CompEssay brageat

April 29, 2022

1 Hvor mange wavelets trengs for å avskrekke Beethovens 5. symfoni?

1.0.1 Et Computational Essay av Brage A. Trefjord

Merknad: Formulering av problemstilling, og utvikling av kode for å løse problemstillingen er gjort i samarbeid med Sigurd Vargdal.

1.1 Introduksjon

I 1808 komponerte Beethoven en symfoni populært kalt "Skjebnesymfonien", eller Beethovens 5. symfoni.[1] Vi har blitt veldig glade i denne symfonien, men skjelver hver gang den sier "DUN DUN DUN DUUUUUN". Vi synes denne delen er altfor skummel, og ønsker å fjerne alle forekomstene fra sangen.

Vi har altså følgende problemstilling: Hvor mange wavelets trenger vi for å fjerne alle tilfellene av "DUN DUN DUN DUUUUUN" fra Beethoven's 5. symfoni?

Gjennomføringen av dette skal vi dele opp i flere steg. Først skal vi sample en lydfil med kun "dun dun dun" delen, og gjøre en Fourieranalyse av denne for å finne ut hvilke frekvenser den består av. Deretter skal vi gjøre det samme med et lengre utdrag av den fullstendige symfonien, og fjerne frekvensene til "dun dun dun" fra symfonien. Deretter skal vi bruke wavelets for å forbedre analysen ved å kun fjerne frekvensene fra de stedene i sangen hvor "dun dun dun"-delen faktisk oppstår.

1.2 Metode

Vi begynner med å gjøre noen nødvendige importer.

[1]:
```
# Nødvendige importer
from matplotlib import style
style.use("default") # Gir finere plot for .ipynb filer i VS Code
import matplotlib.pyplot as plt
import numpy as np
from numba import jit # Denne brukes for å effektivisere koden vår, ettersom␣
 ↪spesielt wavelets er veldig beregningstungt
from scipy.io import wavfile # Denne bruker vi til å lese og skrive lydfiler (.
 ↪wav)
import IPython # Nødvendig for å spille av lydfiler i notebooken
import warnings
```

```
warnings.filterwarnings("ignore") # Ignorerer advarsler fra numba
```

2 Sampling av lydsignal

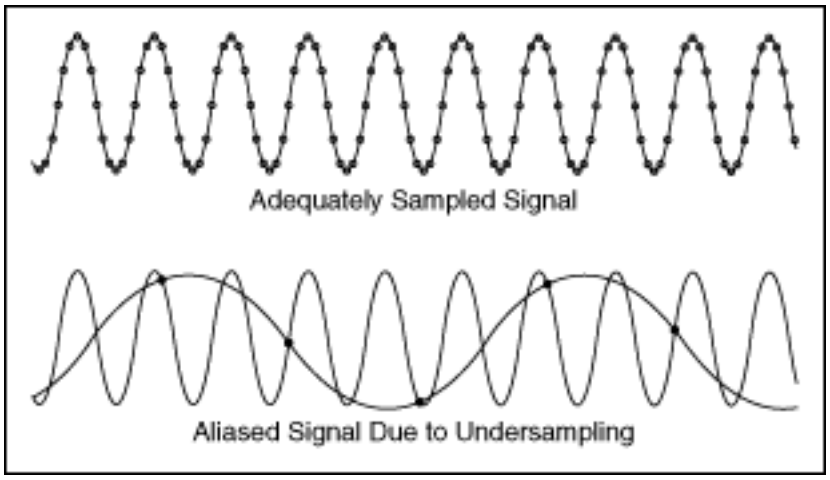
Når vi har en lydfil kan vi gjøre en sampling av den, altså går vi gjennom lydfilen og måler utslaget ved forskjellige tidspunkt t. Antall målepunkter per sekund i lydfilen kalles samplingsfrekvensen f_s. Når vi gjør en sampling av lydfilen er det viktig at samplingsfrekvensen vi bruker er minst dobbelt så stor som den største frekvensen i signalet. Altså må

$$f_s > 2f_{max}$$

hvor f_{max} er den største frekvensen i signalet. Den største frekvensen vi (med god presisjon) kan sample med en samplingsfrekvens f_s kalles Nyquist-frekvensen f_N. Det følger direkte fra samplingsteoremet at denne frekvensen må være halvparten av samplingsfrekvensen:

$$f_N = \frac{f_s}{2}$$

Dersom vi prøver å sample et lydsignal med en samplingsfrekvens som er for lav vil vi få aliasing. Når vi da analyserer signalet videre kommer vi til å jobbe med feil signal, med feil frekvens. Dette er illustrert i følgende bilde:



Dette bildet er hentet fra [2]. I øverste signal ser vi at det er mange målepunkter (høy samplingsfrekvens). Prøver vi å tilpasse en kurve til disse målepunktene ser vi at resultatet vil bli identisk med signalet vi sampler, altså er dette en god sampling. I signalet under derimot har vi svært få målepunkter (lav samplingsfrekvens). Her er det tilpasset en kurve til disse målepunktene, og den resulterende kurven fra denne samplingen har en mye lavere frekvens enn signalet som ble samplet. Her har det altså skjedd aliasing, og samplingen stemmer ikke overens med signalet.

Vi skal nå skrive en funksjon som skal lese av en lydfil, og gjøre en sampling av den. Funksjonen tar inn filnavnet til lydfilen, og skal returnere tre verdier. Tidsverdiene t til lydsignalet, det samplede lydsignalet x_n, og samplingsfrekvensen f_s.

[2]:
```
@jit
def readWav(filename):
    ## Leser lydfilen med navnet filename, og returnerer tidpunkter t, signal␣
 ↪x_n og samplingsfrekvens fs
    fs, data = wavfile.read(filename) # Henter samplingsfrekvens og signaldata␣
 ↪fra lydfilen
    x_n = data[:,0] # Henter ut relevant del av signalet
    N = len(x_n) # lengden på lyd signalet
    T = N/fs #lengden på tidsarrayen
    t = np.linspace(0, T, N) # Lager en array med tidpunkter
    return t, x_n, fs # returnerer tidsarrayet, lydsignalet og␣
 ↪samplingsfrekvensen
```

La oss nå se på en ganske kort lydfil "DDD.wav". Denne lydfilen inneholder de første sekundene av Beethoven's 5. symfoni, hvor sangen sier "dun dun dun duuun" to ganger.

[3]:
```
IPython.display.Audio("DDD.wav") # Spiller av lydfilen
```

[3]:
```
<IPython.lib.display.Audio object>
```

Vi kan nå gjøre en sampling av DDD.wav med funksjonen vi skrev.

[4]:
```
lydfil_ddd = "DDD.wav" # Lydfilen med kun "DUN DUN DUN DUUUN"-delen av␣
 ↪symfonien.
t_ddd, x_n_ddd, fs_ddd = readWav(lydfil_ddd) # Henter data fra lydfilen.
print(f"fs = {fs_ddd}") # Printer samplingsfrekvensen
```

