Cyfrowe Przetwarzanie Sygnałów z użyciem Pythona oraz modułów SciPy, NumPy i Matplotlib

(oraz dlaczego Matlab jest zły !)

Autor: Jacek Nosal
2011 r.

Artykuł zrealizowany będzie w formie tutorialu, który ma na celu przedstawienie podstawowych zagadnień cyfrowego przetwarzania sygnałów przy użyciu języka Python. Jako, że najpopularniejszym obecnie środowiskiem do modelowania i testowania zagadnień z zakresu DSP jest obecnie pakiet Matlab zdecydowałem się stworzyć coś, co pokaże nam jak możemy eksperymentować używając narzędzi open-source. Alternatywą do przedstawionego przeze mnie rozwiązania (połączenie Python + SciPy) może być np. pakiet Scilab, jednak ze względu na możliwości Pythona, oraz moje zainteresowania programistyczne zdecydowałem się zająć się tym właśnie językiem.

Dla ułatwienia, przykłady tj. kod będzie wyróżniony *kolorem niebieskim z kursywą*.

Wstęp

Celem tego tutorialu jest przedstawienie w jaki sposób można połączyć różne moduły i narzędzia przeznaczone bezpośrednio do wykorzystania z językiem Python. Nie będę za bardzo wkraczał w tematykę DSP, pewna wiedza z tego zakresu jest w jakiś sposób wymagana, gdyż jest to dziedzina na tyle szeroka, że nie da się jej „streścić" w kilku paragrafach. Zaznaczam jednak, że jeśli uznam jakąś rzecz wartą dodatkowego wyjaśnienia, to odpowiednia teoria zostanie zaprezentowana. Osobom zainteresowanym DSP nie tylko od strony implementacji programistycznych polecam sięgnięcie do powszechnie dostępnej literatury.

UWAGA !
Jeśli ktoś ma zamiar poeksperymentować z pojawiającym się kodem wymagane jest posiadanie implementacji Pythona oraz trzech modułów: SciPy, NumPy i Matplotlib, są one powszechnie i bezpłatnie dostępne – wystarczy wpisać nazwę w wyszukiwarce i wybrać odpowiednią dla danego systemu implementację.

Dlaczego DSP z użyciem Pythona ?

Zanim przejdę do właściwej teorii chciałbym odpowiedzieć na jedno nasuwające się pytanie: Po co ?, a właściwie – Dlaczego ?. W dzisiejszych czasach na rynku możemy znaleźć wiele narzędzi i bibliotek traktujących o DSP praktycznie dla każdego języka programowania. Dlaczego więc wybrałem Pythona ?

1. Jest to język interpretowany, tzn. w prosty sposób możemy powiedzieć, że mamy do czynienia z powłoką i interpreterem, który czyta wpisany przez nas kod linijka po linijce podejmując konkretne działania. Takie rozwiązanie jest ciekawą alternatywą dla języków kompilowanych – pomijany jest etap tzw. budowania. Innym przykładem języka interpretowanego jest np. Matlak – dlaczego więc nie użyjemy właśnie jego ? I

tu pojawia się problem licencji oraz pieniędzy. Mimo tego, że Matlab jest środowiskiem wielce popularnym wśród uczniów i studentów to jednak próżno szukać nawet okrojonych wersji dostępnych bezpłatnie – w rzeczywistości to uczelnie posiadają licencje na ten pakiet.

2. Python to język, który wciąż się rozwija, mam na myśli, że cały czas pojawiają się nowe moduły, których użycie sprowadza się do szybkiego spojrzenia na dokumentację i skorzystanie z helpa. Do najpopularniejszych należą: SciPy – pakiet dla naukowców i inżynierów, PyGame – moduł pozwalający tworzyć gry, Django – framework będący narzędziem używanym przez webdeveloperów.
3. Ostatnim powodem jest wygoda. Jeśli zdecydujemy się na „DSP z Pythonem" szybko dostrzeżemy potencjał języka i pojawiające się możliwości. Na przykład, kilka dni temu dane mi było użycie wyżej wymienionych modułów wraz z inną (częściowo open-sourceową) Protocol Buffers do stworzenia narzędzia, które generuje sygnały do eksperymentów, rozwiązując problem serializacji/deserializacji oraz zapewniając takie operacje jak obliczanie histogramu, czy modyfikacja parametrów sygnału.

