

# Analisi del sound processing e del funzionamento di un sintetizzatore



## Obiettivi del progetto

- Studio della sintesi del suono
- Provare a ricreare un sintetizzatore ed effetti di base per analizzarli
- Confrontare il tutto con gli strumenti utilizzati sul campo

## Metodi

- *Signal* da numpy per generare forme d'onda differenti
- Trasformate di *Fourier*, *find\_peaks* di signal e *hz\_to\_note* di librosa per visualizzare i picchi e le note ad essi associati
- Ableton Live 11, effettistica inclusa e un emulatore di sintetizzatore Arturia

```
In [1]: import numpy as np
import scipy
from scipy import signal
import pandas as pd
import matplotlib.pyplot as plt
import IPython
%matplotlib inline

sample_rate = 44100
duration = 0.25
t = np.linspace(0, duration, int(sample_rate * duration))
```

## Esperimenti

### Come viene generato un suono?

Un sintetizzatore può produrre suoni fondamentali o "onde" di base. Le onde più comuni includono:


- Onde sinusoidali: Sono onde puramente tonali e prive di armonici.
- Onde quadrate: Caratterizzate da un suono ricco di armonici pari.
- Onde a sega: Hanno una serie di armonici in crescita o decrescita.
- Onde triangolari: Simili alle onde a sega, ma con meno armonici.

## Ricreiamo un semplice sintetizzatore

Con l'aiuto di scipy, nello specifico di signal, sono riuscito ad ottenere una funzione che genera forme d'onda differenti in base ai parametri di input

```
In [2]: def synth(form, note, duration):
        x = np.linspace(0, duration, sample_rate * duration)
        if form == 'square':
            y = signal.square(2 * np.pi * note * x)
        elif form == 'sine':
            y = np.sin(2 * np.pi * note * x)
        elif form == 'sawtooth':
            y = signal.sawtooth(2 * np.pi * note * x)
        elif form == 'triangle':
            y = np.abs(signal.sawtooth(2 * np.pi * note * x * 0.5))
        return y

        audio_sq = synth('square', 440, 1) #A
        IPython.display.Audio(audio_sq, rate=sample_rate)
```

Out[2]: 