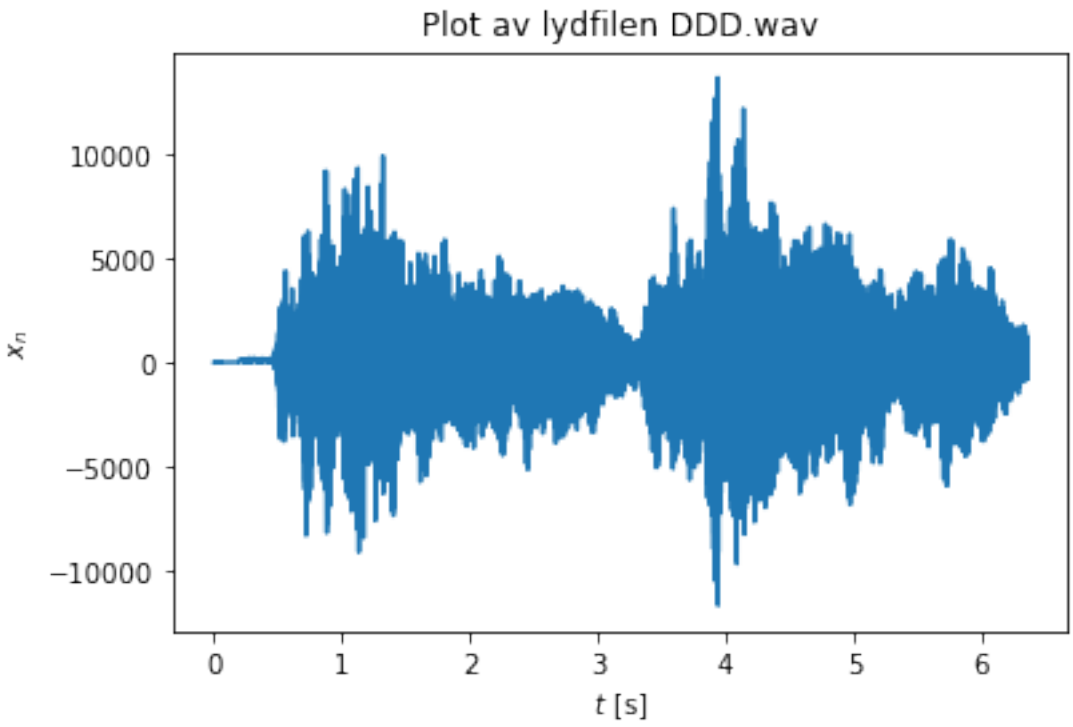
```
fs = 44100
```

Her ser vi at samplingsfrekvensen som blir brukt er $f_s = 44\,100$ Hz. Vi kommer til å se at dette er høyere enn de relevante frekvensene fra selve lydfilen. Grunnen til at scipy har valgt denne frekvensen for å lese lydfiler kommer nok fra frekvensspekteret vi mennesker kan høre. De høyeste frekvensene mennesker kan høre (uten betydelig ubehag) er på omtrent 20 000 Hz (Denne informasjonen er hentet fra [3]). En samplingsfrekvens på 44 100 Hz er altså godt over det dobbelte av denne maksimale hørselsfrekvensen. Etterhvert som vi skal gjøre frekvensanalyse av lydfilene våre kommer vi til å se at de fleste frekvensene vi møter vil være godt under 20 000 Hz.

La oss plotte det samplede lydsignalet

[5]:
```
## Plotter lydsignalet fra DDD
plt.title(f"Plot av lydfilen {lydfil_ddd}")
plt.xlabel("$t$ [s]")
plt.ylabel("$x_n$")
plt.plot(t_ddd, x_n_ddd)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x7f888d3299d0>]
```



Å lese en lydfil på denne måten er vanskelig, men vi kan se at vi har to ganske like sekvenser, den første fra omtrent $t = 0.5$ til $t = 3.25$, og den andre fra omtrent $t = 3.25$ til $t = 6.25$. Disse to sekvensene tilsvarer antakelig hver sin "dun dun dun duuun".

2.1 Fourieranalyse

Når vi har en samplet lydfil kan vi gjøre en Fourieranalyse på den for å finne ut hvilke frekvenser den består av. I matematikk er en Fouriertransformasjon en transformasjon hvor vi mapper en funksjon av rom eller tid til en funksjon av frekvenser. I vårt tilfelle ser vi på lydsignaler som er funksjoner av tid. Vi skal altså bruke en Fouriertransformasjon til å mappe signalet vårt over i et "frekvensrom", hvor vi kan se hvilke frekvenser som forekommer i signalet. Den kontinuerlige Fouriertransformasjonen til en funksjon $f(t)$ ser slik ut:

$$F(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

hvor $F(\omega)$ er den Fouriertransformerte til $f(t)$, og har frekvens ω som argument (husk at vi skal mappe fra et "tidrom" til et "frekvensrom"). Her er $f(t)$ en kontinuerlig funksjon men i vårt tilfelle jobber vi med diskrete verdier. Fra sampling av lydfilen har vi en array x_n med verdier som tilsvarer utslaget til lydsignalet ved tidspunkt t (hvor t også er en array med like mange verdier som x_n). Vi skal derfor gjøre en *diskret* Fourier transformasjon. Likningen for dette er svært lik den kontinuerlige:

$$X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-i \frac{2\pi}{N} kn}$$

Her er x_n arrayet med signalet vi skal transformere (i tidsdomenet), N er lengden til x_n og X_k er de diskret Fouriertransformerte verdiene til x_n (i frekvensdomenet). Her er altså n indeksene til verdiene i *tidsdomenet*, og k er indeksene til verdiene i *frekvensdomenet*. I python skal vi bruke noe som heter *Fast Fourier Transform* (FFT), som er en algoritme som effektiviserer utregningen av den diskret Fouriertransformerte X_k til signalet x_n. Hvordan denne algoritmen er bygd opp er utenfor rammen til denne oppgaven.

Vi lager en funksjon som utnytter numpys FFT-funksjon:

[6]:
```
@jit
def fourier(t, x_n, N):
    ## Gjør en Fourier-analyse på signal x_n med tidpunkter t. N er lengden vi
 ↪ønsker til arrayet X_k fra fft.
    ## Returnerer frekvenser freq og utslag X_k fra fft.
    X_k = 1/N * np.fft.fft(x_n, n=N) # Gjør en fft på x_n. Deler på N for å
 ↪korrigere størrelsen på utslagene.
    dt = t[1] - t[0] #lengden på hvert tidssteg
    freq = np.fft.fftfreq(N, dt) # Finner frekvensområdet til spekteret fra
 ↪analysen.
    return freq, X_k #returnerer frekvensdomenet og fourier transformasjonen
```

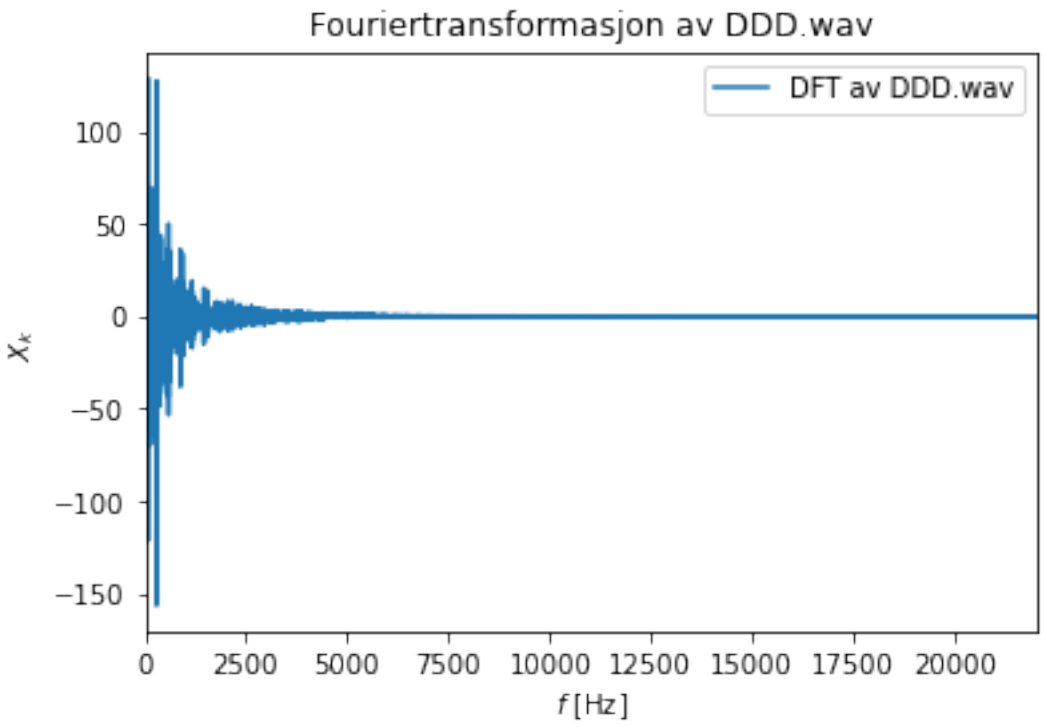
Vi kan nå teste funksjonen vår ved å Fouriertransformere signalet vårt fra DDD.wav:

[7]:
```
N_ddd = len(x_n_ddd)
freq_ddd, X_k_ddd = fourier(t_ddd, x_n_ddd, N_ddd) # Gjør en Fourieranalyse på
 ↪DDD. Bruker N_lu for å få lik lengde på X_k_ddd og X_k_lu
```

Vi plotter resultatet vårt i frekvensrommet:

[8]:
```
## Plotter frekvensspekteret til DDD
plt.xlim(0, fs_ddd/2) # Største mulige frekvens må være halvparten av
 ↪samplingsfrekvensen (Nyquist)
plt.title(f"Fouriertransformasjon av {lydfil_ddd}")
plt.xlabel("$f$ [Hz]")
plt.ylabel("$X_k$")
plt.plot(freq_ddd, X_k_ddd, label=f"DFT av {lydfil_ddd}")
plt.legend()
```

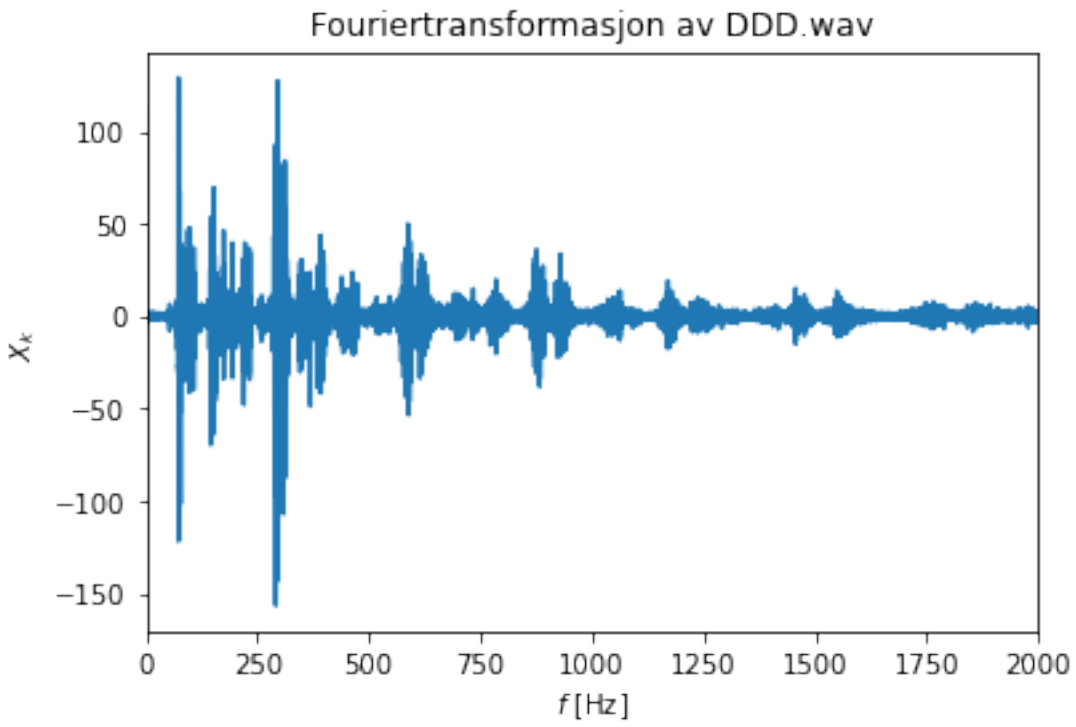
[8]:
```
<matplotlib.legend.Legend at 0x7f888ee56ca0>
```



Her kan du se frekvensene langs x-aksen, og utslaget ved hver frekvens langs y-aksen. Vi nevnte tidligere at samplingsfrekvensen som blir brukt er på $f_s = 44\,100$ Hz, som betyr at den største frekvensen vi her plukker opp (Nyquist frekvensen) er på 22 050 Hz. Vi kom tidligere fram til at dette skyldes at de høyeste frekvensene mennesker kan høre er på omtrent 20 000 Hz, men vår lydfil er i et mye mer behagelig område, som er grunnen til at utslaget dør helt ut på rundt 5 000 Hz, og er sterkest på enda lavere frekvenser. For å få et klarere syn på plottet zoomer vi derfor inn på det området som er interessant.

[9]:
```
## Zoomer inn på plottet av frekvensspekteret til DDD til relevante verdier
plt.xlim(0, 2000)
plt.title(f"Fouriertransformasjon av {lydfil_ddd}")
plt.xlabel("$f$ [Hz]")
plt.ylabel("$X_k$")
plt.plot(freq_ddd, X_k_ddd, label=f"DFT av {lydfil_ddd}")
```

[9]:
```
[<matplotlib.lines.Line2D at 0x7f888ee5c940>]
```



Her ser vi at vi har store utslag på litt over 250 Hz, og rundt 100 Hz. Vi har også flere andre utslag, hvor disse er mindre. Hvilke deler av lydfilen hver topp tilhører er vanskelig å si, ettersom en Fouriertransform ikke forteller *hvor* i signalet frekvensene oppstår, bare at de *erder*. Vi kan likevel tenke oss at de to største toppene muligens svarer til de tre første "dun"-ene i "dun dun dun duuun", siden disse forekommer litt mer enn den korte (de er kortere, men forekommer tre ganger).

2.2 Avskrekking ved hjelp av Fourieranalyse

Nå har vi klart, ved hjelp av Fouriertransformasjon, å gjøre en frekvensanalyse av DDD.wav. Vi ønsker nå å bruke dette til å avskrekke Beethoven's 5. symfoni, ved å fjerne "dun dun dun duuun" fra symfonien. For å gjøre dette skal vi se på et lite utdrag av den fullstendige symfonien. Dette utdraget er på omtrent 30 sekunder, og vi har valgt å kalle filen "liteUtdrag.wav". Vi bruker funksjonene vi har lagd tidligere til å sample lydfilen, og gjøre en Fourieranalyse på den.

