KU LEUVEN kulak

Groep Wetenschap & Technologie

X0B53A – Probleemoplossen en ontwerpen, deel 1
X0A22C – Informaticawerktuigen

Fourieranalyse en fourier-transformatie

Eindverslag Teamopdracht

Team Google

Ibrahim El Kaddouri, Mathis Bossuyt, Florijon Sharku, Iben Troch, Emiel Vanspranghels, Robbe Decapmaker, Hendrik Quintelier en Marijn Braem

Academiejaar 2020 – 2021

Inhoudsopgave

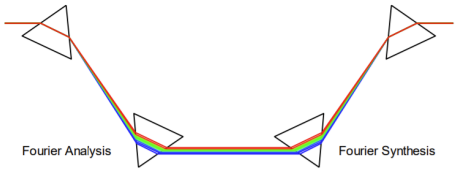
Inleiding

Om trillingen te kunnen benaderen maken we gebruik van de Fouriertransformatie. Die trillingen zijn ook te schrijven als een combinatie van goniometrische functies. Aan de hand van enkele meetpunten van amplitudes kunnen we een benadering van de trilling weergeven. Hoe meer meetpunten we echter krijgen, hoe beter we een trilling kunnen benaderen. Deze trillingen kunnen automatisch berekend worden door een programma. In dit verslag is het de bedoeling om ons zelfgemaakte programma te vergelijken met een geoptimaliseerd programma ontwikkeld door specialisten. Eerst beginnen we met een grondige uitleg over de fourieranalyse, de fourierreeksen met onder andere ook de benadering van functies. Dan leggen we de fouriertransformatie verder uit en zullen we verder in gaan op een snelle methode om deze transformatie uit te voeren. Daarna bespreken we onze implementatie aan de hand van de theorie om tenslotte tot de conclusie van ons eindverslag te komen.

Hoofdstuk 1

Fourier-analyse

Het idee van de Fourieranalyse is om functies van reële variabelen uit te drukken als een lineaire combinatie van functies die afkomstig zijn uit een collectie standaardfuncties. De fourierreeks wordt gebruikt om een periodieke functie te benaderen door een som van sinusoïden. De fouriertransformatie wordt gebruikt om een algemene, niet-periodieke functie te benaderen. Wij gaan eerder fourierreeksen gebuiken, vandaar dat de benadering niet altijd accuraat zal zijn voor niet-periodieke functies. [1] Laten we het duidelijker maken aan de hand van een illustratie. Stel we nemen twee prisma's.Het eerste paar prisma's kan het licht opsplitsen in zijn componentfrequenties (kleuren). Dit wordt Fourieranalyse genoemd. Een tweede paar kan de frequenties opnieuw combineren. Dit wordt Fouriersynthese genoemd.[2]



Figuur 1.1: Fourieranalyse uitgelegd aan de hand van lichtgolven [2]

1.1 Lineaire combinaties en orthogonaliteit van functies

Om een continue functie te kunnen benaderen, moeten we zorgen dat we een goede collectie standaardfuncties hebben. Een eerste functie die we altijd moeten gebruiken als we fouriertransformatie of fourierreeksen willen gebruiken, is de gewichtsfunctie ω. Om het ons wat gemakkelijker te maken, nemen we aan dat de gewichtsfuntie gelijk is aan 1. In onze implementatie maken we ook regelmatig gebruik van de begrippen 'orthogonaal' en 'de norm' die we hieronder

uitleggen.
Maar wat ben je nu met orthogonale functies? Wanneer een basisfunctie (een functie uit een collectie standaardfuncties) orthogonaal is, is de representatie veel kleiner en uniek. Stel dat je een functie $f(x)$ hebt en je wilt deze benaderen met behulp van 3 basisfuncties $g_1(x), g_2(x), g_3(x)$. dan geldt er:

$$\begin{aligned} f(x) =& c_1 \cdot g_1(x) + c_2 \cdot g_2(x) + c_3 \cdot g_3(x) \\ &+ c_{1,2} \cdot g_1(x) \cdot g_2(x) + c_{2,3} \cdot g_2(x) \cdot g_3(x) + c_{1,3} \cdot g_1(x) \cdot g_3(x) \\ &+ c_{1,2,3} \cdot g_1(x) \cdot g_2(x) \cdot g_3(x) \end{aligned} \tag{1.1}$$

Zoals je ziet zijn er nogal wat coëfficiënten (die exponentieel groeien naarmate het aantal functies toeneemt) ook niet uniek. Als de functies orthogonaal zouden zijn, dan zouden alle samengestelde functies nul zijn. De verzameling van de coëfficiënten zou ook uniek zijn voor elke functie waardoor er zou gelden dat:

$$f(x) = c_1 \cdot g_1(x) + c_2 \cdot g_2(x) + c_3 \cdot g_3(x) \tag{1.2}$$

Waarom zouden ze nul zijn? We kiezen namelijk de basisfuncties zodat wanneer je die dan vermenigvuldigt nul uitkomt. Maar dat moet gelden voor alle waarden en voor alle basisfuncties, vandaar dat we een integraal zullen nemen.

