

# RFIC Passive Component Design and Simulation in Python

Dušan Grujić, Pavle Jovanović, Dušan Krčum, and Milan Savić

**Abstract** – In this paper we present a Python based RFIC component layout generator - Passive Component Lab and a linear circuit simulator nicSim. These tools can be used for design and optimization of RFIC passive components and circuits. Capability to simulate S parameters in tabulated or State Space representation allows the simulation of linear amplifiers as well, by using transistor S parameters in given biasing conditions. Implementation in Python offers great flexibility, while the underlying speed and capacity of sparse matrix solvers available in a standard Python module SciPy, implemented in C, allows the simulation of real world problems.

**Keywords** – RFIC, Linear simulator, S parameters, Python.

## I. INTRODUCTION

Circuit simulators have been a backbone of IC design industry since the introduction of SPICE [1] forty years ago. Since then many open source and commercial derivatives have emerged, clearly revealing the academic interest and commercial potential. General purpose simulators are designed for simulation speed and capacity, while it is desirable to be flexible and easily extensible. To achieve the speed and capacity, the simulators are usually implemented in C or C++, while the flexibility and extensibility is ensured by the use of complex data structures or object-oriented paradigm to abstract the device models from simulator engine.

In this paper, we propose a different approach, focused on flexibility and extensibility. Speed and capacity is of secondary importance, since the simulator is designed for a specific purpose, and is intended to be light-weight. However, this does not mean that the simulator is not usable in real-world problems.

Python is an interpreted programming language which has lately been embraced by the scientific community. Acceptance of Python can be attributed to several factors: ease of use, large base of scientific modules for numerical and symbolic calculation, quality data plotting and presentation, and last but not least, open source license.

We have designed two software tools in Python – Passive Component Lab for automatic layout generation, and nicSim for linear circuit simulation. Both tools can be used standalone, but they show full potential when used in conjunction with Python based optimizers, as shown in the

Pavle Jovanović and Dušan Krčum are with the School of Electrical Engineering, University of Belgrade, Bul. kralja Aleksandra 73, 11000 Belgrade, Serbia.

Dušan Grujić, Pavle Jovanović, Dušan Krčum, and Milan Savić are with NovelIC, Omladinskih brigada 86p, 11070 Novi Beograd, Serbia. E-mail: {first.last\_name}@novelic.com

proposed Design Flow.

## II. PASSIVE COMPONENT LAB

Passive Component Lab is a collection of Python classes used for automatic generation of integrated passive components, such as inductors and transformers. All components are fully parametrized, and the output can be exported to CAD tools or EM simulators.

Technology information is contained in a separate technology class, allowing the reuse of the same code in all generator classes. The information is read from a textual file containing the information about grid, available layers, connectivity information and basic design rules. Additional information, such as layer conductivity and integrated circuit BEOL cross section, i.e. dielectric layers and their conductivity, can be included for automatic 3D model generation. Simple example of technology file is given below.

```
grid = 0.01

layer TM2 metal
    GDSIINum = 134
    GDSIIType = 0
endlayer

layer TM1 metal
    GDSIINum = 126
    GDSIIType = 0
endlayer

layer TopVia2 via
    GDSIINum = 133
    GDSIIType = 0

    topmet = TM2
    botmet = TM1
    viaEnc = 0.5
    viaSize = 0.9
    viaSpace = 1.06
endlayer
```

The information contained in example technology file is sufficient for generating DRC correct inductors and transformers in top two metals. The example Python code for generating a transformer balun with 4 primary windings and 3 secondary windings is given below.

```
tech = Technology("technology.txt")
bal=balun4x3(tech)
r=300 # Outer radius
w=8   # Winding width
s=3   # Winding spacing
signalLayer = "TM2"
underPassLayer = "TM1"
bal.emVias=True # Merge vias
bal.setupGeometry( r, w, s, signalLayer,
underPassLayer, "octagon")
bal.genGDSII('bal4x3_w8_s3.gds')
```

Example Python code reads the technology information, generates the balun geometry and exports it to GDSII file, which can be imported into any CAD tool or EM simulator. Connectivity information is extracted from technology file, and the vias connecting metals TM1 and TM2 are inserted automatically at appropriate locations. Property `emVias` is set to merge adjacent vias and simplify the model for EM simulation. The generated transformer balun is shown in Fig. 1.