Plan

Najpierw przedstawiony zostanie moduł NumPy – moduł ułatwiający obliczenia numeryczne przy użyciu Pythona. Następnie, omówię podstawy biblioteki matplotlib oraz moduł SciPy. Ostatecznie popracujemy chwilę używając wszystkich trzech modułów.

NumPy

DSP to „proces", którego podstawą są operacje wykonywane na wartościach dyskretnych. To znaczy, że sygnały analogowe poddawane są próbkowaniu i kwantyzacji w wyniku czego dostajemy dyskretny zbiór próbek zamiast ciągłego zbioru wartości. Wszystkie te operacje A/C i C/A wykonywane są poniekąd dla wygody komputerów (w dużym cudzysłowie). Mikroprocesory operują na danych binarnych więc pojawia się problem „mapowania" czy też odwzorowania nieskończonego zbioru ciągłego w dyskretny zbiór pewnych próbek. Działamy więc na wartościach dyskretnych, których „ilość" jest skończona, jesteśmy więc zobowiązywani do stosowania pewnych standardów indeksowania i zapisu. Spójrzmy na ten kod :

n=[0,1,2,3,...] lub n=[1,2,3,....]

Zdefiniowaliśmy pewien ciąg wartości, na razie nie będziemy zajmować się problemem indeksowania – tj. czy powinno się ono zaczynać od 0 od 1 a może od jeszcze innej wartości. Możemy powiedzieć, że mamy już naszą dziedzine – kolejne próbki będą numerowane np. 1,2,3,4 itd. Wektor wartości może przedstawiać się następująco :

x=[war1,war2,war3,....]

Istnieje więc pewna potrzeba elastycznej, prostej I łatwej do wykorzystania reprezentacji wektorów/zbiorów/wartości. Dlatego właśnie zajmujemy się NumPy. NumPy to moduł Pythona, który zawiera wszystkie niezbędne klasy i funkcje potrzebne do reprezentacji naszych wektorów. Mamy możliwość operacji na tablicach, macierzach itd. mając przy tym dostęp do wszystkich niezbędnych funkcji, jak odwracanie, transpozycja itp.

Przyjrzyjmy się więc przykładowej sesji z Pythonem:
Zacznijmy od zaimportowania modułu NumPy:
from numpy import *
Teraz stworzymy odpowiednie sygnały:
n=linspace(1,100,100) lub n=arange(1,101) // składnia linspace jest prosta //linspace(value_min,value_max,how many)
Stworzylismy wiec 100 elementowy wektor ktory po wypisaniu na ekran prezentuje sie nastepujaco:
print n
[1. 2. 3. 4. 5. ... 99. 100.]
Teraz stworzymy wektor wartosci :
x1=linspace(-1,1,100) //wektor wartosci z przedziału: (-1;1), dlugosc wynosi 100
x2=zeros(100) // wektor zer
x3=array((x1)*2) // tak samo jak x1, ale wartosci są mnozone
x4=array((x1)**2) // x1 do potęgi 2

Widzimy więc, że istnieją pewne funkcje, które ułatwiają nam pracę, jednak spójrzmy na ten wg mnie bardziej realistyczny przykład :

Zacznijmy od generacji indeksów :
n=arange(1,2**10)
stwórzmy wektor czasowy
dt=1/(sampling_frequency)
t=[]
for i in range(1,2**10):
t.append(n[i-1]*dt)
Mamy więc nas czas dyskretny, przyjelismy czestotliwosc probkowania, jestesmy więc w stanie sprobkowac naszą funkcję cosinus:
x=AMPLITUDA*cos(2*pi*t*CZESTOTLIWOSC) //czasami (zalezy od wersji Pythona), ta operacja wymaga uzycia petli – istnieje problem konwersji float<->integer

Może się wydawać, że prześledziłem już spory wycinek kodu a prawie nie korzystałem z NumPy, jednak w powyższym przykładzie stworzyliśmy falę cosinus, która składa się z 1024 próbek, użycie NumPy zaprocentuje nam gdy będziemy chcieli przedstawić nasze dane na wykresie. Po krótkim wstępie, możemy uważać się za użytkowników NumPy, jesteśmy w stanie stworzyć wektory czasu i wartości, wygenerować proste sygnały i obejrzeć wyniki – zachęcam do eksperymentowania i ćwiczenia.