- Twee functies f en g zijn orthogonaal als en slechts als

$$< f, g >=< g, f >= \left(\int_a^b f(x) g(x) \omega(x) dx \right) = 0$$

  [3]

- De norm $||f||$ van een functie f is

$$\sqrt{< f, f >} = \sqrt{\int_a^b f(x)^2 \omega(x) dx}$$

  [3]

- Een functie kunnen we schrijven als een lineaire combinatie $f = a_1 f_1 + a_2 f_2 + \ldots + a_k f_k$ van orthonormale functies. Dit is de verzameling van de orthonormale functies op een interval [a,b]. [3]

- We noemen deze verzameling functies een orthonormaal (orthogonaal + norm = 1) als voor alle i en j geldt dat

$$< f_i, f_j >= \begin{cases} 0 \text{ als } i \neq j \\ 1 \text{ als } i = j \end{cases}$$

  [3]

- De coëfficiënten van die lineaire combinaties voor $i \in \{1, \ldots, k\}$ kunnen we schrijven als

$$a_i = \int_a^b f(x) f_i(x) \omega(x) dx =< f, f_i > [3] \tag{1.3}$$

Hoofdstuk 2

Fourierreeks

Een willekeurig periodieke functie kunnen we schrijven als een lineaire combinatie van basisfuncties. In dit hoofdstuk gaan we de sinus- en cosinusfuncties gebruiken als basisfuncties.
Enkele voorwaarden voor dit concept zijn: [4]

- De functie f moet in een bepaald interval begrensd zijn.
- De functie f mag een eindig aantal discontinuïteiten hebben in een bepaald begrensd interval

2.1 Fourierreeks, sinus-cosinus formule

Laten we eerst de verzameling A basisfuncties opsommen met onze gewichtsfunctie erbij.

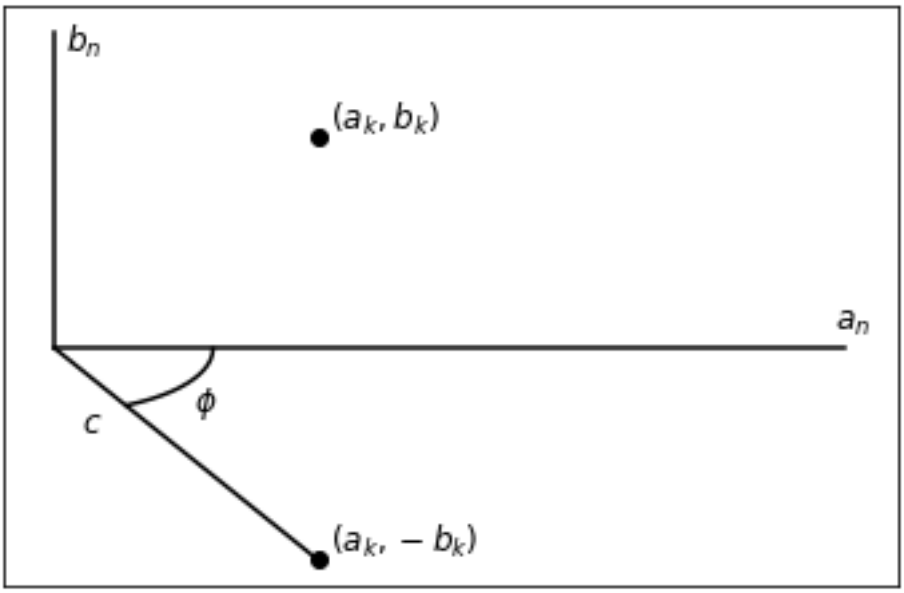
$$A = \{1, sin(x), cos(x)\} \tag{2.1}$$

We breiden de verzameling A uit. Daardoor zal de verzameling oneindig groot worden:

$$A = \{1, sin(x), cos(x), sin(2x), cos(2x), sin(3x), cos(3x), \dots\} \tag{2.2}$$

We kunnen nu een benadering geven voor een bepaalde functie $f(x)$ door een lineaire combinatie te maken van de basisfuncties.
Beschouw nu de volgende formule in het interval $[-\pi, \pi]$, voor alle $k \in \mathbb{N}_0$

$$f(x) = a_0 + \sum_{k=1}^{\infty} a_k cos(kx) + \sum_{k=1}^{\infty} b_k sin(kx) \tag{2.3}$$



Figuur 2.1: illustratief verband met de koppels (C, ϕ) en (a_k, b_k)

2.2 Fourierreeks, amplitude-fase formule

We kunnen de sinus-cosinusformule 2.3 omvormen naar een amplitude-faseformule. We zullen beginnen met de amplitude-faseformule en zullen daaruit de sinus-cosinusformule afleiden. De formule gaat als volgt:

$$f(x) = c_0 + \sum_{k=1}^{\infty} c_k cos(kx - \phi_k) \tag{2.4}$$

We kunnen deze omvorming voorstellen als we de a_k-waarden voorstellen op de x-as en de b_k-waarden op de y-as. Dan zal de c_k de lengte zijn van de rechte met de oorsprong tot aan het punt (a_k, b_k). De fase ϕ_k kunnen we dan beschouwen als de hoek tussen de x-as en de rechte c_k. (zie figuur2.1)
We kunnen beginnen met de cosinus om te vormen aan de hand van de hoeksom- en hoekverschilidentiteiten. [5]

$$\begin{aligned} \sin(\alpha + \beta) &= \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta) \\ \cos(\alpha + \beta) &= \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \end{aligned}$$

We weten ook dat de cosinusfunctie een even functie is terwijl de sinusfunctie een oneven functie is, dat betekent dat $\sin(-\theta) = -\sin(\theta)$ en $\cos(-\theta) = \cos(\theta)$
Als we dat ook toepassen en herschrijven bekomen we:

$$\begin{aligned} c_k cos(kx - \phi_k) &= c_k[cos(kx)cos(-\phi_k) + sin(kx)sin(-\phi_k)] \\ &= c_k[cos(kx)cos(\phi_k) - sin(kx)sin(\phi_k)] \\ &= c_k cos(\phi_k)cos(kx) - c_k sin(\phi_k)sin(kx) \\ &= a_k cos(kx) + b_k sin(kx) \text{ met } a_k = c_k cos(\phi_k) \text{ en } b_k = -c_k sin(\phi_k) \end{aligned}$$

