Python for Microwave and RF Engineers

■ Noyan Kinayman

Python is a powerful programming language for handling engineering and scientific computational tasks efficiently [1]–[5]. Used by companies such as Google and Intel and by organizations including NASA and Los Alamos National Laboratory, it offers an extremely wide selection of tools for tasks such as scientific computing, signal processing, Web site construction, database programming, and graphical user interface (GUI) design. The language is platform independent with most programs running on Linux, Microsoft Windows, or MAC OS virtually unchanged. Python has been distributed as a part of Open Source Initiative, and most versions are General Public License (GPL) compatible. In microwave and radio frequency (RF) engineering, it can be used for numerical programming, automated RF testing, automated monolithic microwave integrated-circuit (MMIC) layout generation, automated netlist generation and simulator sequencing, and other tasks.

A Brief Introduction to Python

Excellent books on Python are available [6]–[9], two special issues on its scientific applications have been published by *IEEE Computing in Science and Engineering* [1], [3], and two comprehensive presentations on scientific computing with Python can be found on the Web [10], [11]. Some of Python's most important features [2], [6] are as follows:

- *object-oriented* from the ground up, although you don't need to use object-oriented principles to write programs in Python
- *interpreted*, so you don't need to compile the Python programs into machine language before running, thereby enabling fast software development and portability
- a clean and coherent *syntax*, with strong indentation rules, documentation strings, and extensive use of name spaces, making Python programs easy to understand and maintain
- a vast number of *library modules*, so you can develop sophisticated applications very quickly
- *mixable*, so you can use Python to call libraries of other languages, e.g., C programs can be called from Python taking the execution speed advantage of C
- uses *dynamic typing*, so you don't need to declare variables before each use, but rather Python keeps track of them and assigns proper types automatically
- a very *wide community*, so you can find accurate and fast help on almost all aspects of the program and the libraries from online resources
- it is *free*! Python comes with extensive standard libraries, and hundreds of third-party libraries are available to users free of charge.

There are also other features of Python that make it a very versatile programming language. For instance, Python has unique built-in data types such as lists and dictionaries. Lists can be thought of as type-agnostic arrays in that you can store any object in them. They can be multidimensional, and elements of a list do not have to be of the same type. Dictionaries, similar to lists, assign a user-defined key to each element, which can then be used to build hash tables. Python also supports various kinds of iterators that enable efficient ways of traversing arrays, which can be used to convert the innermost loops to built-in equivalents, hence increasing execution speed in many cases. Other handy features of Python are the inline lambda and map functions, which can be extremely useful in numerical programming. For more details, see [6] and [7].

Python is typically used from an interactive interpreter. To start, you must first download and install the core libraries and executables, which can be obtained from online repositories. On top of that, the third-party libraries are usually installed. If you are using Microsoft Windows, the installation comes with two standard GUI tools for Python development: IDLE and PythonWin. Table 1 provides version numbers and descriptions of the libraries that the author has mainly used. Although these are usually enough for most microwave and RF software projects, the third-party library support for Python is quite extensive. For instance, in addition to those in Table 1, there are Python libraries for image processing (e.g., PIL), GUI development (e.g., Tkinter), Web programming (e.g., cgi), audio processing (e.g., Audiotools), parallel computing (e.g., mpi4py), symbolic mathematics (e.g., SymPy), and arbitrary-precision mathematics (e.g., mpmath), etc. An exhaustive list can be found in [12].

Noyan Kinayman (noyan.kinayman@ll.mit.edu) is with the MIT Lincoln Laboratory, Group 86—Analog Device Technology, 244 Wood Street, Lexington, Massachusetts, 02420 USA.

Digital Object Identifier 10.1109/MMM.2011.942704
Date of publication: 15 November 2011

Python has unique built-in data types such as lists and dictionaries.

TABLE 1. Author's configuration of Python.

Name	Version	Description
Python [13]	2.6.6	Core Python libraries and executable
Numpy [14]	1.4.1	Matrix and linear-algebra library
Scipy [15]	0.8.0	Numerical analysis and special-functions library
Pyvisa [16]	1.3	Instrumentation library
Gdspy [17]	0.2.4	GDSII layout generation and export library
Matplotlib [18]	1.0.1	Scientific plotting library
MySQLdb [19]	1.2.2	MySQL database interface

Figure 1 provides a pictorial representation of some of the library modules that are commonly used in electrical and electronics engineering.

The newest version of Python is 3.x, which has some important changes from earlier versions, but not all third-party libraries support it at present. For most applications, Python 2.5 or greater, which has broader third-party library support, is adequate.

Examples of commonly used numerical operations are given for reference in Table 2. Note that there are two ways of representing matrices in Python. The array command demonstrated in Table 2 has greater flexibility since it supports multidimensional matrices, but it requires the dot function for the matrix multiplication, which may be cumbersome if you have too many nested matrix operations (regular multiplication operator is reserved for element-wise multiplication) [20]. It should be noted that there are sometimes overlaps between different Python numerical libraries. In certain cases, these overlapped commands call the same underlying algorithms. In other cases, the implementation of the actual algorithm can be slightly different. One can confirm the details of such cases from the documentation of each library.