Matplotlib

Jesteśmy w stanie przechowywać pewną ilość danych w postaci struktur dostarczonych przez Python lub przez dodatkowy moduł NumPy, więc fajnie by było gdybyśmy mogli obejrzeć wyniki naszej pracy. Matplotlib to moduł służący do tworzenia wszelkiego rodzaju wykresów i wizualizacji. Umożliwia kreślenie wykresów 2D oraz 3D, histogramów i wielu więcej diagramów. Szybkie użycie wyszukiwarki i już mamy dostęp do mnóstwa tutoriali związanych bezpośrednio z tym modułem. Ja postaram się przedstawić wg mnie najbardziej

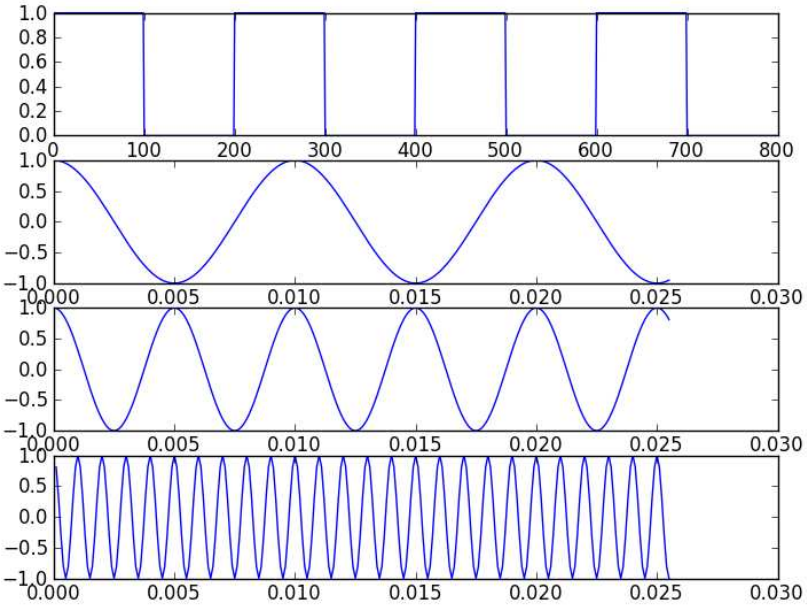
istotne przykłady, których prześledzenie pozwali nam dostrzec możliwości biblioteki zarówno jeśli chodzi o kreślenie wykresów a także jeśli chodzi o DSP ogólnie.

```
import matplotlib.pyplot as plt // zaczynamy od zaimportowania modułu
from numpy import * // dla przyszłego użycia
x1=arange(1,100)
n=arange(1,2**10)
fsampl=10000 // częstotliwość prókownaia – zachęcam do eksperymentowania
t=[]
dt=1.0/fsampl
generacja wektoru czasu:
for i in range(1,2**10):
t.append(n[i-1]*dt)
A1,A2,A3,A4=1,2,4,8 // nasze amplitudy
f1,f2,f3,f4=100,200,400,800 // I częstotliwości
x1,x2,x3,x4=[],[],[],[]
for i in range(1,2**10): x1.append(A1*cos(2*pi*f1*t[i-1]))
x2.append(A2*cos(2*pi*f2*t[i-1]))
x3.append(A3*cos(2*pi*f3*t[i-1]))
x4.append(A4*cos(2*pi*f4*t[i-1]))
Ok, generację mamy za sobą teraz zajmiemy się wykresami
plt.plot(t,x1) // prosty wykres x-y
plt.show() // biblioteka “przechowuje” dane gdzieś w pamięci I wymaga tego
polecenia do zaprezentowania nam okna z wykresem
plt.plot(t,x1,'r.-') // inne mozliwosci : plt.show() plt.plot(t,x1,'ro') plt.show()
plt.plot(t,x1,'r.-',t,x2,'b,-',t,x3,'g.-',t,x4,'y.-') // wszystkie sygnaly na jednym
wykresie – troche chaotyczne
jednak opcja jest uzyteczna jesli chcemy porownac dwa wykresy np. tak :
plt.plot(t,x1,'r.-',t,x2,'g.-')
gdzie b,g,y,r to koloty i „ .- „ – sposób w jaki przedstawimy dane
```