Stel we berekenen $a_k^2 + b_k^2$, dan geldt er:

$$\begin{aligned} a_k^2 + b_k^2 &= c_k^2 cos^2(\phi_k) + c_k^2 sin^2(\phi_k) \\ &= c_k^2[cos^2(\phi_k) + sin^2(\phi_k)] \\ &= c_k^2 \end{aligned}$$

Stel we berekenen de breuk $\frac{b_k}{a_k}$, dan geldt er:

$$\begin{aligned} \frac{b_k}{a_k} &= \frac{c_k sin(\phi_k)}{c_k cos(\phi_k)} \\ &= tan(\phi_k) \\ &\implies \phi_k = \arctan(\frac{b_k}{a_k}) \end{aligned}$$

Deze formule is gemakkelijker om mee te werken aangezien er één term minder is dan de vorige formule. De sinus-en cosinusparen kunnen worden uitgedrukt als een enkele sinusoïde met een faseverschuiving, analoog aan de conversie tussen orthogonale (cartesiaanse) en poolcoördinaten.

$$f(x) = c_0 + \sum_{k=1}^{\infty} c_k cos(kx - \phi_k) \tag{2.5}$$

Hierbij is $c_k = \sqrt{a_n^2 + b_n^2}$ en $\phi_k = \arctan(\frac{b_n}{a_n^2})$ met dus c_k, ϕ_k als poolcoördinaten.

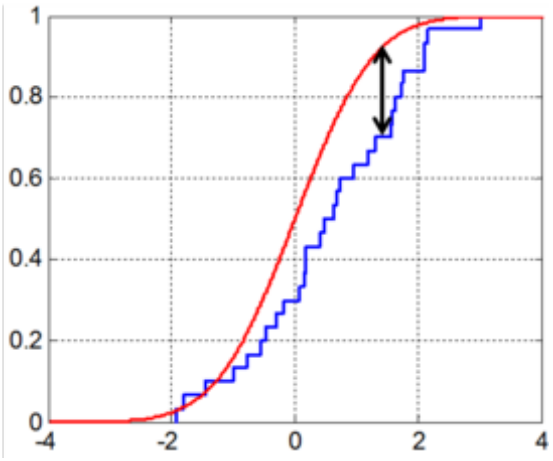
2.3 Afstand tussen functies

Wanneer we eenmaal de benaderde functie en de originele functie hebben, kunnen we zeggen dat de benadering goed is, wanneer de afstand tussen de benaderende en originele functie klein is.
Dit moet niet alleen kloppen voor één waarde van de twee functies, maar voor alle waarden voor de twee functies. Uiteindelijk kunnen we zeggen, hoe kleiner de (kwadratische) oppervlakte tussen de twee functies f en g is, hoe kleiner de foutmarge tussen die twee functies. Als we dit nu allemaal in de Euclidische afstandsfunctie steken, dan krijgen we het volgende:

$$\begin{aligned} d(p,q) &= \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \\ d(f,g) &= ||f-g|| = \sqrt{\int_a^b (f(x) - g(x))^2 \omega(x) dx} \end{aligned} \tag{2.6}$$

We weten al dat we een bepaalde functie $f(x)$ kunnen schrijven als een lineaire combinaties van andere functies.

$$f(x) = a_1 \cdot f_1(x) + a_2 \cdot f_2(x) + a_3 \cdot f_3(x) + \dots \tag{2.7}$$



Figuur 2.2: afstand tussen twee functies
[6]

Laten we de functie die $f(x)$ wilt benaderen $g(x)$ noemen, dan geldt er dat $g(x) = \sum_{i=1}^{m} a_i f_i(x)$. Dus hoe beter de benadering, hoe kleiner de foutenmarge en dus hoe kleiner de afstand tussen de twee functies.

$$d(f,g) = ||f-g|| = \sqrt{\int_a^b (f(x) - \sum_{i=1}^{m} a_i f_i(x))^2 \omega(x) dx))}$$

2.4 Benaderen van een stel meetwaarden

We kunnen functies ook benaderen aan de hand van een stel (functie)waarden. Stel nu dat die functiewaarden op gelijke afstand liggen in een interval $[a,b]$. Dit interval wordt in n gelijke deelintervallen verdeeld. Zo bekomen we de equidistante punten (punten die op gelijke afstand van elkaar liggen):

$$x_0 = a, x_1 = a + \frac{b-a}{n}, x_2 = a + 2\frac{b-a}{n}, \ldots, x_n = b$$

In het hoofdstuk 1.3 hebben we gezien dat: (zie ook [1])

$$\begin{aligned} a_j &= \int_a^b f(x) f_j(x) \omega(x) dx =< f, f_i > \\ &\approx \sum_{k=1}^{n} f(x_k) f_j(k) \omega(x_k)(x_k - x_{k-1}) \\ &= \sum_{k=1}^{n} y_k f_j(x_k) \omega(x_k) \frac{b-a}{n} \end{aligned} \tag{2.8}$$

[1]

Daarmee kunnen we de functie f benaderen bij de meetwaarden $f(x_i) = y_i$. Het aantal meetwaarden is daarmee ook recht evenredig met de nauwkeurigheid van de functie f en dus ook de nauwkeurigheid van de coëfficiënten a_j.

