Analog Filter Design

Part. 2: Scipy (Python) Signals Tools

Python - Scipy

Modules:

- Standard Library
- Optional modules

…………….

…………….

Scientific Python

……………..

……………..

- **numpy:** functions, array, solvers etc.
- **scipy:** scientific and engineering modules
- **matplotlib:** 2D, 3D plotting functions

Anaconda distribution: includes all scientific modules and takes care of all possible dependences

Suggested Editor: **Idle** (<python_dir> \Lib \ Idlelib \Idle.bat)
Suggested .py files open method: place a link to Idle.bat into windows "**send to**" folder (lo locate the "send to" folder, execute the command "shell:sendto" with the Windows "Run" dialog box, which can be called with the keystrokes: Win + R)

Array – like structures in python

Base Python:
Lists: e.g. >a=[1,3,5,"aaa",[2,3]] # (non homogeneous data types)
Tuple: e.g. > a=(1,3,5,"aaa",[2,3]) # (non homogeneous data types)
………

Numpy Arrays (class ndarray)

> import numpy as np
> a=np.array([1.0,3.67,2.9]) (homogeneous numerical data)
Many numpy functions accept both lists (or tuples) and arrays as arguments, but convert everything to array

Array generation functions

Creation of 1D array

a=np.arange(start,stop,step)

a=np.linspace(start, stop, num_points)

a=np.logspace(first_dec,last_dec, num_point)

Array importing and saving (from / to text files)

A=np.loadtxt(“nome_file”)

np.savetxt(A)

It is possible to specify a data delimiter through an optional argument: delimiter=“ “. Default is space.

Array indexing and manipulation

Example: 2D array: a[k,h] (this notation does not work with lists)

Application of functions to arrays: element by element
Example   np.sin(a) -> Returns an array by applying the sin() function
to all the elements of *a*

Array stacking. Example: A=np.array([[1,4],[-2.5,6]]); b=np.array([22,11])

$$A = \begin{pmatrix} 1 & 4 \\ -2.5 & 6 \end{pmatrix}$$

np.row_stack([A,b]) $\Rightarrow$ $\begin{pmatrix} 1 & 4 \\ -2.5 & 6 \\ 22 & 11 \end{pmatrix}$

$$b = \begin{pmatrix} 22 \\ 11 \end{pmatrix}$$

np.column_stack([A,b]) $\Rightarrow$ $\begin{pmatrix} 1 & 4 & 22 \\ -2.5 & 6 & 11 \end{pmatrix}$

Other important array manipulation functions

Append a new value to an array:

np.append(x,value) In 1D vectors, value can be a float (no need to be an array). x can be an empty array.

Empty array:

np.array([ ]) (useful to start a cycle where values are progressively appended to a vector)

Array “slicing” syntax

x[:5] all elements up to 5th (excluded, i.e., index 0,1,2,3,4)

A[:,1] whole second column (index=1) of a 2D array

y[x>0] y values for indices i such that x[i]>0. (x[x>0] is possible)

Matrix algebraic functions

a.T   (transposed of a)

np.dot(a,b)   (matrix product, if a and b are vectors the scalar product is calculated)

np.linalg.inv(a) calculates the inverse of a, if it exists

Note: 1D arrays are not divided into column or row vectors). Thus, the transpose operation applied to a 1D arrays has no effect.

2D - Plotting functions

- import matplotlib.pyplot as plt
- plt.plot(a,b)   # Linear plots
- plt.semilogx(a,b)   # logarithmic X axis
- plt.semilogy(a,b)   # logarithmic Y axis
- plt.loglog(a,b)   # both axes are logarithmic

Labelling:

```
plt.plot(x,y, label="primo")
plt.plot(x1,y1, label="secondo")
plt.legend()
plt.xlabel("X")
plt.label("Y")
```

Customizing Labels Fonts for all plots:

```
import matplotlib as mpl

mpl.rc('xtick', labelsize=16)
mpl.rc('ytick', labelsize=16)
mpl.rc("axes",labelsize=18)
```

Scipy signal module : time continuous filter synthesis

```
> import scipy.signal as signal
```

Order e characteristic frequency determination

```
> signal.cheb1ord(wp, ws, Ap, As, analog=True)
> signal.buttord(wp, ws, Ap, As, analog=True)
> signal.cheb2ord(wp, ws, Ap, As, analog=True)
> signal.ellipord(wp, ws, Ap, As, analog=True)
```

Filter synthesis

```
signal.butter(N, Wn[, btype, analog=True, output])
signal.cheby1(N, rp, Wn[, btype, analog=True, output])
signal.cheby2(N, rs, Wn[, btype, analog=True, output])
signal. ellip(N, rp, rs, Wn[, btype, analog=True, output])
signal. bessel(N, Wn[, btype, analog=True, output])
```

btype : *'lowpass', 'highpass', 'bandpass', 'bandstop'*

output : *'ba', 'zpk', 'sos'*

Synthesis functions: type of output

Examples shown with the butterworth function – others behave in the same way

Polynomial coefficients

b, a =signal.butter(N, Wn[, btype, analog=True, output="ba"])

b: array of numerator polynomial coefficients – b[0] is the 0-order coefficient, b[k] is s^k coeff.

a: array od denominator coefficients - a[0] is the 0-order coefficient, a[k] is s^k coeff.