The code similar to the provided one have been successfully used in the development of high performance UWB CMOS transceiver. The transceiver contained a multitude of inductors and transformer baluns, which would be impossible to draw manually. Performance optimization was greatly simplified by automatic layout generation.

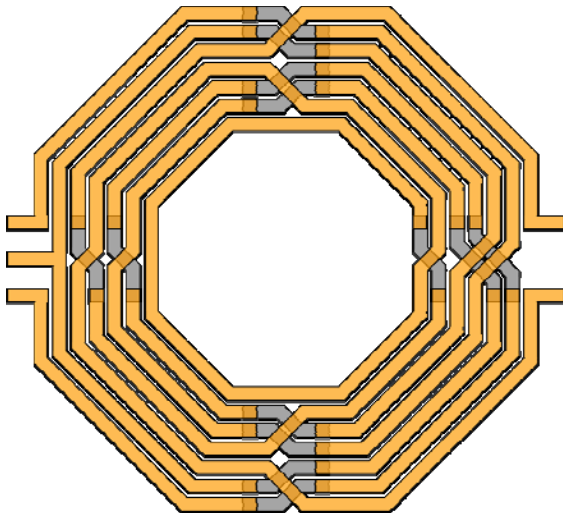


Fig. 1. Transformer balun generated by example code

Automatic layout generator can generate inductors with arbitrary number of windings, with step of 1/4 of a winding, and with arbitrary geometry. Both square and octagonal inductors are supported, in order to have a degree of freedom to choose between maximum inductance for a given area or improved quality factor. Various transformer balun geometries are supported, with transformation ratio from 1:1 to 1:4.

New geometries can be easily added, since the common

geometries are sub-classed and can be easily reused. For example, filling a given area with vias is implemented as a method of a base class, which is inherited by all layout generators. The user has only to specify the coordinates of opposing edges of a rectangle and a via layer; the via drawing method reads the design rules and layer mapping from a technology to produce a DRC correct layout in terms of via size, spacing, and enclosure. Such degree of flexibility allows the creation of very complex parametrized geometries in a matter of minutes.

Additional features, such as predictive models for passive structures are under development. Predictive models for passive structures [2] provide a circuit model for a given passive structure, and can be used for quick performance evaluation and optimization. They can be easily added to layout generator, since the technology file and geometry specification are already present. Being able to generate both predictive circuit models and physical geometry of passive components will make the Passive Component Lab a very powerful tool for every RFIC designer.

### III. NICSIM

Linear circuit simulator `nicSim` is fully implemented in Python, and is intended to be self-contained, with minimum dependency on external libraries. Reducing the dependencies on external libraries and modules makes it easy to install and use, and also light-weight in terms of memory and disk space requirements. The only external dependencies are SciPy [3] and NumPy [4], which are commonly pre-installed on many Linux based systems.

The simulator itself is minimalistic, having only the features that are required for the purpose of simulation and optimization of circuits containing integrated passive components. It uses Modified Nodal Analysis (MNA) [5] formulation for solving the electrical circuit.

Sparse matrix solvers from SciPy are used for solving the system of the form  $Ax=b$ . Direct matrix inversion is not used in solving the system. Instead, a dedicated function for solving the sparse system of linear equations is used. This way the ill-conditioned system can be efficiently handled by element pivoting implemented in a dedicated solver function.

Underlying matrix solvers in SciPy are written in C, so the solver performance is not affected by interpreter nature of Python. This way best of both worlds is utilized: flexibility and rapid development of Python and the sheer speed of C.

The simulator has no frontend netlist parser, since the circuit is built directly from Python. Circuit components, such as resistors, capacitors, inductors, independent and dependent voltage and current sources, are implemented as Python classes. They can be instantiated as any Python object and added to a circuit with a simple call to appropriate method. The circuit itself and simulations are also Python classes, so there is no limit in number of

circuits or simulations, except for the system memory. Currently supported simulation types are DC, AC and S parameter simulation.