## Analisi delle possibili forme d'onda

```
In [3]: fig, ((pl1, pl2), (pl3, pl4)) = plt.subplots(2, 2, figsize=(15, 6))

        pl1.plot(audio_sq)
        pl1.set_xlim(0, 250)
        pl1.set_xlabel('Sample')
        pl1.set_ylabel('Amplitude')
        pl1.set_title('Square Wave')

        #Generiamo anche un segnale sinusoidale
        audio_sin = synth('sine', 440, 1) #A

        pl2.plot(audio_sin, color='red')
        pl2.set_xlim(0, 250)
        pl2.set_xlabel('Sample')
        pl2.set_ylabel('Amplitude')
        pl2.set_title('Sine Wave')

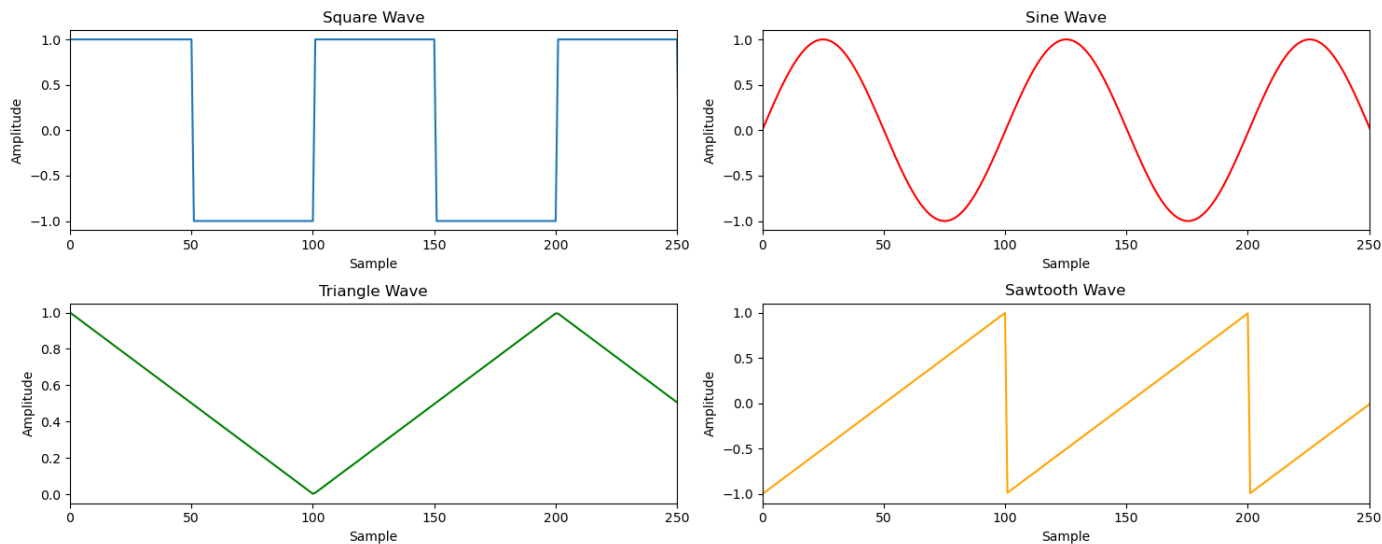
        #Generiamo anche un segnale triangolare
        audio_tri = synth('triangle', 440, 1) #A

        pl3.plot(audio_tri, color='green')
        pl3.set_xlim(0, 250)
        pl3.set_xlabel('Sample')
        pl3.set_ylabel('Amplitude')
        pl3.set_title('Triangle Wave')
```

```
#Generiamo anche un segnale a dente di sega
audio_saw = synth('sawtooth', 440, 1) #A

pl4.plot(audio_saw, color='orange')
pl4.set_xlim(0, 250)
pl4.set_xlabel('Sample')
pl4.set_ylabel('Amplitude')
pl4.set_title('Sawtooth Wave')

fig.tight_layout()
```



## Come facciamo ad avere la nota che vogliamo?

E' più semplice di quanto sembra, visto che quelle che noi chiamiamo note non sono altro che frequenze differenti

NOTE FREQUENCY CHART | HEROIC AUDIO

	Octave 0	Octave 1	Octave 2	Octave 3	Octave 4	Octave 5	Octave 6	Octave 7	Octave 8	Octave 9	Octave 10
C	16.35	32.70	65.41	130.81	261.63	523.25	1046.50	2093.00	4186.01	8372.02	16744.04
C#	17.32	34.65	69.30	138.59	277.18	554.37	1108.73	2217.46	4434.92	8869.84	17739.69
D	18.35	36.71	73.42	146.83	293.66	587.33	1174.66	2349.32	4698.64	9397.27	18794.55
D#	19.45	38.89	77.78	155.56	311.13	622.25	1244.51	2489.02	4978.03	9956.06	19912.13
E	20.60	41.20	82.41	164.81	329.63	659.26	1318.51	2637.02	5274.04	10548.08	
F	21.83	43.65	87.31	174.61	349.23	698.46	1396.91	2793.83	5587.65	11175.30	
F#	23.12	46.25	92.50	185.00	369.99	739.99	1479.98	2959.96	5919.91	11839.82	
G	24.50	49.00	98.00	196.00	392.00	783.99	1567.98	3135.96	6271.93	12543.86	
G#	25.96	51.91	103.83	207.65	415.30	830.61	1661.22	3322.44	6644.88	13289.75	
A	27.50	55.00	110.00	220.00	440.00	880.00	1760.00	3520.00	7040.00	14080.00	
A#	29.14	58.27	116.54	233.08	466.16	932.33	1864.66	3729.31	7458.62	14917.24	
B	30.87	61.74	123.47	246.94	493.88	987.77	1975.53	3951.07	7902.13	15804.26	

```
In [4]: chord_notes = [440, 554.37, 659.26] # A, C#, E
chord_duration = 2 #durata in secondi
```

```

chord_audio = np.zeros(sample_rate * chord_duration)
for note in chord_notes:
    chord_audio += synth('square', note, chord_duration)

#Aggiungiamo un sub-bass
chord_audio += synth('sine', 110, chord_duration)
#Aggiungiamo un alto sawtooth
chord_audio += synth('sawtooth', 880, chord_duration)
IPython.display.Audio(chord_audio, rate=sample_rate)