At this point, it would be helpful to compare Python and other domain-specific languages (DSLs) commonly used in engineering and science. A DSL is a computer language designed to perform specific computing tasks (e.g., matrix algebra) as opposed to a general-purpose programming language (C++, Java, Python, etc.). As far as general-purpose programming is concerned, Python is simply a better choice compared to a DSL because of the

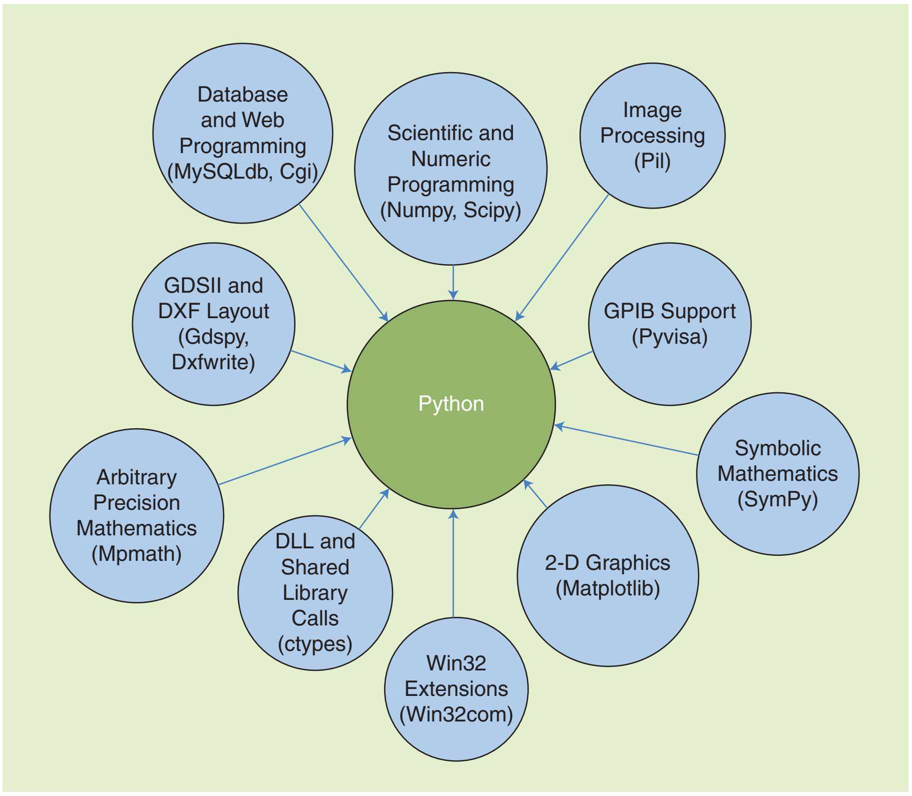

Database and Web Programming (MySQLdb, Cgi)
Scientific and Numeric Programming (Numpy, Scipy)
Image Processing (Pil)
GDSII and DXF Layout (Gdspy, Dxfwrite)
Python
GPIB Support (Pyvisa)
Symbolic Mathematics (SymPy)
Arbitrary Precision Mathematics (Mpmath)
DLL and Shared Library Calls (ctypes)
2-D Graphics (Matplotlib)
Win32 Extensions (Win32com)

Figure 1. *Python library modules that are commonly used in electrical and electronics engineering. Other modules may be available depending on the application.*

aforementioned features. It is a serious contender in numerical and scientific programming as well, thanks to extensive third-party numerical libraries. Last but not least, obtaining Python is free; therefore one does not need to use costly DSL licenses for every computing need. In short, Python can be employed in many engineering projects to generate fast, accurate, and high-quality code.

The following example programs demonstrate typical usage of Python in four microwave and RF engineering application areas. In order to use the programs, user should first install the following Python modules from Table 1: Numpy, Scipy, Pyvisa, Gdspy, and Matplotlib. Then, the programs shown in Figures 2–5 should be copied to the directory where the user starts the Python interpreter. Note that the examples don't include error checking and may not be the most optimal implementations but can easily be extended to add advanced features and error checking.

Numerical Programming

The shown Python code provides an implementation of a vector fitting [22] algorithm that can be used to determine poles and residues of a linear system for model-based parameter estimation (MBPE). In modeling of passive structures, MBPE is advantageous compared to equivalent lumped-circuit model extraction because it can be broadband and the resultant model generation can be automated easily, which will also be demonstrated in the next section. The other commonly used algorithm for MBPE is the frequency-domain Prony's method [21]. Vector fitting is typically more robust compared to the frequency-domain Prony's method for broadband data because the resultant matrix solution is less ill-conditioned. In addition, it allows an iterative way to improve the initial pole estimates. Hence, it has been used extensively in the modeling of large RF and microwave passive structures (e.g., electronic packages). Figure 6 shows flow-chart of the vector fitting algorithm in its basic form.