Zeroes and Poles

z, p, k =signal.butter(N, Wn[, btype, analog=True, output="zpk"])

z: array of zeroes ; p: array of poles, k: constant multiplier (scalar, necessary to make gain=1)

More on TC filter synthesis

- High pass, Band-pass, Band-stop

signal.butter(N, (W_{NL},W_{NH}), [, btype=“bandpass”, analog=True, output])

- Universal filter syntesis

signal.iirdesign(wp, ws, gpass, gstop, analog=True, ftype='ellip', output='ba')

“butter”, “cheby1”, “cheby2”, “ellip”

Determine the type of transfer function (lowpass, etc) considering
whether wp and ws are scalars or tuples and their relative position

Frequency and response of continuous-time filters

signal.freqs(b, a[, worN])	Compute frequency response of analog filter. worN=w_vector (optional, use the specified freq. axis, otherwise determine one on the basis of filter freqs) Returns w,p (frequencies, complex freq. response)
signal.bode(system[, w, n])	Calculate Bode magnitude and phase data w=w_vector (frequency axis); n=number of points to be used if w is not given. system: (b,a) or (p,z,k) or (A,B,C,D) --- autodetect from tuple size

Time response of continuous-time filters

signal.impulse(system[, X0, T, N]) X0: initial state (usually not given), T=time_vector, is the time axis, computed if not given. N, number of time points if T not given. Returns time,yout

signal.step(system[, X0, T, N]) Step response of continuous-time system.

signal.lsim(system, U, T, X0=None, interp=True) Response to an arbitrary stimulus U
U samples are taken at T times (uniform spacing)
interp: True -> linear; false -> step-like

Discrete Time Filters

- It is possible to synthesize IIR filters using the same function as for TC filters setting “analog” to false (default). For example for Butterworth-type:

```
scipy.signal.butter(N, Wn, btype='low', analog=False, output='ba')
scipy.signal.buttord(wp, ws, gpass, gstop, analog=False)
```

Note: frequencies are normalized to Nyquist frequency (fs/2)

- It is also possible to transform a TC transfer function in the Laplace domain into a DT transfer function H(z) that approximates it:

`bz,az=signal.bilinear(b, a, fs=1.0)` use bilinear transformation

`sys_td = signal.cont2discrete(sys, dt,method='zoh')` more general

Note: in order to extract bz and az do the following: bz=sys_td[0][0], az=sys_td[1]

Discrete Time Filters: characterization

Impulse and step response in the discrete time domain

```
signal.dimpulse(system, x0=None, t=None, n=None)
signal.dstep(system, x0=None, t=None, n=None)
```

system= (num, den, dt)

Note: The degree of the denominator should be higher than that of the numerator otherwise an error is generated. In discrete time filters the numerator degree is often higher than that of the denominator (which is absent in FIR filters). To avoid this limitation, create a denominator with coefficients up to the maximum degree of the numerator and place any additional coefficient to zero

Frequency response

w=frequency axis, h=H(w) (complex)

```
w,h=signal.freqz(b[, a, worN=none])
```

$$H(z)=\frac{b[0]+b[1]z^{-1}+\ldots+b[m]z^{-m}}{a[0]+a[1]z^{-1}+\ldots.a[n]z^{-n}}$$

worN: freq. axis (optional)

FIR Filters Synthesis

```
signal.firwin(numtaps, cutoff, window='hamming', pass_zero=True, nyq=1.0)
```

numtaps: number of elements in the filter kernel. Should be odd for maximum flexibility (not relevant for low pass functions)

cutoff: a single frequency for low-pass and high pass
[fL, fh] for band pass or band stop
Frequencies are in fraction of the Nyquist frequency (nyq) i.e. fs/2

window: frequently used values “blackman”, “hamming”, “boxcar” (rectangular)

pass_zero: if true the gain in the assumed pass-band is not zero. The effect is the following:

	cutoff=single frequency	**cutoff=Tuple [fL,fh]**
pass_zero=True	Low pass	Band pass
pass_zero=false	High pass	Band stop

List of free CAD tools useful for Filter design

- **Python (module scipy.signal)**: synthesis and simulation of filter transfer functions. Suggested Python distribution that includes all required modules: anaconda - https://www.anaconda.com/what-is-anaconda/
- **LTSpice: SPICE** – based electrical simulator. Can be used with both idealized component (op-amps, transconductors) or real commercial devices: http://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html
- **ELsie**: synthesis of passive ladder filters and more. Free student version up to 7th order filters: http://www.tonnesoftware.com/elsie.html
- Analog Filter Wizard (by Analog Devices): very simple online tool for the synthesis of filters based on biquad cascades (sallen-key and/or friend-delyannis biquads): http://www.analog.com/designtools/en/filterwizard/