Hoofdstuk 3

Fouriertransformatie en comprimeren van info

3.1 Compressie

We veronderstellen dat we een lijst van getallen $y_0, y_1, \ldots, y_n$ moeten opslaan. Om deze functie te comprimeren, zullen we aan de hand van een 'eenvoudig' voorbeeld tonen hoe het werkt. We nemen de functie $g_1(t) = 3 \cdot sin(\omega t)$ met $\omega = 6 \cdot 2\pi$ rad s^{-1}. (zie 3.1a) Als we de sinus-cosinusformule van de fourierreeks toepassen,

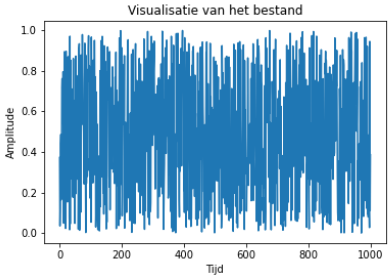
$$f(x) = a_0 + \sum_{k=1}^{\infty} a_k cos\ kx + \sum_{k=1}^{\infty} b_k sin\ kx$$

dan krijgen we dat $a_k = b_k = 0$ behalve voor b_6 die in dit geval 3 is. Deze functie kunnen we schrijven in de amplitude-fase formule 2.4 als $g_1(t) = 3 \cdot cos(\omega t - \frac{\pi}{2})$ met $\omega = 6 \cdot 2\pi$ rad s^{-1}. De Fouriergetransformeerde functie G_1 kunnen we dus schrijven als koppels c_k-waarden en ϕ-waarden(fase). (zie 2.4)

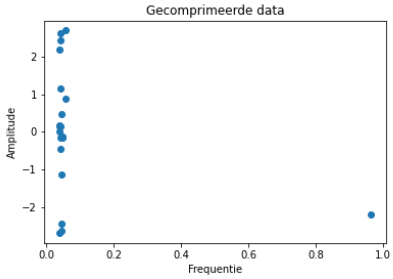
$$G_1(\omega) = \begin{cases} (3, \frac{\pi}{2}) \text{ als } \omega = 6 \cdot 2\pi rads^{-}1 \\ (0, \frac{\pi}{2}) \text{ als } \omega \neq 6 \cdot 2\pi rads^{-}1 \end{cases}$$

Normaal bevat het tijdssignaal g_1 oneindig veel punten (koppels van tijd en amplitude). Met de Fouriertransformatie comprimeren we die info naar de functie G_1 (zie 3.1b) die volgende informatie moet bevatten: frequentie met de daarbijhorende amplitude en faseverschuiving [1].

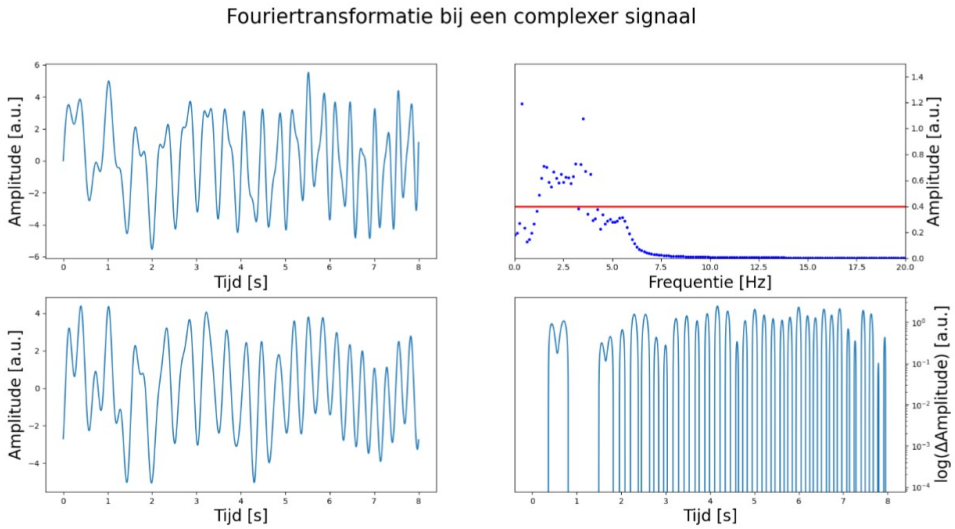
We kunnen dus een samengestelde functie aan de hand van de fouriertransformatie op een andere manier schrijven om zo de computer veel rekenwerk te besparen. We kunnen ook omgekeerd te werk gaan waarbij we dus een fouriergetransformeerde functie naar een gewone functie transformeren. Bij complexere functies kunnen we in de fouriergetransformeerde functie kleine uitwijkingen/aanpassingen verrichten en die terug omzetten naar de gewone functie. Dit is mooi geïllustreerd in 3.2. De figuur linksboven is een willekeurige functie. De afbeelding die rechtsboven staat is de Fouriergetransformeerde functie daarvan. Als men die functie manipuleert door in dit geval een amplitude te nemen die boven 0,4 ligt en die terug om te zetten naar de gewone functie, bekomt men



(a) origineel signaal



(b) gecomprimeerde datapunten



Figuur 3.2: FFT bij een willekeurige functie (figuur ontleend aan [1])

de figuur linksonder. De figuur rechtsonder is ter illustratie die het verschil in amplitude tussen de originele functie en de aangepaste functie weergeeft. [1]

Hoofdstuk 4

Implementatie

Inleiding

In dit hoofdstuk gaan we de geziene theorie omzetten naar de werkelijkheid. Het is namelijk zo dat de fourieranalyse niet evident is om manueel uit te rekenen. Daarom kunnen we computers inschakelen om deze bewerkingen uit te voeren. Deze snelheid is namelijk cruciaal om de toepassingen van de analyse op een tijdige manier uit te kunnen voeren. Wat in dit hoofdstuk volgt, is hoe wij als team een poging hebben gedaan om pen en papier om te zetten naar *Python*-code.