To demonstrate the use of the program, let's assume that we have Y-parameters of a MMIC spiral inductor stored in a file. The inductor parameters used in this example are $w = 10\ \mu$m (line width), $s = 5\ \mu$m (line gap), $n = 6.5$ (number of turns), $r = 30\ \mu$m (inner radius), $h = 250\ \mu$m (substrate height), and $\varepsilon_r = 12.9$ (substrate relative permittivity). Then, one can use the following command sequence from the Python shell to find the poles and residues of the inductor:

Python is typically used from an interactive interpreter.

```
>> import mbpe
>> (alpha, k, error, i) = mbpe.
vectorfit('spiral.csv', 9, 2, (0,0))
```

The first command imports the program into the Python session; the second command calls the function to calculate the poles and residues. The first parameter in the function call simply tells the filename

TABLE 2. Examples of numerical operations in Python.

Operation	Equivalent Python Command Sequence

(1 1 2i).(3 1 4i)	`>> j = complex(0.0, 1.0)`
	`>> (1+2*j)*(3+4*j)`

sin(π/4)	`>> import scipy`
	`>> scipy.sin(scipy.pi/4)`

$\int_1^2 (x^2+x+1)dx$	`>> from scipy import integrate`
	`>> f = lambda x: x*x+x+1`
	`>> integrate.quad(f, 1, 2)`

$min\{\vert -x^2-x+1\vert\}$	`>> from scipy import optimize`
	`>> f = lambda x: abs(-x*x-x+1)`
	`>> optimize.fmin(f, 0.0)`

$roots\{-x^2-x+1\}$	`>> import scipy`
	`>> scipy.roots([-1, -1, 1])`

$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$	`>> import numpy`
	`>> numpy.array([[1, 2],[3, 4]])`

$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}$	`>> import numpy`
	`>> A = numpy.array([[1, 2],[3, 4]])`
	`>> B = numpy.array([[5, 6],[7, 8]])`
	`>> numpy.dot(A,B)`

$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \cdot \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}$	`>> import numpy`
	`>> A = numpy.array([[1, 2],[3, 4]])`
	`>> B = numpy.array([[5, 6],[7, 8]])`
	`>> A*B`

$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \mathbf{x} = \begin{bmatrix}5\\6\end{bmatrix}$	`>> from scipy import linalg`
	`>> import numpy`
	`>> A = numpy.array([[1, 2],[3, 4]])`
	`>> B = numpy.array([[5, 6]])`
	`>> linalg.solve(A,B)`

The program demonstrates an application of vector fitting and can be extended to include passivity enforcement as well.

where the data are stored (the data should be comma-separated values and the first row is assumed to be a comment). The second parameter indicates how many poles we want to extract (in this case nine poles). The third parameter indicates the number of ports. And the fourth parameter tells which Y parameter

```python
import numpy
import scipy

from numpy import linalg

j = complex(0.0, 1.0)

def vectorfit(filename, n, m, param=(0,0)):
    data = numpy.loadtxt(filename, skiprows=1, delimiter=',') # read the Y-parameters
    freq = 2*scipy.pi*numpy.reshape(data[0:,0], (len(data[0:,0]),1))
    ypar = []
    for i in range(0, len(freq)):
        a = numpy.reshape(data[i,1::2], (m,m))
        b = numpy.reshape(data[i,2::2], (m,m))
        ypar.append(a+j*b)

    freq = numpy.concatenate((-freq[-1::-1], freq)) # enforce complex-conjugate poles
    ypar = numpy.concatenate((numpy.conjugate(ypar[-1::-1]), ypar))

    alpha = []
    for i in range(0, n): # generate estimates for the poles
        b = freq[0][0]+i*(freq[-1][0]-freq[0][0])/(n-1)
        a = b/100
        alpha.append(-a+j*b)

    conv = 1.0
    q = 0
    while conv > 1.0E-6 and q < 100:
        B = numpy.reshape(ypar[0::,param[0],param[1]], (len(freq),1))
        A = numpy.empty((len(freq), 2*n), dtype=complex)
        for i in range(0, n):
            A[0:len(freq),i] = (1/(j*freq-alpha[i]))[0:len(freq),0]
        for i in range(0, n):
            A[0:len(freq),i+n] = (-B/(j*freq-alpha[i]))[0:len(freq),0]

        x = numpy.dot(linalg.pinv(A, rcond=1E-15), B) # solve the augmented model

        H = numpy.diag(alpha)-numpy.dot(numpy.ones((n, 1)), x[n::].transpose())
        z = linalg.eig(H)[0] # find the new pole estimates

        for i in range(0, len(z)):
            if z[i].real > 0: # flip the right-hand poles
                z[i] = -z[i].real+j*z[i].imag
            if scipy.absolute(z[i].imag) < 1.0: # remove small imaginary parts
                z[i] = z[i].real

        conv = linalg.norm(alpha-z)
        alpha = z # update the poles with the new estimates
        q = q + 1

    k = numpy.empty((m, m, n), dtype=complex)
    for i in range(0, m):
        for l in range(0, m): # find the residues
            B = numpy.reshape(ypar[0::,i,l], (len(freq),1))
            A = numpy.empty((len(freq), n), dtype=complex)
            for p in range(0, n):
                A[0:len(freq),p] = (1/(j*freq-alpha[p]))[0:len(freq),0]

            k[i, l, 0:n] = numpy.dot(linalg.pinv(A, rcond=1E-15), B)[0:n,0]

    return (alpha, k, conv, q)
```

