using LTSteps.py or raw_read.py
    log_info = LTSpiceLogReader(log_filename)
    log_info.read_measures()
    rise, measures = log_info.dataset["rise_time"]
    return rise, measures
```

Callback functions can be either passed directly to the run function, and they are called once the simulation is finished.

There are two ways of passing a callback function depending on whether we want it to be executed as a Thread or as a Process. The key differences is that Threads are executed on the same memory space and therefore on the same core. Processes are executed in completely different memory spaces and different processor resources. Processes are slower to start, so, it's usage is only justified when parsing simulation results is really costly.

The callback functions are optional. As seen in the previous sections, if no callback function is given, the thread is terminated just after the simulation is finished.

Threads

In order to use threads, it suffices to include the name of the function with the named parameter `callback`.

```
for supply_voltage in (5, 10, 15):
    netlist.set_component_value('V1', supply_voltage)
    netlist.set_component_value('V2', -supply_voltage)
    runner.run(netlist, callback=processing_data)

for rise, measures in runner:
    print("The return of the callback function is ", rise, measures)
```

Processes

In order to use processes, the callback function needs to be encapsulated as a static method in a subclass of the special class called `ProcessCallback` and very importantly, all the code used to prepare and launch the simulation should be inside a `if __name__ == "__main__":` clause.

The reason for this is that since the module is going to be imported two times, first by the python.exe __main__ function and multiple times after by python processes searching for ProcessCallback subclass. The equivalent of the previous code using processes looks like this.

```python
from PyLTSpice.sim.process_callback import ProcessCallback  # Importing the␣
↪ProcessCallback class type

class CallbackProc(ProcessCallback):
    """Class encapsulating the callback function. It can have whatever name."""

    @staticmethod  # This decorator defines the callback as a static method, i.e., it␣
↪doesn't receive the `self`.
    def callback(raw_file, log_file):  # This function must be called callback
        '''This is a call back function that just prints the filenames'''
        print("Simulation Raw file is %s. The log is %s" % (raw_filename, log_filename)
        # Other code below either using LTSteps.py or raw_read.py
        log_info = LTSpiceLogReader(log_filename)
        log_info.read_measures()
        rise, measures = log_info.dataset["rise_time"]
        return rise, measures

if __name__ == "__main__":  # The code below must be only executed once.
                            # Without this clause, it doesn't work. Don't forget to␣
↪indent ALL the code below
    runner = SimRunner(output_folder='./temp', simulator=LTspice)  # Configures the␣
↪output folder and simulator
    for supply_voltage in (5, 10, 15):
        netlist.set_component_value('V1', supply_voltage)
        netlist.set_component_value('V2', -supply_voltage)
        runner.run(netlist, callback=CallbackProc)

    for rise, measures in runner:
        print("The return of the callback function is ", rise, measures)
```

The ProcessCallback class which is imported from PyLTSpice.sim.process_callback already defines the __init__ function and creates all the environment for the calling and callback function, and creates the Queue used to pipe the result back to the main process.

1.2.3 Processing of simulation outputs

The previous sections described the way to launch simulations. The way to parse the simulation results contained in the RAW files are described in *Reading Raw Files*. For parsing information contained in the LOG files, which contain information about measurements done with .MEAS primitives, is implemented by the class `PyLTSpice.SpiceEditor`

1.3 Reading Raw Files

The RawRead class is used to (surprise…) read .RAW-files. The file to read is given as parameter when creating the RawRead object. The object wil read the file and construct a structure of objects which can be used to access the data inside the .RAW-file. All traces on the .RAW-file are uploaded into memory.

See *RAW File Structure* for details of the contents of a .RAW-file.

The .RAW-file contains different traces for voltages and currents in the simulation.

Typically (for a transient analysis), a trace contain a value (voltage/current) for each different time point in the simulation. There is a list of time values, and a separate list of trace values for each trace. So each trace uses the same time values. If there were different steps (e.g. for a DC sweep), then there is a set of lists with time/value data for each step.

Note that for an AC analysis, the traces are *frequency-versus-value* instead of *time-versus-value*. We will use 'time' as an example further in this text.

The RawRead class has all the methods that allow the user to access the X-axis and trace values. If there is any stepped data (.STEP primitives), the RawRead class will try to load the log information from the same directory as the raw file in order to obtain the STEP information.

You can get a list of all trace names using the `get_trace_names()` method.

Use method `get_trace()` to get the trace data, which consists of values for 1 or more simulation steps. It will return a `PyLTSpice.raw_classes.Trace` object. Use this object's `get_wave()` method to get the actual data points for a step.

Use the method `get_axis()` to get the 'time' data. If there were multiple steps in the simulation, specify the number for which step you want to retrieve the time data.

Now that you have lists with the times and corresponding values, you can plot this information in an X/Y plot.

Note that all the data will be returned as numpy arrays.

See the class documentation for more details :

- `PyLTSpice.raw_read.RawRead`
- `PyLTSpice.raw_classes.Trace`

1.3.1 Example

The example below demonstrates the usage of the RawRead class. It reads a .RAW file and uses the matplotlib library to plot the results of two traces in a separate subplots.

```python
from PyLTSpice import RawRead
import matplotlib.pyplot as plt         # use matplotlib for plotting the results

raw = RawRead("some_random_file.raw")   # Read the RAW file contents from disk

print(raw.get_trace_names())            # Get and print a list of all the traces
print(raw.get_raw_property())           # Print all the properties found in the Header␣
↪section

vin = raw.get_trace('V(in)')            # Get the trace data
vout = raw.get_trace('V(out)')          # Get the second trace

steps = raw.get_steps()                 # Get list of step numbers ([0,1,2]) for sweeped␣
↪simulations
                                        # Returns [0] if there is just 1 step

plt.figure()                            # Create the canvas for plotting

_, (ax1, ax2) = plt.subplots(2, 1, sharex=True)  # Create two subplots

for ax in (ax1, ax2):                   # Use grid on both subplots
    ax.grid(True)

plt.xlim([0.9e-3, 1.2e-3])              # Limit the X axis to just a subrange
```

(continues on next page)

(continued from previous page)

```
xdata = raw.get_axis()                  # Get the X-axis data (time)

ydata = vin.get_wave()                  # Get all the values for the 'vin' trace
ax1.plot(xdata, ydata)                  # Do an X/Y plot on first subplot

ydata = vout.get_wave()                 # Get all the values for the 'vout' trace
ax1.plot(xdata, ydata)                  # Do an X/Y plot on first subplot as well

for step in steps:                      # On the second plot, print all the STEPS of Vout
    ydata = vout.get_wave(step)         # Retrieve the values for this step
    xdata = raw.get_axis(step)          # Retrieve the time vector
    ax2.plot(xdata, ydata)              # Do X/Y plot on second subplot

plt.show()                              # Show matplotlib's interactive window with the␣
↪plots
```

1.4 Reading Log Information

This module defines a class that can be used to parse LTSpice log files where the information about .STEP information is written.

There are two possible usages of this module, either programmatically by running the utility *LTSteps*, or by accessing data through the class as exemplified here:

```
from PyLTSpice.LTSteps import LTSpiceLogReader

data = LTSpiceLogReader("Batch_Test_AD820_15.log")

print("Number of steps  :", data.step_count)

# Get the names of the variables that were stepped, and the measurements taken
step_names = data.get_step_vars()
meas_names = data.get_measure_names()

# Print headers for a table with steps and measurements
print('\t'.join([f"{step}" for step in step_names]), end='\t')
print('\t'.join([f"{name}" for name in meas_names]), end='\n')

# Print the data.  First values of all step variables, then values of the measurements
for i in range(data.step_count):
        print('\t'.join([f"{data[step][i]}" for step in step_names]), end='\t')
        print('\t'.join([f"{data[name][i]}" for name in meas_names]), end='\n')

print("Total number of measurements found :", data.measure_count)
```

For more information, see `PyLTSpice.LTSteps.LTSpiceLogReader`

1.5 Writing Raw Files

The RawWrite class can be used to generate .RAW-files with generated or calculated data.

The functionality is limited to adding traces with a single series of values. Stepped data (like with a DC sweep) is not supported.

First, generate your trace data, e.g. using numpy.
Then, make a Trace object from this data.
Then, add all Trace objects to the RawWrite object.
Lastly, write the data to a file on disk.

The following example writes a RAW file with a 3 milliseconds transient simulation, containing a 10kHz sine and a 9.997kHz cosine wave.