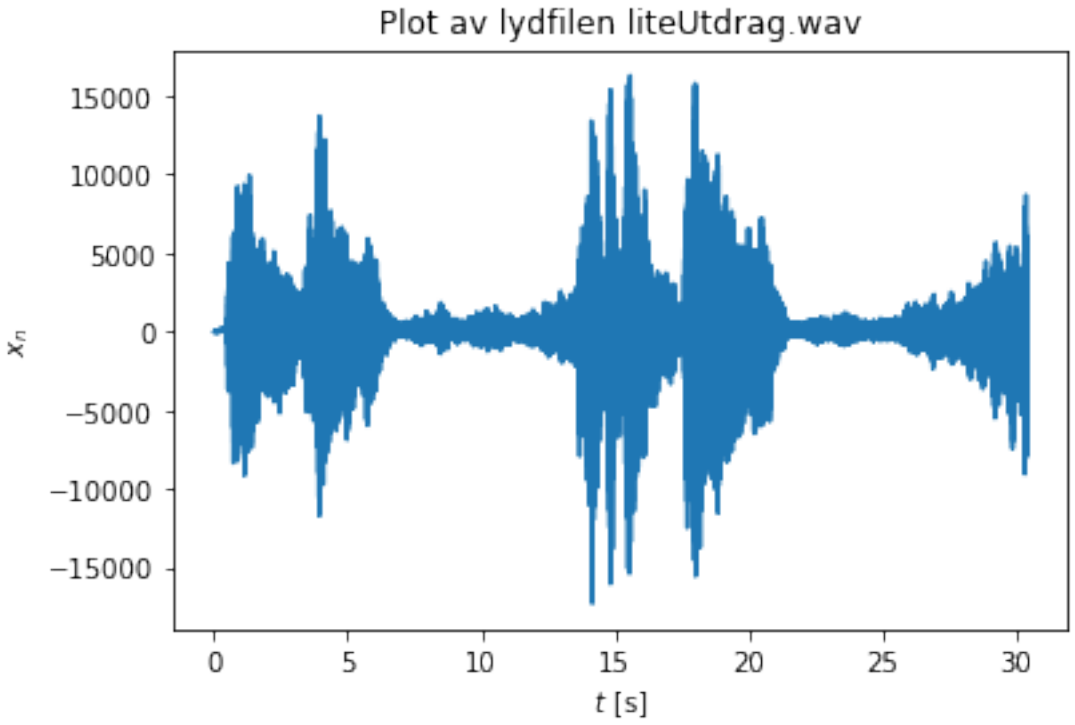
[10]:
```
lydfil_lu = "liteUtdrag.wav" # Lydfilen med et 30s utdrag av symfonien
t_lu, x_n_lu, fs_lu = readWav(lydfil_lu) # Henter data fra lydfilen
N_lu = len(x_n_lu) # Finner lengden til signalet (antall datapunkter)
freq_lu, X_k_lu = fourier(t_lu, x_n_lu, N_lu) # Gjør en Fourieranalyse på␣
↪utdraget
```

Vi kan dermed plotte signalet vårt i tidsdomenet:

[11]:
```
## Plotter lydsignalet fra utdraget på 30s
plt.title(f"Plot av lydfilen {lydfil_lu}")
```

```
plt.xlabel("$t$ [s]")
plt.ylabel("$x_n$")
plt.plot(t_lu, x_n_lu)
```

[11]: [<matplotlib.lines.Line2D at 0x7f888efafc70>]

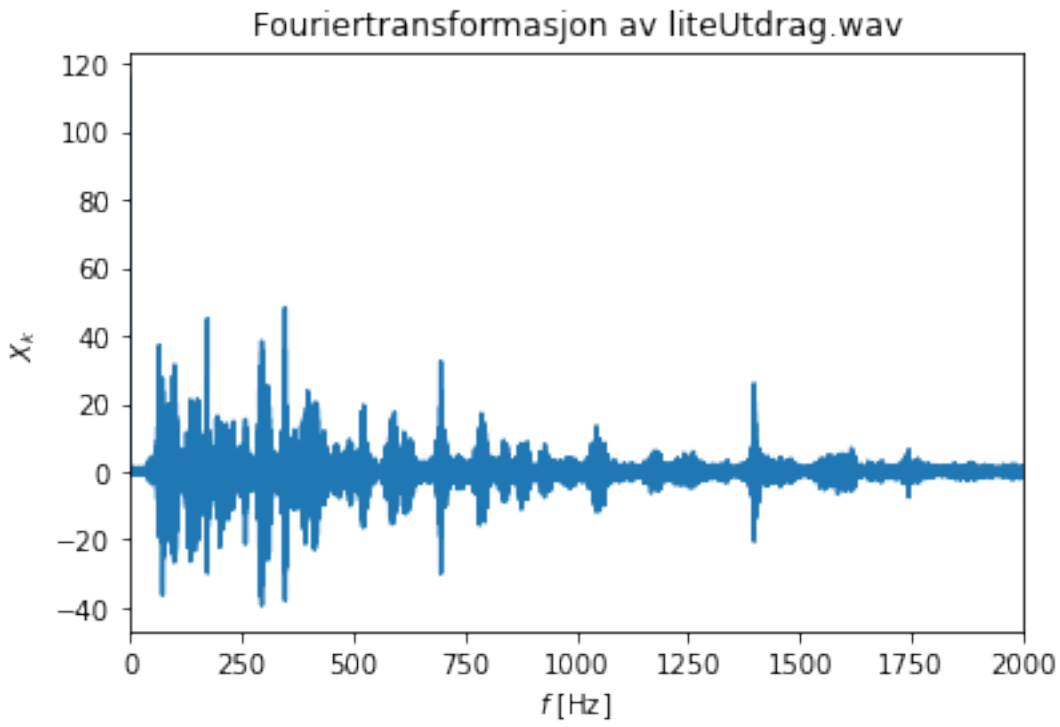


Vi ser at signalet starter med to store "blobber". Disse ser ut til å være identiske med de to sekvensene i signalet til DDD.wav. Dette er fordi de *er* identiske. Dette utdraget er de første 30 sekundene i symfonien, mens DDD.wav inneholdt de første 6 sekundene, altså er de 6 første sekundene til liteUtdrag.wav identisk med DDD.wav. Når vi skal forsøke å fjerne "dun dun dun" fra utdraget vårt håper vi at denne delen av signalet skal forsvinne helt.

La oss også plotte Fouriertransformasjonen til liteUtdrag.wav i frekvensdomenet:

[12]:
```
## Plotter frekvensspekteret til utdraget
plt.xlim(0, 2000)
plt.title(f"Fouriertransformasjon av {lydfil_lu}")
plt.xlabel("$f$ [Hz]")
plt.ylabel("$X_k$")
plt.plot(freq_lu, X_k_lu, label=f"DFT av {lydfil_lu}")
```

[12]: [<matplotlib.lines.Line2D at 0x7f888d2191c0>]



Igjen ser vi at vi har mange topper på lave frekvenser, rundt 100 Hz og oppover. Denne gangen har vi flere topper enn vi hadde for DDD.wav. Dette skyldes at liteUtdrag.wav er lengre, og naturligvis inneholder flere frekvenser enn DDD.wav.

Vi skal nå forsøke å fjerne frekvensene som oppstår i DDD.wav fra liteUtdrag.wav. For å gjøre dette skal vi rett fram fjerne hele arrayet med frekvensene til DDD.wav fra arrayet med frekvensene til liteUtdrag.wav. Eneste problemet her er at de ikke har samme lengde. Vi gjør en ny frekvensanalyse av DDD.wav, men sørger nå for at det resulterende arrayet blir like langt som arrayet til liteUtdrag.wav.

[13]:
```
freq_ddd, X_k_ddd = fourier(t_ddd, x_n_ddd, N_lu) # Gjør en Fourieranalyse på␣
↪DDD. Bruker N_lu for å få lik lengde på X_k_ddd og X_k_lu
```

Vi fjerner dermed frekvensene til DDD.wav fra liteUtdrag.wav:

[14]:
```
X_k_ny = X_k_lu - X_k_ddd # Fjerner frekvensene til DDD fra utdraget
```