Example Python code for S parameter simulation of 3 dB matched attenuator is given below.

```
import nicSim as sim
import numpy as numpy
cir=sim.circuit()
res = sim.resistor
r1 = res('R1', 'N1', 'N2', 17.6)
r2 = res('R2', 'N1', '0', 292.4)
r3 = res('R3', 'N2', '0', 292.4)
p1 = sim.port('P1', 'N1', '0')
p2 = sim.port('P2', 'N2', '0')
cir.addElement([r1, r2, r3, p1, p2])
spsim=sim.sp_analysis(['P1','P2'])
f_list=numpy.arange(1e6,100e6,1e6)
spsim.simulate(cir, f_list)
```

Simulation results can be easily plotted in publication quality with Matplotlib [6], by using the following code.

```
from pylab import *
s11 = spsim.sParam[:,0,0]
s11db = 20*log10(abs(s11))
s21 = spsim.sParam[:,1,0]
s21db = 20*log10(abs(s21))
f = spsim.f
plot(f,s11db)
plot(f,s21db)
```

All components in nicSim are implemented as Python classes. The component class contains the node names, parameters and other data, such as frequency response, and methods for parameter evaluation and matrix stamping. This approach is similar to the one used in SPICE simulator, where the simulator provides the interfaces for matrix stamping and does not go into details of device implementation. Adding new devices to nicSim is easy, since the simulator does not need to be changed – only the new component class with appropriate methods for initialization and matrix stamping has to be designed.

Python is not a strongly typed programming language, so variable type is dynamic, and can change during program execution. This opens an opportunity for exciting and diverse features, which would be very difficult to implement in strongly typed languages, such as C.

One of the most obvious use of dynamic typing is the use of expressions in component parameters. Early SPICE implementations allowed only the use of numerical constants for component parameter values. Newer SPICE versions and commercial simulators allow the use of variables and limited set of expressions for component parameters. Implementation of such feature is by no means simple and easy, since it requires the design of expression parser and evaluator. Python can handle variables and expressions in component parameters in a very simple manner. The type of passed component parameter can be a numerical constant or a string expression. Python built-in

*evaluate* function will evaluate the given expression in a scope of defined variables. Evaluating the expression with variables is nothing new. New is the possibility to pass a function reference as a component parameter value. Provided function will be called each time the component parameter is evaluated, and its return value will be used. This opens a possibility to have an arbitrarily complex function, table look-up or even database or file based component value.

Another example of Python dynamic typing use in component values is symbolic circuit solving. Symbolic circuit solvers have been designed in variety of ways [7], but Python provides a natural way of implementation. To convert a standard linear circuit simulator into symbolic, one would only need to change the matrix stamping and solver routine. Component matrix stamping routine would have to be changed to stamp the string expression instead of numeric value. The solver would have to be replaced by symbolic solver, which are readily available for Python. This approach was used in Ahkab circuit simulator [8], which can solve the circuit both numerically and in a symbolic fashion.

Besides the linear, frequency independent components, nicSim supports  $n$  port S parameter blocks. This feature is important since the simulator is intended for RF passive network design and optimization, and it can include measured or simulated component S parameters. Additionally, S parameter block allows the simulation of linear amplifiers, where the transistors are replaced by S parameters. This way the whole amplifier can be simulated and optimized.

S parameters are usually provided in Touchstone file format, which contains the S matrix elements at a given number of frequencies. Availability of S parameters at discrete frequency points requires the use of interpolation techniques. Another commonly used way of representing S parameters is State Space representation:

$$\begin{aligned} E\dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \quad (1)$$

where  $x$  represents the state vector,  $u$  is the input vector, and the  $y$  is the output vector. Transfer function in frequency domain, which is the S parameter is then given by:

$$S_{ij}(s) = C(sE - A)^{-1}B + D \quad (2)$$

Formulations given in (1) and (2) are commonly used by EM simulators to perform the adaptive frequency sweep, for example Agilent Momentum. As a result, State Space representation of S parameters is available and response can be calculated at any frequency within the valid frequency range.