4.1 Bereken waarden

De functie *bereken_Waarden* in ons programma is de eenvoudigste functie die we gebruiken. Aan *bereken_Waarden* dien je drie getallen mee te geven: a, b en n. Deze getallen stellen respectievelijk de startwaarde, de eindwaarde en hoeveel waarden je wilt in het interval voor. Je krijgt een lijst terug met n+1 equidistante punten tussen a en b. Geef je bijvoorbeeld a=5, b=10 en n=5, dan krijg je als output: [5,6,7,8,9,10].

```
def bereken_Waarden(a,b,n):
lijstMetWaarden = []
for k in range(n + 1):
x = a + k*((b-a)/n)
lijstMetWaarden.append(x)
return lijstMetWaarden
```

4.2 Fourierreeks

De functie *fourierReeks* berekent a_k- en b_k-waarden die van belang zullen zijn binnen andere functies. De berekening van de a_k- en b_k-waarden gebeurt hier aan de hand van de sommatie 2.8.

```
def fourierReeks(y,n):
ak_coefficienten = []
bk_coefficienten = []
ak_en_bk =  []

x = sp.symbols('x',real=True)
lijstMetWaarden = bereken_Waarden(-float(np.pi), float(np.pi), len(testwaarden))

for k in range(n):
ak = 0
bk = 0
for l in range(len(testwaarden)):
ak += testwaarden[l]*np.cos((k+1)*lijstMetWaarden[l])*2/len(testwaarden)
bk += testwaarden[l]*np.sin((k+1)*lijstMetWaarden[l])*2/len(testwaarden)

ak_coefficienten.append(ak)
bk_coefficienten.append(bk)

ak_en_bk = [ak_coefficienten , bk_coefficienten]


return ak_en_bk
```

Er dient een lijst met amplitudewaarden (y) en een gewenst aantal index-waarden (n) meegegeven te worden. De output is een geneste lijst met zowel a_k- als b_k-waarden, zodat deze toegankelijk en te onderscheiden zijn voor het gebruik ervan binnen andere functies.

4.3 Innamewav

De functie *innamewav* zorgt ervoor dat *.wav*-files ingelezen en omgezet kunnen worden naar bruikbare data. De enige parameter is hier het *path* van de *.wav*-file en de output is ook hier een geneste lijst, met daarin een lijst met tijdswaarden en een lijst met amplitudewaarden.

4.4 Fouriercomplex

De functie *Fouriercomplex* werkt gelijkaardig aan de functie *Fourierreeks*, maar geeft geen lijst met a_k- en b_k-waarden terug, maar een lijst met amplitudes c_k en een lijst met faseverschuivingen ϕ_k.

4.4.1 c_k-waarden en faseverschuiving

We hebben de c_k waarden kunnen bereken aan de hand van de a_k en b_k waarden (zie 2.4). Met deze waarden kunnen we de amplitude gaan reconstrueren volgens de functie:

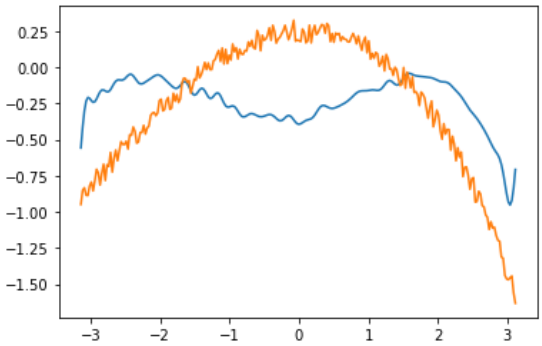
$$f(x) = c_0 + \sum_{k=1}^{+\infty} cos(kxw - \phi_k) \tag{4.1}$$

$$c_k = \sqrt{a_k^2 + b_k^2} \tag{4.2}$$

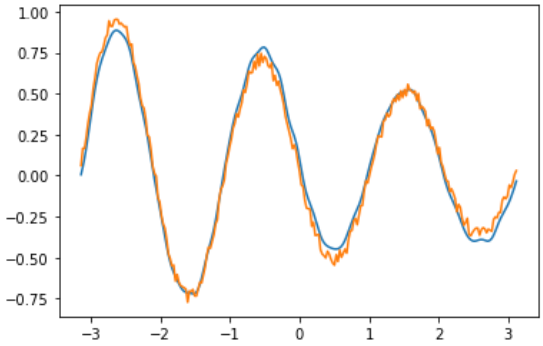
$$\phi_k = \arctan \frac{b_k}{a_k} \tag{4.3}$$

We zijn vertrokken vanuit de code die we geschreven hadden bij *Fourierreeks*. Dus we hebben onze a_k en b_k-waarden. Vervolgens vulden we de formules in voor de c_k 4.2 en ϕ_k 4.3 waarden. Deze plaatsten we dan in een lijst om ze vervolgens weer op te roepen en de amplitude te reconstrueren met formule 4.1. In de code berekenen we ook het gemiddelde verschil tussen de benaderde en de werkelijke waarden en we zetten die dan ook weer vervolgens in een lijst. We berekenen eigenlijk twee keer de formule 4.1, de eerste keer om de benaderende waarden te verkrijgen, en de volgende keer om de gemiddelde waarden daarvan af te trekken. Je kunt dit voorstellen als de evenwichtsstand te veranderen. De algemene sinusfunctie is $f(x) = Asin(bx + c) + d$ met d de evenwichtstoestand.