Ok pokazałem w jaki sposób możemy wykreślić nasze sygnały – jest to wystarczające w większości przykładów – zmieniają się sygnały i dane ale sposób prezentacji danych pozostaje ten sam. Warto wspomnieć o jeszcze jednej opcji wyświetlania danych: plt.plot(x) – można z nią poeksperymentować, jednak czasami polecenie może być zawodne.

```
import matplotlib.pyplot as plt
from numpy import *
n=arange(1,2**8)
fsampl=10000
dt=1.0/fsampl
x1,x2,x3=[],[],[]
f1,f2,f3=100,200,1000
n1=ones(100)
n2=zeros(100)
wave=hstack([n1,n2,n1,n2,n1,n2,n1,n2]) // generujemy falę podobną do unipolarnej
```

```
t=[]
for i in range(1,2**8):
t.append(n[i-1]*dt)
x1.append(cos(2*pi*f1*t[i-1]))
x2.append(cos(2*pi*f2*t[i-1]))
x3.append(cos(2*pi*f3*t[i-1]))
plt.subplot(4,1,1) // subplot pozwala na wyswietlenie osobnych wykresów w jednym oknie
plt.plot(wave) // skladnia subplot dokladnie opisana jest w dokumentajci
plt.subplot(4,1,2)
plt.plot(t,x1)
plt.subplot(4,1,3)
plt.plot(t,x2)
plt.subplot(4,1,4)
plt.plot(t,x3)
plt.show()
```



Podsumowując, matplotlib to bardzo użyteczny moduł, najlepszy jeśli chodzi o tworzenie wykresów. Dla nas będzię ono bardzo poręczne jeśli przejdziemy do prezentacji efektów działania fft czy filtracji.

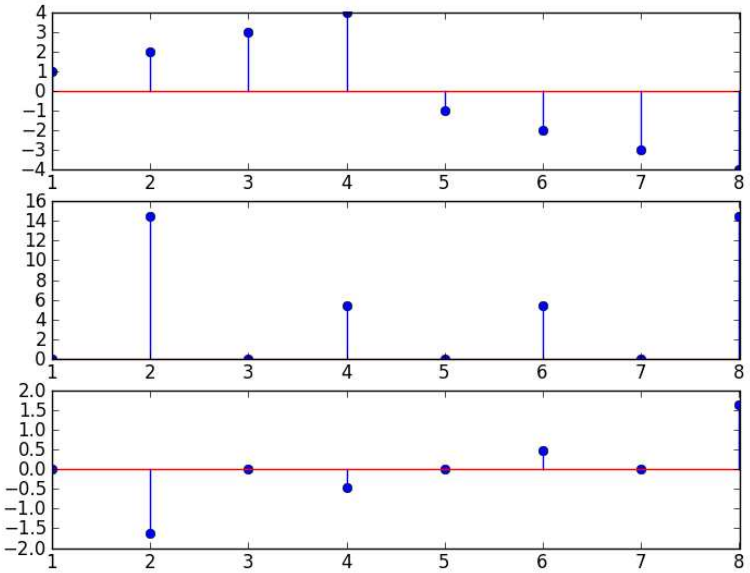
FFT, Splot, Reprezentacja Sygnałów

FFT – Fast Fourier Transform, której definicji nie zamierzam tutaj zamieszczać, daje nam informację o częstotliwościach powiązanych z naszym sygnałem. Pokrótce, transformata „przekształca" jedną funkcję w drugiej, jednak zmieniana jest dziedzina – z dziedziny czasu przechodzimy do dziedziny częstotliwości. Najwspanialszą rzeczą jeśli chodzi o FFT jest jej wydajność, w porównaniu z DFT (oryginalną definicją dyskretnej transformaty Fouriera) algorytm jest znacznie szybszy. Ideę transformaty prześledzimy na przykładzie : mając

sygnał: x = 2*cos(2*pi*1000*t)+sin(2*pi*700*t) spodziewamy się, że FFT po przeliczeniu i prezentacji wyników da nam informację o udziale częstotliwości 700 Hz i 1000 Hz. Wpierw, zobaczmy w jaki sposób możemy obliczać FFT i wyświetlać wyniki :