Figure 2. *Python script (mbpe.py) for the vector fitting algorithm.*

should be used for the pole extraction. Note that we are assuming that all the Y-parameters of the passive circuit share the same poles (although they may have different residues). Therefore, we should pick one of the entries in the Y-matrix to calculate the poles first before determining the residues for each parameter. In the above example, we are telling the program to use $Y_{0,0}$ to determine the poles of the inductor. The program then returns the found poles, residues, convergence of the poles, and total number of poles iterations. One should inspect the returned error to ensure that the convergence of the poles is satisfactory. Once we have the poles and residues, we can use the resultant rational model to generate a broadband circuit model of the inductor. Note that we haven't employed any rigorous passivity enforcement in the program. Nevertheless, the program demonstrates an

Another interesting application of Python in microwave and RF engineering is automatic layout generation.

```python
import numpy
import scipy

j = complex(0.0, 1.0)
pi = scipy.pi

def pr(filename, p, r, m, unique):
    fid = open(filename, 'w')
    fid.write('globalnode 0\n')
    fid.write('Options:OPT1 ResourceUsage=yes UseNutmegFormat=no ASCII_Rawfile=yes ')
    fid.write('Verbose=yes TopDesignName="spiral" ')
    fid.write('GiveAllWarnings=yes MaxWarnings=30\n')
    fid.write('S_Param:SP1 CalcS=yes CalcY=yes CalcZ=no CalcGroupDelay=yes ')
    fid.write('GroupDelayAperture=1e-4 ')
    fid.write('SweepVar="freq" SweepPlan="SP1_stim" ')
    fid.write('FreqConversion=no \n')
    fid.write('SweepPlan:SP1_stim ')
    fid.write('Start=%.6f GHz Stop=%.6f GHz Step=%.6f GHz\n' % (1.0, 100.0, 0.1))

    for i in range(0, m): # add the ports
        fid.write('Port:Term%02d %d %d Num=%02d Z=50 Ohm Noise=yes\n' %\
                  (i+1, i+1, 0, i+1))

    fid.write('PR_MODEL:PR_MODEL%02d %s\n' %\
              (0, ''.join(['%d ' % (x+1) for x in range(0, m)])))
    fid.write('define PR_MODEL (%s)\n' %\
              ''.join(['%d ' % (x+1) for x in range(0, m)]))

    pol = ''
    for k in range(0, len(p)): # prepare the pole list
        if k in unique:
            pol = pol+'%.8e,%.8e,' % (-p[k].real*1.0E9, -p[k].imag*1.0E9/2/pi)
    pol = pol[0:-1]

    for i in range(0, m):
        for l in range(0, m):
            res = ''
            for k in range(0, len(p)): # prepare the residue list
                if k in unique:
                    res = res+'%.8e,%.8e,' %\
                          (r[i][l][k].real*1.0E9, r[i][l][k].imag*1.0E9)
            res = res[0:-1]

            fid.write('VCCS_PZR:SRC%02d %d %d %d %d ' % (i*m+l, l+1, 0, i+1, 0))
            fid.write('Poles=list(1.0,%s) Residues=list(0.0,0.0,%s) ' % (pol, res))
            fid.write('Scale=%.6e\n' % (1.0))

        fid.write('R:R%02d %d %d R=1.0E6 Ohm Noise=no\n' % (i, i+1, 0))

    fid.write('end PR_MODEL\n')

    fid.close()
```

Figure 3. *Python script (netlist.py) for Agilent ADS netlist generation.*

With the use of third-party libraries, Python can easily communicate with RF instruments, enabling fast and easy development of automated RF test software.

application of vector fitting and can be extended to include passivity enforcement as well. A good discussion on the topic can be found in [23].

This example is a simple demonstration of Python in numerical programming in microwave and RF engineering. It shows many important aspects of the language, such as matrix creation, matrix slicing, linear algebra, iterators, and lists. One can see that the third-party libraries (in this case `numpy` and `scipy`) make Python codes very compact and easy to follow.

Automated Netlist Generation

The previous example showed how to extract poles and residues (i.e., a rational model) of a passive structure using the vector fitting algorithm. This present example demonstrates automated netlist generation from the extracted rational model. Automated netlist generation from a rational model can be very helpful because it allows a systematic way of modeling passive structures that are commonly used in MMIC design. Note that the example program generates the netlist syntax of Agilent Advanced Design System (ADS) [25, p. 211], but it can easily be modified to support other netlist formats.