Dan was er soms toch geen goede benadering voor de meeste datasets, maar voor een aantal datasets was de benadering wel al vrij acceptabel. De reden hierachter is dat de datasets waarbij onze code faalt, niet periodiek zijn. Deze functie is met andere woorden enkel maar te gebruiken op periodieke data zoals geluidsgolven.



Figuur 4.1: Dataset 2 wordt slecht benaderd door onze complexe reconstructie.



Figuur 4.2: Onze reconstructie voor dataset 7 is al een betere benadering.

```
def complexReeks(Ylijst,n):

AkEnBk = fourierReeks(Ylijst,n)

N = len(Ylijst)
Xlijst = bereken_Waarden(-float(sp.pi) ,float(sp.pi) ,N)

Ck = []
for k in range(len(AkEnBk[0])):
waarde = m.sqrt(float(AkEnBk[0][k])**2 + float(AkEnBk[1][k])**2)
Ck.append(waarde)

fase = []
for k in range(len(Ck)):
waarde = np.arctan2(float(AkEnBk[1][k]),float(AkEnBk[0][k]))
fase.append(waarde)

f = []
for x in Xlijst:
sommatie = 0

for k in range (1,len(Ck)):
phi = (((k+1) * x) - fase[k])
sommatie += Ck[k] * np.cos(phi)

uitkomst = Ck[0] + sommatie
f.append(uitkomst)

verschil = 0
for i in range(len(f)):
verschil += f[i]-Ylijst[i]
gemverschil = verschil/len(Ylijst)

F = []
for x in Xlijst:
```

```
sommatie = 0

for k in range (1,len(Ck)):
phi = (((k+1) * x) - fase[k])
sommatie += Ck[k] * np.cos(phi)

uitkomst = Ck[0] + sommatie - float(gemverschil)
F.append(uitkomst)

return F
```

In de voorgaande code moet je opmerken dat we de boogtangens enkel in het eerste en vierde kwadrant situeren. Er is een zekere analogie tussen periodieke en niet-periodieke benadering. Zoals je ziet is dataset 7 periodiek en dataset 2 niet. Als je alle andere plots ziet dan kan je vaststellen dat de complexe benadering enkel goed is voor periodieke geluidsgolven.

4.5 Datacompressie

Er is soms een overvloed aan informatie terwijl we met minder data ongeveer hetzelfde resultaat bereiken. In deze functie passen we dit concept toe op onze geluidssignalen.

4.5.1 Comprimeren van het signaal

De functie *comprimeerSignaal* is een functie die een overvloed aan informatie comprimeert tot de meest belangrijke info. In deze functie geven we 3 variabelen op: Y, K en ϕ. Y is een lijst van datapunten, K is de factor waarmee de lijst met datapunten wordt beperkt en ϕ (*boolean*) bepaalt of we via faseverschuiving of aan de hand van de normale fourierreeks zullen werken. Bij het uitvoeren van deze functie krijgen we een grafiek met de oorspronkelijke data en daarna krijgen we een grafiek met de gecomprimeerde datapunten.

4.5.2 Reconstrueren van het signaal

De functie *reconstrueerSignaal* is een functie die vanuit de gecomprimeerde datapunten opnieuw de oorspronkelijke datapunten probeert te benaderen. In deze functie geven we vier variabelen op: *amplituden*, *plaats*, *hoeveelheid_Y* en ϕ. *Amplituden* is de lijst van de a_k- of c_k- waarden, *plaats* is de lijst van de b_k of fasewaarden, *hoeveelheid_Y* is het aantal punten dat je wilt terug krijgen en ϕ geeft de faseverschuiving weer. Bij het uitvoeren van deze functie wordt een nieuwe lijst aangemaakt met de gereconstrueerde amplituden.

4.6 Voorbeelden

Het is natuurlijk ook belangrijk om de geschreven routines met elkaar samen te doen werken. Hiervoor werd er geopteerd voor een systeem met meerdere bestanden. We kozen er voor om twee bestanden aan te maken. Het bestand *TeamG_procedures.py* is verantwoordelijk voor de functionaliteit van ons programma. Het bevat alle besproken onderwerpen, samen met enkele ondersteunende componenten. Daarnaast hebben we ook nog het bestand *TeamG_voorbeelden.py* aangemaakt. Het doel van dit bestand is om de beschreven processen te visualiseren. De laatste importeert de eerste, om gebruik te maken van z'n functionaliteit.

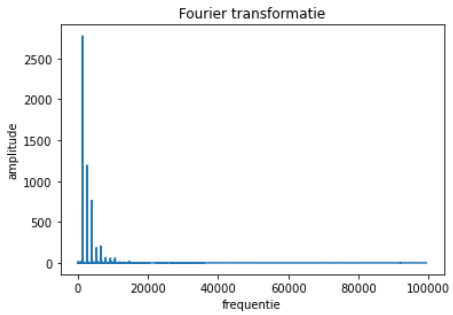
Hoofdstuk 5

Fast Fourier Transform

5.1 Fast Fourier Transform

De fouriertransformatie is ongetwijfeld één van de belangrijkste wiskundige concepten in onze moderne samenleving. We gebruiken zo'n transformatie om een frequentiespectrum te maken van data (Figuur: 5.1). Deze spectra geven ons een schat aan informatie over een geluidssignaal. Tevens maken ze het mogelijk om complexe bewerkingen te doen met het signaal. Een mooi voorbeeld is om ruis te verwijderen uit geluidsopnames. Dit kan bereikt worden door de juiste frequentie te elimineren uit de transformatie. Daarnaast bestaat er ook een manier om van de getransformeerde functie terug te keren naar een geluidsbestand. Het hele proces kan gebeuren met een DFT oftewel een discrete fouriertransformatie. Dit algoritme neemt geluidsbestanden in, en maakt een aantal wiskundige bewerkingen om een getransformeerde functie te kunnen bekomen. Deze poging is een goede eerste stap, maar er is een probleem. Als we ons programma draaien, merken we dat het relatief traag met resultaten naar de tafel komt. Dit probleem wordt pas echt duidelijk als we langere geluidsbestanden proberen te analyseren. Dit is natuurlijk onacceptabel als we werken met tijdsgevoelige scenario's. Het FFT-algoritme is een variatie van de DFT waarbij er een aantal wiskundige concepten uit de lineaire algebra worden toegepast om het proces te versnellen.