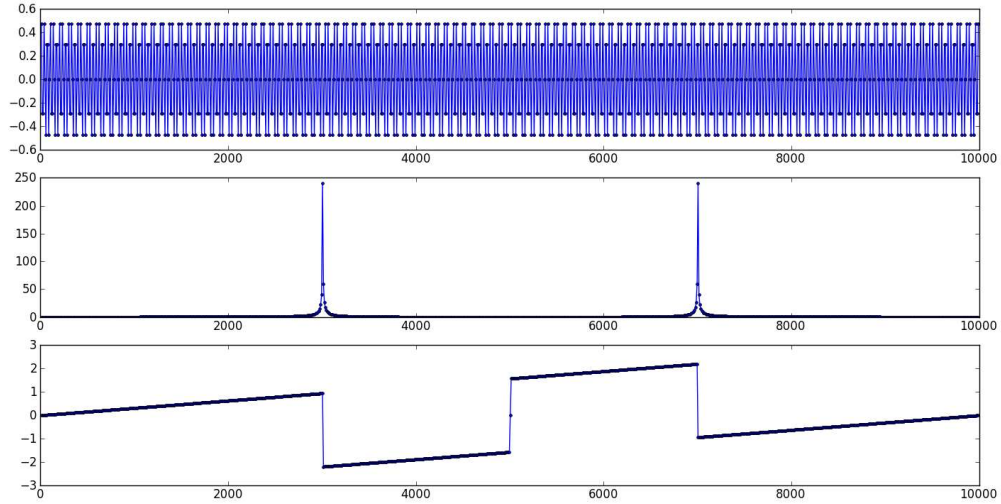
Na początek imporutjemy moduły :

```
from numpy import *
from pylab import *
from scipy.signal import *
n=arange(1,2**10+1)
fsamp=10000
dt=1.0/fsamp
df=1.0/((2**10+1)*dt)
t,f=[],[]
for i in range(1,2**10+1):
t.append(n[i-1]*dt)
f.append(n[i-1]*df)
x1=[1,2,3,4,-1,-2,-3,-4]
n1=[1,2,3,4,5,6,7,8]
x2,x3,x4=[],[],[]
for i in range(1,2**10+1): x2.append(0.5*sin(2*pi*3000*t[i-1]))
x3.append(5*cos(2*pi*2000*t[i-1]))
x4.append(2*cos(2*pi*4000*t[i-1]))
X1=fft(x1)
subplot(3,1,1)
stem(n1,x1)
subplot(3,1,2)
stem(n1,abs(X1)
subplot(3,1,3)
stem(n1,angle(X1)
show()
```



Zdjęcie pokazuje rezultaty tylko dla x1, pierwszy wykres – sygnał wejsciowy, drugi wykres – abs(fft(x1)), trzeci wykres – angle(fft(x1)). Jako, że fft daje nam wyniki zespolone istnieje pewien zwyczaj przedstawiania modułu i argumentu tych liczb zespolonych. Na drugim wykresie możemy zobaczyć kilka „peeków", które w jakiś sposób dają nam informacje o sygnale. Pierwszy wniosek – występuje pewna symetria, moduł jest parzysty, a argument nieparzysty. Jednak, w tym momencie nie jesteśmy w stanie powiedzieć jakie częstotliwości są powiązane z naszym sygnałem z powodu tego, że przedstawiliśmy wykres w postaci indeks_próbki – wartość a nie częstotliwość – wartość. Następny przykład pokaże nam w jaki sposób przedstawiać sygnały w dziedzinie częstotliwości.

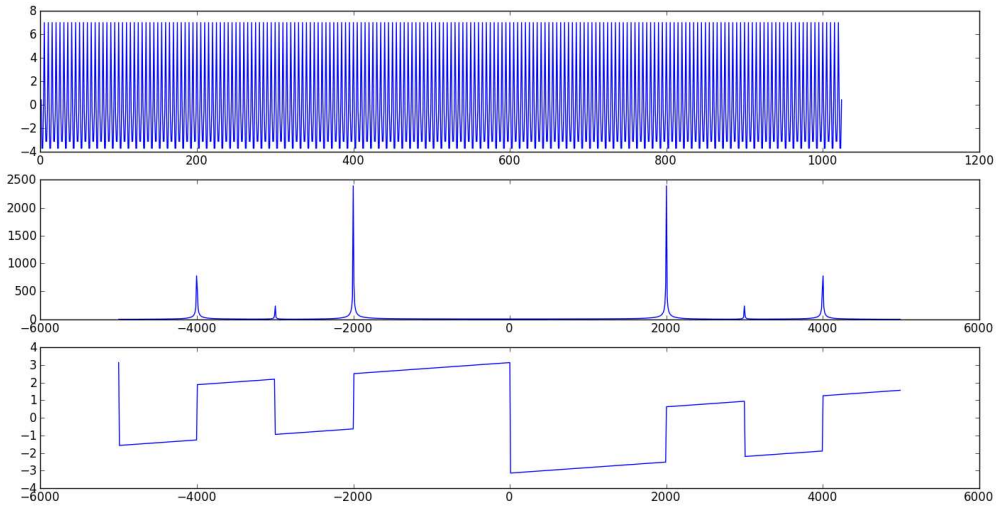
```
subplot(3,1,1)
plot(f,x2,'.-')
subplot(3,1,2)
plot(f,abs(fft(x2)),'.-')
subplot(3,1,3)
plot(f,angle(fft(x2)),'.-')
```