Let's assume that we have found the poles and residues of an MMIC spiral inductor using the Vector Fitting algorithm given in the previous section. In order to generate an equivalent circuit model for the same inductor, one can then use the following command sequence:

```
>> import netlist
>> netlist.pr('spiral.net', alpha,
k, 2, [0,2,3,6,7,8])
```

The first parameter in the function call gives the filename where the netlist will be saved. The second and third parameters are the pole and residue arrays, respectively. The fourth parameter indicates the number of ports. Finally, the last parameter provides a list showing which poles should be included in the netlist

```
import math
import numpy
import scipy
import gdspy

from scipy import interpolate

j = complex(0.0, 1.0)
pi = math.pi

def spiral(filename, w, g, ri, m):
    a = (w+g)/(2*pi)
    b = ri/a
    n = 1.0

    path = lambda t: a*scipy.power(t+b, n)*numpy.array([scipy.cos(t), scipy.sin(t)])

    t = numpy.linspace(0.0, m*2*pi, 1001) # generate the sample points

    x, y = path(t) # generate the x-y coordinates of the spiral

    path_x = interpolate.InterpolatedUnivariateSpline(t, x)
    path_y = interpolate.InterpolatedUnivariateSpline(t, y)

    main_cell = gdspy.Cell('main') # creates the main GDS cell

    f = lambda t: (path_x(m*2*pi*t), path_y(m*2*pi*t)) # parameterize the spiral

    path = gdspy.Path(w, initial_point=(0, 0), number_of_paths=1) # start of the path
    path.parametric(2, f, final_width=w, number_of_evaluations=601) # draw the spiral
    main_cell.add(path) # add the spiral to the main cell

    fid = open(filename, 'wb')
    gdspy.gds_print(fid, [main_cell], unit=1.0E-6, precision=1.0e-9) # export the file
    fid.close()
```

Figure 4. *Python script (layout.py) for GDSII layout generation.*

generation. Owing to the details of the simulation tool employed, only one of the extracted complex-conjugate pole pairs should be passed to the netlist generation. Note that the pole indices given in the list above are specific for a given Y-parameter matrix and will be different for different structures.

The algorithm essentially uses voltage-controlled current sources (VCCSs) [26, p. 25] to realize the Y-matrix of the circuit as shown in Figure 7. In ADS, the generated netlist can be simulated from the command line by using the `hpeesofsim` command [25, p. 209]. After the simulation, the results will be put in a `spiral.ds` file, which subsequently can be plotted again in ADS. It is important to note that the program assumes that a VCCS, which accepts a complex transfer function in the form of poles and residues, is available

```
import math
import numpy
import scipy
import visa

from matplotlib import pyplot

SG = visa.instrument('GPIB::19') # gets the instrument handle
SG.write('*RST') # sends reset
SG.write('*CLS') # sends clear
SG.write('SOUR:FREQ:CW 10.0 GHz') # output frequency
SG.write('SOUR:POW:LEV:IMM:AMPL -100 dBm') # output amplitude
SG.write('SOUR:POW:ALC:STAT ON') # automatic level control on
SG.write('OUTP:STAT ON') # RF output on

SA = visa.instrument('GPIB::17') # gets the instrument handle
SA.write('*RST') # sends reset
SA.write('*CLS') # sends clear
SA.write('INST:SEL SA') # selects spectrum-analyzer
SA.write('SENS:FREQ:CENT 10.0 GHz') # sets center frequency
SA.write('SENS:FREQ:SPAN 1.0 MHz') # sets frequency span
SA.write('SENS:BAND:RES 3.0 KHz') # sets resolution bandwidth
SA.write('DISP:WIND:TRAC:Y:SCAL:RLEV 30 dBm') # sets reference level
SA.write('INIT:CONT OFF') # manual trigger

power_start = -50.0 # start power
power_stop = 20.0 # stop power
points = 40 # number of steps

x = []
y = []

for i in range(0, points):
    input_power = power_start+(power_stop-power_start)*i/(points-1)

    SG.write('SOUR:POW:LEV:IMM:AMPL %.3f dBm' % (input_power))
    SG.write('*OPC?')
    status = SG.read()

    SA.write('INIT:IMM')
    SA.write('*WAI')
    SA.write('CALC:MARK1:MAX')
    SA.write('CALC:MARK1:Y?')
    output_power = float(SA.read())

    x.append(input_power)
    y.append(output_power)

    print '%d, %.3f, %.3f' % (i, x[-1], y[-1])

fig = pyplot.figure(1)
axis = fig.add_subplot(111)
axis.plot(x, y)
axis.set_xlabel('Input Power [dBm]')
axis.set_ylabel('Output Power [dBm]')
fig.show()
```

Figure 5. *Python script (rftest.py) for automated amplifier test over GPIB.*

The application areas of Python, however, are vast. The strong logical flow of the language makes it a perfect candidate for a variety of engineering tasks.

in the simulator's netlist syntax. Other algorithms are available that can generate a state-space model [24] and hence do not rely on the availability of the VCCS defined by a complex transfer function. A state-space implementation also provides more rigorous stability checks and model reduction, which are very important topics in modeling electrically large interconnects and electronic packages.

This example demonstrates the scripting capabilities of Python to automatically generate netlist file of a passive structure from the given transfer function. Python has a very rich collection of string methods, iterators, and other utilities, which are extremely helpful in scripting [6], [7].