```

Out[4]:



## Fondamenti di sound processing

Prima di poter processare il suono dobbiamo capire "dove siamo" nello spettro delle frequenze

```

In [5]: from scipy.signal import find_peaks
import librosa

# Applichiamo una trasformata di Fourier
chord_freq = np.fft.fftfreq(len(chord_audio), 1/sample_rate)
chord_audio_transf = np.fft.fft(chord_audio/len(chord_audio))
chord_audio_transf = np.fft.fftshift(chord_audio_transf)
chord_freq = np.fft.fftshift(chord_freq)
N = len(chord_audio)
f = sample_rate/N * np.arange(N)

fig, (pl1, pl2) = plt.subplots(1, 2, figsize=(12, 3))

pl1.plot(chord_freq, np.abs(chord_audio_transf))
pl1.set_xlim(0, 20000)
pl1.set_xlabel('Frequency (Hz)')
pl1.set_ylabel('Amplitude')

#vediamolo più da vicino

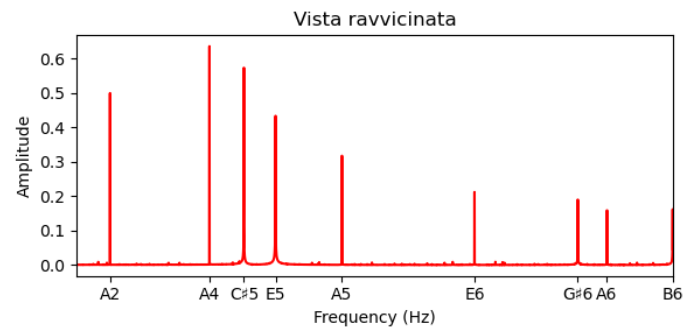
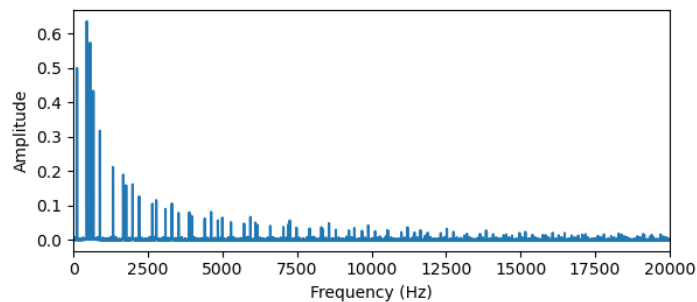
#Troviamo i picchi
peaks, _ = find_peaks(np.abs(chord_audio_transf), height=0.15)
#Tengo solo i picchi positivi
peaks = peaks[chord_freq[peaks] > 0]

pl2.plot(chord_freq, np.abs(chord_audio_transf), color='red')

pl2.set_title('Vista ravvicinata')
pl2.set_xlim(0, 1000)
pl2.set_xlabel('Frequency (Hz)')
pl2.set_ylabel('Amplitude')
pl2.set_xticks(chord_freq[peaks], labels=librosa.hz_to_note(chord_freq[peaks]))

fig.tight_layout()
plt.show()

```



Grazie alla funzione `hz_to_note()` di `librosa`, possiamo osservare come i picchi corrispondano alle frequenze relative alle note dell'accordo. Notiamo inoltre come il segnale non sia in realtà composto da solo le tre note "suonate", ma da queste (e non solo) che vengono ripetute periodicamente