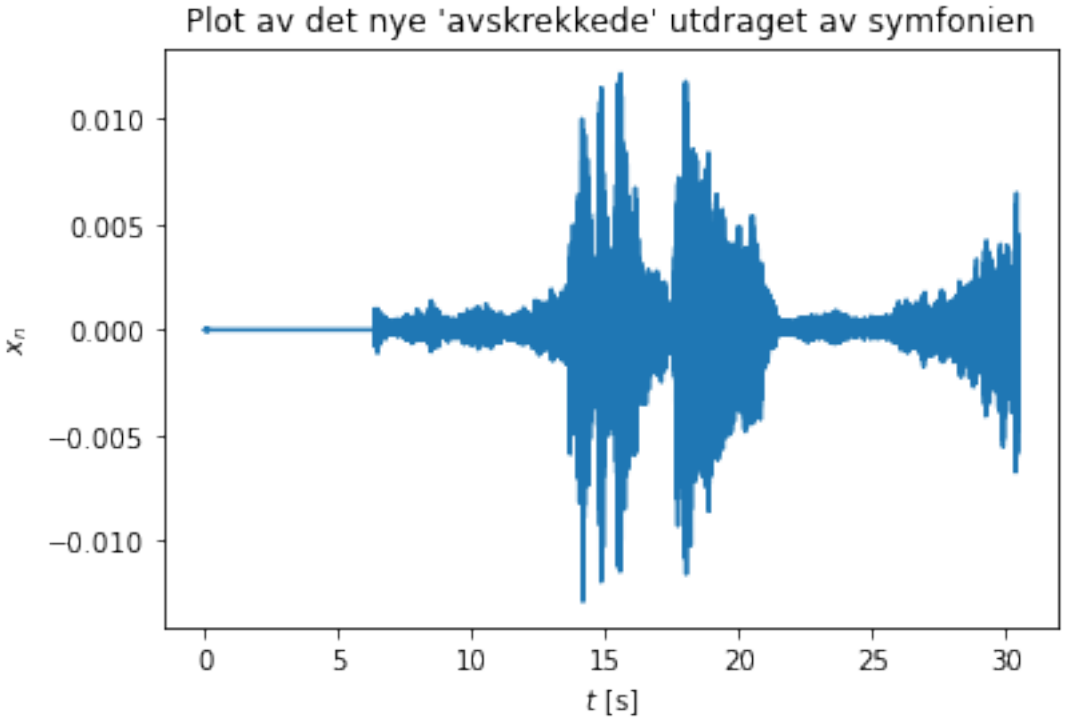
Vi kan nå konvertere vårt nye frekvensspekter $X_{k,ny}$ tilbake til et lydsignal ved å gjøre en invers Fouriertransformasjon. Dette gjør vi med numpy *Inverse Fast Fourier Transform* (IFFT) funksjon.

[15]:
```
x_n_ny = np.fft.ifft(X_k_ny) # Invers Fouriertransformasjon, for å få tilbake␣
↪lydsignalet uten DDD
```

Vi kan så plotte det nye lydsignalet:

[16]:
```
## Plotter det nye lydsignalet
plt.title(f"Plot av det nye 'avskrekkede' utdraget av symfonien")
plt.xlabel("$t$ [s]")
plt.ylabel("$x_n$")
plt.plot(t_lu, x_n_ny)
```

[16]: [<matplotlib.lines.Line2D at 0x7f88931e65e0>]



Her ser vi at de første 6 sekundene har forsvunnet helt, noe som tyder på at metoden vår har fungert. La oss konvertere signalet til en ny lydfil, som vi kan lytte til:

[17]:
```
wavfile.write("avskrekket_fft.wav", fs_lu, np.real(x_n_ny)) # Skriver␣
↪lydsignalet ut til ny lydfil
IPython.display.Audio("avskrekket_fft.wav") # Spiller av lydfilen
```

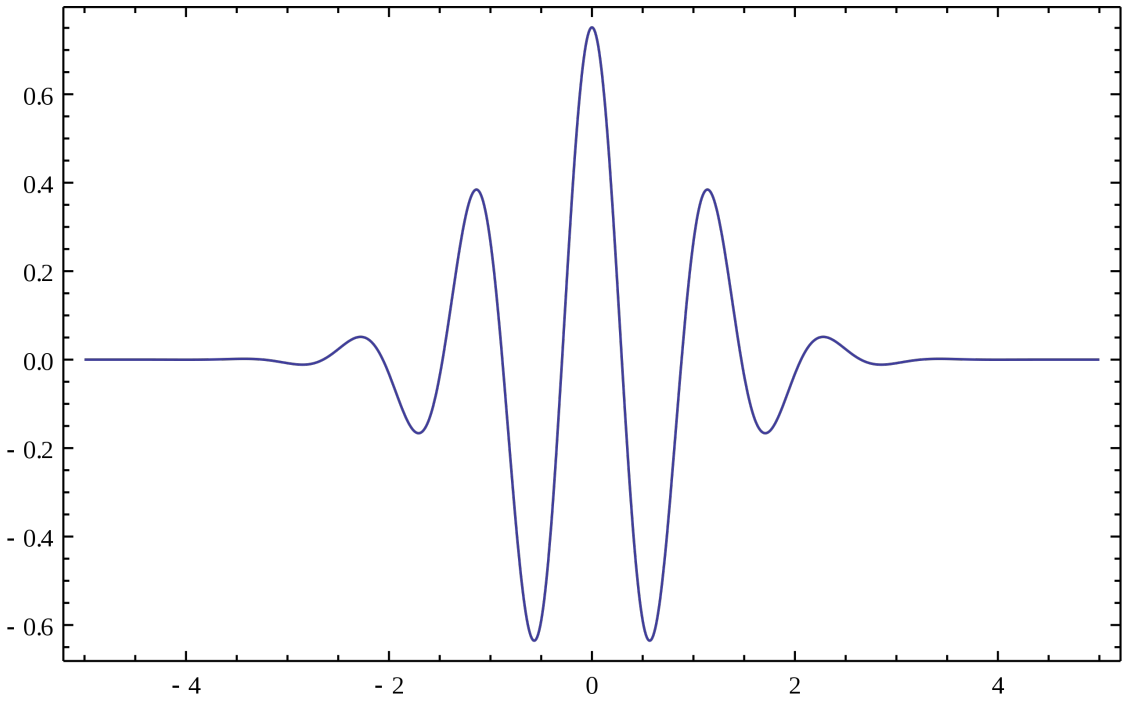
[17]: <IPython.lib.display.Audio object>

Ved å lytte til den nye lydfilen hører vi raskt at lyden har forsvunnet de første 6 sekundene, som er akkurat det vi ville. Metoden vår har likevel en veldig stor svakhet, nemlig at den fjerner frekvensene fra hele sangen. Vi kan se i Fouriertransformasjon til DDD.wav at vi har små (men ikke ubetydelige) bidrag til veldig mange andre frekvenser enn de vi er ute etter. Disse små bidragene blir også fjernet fra liteUtdrag.wav, i tillegg til at de frekvensene som bidrar mest i DDD.wav fjernes fra alle steder i liteUtdrag.wav, ikke bare "dun dun dun" delen. Resultatet er en lydfil hvor "dun dun dun duuun" er fjernet fra begynnelsen, men hvor også hele resten av lydfilen er utrolig

svak. I et forsøk på å løse dette problemet skal vi gå over til en mer avansert metode, nemlig wavelet-transformasjon.