```
import numpy as np
from PyLTSpice import Trace, RawWrite

LW = RawWrite()

tx = Trace('time', np.arange(0.0, 3e-3, 997E-11))
vy = Trace('N001', np.sin(2 * np.pi * tx.data * 10000))
vz = Trace('N002', np.cos(2 * np.pi * tx.data * 9970))

LW.add_trace(tx)
LW.add_trace(vy)
LW.add_trace(vz)

LW.save("test_sincos.raw")
```

For more information, see :

- *RAW File Structure*
- `PyLTSpice.raw_write.RawWrite`
- `PyLTSpice.raw_write.Trace`

1.6 Semiconductor Operating Point Reader

1.7 Client Server

Simulations take a huge computational power, and having the possibility of extending your own computer's capabilities can come as very handy. For this reason, PyLTspice has implemented a client-server architecture, where the server is the one machine and the client is the other. The server is the one that will run the simulation. The client will send the commands to the server and will receive the simulation raw and log files. At this moment there is no way to make the server machine execute the processing of raw files. It is therefore advised to make use of the .meas command in the netlist and to use the .raw file only for plotting purposes.

The server machine can be any machine that has LTspice installed. To start the server use the following command:

```
python -m PyLTSpice.run_server --port 9000 --parallel 4 --output ./temp
```

Make sure that each machine is on the same network and that the port is open. If port is not specified, the default port is 9000.

On the client side, you can use the following code to run the simulation:

```
import os.path
import zipfile
from PyLTSpice.client_server.sim_client import SimClient
import logging

# In order for this, to work, you need to have a server running. To start a server, run␣
↪the following command:
# python -m PyLTSpice.run_server --port 9000 --parallel 4 --output ./temp

_logger = logging.getLogger("PyLTSpice.SimClient")
_logger.setLevel(logging.DEBUG)

server = SimClient('http://localhost', 9000)
print(server.session_id)
runid = server.run("./testfiles/testfile.net")
print("Got Job id", runid)
for runid in server:  # Ma
    zip_filename = server.get_runno_data(runid)
    print(f"Received {zip_filename} from runid {runid}")
    with zipfile.ZipFile(zip_filename, 'r') as zipf:  # Extract the contents of the zip␣
↪file
        print(zipf.namelist())  # Debug printing the contents of the zip file
        zipf.extract(zipf.namelist()[0])  # Normally the raw file comes first

        ## Any treatment of the files can be done here

    os.remove(zip_filename)  # Remove the zip file

server.close_session()
```

The server will run the simulation and will send the raw and log files to the client. In order to minimize the data transfer between client and server, all files are zipped before being sent. When the run() command is called, the client will automatically zip the netlist file and send it to the server. The server will execute the simulation and will send the raw and log files back to the client also in zip format. The client has to unzip the files and then use them as needed.

In the example above the client is unzipping the first file and then deleting the zip file.

1.8 Sim Analysis Toolkit

The Sim Analysis toolkit is a collection of tools for setting up, running and analyzing simulations.

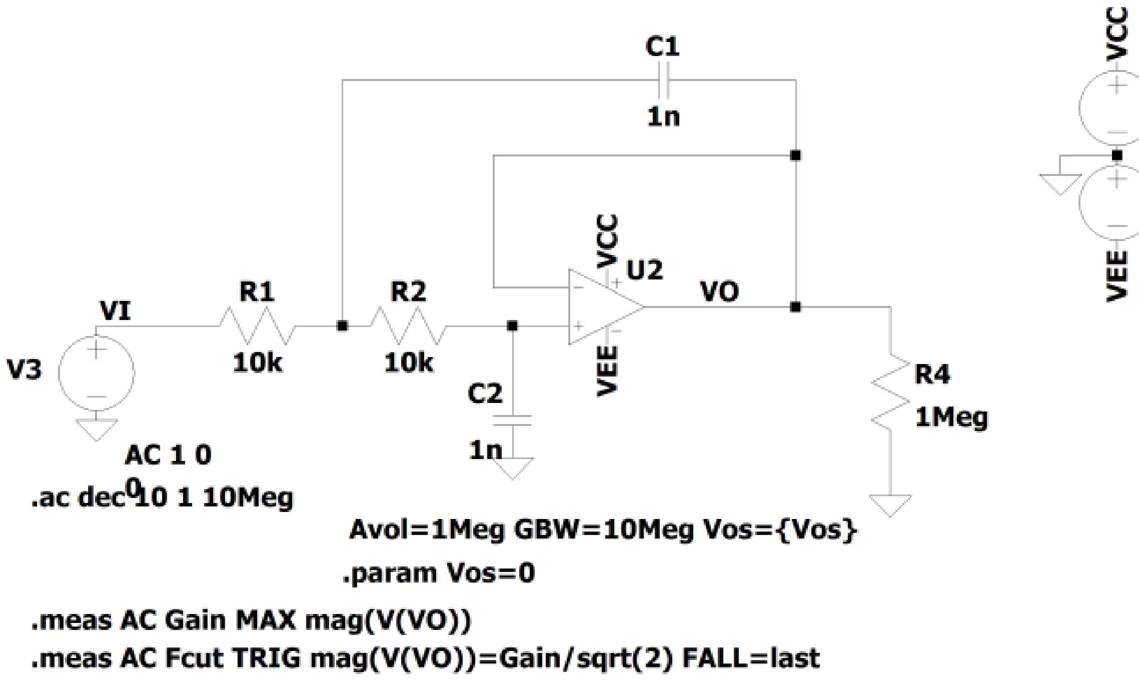
At present there are two main tools:

- Montecarlo simulation setup
- Worst Case simulation setup

Other tools will be added in the future.

## Simulation Setup ##

Let's consider the following circuit:


C1
1n
VCC
V1
15
V2
15
VEE
VCC
U2
VO
VI
R1
10k
R2
10k
V3
C2
1n
VEE
R4
1Meg
AC 1 0
0
.ac dec 10 1 10Meg
Avol=1Meg GBW=10Meg Vos={Vos}
.param Vos=0
.meas AC Gain MAX mag(V(VO))
.meas AC Fcut TRIG mag(V(VO))=Gain/sqrt(2) FALL=last

When performing a Monte Carlo simulation on this circuit, we need to manually modify the value of each component, and then add the .step command for making several runs on the same circuit. To simplify this process, the Montecarlo class can be used as exemplified below:

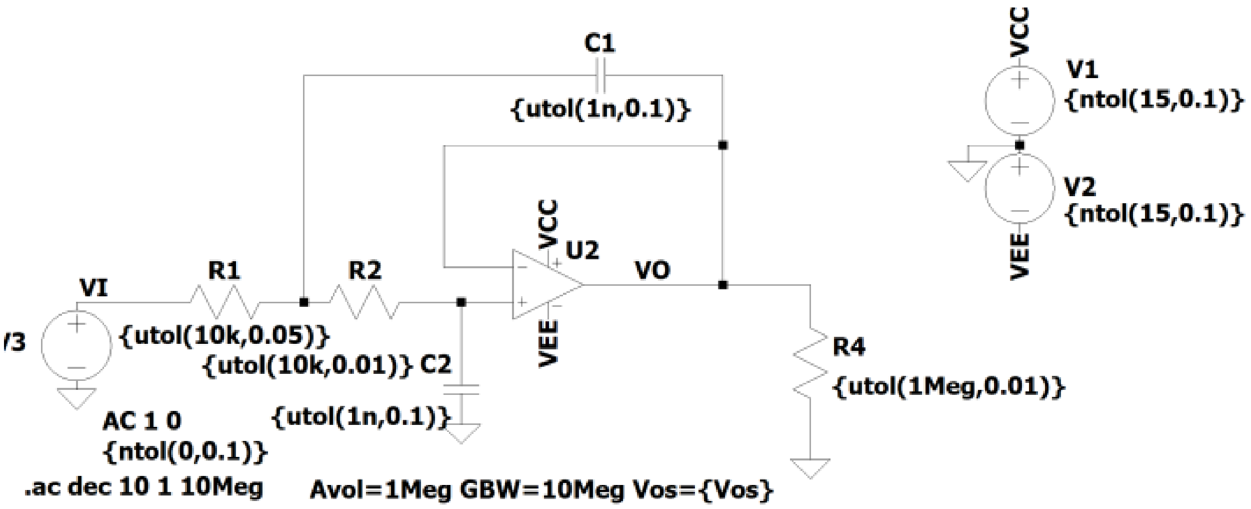
```
from PyLTSpice import AscEditor  # Imports the class that manipulates the asc file
from PyLTSpice.sim.tookit.montecarlo import Montecarlo  # Imports the Montecarlo toolkit␣
↪class

sallenkey = AscEditor("./testfiles/sallenkey.asc")  # Reads the asc file into memory

mc = Montecarlo(sallenkey)  # Instantiates the Montecarlo class, with the asc file␣
↪already in memory

# The following lines set the default tolerances for the components
mc.set_tolerance('R', 0.01)  # 1% tolerance, default distribution is uniform
mc.set_tolerance('C', 0.1, distribution='uniform')  # 10% tolerance, explicit uniform␣
↪distribution
```

(continues on next page)

(continued from previous page)

```
mc.set_tolerance('V', 0.1, distribution='normal')  # 10% tolerance, but using a normal␣
↪distribution

# Some components can have a different tolerance
mc.set_tolerance('R1', 0.05)  # 5% tolerance for R1 only. This only overrides the␣
↪default tolerance for R1

# Tolerances can be set for parameters as well
mc.set_parameter_deviation('Vos', 3e-4, 5e-3, 'uniform')  # The keyword 'distribution' is␣
↪optional
mc.prepare_testbench(1000)  # Prepares the testbench for 1000 simulations

# Finally the netlist is saved to a file
mc.save_netlist('./testfiles/sallenkey_mc.net')
```

When opening the created sallenkey_mc.net file, we can see that the following circuit.