## Applicazione di effetti di base

Applichiamo un filtro passa banda, nello specifico terremo tutto tra i 200 e i 15k Hz

```
In [6]: #Tagliamo sotto i 15000Hz e sopra i 200Hz
ftfilt = chord_audio_transf.copy()
indici = np.abs(chord_freq) > 15000
indice2 = np.abs(chord_freq) < 200
ftfilt[indici] = 0
ftfilt[indice2] = 0

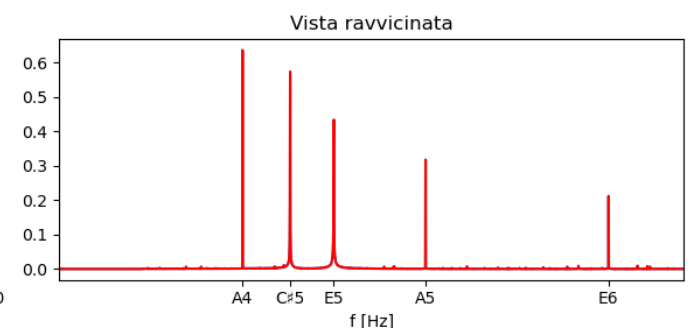
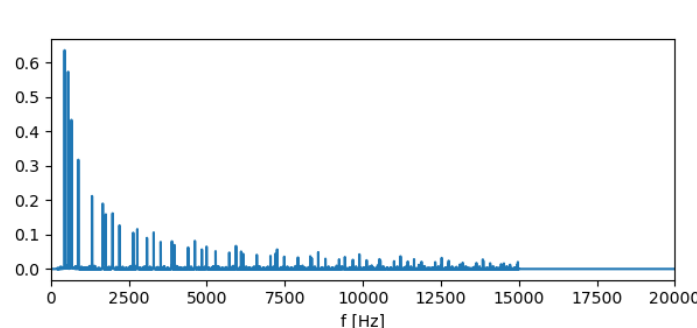
#Stampa del grafico per i confronti
fig, (pl1, pl2) = plt.subplots(1, 2, figsize=(12, 3))

#Vediamolo da vicino
pl1.plot(chord_freq, np.abs(ftfilt))
pl1.set_xlim(0, 20000)
pl1.set_xlabel('f [Hz]')

#Vediamo i picchi
peaks, _ = find_peaks(np.abs(ftfilt), height=0.2)
#Tengo solo i picchi positivi
peaks = peaks[chord_freq[peaks] > 0]
pl2.plot(chord_freq, np.abs(ftfilt))

pl2.set_title('Vista ravvicinata')
pl2.plot(chord_freq, np.abs(ftfilt), color='red')
pl2.set_xlabel('f [Hz]')
pl2.set_xlim(0, 1500)
pl2.set_xticks(chord_freq[peaks], labels=librosa.hz_to_note(chord_freq[peaks]))

fig.tight_layout()
```



E' evidente l'assenza del picco a 110Hz, corrispondente alla nota di basso dell'accordo, ovvero A1

# Analisi di un "vero" sintetizzatore



Usando Ableton Live, una digital audio workstation, ho suonato lo stesso accordo con un emulatore di sintetizzatore per poi applicarci degli effetti di base

```
In [7]: #importiamo un accordo da un sintetizzatore
sample_rate, juno = scipy.io.wavfile.read('juno_A.wav')
juno = juno[:,0]
juno = juno/np.max(juno)
#IPython.display.Audio(juno, rate=sample_rate)
```