Figuur 5.1: Fourier transformatie



5.2 Werking van FFT

Het algoritme werkt op basis van een aantal concepten in de lineaire algebra. Het probeert om de DFT-berekeningen te vereenvoudigen. De manier waarop dat de DFT werkt kan geschreven worden als volgende matrixvermenigvuldiging:[7]

$$\begin{Bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{Bmatrix} = \begin{Bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{Bmatrix} \cdot \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{Bmatrix} \tag{5.1}$$

Hierbij is A_n de getransformeerde functiewaarde, ω word gegeven door volgende vergelijking: $e^{-2\pi i/n}$ waarbij n het aantal datapunten is. a_n zijn de datapunten.

De matrix die hier wordt voorgesteld is klein, en is bijgevolg niet moeilijk om uit te rekenen. De situatie verandert echter nogal snel als we naar de realiteit terugkeren. Geluidsbestanden bevatten een grote hoeveelheid data, en zorgen er dus voor dat matrices van deze soort enorm groot worden. Dit zal ons onvermijdelijk in de situatie brengen waar de benodigde tijd om deze matrix uit te rekenen gewoon te groot wordt. De FFT biedt hier een oplossing voor ons. Door een aantal wiskundige principes[7] toe te passen kunnen we het probleem vereenvoudigen.

$$\begin{Bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{Bmatrix} = \begin{Bmatrix} I_2 & -D_2 \\ I_2 & -D_2 \end{Bmatrix} \cdot \begin{Bmatrix} F_2 & 0_2 \\ 0_2 & F_2 \end{Bmatrix} \cdot \begin{Bmatrix} a_{\text{even}} \\ a_{\text{oneven}} \end{Bmatrix} \tag{5.2}$$

We zien nu dat de DFT gesplitst is in enkele matrices. Hierbij staan onder andere de identiteitsmatrix I en de nulmatrix 0. De matrix F is equivalent aan de centrale matrix uit vergelijking 5.1. Tot slot is er ook nog de matrix D, deze ziet er als volgt uit:

$$\begin{Bmatrix} \omega^0 & 0 \\ 0 & \omega^1 \end{Bmatrix} \tag{5.3}$$

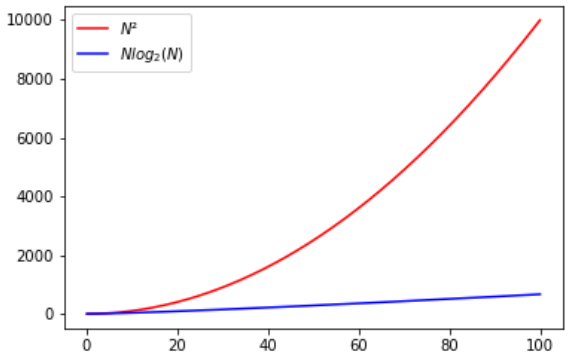
Het is dus duidelijk dat de berekeningen een stuk sneller zullen gaan, er zijn namelijk enorm veel termen die weg vallen omdat er zoveel nullen aanwezig zijn. bovendien valt het proces van opsplitsen te herhalen. Als we starten met een $8X8$ matrix, kunnen we die opdelen naar een $4X4$ matrix, die we vervolgens kunnen opdelen volgens proces 5.2.

$$\begin{Bmatrix} A_0 \\ A_1 \\ \vdots \\ A_6 \\ A_7 \end{Bmatrix} = \begin{Bmatrix} I_4 & -D_4 \\ I_4 & -D_4 \end{Bmatrix} \cdot \begin{Bmatrix} Q_4 & 0_4 \\ 0_4 & Q_4 \end{Bmatrix} \cdot \begin{Bmatrix} a_{\text{even}} \\ a_{\text{oneven}} \end{Bmatrix} \tag{5.4}$$

met

$$Q_4 = \begin{Bmatrix} F_2 & 0_2 \\ 0_2 & F_2 \end{Bmatrix} \tag{5.5}$$

Figuur 5.2: Complexiteit van de verschillende algoritmen



5.3 De voordelen van FFT

Het grote voordeel van de FFT is dat het een stuk sneller loopt dan de gewone DFT. Dit verschil werd gemeten[8] door de complexiteit te berekenen van de 2 verschillende algoritmen. De complexiteit van de DFT is $\mathcal{O}(n^2)$, dit heeft catastrofale gevolgen bij grote invoerlijsten. Vooral als we in rekening houden dat een geluidsbestand van 5 seconden al 220 000 datapunten bevat. Als we dit vergelijken (Figuur: 5.2) met de complexiteit van het FFT algoritme $\mathcal{O}(n \log n)$, dan zien we een enorme vooruitgang. Het is namelijk zo dat de FFT-complexiteit zich bij benadering lineair gedraagt.