```
.param Vos={urng(0,0.002650,-0.002350)}
.meas AC Gain MAX mag(V(VO))
.meas AC Fcut TRIG mag(V(VO))=Gain/sqrt(2) FALL=last
.function utol(nom,tol) if(run<0, nom, mc(nom,tol))
.function ntol(nom,tol) if(run<0, nom, nom*(1+gauss(tol/3)))
.function urng(nom,mean,df2) if(run<0, nom, mean*flat(df2))
.step param run -1 1000 1
.param run=-1
```

The following updates were made to the circuit:

- The value of each component was replaced by a function that generates a random value within the specified tolerance.
- The .step param run command was added to the netlist. Starts at -1 which it's the nominal value simulation, and finishes that the number of simulations specified in the prepare_testbench() method.
- A default value for the run parameter was added. This is useful if the .step param run is commented out.
- The R1 tolerance is different from the other resistors. This is because the tolerance was explicitly set for R1.
- The Vos parameter was added to the .param list. This is because the parameter was explicitly set using the set_parameter_deviation method.

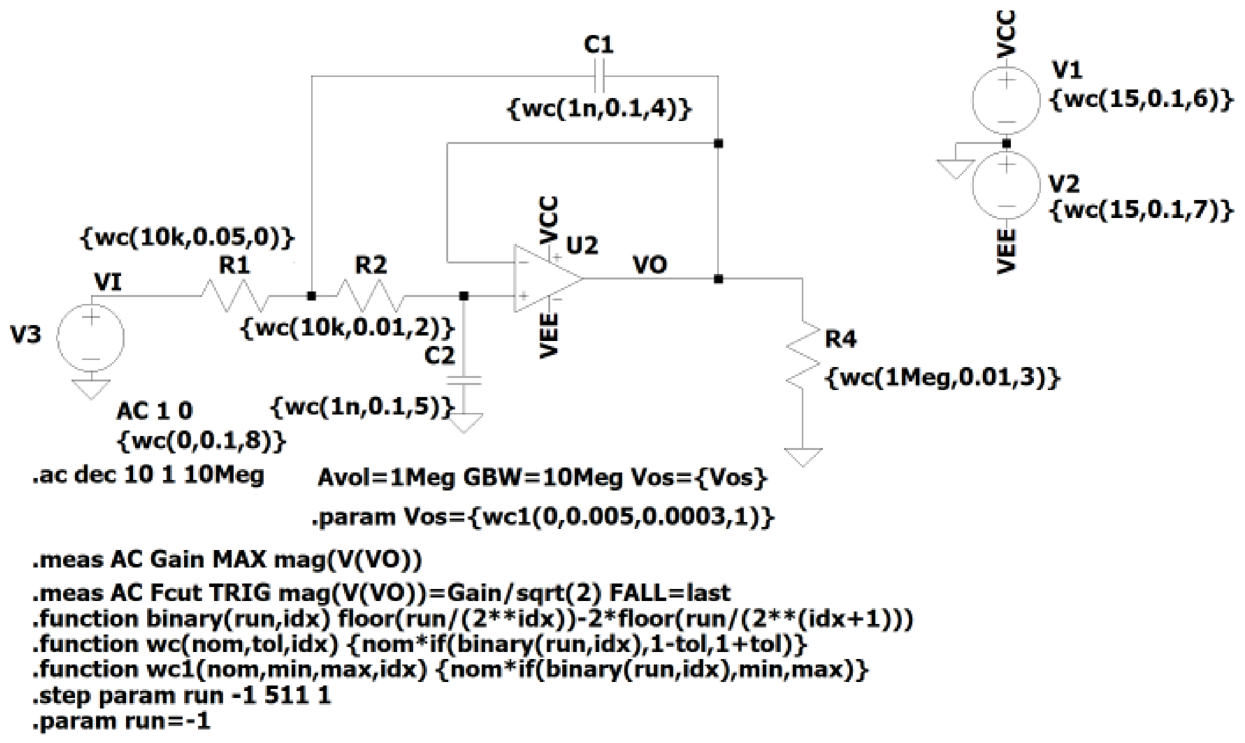
- Functions utol, ntol and urng were added to the .func list. These functions are used to generate random values.

Uniform distributions use the LTSpice built-in mc(x, tol) and flat(x) functions, while normal distributions use the gauss(x) function.

Similarly, the worst case analysis can also be setup by using the class WorstCaseAnalysis, as exemplified below:

```python
from PyLTSpice import AscEditor  # Imports the class that manipulates the asc file
from PyLTSpice.sim.tookit.worst_case import WorstCaseAnalysis

sallenkey = AscEditor("./testfiles/sallenkey.asc")  # Reads the asc file into memory

wca = WorstCaseAnalysis(sallenkey)  # Instantiates the Worst Case Analysis class

# The following lines set the default tolerances for the components
wca.set_tolerance('R', 0.01)  # 1% tolerance
wca.set_tolerance('C', 0.1)  # 10% tolerance
wca.set_tolerance('V', 0.1)  # 10% tolerance. For Worst Case analysis, the distribution is irrelevant

# Some components can have a different tolerance
wca.set_tolerance('R1', 0.05)  # 5% tolerance for R1 only. This only overrides the default tolerance for R1

# Tolerances can be set for parameters as well.
wca.set_parameter_deviation('Vos', 3e-4, 5e-3)

# Finally the netlist is saved to a file
wca.save_netlist('./testfiles/sallenkey_wc.asc')
```

When opening the created sallenkey_wc.net file, we can see that the following circuit.



```
.ac dec 10 1 10Meg
Avol=1Meg GBW=10Meg Vos={Vos}
.param Vos={wc1(0,0.005,0.0003,1)}
.meas AC Gain MAX mag(V(VO))
.meas AC Fcut TRIG mag(V(VO))=Gain/sqrt(2) FALL=last
.function binary(run,idx) floor(run/(2**idx))-2*floor(run/(2**(idx+1)))
.function wc(nom,tol,idx) {nom*if(binary(run,idx),1-tol,1+tol)}
.function wc1(nom,min,max,idx) {nom*if(binary(run,idx),min,max)}
.step param run -1 511 1
.param run=-1
```

The following updates were made to the circuit:

- The value of each component was replaced by a function that generates a nominal, minimum and maximum value depending on the run parameter and is assigned a unique index number. (R1=0, Vos=1, R2=2, … V2=7, VIN=8) The unique number corresponds to the bit position of the run parameter. Bit 0 corresponds to the minimum value and bit 1 corresponds to the maximum value. Calculating all possible permutations of maximum and minimum values for each component, we get 2**9 = 512 possible combinations. This maps into a 9 bit binary number, which is the run parameter.
- The .step param run command was added to the netlist. It starts at -1 which it's the nominal value simulation, then 0 which corresponds to the minimum value for each component, then it makes all combinations of minimum and maximum values until 511, which is the simulation with all maximum values.
- A default value for the run parameter was added. This is useful if the .step param run is commented out.
- The R1 tolerance is different from the other resistors. This is because the tolerance was explicitly set for R1.
- The wc() function is added to the circuit. This function is used to calculate the worst case value for each component, given a tolerance value and its respective index.
- The wc1() function is added to the circuit. This function is used to calculate the worst case value for each component, given a minimum and maximum value and its respective index.

CHAPTER

TWO

EDITOR CLASSES

2.1 SpiceEditor

Class used for manipulating SPICE netlists. Inherits from SpiceCircuit.

class PyLTSpice.editor.spice_editor.**SpiceEditor**(*netlist_file: Union[str, Path]*, *encoding='autodetect'*)

Bases: *SpiceCircuit*

This class implements interfaces to manipulate SPICE netlist files. The class doesn't update the netlist file itself. After implementing the modifications the user should call the "write_netlist" method to write a new netlist file.

Parameters

- **netlist_file** (*str or Path*) – Name of the .NET file to parse
- **encoding** (*str, optional*) – Forcing the encoding to be used on the circuit netlile read. Defaults to 'autodetect' which will call a function that tries to detect the encoding automatically. This however is not 100% fool proof.

add_instruction(*instruction: str*) → None

Serves to add SPICE instructions to the simulation netlist. For example:

```
.tran 10m ; makes a transient simulation
.meas TRAN Icurr AVG I(Rs1) TRIG time=1.5ms TARG time=2.5ms" ; Establishes a␣
↪measuring
.step run 1 100, 1 ; makes the simulation run 100 times
```

Parameters

instruction (*str*) – Spice instruction to add to the netlist. This instruction will be added at the end of the netlist, typically just before the .BACKANNO statement

Returns

Nothing

property circuit_file: Path

This is only here to avoid breaking compatibility with the BaseEditor superclass. It will always return ''

static find_subckt_in_lib(*library*, *subckt_name*) → Optional[*SpiceCircuit*]

Finds returns a Subckt from a library file.

Parameters

- **library** (*str*) – path to the library to search
- **subckt_name** (*str*) – sub-circuit to search for

Returns
Returns a SpiceCircuit instance with the sub-circuit found or None if not found

Return type
SpiceCircuit

remove_instruction(*instruction*) → None

Usage a previously added instructions. Example:

```
LTC.remove_instruction(".STEP run -1 1023 1")
```

This only works if the instruction exactly matches the line on the netlist. This means that space characters, and upper case and lower case differences will not match the line.

Parameters
instruction (*str*) – The list of instructions to remove. Each instruction is of the type 'str'

Returns
Nothing

reset_netlist() → None

Removes all previous edits done to the netlist, i.e. resets it to the original state.

Returns
Nothing

run(*wait_resource: bool = True*, *callback: Optional[Callable[[str, str], Any]] = None*, *timeout: float = 600*, *run_filename: Optional[str] = None*, *simulator=None*)

(Deprecated)

Convenience function for maintaining legacy with legacy code.

write_netlist(*run_netlist_file: Union[str, Path]*) → None

Writes the netlist will all the requested updates into a file named <run_netlist_file>.

Parameters
run_netlist_file (*Path or str*) – File name of the netlist file.

Returns
Nothing

2.2 SpiceCircuit

class PyLTSpice.editor.spice_editor.**SpiceCircuit**

Bases: `BaseEditor`

The Spice Circuit represents sub-circuits within a SPICE circuit and since sub-circuits can have sub-circuits inside them, it serves as base for the top level netlist. See class SpiceEditor This hierarchical approach helps to encapsulate and protect parameters and components from edits made at a higher level.

The netlist information is stored in a list, each element of the list corresponds to a SPICE instruction. If an instruction spawns more than a line with the '+' operator, it is contained in the same element.

This class serves as subclass to the SpiceEditor class.

add_instruction(*instruction: str*) → None

Serves to add SPICE instructions to the simulation netlist. For example:

```
.tran 10m ; makes a transient simulation
.meas TRAN Icurr AVG I(Rs1) TRIG time=1.5ms TARG time=2.5ms" ; Establishes a␣
↪measuring
.step run 1 100, 1 ; makes the simulation run 100 times
```

Parameters
: `instruction` (*str*) – Spice instruction to add to the netlist. This instruction will be added at the end of the netlist, typically just before the .BACKANNO statement

Returns
: Nothing

`static add_library_search_paths`(*paths: Union[str, List[str]]*) → None

Adding search paths for libraries. By default, the local directory and the ~username/"Documents/LTspiceXVII/lib/sub will be searched forehand. Only when a library is not found in these paths then the paths added by this method will be searched. Alternatively PyLTSpice.SpiceEditor.LibSearchPaths.append(paths) can be used."

Parameters
: `paths` (*str*) – Path to add to the Search path

Returns
: Nothing

Return type
: None

`property circuit_file: Path`

This is only here to avoid breaking compatibility with the BaseEditor superclass. It will always return ''

`clone`(***kwargs*) → *SpiceCircuit*

Creates a new copy of the SpiceCircuit. Change done at the new copy are not affecting the original

Key new_name
: The new name to be given to the circuit

Key type new_name
: str

Returns
: The new replica of the SpiceCircuit object

Return type
: *SpiceCircuit*

`get_all_nodes`() → List[str]

A function that retrieves all nodes existing on a Netlist

Returns
: Circuit Nodes

Return type
: list[str]

`get_component_info`(*component*) → dict

Retrieves the component information as defined in the corresponding REGEX. The line number is also added.

Parameters
`component` (*str*) – Reference of the component

Returns
Dictionary with the component information

Return type
dict

Raises
UnrecognizedSyntaxError when the line doesn't match the expected REGEX. NotImplementedError of there isn't an associated regular expression for the component prefix.

get_component_nodes(*element: str*) → List[str]

Returns the nodes to which the component is attached to.

Parameters
`element` (*str*) – Reference of the circuit element to get the nodes.

Returns
List of nodes

Return type
list

get_component_value(*element: str*) → str

Returns the value of a component retrieved from the netlist.

Parameters
`element` (*str*) – Reference of the circuit element to get the value.

Returns
value of the circuit element .

Return type
str

Raises
ComponentNotFoundError - In case the component is not found

NotImplementedError - for not supported operations

get_components(*prefixes='*'*) → list

Returns a list of components that match the list of prefixes indicated on the parameter prefixes. In case prefixes is left empty, it returns all the ones that are defined by the REPLACE_REGEXES. The list will contain the designators of all components found.

Parameters
`prefixes` (*str*) – Type of prefixes to search for. Examples: 'C' for capacitors; 'R' for Resistors; etc… See prefixes in SPICE documentation for more details. The default prefix is '*' which is a special case that returns all components.

Returns
A list of components matching the prefixes demanded.

get_parameter(*param: str*) → str

Retrieves a Parameter from the Netlist

Parameters
`param` (*str*) – Name of the parameter to be retrieved

Returns
Value of the parameter being sought

Return type
str

Raises
ParameterNotFoundError - In case the component is not found

`name()` → str

Returns the name of the Sub-Circuit -> str.

Return type
str

`remove_component`(*designator: str*) → None

Removes a component from the design. Note: Current implementation only allows removal of a component from the main netlist, not from a sub-circuit.

Parameters
designator (*str*) – Component reference in the design. Ex: V1, C1, R1, etc…

Returns
Nothing

Raises
ComponentNotFoundError - When the component doesn't exist on the netlist.

`remove_instruction`(*instruction: str*) → None

Usage a previously added instructions. Example:

```
LTC.remove_instruction(".STEP run -1 1023 1")
```

This only works if the instruction exactly matches the line on the netlist. This means that space characters, and upper case and lower case differences will not match the line.

Parameters
instruction (*str*) – The list of instructions to remove. Each instruction is of the type 'str'

Returns
Nothing

`reset_netlist()` → None

Resets the netlist to the original state

`set_component_value`(*device: str, value: Union[str, int, float]*) → None

Changes the value of a component, such as a Resistor, Capacitor or Inductor. For components inside sub-circuits, use the sub-circuit designator prefix with ':' as separator (Example X1:R1) Usage:

```
LTC.set_component_value('R1', '3.3k')
LTC.set_component_value('X1:C1', '10u')
```

Parameters

- **device** (*str*) – Reference of the circuit element to be updated.
- **value** (*str, int or float*) – value to be set on the given circuit element. Float and integer values will be automatically formatted as per the engineering notations 'k' for kilo, 'm', for mili and so on.

Raises
ComponentNotFoundError - In case the component is not found

ValueError - In case the value doesn't correspond to the expected format

NotImplementedError - In case the circuit element is defined in a format which is not supported by this version.

If this is the case, use GitHub to start a ticket. https://github.com/nunobrum/PyLTSpice

set_element_model(*element: str*, *model: str*) → None

Changes the value of a circuit element, such as a diode model or a voltage supply. Usage:

```
LTC.set_element_model('D1', '1N4148')
LTC.set_element_model('V1' "SINE(0 1 3k 0 0 0)")
```

Parameters

- **element** (*str*) – Reference of the circuit element to be updated.
- **model** (*str*) – model name of the device to be updated

Raises

ComponentNotFoundError - In case the component is not found

ValueError - In case the model format contains irregular characters

NotImplementedError - In case the circuit element is defined in a format which is not supported by this version.

If this is the case, use GitHub to start a ticket. https://github.com/nunobrum/PyLTSpice

set_parameter(*param: str*, *value: Union[str, int, float]*) → None

Adds a parameter to the SPICE netlist.

Usage:

```
LTC.set_parameter("TEMP", 80)
```

This adds onto the netlist the following line:

```
.PARAM TEMP=80
```

This is an alternative to the set_parameters which is more pythonic in it's usage, and allows setting more than one parameter at once.

Parameters

- **param** (*str*) – Spice Parameter name to be added or updated.
- **value** (*str, int or float*) – Parameter Value to be set.

Returns

Nothing

setname(*new_name: str*)

Renames the sub-circuit to a new name. No check is done to the new game give. It is up to the user to make sure that the new name is valid.

Parameters

new_name (*str*) – The new Name.

Returns

Nothing

Return type
None

write_lines(*f*)

Internal function. Do not use.

write_netlist(*run_netlist_file: Union[str, Path]*) → None

Writes the netlist to a file

2.3 AscEditor

Class used for manipulating LTSpice asc files.

class PyLTSpice.editor.asc_editor.**AscEditor**(*asc_file: str*)

Bases: `BaseEditor`

Class made to update directly the ltspice ASC files

add_instruction(*instruction: str*) → None

Serves to add SPICE instructions to the simulation netlist. For example:

```
.tran 10m ; makes a transient simulation
.meas TRAN Icurr AVG I(Rs1) TRIG time=1.5ms TARG time=2.5ms" ; Establishes a␣
↪measuring
.step run 1 100, 1 ; makes the simulation run 100 times
```

Parameters
instruction (*str*) – Spice instruction to add to the netlist. This instruction will be added at the end of the netlist, typically just before the .BACKANNO statement

Returns
Nothing

property circuit_file: Path

Returns the netlist as a string

get_component_info(*component*) → dict

Returns the component information as a dictionary

get_component_value(*element: str*) → str

Returns the value of a component retrieved from the netlist.

Parameters
element (*str*) – Reference of the circuit element to get the value.

Returns
value of the circuit element .

Return type
str

Raises
ComponentNotFoundError - In case the component is not found

NotImplementedError - for not supported operations

get_components(*prefixes='*'*) → list

Returns a list of components that match the list of prefixes indicated on the parameter prefixes. In case prefixes is left empty, it returns all the ones that are defined by the REPLACE_REGEXES. The list will contain the designators of all components found.

Parameters
: **prefixes** (*str*) – Type of prefixes to search for. Examples: 'C' for capacitors; 'R' for Resistors; etc… See prefixes in SPICE documentation for more details. The default prefix is '*' which is a special case that returns all components.

Returns
: A list of components matching the prefixes demanded.

get_parameter(*param: str*) → str

Retrieves a Parameter from the Netlist

Parameters
: **param** (*str*) – Name of the parameter to be retrieved

Returns
: Value of the parameter being sought

Return type
: str

Raises
: ParameterNotFoundError - In case the component is not found

remove_component(*designator: str*)

Removes a component from the design. Note: Current implementation only allows removal of a component from the main netlist, not from a sub-circuit.

Parameters
: **designator** (*str*) – Component reference in the design. Ex: V1, C1, R1, etc…

Returns
: Nothing

Raises
: ComponentNotFoundError - When the component doesn't exist on the netlist.

remove_instruction(*instruction: str*) → None

Usage a previously added instructions. Example:

```
LTC.remove_instruction(".STEP run -1 1023 1")
```

This only works if the instruction exactly matches the line on the netlist. This means that space characters, and upper case and lower case differences will not match the line.

Parameters
: **instruction** (*str*) – The list of instructions to remove. Each instruction is of the type 'str'

Returns
: Nothing

reset_netlist()

Resets the netlist to the original state

set_component_value(*device: str*, *value: Union[str, int, float]*) → None

Changes the value of a component, such as a Resistor, Capacitor or Inductor. For components inside sub-circuits, use the subcirciut designator prefix with ':' as separator (Example X1:R1) Usage:

```
LTC.set_component_value('R1', '3.3k')
LTC.set_component_value('X1:C1', '10u')
```

Parameters

- **device** (*str*) – Reference of the circuit element to be updated.
- **value** (*str, int or float*) – value to be be set on the given circuit element. Float and integer values will automatically formatted as per the engineering notations 'k' for kilo, 'm', for mili and so on.

Raises

ComponentNotFoundError - In case the component is not found

ValueError - In case the value doesn't correspond to the expected format

NotImplementedError - In case the circuit element is defined in a format which is not supported by this version.

If this is the case, use GitHub to start a ticket. https://github.com/nunobrum/PyLTSpice

set_element_model(*element: str*, *model: str*) → None

Changes the value of a circuit element, such as a diode model or a voltage supply. Usage:

```
LTC.set_element_model('D1', '1N4148')
LTC.set_element_model('V1' "SINE(0 1 3k 0 0 0)")
```

Parameters

- **element** (*str*) – Reference of the circuit element to be updated.
- **model** (*str*) – model name of the device to be updated

Raises

ComponentNotFoundError - In case the component is not found

ValueError - In case the model format contains irregular characters

NotImplementedError - In case the circuit element is defined in a format which is not supported by this version.

If this is the case, use GitHub to start a ticket. https://github.com/nunobrum/PyLTSpice

set_parameter(*param: str*, *value: Union[str, int, float]*) → None

Adds a parameter to the SPICE netlist.

Usage:

```
LTC.set_parameter("TEMP", 80)
```

This adds onto the netlist the following line:

```
.PARAM TEMP=80
```

This is an alternative to the set_parameters which is more pythonic in it's usage, and allows setting more than one parameter at once.

Parameters

- **param** (*str*) – Spice Parameter name to be added or updated.

- **`value`** (*`str, int or float`*) – Parameter Value to be set.

Returns
Nothing

`write_netlist`(*run_netlist_file: Union[str, Path]*) → None

Writes the netlist to a file

CHAPTER

THREE

SIMULATION CLASSES

3.1 SimRunner

class PyLTSpice.sim.run_task.**RunTask**(*simulator: Type[Simulator], runno, netlist_file: Path, callback: Union[Type[ProcessCallback], Callable[[Path, Path], Any]], switches, timeout: Optional[float] = None, verbose=True*)

This is an internal Class and should not be used directly by the User.

get_results() → Union[None, Any, Tuple[str, str]]

Returns the simulation outputs if the simulation and callback function has already finished. If the simulation is not finished, it simply returns None. If no callback function is defined, then it returns a tuple with (raw_file, log_file). If a callback function is defined, it returns whatever the callback function is returning.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

wait_results() → Union[Any, Tuple[str, str]]

Waits for the completion of the task and returns a tuple with the raw and log files. :returns: Tuple with the path to the raw file and the path to the log file :rtype: tuple(str, str)

3.2 Simulators

class PyLTSpice.sim.simulator.**Simulator**

Bases: ABC

Pure static class template for Spice simulators. This class only defines the interface of the subclasses. The variables below shall be overridden by the subclasses. Instantiating this class will raise a SpiceSimulatorError exception.

A typical subclass for a Windows installation is:

```
class MySpiceWindowsInstallation(Simulator):
    spice_exe = ['<path to your own ltspice installation>']
    process_name = "<name of the process on Windows Task Manager>"
```

or on a Linux distribution:

```
class MySpiceLinuxInstallation(Simulator):
    spice_exe = ['<wine_command', '<path to your own ltspice installation>']
    process_name = "<name of the process>"
```

The subclasses should then implement at least the run() function as a classmethod.