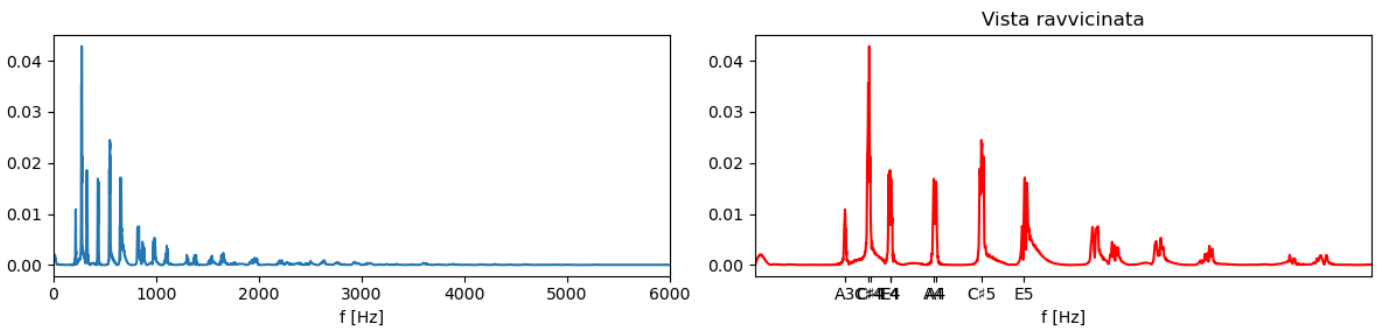
```
In [11]: #Applichiamo una trasformata di Fourier
juno_freq = np.fft.fftfreq(len(juno), 1/sample_rate)
juno_freq = np.fft.fftshift(juno_freq)
juno_transf = np.fft.fft(juno/len(juno))
juno_transf = np.fft.fftshift(juno_transf)

#Stampa del grafico per i confronti
fig, (p11, p12) = plt.subplots(1, 2, figsize=(12, 3))
#Visualizziamo lo spettro
p11.plot(juno_freq, np.abs(juno_transf))
p11.set_xlabel('f [Hz]')
p12.set_title('Sintetizzatore senza filtri')
p11.set_xlim(0, 6000)

#Vediamolo da vicino
#Troviamo i picchi
peaks, _ = find_peaks(juno_transf, height=0.008)
#Tengo solo i picchi positivi
peaks = peaks[juno_freq[peaks] > 0]
#Teniamolo soli i picchi fino a 1500Hz
peaks = peaks[juno_freq[peaks] < 1500]

p12.plot(juno_freq, np.abs(juno_transf), color = 'red')
p12.set_xlabel('f [Hz]')
p12.set_title('Vista ravvicinata')
p12.set_xlim(0, 1500)
p12.set_xticks(juno_freq[peaks], labels=librosa.hz_to_note(juno_freq[peaks]))
```

```
fig.tight_layout()
```



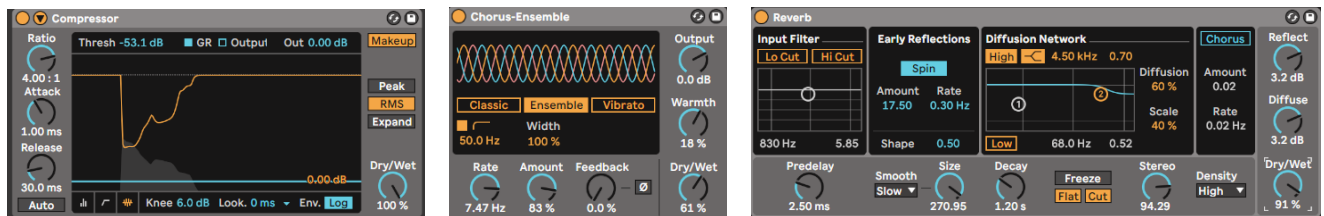
Le apparenti imprecisioni del grafico di destra sono dovute all'imprecisione del sintetizzatore utilizzato: avendo diverse note uguali ma allo stesso tempo leggermente "stonate", ha rilevato più picchi vicini

Possiamo quindi notare come il suono risulti meno preciso rispetto a quello da noi generato precedentemente, questo perché i sintetizzatori cercano di generare un suono più musicale e tendono a "sporcicare" il segnale per rendere il risultato più naturale

## Analisi del sound processing "in studio"

Ora applichiamo degli effetti digitali inclusi in una digital audio workstation.