2.3 Wavelet-transformasjon

En wavelet-transformasjon er en metode vi bruker for å analysere frekvenser til et signal *over tid*. Denne metoden ligner veldig på Fouriertransformasjon, men har noen veldig viktige forskjeller. Ved å gjøre en Fouriertransformasjon fikk vi en slags "diskret funksjon" $X_k(\omega)$ som var avhengig av frekvenser. Når vi gjør en wavelet-transformasjon skal vi finne en "funksjon" $\gamma_K(\omega_a, t_k)$, som er avhengig av både frekvens *og* tid. For Fourier ganget vi signalet vårt x_n med den komplekse eksponentialfunksjonen $e^{-i\frac{2\pi}{N}kn}$. Når vi nå skal bruke wavelet-transformasjon skal vi istedet gange signalet vårt med en bølgepakke (wavelet) som er avgrenset i tid. Vi skal bruke en "Morlet wavelet", som er en bølgepakke med Gaussisk avgrensning, som vist her:



Dette bildet er hentet fra [5]. Funksjonen for en Morlet wavelet ser slik ut:

$$\Psi_{\omega_a,K,t_k}(t_n) = C\left(e^{-i\omega_a(t_n-t_k)} - e^{-K^2}\right)e^{\frac{-\omega_a^2(t_n-t_k)^2}{(2K)^2}}$$

Her er Morlet waveleten Ψ en funksjon av tid t_n, ω_a frekvensen til bølgen i waveleten, K er en skarphetsparameter som beskriver bredden til waveleten (stor K gir høy presisjon i frekvens, men lav presisjon i tid, og liten K gir høy presisjon i tid, men lav presisjon i frekvens), t_k beskriver sentrum av waveleten (i bildet over er $t_k = 0$). C er en normaliseringskonstant, som vi skal sette til

$$C = \frac{0.798\omega_a}{f_s K}$$

Vi kommer i denne oppgaven kun til å se på Morlet wavelets, så når jeg skriver "wavelet" fra nå av er det implisitt at det er snakk om en Morlet wavelet. I funksjonen vår ovenfor for waveleten kan vi kjenne igjen et par deler. I parentesen har vi et ledd $e^{-i\omega_a(t_n-t_k)}$. Dette leddet ligner veldig på den komplekse potensen vi ganger inn i Fouriertransformasjonen, noe som ikke er tilfeldig, ettersom wavelet-transformasjon er samme prinsipp, bare at vi nå avgrenser denne funksjonen i tid. Faktoren $e^{\frac{-\omega_a^2(t_n-t_k)^2}{(2K)^2}}$ lengst til høyre ser vi ligner en Gaussisk funksjon. Det er denne som gjør at bølgen vår blir avgrenset Gaussisk. Vi implementerer følgende funksjon for waveleten Ψ:

[18]:
```
@jit
def psi(t,wa,tk,K,fs): # Likning 14.8 i Vistnes
    ## Funksjon som generer Waveleten som skal multipliseres med signalet
    # Disse tre er bare deler av en større funksjon som legges til tilslutt:
    w = np.exp(-1j*wa*(t-tk))
    k = np.exp(-K**2)
    gaus = np.exp(-(wa**2 * (t-tk)**2)/(2*K)**2)
    C=0.798*wa/(fs*K) # Ligning 14.7 i Vistnes
    return C*(w-k)*gaus #returnerer waveleten
```

Vi kan nå bruke denne funksjonen for Ψ til å finne wavelet-transformasjonen $\gamma_K(\omega_a, t_k)$. Vi finner den wavelet-transformerte av et signal x_n med følgende formel:

$$\gamma_K(\omega_a, t_k) = \sum_{n=1}^{N} x_n \Psi^*_{\omega_a, K, t_k}(t_n)$$

Denne funksjonen finner hvilke frekvenser som er til stedet ved forskjellige tidspunkt. Dette gjør den ved å flytte waveleten til mange forskjellige tidspunkter t_k, løpe gjennom en array med aktuelle frekvenser ω_a for hver t_k, og dermed gå over alle tidspunkter t_n (for hver kombinasjon av ω_a og t_k) og sammenligne den konjugerte av waveleten Ψ med signalet x_n for å finne tilstedeværelsen av frekvensene ved de tidspunktene. Resultatet fra funksjonen vår for γ_K i python blir altså en to-dimensjonal matrise, med frekvenser ω_a langs den ene aksen, og tider t_k langs den andre. Implementasjonen vår i python av dette ser slik ut:

[19]:
```
@jit
def gamma(xn,t,w,K,fs): # Likning 14.9 i Vistnes
    ## Funksjon som kjører en versjon av alle frekvenser til alle tidspunkter␣
 ↪og summerer produktet av signalet og waveleten og lager en matrise til␣
 ↪verdiene
    N = int(len(w)) #Henter lengden på frekvensarrayet
    T = int(len(t)) #Henter lengden på tidsarrayet
    gamma = np.zeros((N,T),dtype=np.complex_) #genererer en tom matrise
    for i in range(N): #løper gjennom alle frekvenser
        for ii in range(T): #løper gjennom alle tidspunkter
            psi1 = psi(t,w[i],t[ii],K,fs) #genererer den gjeldene waveleten
            gamma[i,ii] = np.sum(xn*np.conjugate(psi1)) #summerer opp produktet␣
 ↪av signalet og waveleten
    return gamma #returnerer matrisen
```

Vi kan nå definere en funksjon som finner γ_K, og gir et contour-plot av resultatet:

[20]:
```
@jit
def wavelet(data, t, w, K, fs):
    ## Kjører gamma funksjonen og genererer et contour plot med absolutverdien␣
 ↪av matrisen siden den blir kompleks
    grid = gamma(data,t,w,K,fs) #genererer matrisen
    plt.contourf(t,w,np.abs(grid)) # plotter griddet med frekvensene og␣
 ↪tidspunktene
    plt.colorbar() #viser hvilke verdier fargene har
    plt.xlabel("Tid $t$ [s]") #x-akse tittel
    plt.ylabel("Frekvens $\\omega_a$ [Hz]") #y-aksetittel
    plt.show() #viser plottet
    return grid # Returnerer gamma-griddet
```

Vi kan allerede nå se at wavelet-transformasjoner er veldig beregningstungt. Å gjøre en transformasjon på signalene våre, i sin helhet, vil ta unødvendig lang tid. Det er derfor hensiktsmessig å dele opp arrayene vi bruker i mindre arrayer, for å lette på beregningstiden. Til dette har vi laget følgende funksjon:

[21]:
```
@jit
def sliced_wavelet(t, x_n, freq, jump, max_freq_index):
    ## Lager sliced versjoner av t, x_n og frekvensene freq for at wavelet()␣
 ↪skal kjøre raskere
    ## jump forteller funksjonen hvor store hopp i arrayene vi skal gjøre for å␣
 ↪korte de ned
    ## max_freq_index forteller hvor stor del av frekvensarrayet vi skal se på,␣
 ↪siden vi vet at store deler av arrayet er uinteressant for oss.
    t_sliced = t[::jump]
    x_n_sliced = x_n[::jump]
    freq_sliced = freq[0:max_freq_index:jump]
    return t_sliced, x_n_sliced, freq_sliced # returnerer sliced arrayer
```

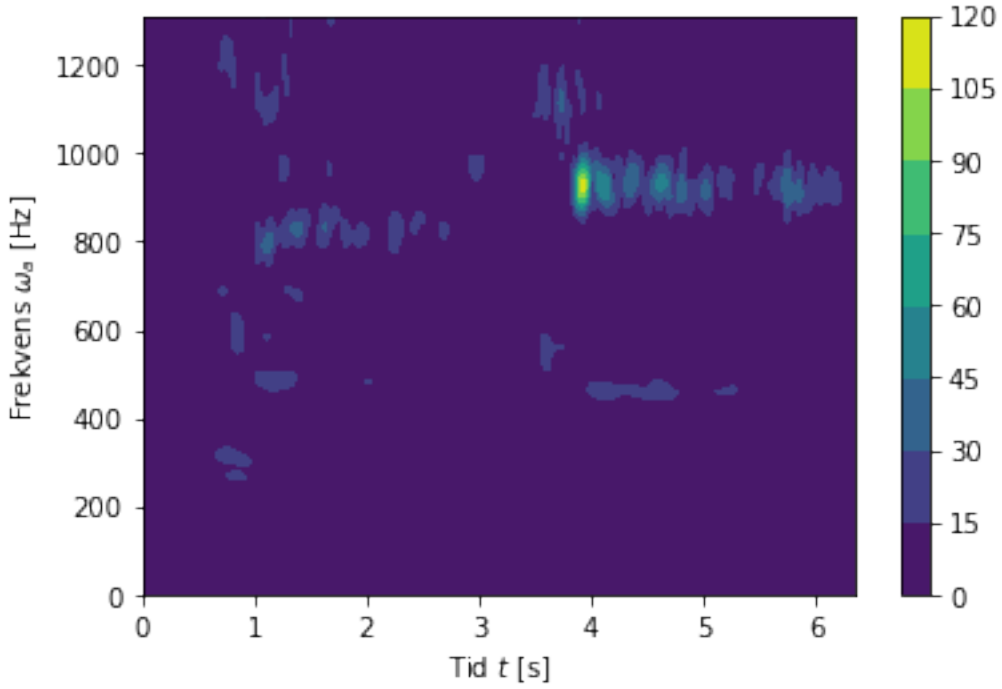
2.4 Avskrekking ved hjelp av wavelet-analyse