Teraz zajmujemy się x2 – falą cosinusową, której częstotliwość wynosi 3000 Hz. Z powodu tak wysokiej częstotliwości wykres jest trochę nieczytelny. Teraz możemy zauważyć dwa „peeki" w dziedzinie częstotliwości jeden w pobliżu właśnie 3000 Hz a drugi 7000 Hz. Problem jest bardzo powszechny i jest spowodowany zjawiskiem symetrii fft oraz jej okresowością. Pokrótce, pierwszy prążek reprezentuje naszą częstotliwość główną f0 a następny fp-f0=10000 – 3000 = 7000 Hz. Więc środkiem symetrii naszego wykresu jest częstotliwość fsampling/2 = 5000 Hz. Ze względów estetycznych wolelibyśmy aby środek symetrii znajdował się 0 – żeby w łatwiejszy sposób interpretować wyniki.

```
x=[]
for i in range(1,2**10+1):
x.append(x2[i-1]+x3[i-1]+x4[i-1])
1024 – punktowe fft
w=fftfreq(1024)*fsamp #generacja i normalizacja wektora częstotliwoścIa z wykorzystaniem częstotliwości próbkowania
X=fft(x,1024) # 1024-FFT – FFT, które wylicza 1024 próbki
X=fftshift(X) # operacja „shirtingu" /wartości
w=fftshift(w) # operacja „shirtingu" /wektor częstotliwości
```

```
subplot(3,1,1)
plot(n,x,'r.-')
subplot(3,1,2)
plot(w,abs(X))
subplot(3,1,3)
plot(w,angle(X))
show()
```



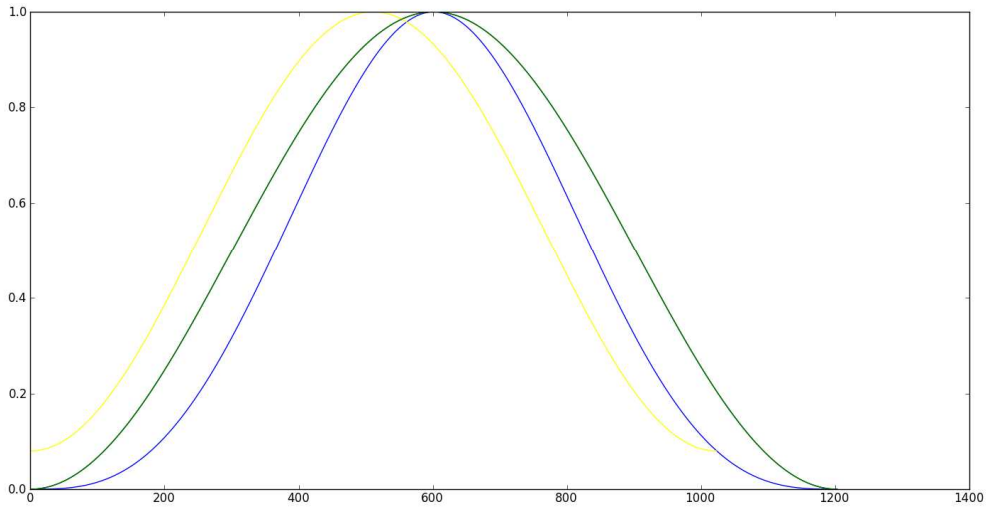
Teraz wykonaliśmy operację sumowania trzech sygnałów, mamy więc do czynienia z trzema częstotliwościami 2000,3000 oraz 4000 Hz. Po wykonaniu 1024-FFT i operacji shftingu możemy faktycznie stwierdzić, że te częstotliwości reprezentują nasz sygnał w domenie częstotliwości. Jesteśmy więc w stanie wykonać analizę spektralną praktycznie każdego sygnału (warto poeksperymentować z zaszumieniem) oraz zaprezentować rezultaty.