Guarderemo in dettaglio: **compressore**, **riverbero** e **chorus**



Screenshot effettivi dei filtri applicati

## Compressore

Cosa ci aspettiamo?

- Un intervento sulla dinamica del suono, avvicinando i picchi e i livelli più bassi del campione

```
In [9]: #importiamo il suono dopo aver applicato un compressore
sample_rate, juno_comp = scipy.io.wavfile.read('juno_compresso.wav')
juno_comp = juno_comp[:,0]
juno_comp = juno_comp/np.max(juno_comp)
IPython.display.Audio(juno_comp, rate=sample_rate)

#Osserviamo la differenza tra i due suoni
#Dinamica perchè il compressore agisce su di essa
fig, (p11, p12) = plt.subplots(1, 2, figsize=(12, 3))
p11.plot(juno)
p11.set_xlabel('Sample')
p11.set_ylabel('Amplitude')
p11.set_ylim(-1, 1)
p11.set_title('Sintetizzatore senza filtri')

p12.plot(juno_comp, color='red')
p12.set_xlabel('Sample')
```

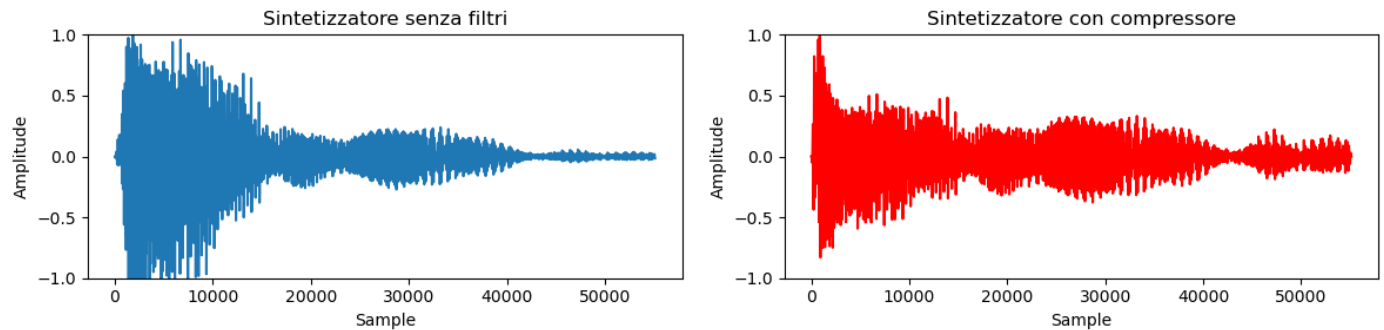


```

p12.set_ylabel('Amplitude')
p12.set_ylim(-1, 1)
p12.set_title('Sintetizzatore con compressore')

fig.tight_layout()

```



## Osservazioni:

- Il suono risulta effettivamente essere più compatto, diminuendo la distanza tra i punti più bassi e i picchi del campione
- Abbiamo comunque un picco iniziale, questo perchè il valore "Attack" permette di decidere con quanta prontezza il compressore agisce
- L'accordo risulta naturale ma con una "coda" o sustain più accentuata
- Se non avessi selezionato l'opzione "Makeup", il suono sarebbe risultato più basso. Questa opzione permette di recuperare l'intensità persa durante la compressione
- Non serve osservare la trasformata di Fourier perchè abbiamo una modulazione tonale trascurabile

**Il compressore permette quindi di aumentare il dettaglio del campione, portando in primo piano anche suoni più lievi che altrimenti potrebbero venire persi**

## Chorus

Cosa ci aspettiamo?