Vi skal nå prøve å bruke wavelet-transformasjon for å fjerne "dun dun dun" fra liteUtdrag.wav. Vi begynner med å gjøre en wavelet-analyse av signalet fra DDD.wav. For å gjøre dette trenger vi kortere arrayer for t, x_n og f:

[22]:
```
K = 20 #setter K verdien
slice_jump = 100
max_freq_index = 40000 # Her har vi prøvd oss fram, og funnet at ved å begrense␣
 ↪indeksen til 40 000 får vi frekvenser opp til omtrent 1300Hz
t_ddd_sliced, x_n_ddd_sliced, freq_ddd_sliced = sliced_wavelet(t_ddd, x_n_ddd,␣
 ↪freq_ddd, slice_jump, max_freq_index)
```

Vi bruker disse mindre arrayene til å gjøre en wavelet-analyse av signalet fra DDD.wav:

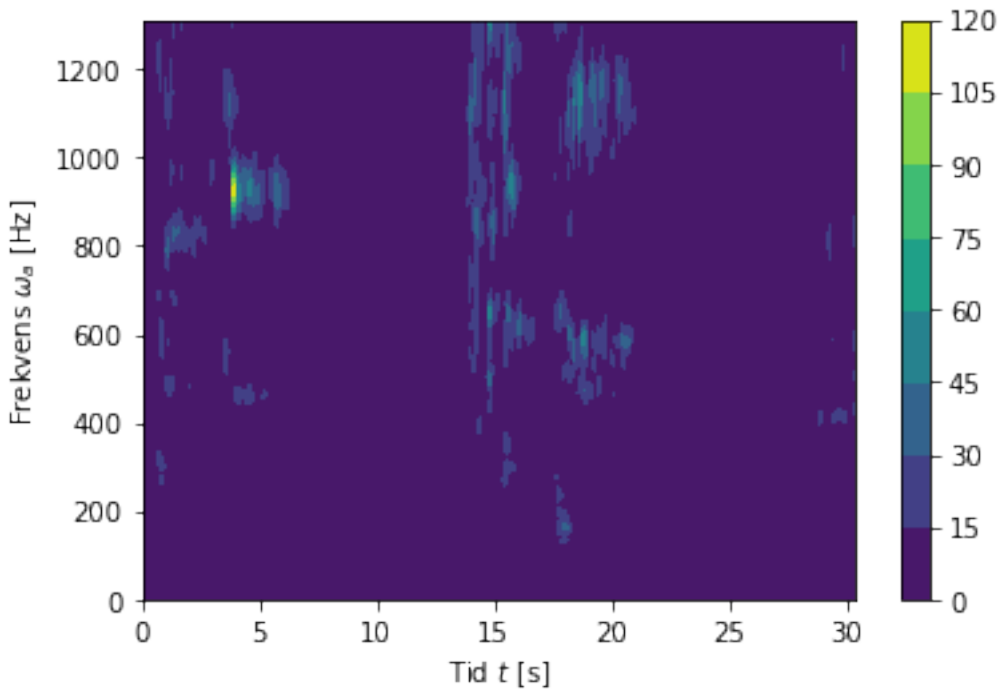
[23]:
```
grid_ddd = wavelet(x_n_ddd_sliced, t_ddd_sliced, freq_ddd_sliced, K, fs_ddd)
```



Dette plottet er et contourplot, med tid langs x-aksen, og frekvenser langs y-aksen. Styrken til contourplottet på et koordinat (t_i, ω_i) sier noe om forekomsten av frekvensen ω_i ved tidspunktet t_i. Dette er akkurat det vi ville ha. Nå har vi data som ikke bare forteller oss *hvilke* frekvenser som forekommer i symfonien, men også *når* frekvensene forekommer. La oss gjøre en tilsvarende waveletanalyse av liteUtdrag.wav.

[24]:
```
t_lu_sliced, x_n_lu_sliced, freq_lu_sliced = sliced_wavelet(t_lu, x_n_lu,␣
↪freq_lu, slice_jump, max_freq_index)
```

[25]:
```
grid_lu = wavelet(x_n_lu_sliced, t_lu_sliced, freq_lu_sliced, K, fs_lu)
```



Nå som vi har gjort en waveletanalyse av DDD.wav og liteUtdrag.wav kan vi forsøke å fjerne "dun dun dun" fra sistnevnte. For å gjøre dette skal vi utnytte at resultatene fra waveletanalysen (grid_ddd og grid_lu i koden) er $N_\omega \times N_t$ matriser, hvor N_ω er antall rader, og N_t antall kolonner i griddet. Matrisen til griddene er veldig store:

[26]:
```
print(f"Dimensjon til grid_ddd : {np.shape(grid_ddd)}")
print(f"Dimensjon til grid_lu : {np.shape(grid_lu)}")
```

```
Dimensjon til grid_ddd : (400, 2803)
Dimensjon til grid_lu : (400, 13416)
```

så i forklaringen av metoden vår skal vi bruke noen enklere matriser. Vi kaller disse matrisene A_d (grid_ddd) og A_l (grid_lu). Når vi velger dimensjonen til disse enklere matrisene må vi huske på to ting: A_d og A_l må ha like mange rader, mens A_l må ha flere kolonner enn A_d (som vi ser fra dimensjonene til de virkelige matrisene). La oss se for oss at A_d er en 2×3 matrise, mens A_l er en 2×10 matrise. Vi definerer:

$$A_d \equiv \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

og

$$A_l \equiv \begin{bmatrix} 5 & 2 & 7 & 1 & 2 & 3 & 2 & 1 & 9 & 8 \\ 6 & 1 & 4 & 3 & 2 & 1 & 8 & 2 & 1 & 8 \end{bmatrix}$$

Målet vårt er å finne ut om A_d forekommer i A_l. Hvis vi teller indekser fra 0 ser vi at A_d forekommer i A_l i kolonne 3 til 5.

$$A_l \equiv \begin{bmatrix} 5 & 2 & 7 & 1 & 2 & 3 & 2 & 1 & 9 & 8 \\ 6 & 1 & 4 & 3 & 2 & 1 & 8 & 2 & 1 & 8 \end{bmatrix}$$

Hva betyr dette? Vi vet at radene forteller noe om frekvensene, og kolonnene forteller noe om tidene frekvensene oppstår. At vi finner A_d et stedde inni A_l betyr altså at hele *sekvensen* fra DDD.wav forekommer i liteUtdrag.wav. Altså finner vi nå ikke bare frekvensene til "dun dun dun" i utdraget, men også tiden de frekvensene oppstår i den rekkefølgen vi vil forvente. Vi kan nå altså skille ut at forekomsten av frekvensene faktisk er "dun dun dun", og ikke en eller annen kombinasjon av frekvenser som *tilfeldigvis* også inneholder noen av frekvensene til "dun dun dun". Til sammenligning med resultatet fra Fourieranalysen, hvor hele sangen ble dempet på grunn av tilfeldige forekomster av DDD.wav sine frekvenser, kan vi nå fjerne *kun* "dun dun dun", og ikke andre deler av sangen. For å finne tidspunktet hvor sekvensen oppstår trenger vi bare å se på indeksen i_{start} til den første kolonnen i sekvensen (i eksempelet vårt er denne 3), og finne tidspunktet til den tilhørende indeksen. Hele sekvensen finner vi da ved å se på kolonnene fra i_{start} og frem til $i_{start} + N_{t,d}$ hvor $N_{t,d}$ er antall kolonner i A_d. I python skal vi bruke numpy sin "allclose" funksjon for å sjekke om alle verdiene til A_d er omtrent lik verdiene i det området vi ser på i A_l innenfor en viss toleranse.

[27]:
```
def analyze_wavelet(grid_big, grid_small, tol):
    # Finner indeksene i tidsarrayet hvor grid_small forekommer i grid_big
    Nw_big, Nt_big = np.shape(grid_big)
    Nw_small, Nt_small = np.shape(grid_small)
    N = Nt_big + 2 - Nt_small # Antall startverdier vi ser på for t
    indices = np.array([]) # Array med indekser. Her bør det komme en indeks␣
↪for hver forekomst av "dun dun dun" i sangen
    for i in range(Nt_big-Nt_small+1):
        # Løper gjennom alle mulige startindekser for delmatrisene i grid_big
        grid_i = grid_big[:, i:i+Nt_small] # Henter ut delmatrisen
        if np.allclose(grid_small, grid_i, atol=tol):
            # Sjekker om delmatrisen er lik grid_i, og appender indeksen hvis␣
↪det er tilfellet
            indices = np.append(indices, i)
    return indices # Returnerer indeksene
```

Vi kan nå bruke dette til å finne hvor "dun dun dun" forekommer i liteUtdrag.wav:

[28]:
```
indices = analyze_wavelet(grid_lu, grid_ddd, 5) # Finner forekomstene av "dun␣
↪dun dun" i grid_lu
indices = indices * 100 # Ganger med 100 for å få tilbake de virkelige␣
↪indeksene fra de slicede
print(f"Startindekser fra wavelet-analyse: {indices}")
```

```
Startindekser fra wavelet-analyse: [0.]
```

Når vi har funnet indeksene kan vi bruke disse til å fjerne de delene av symfonien som sier "dun dun dun" ved å ganske enkelt nullstille signalet vårt i det området hvor vi har funnet at det forekommer.

[29]:
```
Nw_small, Nt_small = np.shape(grid_ddd)
Nt_small = Nt_small * 100 # Ganger med 100 for å få lengden til den egentlige␣
↪tidsverdien, ikke den slicede
x_n_new = np.copy(x_n_lu) # Lager en kopi av x_n_lu
for index in indices:
    # Løper gjennom indeksene fra wavelet-analysene
    index = int(index)
    x_n_new[index:index+Nt_small] = np.zeros(Nt_small) # Fjerner lyden hvor␣
↪"dun dun dun" oppstår
```

Vi kan nå lytte til det ferdige signalet, og nyte Beethoven's avskrekkede symfoni (ihvertfall utdraget av det).

[30]:
```
wavfile.write("avskrekket_wavelet.wav", fs_lu, np.real(x_n_new))
IPython.display.Audio("avskrekket_wavelet.wav") # Spiller av lydfilen
```

[30]:
```
<IPython.lib.display.Audio object>
```

2.5 Resultat

2.5.1 Hvor mange wavelets trengte vi for å avskrekke symfonien?

Fra funksjonen vår gamma() ser vi at antall wavelets vi genererer er $N \cdot T$, hvor N er lengden på frekvensarrayet vi bruker, og T er lengden på tidsarrayet. Vi måtte gjøre to wavelet-analyser: en for DDD.wav, og en for liteUtdrag.wav. Lengden til frekvens- og tidsarrayene til disse kan vi enkelt finne:

[31]:
```
print(f"N_ddd = {len(freq_ddd_sliced)}")
print(f"N_lu = {len(freq_lu_sliced)}")
print(f"T_ddd = {len(t_ddd_sliced)}")
print(f"T_lu = {len(t_lu_sliced)}")
```

```
N_ddd = 400
N_lu = 400
T_ddd = 2803
T_lu = 13416
```

Vi bruker disse til å finne antall wavelets vi måtte bruke til hver av analysene. $N_{wave,DDD}$ er antall wavelets vi trengte for å analysere DDD.wav, og $N_{wave,lu}$ er antall wavelets vi trengte for å analysere liteUtdrag.wav.

$$N_{wave,DDD} = N_{DDD} \cdot T_{DDD} = 400 \cdot 2803 = 1\,121\,200$$

$$N_{wave,lu} = N_{lu} \cdot T_{lu} = 400 \cdot 13416 = 5\,366\,400$$

Det lille utdraget vårt er på 30 sekunder, mens den fulle symfonien varer i 7min = 420s. Dette er $\frac{420}{30} = 14$ ganger så langt som utdraget vårt, og vi antar derfor at antall wavelets vi ville trengt

for å analysere hele sangen er 14 ganger så mange som vi trengte for å analysere utdraget. Antall wavelets for å analysere hele sangen blir altså:

$$N_{wave,sym} = 14N_{wave,lu} = 14 \cdot 5\,366\,400 = 75\,129\,600$$

Legger vi til waveletene vi brukte for å analysere DDD.wav (siden vi fortsatt ville trengt denne for å analysere hele symfonien) får vi:

$$N_{wave} = N_{wave,DDD} + N_{wave,sym} = 1\,121\,200 + 75\,129\,600 = 76\,250\,800$$

For å avskrekke hele symfonien ville vi altså trengt 76 250 800 wavelets.

2.6 Konklusjon

Vi har kommet fram til at vi trenger 76 250 800 wavelets for å avskrekke Beethoven's 5. symfoni, men metoden vår har noen svakheter. For det første går vi ut fra at alle tilfellene av "DUN DUN DUN" i symfonien er helt identiske, altså at de har like frekvenser og samme tempo. Dette er dessverre ikke tilfellet med symfonien. Noen av forekomstene av "DUN DUN DUN" har høyere eller lavere frekvenser enn den aller første (den vi har brukt til sammenligning i analysen). Noen av "DUN DUN DUN"-ene sakker også ned tempoet sammenlignet med den første. Disse forskjellene gjør at vi ikke ville plukket opp disse "DUN DUN DUN"-ene, selv om vi hadde gjort en waveletanalyse av hele symfonien. En annen ting å merke seg er at analysen er utrolig beregningstung, og tar svært lang tid. Analysen vår av liteUtdrag.wav, som er på 30 sek, tok litt over en time. Skulle vi gjort en analyse av hele symfonien på 7 min ville det tatt enda lengre tid. For effektivitetens del ville det nok da være både raskere og lettere å bruke et lydredigeringsprogram, og manuelt fjerne alle forekomstene av "DUN DUN DUN" derfra.

Alt i alt kan vi altså si at metoden vår er treg og lite nøyaktig, men fungerer under antagelsen at alle forekomstene av "DUN DUN DUN" er identiske. Antall wavelets vi brukte fungerte fint, men hadde vi hatt lengre tid kunne vi funnet ut om ikke et lavere antall også ville fungert. Vi kan iallfall anse 76 250 800 som en øvre grense for antall nødvendige wavelets.

2.7 Referanser:

[0] Physics of Oscillations and Waves, Arnt Inge Vistnes (Publisert 2018)

[1] https://no.wikipedia.org/wiki/Symfoni_nr._5_(Beethoven)

[2] https://zone.ni.com/reference/de-XX/help/371361R-0113/lvanlsconcepts/aliasing/

[3] https://www.widex.com/en-us/blog/global/human-hearing-range-what-can-you-hear/

[4] https://en.wikipedia.org/wiki/Fourier_series#Exponential_form

[5] https://en.wikipedia.org/wiki/Morlet_wavelet