5.4 Implementatie van FFT

We hebben er als team ook voor gekozen om een programma te schrijven in Python dat toont hoe de FFT werkt. Het is namelijk zo dat dit algoritme zo vaak gebruikt wordt, dat er in een grote hoeveelheid programmeertalen een bibliotheek bestaat met de FFT functionaliteit. In Python heet ze `numpy`. Wij gebruiken de ingebouwde functie, niet omdat het te moeilijk zou zijn om hem zelf te maken, maar omdat het de bedoeling is om deze te gebruiken. Deze bibliotheek is namelijk sterk geoptimaliseerd om zo efficiënt mogelijk te kunnen draaien, bovendien bespaart het veel tijd uit voor de programmeur.

Ons programma neemt een geluidsbestand in, en laat de FFT erop los. Daarna nemen we de resultaten en geven we die weer op een grafiek. Tot slot nemen we de omgekeerde FFT van ons resultaat en exporteren we die als een `.wav` bestand. We doen dit om aan te tonen dat we zonder problemen tussen de data en de transformatie kunnen gaan. Afhankelijk van het doel van ons programma, kunnen we tussen deze stappen een aantal aanpassingen doen aan de transformatie. Dit is tevens het principe waarop technologie zoals auto-tune werkt[9].

```
# in nemen van bestanden en ze visualiseren
path = 'dwarsfluit re.wav'
geluidsbestand = innamewav(path)
```

```
plt.plot(geluidsbestand[0], geluidsbestand[1])
plt.title("Origineel geluidsbestand")
plt.ylabel("Amplitude")
plt.xlabel("Tijd(ms)")
plt.show()


# berekenen van de FFT
start = time.time()

fourier = np.fft.rfft(geluidsbestand[1])

stop = time.time()
print("het duurde",str(stop - start),"seconden om de fourier transformatie te berekenen.")


# Tonen van de FFT
plt.plot(abs(fourier))
plt.title("Fourier transformatie")
plt.xlabel("frequentie")
plt.ylabel("amplitude")
plt.show()


#De omgekeerde fouriertransformatie toepassen

omgekeerde = irfft(fourier)

plt.plot(omgekeerde)
plt.title("Omgekeerde Fourier transformatie")
plt.xlabel('datapunten')
plt.ylabel("amplitude")
plt.show()

#Geluidsbestand maken van de inverse FFT
write('geluidsbestand.wav', 44000, omgekeerde)
print("Het nieuwe geluidsbestand werd opgeslagen!")
```

Hierbij is het ook nog nuttig om te vermelden dat ons programma zelf bijhoudt hoe snel de FFT werkt. Dit dient om nog eens extra aan te duiden hoe snel we een transformatie kunnen verkrijgen.

Conclusie

We kunnen concluderen dat het niet simpel is om een fourieranalyse te implementeren. Hiervoor zijn namelijk heel wat ingewikkelde formules voor nodig. Als team zijn we er wel in geslaagd om dit project tot een relatief goed einde te brengen.

We hebben geleerd dat fourierreeks een onderdeel is van de fourieranalyse. We hebben ook geleerd dat fourierreeks toegepast wordt om periodieke functies zo goed mogelijk te benaderen. De Fourierreeks is een manier om periodieke functies voor te stellen als een oneindige som van meer eenvoudige sinus- en cosinusgolven. Van benaderingstheorie tot signaalverwerking, alles met een herkenbaar patroon kan worden beschreven met wisselende sinus en cosinusgolven. De Fast Fourier Transform is een manier om de efficiëntie van de Discrete Fourier Transformatie te laten stijgen. De FFT is in staat om een resultaat te bereiken in $\mathcal{O}(n \log n)$-tijd, de basis hiervan ligt bij het feit dat we de DFT kunnen opsplitsen in kleinere operaties om het rekenwerk dus te verminderen.

Als team is het ons ons gelukt om namelijk zonder al te veel problemen de theoretische achtergrond van het onderwerp uit te werken. Daarna kwam natuurlijk onze implementatie, deze ging niet zonder slag of stoot. Het is namelijk zo dat we tot op de laatste uren van de teamopdracht met serieuze fouten te kampen hadden.
Wat wij beter hadden kunnen doen, is om versiecontrole te implementeren. Versieconflicten zouden dan ook niet meer bestaan. We hebben veel tijd verspild en dus dubbel werk moeten verrichten. We konden Github gebruiken om daar onze bestanden op te slaan. We kunnen dan de vorige versies herbekijken om mogelijke kleine foutjes die in de toekomst grote problemen kunnen aannemen vermijden.
De organisatie van het team is wel vlotjes gebeurt, iedereen wist wat te doen en er was een goede sfeer binnen de groep. Iedereen heeft op zijn eigen manier een steentje kunnen bijdragen om dit project mee tot een goed einde te kunnen brengen.

Bibliografie

[1] Orthogonale functies en fourierbenaderingen.

[2] Mike Brookes. E1.10 fourier series and transforms. 2014.

[3] David Morin. waves. https://scholar.harvard.edu/david-morin/waves.

[4] Eric W Weisstein. Dirichlet conditions. https://mathworld.wolfram.com/DirichletFourierSeriesConditions.html, laatste wijziging 19 Nov 2020.

[5] Wikipedia. lijst van goniometrische gelijkheden. https://cutt.ly/YhQxGQe, laatste wijziging 2 december 2020.

[6] Wikipedia. Kolmogorov–smirnov test. https://en.wikipedia.org/wiki/Kolmogorovlaatste wijziging 2 december 2020.

[7] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2019.

[8] Paul S. Heckbert. Fourier transforms and the fast fourier transform (fft) algorithm. 1998.

[9] Michael A Peimani. *Pitch correction for the human voice*. PhD thesis, PhD. Thesis, University of California, Santa Cruz, 2009.

Team Google
Wetenschap & Technologie
KU Leuven Kulak
Etienne Sabbelaan 53, 8500 Kortrijk