- Modulazione del campione e con ritardi leggere per simulare una molteplicità di suoni simili, proprio come un coro

```

In [10]: #Applichiamo effetto chorus
sample_rate, juno_chorus = scipy.io.wavfile.read('juno_chorus.wav')
juno_chorus = juno_chorus[:,0]
juno_chorus = juno_chorus/np.max(juno_chorus)
#IPython.display.Audio(juno_chorus, rate=sample_rate)

#Osserviamo la differenza tra i due suoni
#Fourier perchè il chorus agisce su distorsione tonale e delay
juno_chorus_transf = np.fft.fft(juno_chorus/len(juno_chorus))
juno_chorus_transf = np.fft.fftshift(juno_chorus_transf)
juno_freq = np.fft.fftfreq(len(juno_chorus), 1/sample_rate)
juno_freq = np.fft.fftshift(juno_freq)

juno_comp_transf = np.fft.fft(juno_comp/len(juno_comp))
juno_comp_transf = np.fft.fftshift(juno_comp_transf)

```



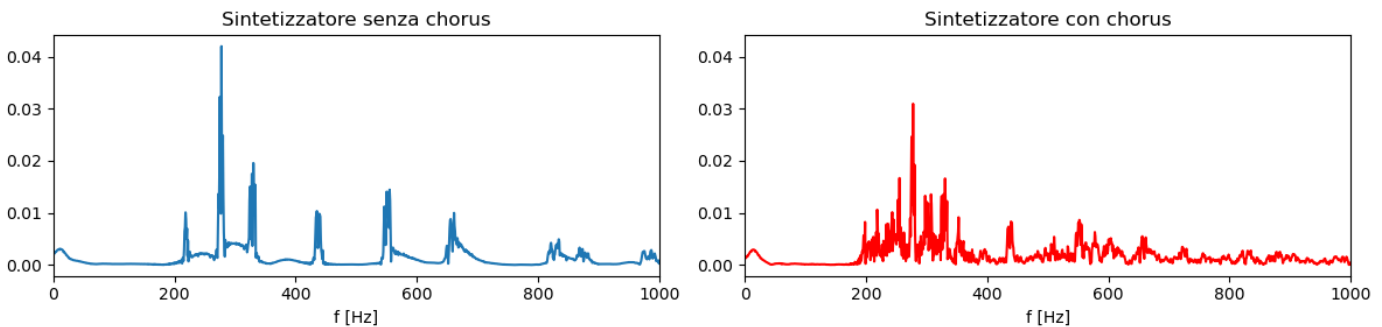
```

fig, (p11, p12) = plt.subplots(1, 2, figsize=(12, 3))
p11.plot(juno_freq, np.abs(juno_comp_transf))
p11.set_xlabel('f [Hz]')
p11.set_title('Sintetizzatore senza chorus')
p11.set_xlim(0, 1000)
p11.sharey(p12)

p12.plot(juno_freq, np.abs(juno_chorus_transf), color='red')
p12.set_xlabel('f [Hz]')
p12.set_title('Sintetizzatore con chorus')
p12.set_xlim(0, 1000)

fig.tight_layout()

```



### Osservazioni:

- La trasformata di Fourier enfatizza l'evidente modulazione dal punto di vista tonale, che arricchisce il suono
- I picchi principali rimangono gli stessi, infatti l'effetto non snatura completamente il suono, ma crea la cosiddetta "texture"

**Il chorus permette di modificare più pesantemente il suono, aumentando quindi la quantità effettiva di informazione contenuta nel segnale**

## Conclusioni

L'idea iniziale di creare un sintetizzatore completo DIY avrebbe significato anche implementare i filtri che abbiamo visto ma avrebbero richiesto un progetto ciascuno. Ho ritenuto altrettanto valido dare un'infarinatura di base, in quanto *sound designer da cameretta* sul processo dietro alla sintesi dei suoni, dalla loro nascita ad alcuni semplici ma interessanti effetti ai quali vengono sottoposti nel processo di sound design (nella presentazione ho dovuto scartare il riverbero, il meno "sorprendente di tutti") Con questo progetto spero di aver mostrato all'ascoltatore il processo per il quale i segnali audio passano, prima di trovare spazio nelle canzoni che ascoltiamo tutti i giorni