Automated Layout Generation

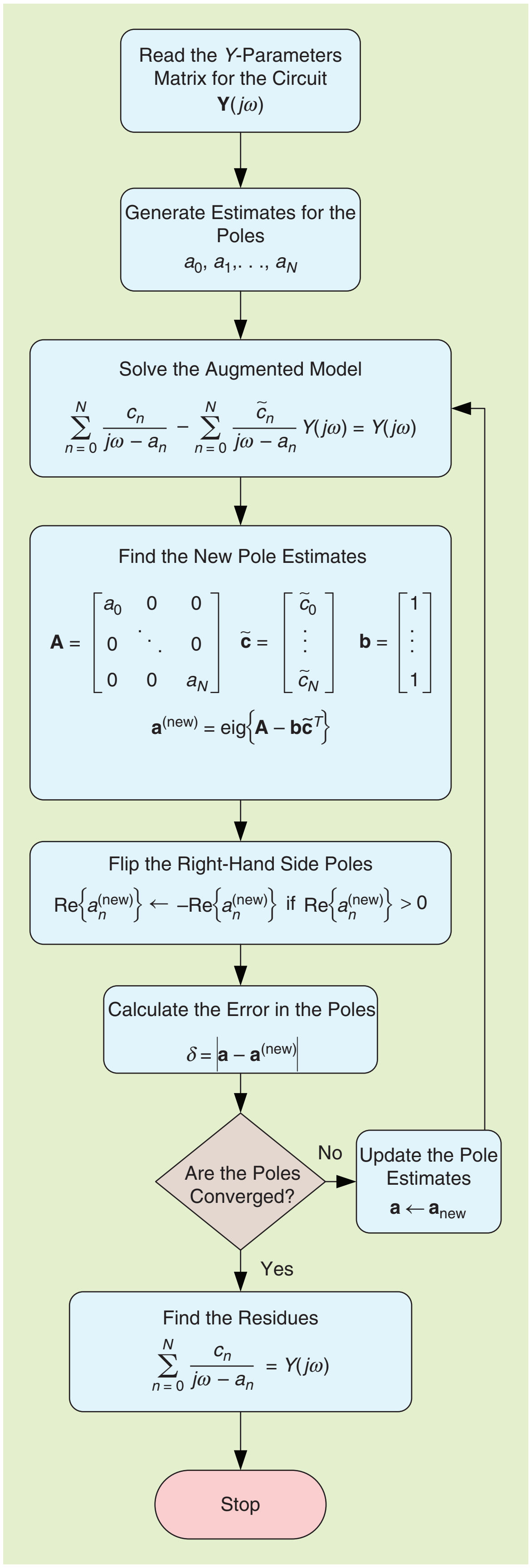
Another interesting application of Python in microwave and RF engineering is automatic layout generation. So far, we have modeled a MMIC spiral-inductor and generated an equivalent circuit from the extracted rational model. We now generate a GDSII layout of the spiral-inductor from given parameters, which can be imported into a MMIC layout tool to be used in a circuit.

In order to generate the GDSII layout of a round spiral-inductor, we can use the following command sequence (note that the program does not draw the air-bridge connection and should be added separately by the user):

```
>> import layout
>> layout.spiral('spiral.gds', 10,
5, 30, 6.5).
```

The first parameter in the function call is the name of the GDSII file where the geometry will be written. The second and third parameters give the linewidth and gap of the spiral-inductor, respectively. The fourth parameter is the inner radius, and the last parameter gives the number of turns. All dimensions must in microns. Note that the program defines the path of the spiral as a parametric function. Therefore, it is quite flexible; one can draw any shape of spiral (e.g., an oval) as long as its path can be defined as a parametric function.

This example demonstrates the use of Python to generate GDSII layouts. What is remarkable about this example is the ease of generating a parametric path (defining the spiral) and then exporting it in GDSII format with minimal effort. The approach can be used to automatically generate layouts for 2-1/



Read the Y-Parameters Matrix for the Circuit $\mathbf{Y}(j\omega)$
Generate Estimates for the Poles $a_0, a_1, \ldots, a_N$
Solve the Augmented Model $\sum_{n=0}^{N} \frac{c_n}{j\omega - a_n} - \sum_{n=0}^{N} \frac{\tilde{c}_n}{j\omega - a_n} Y(j\omega) = Y(j\omega)$
Find the New Pole Estimates $\mathbf{A} = \begin{bmatrix} a_0 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_N \end{bmatrix}$ $\tilde{\mathbf{c}} = \begin{bmatrix} \tilde{c}_0 \\ \vdots \\ \tilde{c}_N \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$ $\mathbf{a}^{(new)} = \text{eig}\{\mathbf{A} - \mathbf{b}\tilde{\mathbf{c}}^T\}$
Flip the Right-Hand Side Poles $\text{Re}\{a_n^{(new)}\} \leftarrow -\text{Re}\{a_n^{(new)}\}$ if $\text{Re}\{a_n^{(new)}\} > 0$
Calculate the Error in the Poles $\delta = |\mathbf{a} - \mathbf{a}^{(new)}|$
Are the Poles Converged?
No
Update the Pole Estimates $\mathbf{a} \leftarrow \mathbf{a}_{new}$
Yes
Find the Residues $\sum_{n=0}^{N} \frac{c_n}{j\omega - a_n} = Y(j\omega)$
Stop

Figure 6. *Flow-chart of the vector fitting algorithm in its basic form. Note that a rigorous enforcement of stability and passivity may be necessary depending on the problem.*

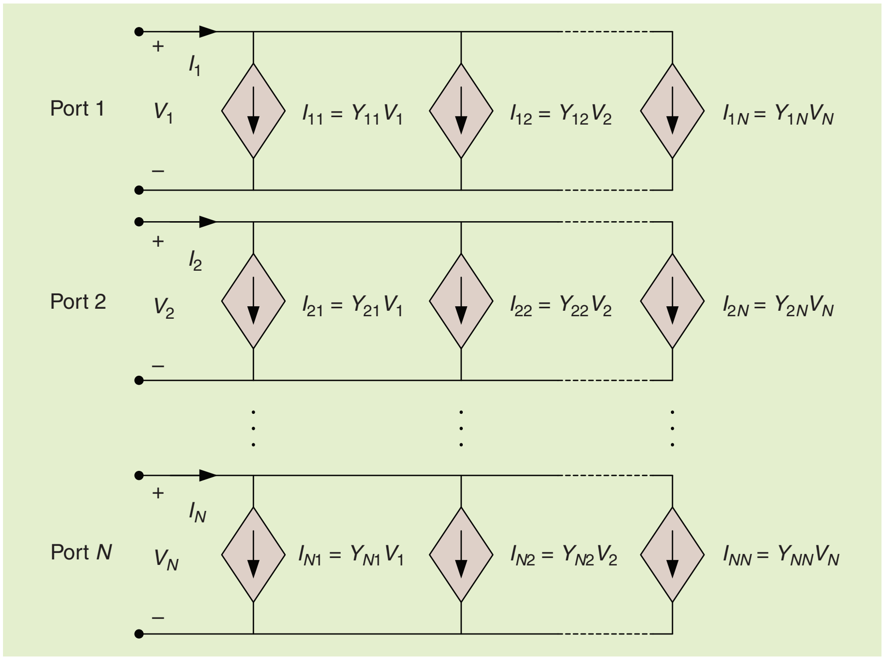


Figure 7. *Simple circuit modeling of passive N-port from the Y-parameters using controlled sources.*

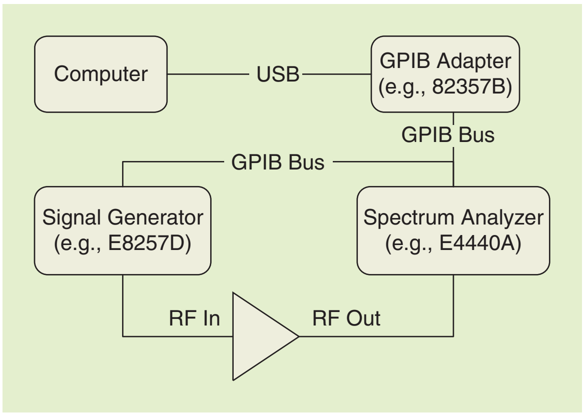
two-dimensional (2-D) electromagnetic simulators, which can be quite useful for structures containing repetitive objects (e.g., metamaterials). As another example, one can use a similar approach to generate, let's say, primitives for a family of logic gates and then automatically generate combinatorial logic circuit layouts from a truth table for integrated circuit (IC) design.

RF Instrumentation and Control

With the use of third-party libraries, Python can easily communicate with RF instruments, enabling fast and easy development of automated RF test software. As a matter of fact, by combining this approach with the object-oriented philosophy to promote code reuse, test software for various RF tests and instruments can be developed very efficiently. Here, we demonstrate simple test software written in Python that generates the power compression characteristics of an amplifier. Note that, in order to run this program, general-purpose interface bus (GPIB) hardware (e.g., Agilent 82357B) must be installed on the computer so that it can transfer commands to the GPIB bus. We also assume a signal generator (Agilent E8257D) and a spectrum analyzer (Agilent E4440A) are available and connected to the device under test (DUT) as shown in Figure 8.

In order to run the test program, one can use the following simple command from the Python shell (note that we use function calls in the previous examples, and in this case the program is simply being executed as a script):

```
>> execfile('rftest.py')
```



Computer
USB
GPIB Adapter (e.g., 82357B)
GPIB Bus
GPIB Bus
Signal Generator (e.g., E8257D)
Spectrum Analyzer (e.g., E4440A)
RF In
RF Out

Figure 8. *Block diagram of the amplifier test circuit. Note that one can easily control RF instruments over GPIB or Ethernet by using Python.*

The first part of the program creates the device objects and sends simple initialization sequences. One can also see a definition of the basic parameters of the test just after the instrument initialization. After initializing the instruments and defining the test parameters, we simply sweep the RF power in a loop and collect results. Finally, we show the results in a plot using the Matplotlib, which is the 2-D graphing library.

This example has shown an instrumentation and control application using Python. Plotting x-y graphs is also demonstrated. The program can easily be generalized to incorporate additional instruments and more elaborate test procedures. Note that one can combine this approach with the database interfaces of Python to create very powerful and flexible RF test programs.

Conclusion

Four examples have been chosen to demonstrate unique features and third-party libraries for use in microwave and RF engineering. The application areas of Python, however, are vast. The strong logical flow of the language makes it a perfect candidate for a variety of engineering tasks. Because of tremendous third-party library support, Python enables rapid application development (RAD). Some topics not included here, but very useful for microwave and RF engineering, are the database applications and GUI development. Interested readers should refer to the literature for more information on these topics [7]. Python comes essentially free-of-charge and with the support of thousands of enthusiastic users and experts around the world.

Acknowledgement

This work is sponsored by the Department of the Air Force under Air Force Contract number FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

References

[1] P. F. Dubois, "Python: Batteries included," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 7–9, May/June 2007.

[2] T. E. Oliphant, "Python for scientific programming," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 10–20, May/June 2007.

[3] K. J. Millman and M. Aivazis, "Python for scientists and engineers," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 9–12, Mar./Apr. 2011.

[4] R. Lytle, "The numeric Python EM project," *IEEE Antennas Propagat. Mag.*, vol. 44, no. 6, p. 146, Feb. 2002.

[5] J. P. Swartz, "A python toolbox for computing solutions to canonical problems in electromagnetics," *IEEE Antennas Propagat. Mag.*, vol. 48, no. 3, pp. 78–81, Feb. 2006.

[6] M. Lutz, *Learning Python, 3rd ed.* CA: O'Reilly, 2008.

[7] M. Lutz, *Programming Python, 3rd ed.* CA: O'Reilly, 2006.

[8] W. J. Chun, *Core Python Programming, 2nd ed.* Englewood Cliffs, NJ: Prentice-Hall, 2007.

[9] A. Martelli, A. M. Ravenscroft, and D. Ascher, *Python Cookbook, 2nd ed.* CA: O'Reilly, 2005.

[10] E. Jones and T. Oliphant. (2004, Oct. 24). Python Talk 1. [Online]. Available: http://nanohub.org/site/archive/2004.10.24-Python_talk1.pdf

[11] E. Jones and T. Oliphant. (2004, Oct. 24). Python Talk 2. [Online]. Available: http://nanohub.org/site/archive/2004.10.24-Python_talk2.pdf

[12] Python. [Online]. Available: http://pypi.python.org/pypi/

[13] Python. Downloads. [Online]. Available: http://www.python.org/download/releases/2.6.6/

[14] Python. Numpy software. [Online]. Available: http://sourceforge.net/projects/numpy/files/NumPy/1.4.1/

[15] Python. Scipy software. [Online]. Available: http://sourceforge.net/projects/scipy/files/scipy/0.8.0/

[16] T. Bronger and G. Thalhamme. PyVISA. [Online]. Available: http://sourceforge.net/projects/pyvisa/files/PyVISA/1.3/

[17] [Online]. Available: http://sourceforge.net/projects/gdspy/files/gdspy/gdspy-0.2.4/

[18] [Online]. Available: http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.0.1/

[19] A. Dustman. mysql. [Online]. Available: http://sourceforge.net/projects/mysqlpython/

[20] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, Mar./Apr. 2011.

[21] J. N. Brittingham, E. K. Miller, and J. L. Willows, "Pole extraction from real-frequency information," *Proc. IEEE*, vol. 68, no. 2, pp. 263–273, Feb. 1980.

[22] B. Gustavsen and A. Semlyen, "Rational approximation of frequency domain responses by vector fitting," *IEEE Trans. Power Delivery*, vol. 14. no. 3, pp. 1052–1061, July 1999.

[23] P. Triverio, S. Grivet-Talocia, M. S. Nakhla, F. G. Canavero, and R. Achar, "Stability, causality, and passivity in electrical interconnect models," *IEEE Trans. Advanced Packag.*, vol. 30, no. 4, pp. 795–808, Nov. 2007.

[24] A. C. Cangellaris, M. Celik, S. Pasha, and L. Zhao, "Electromagnetic model order reduction for system-level modeling," *IEEE Trans. Microwave Theory Tech.*, vol. 47, no. 6, pp. 840–850, June 1999.

[25] Agilent Technologies, *ADS 2009 Update 1—Using Circuit Simulators*, Oct. 2009.

[26] Agilent Technologies, *ADS 2009 Update 1—Sources*, Oct. 2009.

Backscatter *(continued from page 24)*

minimum health and safety requirements regarding the exposure of workers to the risks arising from physical agents (electromagnetic fields) (18th individual Directive within the meaning of Article 16(1) of Directive 89/391/EEC).

[2] J. P. Reilly, "An analysis of differences in the low frequency electric and magnetic field exposure standards of ICES and ICNIRP," *Health Phys.*, vol. 89, no. 1, pp. 71–80, 2005.

[3] ICNIRP (International Commission on Non-Ionizing Radiation Protection), "Guidelines for limiting exposure to time-varying electric, magnetic and electromagnetic fields (up to 300 GHz)," *Health Phys.*, vol. 74, no. 4, pp. 494–522, 1998.

[4] IEEE Standard for Safety Levels with Respect to Human Exposure to Electromagnetic Fields, 0–3 kHz, IEEE Standard C95.6-2002 (R2007).

[5] ICNIRP (International Commission on Non-Ionizing Radiation Protection), "Guidelines for limiting exposure to time-varying electric and magnetic fields (1 Hz to 100 kHz)," *Health Phys.*, vol. 99, no. 6, pp. 818–836, 2010.