#### IV. DESIGN FLOW

Design flow using Passive Component Lab and nicSim is shown in Fig. 2. Layout generator and circuit simulator, described in this paper, can be coupled with a user supplied

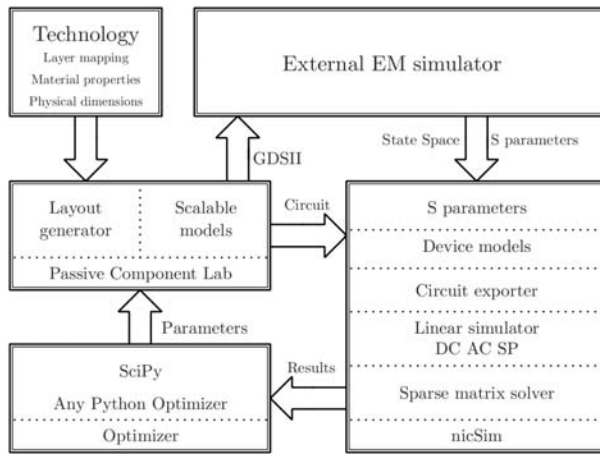


Fig. 2. Design Flow with user supplied optimizer

Python-based optimizer to fit the given circuit model parameters to simulated or measured S parameters of passive component, such as integrated inductor or transformer. Since both Passive Component Lab and nicSim are written in Python, user supplied optimizer can read the simulation data directly from data structures, which eliminates the need for data translation. This is a major advantage, since there are many Python based optimizers to choose from.

Example of circuit parameter optimization to fit the measurements are given in Fig. 3. In this case, the passive component is an inductor, and the assumed circuit model is single- $\pi$  inductor model [9]. Optimized circuit model's Q factor is very close to measured one up to self-resonant frequency.

## V. CONCLUSION

Despite being an interpreted programming language, Python can be used for specialized circuit simulators. Penalty in performance can in some cases be of secondary importance, when flexibility and extensibility are of interest.

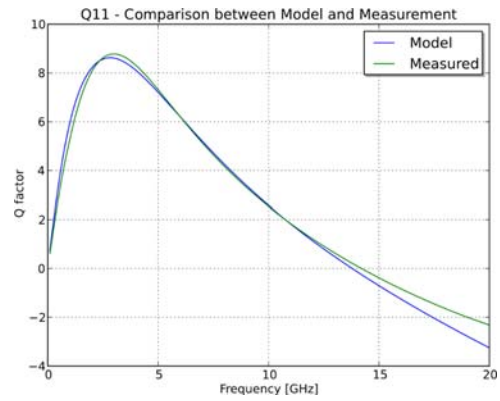


Fig. 3. Optimized circuit model Q factor vs measurements

## REFERENCES

- [1] Nagel, L. W. and Pederson, D. O., "*SPICE (Simulation Program with Integrated Circuit Emphasis)*", Memorandum No. ERL-M382, University of California, Berkeley, Apr. 1973
- [2] Gao, W., and Yu, Z., "*Scalable compact circuit model and synthesis for RF CMOS spiral inductors*", IEEE Transactions on Microwave Theory and Techniques, Vol. 54, No. 3, March 2006., pp 1055-1064.
- [3] SciPy, available at <http://www.scipy.org/>
- [4] NumPy, available at <http://www.numpy.org/>
- [5] Litovski, V., Zwolinski, M., "*VLSI Circuit Simulation and Optimization*", Chapman and Hall, London, 1997.
- [6] Matplotlib, available at <http://matplotlib.org/>
- [7] Đorđević, S., Petković, P., "*A Hierarchical Approach to Large Circuit Symbolic Simulation*", Microelectronics Reliability, 41, (2001), pp. 2941-2049
- [8] Ahkab, available at <http://ahkab.github.io/ahkab/>
- [9] Cao, Y. et al, "*Frequency-independent equivalent-circuit model for on-chip spiral inductors*", IEEE Journal of Solid-State Circuits, Vol. 38, No. 3, March 2003, pp. 419-426.