```
@classmethod
def run(cls, netlist_file, cmd_line_switches, timeout):
    '''This method implements the call for the simulation of the netlist file. '''
    cmd_run = cls.spice_exe + ['-Run'] + ['-b'] + [netlist_file] + cmd_line_switches
    return run_function(cmd_run, timeout=timeout)
```

The `run_function()` can be imported from the simulator.py with `from PyLTSpice.sim.simulator import run_function` instruction.

classmethod create_from(*path_to_exe*, *process_name=None*)

Creates a simulator class from a path to the simulator executable :param path_to_exe: :type path_to_exe: pathlib.Path or str :param process_name: assigning a process_name to be used for killing phantom processes :return: a class instance representing the Spice simulator :rtype: LTspice

process_name = ''

raw_extension = '.raw'

abstract classmethod run(*netlist_file*, *cmd_line_switches*, *timeout*)

This method implements the call for the simulation of the netlist file. This should be overriden by its subclass.

spice_exe = []

abstract classmethod valid_switch(*switch*, *switch_param*) → list

This method validates that a switch exist and is valid. This should be overriden by its subclass.

3.3 LTSpice

class PyLTSpice.sim.ltspice_simulator.**LTspice**

Bases: *Simulator*

Stores the simulator location and command line options and is responsible for generating netlists and running simulations.

classmethod create_netlist(*circuit_file: Union[str, Path]*) → Path

default_folder = '/home/docs/.wine/drive_c/Program Files/LTC/LTspiceXVII'

ltspice_args = {'FixUpSchematicFonts': ['-FixUpSchematicFonts'], 'FixUpSymbolFonts': ['-FixUpSymbolFonts'], 'I': ['-I<path>'], 'PCBnetlist': ['-PCBnetlist'], 'SOI': ['-SOI'], 'alt': ['-alt'], 'ascii': ['-ascii'], 'big': ['-big'], 'encrypt': ['-encrypt'], 'fastaccess': ['-FastAccess'], 'ini': ['-ini', '<path>'], 'max': ['-max'], 'netlist': ['-netlist'], 'norm': ['-norm'], 'sync': ['-sync'], 'uninstall': ['-uninstall']}

process_name = 'XVIIx64.exe'

classmethod **run**(*netlist_file*, *cmd_line_switches*, *timeout*)

This method implements the call for the simulation of the netlist file. This should be overriden by its subclass.

spice_exe = ['wine', '/home/docs/.wine/drive_c/Program Files/LTC/LTspiceXVII/XVIIx64.exe']

spice_executable = None

spice_folder = None

classmethod **valid_switch**(*switch*, *path=''*) → list

Validates a command line switch. The following options are available for LTSpice:

- 'alt' : Set solver to Alternate.
- 'ascii' : Use ASCII.raw files. Seriously degrades program performance.
- **'encrypt'**
  [Encrypt a model library.For 3rd parties wishing to allow people to use libraries without] revealing implementation details. Not used by AnalogDevices models.
- 'fastaccess': Batch conversion of a binary.rawfile to Fast Access format.
- **'FixUpSchematicFonts'**
  [Convert the font size field of very old user - authored schematic text to the] modern default.
- **'FixUpSymbolFonts'**
  [Convert the font size field of very old user - authored symbols to the modern] default. See Changelog.txt for application hints.
- 'ini <path>' : Specify an .ini file to use other than %APPDATA%LTspice.ini
- **'I<path>'**
  [Specify a path to insert in the symbol and file search paths. Must be the last specified] option.
- 'netlist' : Batch conversion of a schematic to a netlist.
- 'normal' : Set solver to Normal.
- 'PCBnetlist': Batch conversion of a schematic to a PCB format netlist.
- 'SOI' : Allow MOSFET's to have up to 7 nodes even in subcircuit expansion.
- 'sync' : Update component libraries
- 'uninstall' : Executes one step of the uninstallation process. Please don't.

Parameters

- **switch** (*str*) – switch to be added. If the switch is not on the list above, it should be correctly formatted with the preceding '-' switch
- **path** (*str, optional*) – path to the file related to the switch being given.

Returns
Nothing

Return type
None

3.4 Others

class `PyLTSpice.sim.ngspice_simulator.`**`NGspiceSimulator`**

Bases: *`Simulator`*

Stores the simulator location and command line options and runs simulations.

`default_run_switches = ['-b', '-o', '-r', '-a']`

`ngspice_args = {'--autorun': ['--autorun'], '--batch': ['--batch'], '--circuitfile': ['--circuitfile', '<FILE>'], '--completion': ['--completion'], '--define': ['--define', 'var_value'], '--help': ['--help'], '--interactive': ['--interactive'], '--no-spiceinit': ['--no-spiceinit'], '--output': ['--output', '<FILE>'], '--pipe': ['--pipe'], '--rawfile': ['--rawfile', '<FILE>'], '--server': ['--server'], '--soa-log': ['--soa-log', '<FILE>'], '--term': ['--term', '<TERM>'], '--version': ['--version'], '-D': ['-D', 'var_value'], '-a': ['-a'], '-b': ['-b'], '-c': ['-c', '<FILE>'], '-h': ['-h'], '-i': ['-i'], '-n': ['-n'], '-o': ['-o', '<FILE>'], '-p': ['-p'], '-q': ['-q'], '-r': ['-r'], '-s': ['-s'], '-t': ['-t', '<TERM>'], '-v': ['-v']}`

`process_name = 'ngspice.exe'`

`classmethod` **`run`**(*netlist_file*, *cmd_line_switches*, *timeout*)

This method implements the call for the simulation of the netlist file. This should be overriden by its subclass.

`spice_exe = ['C:/Apps/NGSpice64/bin/ngspice.exe']`

`classmethod` **`valid_switch`**(*switch*, *parameter=''*) → list

Validates a command line switch. The following options are available for NGSpice:

Parameters

- **`switch`** (*`str`*) – switch to be added. If the switch is not on the list above, it should be correctly formatted with the preceding '-' switch
- **`parameter`** (*`str, optional`*) – parameter for the switch

Returns

the correct formatting for the switch

Return type

list

class `PyLTSpice.sim.xyce_simulator.`**`XyceSimulator`**

Bases: *`Simulator`*

Stores the simulator location and command line options and runs simulations.

`process_name = 'XVIIx64.exe'`

`classmethod` **`run`**(*netlist_file*, *cmd_line_switches*, *timeout*)

This method implements the call for the simulation of the netlist file. This should be overriden by its subclass.

`spice_exe = ['C:/Program Files/Xyce 7.6 NORAD/bin/xyce.exe']`

classmethod **valid_switch**(*switch*, *parameter=''*) → list

Validates a command line switch. The following options are available for Xyce:

Parameters

- **switch** (*str*) – switch to be added. If the switch is not on the list above, it should be correctly formatted with the preceding '-' switch
- **parameter** (*str, optional*) – parameter for the switch

Returns
the correct formatting for the switch

Return type
list

xyce_args = {'-a': ['-a', '<path>'], '-b': ['-b'], '-capabilities': ['-capabilities'], '-count': ['-count'], '-delim': ['-delim', '<delim_option>'], '-doc': ['-doc', '<param_options>'], '-doc_cat': ['-doc_cat', '<param_options>'], '-h': ['-h'], '-hspice-ext': ['-hspice-ext', '<hsext_options>'], '-jacobian_test': ['-jacobian_test'], '-l': ['-l', '<path>'], '-license': ['-license'], '-linsolv': ['-linsolv', '<solver>'], '-max': ['-max', '<int_option>'], '-maxord': ['-maxord', '<int_option>'], '-namesfile': ['-namesfile', '<path>'], '-noise_names_file': ['-noise_names_file', '<path>'], '-norun': ['-norun'], '-nox': ['-nox', 'onoff_option'], '-o': ['-o', '<basename>'], '-param': ['-param', '<param_options>'], '-per-processor': ['-per-processor'], '-prf': ['-prf', '<path>'], '-quiet': ['-quiet'], '-r': ['-r', '<path>'], '-randseed': ['-randseed', '<int_option>'], '-redefined_params': ['-redefined_params', '<redef_param_option>'], '-remeasure': ['-remeasure', '<path>'], '-rsf': ['-rsf', '<path>'], '-subckt_multiplier': ['-subckt_multiplier', '<truefalse_option>'], '-syntax': ['-syntax'], '-v': ['-v']}

3.5 SimClient

3.6 SimServer

CHAPTER

FOUR

DEPRECATIONS

4.1 SimCommander

CHAPTER

FIVE

RAW AND LOG FILE CLASSES

5.1 RawRead

class PyLTSpice.raw.raw_read.**RawRead**(*raw_filename: str*, *traces_to_read: Optional[Union[str, List[str], Tuple[str, ...]]] = '*'*, ***kwargs*)

Bases: `object`

Class for reading LTSpice wave Files. It can read all types of Files. If stepped data is detected, it will also try to read the corresponding LOG file so to retrieve the stepped data.

Parameters

- **raw_filename** (*str | pahtlib.Path*) – The file containing the RAW data to be read
- **traces_to_read** (*str, list or tuple*) – A string or a list containing the list of traces to be read. If None is provided, only the header is read and all trace data is discarded. If a '*' wildcard is given or no parameter at all then all traces are read.

Key headeronly

Used to only load the header information and skip the trace data entirely. Use *headeronly=True*.

ACCEPTED_PLOTNAMES = ('AC Analysis', 'DC transfer characteristic', 'Operating Point', 'Transient Analysis', 'Transfer Function', 'Noise Spectral Density', 'Frequency Response Analysis')

export(*columns: Optional[list] = None*, *step: Union[int, List[int]] = -1*, ***kwargs*) → Dict[str, list]

Returns a native python class structure with the requested trace data and steps. It consists of an ordered dictionary where the columns are the keys and the values are lists with the data.

This function is used by the export functions.

Parameters

- **step** (*int*) – Step number to retrieve. If not given, it will return all steps
- **columns** (*list*) – List of traces to use as columns. Default is all traces
- **kwargs** (***dict*) – Additional arguments to pass to the pandas.DataFrame constructor

Returns

A pandas DataFrame

Return type

pandas.DataFrame

`get_axis`(*step: int = 0*)

This function is equivalent to get_trace(0).get_wave(step) instruction. It also implements a workaround on a LTSpice issue when using 2nd Order compression, where some values on the time trace have a negative value. :param step: Step number :type step: int :returns: Array with the X axis :rtype: list[float] or numpy.array

`get_len`(*step: int = 0*) → int

Returns the length of the data at the give step index. :param step: Optional parameter the step index. :type step: int :return: The number of data points :rtype: int

`get_raw_property`(*property_name=None*)

Get a property. By default, it returns all properties defined in the RAW file.

Parameters
: **property_name** (*str*) – name of the property to retrieve.

Returns
: Property object

Return type
: str

Raises
: ValueError if the property doesn't exist

`get_steps`(***kwargs*)

Returns the steps that correspond to the query set in the * * kwargs parameters. Example:

```
raw_read.get_steps(V5=1.2, TEMP=25)
```

This will return all steps in which the voltage source V5 was set to 1.2V and the TEMP parameter is 24 degrees. This feature is only possible if a .log file with the same name as the .raw file exists in the same directory. Note: the correspondence between step numbers and .STEP information is stored on the .log file.

Key kwargs
: key-value arguments in which the key correspond to a stepped parameter or source name, and the value is the stepped value.

Returns
: The steps that match the query

Return type
: list[int]

`get_time_axis`(*step: int = 0*)

(Deprecated) Use get_axis method instead

This function is equivalent to get_trace('time').get_time_axis(step) instruction. It's workaround on a LTSpice issue when using 2nd Order compression, where some values on the time trace have a negative value.

`get_trace`(*trace_ref: Union[str, int]*)

Retrieves the trace with the requested name (trace_ref).

Parameters
: **trace_ref** (*str or int*) – Name of the trace or the index of the trace

Returns
: An object containing the requested trace

Return type
: DataSet subclass

Raises
`IndexError` – When a trace is not found

`get_trace_names()`
Returns a list of exiting trace names of the RAW file.

Returns
trace names

Return type
list[str]

`get_wave`(*trace_ref: Union[str, int]*, *step: int = 0*)
Retrieves the trace data with the requested name (trace_ref), optionally providing the step number.

Parameters

- **trace_ref** (*str or int*) – Name of the trace or the index of the trace
- **step** (*int*) – Optional parameter specifying which step to retrieve.

Returns
A numpy array containing the requested waveform.

Return type
numpy.array

Raises
`IndexError` – When a trace is not found

header_lines = ('Title', 'Date', 'Plotname', 'Output', 'Flags', 'No. Variables', 'No. Points', 'Offset', 'Command', 'Variables', 'Backannotation')

`to_csv`(*filename: Union[str, Path]*, *columns: Optional[list] = None*, *step: Union[int, List[int]] = -1*, *separator=','*, ***kwargs*)
Saves the data to a CSV file.

Parameters

- **filename** (*str*) – Name of the file to save the data to
- **columns** (*list*) – List of traces to use as columns. Default is all traces
- **step** (*int*) – Step number to retrieve. If not given, it
- **separator** (*str*) – separator to use in the CSV file
- **kwargs** (***dict*) – Additional arguments to pass to the pandas.DataFrame.to_csv function

`to_dataframe`(*columns: Optional[list] = None*, *step: Union[int, List[int]] = -1*, ***kwargs*)
Returns a pandas DataFrame with the requested data.

Parameters

- **step** (*int*) – Step number to retrieve. If not given, it
- **columns** (*list*) – List of traces to use as columns. Default is all traces
- **kwargs** (***dict*) – Additional arguments to pass to the pandas.DataFrame constructor

Returns
A pandas DataFrame

Return type
pandas.DataFrame

to_excel(*filename: Union[str, Path]*, *columns: Optional[list] = None*, *step: Union[int, List[int]] = -1*, ***kwargs*)

Saves the data to an Excel file. :param filename: Name of the file to save the data to :type filename: str :param columns: List of traces to use as columns. Default is all traces :type columns: list :param step: Step number to retrieve. If not given, it :type step: int :param kwargs: Additional arguments to pass to the pandas.DataFrame.to_excel function :type kwargs: **dict

5.2 RawRead Trace

class PyLTSpice.raw.raw_classes.**Axis**(*name: str*, *whattype: str*, *datalen: int*, *numerical_type: str = 'double'*)

Bases: `DataSet`

This class is used to represent the horizontal axis like on a Transient or DC Sweep Simulation. It derives from the DataSet and defines additional methods that are specific for X axis. This class is constructed by the get_time_axis() method or by a get_trace(0) command. In RAW files the trace 0 is always the X Axis. Ex: time for .TRAN simulations and frequency for the .AC simulations.

To access data inside this class, the get_wave() should be used, which implements the support for the STEPed data. IF Numpy is available, get_wave() will return a numpy array.

In Transient Analysis and in DC transfer characteristic, LTSpice uses doubles to store the axis values.

get_len(*step: int = 0*) → int

Returns the length of the axis. :param step: Optional parameter the step index. :type step: int :return: The number of data points :rtype: int

get_point(*n*, *step: int = 0*) → Union[float, complex]

Get a point from the dataset :param n: position on the vector :type n:int :param step: step index :type step: int :returns: Value of the data point :rtype: float or complex

get_position(*t*, *step: int = 0*) → Union[int, float]

Returns the position of a point in the axis. If the point doesn't exist, an interpolation is done between the two closest points. For example, if the point requested is 1.0001ms and the closest points that exist in the axis are t[100]=1ms and t[101]=1.001ms, then the return value will be 100 + (1.0001ms-1ms)/(1.001ms-1ms) = 100.1

Parameters

- **t** (*float*) – point in axis to search for
- **step** (*int*) – step number

Returns
The position of parameter /t/ in the axis

Return type
int, float

get_time_axis(*step: int = 0*)

Deprecated. Use get_wave() instead.

Returns the time axis raw data. Please note that the time axis may not have a constant time step. LTSpice will increase the time-step in simulation phases where there aren't value changes, and decrease time step in the parts where more time accuracy is needed.

Parameters
step (*int*) – Optional step number if reading a raw file with stepped data.

Returns
time axis

Return type
numpy.array

`get_wave`(*step: int = 0*) → array

Returns a vector containing the wave values. If numpy is installed, data is returned as a numpy array. If not, the wave is returned as a list of floats.

If stepped data is present in the array, the user should specify which step is to be returned. Failing to do so, will return all available steps concatenated together.

Parameters
step (*int*) – Optional step in stepped data raw files.

Returns
The trace values

Return type
numpy.array

`step_offset`(*step: int*)

In Stepped RAW files, several simulations runs are stored in the same RAW file. This function returns the offset within the binary stream where each step starts.

Parameters
step (*int*) – Number of the step within the RAW file

Returns
The offset within the RAW file

Return type
int

class PyLTSpice.raw.raw_classes.**TraceRead**(*name*, *whattype*, *datalen*, *axis*, *numerical_type='real'*)

Bases: `DataSet`

This class is used to represent a trace. It derives from DataSet and implements the additional methods to support STEPed simulations. This class is constructed by the get_trace() command. Data can be accessed through the [] and len() operators, or by the get_wave() method. If numpy is available the get_wave() method will return a numpy array.

`get_len`(*step: int = 0*) → int

Returns the length of the axis. :param step: Optional parameter the step index. :type step: int :return: The number of data points :rtype: int

`get_point`(*n: int*, *step: int = 0*) → Union[float, complex]

Implementation of the [] operator.

Parameters

- **n** (*int*) – item in the array
- **step** (*int*) – Optional step number

Returns
float value of the item

Return type
float

`get_point_at`(*t*, *step: int = 0*) → Union[float, complex]

Get a point from the trace at the point specified by the /t/ argument. If the point doesn't exist on the axis, the data is interpolated using a linear regression between the two adjacent points. :param t: point in the axis where to find the point. :type t: float, float32(numpy) or float64(numpy) :param step: step index :type step: int

`get_wave`(*step: int = 0*) → array

Returns the data contained in this object. For stepped simulations an argument must be passed specifying the step number. If no steps exist, the argument must be left blank. To know whether stepped data exist, the user can use the get_raw_property('Flags') method.

If numpy is available the get_wave() method will return a numpy array.

Parameters

`step` (`int`) – To be used when stepped data exist on the RAW file.

Returns

a List or numpy array (if installed) containing the data contained in this object.

Return type

numpy.array

5.3 LTSpiceLogReader

5.4 RawWrite

`class PyLTSpice.raw.raw_write.RawWrite`(*plot_name=None*, *fastacces=True*, *numtype='auto'*, *encoding='utf_16_le'*)

Bases: `object`

This class represents the RAW data file being generated. Contrary to the RawRead this class doesn't support stepped data.

`add_trace`(*trace:* Trace)

Adds a trace to the RAW file. The trace needs to have the same size as trace 0 ('time', 'frequency', etc..) The first trace added defines the X-Axis and therefore the type of RAW file being generated. If no plot name was defined, it will automatically assign a name. :param trace: Needs to be of the :type trace: :return: Nothing :rtype: None

`add_traces_from_raw`(*other:* RawRead, *trace_filter: Union[list, tuple, str]*, ***kwargs*)

(Not fully implemented)

Merge two RawWrite classes together resulting in a new instance :param other: an instance of the RawRead class where the traces are going to be copied from. :type other: RawRead :param trace_filter: A list of signals that should be imported into the new file :type trace_filter: list, Tuple, or just a string for one trace

Parameters

- `force_axis_alignment` – If two raw files don't have the same axis, an attempt is made to sync the two
- `admissible_error` – maximum error allowed in the sync between the two axis
- `rename_format` – when adding traces with the same name, it is possible to define a rename format. For example, if there are two traces named N001 in order to avoid duplicate names the rename format can be defined as "{}_{kwarg_name} where kwarg_name is passed as

a keyword argument of this function. If just one trace is being added, this can be used to simply give the new name.

- **step** – by default only step 0 is added from the second raw. It is possible to add other steps, by using this keyword parameter. This is useful when we want to "flatten" the multiple step runs into the same view.

Keyword
minimum_timestep: This parameter forces the two axis to sync using a minimum time step. That is, all time increments that are less than this parameter will be suppressed.

Returns
Nothing

`get_trace`(*trace_ref*)

Retrieves the trace with the requested name (trace_ref).

Parameters
`trace_ref` (*`str`*) – Name of the trace

Returns
An object containing the requested trace

Return type
DataSet subclass

`save`(*filename: str*)

Saves the RAW file into a file. The file format is always binary. Text based RAW output format is not supported in this version. :param filename: filename to where the RAW file is going to be written. Make sure that the extension of the file is .RAW.

Returns
Nothing

Return type
None

5.5 RawWrite Trace

`class PyLTSpice.raw.raw_write.`**`Trace`**(*name*, *data*, *whattype='voltage'*, *numerical_type=''*)

Bases: `DataSet`

Helper class representing a trace. This class is based on DataSet, therefore, it doesn't support STEPPED data. :param name: name of the trace being created :type name: str :param whattype: time, frequency, voltage or current :type whattype: str :param data: data for the data write :type data: list or numpy.array :param numerical_type: real or complex :type numerical_type: str

CHAPTER

SIX

UTILITIES

PyLTSpice contains some utilities that can be run using the command line.

E.g.: `python -m PyLTSpice.Histogram`

6.1 LTSteps

This module allows to process data generated by LTSpice during simulation. There are three types of files that are handled by this module.

- log files - Files with the extension '.log' that are automatically generated during simulation, and that are normally accessible with the shortcut Ctrl+L after a simulation is ran.Log files are interesting for two reasons.

    1. If .STEP primitives are used, the log file contain the correspondence between the step run and the step value configuration.

    2. If .MEAS primitives are used in the schematic, the log file contains the measurements made on the output data.

  LTSteps.py can be used to retrieve both step and measurement information from log files.

- txt files - Files exported from the Plot File -> Export data as text menu. This file is an text file where data is saved in the text format. The reason to use PyLTSpice instead of another popular lib as pandas, is because the data format when .STEPS are used in the simulation is not not very practical. The PyLTSpice LTSteps.py can be used to reformat the text, so that the run parameter is added to the data as an additional column instead of a table divider. Please Check LTSpiceExport class for more information.

- mout files - Files generated by the Plot File -> Execute .MEAS Script menu. This command allows the user to run predefined .MEAS commands which create a .mout file. A .mout file has the measurement information stored in the following format:

```
Measurement: Vout_rms
step        RMS(V(OUT))     FROM    TO
 1  1.41109 0       0.001
 2  1.40729 0       0.001

Measurement: Vin_rms
  step      RMS(V(IN))      FROM    TO
     1      0.706221        0       0.001
     2      0.704738        0       0.001

Measurement: gain
  step      Vout_rms/Vin_rms
```

(continues on next page)

(continued from previous page)

```
     1      1.99809
     2      1.99689
```

The LTSteps.py can be used directly from a command line by invoking python with the -m option as exemplified below.

```
$ python -m PyLTSpice.LTSteps <path_to_filename>
```

If <*path_to_filename*> is a log file, it will create a file with the same name, but with extension .tout that is a tab separated value (tsv) file, which contains the .STEP and .MEAS information collected.

If <*path_to_filename*> is a txt exported file, it will create a file with the same name, but with extension .tsv a tab separated value (tsv) file, which contains data reformatted with the step number as one of the columns. Please consult the reformat_LTSpice_export() function for more information.

If <*path_to_filename*> is a mout file, it will create a file with the same name, but with extension .tmout that is a tab separated value (tsv) file, which contains the .MEAS information collected, but adding the STEP run information as one of the columns.

If <*path_to_filename*> argument is ommited, the script will automatically search for the newest .log/.txt/.mout file and use it.

6.2 Histogram.py

This module uses matplotlib to plot an histogram of a gaussian distribution and calculates the project n-sigma interval.

The data can either collected from the clipboard or from a text file. Use the following command line text to call this module.

```
python -m PyLTSpice.Histogram [options] [data_file] TRACE
```

The help can be obtained by calling the script without arguments

```
Usage: Histogram.py [options] LOG_FILE TRACE

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -s SIGMA, --sigma=SIGMA
                        Sigma to be used in the distribution fit. Default=3
  -n NBINS, --nbins=NBINS
                        Number of bins to be used in the histogram. Default=20
  -c FILTERS, --condition=FILTERS
                        Filter condition writen in python. More than one
                        expression can be added but each expression should be
                        preceded by -c. EXAMPLE: -c V(N001)>4 -c parameter==1
                        -c  I(V1)<0.5
  -f FORMAT, --format=FORMAT
                        Format string for the X axis. Example: -f %3.4f
  -t TITLE, --title=TITLE
                        Title to appear on the top of the histogram.
  -r RANGE, --range=RANGE
                        Range of the X axis to use for the histogram in the
                        form min:max. Example: -r -1:1
```

(continues on next page)

(continued from previous page)

```
  -C, --clipboard       If the data from the clipboard is to be used.
  -i IMAGEFILE, --image=IMAGEFILE
                        Name of the image File. extension 'png'
```

CHAPTER

SEVEN

VARIA

7.1 RAW File Structure

This section is written to help understand the why the structure of classes is defined as it is.

The RAW file starts with a text preamble that contains information about the names of the traces the order they appear on the binary part and some extra information. In the preamble, the lines are always started by one of the following identifiers:

- Title: => Contains the path of the source .asc file used to make the simulation preceded by *
- Date: => Date when the simulation started
- **Plotname: => Name of the simulation. The known Simulation Types are:**
  - Operation Point
  - DC transfer characteristic
  - AC Analysis
  - Transient Analysis
  - Noise Spectral Density - (V/Hz½ or A/Hz½)
  - Transfer Function
- **Flags: => Flags that are used in this plot. The simulation can have any combination of these flags.**
  - "real" -> The traces in the raw file contain real values. As for exmple on a TRAN simulation.
  - "complex" -> Traces in the raw file contain complex values. As for exmple on an AC simulation.
  - "forward" -> Tells whether the simulation has more than one point. DC transfer characteristic, AC Analysis, Transient Analysis or Noise Spectral Density have the forward flag. Operating Point and Transfer Function don't have this flag activated.
  - "log" -> The preferred plot view of this data is logarithmic.
  - "stepped" -> The simulation had .STEP primitives.
  - "FastAccess" -> Order of the data is changed to speed up access. See Binary section for details.
- No. Variables: => number of variables contained in this dataset. See section below for details.
- No. Points: => number of points per each variable in
- Offset: => when the saving of data started
- Command: => Name of the simulator executable generating this file.
- Backannotation: => Backannotation alerts that occurred during simulation

- Variables: => a list of variable, one per line as described below
- Binary: => Start of the binary section. See section below for details.

7.1.1 Variables List

The variable list contains the list of measurements saved in the raw file. The order of the variables defines how they are stored in the binary section. The format is one variable per line, using the following format:

<tab><ordinal number><tab><measurement><tab><type of measurement>

Here is an example:

```
0   time    time
1   V(n001)    voltage
2   V(n004)    voltage
3   V(n003)    voltage
4   V(n006)    voltage
5   V(adcc)    voltage
6   V(n002)    voltage
7   V(3v3_m)   voltage
8   V(n005)    voltage
9   V(n007)    voltage
10  V(24v_dsp) voltage
11  I(C3)      device_current
12  I(C2)      device_current
13  I(C1)      device_current
14  I(I1)      device_current
15  I(R4)      device_current
16  I(R3)      device_current
17  I(V2)      device_current
18  I(V1)      device_current
19  Ix(u1:+)   subckt_current
20  Ix(u1:-)   subckt_current
```

7.1.2 Binary Section

The binary section of .RAW file is where the data is usually written, unless the user had explicitly specified an ASCII representation. In this case this section is replaced with a "Values" section. LTSpice stores data directly onto the disk during simulation, writing per each time or frequency step the list of values, as exemplified below for a .TRAN simulation.

<timestamp 0><trace1 0><trace2 0><trace3 0>…<traceN 0>

<timestamp 1><trace1 1><trace2 1><trace3 1>…<traceN 1>

<timestamp 2><trace1 2><trace2 2><trace3 2>…<traceN 2>

…

<timestamp T><trace1 T><trace2 T><trace3 T>…<traceN T>

Depending on the type of simulation the type of data changes. On TRAN simulations the timestamp is always stored as 8 bytes float (double) and trace values as a 4 bytes (single). On AC simulations the data is stored in complex format, which includes a real part and an imaginary part, each with 8 bytes. The way we determine the size of the data is dividing the total block size by the number of points, then taking only the integer part.

7.1.3 Fast Access

Once a simulation is done, the user can ask LTSpice to optimize the data structure in such that variables are stored contiguously as illustrated below.

> <timestamp 0><timestamp 1>…<timestamp T>
>
> <trace1 0><trace1 1>…<trace1 T>
>
> <trace2 0><trace2 1>…<trace2 T>
>
> <trace3 0><trace3 1>…<trace3 T>
>
> …
>
> <traceN T><traceN T>…<tranceN T>

This can speed up the data reading. Note that this transformation is not done automatically. Transforming data to Fast Access must be requested by the user. If the transformation is done, it is registered in the Flags: line in the header. PyLTSpice supports both Normal and Fast Access formats

7.2 Installing LTspice

LTspice can be downloaded from : https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html

7.2.1 Windows

LTspice will be installed in C:Program FilesLTCLTspiceXVII

7.2.2 MacOS

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX