Signal Processing and Communications: Teaching and Research Using IPython Notebook

Mark Wickert‡*

https://www.youtube.com/watch?v=xWREmn7EajM

Abstract—This paper will take the audience through the story of how an electrical and computer engineering faculty member has come to embrace Python, in particular IPython Notebook (IPython kernel for Jupyter), as an analysis and simulation tool for both teaching and research in signal processing and communications. Legacy tools such as MATLAB are well established (entrenched) in this discipline, but engineers need to be aware of alternatives, especially in the case of Python where there is such a vibrant community of developers. In this paper case studies will also be used to describe domain specific code modules that are being developed to support both lecture and lab oriented courses going through the conversion from MATLAB to Python. These modules in particular augment `scipy.signal` in a very positive way and enable rapid prototyping of communications and signal processing algorithms. Both student and industry team members in subcontract work, have responded favorably to the use of Python as an engineering problem solving platform. In teaching, IPython notebooks are used to augment lecture material with live calculations and simulations. These same notebooks are then placed on the course Web Site so students can download and *tinker* on their own. This activity also encourages learning more about the language core and Numpy, relative to MATLAB. The students quickly mature and are able to turn in homework solutions and complete computer simulation projects, all in the notebook. Rendering notebooks to PDF via LaTeX is also quite popular. The next step is to get other signals and systems faculty involved.

Index Terms—numerical computing, signal processing, communications systems, system modeling

Introduction

This journey into Python for electrical engineering problem solving began with the writing of the book *Signals and Systems for Dummies* [Wic2013], published summer 2013. This book features the use of Python (`Pylab`) to bring life to the mathematics behind signals and systems theory. Using Python in the Dummies book is done to make it easy for all readers of the book to develop their signals and system problem solving skills, without additional software tools investment. Additionally, the provided custom code module `ssd.py` [ssd], which is built on top of `numpy`, `matplotlib`, and `scipy.signal`, makes it easy to work and extend the examples found in the book. Engineers love to visualize their work with plots of various types. All of the plots in the book are created using Python, specifically `matplotlib`.

* *Corresponding author: mwickert@uccs.edu*
‡ *University of Colorado Colorado Springs*

The next phase of the journey, focuses on the research and development side of signals and systems work. During a recent sabbatical[1] Python and IPython notebook (IPython kernel for Jupyter) served as the primary digital signal processing modeling tools on three different projects. Initially it was not clear which tool would be best, but following discussions with co-workers[2] Python seemed to be the right choice. Note, for the most part, the analysis team was new to Python, all of us having spent many years using MATLAB/Octave [MATLAB]/[Octave]. A nice motivating factor is that Python is already in the workflow in the real-time DSP platform used by the company.

The third and current phase of the Python transformation began at the start of the 2014-2015 academic year. The move was made to push out Python to the students, via the IPython Notebook, in five courses: digital signal processing, digital communications, analog communications, statistical signal processing, and real-time signal processing. Four of the courses are traditional lecture format, while the fifth is very hands-on lab oriented, involving embedded systems programming and hardware interfacing. IPython Notebook works very well for writing lab reports, and easily allows theoretical and experimental results to be integrated. A notebook interface is not a new concept in scientific computing tool sets[3]. Both of these tools are very powerful for specific problem classes.

The remainder of this paper is organized into the following sections: arriving at Python for communications and signal processing modeling, describing IPython notebook usage, case studies, and conclusions.

Arriving at Python for Communications and Signal Processing Modeling

About three years ago while working on a study contract for a small business, I started investigating the use of open-source alternatives over MATLAB. I initially homed in on using Octave [Octave] for its syntax compatibility with MATLAB. Later I started to explore Python and became fascinated by the ease of use offered by the IPython (QT) console and the high quality of `matplotlib` 2D plots. The full power of Python/IPython

1. Academic year 2013-2014 was spent working for a small engineering firm, Cosmic AES.

2. Also holding the Ph.D. and/or MS in Electrical Engineering, with emphasis in communications and signal processing.

3. See for example *Mathematica* [Mathematica] (commercial) and *wxMaxima* [Maxima] (open source).

for engineering and scientific computing gradually took hold as I learned more about the language and the engineering problem capabilities offered by `pylab`.

When I took on the assignment of writing the *Signals and Systems for Dummies* book [Wic2013] Python seemed like a good choice because of the relative ease with which anyone could obtain the tools and then get hands-on experience with the numerical examples I was writing into the book. The power of `numpy` and the algorithms available in `scipy` are very useful in this discipline, but I immediately recognized that enhancements to `scipy.signal` are needed to make signals and systems tinkering user friendly. As examples were written for the book, I began to write support functions that fill in some of the missing details not found in `scipy`. This is the basis for the module `ssd.py`, a constant work in progress to make open source signals and systems software more accessible to the engineering community.

Modules Developed or Under Development

As already briefly mentioned, the first code module I developed is `ssd.py`[4]. This module contains 61 functions supporting *signal* generation, manipulation, and display, and *system* generation and characterization. Some of the functions implement subsystems such as a ten band audio equalization filter or the model of an automobile cruise control system. A pair of wrapper functions `to_wav()` and `from_wav()` make it easy for students to write and read 1D `ndarrays` from a wave file. Specialized plotting functions are present to make it easy to visualize both signals and systems. The collection of functions provides general support for both continuous and discrete-time signals and systems, as well as specific support for examples found in [Wic2013]. Most all of functions are geared toward undergraduate education. More modules have followed since then.

The second module developed, `digitalcom.py`, focuses on the special needs of digital communications, both *modulation* and *demodulation*. At present this module contains 32 functions. These functions are focused on waveform level simulation of contemporary digital communication systems. When I say simulation I specifically mean *Monte Carlo* techniques which involve the use of *random bit streams*, noise, channel fading, and interference. Knowledge of digital signal processing techniques plays a key role in implementation of these systems. The functions in this module are a combination of communcation waveform generators and specialized signal processing building blocks, such as the *upsampler* and *downsampler*, which allow the sampling rate of a signal to be raised or lowered, respectively. More functions are under development for this module, particularly in the area of orthogonal frequency division multiplexing (OFDM), the key modulation type found in the wireless telephony standard, long term evolution (LTE).

A third module, `fec_conv.py`, implements a rate 1/2 *convolutional encoding* and *decoding* class [Zie2015]. In digital communications digital information in the form of *bits* are sent from the transmitter to the receiver. The transmission channel might be wired or wireless, and the signal carrying the bits may be at *baseband*, as in say Ethernet, or *bandpass* on a *carrier frequency*, as in WiFi. To error protect bits sent over the channel *forward error correction* (FEC) coding, such as *convolutional codes*, may be employed. Encoding is applied before the source bits are modulated onto the carrier to form the transmitted signal. With a rate 1/2 convolutional code each source bit is encoded into two channel bits using a *shift register* of length K (termed *constraint length*) with *excusive or* logic gate connections. The class allows arbitrary constraint length codes and allows *puncturing* and *depuncturing* patterns. With pucturing/depuncturing certain code bits are *erased*, that is not sent, so as to increase the code rate from 1/2 to say 3/4 (4 channel bits for every three source bits).

For decoding the class implements the Viterbi algorithm (VA), which is a *dynamic programming* algorithm. The most likely path the received signal takes through a *trellis structure* is how the VA recovers the sent bits [Zie2015]. Here the *cost* of traversing a particular trellis branch is established using *soft decision metrics*, where soft decision refers to how information in the *demodulated* radio signal is converted metric values.

The class contains seven methods that include two graphical display functions, one of which shows the *survivor traceback paths* through the trellis back in time by the decoder *decision depth*. The traceback paths, one for each of the 2^{K-1} trellis states, give students insight into the operation of the VA. Besides the class, `fec_conv` also contains four functions for computing error probability bounds using the *weight structure* of the code under both *hard* and *soft* branch metric distance calculations [Zie2015].

A fourth module, `synchronization.py`, was developed while teaching a *phase-locked loops* course, Summer 2014. Synchronization is extremely important is all modern communications communications schemes. Digital communication systems fail to get data bits through a wireless link when synchronization fails. This module supplies eight simulation functions ranging from a basic phase-locked loop and both carrier and symbol synchronization functions for digital communications waveforms. This module is also utilized in an analog communications course taught Spring 2015.

Describing IPython Notebook Use Scenarios

In this section I describe how Python, and in particular the IPython notebook, has been integrated into teaching, graduate student research, and industry research and development.

Teaching

To put things into context, the present lecturing style for all courses I teach involves the use of a tablet PC, a data projector, a microphone, and audio/video screen capture software. Live Python demos are run in the notebook, and in many cases all the code is developed in real-time as questions come from the class. The notebook is more than just a visual experience. A case in point is the notebook audio control which adds sound playback capability. A 1D `ndarray` can be saved as a *wave file* for playback. Simply put, signals do make sounds and the action of systems changes what can be heard. Students enjoy hearing as well as seeing results. By interfacing the tablet *lineout* or *headphone* output to the podium interface to the classroom speakers, everyone can hear the impact of algorithm tweaks on what is being heard. This is where the fun starts! The modules `scipy.signal` and `ssd.py`, described earlier, are imported at the top of each notebook.

For each new chapter of lecture material I present on the tablet PC, a new IPython notebook is created to hold corresponding numerical analysis and simulation demos. When appropriate, starter content is added to the notebook before the lecture. For example I can provide relevant theory right in the notebook to

4. `http://www.eas.uccs.edu/wickert/SSD/docs/python/`

transition between the lecture notes mathematics and the notebook demos. Specifically, text and mathematics are placed in *markdown cells*. The notebook theory is however very brief compared to that of the course lecture notes. Preparing this content is easy, since the lecture notes are written in LaTeX I drop the selected equations right into mark down cells will minimal rework. Sample calculations and simulations, with corresponding plots, are often generated in advance, but the intent is to make parameter changes during the lecture, so the students can get a feel for how a particular math model relates to real-word communications and signal processing systems.

Computer projects benefit greatly from the use of the notebook, as sample notebooks with starter code are easily posted to the course Web Site. The sample notebook serves as a template for the project report document that the student will ultimately turn in for grading. The ability to convert the notebook to a LaTeX PDF document works for many students. Others used *screenshots* of selected notebook cells and pasted them into a word processor document. In Spring 2015 semester students turned in printed copies of the notebook and as backup, supplied also the notebook file. Marking on real paper documents is still my preference.

Graduate Student Research

In working with graduate students on their research, it is normal to exchange code developed by fellow graduate students working on related problems. Background discussions, code implementations of algorithms, and worked examples form a perfect use case for IPython notebook. The same approach holds for faculty interaction with their graduate students. In this scenario the faculty member, who is typically short on free time, gains a powerful advantage in that more than one student may need to brought up to speed on the same code base. Once the notebook is developed it is shared with one or more students and often demoed in front the student(s) on a lab or office computer. The ability to include figures means that system block diagrams can also be placed in the notebook.

As the student makes progress on a research task they document their work in a notebook. Faculty member(s) are briefed on the math models and simulation results. Since the notebook is live, hypothetical questions can be quickly tested and answered.

Industry Research and Development

With the notebook engineers working on the same team are able to share analytical models and development approaches using markdown cells. The inclusion of LaTeX markup is a welcome addition and furthers the establishment of notational conventions, during the development of signal processing algorithms.

Later, prototype algorithm development is started using code cells. Initially, computer synthesized signals (waveforms) are used to validated the core functionality of an algorithm. Next, signal captures (date files) from the actual real-time hardware are used as a source of test vectors to verify that performance metrics are being achieved. Notebooks can again be passed around to team members for further algorithm testing. Soon code cell functions can be moved to code modules and the code modules distributed to team members via `git` [git] or some other distributed revision control system. At every step of the way `matplotlib` [matpltlib] graphics are used to visualize performance of a particular algorithm, versus say a performance bound.

Complete subsystem testing at the Python level is the final step for pure Python implementations. When Python is used to construct a behavioral level model, then more testing will be required. In this second case the code is moved to a production environment and recoding to say C/C++. It might also be that the original Python model is simply an abstraction of real electronic hardware, in which case a hardware implementer uses the notebook (maybe just a PDF version) to create a hardware prototype, e.g., a *field programable gate array* (FPGA) or custom integrated circuit.

Live From the Classroom

Here live from the classroom means responding to questions using on-the-fly IPython notebook demos. This is an excellent way to show off the power of Python. Sometimes questions come and you feel like building a quick model right then and there during a lecture. When successful, this hopefully locks in a solid understanding of the concepts involved for the whole class. The fact that the lecture is being recorded means that students can recreate the same demo at their leisure when they watch the lecture video. The notebook is also saved and posted as a supplement/companion to the lecture. As mentioned earlier, there is a corresponding notebook for each chapter of lecture material[5]. I set the goal of re-posting the chapter notebooks each time a new lecture video is posted. This way the students have something to play with as they work on the current homework assignment.

Case Studies

In this section I present case studies that present the details on one or more of the IPython notebook use cases described in the previous section of this paper. Case studies from industry R&D are not included here due to the propriety nature of the work.

In all of the case studies you see that graphical results are produced using the `pylab` interface to `matplotlib`. This is done purposefully for two reasons. The first stems from the fact that currently all students have received exposure to MATLAB in a prior course, and secondly, I wish to augment, and not replace, the students' MATLAB knowledge since industry is still lagging when it comes to using open source tools.

Digital Signal Processing

As a simple starting point this first case study deals with the mathematical representation of signals. A step function sequence $u[n]$ is defined as

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Here I consider the difference between two step sequences starting at $n = 0$ and the other starting at $n = 5$. I thus construct in Python

$$x_3[n] = x_1[n] - x_2[n] = u[n] - u[n-5], \tag{2}$$

which forms a pulse sequence that *turns on* at $n = 0$ and *turns off* at $n = 5$. A screen capture from the IPython notebook is shown in Fig. 1.

Of special note in this case study is how the code syntax for the generation of the sequences follows closely the mathematical form. Note to save space the details of plotting $x_2[n]$ and $x_3[n]$ are omitted, but the code that generates and plots $x_3[n]$ is simply:

```
stem(n,x1 - x2)
```

5. Notebook postings for each course at `http://www.eas.uccs.edu/wickert/`

Notebook Screen Capture

Create Two Step Sequences

The module ssd (file ssd.py) contains the function `ssd.dstep(n)` which produces a step function output using the index vector `n` as the input that turns on at `n = 0`. If you input `n-5` the step will now turn on at `n=5`.

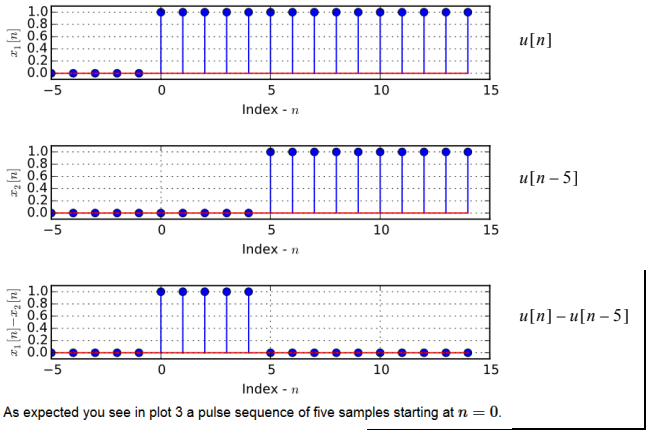
```
n = arange(-5, 15)
x1 = ssd.dstep(n) # step turns on at n = 0
x2 = ssd.dstep(n-5) # Step turns on at n = 5
```

Plot Waveforms using the Stem function

Create a 3x1 array subplots. The first two contain $x_1[n] = u[n]$ and $x_2[n] = u[n-5]$ respectively. The third plot is the difference of the first minus the second, i.e., $x_1[n] - x_2[n] = u[n] - u[n-5]$, which should be a rectangular pulse of duration five samples starting at $n = 0$.

```
figure(figsize=(6, 1.0))
stem(n, x1)
grid()
axis([-5, 15, -.1, 1.1])
xlabel(r'Index - $n$')
ylabel(r'$x_1[n]$')
```

(...Repeat for two more plots)



As expected you see in plot 3 a pulse sequence of five samples starting at $n = 0$.

Fig. 1: *Discrete-time signal generation and manipulation.*

Convolution Integral and LTI Systems

A fundamental signal processing result states that the signal output from a *linear* and *time invariant* (LTI) system is the *convolution* of the input signal with the system *impulse response*. The impulse response of a continuous-time LTI system is defined as the system output $h(t)$ in response to the input $\delta(t)$, where $\delta(t)$ is the *dirac delta function*. A block diagram of the system model is shown in Fig. 2.

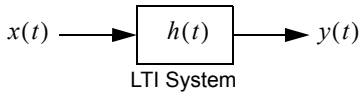


Fig. 2: *Simple one input one output LTI system block diagram.*

In mathematical terms the output $y(t)$ is the integral

$$y(t) = \int_{-\infty}^{\infty} h(\lambda)x(t-\lambda)\,d\lambda \tag{3}$$

Students frequently have problems setting up and evaluating the convolution integral, yet it is an important concept to learn. The waveforms of interest are typically piecewise continuous, so the integral must be evaluated over one or more contiguous intervals. Consider the case of $x(t) = u(t) - u(t-T)$, where $u(t)$ is the unit step function, and $h(t) = ae^{-at}u(t)$, where $a > 0$. To avoid careless errors I start with a sketch of the integrand $h(\lambda)x(t-\lambda)$, as shown in Fig. 3. From there I can discover the support intervals or *cases* for evaluating the integral.

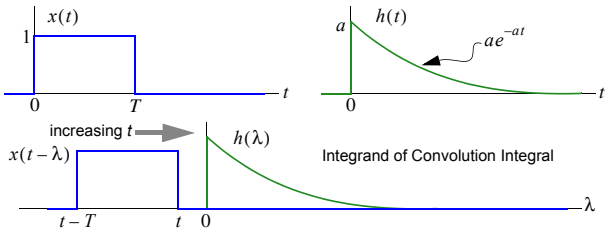


Fig. 3: *Sketches of $x(t)$, $h(t)$, and $h(\lambda)x(t-\lambda)$.*

A screen capture of a notebook that details the steps of solving the convolution integral is given in Fig. 4. In this same figure we see the analytical solution is easily plotted for the case of $T = 1$ and $a = 5$.

Notebook Screen Capture

Convolution Integral Simulation

For a continuous-time linear time invariant (LTI)system having impulse response $h(t)$ and input signal $x(t)$, the output, $y(t)$ can be written in terms of a convolution integral:

$$y(t) = \int_{-\infty}^{\infty} x(\lambda)h(t-\lambda)\,d\lambda = \int_{-\infty}^{\infty} h(\lambda)x(t-\lambda)\,d\lambda$$

Special Case

Consider $x(t) = u(t) - u(t-T)$ a rectangular pulse of duration Ts and $h(t) = ae^{-at}u(t)$ an exponential, where $a > 0$. **Note**: The impulse response is of the form of the well known RC lowpass filter if we let $a = 1/RC$.

Writing out and evaluating the convolution integral for the given $x(t)$ and $h(t)$ results in a piecewise solution involving three contiguous support intervals: (Case 1) $t < 0$, (Case 2) $0 \geq t < T$, and (Case 3) $t \geq T$. The integrand is zero for Case 1. Using the second form of the convolution integral, Case 2 evaluates to:

$$y(t) = \int_0^t ae^{-(t-\lambda)}\,d\lambda = -e^{-a\lambda}\Big|_0^t = 1 - e^{-at},\ 0 \leq t < T$$

For Case 3 we have

$$y(t) = \int_{t-T}^t ae^{-(t-\lambda)}\,d\lambda = -e^{-a\lambda}\Big|_{t-T}^t = e^{-a(t-T)}\left[1 - e^{-aT}\right],\ t \geq T$$

In summary:

$$y(t) = \begin{cases} 0, & t < 0 \\ 1 - e^{-at}, & 0 \leq t < T \\ e^{-a(t-T)}\left[1 - e^{-aT}\right], & t \geq T \end{cases}$$

Plot the piecewise solution:

```
# Let T = 1s and a = 5
figure(figsize=(6, 2))
T = 1; a = 5
tt = arange(-1, 3.001, .01)
yt = (1-exp(-a*tt))*(ssd.step(tt)-ssd.step(tt-T))\
     +exp(-a*(tt-T))*(1-exp(-a*T))*ssd.step(tt-T)
plot(tt, yt,'g')
```

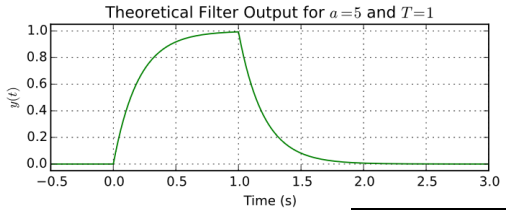


Fig. 4: *Solving the convolution integral in the notebook .*

To bring closure to the tedious analytical solution development, I encourage students check their work using computer simulation. The function `ssd.conv_integral()` performs numerical evaluation of the convolution integral for both finite and semi-infinite extent limits. I simply need to provide an array of signal/impulse response sample values over the complete support interval. The screen capture of Fig. 5 shows how this is done in a notebook. Parameter variation is also explored. Seeing the two approaches provide the same numerical values is rewarding and a powerful testimony to how the IPython notebook improves learning and understanding.

Notebook Screen Capture

Check on the Analytical Solution

```
# Let T = 1s and a = 1
figure(figsize=(6, 2))
a = 1
t = arange(-1, 3.001, .001)
x = ssd.step(t) - ssd.step(t-1)
h = a*exp(-a*t)*ssd.step(t)
y, ty = ssd.conv_integral(x, t, h, t)
plot(ty, y)
```

Generate $x(t)$ and $h(t)$ then numerically convolve with *scipy.signal.convolve* used in the core calculation

(...Repeat for two more plots with a = 5 and 10)

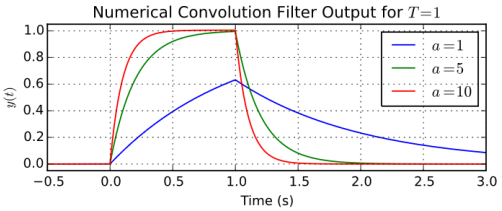


Fig. 5: *Plotting $y(t)$ for $a = 1, 5,$ and 10.*

Convolutional Coding for Digital Communications

In this case study the coding theory class contained in `fec_conv.py` is exercised. Here the specific case is taken from a final exam using a rate 1/2, $K = 5$ code. Fig. 6 shows the construction of a `fec_conv` object and a plot of one code symbol of the trellis.

Notebook Screen Capture

Part a: *K* = 5 Rate 1/2 Code

In this first part you will create a fec_conv object for the $K = 5$ code of Table 1. You will create a BEP plot similar to that found on page 7-41 of the Chapter 7 (text Chapter 12) notes. Note: You will need to increase D to about $5 \times 5 = 25$. Unlike the notes example, your results will be for soft-decision decoding. Functions for computing soft decision decoding upper bounds are contained in the module fec_conv.py. In addition to the BEP plot, also provide the trellis plot and a traceback plot under low and high SNR values.

```
In [5]: # Instantiate a fec_conv coder/decoder object
        cc1 = fec.fec_conv(('10011', '11101'), 25)
        cc1.trellis_plot()
```

Note: $2^{K-1} = 16$ states.
Blue transitions for '0' input bit
Green transitions for '1' input bit

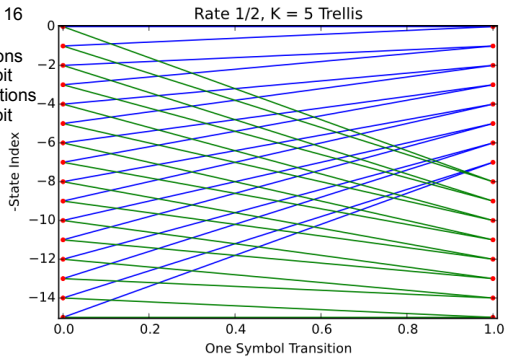


Fig. 6: *Construction of a `fec_conv` object and the corresponding trellis structure for the transmission of one code symbol.*

At the digital communications receiver the received signal is demodulated into *soft decision* channel bits. The soft values are used to calculate *branch metrics*, which then are used to update cumulative metrics held in each of the 16 states of the trellis. There are two possible paths arriving at each state, but the *surviving* path is the one producing the minimum cumulative metric.

Fig. 7 shows the survivor traceback paths in the 16-state trellis while sending random bits through the encoding/decoding process. Additive noise in the communications channel introduces confusion in the formation of the traceback paths. The channel *signal-to-noise ratio* (SNR), defined as the ratio of received signal

Notebook Screen Capture

High SNR Traceback Plot

```
cc1 = fec.fec_conv(('10011', '11101'), 25)
EbN0 = 7
# Create 1000 random 0/1 bits
x = randint(0, 2, 1000)
# Encode with shift register starting state of '0000'
state = '0000'
y, state = cc1.conv_encoder(x, state)
# Add channel noise to bits translated to +1/-1
yn = dc.cpx_AWGN(2*y-1, EbN0-3, 1)  # Channel SNR is dB less
# Translate noisy +1/-1 bits to soft values on [0, 7]
yn = (yn.real+1)/2*7
z = cc1.viterbi_decoder(yn)
# Look at the traceback in the VA trellis
cc1.traceback_plot()
```

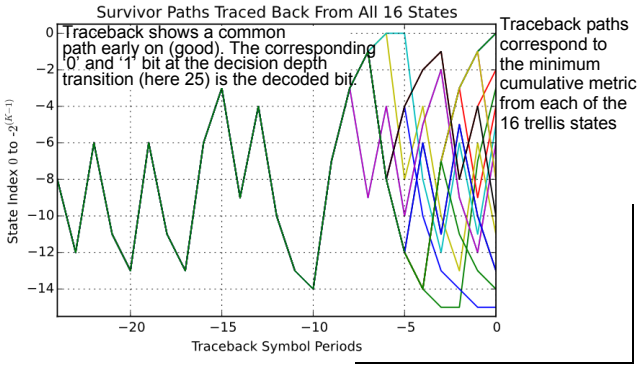


Fig. 7: *Passing random bits through the encoder/decoder and plotting an instance of the survivor paths.*

power to background noise power, sets the operating condition for the system. In Fig. 7 the SNR, equivalently denoted by E_b/N_0), is set at 7 dB. At a *decision depth* of 25 code symbols, all 16 paths merge to a common path, making it very likely that the probability of a bit error, is very very small. At lower a SNR, not shown here, the increased noise level makes it take longer to see a traceback merge and this is indicative of an increase in the probability of making a bit error.

Real-Time Digital Signal Processing

In the real-time digital signal processing (DSP) course C-code is written for an embedded processor. In this case the processor is an ARM Cortex-M4. The objective of this case study is to implement an equal-ripple *finite impulse response* (FIR) lowpass filter of prescribed amplitude response specifications. The filter is also LTI. Python (`scipy.signal`) is used to design the filter and obtain the filter coefficients, $b_1[n]$, $n = 0, \ldots, M$, in `float64` precision. Here the filter order turns out to be $M = 77$. As in the case of continuous-time LTI systems, the relation between the filter input and output again involves a convolution. Since a digital filter is a discrete-time system, the *convolution sum* now appears. Furthermore, for the LTI system of interest here, the convolution sum can be replaced by a *difference equation* representation:

$$y[n] = \sum_{k=0}^{M} x[n]b[n-k], \quad -\infty < n < \infty \tag{4}$$

In real-time DSP (4) becomes an algorithm running in real-time according to the system sampling rate clock. The processor is working with `int16` precision, so once the filter is designed the coefficients are scaled and rounded to 16 bit signed integers as shown in Fig. 8. The fixed-point filter coefficients are written to a C header file using a custom function defined in the notebook (not shown here).

Notebook Screen Capture

Remez Equi-Ripple Design

```
# Load an equi-ripple design code module from GNU Radio
#(in notebook ZIP package)
import optfir
```

```
d_pass = 0.2
d_stop = 60.0
fs = 48000
f_pass = 3500
f_stop = 5000
n, ff, aa, wts=optfir.remezord([f_pass, f_stop], [1, 0],
                               [1-10**(-d_pass/20), 10**(-d_stop/20)],
                               fsamp=48000)
# Bump up the order by 5 to bring down the final d_pass & d_stop
n_bump = n + 5
b1 = signal.remez(n_bump, ff, aa[0::2], wts, Hz=2)
```

Note: The original amplitude response requirements have been changed. The passband ripple is now 0.2 db and the passband critical frequency is reduced from 4000 to 3500 Hz. This reduces the filter order.

Fixed-Point Coefficients (`int16_t`)

```
b1_fix = int16(rint(b1*2**15))
b1_fix
```

```
array([   14,   -13,   -33,   -63,   -97,  -124,  -134,  -118,   -72,
          -2,    80,   151,   191,   179,   109,    -9,  -148,  -269,
        -332,  -304,  -176,    32,   274,   481,   581,   520,   281,
        -103,  -549,  -936, -1128, -1009,  -512,   360,  1519,  2810,
        4036,  4993,  5518,  5518,  4993,  4036,  2810,  1519,   360,
        -512, -1009, -1128,  -936,  -549,  -103,   281,   520,   581,
         481,   274,    32,  -176,  -304,  -332,  -269,  -148,    -9,
         109,   179,   191,   151,    80,    -2,   -72,  -118,  -134,
        -124,   -97,   -63,   -33,   -13,    14], dtype=int16)
```

```
FIR_fix_header('s4_p1_remez.h',b1)
```

Fig. 8: *Designing an equal-ripple lowpass filter using scipy.signal.remez for real-time operation.*

Notebook Screen Capture

Import Measured Results

```
fs_48, remez78_q15 = loadtxt('spec_noise_b_remez78_fs48arm_q15.csv',
                             delimiter=',', skiprows=1, usecols=(0, 1),
                             unpack=True)
```

```
figure(figsize=(6, 3))
f = arange(0, 1.0, .001)
w, B = signal.freqz(b1, 1, 2*pi*f)
w, Bq = signal.freqz(b1_fix, 1, 2*pi*f)
plot(f*48, 20*log10(abs(B)))
plot(f*48, 20*log10(abs(Bq)/sum(b1_fix)))
#plot(fs_48[:600]/1000, remez78_q15[:600]-mean(remez78_q15[:20]))
title(r'Equiripple Lowpass Theory: %d Taps' % n_bump)
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz ($f_s =$ %d kHz)' % (fs/1e3,))
legend((r'Theory float64', r'Theory int16'), loc='upper right')
ylim([-70, 2])
xlim([0, fs/1e3/2])
grid();
```

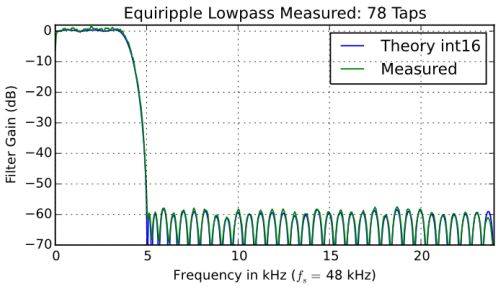


Fig. 9: *Comparing the theoretical fixed-point frequency response with the measured.*

The filter frequency response magnitude is obtained using a noise source to drive the filter input (first passing through an analog-to-digital converter) and then the filter output (following digital-to-analog conversion) is processed by instrumentation to obtain a spectral estimate. Here the output spectrum estimate corresponds to the filter frequency response. The measured frequency response is imported into the notebook using `loadtxt()`. Fig. 9 compares the theoretical frequency response, including quantization errors, with the measured response. The results compare favorably. Comparing theory with experiment is something students are frequently asked to do in lab courses. The fact that the stopband response is not quite equal-ripple is due to coefficient quantization. This is easy to show right in the notebook by overlaying the frequency response using the original `float64` coefficients `b1`, as obtained in Fig. 8, with the response obtained using the `b1_fix` coefficients as also obtained in Fig. 8 (the plot is not shown here).

An important property of the equal-ripple lowpass is that the filter coefficients, $b[n]$, have even symmetry. This means that $b_1[M-n] = b_1[n]$ for $0 \le n \le M$. Taking the z-transform of both sides of (4) using the convolution theorem [Opp2010] results in $Y(z) = H(z)X(z)$, where $Y(z)$ is the z-transform of $y[n]$, $X(z)$ is the z-transform of $x[n]$, and $H(z)$, known as the *system function*, is the z-transform of the system impulse response. The system function $H(z)$ takes the form

$$H(z) = \sum_{n=0}^{M} b_n z^{-n} \overset{\text{also}}{=} \frac{1}{z^M} \prod_{n=1}^{M} (z - z_n), \tag{5}$$

In general $H(z) = N(z)/D(z)$ is a rational function of z or z^{-1}. The roots of $N(z)$ are the system zeros and roots of $D(z)$ are the system poles. Students are taught that a *pole-zero* plot gives much insight into the frequency response of a system, in particular a filter. The module `ssd.py` provides the function `ssd.zplane(b,a)` where `b` contains the coefficients of $N(z)$ and `a` contains the coefficients of $D(z)$; in this case `a = [1]`. The even symmetry condition constrains the system zeros to lie at conjugate reciprocal locations [Opp2010] as seen in Fig. 10.

```
ssd.zplane(b1, [1], auto_scale=False, size=1.2)
```

```
(77, 0)
```

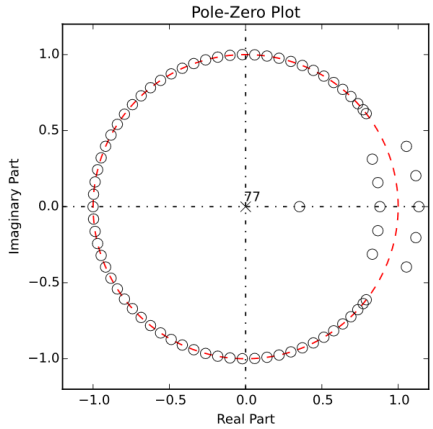


Fig. 10: *Pole-zero plot of the equal-ripple lowpass which confirms that $H(z)$ is linear phase.*

With real filter coefficients the zeros must also occur in conjugate pairs, or on the real axis. When the student sees the pole-zero plot of Fig. 10 whats jumps off the page is all of the zeros on the unit circle for the filter stopband. Zeros on the unit circle block signals from passing through the filter. Secondly, you see conjugate reciprocal zeros at angles over the interval $[-\pi/4, \pi/4]$ to define the filter passband, that is where signals pass through the filter. As a bit of trivia, zeros not on the unit circle or real axis **must** occur as quadruplets, and that is indeed what is seen in

Fig. 10. Note also there are 77 poles at $z = 0$, which is expected since $M = 77$. The pole-zero plot enhances the understanding to this symmetrical FIR filter.

Statistical Signal Processing

This case study is taken from a computer simulation project in a statistical signal processing course taken by graduate students. The problem involves the theoretical calculation of the probability density function of a random variable (RV) $\mathbf{w}$ where

$$\mathbf{w} = \mathbf{xy} + \mathbf{z}$$

is a function of the three RVs $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. Forming a new RV that is a function of three RV as given here, requires some serious thinking. Having computer simulation tools available to check your work is a great comfort.

The screenshot of Fig. 11 explains the problem details, including the theoretical results written out as the piecewise function `pdf_proj1_w(w)`.

Notebook Screen Capture

Problem 1

The random variable w is defined in terms of the random variables x, y, and z, to be

$$\mathbf{w} = \mathbf{xy} + \mathbf{z}$$
$$= \mathbf{v} + \mathbf{z}$$

The input rv are assumed to be mutually independent, with $x\ U(-1,1)$, $y\ U(-1,1)$, and $z\ U(0,1)$.

Find the theoretical pdf . Start by first finding the pdf on $\mathbf{v} = \mathbf{xy}$ using the fact that for independent $\mathbf{x}$ and $\mathbf{y}$,

$$f_v(v) = \int_{-\infty}^{\infty} \frac{1}{w} f_x(w) f_y\left(\frac{v}{w}\right) dw$$
$$= \int_{-\infty}^{\infty} \frac{1}{w} f_y(w) f_x\left(\frac{v}{w}\right) dw$$

Then find the pdf on the sum $\mathbf{w} = \mathbf{v} + \mathbf{z}$ from a convolution. Note that the rv $\mathbf{v}$ and $\mathbf{z}$ are also independent. Why?

Theoretical Analysis

```
def pdf_proj1_w(w):
    """
    fw = pdf_proj1_w(w)
    Function plot the pdf of w = x*y + z where x~U(-1,1), y~U(-1,1), and
    z~U(0,1).

    Mark Wickert March 2015
    """

    fw = zeros_like(w)

    for k, wk in enumerate(w):
        if wk >= -1 and wk <= 0:
            fw[k] = -1/2*(wk*log(-wk)-wk-1)
        elif wk > 0 and wk <= 1:
            fw[k] = 1/2*(1 + (wk-1)*log(1-wk) - wk*log(wk))
        elif wk > 1 and wk <= 2:
            fw[k] = 1/2*(2 - wk + (wk-1)*log(wk-1))
        else:
            fw[k] = 0
    return fw
```

Fig. 11: *One function of three random variables simulation problem.*

Setting up the integrals is tedious and students are timid about pushing forward with the calculus. To build confidence a simulation is constructed and the results are compared with theory in Fig. 12.

Conclusions and Future Work

Communications and signal processing, as a discipline that sits inside electrical computer engineering, is built on a strong mathematical modeling foundation. Undergraduate engineering students, despite having taken many mathematics courses, are often intimidated by the math they find in communications and signals processing course work. I cannot make the math go away, but good modeling tools make learning and problem solving fun and exciting. I have found, and hopefully this paper shows, that

Notebook Screen Capture

Simulation

Create the Random Variates

```
x = 2*rand(1000000, 1)-1
y = 2*rand(1000000, 1)-1
z = rand(1000000, 1)
w = x*y + z
v = x*y
```

```
figure(figsize=(6, 3))
hist(w, 51, (-1, 2), normed=True, cumulative=False);
xlabel(r'$w$')
ylabel(r'Probability Density $f_w(w)$')
title(r'Probability Density of $w = xy + z$')
wr = arange(-1.2, 2.2, .001)
plot(wr, pdf_proj1_w(wr), 'r')
grid();
```

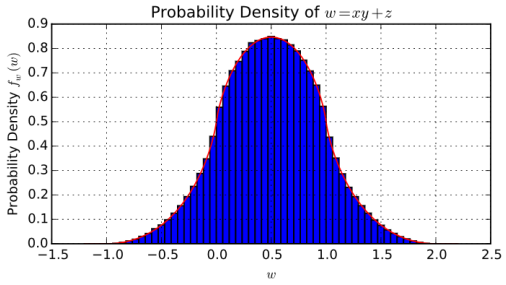


Fig. 12: *The simulation of random variable* $\mathbf{w}$ *and the a comparison plot of theory versus a scaled histogram.*

IPython notebooks are valuable mathematical modeling tools. The case studies show that IPython notebook offers a means for students of all levels to explore and gain understanding of difficult engineering concepts.

The use of open-source software is increasing and cannot be overlooked in higher education. Python is readily accessible by anyone. It is easy to share libraries and notebooks to foster improved communication between students and faculty members; between researchers, engineers, and collaborators. IPython and the IPython notebook stand out in large part due to the enthusiasm of the scientific Python developer community.

What lies ahead is exciting. What comes to mind immediately is getting other faculty on-board. I am optimistic and look forward to this challenge as tutorial sessions are planned over summer 2015. Other future work avenues I see are working on more code modules as well as enhancements to the existing modules. In particular in the convolutional coding class both the encoder and especially the Viterbi decoder, are numerically intensive. Speed enhancements, perhaps using *Cython*, are on the list of things to do. Within the notebook I am anxious to experiment with notebook controls/widgets so as to provide dynamic interactivity to classroom demos.

Acknowledgments

The author wishes to thank the reviewers for their helpful comments on improving the quality of this paper.

References

[Wic2013] M.A. Wickert. *Signals and Systems for Dummies*, Wiley, 2013.
[ssd] http://www.eas.uccs.edu/wickert/SSD/.
[MATLAB] http://www.mathworks.com/.
[Octave] https://en.wikipedia.org/wiki/GNU_Octave.
[Mathematica] https://en.wikipedia.org/wiki/Mathematica.
[Maxima] http://andrejv.github.io/wxmaxima/.

[Zie2015] R.E. Ziemer and W.H. Tranter *Principles of Communications*, seventh edition, Wiley, 2015.

[git] `https://git-scm.com/`

[matpltlib] `http://matplotlib.org/`

[Opp2010] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing* (3rd ed.), Prentice Hall, 2010.