Przez chwilę porozmawiajmy o splocie. Splot to podstawowy proces jeśli chodzi o filtrację (w zasadzie to on realizuję operację filtracji) sygnałów audio czy obrazów. Jednak inżynierowie jak to inżynierowie nie lubią operacji, której reprezentacja jest nieczytelna, obliczanie żmudne i w ogóle nie można o niej mówić „user-friendly”. ALE ! Mamy nasze FFT, a jedną z zalet FFT, lub generalnie transformaty Fouriera jest to, że operację splotu w dziedzinie czasu możemy przedstawić jako mnożenie transformat sygnałów w dziedzinie częstotliwości. Zachęcam do prześledzenia poniższego przykładu, warto wspomnieć, że są dwa rodzaje splotu – splot liniowy oraz kołowy. FFT oraz odwrotna do FFT IFFT realizują operację splotu kołowego, jeśli chodzi o splot liniowy troszkę trudniej się z nim „zmaga”.

```
convolve(x1,x2) #splot liniowy dwóch sygnałów
fftconvolve(x1,x2) #where x1,x2 - arrays, fftconvolve mogą być znalezione w scipy.signal.signaltools
x1=[ ... ], x2 =[ ... ], x=fft(x1)*fft(x2) x=ifft(x) #tak wykonujemy splot kołowy
PODOBNIE BĘDZIE W PRZYPADKU KORELACJI SYGNAŁÓW !
```

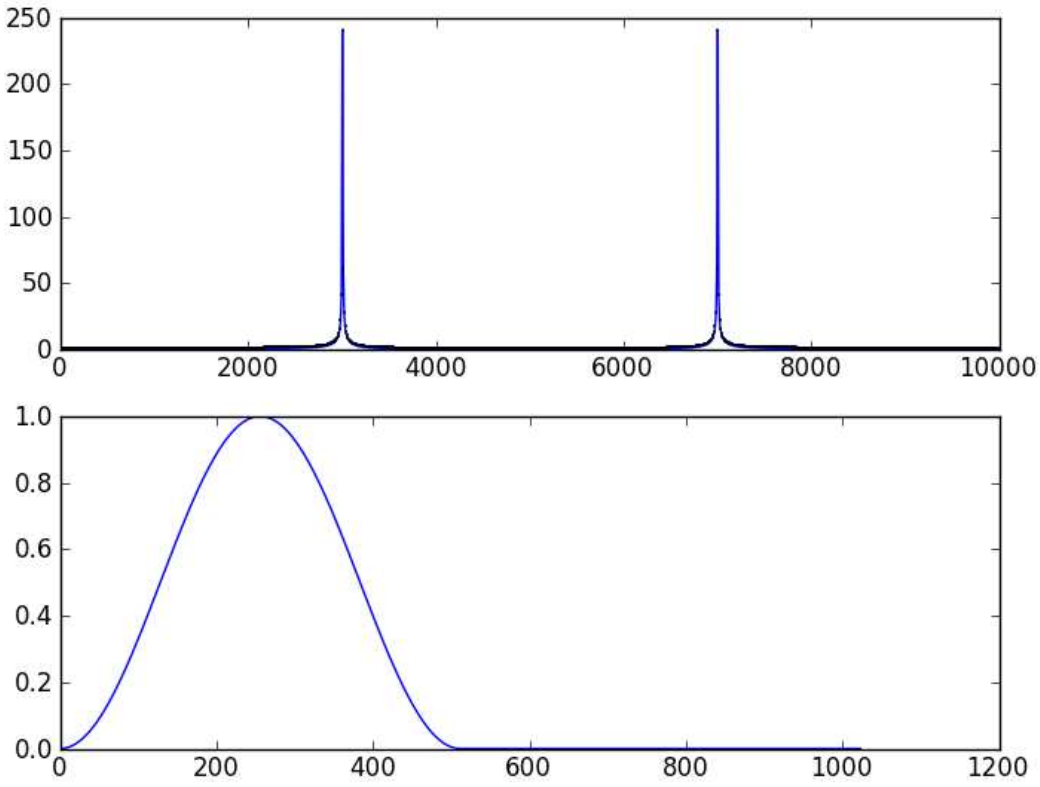
Okna

Okna często używane są razem z FFT. Nie wiem czy pamiętacie, ale z powodu symetrii fft i okresowości sygnału obserwowaliśmy prążki w miejscach niekoniecznie nas interesujących. Rozwiązaliśmy problem stosując FFT-1024 oraz fftshift, innym pomysłem jest zastosowanie okna względem sygnału tak by „wyciąć" pożądaną część. Poniżej można zobaczyć obraz prezentujący kilka rodzajów okien
ŻÓŁTY – Hamming, ZIELONY – Hann, NIEBIESKI – Bohmann, długość okien 1024
Widzimy, że okna mimo podobieństw jednak w jakiś sposób się różnią, rodzaj okna ma wpływ na parametry sygnału wyjściowego – otrzymanego po zastosowaniu okna.

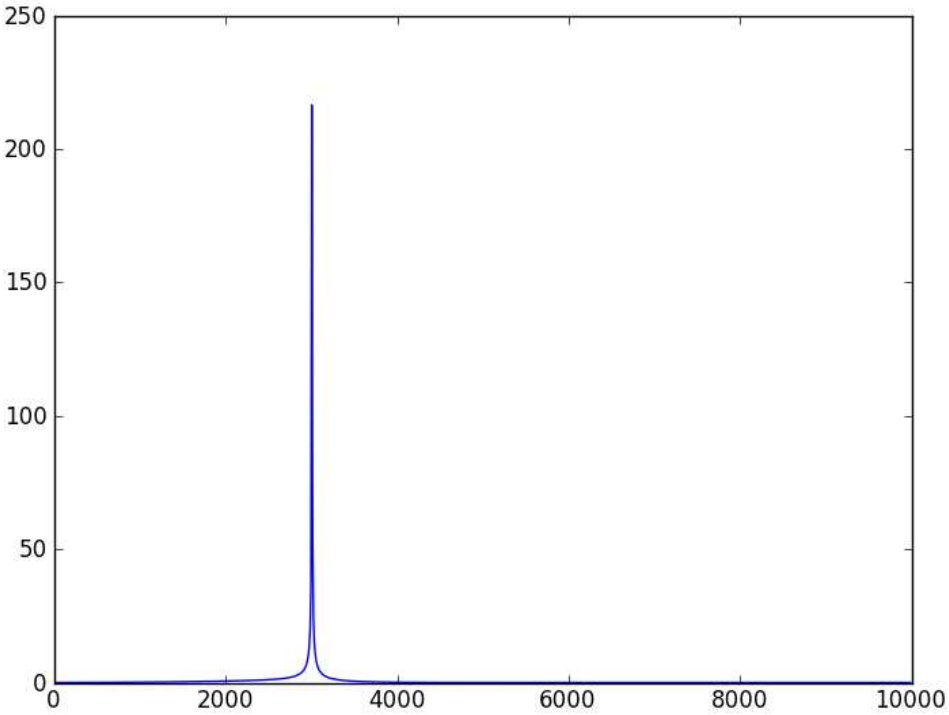


Przykładem praktycznym będzie zastosowanie okien do prezentacji wyników FFT :

```
import scipy.signal as tool
X2=fft(x2)
window=tool.hann(512)
zeros=zeros(512)
fun=hstack([window,zeros])
subplot(2,1,1)
plot(f,abs(X2)),'.-')
subplot(2,1,2)
plot(fun)
X2=X2*fun
plot(f,abs(X2)
show()
```



Po zastosowaniu okna :



Co dalej

Moim zamiarem było zachęcenie do stosowania open-sourceowych narzędzi do operacji związanych z cyfrowym przetwarzaniem sygnałów oraz popularyzacja języka Python. Generalnie środowisko i społeczność związana z tym językiem zawzięcie promuje i promuje bezpłatne aplikacje i wsparcie dla ludzi związanych z dziedziną IT oraz użytkowników narzędzi czy aplikacji. Również mimo poszukiwań nie znalazłem żadnego samouczka ani artykułu traktującego o przetwarzaniu sygnałów z pomocą Pythona w języku polskim.

Podsumowując, zachęcam do pogłębiania wiedzy z zakresu Pythona i programowania ogólnie a także tematyki dsp. Rozwiązanie zaprezentowane przeze mnie jest moim zdaniem znacznie lepsze (a na pewno nie gorsze) niż toolbox Matlaba – mamy większą kontrolę nad implementacją, możemy opakować nasz skrypt w odpowiednie funkcje i klasy i storzyć mini-toolbox do przetwarzania sygnałów. Ponadto zerowym nakładem pieniężnym i podobnym czasowo uzyskujemy takie same rezultaty.

Na koniec sugeruję i gorąco zachęcam napisanie czegoś przy użyciu tego języka – jego możliwości i prostota są zaskakujące. Jeśli ktoś jest zainteresowany przetwarzaniem sygnałów, ale w postaci 2D tj. obrazów to zachęcam do spojrzenia na Python Imaging Library.