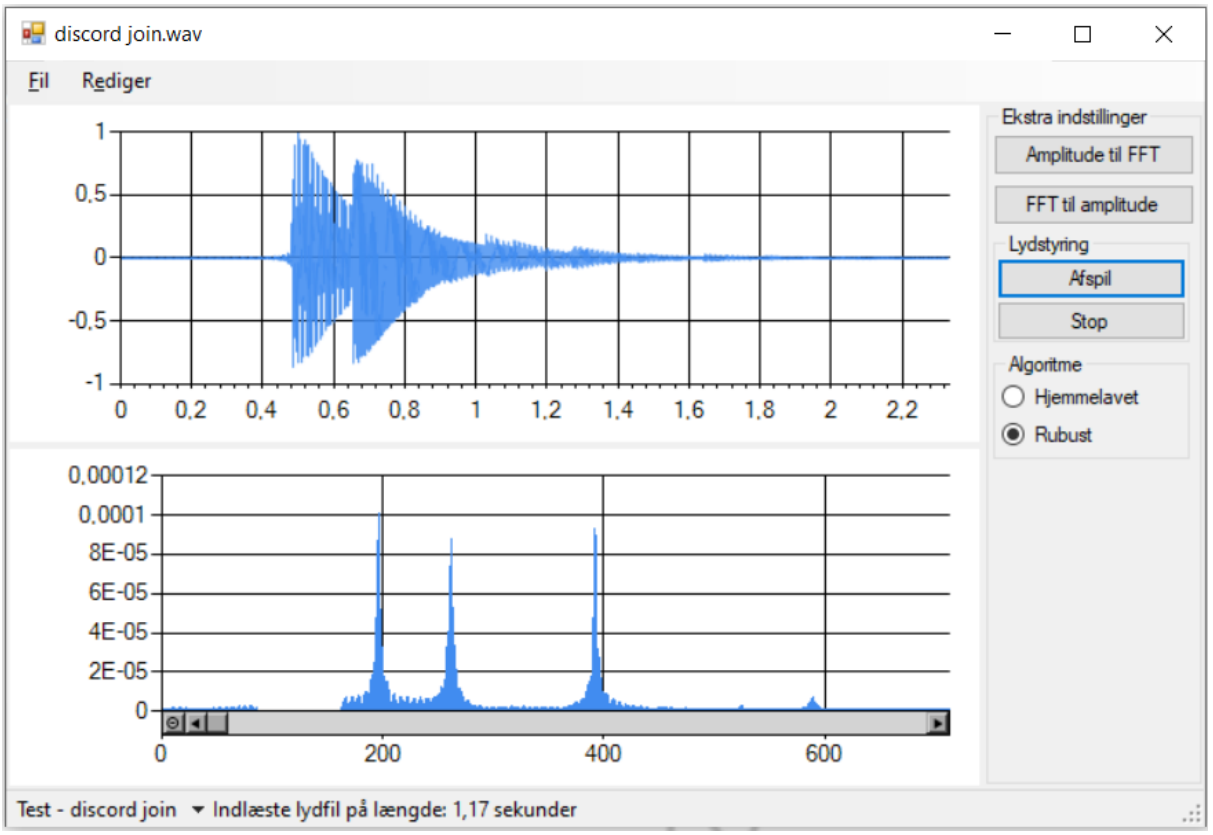


Fourier transformeringer

Navn: Kresten Sckerl
Klasse: 3.mi
Gymnasium: Learnmark Horsens HTX
Skoleår: 2021/2022
Dato: 21-12/2021

Fagvalg: Programmering og Matematik
Vejledere: Line T. og Annika C.
Antal tegn: 47999

Resumé

På baggrund af den given opgaveformulering er der udredt nogle regneregler for komplekse tal. Disse regneregler er væsentlige at kende til, da der beregnes et konkret eksempel på fourier transformation, hvor en sinuskurve med frekvensen π på diskret form, betragtes. Det er under denne beregning, hvor emnets kerne giver sig til kende. Beregningen viser intuitivt hvordan indeksene som før svarede til tid, nu svarer til frekvens. Der betragtes yderligere, hvordan det faktisk er muligt at genskabe den originale funktion (til en vis præcision), når den er i frekvensdomænet, ved at aflæse amplituden af kurven. Udover udredningen af den intuitive forklaring og beregning af fourier transformationen, dokumenteres der senere hen, hvordan man kan implementere den selv, og hvordan den tages i brug praktisk, for at arbejde med større datasæt. Følgende bliver valgene under implementeringen bakket op ved at tage nogle programmeringsfaglige metoder i brug til yderligere dokumentation af programmet, heriblandt flowcharts, pseudokode og klassediagrammer. Konkluderende reflekteres der over arbejdsprocessen, design valgene, og fyldestgørelse af besvarelsen for opgaven.

Indholdsfortegnelse

Indledning

I opgaven bliver der redegjort for komplekse tal og fourier transformation. Først betragtes teorien, som omfavner de to store emner, og sidenhen bliver udviklingen af et program dokumenteret. Selve projektet handler om at undersøge, hvordan man arbejder med fourier transformering i kode, det gøres ved at undersøge, hvordan den implementeres.

Opgaven tager udgangspunkt i følgende problemformulering:

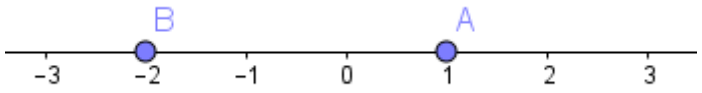
- Redegør for komplekse tal og fourier transformation, samt deres sammenhæng.
- Udarbejd et eksempel på en implementation af fast fourier transformation.
- Udarbejd et program, som kan bearbejde lydfiler i praksis.

For bedre at kunne udarbejde en besvarelse til problemformuleringen tages forskellige metoder indenfor fagene, matematik og programmering, i brug. Der bruges primært teoretiske metoder fra fagene, men emnet udforskes yderligere ved brug af forskellige empiriske metoder, såsom implementeringen af et program til spektralanalyse, bearbejdelse af filformat, og filtrering af lyd. Der vil også blive udredt matematiske formler til implementering i programmet og videre udarbejdelse af emnet.

Redegør for komplekse tal og fourier transformation, samt deres sammenhæng

For at forstå sammenhængen mellem komplekse tal og fourier transformation, skal man have en forståelse inden for de to emner hver især.

Et komplekst tal, er et tal, som består af to dele; én imaginær del og én reel del. De reelle tal, eller det reelle tallegeme, består af heltal, rationelle tal (de tal som kan udtrykkes med kvotienter af heltal) og irrationelle tal (tal som udtrykkes ved hjælp af rækker såsom pi). Et komplekst tal består i virkeligheden af 2 reelle tal, men det ene tal har den imaginære enhed som multiplikand. Den imaginære enhed, i, er defineret ved $i = \sqrt{-1}$. Det får den betydning at man går fra at et tal repræsenterer 1-dimension, til at repræsentere 2-dimensioner — ligesom med vektorer[1].[2]

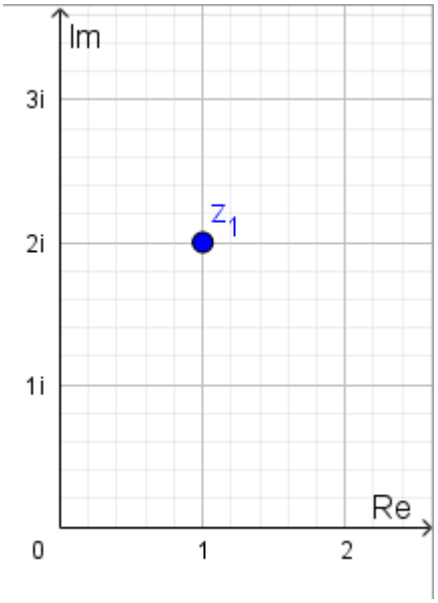


Figur 1: 1-dimensionalt koordinatsystem (Geogebra)

Et eksempel på et komplekst tal er $z_1 = (1, 2)$, hvilket kan skrives på denne form: $z_1 = 1 + 2i$, hvor i er den imaginære enhed. Tallet kan både repræsenteres som et sted, og som en beregning. Umiddelbart giver det ikke mening at betragte et tal som et sted, da man normalt udelukkende betragter koordinatsæt med 2 akser eller 3 akser, hhv. $(x; y)$ og $(x; y; z)$. Men det medfølger at det også er muligt at have et koordinatsæt (x), hvilket gør det muligt betragte et tal som et punkt/sted. På figur 1 er blot den første akse synlig. I dette koordinatsystem er punkterne $B = (-2)$ og $A = (1)$ indtegnet.

Når et komplekst tal betragtes som et sted, giver det mening at vælge den første akse til at repræsentere den reelle del af tallet, derefter dannes en akse mere, som svarer til den imaginære del.

[1] Vektorer er ikke tal.
[2] (McLean, 2020, Repetition 12)



Figur 2: Kompleks talplan med punktet $z_1 = 1 + 2i$

I figur 2 er punktet $z_1 = 1 + 2i$. For at indtegne punktet, går man 1 ud ad førsteaksen, hvilket svarer til $Re(z_1)$[3] , og man er gået 2 ud ad andenaksen, hvilket svarer til $Im(z_1)$ [4]. På baggrund af det kan z_1 defineres således: $z_1 = (Re(z_1); Im(z_1))$.

For komplekse tal kan man bruge de samme regnearter som dem for reelle tal. Addition og subtraktion for komplekse tal, gøres blot ved at addere/subtrahere hhv. den reelle del og den imaginære del.

$$z_1 \pm z_2 = Re(z_1) \pm Re(z_2) + i(Im(z_1 \pm Im(z_2)))$$

$$z_1 = 1 + 2i$$
$$z_2 = 2 + 1i$$

Ved addition:

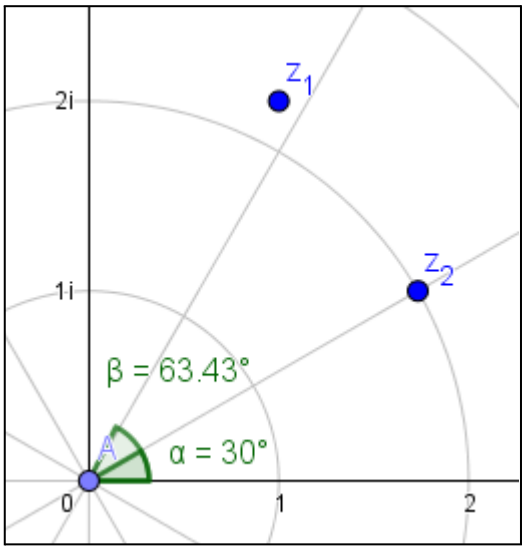
$$1 + 2 + i(2 + 1) = 3 + 3i$$

Ved subtraktion:

$$1 - 2 + i(2 - 1) = -1 + 1i$$

[3] En anden måde at skrive reel-del på.
[4] Den imaginære del.

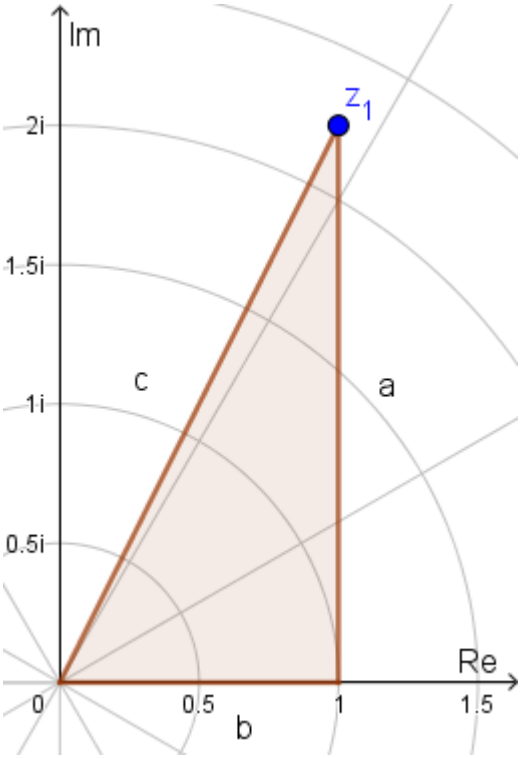
Den form det komplekse tal er udtrykt ved kaldes den rektangulære form. Lige nu ligner det blot en stedvektor, $\vec{v} = \frac{1}{2}$, men ligesom med vektorer, kan de repræsenteres på flere former. Formen kaldet polær form er specielt interessant at betragte.



Figur 3: (Geogebra) Samme komplekse talplan som i figur 2. Denne gang med endnu et punkt z_2 (ikke det samme punkt som før)

I figur 3 er gitteret i baggrunden ændret til polær imodsætning til kartesisk, for at understrege forskellen, og nemmere kunne aflæse punktet. I et polært koordinatsystem er der to væsentlige mål, det er afstand fra polpunktet[5] og vinkel med førsteaksen. Punktet z_2 aflæses til at have afstanden 2 og vinklen 30 deg[6], hvilket kan opskrives på denne form: $z_2 = |2|\angle 30 \text{ deg}$[7]. For at finde de to egenskaber for $z_1$dannes en trekant mellem polpunktet, punktet selv og punktets projektion i førsteaksen.

[5] Svarer til origo i et kartesisk koordinatsystem.
[6] Den ligger på en linje som er 30 grader fra førsteaksen.
[7] (webmatematik.dk, 2012, Polært koordinatsystem)



Figur 4: (Geogebra) Punkt og trekant.

Figur 4 illustrerer den trekant, der dannes for at finde den numeriske værdi af z_1.

Kateterne kendes, da $a = Im(z_1)$ og $b = Re(z_1)$, ved hjælp af pythagoras beregnes siden c

$$a^2 + b^2 = c^2$$

$$\sqrt{a^2 + b^2} = c$$

$$\sqrt{Re(z_1)^2 + Im(z_1)^2} = |z_1|$$

$$|z_1| = \sqrt{1^2 + 2^2} = |\sqrt{5}|$$

Dermed kendes den generelle formel for den numeriske værdi af et komplekst tal:

$$|z| = \sqrt{Re(z)^2 + Im(z)^2}$$

For at kunne opstille z_1 på polær form mangler vinklen mellem punktet og førsteaksen blot at blive fundet. Den findes ved at bruge den formel, som kendes i forvejen, $v = \tan^{-1}(\frac{y}{x})$. I forbindelse med komplekse tal omtales denne vinkel, et *argument*, og noteres med $\arg(z) = \tan^{-1}(\frac{Im(z)}{Re(z)})$.

Men formlen er ikke generel, da den kun returnerer tallets hovedargument, når punktet for tallet ligger i 1. eller 4. kvadrant. Hvis det ligger i 2. eller 3. kvadrant er vinklen forskudt. Den modificerede formel til 2. og 3. kvadrant er dermed:

$$\arg(z) = \tan^{-1}(Im(z)/Re(z)) + \pi.$$

Siden punktet z_1 ligger i 1. kvadrant kan vinklen beregnes ved brug af den første formel, $arg(z_1) = \tan^{-1}(2/1) \simeq 63,43 \text{ deg}$[8][9]. Hermed, opstilles de to punkter,

$$z_1 = \left|\sqrt{5}\right| \angle 63,43 \text{ deg}$$

$$z_2 = |2| \angle 30 \text{ deg}$$

Når et komplekst tal er opstillet på polær form, findes der flere notationer. En notationsform, der gør det trivielt at foretage multiplikation og division af komplekse tal, er Eulers notation. Den er således, $|z|e^{i*v}$. De to punkter z_1 og z_2 kan udtrykkes på følgende vis:

$$z_1 = \sqrt{5} * e^{i*63,43\text{deg}}$$
$$z_2 = 2 * e^{i*30\text{deg}}$$

Når komplekse tal er noteret ved Eulers notation kan produktet findes ved at regne med potenser[10].

$$z_3 = z_1 * z_2$$
$$z_3 = \sqrt{5} * e^{i*63,43\text{deg}} * 2 * e^{i*30\text{deg}}$$

[8] Fra figur ?a3?
[9] (McLean, 2021, Komplekse tal 2)
[10]

Der flyttes på leddene,

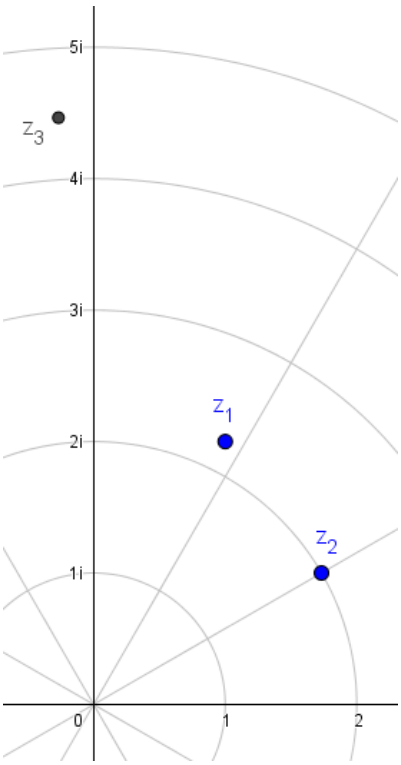
$$z_3 = \sqrt{5} * 2 * e^{i*63{,}43\text{deg}} * e^{i*30\text{deg}}$$

Eksponenterne samles,

$$z_3 = \sqrt{5} * 2 * e^{i*63{,}43\text{deg}+i*30\text{deg}}$$

Der foretages kompleks addition i eksponenten,

$$z_3 = 2\sqrt{5} * e^{i*90\text{deg}}$$



Figur 5: Punkterne z_1 og z_2, samt deres produkt, z_3.

Produktet af den komplekse multiplikation er indtegnet på figur 5. Ud fra de trin, der blev foretaget, kan en generel formel for multiplikation blive udledt,

$$z_3 = |z_1 \cdot z_2| \cdot e^{i \cdot (arg(z_1)+arg(z_2))}$$

Ligesom med multiplikation er division også ligetil, når tallet står på polær form.

$$z_3 = \frac{z_1}{z_2}$$

Det kan pakkes ud så der er mere at arbejde med:

$$z_3 = \frac{|z_1|*e^{i*argv(z_1)}}{|z_2|*e^{i*argv(z_2)}}$$

Når der er to potenser i en brøk (i tæller og nævner) som begge har samme grundtal, kan man bruge regnereglen, hvor dens eksponenter subtraheres.

$$z_3 = \frac{|z_1|}{|z_2|} * e^{i*\arg(z_1)-\arg(z_2)}$$

Endeligt, efter en enkelt reducering yderligere, er formlen for division af komplekse tal, udledt:

$$z_3 = \left|\frac{z_1}{z_2}\right| \cdot e^{i\cdot\arg(z_1)-\arg(z_2)}$$

En anden regneform, som er væsentlig at kende til, for at udlede formler til at arbejde med komplekse tal, er konjugering. Konjugering er den mest simple af alle disse typer beregninger, da man blot skal omvende et fortegn. For at konjugere et komplekst tal omvendes fortegnet for andet led. Når man betragter et komplekst tals sted, bemærkes det at det komplekse tal spejles i den reelle akse.
Givet $z_1 = 1 + 1i$, vil $\underline{z_1} = 1 - 1i$, konjugeringen med et tal markeres ved at sætte en streg over. Umiddelbart virker konjugering ikke særlig brugbar, da det blot er et omvendt fortegn. Følgende bevis, viser hvorfor den er brugbar.

Det er svært at foretage division på rektangulær form uden at tage den konjugerede divisor i brug. Divisionen af to komplekse tal foregår således, givet $z_2 = 2 + 3i$:

$$z_3 = \frac{z_1}{z_2}$$
$$z_3 = \frac{1+1i}{2+3i}$$

For at have mere at arbejde med under reduceringen, kan den konjugerede nævner multipliceres både i tæller og nævner. Her er den konjugerede nævner,

$$\underline{z_2} = 2 - 3i$$
$$z_3 = \frac{1+1i}{2+3i} * \frac{2-3i}{2-3i}$$

Formålet med at gøre det, er at reducere den imaginære enhed ud af nævneren. Til det bruges den originale definition $i = \sqrt{-1}$ eller $i^2 = -1$

$$z_3 = \frac{2-3i+2i-(1i*3i)}{4-6i+6i-(3i*3i)}$$

Her udregnes parentesen i hhv. tælleren og nævneren:

$$1i * 3i = 3 * i^2 = -3$$
$$3i * 3i = 9 * i^2 = -9$$

$$z_3 = \frac{2-3i+2i+3}{4+9} = \frac{5-1i}{13}$$

Endeligt, omskrives det til rektangulær form man er vant til

$$z_3 = \frac{5}{13} - \frac{1}{13}i$$

På samme måde, kan formlen udledes:

$$z_3 = \frac{z_1}{z_2}$$
$$z_3 = \frac{a+b*i}{c+d*i}$$
$$z_3 = \frac{a+b*i}{c+d*i} * \frac{c-d*i}{c-d*i} = \frac{ac-ad*i+bc*i-bd*i^2}{c^2-cd*i+cd*i-d^2*i^2}$$
$$z_3 = \frac{ac+bd-ad\cdot i+bc\cdot i}{c^2+d^2}$$

Der er til dels redegjort for komplekse tal og de regneregler, der gælder for komplekse tal — nogle både på polær- og rektangulær form. Det er mere relevant senere i opgaven, når der bliver kigget på sammenhængen mellem de komplekse tal og fourier transformation. Der vil blive redegjort for begreberne bag fourier transformation, og så kigges der på transformeringen rent matematisk, men først bliver der redegjort for summation.

Summation er en matematisk regneart, som er defineret som den gentagne addition af nogle tal, ofte ved hjælp af en variabel, som løbende inkrementeres[11]. For at bedre forstå, hvad det handler om, beregnes der et eksempel:

[11] Stigning i talværdi med 1

$$\sum_{n=0}^{N}(n)$$

I regneudtrykket, er store sigma, hvilket er det symbol, der bruges for at indikere summation. Under symbolet er den (heltallige) iterations variable defineret til at være 0. Ovenover symbolet ser man det heltal den *tæller* til. Ud for symbolet er det udtryk, som den iterations variable $n = 0$ indsættes i. For at udregne eksemplet, defineres $N = 5$

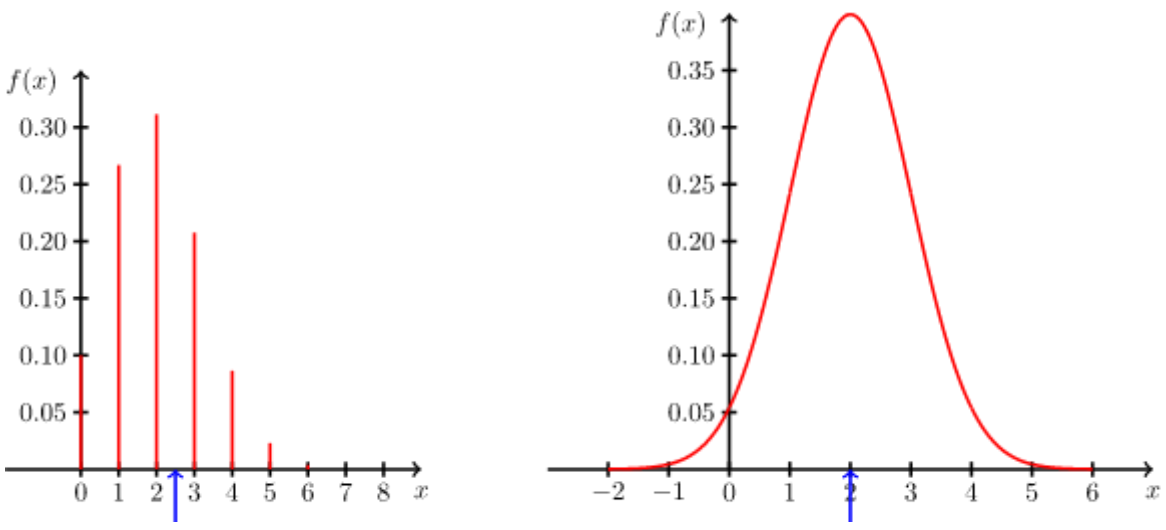
$$\sum_{n=0}^{5}(n) = (0) + (1) + (2) + (3) + (4) + (5) = 15$$

Det kaldes også for en sum række, da man løbende summere den talrække, der er udtrykt. Derudover er det også muligt at have mere avancerede udtryk:

$$\sum_{n=1}^{5}(n^2) = (1^2) + (2^2) + (3^2) + (4^2) + (5^2) = 51$$

I udtrykket ovenfor beregnes sumrækken for den iterations variable opløftet i 2. eksponent[12].

Sumrækker og komplekse tal er essentielle at kende til for at betragte diskret fourier transformering. Nedenfor vil der blive redegjort for DFT, forskellen på DFT og FFT, og hvorfor komplekse tal er smarte at arbejde med.

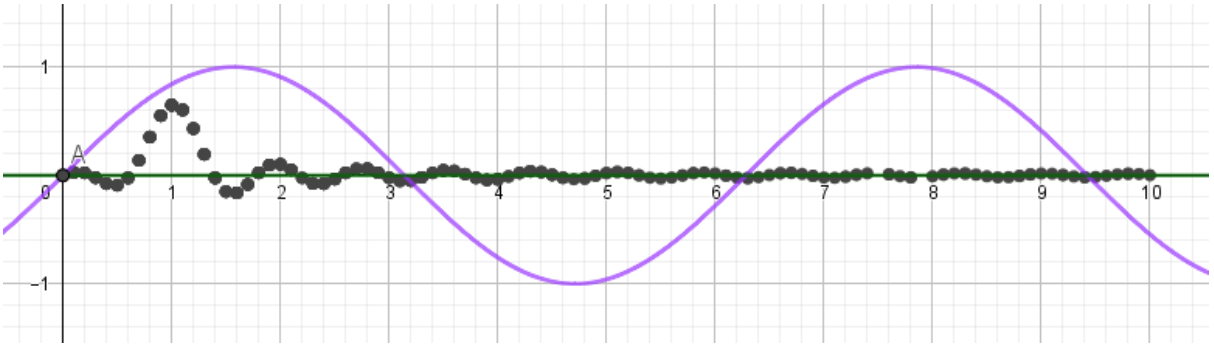


[12] (webmatematik.dk, 2012, Summationstegn)

Figur 6: Forskellen mellem diskret og kontinuerlig (NTNU, n.d.)

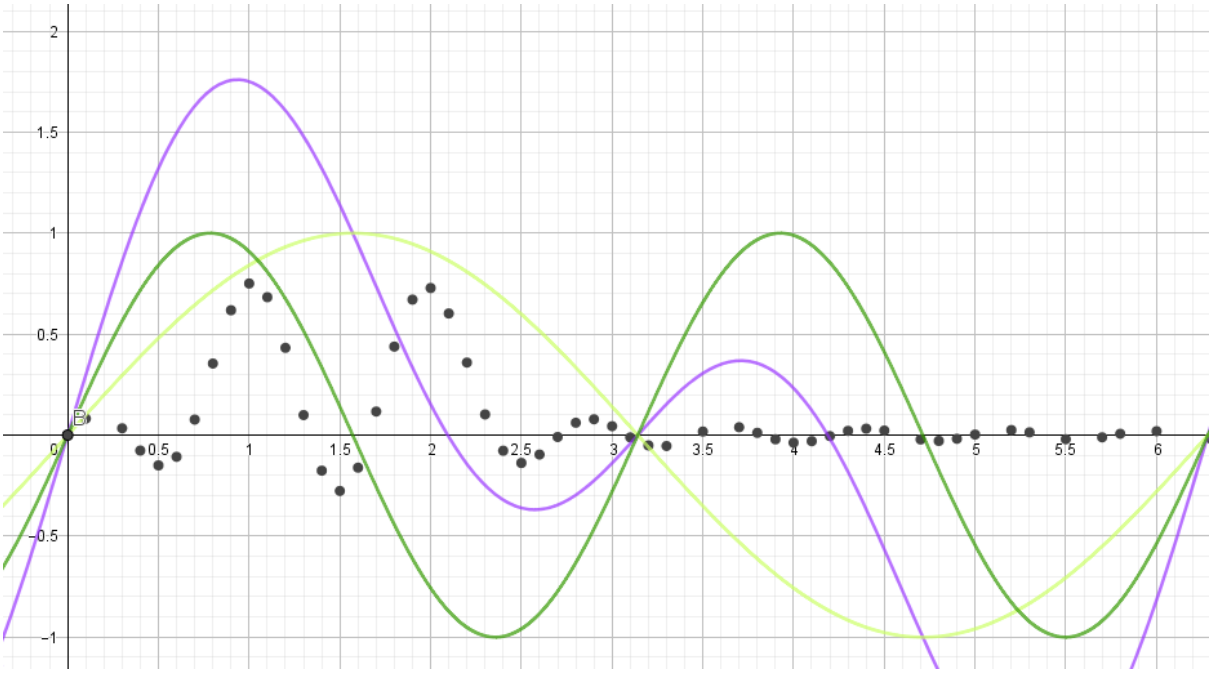
Fourier transformation kommer til udtryk i flere forskellige former og forkortelser. De mest brugte begreber er FT, DFT og FFT, hvilket henholdsvis står for *Fourier Transform*, *Discrete Fourier Transform* og *Fast Fourier Transform*.
Den første er det mest generelle begreb. Når begrebet bruges i samme kontekst som f.eks. *DFT*, henvises der ofte til forskellen mellem de to former — den førstnævnte er kontinuerlig, den anden er diskret. Den kontinuerlige er den teoretiske, mens den diskrete kan bruges i praksis. På figur 6 demonstreres forskellen på kontinuerlig og diskret, der er et histogram i venstre side, og en funktions graf i højre side.

Begrebet transformation er et vidt begreb inden for matematik. Det handler i virkeligheden blot om forskellige måder at repræsentere samme data på. Der kendes allerede forskellige måder at betragte en funktion. Eksempelvis hvis man finder den afledte funktion, betragter man væksten over den originale funktion — samme data, blot repræsenteret på en anden måde. Sagt på en anden måde, man justerer domænet af en funktion så dataene kan betragtes på en anden måde. Det er netop dette fourier transformation gør, men i stedet for at præsentere en funktions vækst, eller ændringen i vækst, betragter den en matematisk funktion med tid som domæne, og omformer den til at domænet repræsenterer frekvens, altså $1/tid$. Betragt følgende eksempel.



Figur 7: Geogebra opsætning af fourier transformation. Den lilla funktion er en vilkårlig funktion med tidsdomæne. De sorte prikker udformer en graf som svarer til samme funktion i frekvensdomæne.

På figur 7, er skitseret en funktion i sort, som kaldes $\hat{f}(\omega)$, som svarer til fourier transformationen for funktionen $f(t) = \sin(f_s * t)$[13], hvor $f_s = 1$ (frekvens). På samme måde som at man noterer differentialfunktionen med et lille tegn: f', bruges der en notation for en fourier transformeret funktion: $\hat{f}$. For grafen på figuren, gør det sig gældende at $\hat{f}(\omega)$ har maksimum i $\hat{f}(1)$, da $\hat{f}$ svarer til frekvensdomænet af $f(t) = \sin(1 * t)$, hvor frekvensen er 1. Ideelt set vil $\hat{f}(1) = 1$, siden amplituden af sinuskurven med frekvensen 1 er lig 1.



Figur 8: Samme opsætning som før, men nu med to sinuskurver, hvor den lilla er summen

For at tage udgangspunkt i et eksempel, betragt funktionerne på figur 8. Den lilla er en summation af to sinuskurver med henholdsvis frekvens på 1 og 2, hvor de grønne er dens addender. Udover de tre funktioner, er der igen skitseret fourier transform for den lilla funktion. Den har toppunkter i 1 og 2, hvilket giver mening, da den lilla funktionsforskrift er således:

$$f(t) = sin(t) + sin(2t).$$

Det er den primære egenskab fourier transformation har, og det er derfor den er brugbar indenfor mange fag. Når der arbejdes med fouriertransformation praktisk, er det ofte

[13] Bemærk, den uafhængige variabel er *t* for tid, og *fs* er frekvensen.

den diskrete udgave som benyttes, da man ofte arbejder med diskrete data sæt. I forklaringen af den diskrete fouriertransformation, vil der blive taget udgangspunkt i den diskrete fouriertransformation der defineres således:

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-\frac{i2\pi}{N}k*n}$$

Hvor X er defineret som det sæt af komplekse tal, der ender med at repræsentere frekvensdomænet for det datasæt man indsætter. Her er k hvert indeks af det sæt af tal man arbejder med. Den komplekse liste er defineret ved en summation. Udtrykket inde i summationen er x_n, hvilket er de datapunkter der skal transformeres fra tidsdomænet til frekvensdomænet. Multipliceret x_n er udtrykket $(-\frac{i2*\pi}{N}k * n)$ som eksponent til den naturlige eksponentialfunktion. Summationen går fra $n = 0$ til N, hvor N er antallet af punkter. Der findes også en omvendt funktion, som transformerer et signal tilbage til tidsdomænet.

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n * e^{\frac{i2\pi}{N}k*n}$$

Formlerne indeholder et komplekst led, som eksponent til Eulers tal. Som tidligere beskrevet, ligger komplekse tal tæt op ad vektorer, så hvorfor bruger man ikke vektorer i stedet for komplekse tal, når komplekse tal virker, som en overkomplicering af problemet.

Betragter man formlen med den hensigt at bruge vektorer, kan man tage Eulers formel i brug. Eulers formel beskriver sammenhængen mellem en kompleks eksponentialfunktion og de trigonomiske funktioner[14].

$$e^{i\cdot x} = \cos(x) + \sin(x)\cdot i$$

Hvilket gør det muligt at opstille formlen på følgende vis:

$$X_k = \sum_{n=0}^{N-1} x_n * \cos(\frac{2\pi}{N}k * n) - \sin(\frac{2\pi}{N}k * n) * i$$

Hvilket svarer til en vektor, hvor cos er x-komposanten, og sin er y-komposanten.

[14] (Wikipedia, n.d., Eulers formel)

$$X_k = \sum_{n=0}^{N-1} x_n * \frac{\cos(\frac{2\pi}{N} k*n)}{-\sin(\frac{2\pi}{N} k*n)}$$

Taget udgangspunkt i den første udledte formel, svarer det også til følgende (ifølge regneregler for summation):

$$X_k = \sum_{n=0}^{N-1} (x_n * \cos(\frac{2\pi}{N} k * n)) - i * \sum_{n=0}^{N-1} (x_n * \sin(\frac{2\pi}{N} k * n))$$

Det viser sig at komplekse tal har en væsentlig egenskab som vektorregning ikke har — multiplikation. Siden multiplikation ikke er defineret for vektorer, er det ikke muligt at samle det til ét udtryk, som både repræsenterer x- og y-komponentet, og derfor er det nødvendig at foretage denne samme beregning to gange i stedet for blot en. Det er fordelen bag komplekse tal inden for dette felt.

Noget andet væsentligt at forstå inden for feltet, er når man arbejder med diskrete frekvenser så omtaler man de resulterende frekvenser som frekvens grupper, eller *frekvensspande*[15]. Årsagen bag dette er at det ikke er muligt at repræsentere enhver frekvens med en diskret opløsning[16]. Derudover vil de resulterende tal fra transformationsfunktionen ikke korrelere direkte med frekvensområde. Ud fra Eulers formel tidligere gør det sig gældende at den reelle del af det komplekse resultat svarer til en given frekvens' indhold af cosinus-forskudte sinuskurver, mens den imaginære del svarer til sinuskurver. Dermed svarer amplituden til modulus af tallet.

Som følge af definitionen på udtrykket, er hver frekvens gruppe defineret ved udtrykket

$$x_n * e^{-\frac{i*2\pi}{N} k*n}$$

For at bedre forstå, hvordan udtrykket udformer sig ifølge forskellige variable og størrelser, vil der blive foretaget et regneeksempel.
Hvis man lader $x_n = \{1, -1, 1, -1\}$, og efterligne en periodisk funktion. Da antallet af elementer i $x_n = 4$, bliver $N = 4$.

[15] Fra det engelske begreb *frequency bin*.
[16] I modsætning til kontinuert signal

$$X_0 = \sum_{n=0}^{4-1} x_n * e^{-\frac{i*2\pi}{4}0*n} = (1 * e^{-\frac{i*2\pi}{4}0*0}) + (-1 * e^{-\frac{i*2\pi}{4}0*1}) + (1 * e^{-\frac{i*2\pi}{4}0*2}) + (-1 *$$

Eksponenten til potensen i udtrykket går 0, da multiplikatoren til kvotienten er 0 ($k = 0$)

$$X_0 = \sum_{n=0}^{3} x_n * e^{-\frac{i*2\pi}{4}k*n} = (1) + (-1) + (1) + (-1) = 0$$

Det samme gælder de andre udtryk for X

$$X_1 = \sum_{n=0}^{4-1} x_n * e^{-\frac{i*2\pi}{4}1*n} = (1 * e^{-\frac{i*2\pi}{4}1*0}) + (-1 * e^{-\frac{i*2\pi}{4}1*1}) + (1 * e^{-\frac{i*2\pi}{4}1*2}) + (-1 * e^{-\frac{i*2\pi}{4}1*$$

Her benyttes nogle af de regneregler for komplekse tal, der blev redegjort for tidligere.

$$X_1 = \sum_{n=0}^{3} x_n * e^{-\frac{i*2\pi}{4}1*n} = (1) + (-1 *- 1i) + (1 *- 1) + (-1 * 1i) = 1 + 1i + (-1) + (-$$

Dette reduceres yderligere:

$$1 + 1i + (-1) + 1i = 1 + (-1) + 1i + (-1i) = 0$$

De to sidste iterationer af X_k beregnes udelukkende i Mathcad.

$$X_2 = \sum_{n=0}^{3} x_n * e^{-\frac{i*2\pi}{4}2*n} = 1 + 1 + 1 + 1 = 4$$

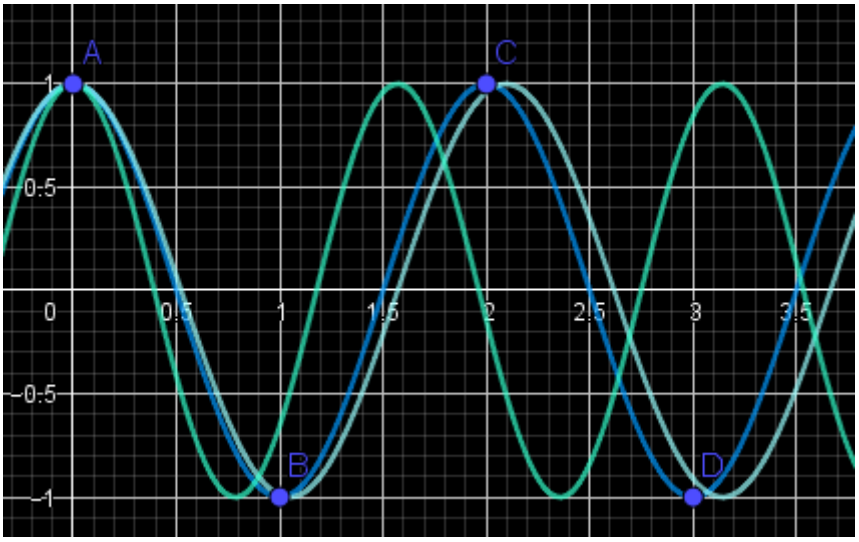
$$X_3 = \sum_{n=0}^{3} x_n * e^{-\frac{i*2\pi}{4}3*n} = 1 + (1i) + (-1) + 1i = 0$$

Ud fra beregningerne ovenfor, kan X_k defineres til følgende $X_k = \{0, 0, 4, 0\}$, hvilket kan forklares ved at gennemgå mere teori for fouriertransformation. Som nævnt tidligere svarer X ikke direkte til amplituden af signalet for hver frekvens, men det gør modulus af X_k divideret med N, da den samlede amplitude af frekvensdomænet skal svare til den samlede amplitude af tidsdomænet.

$$|X_k| = \{0, 0, \sqrt{4^2 + 0^2}, 0\} = \{0, 0, 4, 0\}$$

$$\frac{|X_k|}{N} = \{\frac{0}{4}, \frac{0}{4}, \frac{4}{4}, \frac{0}{4}\} = \{0, 0, 1, 0\}$$

Siden domænet nu svarer til frekvens, ikke tid, svarer elementernes indeks til den frekvens gruppe de er i. Den første gruppe: [0; 1[udgør en styrke på $\frac{0}{4}$ i den originale funktion, det samme gælder den anden gruppe [1; 2[og den fjerde gruppe: [3; 4[. Den tredje gruppe: [3; 4[udgør en signalstyrke i den originale graf på $\frac{4}{4}$.



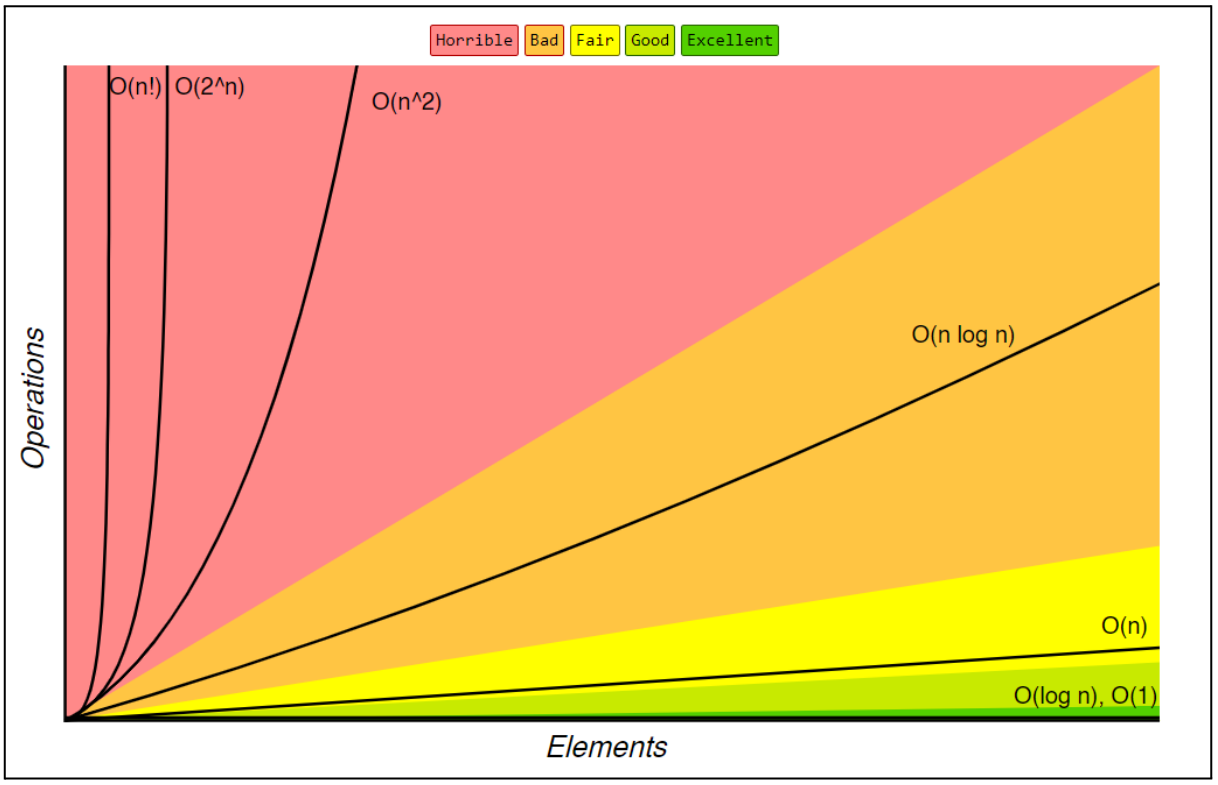
Figur 9: Repræsentation af den frekvens gruppe, signalet ligger indenfor, turkis er den højeste frekvens repræsenteret (4), mørkeblå er den faktisk frekvens (π) og lyseblå er den mindste frekvens (3)

På figur 9 ses punkterne $x = \{1, -1, 1, -1\}$ ($y = \{0, 1, 2, 3\}$). Derudover er funktionen $1 \cdot \cos(3t)$(lyseblå), $1 \cdot \cos(4t)$(turkis) og den faktiske funktion $1 \cdot \cos(\pi)$(blå) indtegnet. Grunden til cosinus bliver brugt, og ikke sinus, er teorien fra tidligere: Da resultatet udelukkende er den reelle del, betyder det at en funktion af sin indenfor frekvens [3; 4[udgør en styrke på $\frac{Im(X_3)}{4} = 0$. Der er indtegnet flere funktioner for at illustrere det interval frekvens gruppen dækker, fra den lyseblå til turkise funktion.

Hermed er der redegjort for DFT og dens sammenhæng med komplekse tal og signalbehandling. Samlet set, har fourier transformering en egenskab, der gør det muligt at dekomponere et enkelt signal til en sum af sinuskurver, hvilket gør mange forskellige

discipliner inden for signalbehandling mulig. Følgende, vil der blive redegjort for forskellene mellem DFT, FT og FFT.

Forskellen på Fourier Transformation og den diskrete udgave er tydelig. Derimod er forskellen på DFT og FFT mindre åbenbar, siden FFT, ligesom DFT, også bearbejder diskrete input. Ved at kigge på store-O notationen for de to algoritmer, kommer forskellen til kende. Indenfor computer-videnskaben bruges notationen $O(n^{a} * b)$ (eller lignende med andre matematiske funktioner såsom fakultet), hvor n er størrelsen på det input funktionen er givet (det kan f.eks. være antallet af elementer i en matrix, eller bare størrelsen på et tal), og a, $b$er konstanter, som tilpasses baseret på algoritmen der kigges på. Det man beskriver med store-O udtryk er typisk den tid en algoritme kører i, eller den plads den kræver for at kunne udregne resultatet[17].



Figur 10: Diagram over klassiske store-O varianter. (Drowell, n.d.)

Ved analyse af DFT, kan man drage følgende konklusioner:

[17] Når et store-O udtryk beregnes er der ikke nogen enheder eller andre former for dimensioner på, så der beregnes ikke noget konkret. Det svarer til et størrelsesforhold. Det bruges derfor mest til at sammenligne algoritmer.

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-\frac{i2\pi}{N}k*n}$$

Siden længden af X er lig N, og hvert enkelt element X_k defineres ved en sumrække, som itererer ved $n = 0$ til $N - 1$, gør det sig gældende at der forekommer en indlejret løkke, som itererer over samme tid som den yderste løkke. Sagt med andre ord, for at beregne X foretages der $N * N$ beregninger. Som store-O notation, udtrykkes dette $O(n^2)$. I oversigten på figur 10 er dette udtryk i det røde felt, da antallet af beregninger, der skal foretages, vokser så kraftigt sammen med N at algoritmen er uanvendelig for større datasæt, naturligt som følge af dette er der efterspørgsel på en hurtigere algoritme. Denne algoritme kaldes *hurtig* fourier transformering. Den specifikke FFT, der tages udgangspunkt i foregår i $O(n * log(n))$, ved at udrydde nogle overflødige beregninger fra DFT. Det vil sige at den bygger på den mere fundamentale, Discrete Fourier Transform.

Med det, er den endelige redegørelse over komplekse tal, fourier transformering, og deres sammenhæng draget. Det viser sig at komplekse tal bliver brugt som et værktøj indenfor DFT, for at undgå at foretage dobbelt så mange beregninger, som der normalt ville blive lavet. Derudover er DFT en fundamental formel til om transformering af en funktions med tidsdomæne til samme funktion men med frekvens som domæne, som gør det muligt at manipulere med signaler og repræsentere dem på en anden måde.

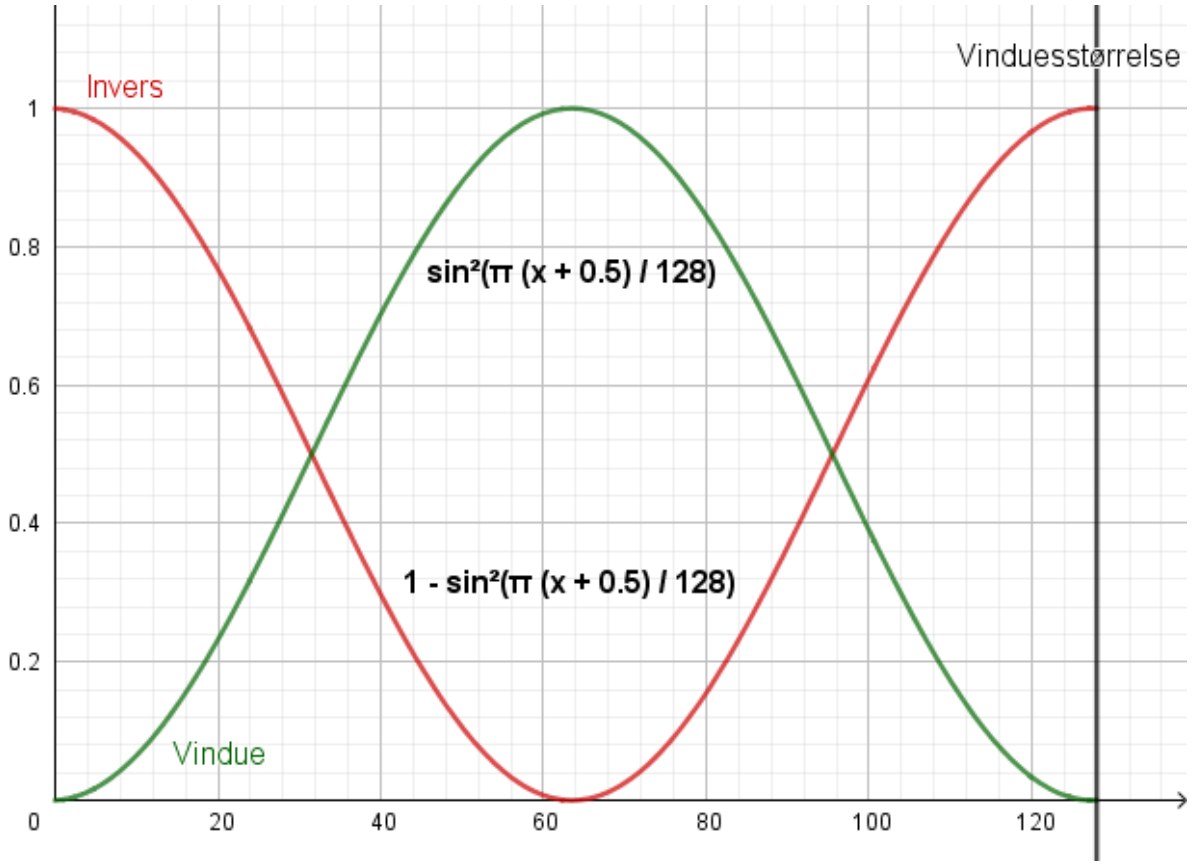
Udarbejd et eksempel på en implementering af fast fourier transformation

Undersøgelsen af fast fourier transformerings algoritmer, som er let tilgængelige, har bragt en række forskellige algoritmer at vælge imellem. Den algoritme der specifikt er taget udgangspunkt i, er kaldet *Cooley-Tukey*.

For at implementere fast fourier transformering, skal man først have en forståelse for, hvad den teoretiske forskel på FFT og DFT er. Tidligere i opgaven blev der kigget på den væsentlige del af FFT — dens komputationelle hastighed, men ikke hvad FFT er. Som

nævnt tidligere, er den hurtige fourier transformation ikke en variant af DFT, men bygger derimod ovenpå den og gør den hurtigere.

Når man blot beregner den diskrete fouriertransformation for en given serie af punkter, vil der være nogle forskellige effekter/artefakter som er svære at gøre op for, og som gør de beregnede værdier mindre brugbare. En af disse effekter er spektrallækkage, hvilket sker når signalet man arbejder med, ikke starter og slutter med en styrke på 0. Når man foretager FT på data, antager algoritmen at dataene går i ring, altså at det er et slags kontinuert signal. Måden hvorpå man forebygger denne effekt og udbedrer den er ved at benytte en vinduesfunktion. En vinduesfunktion er en matematisk model som udstrækkes i $[0; N]$, hvor N er antallet af datapunkter. En forudsætning for en vinduesfunktion er $w(0) = 0 \wedge w(N) = 0$, dette er en af de mest væsentlige dele af vinduesfunktionen. Som forklaret ovenfor er det vigtigt at det udsnit af data man kigger på, starter og slutter i 0, for at fouriertransformation funktionen betragter dataene korrekt.



Figur 11: Graf over den vinduesfunktion, der bliver brugt i den hjemme-implementerede FFT, og dens inverse funktion (i forhold til $Vm(vindue) = [0; 1]$). Begge definerede ved at $N = 128$

På figur 11 er illustreret den matematiske funktion:

$$w(t) = \sin(\pi\frac{x+0.5}{N})^2, 0 < t < N, \text{hvor } N = 128$$

Den følger de krav der blev stillet tidligere om at toppunktets y-værdi er 1 og ligger i $N/2$, og at dens $w(0) = 0$ og $w(N) = 0$. Med andre ord, denne funktion virker fornuftig til brug af vinduesfunktion. Den omvendte funktion, findes nemt ved at omvende funktionens y-værdier ved at omvende fortegnet, og udtrykket skal så være trukket fra 1, i stedet for 0, for at holde værdierne over x-aksen.

Sideløbende den teoretiske forklaring, vil der blive vist kode udklip fra implementeringen af FFT. Fordi FFT typisk tager form som en rekursiv funktion[18], bliver koden til selve implementeringen skrevet i det funktionelle sprog F#. Derudover, da det funktionelle programmeringsparadigme bygger på den matematiske forståelse af en funktion. Derfor giver det også god mening at implementere denne del af koden i F#. Endeligt, siden både F# og, det objektorienterede sprog, C# er CLR-sprog og begge bygger på .NET, kan de godt interagere. Dette er praktisk siden der senere vil blive udarbejdet et program i C#, som skal kunne benytte implementeringen.

```
let Forward (amplitudes : float array) =
    let N = amplitudes.Length
    let window k = Math.Sin(Math.PI * (float k + 0.5) / float N) ** 2.0
```

Figur 12: F# kode for vinduesfunktionen, hele vindue funktionen er defineret på linje 3

På figur 12 findes metoden *Forward*, som tager imod en matrix af decimaltal. Konkret i forhold til lyd svarer parameteren til det *vindue* af amplituder, man vil oversætte til frekvensdomænet.

[18] Programmatisk set; inden for matematikkens verden er metoden altid rekursiv.

```
let X = [| for k in 0..(N-1) -> Complex(amplitudes.[k] * window(k), 0.0) |]
forwardComputations X
```

Figur 13: Koden som starter beregningerne

På figuren ses den del af koden som endeligt kalder *forwardComputations*, hvilket er den metode, som er rekursiv og laver selve beregningerne. Men før det, skal input dataene behandles så de er klar til at komme ind i den rekursive metode. Der er en løkke som definerer *X*, den omformer den array med float værdier til et array af *Complex*. Det sker idét den bruger vinduesfunktionen på værdierne.

Før der dykkes yderligere ned i koden, bliver teorien bag FFT uddybet. Som nævnt tidligere, bygger FFT's hastighed på refaktorering af DFT.

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} * e^{-\frac{2\pi * i}{N}(2n)*k} + \sum_{n=0}^{N/2-1} x_{2n+1} * e^{-\frac{2\pi * i}{N}(2n+1)*k}$$

Formlen ovenfor er DFT men delt op i en lige og en ulige del. Den virker ved at problemet bliver delt op i fortsat mindre dele. Den første del, er alle de lige indeks i listen *x* og det andet led er alle de ulige indekser. Denne opdeling fortsætter, det vil sige at den samme opdeling af lige og ulige indekse sker i det første led. Dette fortsætter indtil $N = 1$. Denne teknik kaldes *divide and conquer* indenfor computervidenskab[19].

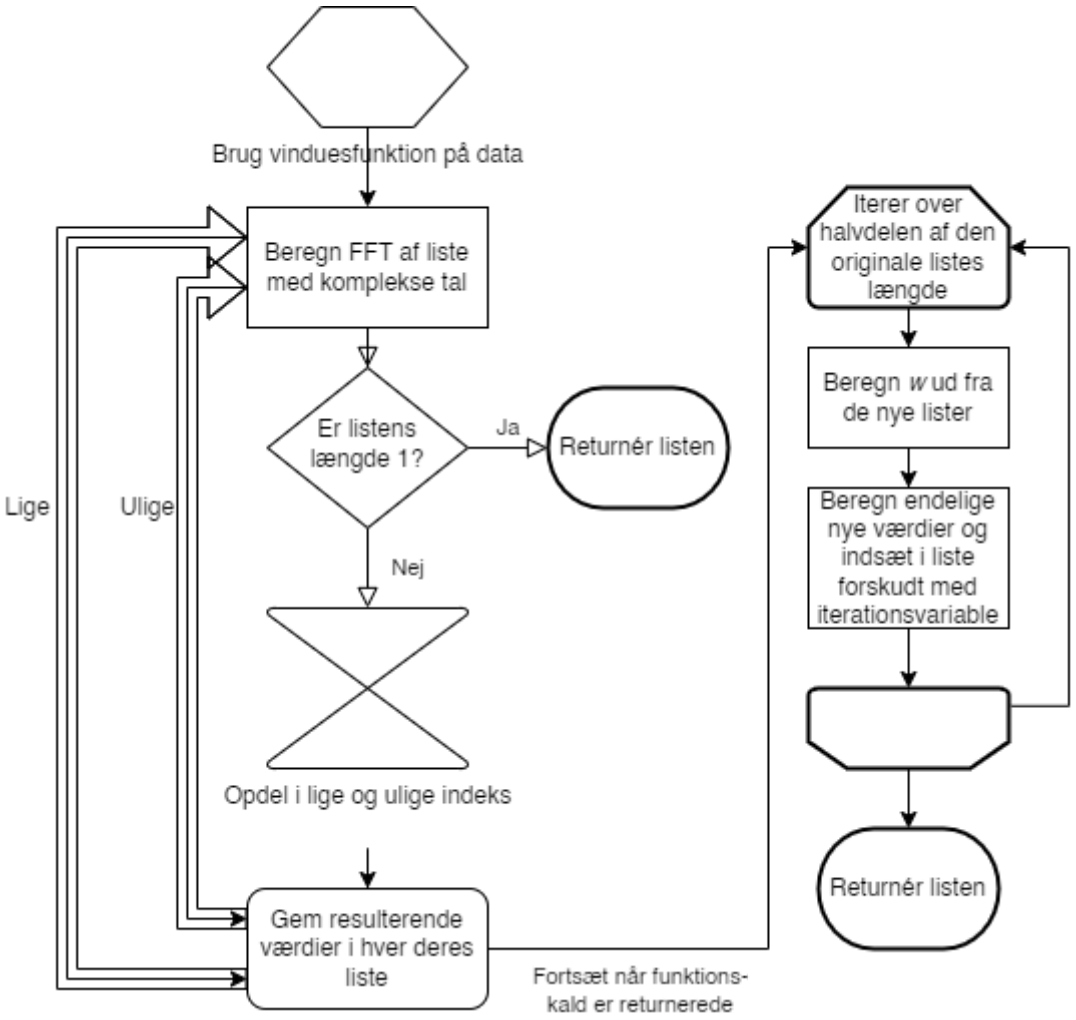
```
let rec forwardComputations (amplitudes : Complex array) =
    let N = amplitudes.Length

    if N = 1 then
        amplitudes
    else
        let (even, odd) = splitArray(amplitudes)

        let evenArray = forwardComputations (even |> Array.ofList)
        let oddArray = forwardComputations (odd |> Array.ofList)

        for k in 0..(N/2-1) do
            let w = exp(Complex(0.0, -2.0 * Math.PI * float k) / Complex(float N, 0.0)) * oddArray.[k]
            amplitudes.[k] <- evenArray.[k] + w
            amplitudes.[N/2 + k] <- evenArray.[k] - w

        amplitudes
```

Figur 14: Den rekursive funktion, som tager et array og følger Cooley-Tukey algoritmen.

[19] (Xu, 2015)

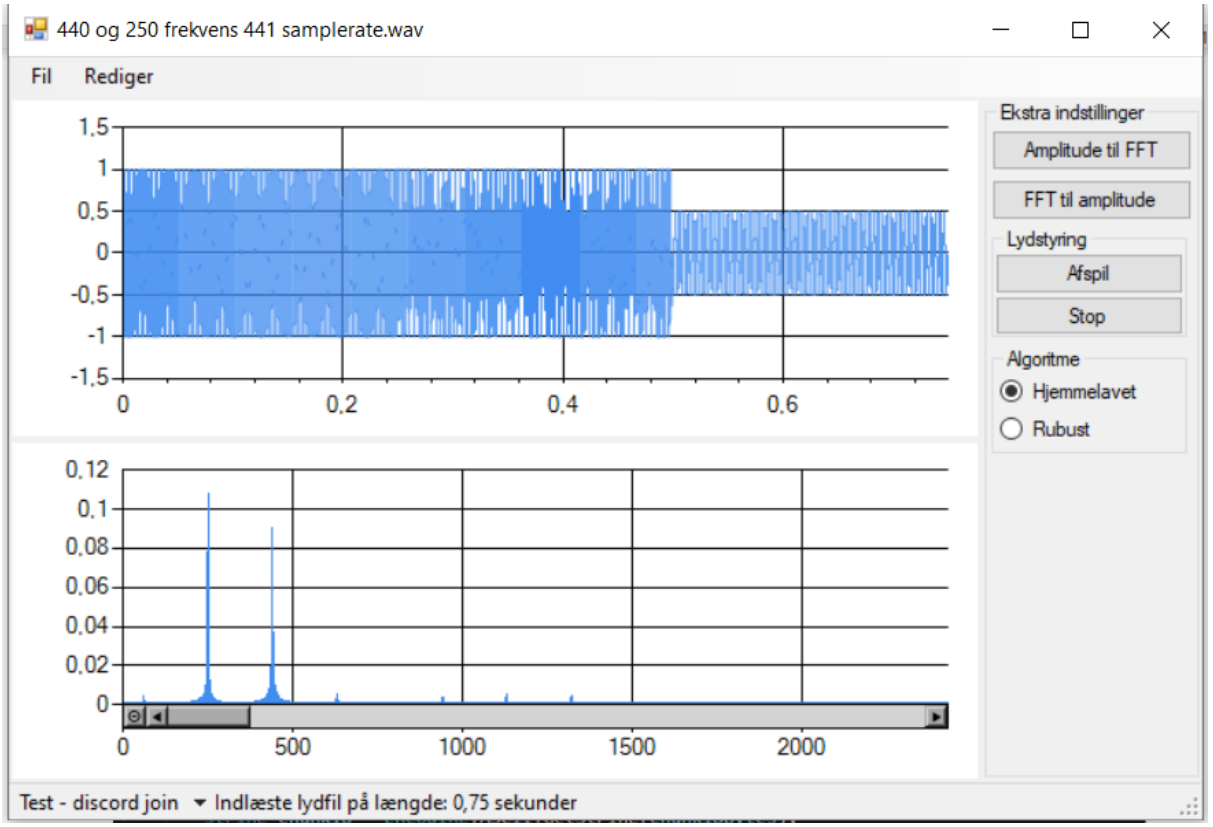
I F# skal en funktion have keywordet "rec" for at det er muligt for den at kalde sig selv. Det første, der sker, er at *N*bliver defineret øverst til antallet af elementer i listen *Amplitude*. Derefter kan man se det der blev nævnt tidligere, med at når $N = 1$ så returnerer den. Hvis $N > 1$ så deles *amplitudes* op i 2 arrays ved at bruge metoden *splitArray*. Så kalder metoden sig selv og det er her rekursionen i virkeligheden begynder. Rent programmeringsteknisk bliver resultatet af split array lavet om til et array igen ved at bruge en operatør ved navn *forward-pipe*. Den fører bare outputtet af en funktion ind i funktionen efter tegnet, listen bliver dermed ført ind i funktionen *Array.ofList*. Siden metoden er rekursiv, danner den et træ af metodekald som i starten alle er lige indeks, til sidst når den en hvor $N = 1$, så begynder det langsomt at arbejde sig tilbage op gennem call stack, hvor den afvikler de lige arrays først.

Når et metodekald har afviklet alle sine rekursive kald, så mangler den bare den nederste løkke, hvor den muterer *amplitudes* ifølge formlen. Først udregnes og defineres udtrykket i den naturlige eksponentialfunktion til variablen $w = e^{-\frac{2*\pi*k}{N}}$. For at forkorte de andre udtryk og blot addere variablen *w*, er *w* i virkeligheden $x_{2n} * e^{-\frac{2\pi*k}{N}}$. Til sidst i metoden returneres den array der er muteret.



Figur 15: Flowchart over Cooley-Tukey algoritmen

I flowchartet på figur 15 er forsøgt at fange de væsentlige dele af rekursionen og behandlingen af lister i algoritmen. Udover den standardiserede ikonografi, er der hjemmelavede pile, som skal gengive rekursions effekten. Pilene i venstre side skal forestille et kald til funktionen selv (de store pile), som returnerer sin værdi til at blive gemt i en liste hver især (de små sorte pile inde i de store pile).



Figur 16: Billede af spektralanalyse af et lydsignal bestående af 440 Hz og 250 Hz sinus

Algoritmen tages i brug praktisk, da den er implementeret i programmet (som dokumenteres senere). På figur 16 er vist hvor akkurat spektralanalysen af signalet på 440 Hz og 250 Hz. I histogrammet, ser man to søjler ved 440 Hz og 250 Hz, hvilket viser at den er i stand til faktisk at give en idé om, hvilken frekvens sammensætning signalet består af.

Der er hermed udarbejdet en implementering af Cooley-Tukey algoritmen, som er i stand til at omforme data til frekvensdomænet til fin præcision. Denne implementering er lavet i et funktionelt sprog for at gengive den matematiske definition så godt som muligt. Implementeringen er blevet testet i det endelige program udarbejdet i denne opgave.

Udarbejd et program, som kan bearbejde lydfiler i praksis

Når en lydfil skal behandles og bearbejdes, er der nogle generelle koncepter, der er væsentlige at forstå. Først og fremmest skal det lydfilformat, der læses, kendes. Den information man gerne vil have er blandt andet hvilken standard den er beskrevet efter. For opgaven, er lydfiltypen, *Wave,* specifikt blevet undersøgt.

Wave-formatet er opbygget efter RIFF-standarden, hvilket er en filstandard, som er baseret på idéen om at opdele en fil i flere *chunks*[20]. Hver blok starter med et ID, og størrelsen på resten af blokken. Blokopdelingen bliver brugt til at organisere data i en fil, eksempelvis for en RIFF billedfil kunne man gemme dimensionerne for billedet i en blok og så pixeldata i en anden blok[21]. Wave-formatet er opdelt i tre dele. Først er der headeren af filen, det er blokken, den indeholder nogle kendte værdier, og så indeholder den to mindre dele kaldet underblokke, siden de er underordnet den blok de er i.

En bloks ID er en kort tekststreng, som identificerer en blok. Headeren af en Wave-fil indeholder altid teksten "RIFF", det er for at programmer som læser filen har andet at tage udgangspunkt i end blot slutningen på filens navn. Fagbegrebet for dette er *magic bytes*. Efter ID'et, er størrelsen på resten af filen.

Når man læser en fil i et program, læser man den få stumper ad gangen. Det vil sige, at i koden bliver der slavisk tjekket om de værdier der er i filen, stemmer overens med det man forventer, og så gemmes de.

[20] Chunks vil blive refereret til som blokke.
[21] (Library of Congress, n.d., WAVE-formatet)

```
/// <exception cref="ArgumentNullException">filePath er null.</exception>
/// <exception cref="FileNotFoundException">Filen blev ikke fundet.</exception>
/// <exception cref="UnknownFileFormatDescriptorException">Filens chunk ID var ikke 'RIFF'.</exce
/// <exception cref="UnknownFileFormatException">Filens format ID var ikke 'WAVE'.</exception>
/// <exception cref="MissingSubchunkException">Filen indeholder ikke alle de nødvendige subchunks
public WaveFile(string filePath) : base(filePath)
{
    if (filePath is null)
    {
        throw new ArgumentNullException(nameof(filePath));
    }

    // Kast exception hvis filen ikke eksisterer.
    using (FileStream stream = File.OpenRead(filePath))
    {
        // Primary chunk
        byte[] chunkIDBytes = new byte[sizeof(int)];
        stream.Read(chunkIDBytes, 0, chunkIDBytes.Length);
        string chunkID = Encoding.ASCII.GetString(chunkIDBytes);

        if (chunkID != "RIFF")
        {
            throw new UnknownFileFormatDescriptorException($"Was '{chunkID}', expected 'RIFF'");
        }
```

Figur 17: Starten af constructoren for klassen Aud.IO.WaveFile

Koden i figur 17 viser starten af implementeringen af Wave-formatet. Den tager imod en parameter, som indeholder stien til den fil, der skal læses. Det første, som sker i constructoren, er at der verificeres at parameteren *filePath* ikke er null. Siden streng-værdier er referencetyper og kan derfor have værdien *null*, hvis de ikke er blevet tildelt en værdi. I dette tilfælde er det ikke meningen at den er *null*, så der gives klar besked, hvis værdien er det.

Derefter åbnes en stream for filen i et using-statement for at streamen automatisk lukkes, når den er færdig med at blive brugt. Derfra kan man begynde at læse filens indhold. En byte array på størrelsen af en integer instantieres[22]. Sizeof bruges for at understrege, hvor størrelsen kommer fra (modsætning til at skrive 4). Derefter bliver de *næste* 4 bytes læst fra filen og ind i det array, som blev instantieret[23]. Endeligt oversættes de 4 bytes til en ASCII-streng, og programmet verificerer at der står "RIFF". Hvis der ikke gør, kaster den en exception, som forklarer at filen ikke er en korrekt wave-fil. Denne

[22] En integer, er en datastørrelse som svarer til 4 bytes, hvilket giver mening når ID'et er 4 tegn på 1 byte hver.
[23] Ordet næste er kursiveret for at understrege at man altid læser den næste data, som er tilgængelig, hvilket i dette tilfælde er den første data.

exception er dokumenteret med XML-kommentar før metoden, for at man kan forebygge og håndtere denne type exception, når man bruger denne klasse.

Ud fra den information man lærte om wave-tidligere, kan man forvente at det næste i filen er størrelsen på resten af filen, fordi man nu har tillid til filen om at den følger RIFF. Det sidste filens header indeholder, er specifikt for hovedblokken, det er filens format. Dette felt er også en type ID, da det bruges til at konkret identificere filtypen som wave, så det ikke bare er en anden filtype som følger RIFF-standarden, man er i gang med at læse.

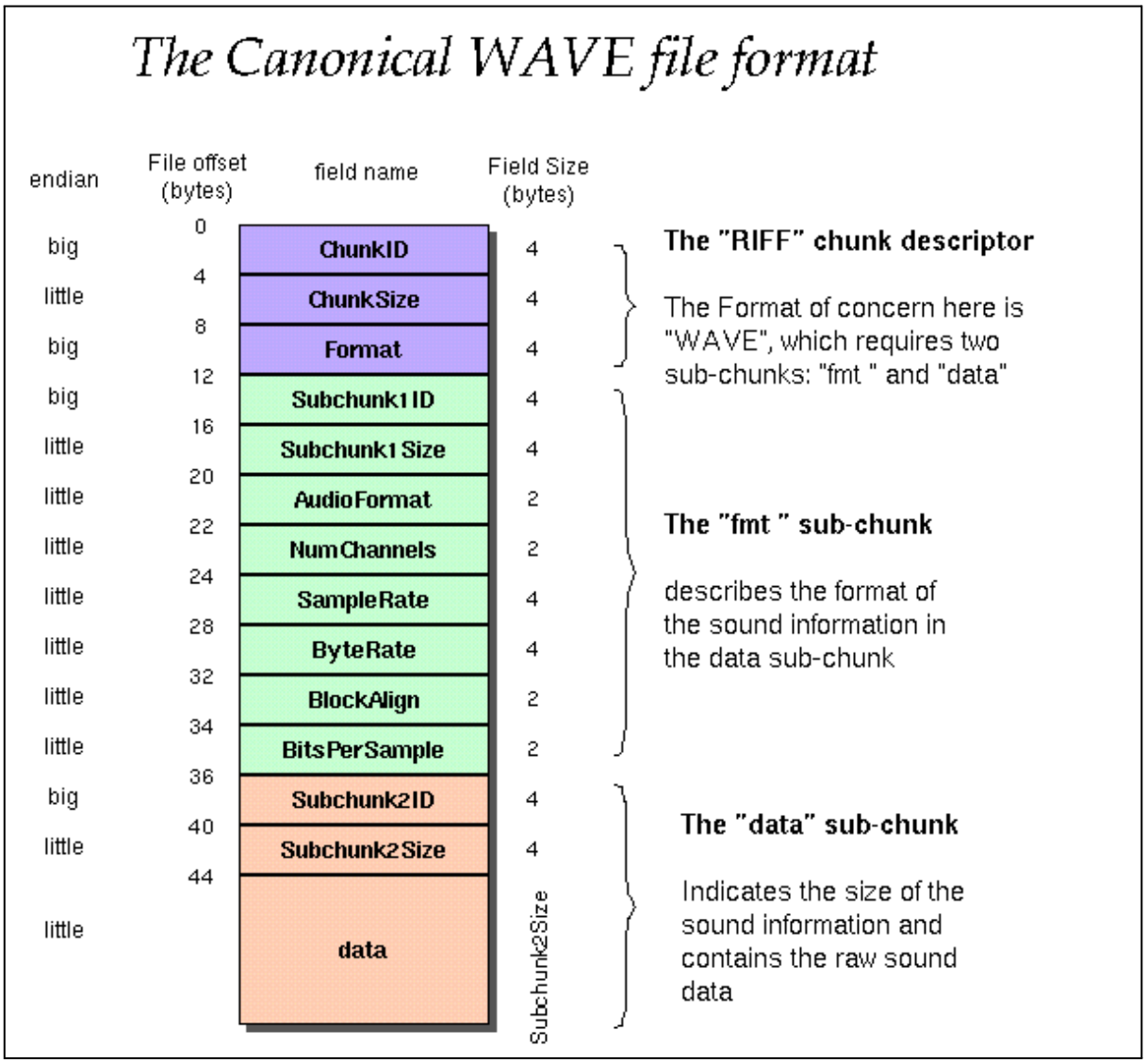
```
byte[] chunkSizeBytes = new byte[sizeof(uint)];
byte[] formatBytes = new byte[sizeof(int)];
stream.Read(chunkSizeBytes, 0, chunkSizeBytes.Length);
stream.Read(formatBytes, 0, formatBytes.Length);
string format = Encoding.ASCII.GetString(formatBytes);

if (format != "WAVE")
{
    throw new UnknownFileFormatException($"Was '{format}', expected 'WAVE'");
}
```

Figur 18: Næste del af constructoren for klassen Aud.IO.Formats.WaveFile

I figur 18 dokumenteres den næste stump af filen. Dette er størrelsen, og formatet. Ligesom tidligere, instantieres nogle byte arrays for at læse de næste værdier. Bemærk her bruges sizeof(uint) ved den ene, det er endnu en gang for at understrege, hvilken datatype den aflæste værdi er. Siden uint og int begge er 4 bytes, er forskellen blot at når værdien er aflæst, kan den kun have positive talværdier, hvilket praktisk set forlænger de værdier den kan have. Normalt kan en int gemme værdier mellem 2^{31} til $2^{31} - 1$, mens uint betragter værdier mellem 0 og 2^{32}.

De to værdier aflæses fra filstream til de to arrays, og formatet oversættes til ASCII-streng som følger samme konvention som tidligere — er værdien uventet: kastes en exception, for at understrege det. Størrelsesværdien oversættes ikke fra bytes, men læses alligevel for at følge samme mønster, og skubbe stream pointeren frem for at læse

de næste værdier. Grunden til at værdien ikke benyttes er, at den bliver udregnet bagefter, og der er ingen grund til at bruge den når hele filen bliver læst alligevel.



Figur 19: Hjælpende oversigt over WAVE-formatet. (Sapp, n.d.)

Når filformatoplysningerne er læst, er den første blok læst færdig og stream pointeren er nu nået til den første underblok. Figur 19 er en oversigt over de felter formatet gemmer på, hvad de indeholder, og hvor store de er. Den første underblok den beskriver er *Format*-underblokken. Denne blok indeholder information om, hvordan lyddataen er gemt, og hvordan den skal tolkes. Den sidste underblok er *Data*-underblokken, den indeholder bare selve lyddataen.

Ud fra den information virker det ligetil at læse resten af filen. Man kunne fortsætte med præcis det samme som der er gjort indtil videre: Læs værdi <-> Oversæt værdi & verificer værdi. Problemet med denne tilgang er er at RIFF er et format som kan udvides på, og støtter at tilføje ekstra specialiserede underblokke, som nogle lydprogrammer måske benytter sig af. Den betydning det har for implementeringen er at i nogle lydfiler kan der f.eks. være mere end 2 underblokke, og de underblokke kan både være før, mellem og efter format- og datablokkene. Det er også *lovligt* ifølge RIFF at bytte om på rækkefølgen af underblokke[24].

```
/// <summary>
/// Beskriver det format lyden er gemt i.
/// </summary>
[StructLayout(LayoutKind.Explicit, Size = 24)]
public readonly struct FormatSubchunk
{
    /// <summary>
    /// Indeholder altid teksten "fmt ".
    /// Big-endian ASCII streng.
    /// </summary>
    [FieldOffset(0)]
    public readonly uint ID;
    /// <summary>
    /// Størrelse på Format subchunk.
    /// </summary>
    [FieldOffset(4)]
    public readonly uint Size;
    /// <summary>
    /// Er altid 1.
    /// </summary>
    [FieldOffset(8)]
    public readonly ushort AudioFormat;
```

Figur 20: Strukturen for at gemme på format-blokken(Aud.IO.Formats.FormatSubchunk)

[24] (Library of Congress, 2017, RIFF-formatet)

For at kunne understøtte flere versioner af wave-filer, er programmet struktureret sådan at det læser underblokkene, blok-vis. Af den årsag er hver underblok sin egen struktur, som indeholder blokken felter. På figur 20 bruges attributter for hver feltvariabel for at fortælle compileren, hvordan den skal organisere den data i hukommelsen, så den er struktureret på samme måde som i filen.

```
FormatSubchunk? formatSubchunk = null;
DataSubchunk? dataSubchunk = null;

// Læs subchunks indtil filen er færdig-læst.
while (stream.Length != stream.Position)
{
    // Aflæs subchunk ID
    byte[] subchunkIDBytes = new byte[sizeof(int)];
    byte[] subchunkSizeBytes = new byte[sizeof(uint)];
    stream.Read(subchunkIDBytes, 0, subchunkIDBytes.Length);
    stream.Read(subchunkSizeBytes, 0, subchunkSizeBytes.Length);

    string subchunkID = Encoding.ASCII.GetString(subchunkIDBytes);
    uint subchunkSize = BitConverter.ToUInt32(subchunkSizeBytes, 0);

    // Behandl subchunk baseret på dens ID.
    switch (subchunkID)
    {
```

Figur 21: Næste del af constructor koden i Aud.IO.Formats.WaveFile

Øverst på figur 21 bliver to variable, som svarer til de to underblok-typer, deklareret som *nullable*-værdityper. Det betyder egentlig bare at de kan have værdien *ingenting*, som værdi-typer normalt ikke kan have. Den egenskab bliver brugt senere i koden.

På figuren er der også en løkke, som bliver ved med at iterere indtil slutningen af filen er nået. Inde i løkken bliver de felter som er ens for underblokke — det er ID'et og størrelsen — læst og behandlet. Så bliver ID'et matched i en switch statement.

```
case "fmt ":
    byte[] formatSubchunkBytes = new byte[sizeof(ushort) * 4 + sizeof(uint) * 2];
    stream.Read(formatSubchunkBytes, 0, formatSubchunkBytes.Length);

    formatSubchunk = new FormatSubchunk(
        BitConverter.ToUInt16(formatSubchunkBytes, 0),
        BitConverter.ToUInt16(formatSubchunkBytes, 2),
        BitConverter.ToUInt32(formatSubchunkBytes, 4),
        BitConverter.ToUInt32(formatSubchunkBytes, 8),
        BitConverter.ToUInt16(formatSubchunkBytes, 12),
        BitConverter.ToUInt16(formatSubchunkBytes, 14));

    // Hvis der er ukendte ekstra parametre (ifl. standarden er det muligt)
    if (formatSubchunkBytes.Length < subchunkSize)
    {
        // Spring over dem.
        stream.Seek(subchunkSize - formatSubchunkBytes.Length, SeekOrigin.Current);
    }
    break;
case "data":
    byte[] dataSubchunkBytes = new byte[subchunkSize];
    stream.Read(dataSubchunkBytes, 0, dataSubchunkBytes.Length);

    dataSubchunk = new DataSubchunk(dataSubchunkBytes);
    break;
default:
    // ukendt subchunk, ignorer den.
    stream.Seek(subchunkSize, SeekOrigin.Current);
    break;
```

Figur 22: Switch statement over blok ID

I switch statementet på figur 22 er der en case for hver underblok. I hver case bliver hele blokken læst (ud fra størrelsen læst tidligere). I format-blokken bliver de læste bytes læst over i en ny instans af FormatSubchunk, og den tildeles den variabel, der blev deklareret tidligere. Der er en ekstra detalje i standarden omkring at den støtter ekstra parametre i format-blokken — de bliver ignoreret, hvis der er nogle. For data-blokken gælder det samme, forskellen er bare at data-blokken kun har 1 felt tilbage. Det felt læses og det gemmes i en ny instans af *DataSubchunk*, og blokken støtter ikke ekstra parametre, så den kræver ikke andet hensyntagen end det.

I tilfældet af at blok ID'et er ukendt, ender switch statementet i default-casen, som springer over det antal bytes som størrelsen af den ukendte blok er.

```
if (formatSubchunk is null)
{
    throw new MissingSubchunkException("fmt ");
}

if (dataSubchunk is null)
{
    throw new MissingSubchunkException("data");
}

waveData = new WaveStructure(formatSubchunk.Value, dataSubchunk.Value);
```

Figur 23: Den sidste del af constructoren til Aud.IO.Formats.WaveFile

På figur 23 er den sidste del af constructoren, her bliver der tjekket at de to blokke har været læst. Det sker ved at bruge egenskaben at de kan være null, da hvis de to variabler stadig er null, når den har læst hele filen igennem, betyder det at de mangler. Hvis de mangler, bliver der kastet en exception som beskriver, hvilken blok der mangler. Endeligt bliver de to underblokke gemt i en ny *WaveStructure*, som er den struktur, der indeholder headeren og underblokkene. Til sidst forlader den *using*s scope, så den åbne filstream bliver disposed og lukket.

Der er lavet en delapplikation som demonstration og af testning af den data, der bliver læst. Den kaldes *AudioMetadataViewer*.

```
WaveFile audioFile = new WaveFile(args[0]);

WaveStructure data = audioFile.GetWaveData();

StringBuilder sb = new StringBuilder();
sb.AppendLine($"Chunk ID: {Encoding.ASCII.GetString(BitConverter.GetBytes(data.ChunkID))}");
sb.AppendLine($"Chunk størrelse: {data.ChunkSize}");
sb.AppendLine($"Format: {data.Format}");
sb.AppendLine("===");
sb.AppendLine($"Subchunk ID: {Encoding.ASCII.GetString(BitConverter.GetBytes(data.Subchunk1.ID))}");
sb.AppendLine($"Størrelse: {data.Subchunk1.Size} bytes");
sb.AppendLine($"Lyd format: {data.Subchunk1.AudioFormat}");
sb.AppendLine($"Antal lydkanaler (mono/stereo): {data.Subchunk1.NumChannels}");
sb.AppendLine();
sb.AppendLine($"Indsamlingsfrekvens: {data.Subchunk1.SampleRate} Hz");
sb.AppendLine($"Data punkt størrelse: {data.Subchunk1.BitsPerSample} bits");
sb.AppendLine($"Blockalign, data punkt størrelse: {data.Subchunk1.BlockAlign}");
sb.AppendLine($"Bytes i sekundet(?): {data.Subchunk1.ByteRate}");
sb.AppendLine("---");
sb.AppendLine($"Subchunk ID: {Encoding.ASCII.GetString(BitConverter.GetBytes(data.Subchunk2.ID))}");
sb.AppendLine($"Størrelse: {data.Subchunk2.Size} bytes");

Console.WriteLine(sb.ToString());
```

Figur 24: Entry-metoden i AudioMetadataViewer

På figur 24 bruges den API, der blev dokumenteret ovenfor, til at tilgå de forskellige aflæste værdier fra formatblokken. Den skriver alle variablernes værdi i en stringbuilder og skriver det til konsollens vindue. Den blev brugt til at teste om værdierne blev læst korrekt, under udviklingen af programmet.

```
[TestMethod]
[DeploymentItem(@"lydfil\440 frekvens 441 samplerate sinus ny.wav")]
public void TestWaveFileHeader()
{
    WaveFile audioFile = null;
    try
    {
        audioFile = new WaveFile(@"lydfil\440 frekvens 441 samplerate sinus ny.wav");
    }
    catch (System.Exception)
    {
        Assert.Fail();
    }

    WaveStructure data = audioFile.GetWaveData();

    // Har samplerate på 44100
    Assert.AreEqual<uint>(44100, audioFile.SampleRate);
    // Der er 88200 samples i filen.
    Assert.AreEqual<int>(88200, audioFile.Samples);
    // Mono
    Assert.AreEqual<ushort>(1, data.Subchunk1.NumChannels);
    // 16 bits per sample.
    Assert.AreEqual<ushort>(16, data.Subchunk1.BitsPerSample);
}
```

Figur 25: Unit tests af WaveFile, i Aud.IO.Tests

Udover demoprogrammet ovenfor, blev der også udviklet nogle tests løbende, for at være sikker på at den stadig indlæste lydfiler korrekt efter ændringer på den, disse tests er på figur 25.

Indtil videre bliver lyddataen i filen, som læses, blot betragtet som et array af bytes, men der er nogle skridt fra den er på byte form til den er på et brugbart format. Givet et analogt lydsignal, som repræsenterer lydstyrke mellem -1 og 1, bliver det moduleret til et digitalt signal for at kunne gemmes digitalt. For wave bruges den moduleringsmetode, som kaldes LPCM, eller *lineær pulserende-kode modulering*.

```
short[] modulatedAudio = new short[audio.Length];

double linearScalingFactor = Math.Pow(2, BitsPerSample) / 2 - 1;
for (int i = 0; i < audio.Length; i++)
{
    modulatedAudio[i] = (short)Math.Round(audio[i] * linearScalingFactor);
}

byte[] audioBytes = new byte[modulatedAudio.Length * (BitsPerSample / 8)];
Buffer.BlockCopy(modulatedAudio, 0, audioBytes, 0, audioBytes.Length);
```

Figur 26: Billede af metoden SetDemodulatedAudio fra WaveFile

I kode udsnittet på figur 26 er implementeret LPCM, for analog til digital. Givet et array med decimaltal (af typen *double*) kaldet *audio*, omregner koden signalet ved hjælp af LPCM. Først beregnes den lineære skaleringsfaktor til at være $2^{BitsPerSample}/2 - 1$, hvor *BitsPerSample* er en af header felterne, som typisk indeholder et tal mellem 8 og 16 (byte og short), der svarer til størrelsen af opløsningen i bits. Den maksimale værdi den kan repræsentere kan dermed beregnes ved at bruge formlen. Grunden til divisionen med 2, er at tallet gemmes i *signed* format, og det svarer til halvdelen af størrelsen subtraheret 1 .

Alle værdierne i *audio* bliver dermed skaleret med faktoren, og er nu gemt med heltid i stedet for decimaltal. Til sidst bliver alle værdierne flyttet over i et ny array med en passende størrelse, ved hjælp af blockcopy. Det er den samme proces når det skal demoduleres, der bliver værdien bare divideret med skaleringsfaktoren[25].

Programdokumentation

I dette underafsnit, vil programmet blive dokumenteret ved hjælp af diverse metoder for dokumentation indenfor programmering. Der vil blive benyttet underafsnit da det er en systematisk tilgang til dokumentationen. Under udarbejdelsen af programmet er programmet løbende blevet gemt i et Git-filsystem, for at holde styr på de ændringer, som er blevet lavet. Al den kode, som er udarbejdet for dette projekt, er vedhæftet som figur 27.

[25] Teknisk set, er det en divisor i dette tilfælde, men i koden kaldes begge variable faktorer.

Mappeplacering: `Fourier Transformation\`

Figur 27: Visual Studio 2021 solution over projektet *Fourier Transformation*. Vedhæftet som figur, siden bilag ikke indgår i bedømmelsen.

Kravspecifikation

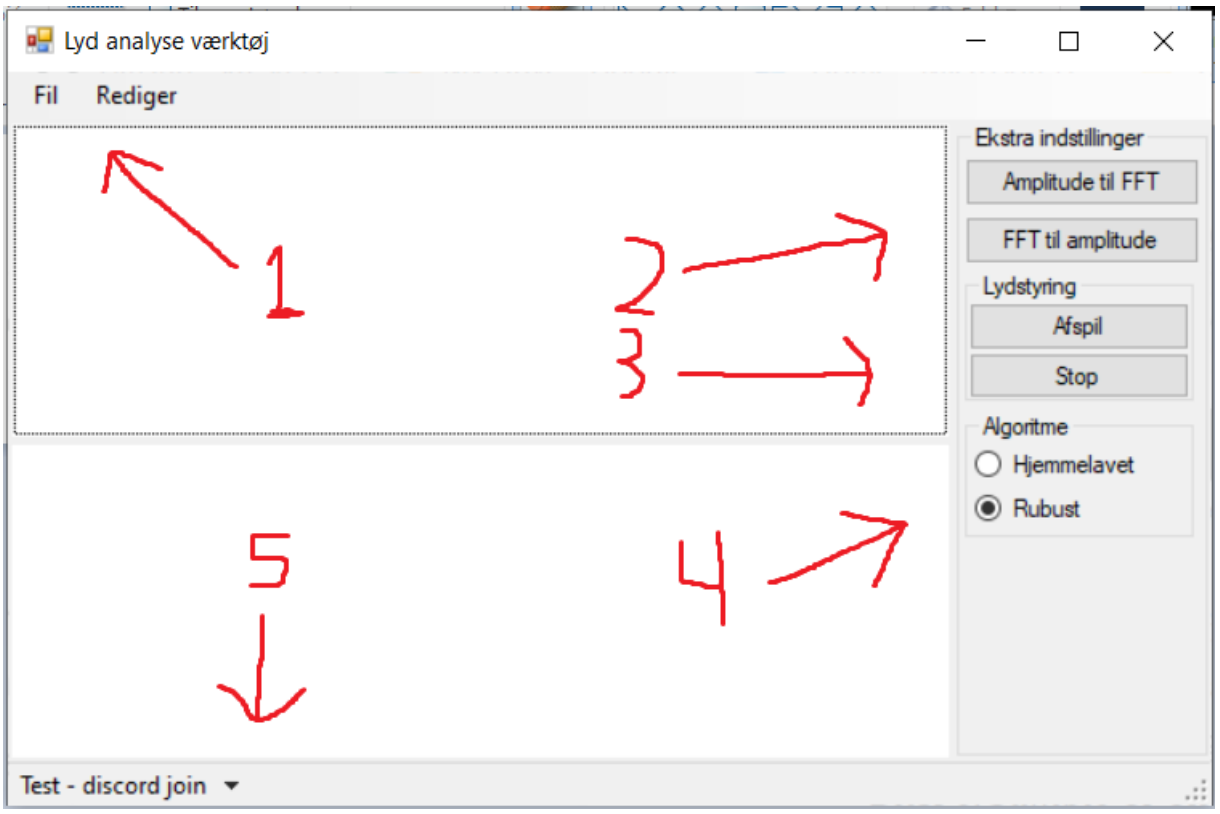
Der blev opstillet følgende krav:

- Man skal kunne åbne lydfiler
- Man skal kunne lave spektralanalyse af dem
    - Man skal kunne gøre det for dele af grafen
- Programmet skal kunne gøre dette uden at loade i for lang tid.
- Man skal kunne gemme lydfilerne igen.
- Man skal kunne filtrere frekvenser ud.

Alle kravene endte med at blive implementeret, og vil blive dokumenteret i følgende afsnit. Derudover var der feature creep, da der blev introduceret flere funktioner end kravene beskrev.

Funktionalitet og brugergrænseflade

Her vil kort blive redegjort for programmets funktionalitet og hvordan man tager dem i brug. Vedhæftet som bilag (bilag 1, 2, 3 og 4) er forskellige lydfiler, som er vedhæftet og programmet kan testes på.



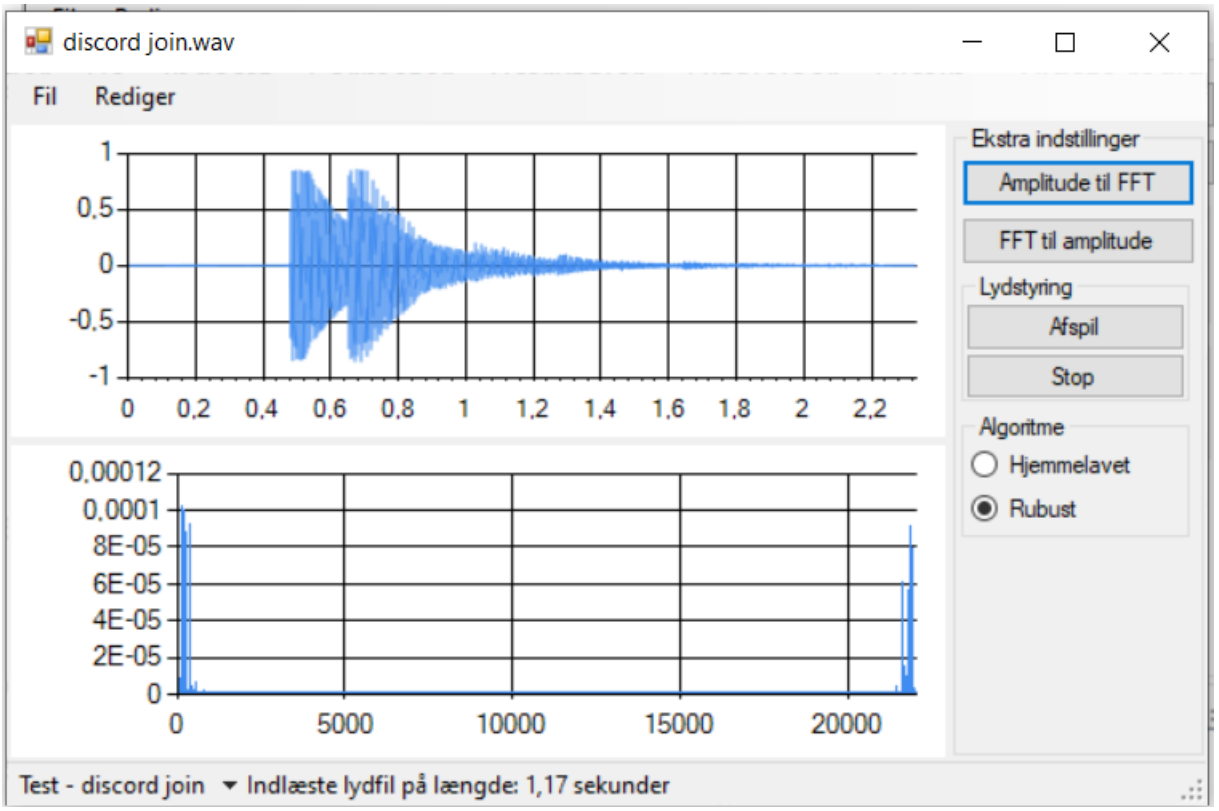
Figur 28: Skærmbillede af forside af applikationen uden at nogen lydfil er blevet indlæst

Figur 28 er et skærmbillede med markeringer på for bedre at kunne henvise til de forskellige knapper og dippedutter. Den første markering henviser til den MenuStrip, som er indsat i programmet. Den indeholder to menuer, fil og rediger, hvilket er standard betegnelser for det de indeholder. *Fil* indeholder *Åben lydfil…*, *Gem*, *Gem som…*, og *Luk*. De er selvforklarende, og alle knapperne bør virke. Grunden til at nogle af bogstaverne i navnene er understreget, er for at dokumentere at de har det som kaldes *Access keys*, hvilket er en standard forkortelse til at klikke på ting. Hvis man holder *Alt*-knappen nede på samme tid med at klikker på det bogstav som er understreget, trykkes der på knappen.

Den anden, tredje og fjerde markering, henviser til kontrolpanelet ude i siden, den øverste knap, er den som tager tidsdomæne grafen (øverste rude) og beregner FFT grafen for den. Den nedenunder er til at lave den tilbage igen, efter man har lavet ændringer i den.

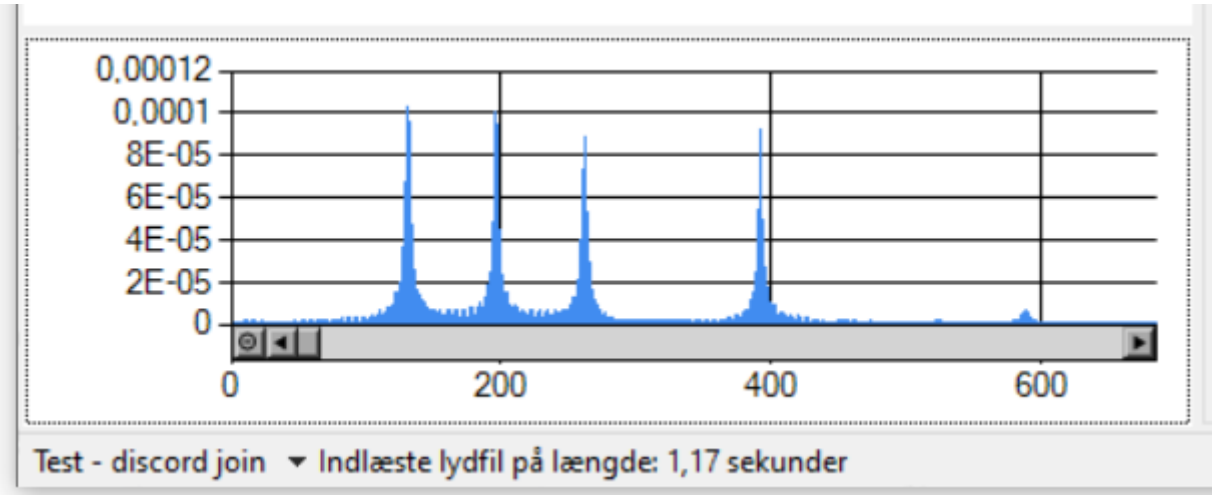
Den tredje markering er selvforklarende. Den fjerde markering lader brugeren vælge, hvilken implementering af algoritme de vil bruge. *Hjemmelavet* er den implementering

der blev udarbejdet i afsnittet før dette, *robust* er en implementering, som stammer fra MathNet biblioteket.
Den femte markering viser StatusStrip. Men når man foretager en handling, vil der typisk dukke noget tekst op her, som forklarer hvad der er sket for at give bedre feedback.



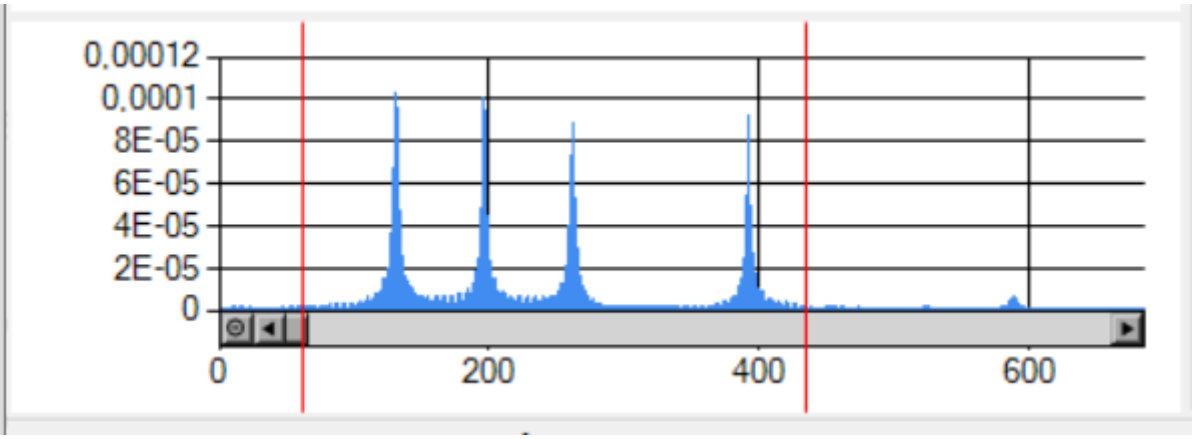
Figur 29: Skærmbillede af en lydfil, som er indlæst

Nu er der blevet indlæst en lydfil, og de to grafer, som før var blanke, er nu fyldt ud (figur 29). De skridt, som er taget for at nå her, er brug af Fil menuen -> åben fil, lydfilen er åbnet, og knappen *Amplitude til FFT* er trykket for at danne den nederste graf. Ser man på StatusStrip, kan man se den har opdateret, men der er en uoverensstemmelse, teksten siger at en lydfil på længden *1,17 sekunder* er indlæst, mens tidsgrafen (øverste rude) går helt til *2,2 sekunder*. Dette skyldes at biblioteket, delvist dokumenteret først i afsnittet, ikke støtter stereolyd. I wave-filer er stereolyd gemt som ekstra bytes lige ved siden af der hvor mono ville være, så hver anden tilhører venstre, mens de andre tilhører højre højttaler.

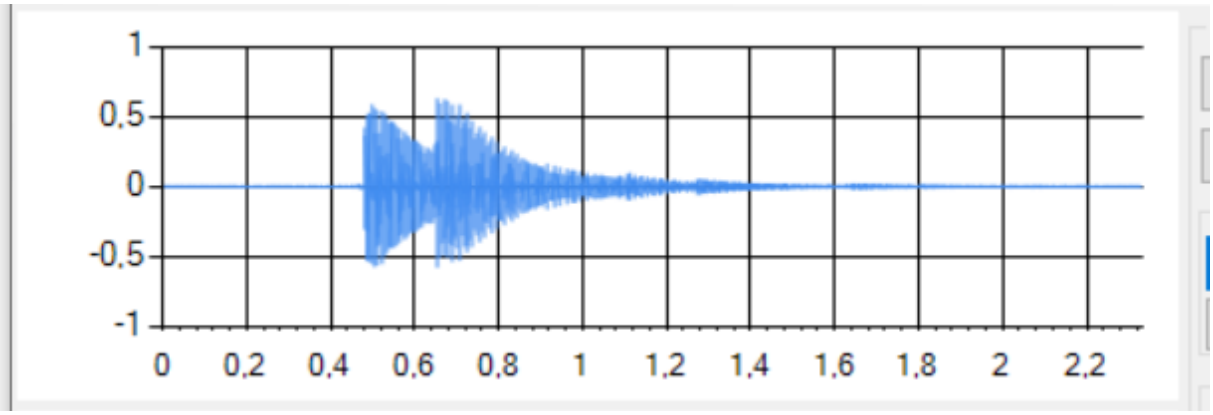


Figur 30: Zoomet ind på grafen

Hvis man nu kigger på spektralanalysegrafen (nederste rude) og begynder at markere for at zoome ind. Man bruger venstreklik til at zoom ind. Figur 30 er zoomet ind på de lavfrekvente lyde af den originale lyd. Til venstre for scrollbaren, er en lille knap, den zoomer ud. Man kan nu markere det man vil filtrere ud med højreklik -> træk.

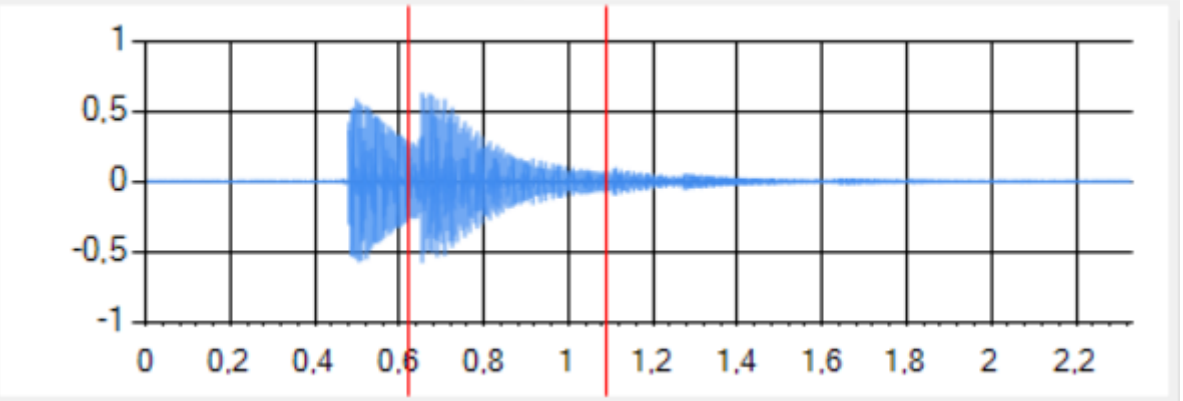


Figur 31: Højreklik - markering



Figur 32: Den nye lydgraf

Figur 31 viser en sådan markering, ved at trykke *delete* (på tastaturet) ændrer man nu alle de værdier til 0. Derefter kan man klikke på knappen *FFT til amplitude* for at se den ændrede lyd graf, og trykke afspil for at høre den. Så kan man gemme lydfilen i Filmenuen.



Figur 33: Markering af lydgrafen

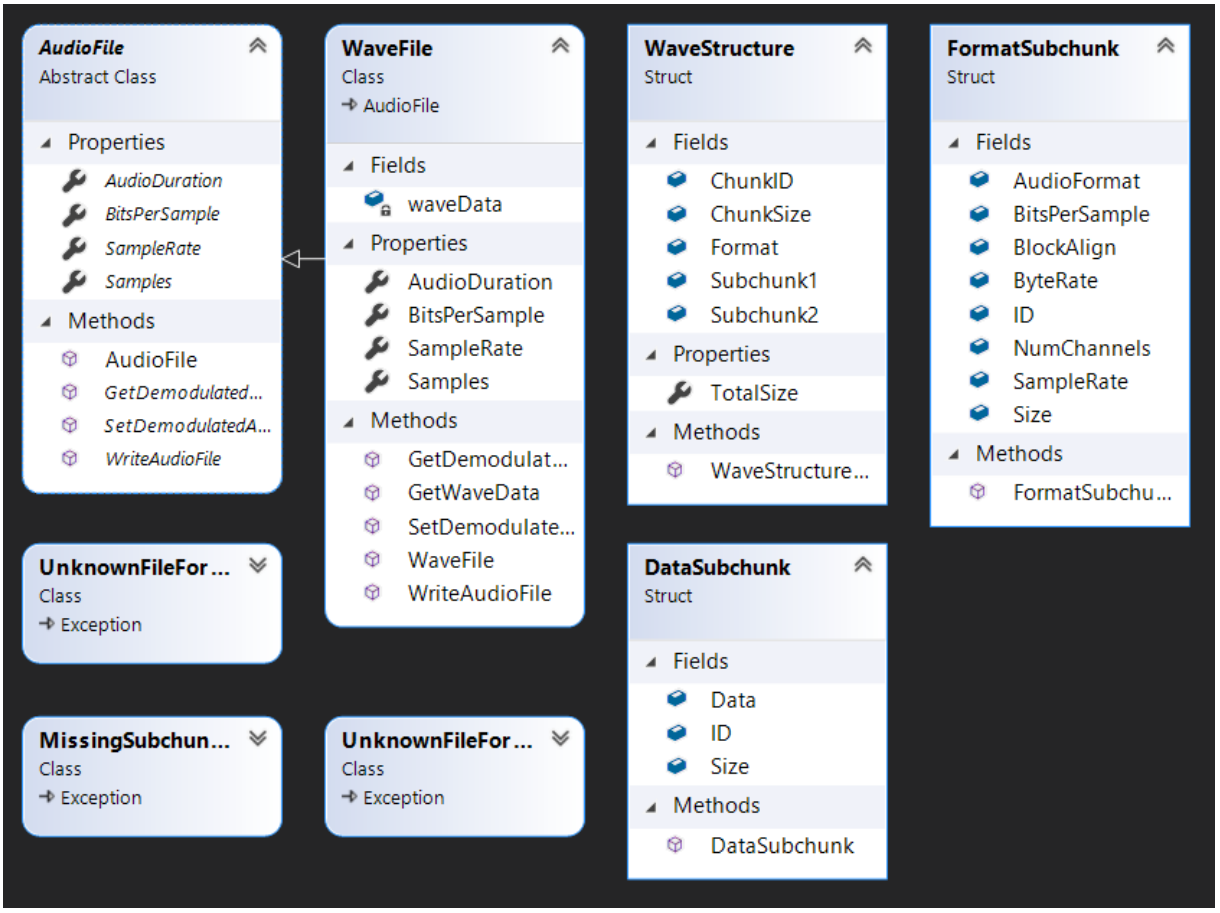
Det er også muligt at markere på lydgrafen. Gør man dette vil knappen *Amplitude til FFT* kun lave en spektralanalyse for det markerede stykke lyd (Figur 33).

Design af projektet og klasser

Programmet består af flere forskellige projekter, som hver har deres formål.

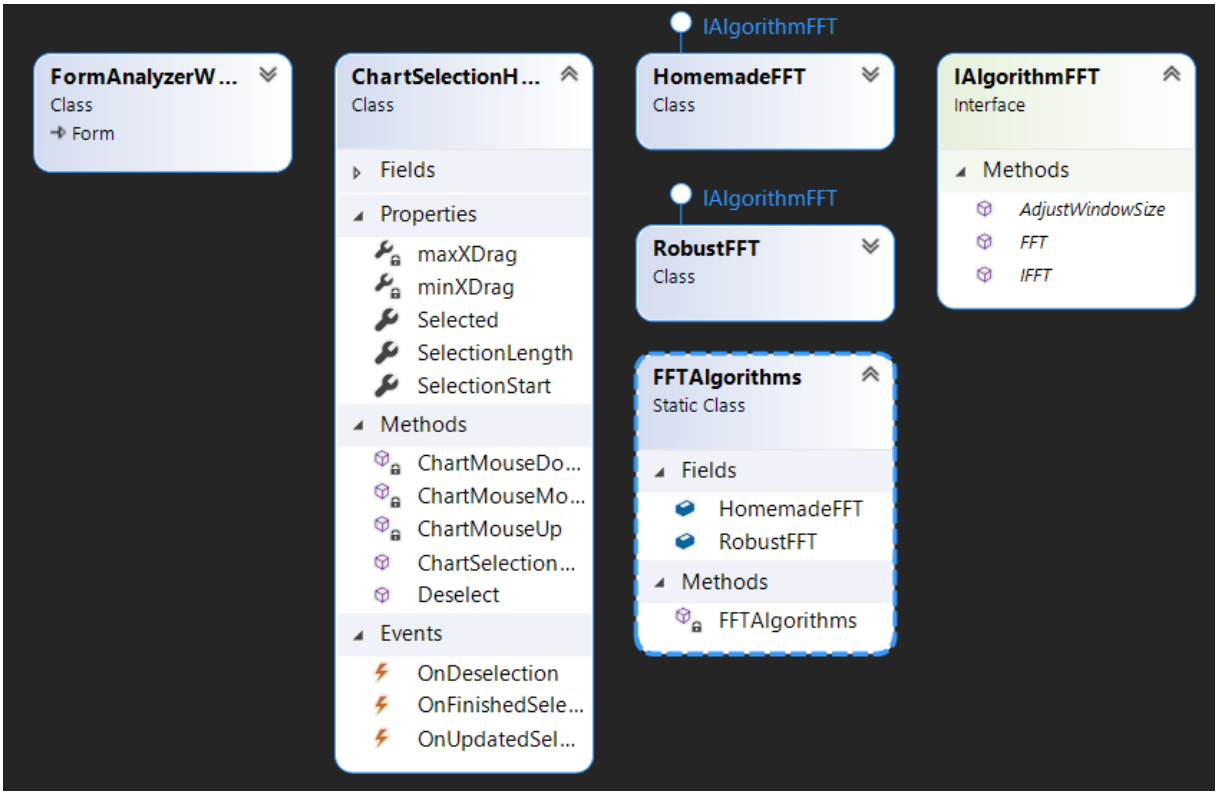
- AudioAnalyzer - C# Winforms

    - Dette er programmet primære brugergrænseflade, det er et Windows forms projekt som interagerer med de andre projekter.
- AudioMetadataViewer - C# Konsol applikation
    - Er en konsol brugergrænseflade til at vise den metadata, som bliver læst fra en lydfil.
- Aud.IO - C# Biblioteksmodul
    - Alle gode biblioteker er navngivet efter ordspil. Dette bibliotek dækker indlæsning (deserialization), modificering, og gemning af lydfiler (serialization). Det er det modul som blev dokumenteret først i afsnittet.
- Aud.IO.Tests - C# MS testbibliotek
    - Indeholder tests af *Aud.IO* biblioteksmodulet.
- Aud.IO.Algorithms - F# Biblioteksmodul
    - Dette biblioteksmodul er skrevet i sproget F#, da det giver mest mening når man arbejder med rekursion og matematik. Den indeholder blot klassen *CooleyTukey*, og implementeringen af den algoritme.
- Aud.IO.Algorithms.Tests - C# MS testbibliotek
    - Dette testbibliotek indeholder tests der blev lavet under udarbejdelsen af *Aud.IO.Algorithms*.



Figur 34: Klasse diagram over Aud.IO, bokse med kantede hjørner er structs

Klassediagrammet på figur 34 viser de forskellige klasser og strukturer. I venstre side, er de generelle elementer af biblioteket. Det er blandt andet *AudioFile* og de forskellige typer exceptions. De ville kunne bruges, havde man implementeret flere filformater. Ved siden af er klassen som står for at indlæse og gemme Wave-filer, den nedarver fra *AudioFile* og implementerer dens abstrakte metoder, (heriblandt Set/*GetDemodulatedAudio* og *WriteAudioFile*). I højre side er de strukturer som bruges internt i *WaveFile* til at repræsentere lydfilen i hukommelsen. *WaveStructure* indeholder instanser af underdelene og information om formatet.



Figur 35: Klasse diagram over AudioAnalyzer

Klassediagrammet på figur 35 er en oversigt over hjælpeklasser, og strukturering af WinForms applikationen. Længst til venstre er selve formsklassen, den indeholder alle de elementer, som indgår i brugergrænsefladen. Ved siden af er *ChartSelectionHelper*, som er en hjælpeklasse, som følger decorator-design mønsteret, som går ud på at udvide en eksisterende klasse og undgå unødvendig nedarvning. Det den gør er at udvide winforms elementet *Chart* så den støtter sekundær markering, (det er højreklik markeringen), den har nogle events som bliver kaldt ved ændringer i markeringen. Længst til højre på klassediagrammet er algoritmesystemet, der er et interface som har de metoder man forventer af en FFT-algoritme. Den statiske hjælpeklasse *FTFAlgorithms* instantiere de to klasser, som nedarver fra *IAlgorithmFFT*.

```
private void PlayAudio()
{
    if (editedWaveFile is null)
    {
        UpdateStatusStrip("Ingen lyd at afspille");
        return;
    }

    // Write current audio represented in editedWaveFile.
    string tempFile = Path.GetTempFileName();
    editedWaveFile.WriteAudioFile(tempFile);

    // Load temporary file into memory to delete it afterwards.
    MemoryStream waveFile = new MemoryStream(File.ReadAllBytes(tempFile));
    File.Delete(tempFile);

    // Play the audio file from memory.
    soundPlayer = new SoundPlayer(waveFile);
    soundPlayer.Play();
}
```

FIgur 36: Metoden, der kaldes når 'afspil lyd' knappen trykkes

For at udarbejde et eksempel på pseudokode, tages der udgangspunkt i koden til at afspille lyd på figur 36.

Hvis lyd ikke er indlæst
  Returnér

Kald API for at få tilfældig genereret filnavn placeret i temp mappen
Kald Aud.IO API med det generede filnavn for midlertidigt at gemme lydfilen

Læs fil ind i hukommelsen
Slet filen

Afspil lyden ved hjælp af API med filen fra hukommelsen

Figur 37: Pseudokode over metoden fra figur 36

Selvom pseudokoden (figur 37) tager udgangspunkt i API tung kode, viser det stadig den proces den går igennem for at afspille lyden.

Det udarbejdede program er hermed dokumenteret ved hjælp af forskellige metodiske tilgange fra programmingsfaget.

Konklusion

I opgaven blev følgende problemstillinger bragt op, først skulle der redegøres for komplekse tal og fourier transformation, samt deres sammenhæng. Der blev blandt andet kigget på den forskel der er i store-O notationen for diskret fourier transformering og hurtig fourier transformering. Derudover blev der set på hvordan komplekse tal er med til at afkorte antallet af beregninger en computer skal lave, fremfor hvis man brugte vektorer.

Derudover blev der udarbejdet en implementering af Cooley—Tukey algoritmen. Her blev både det teoretiske bag algoritmen beskrevet rent matematisk, og så blev algoritmen dokumenteret, ved brug af flowcharts. Implementeringen blev demonstreret ved brug af det endelige program, og den endte med at fungere fint i praksis.

Det endelige program blev også udviklet på baggrund af de krav, der blev opstillet og den endte med at følge kravene. Udover kravene, har der været feature creep siden, der er blevet implementeret flere funktioner end kravspecifikationen beskrev. Det udarbejdede program blev et forholdsvist brugervenligt program, som endte med at være i stand til at behandle, filtere og indlæse lydfiler.

Litteraturliste

Drowell, E. (n.d.). *Big-O Complexity Chart*. Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. Læst december 2, 2021, fra https://www.bigocheatsheet.com/

Let's Clear Up Some Things About FFT…. (2017, februar 21). NTi Audio. Læst december 3, 2021, fra https://www.nti-audio.com/en/news/lets-clear-up-some-things-about-fft-part-1

Library of Congress. (n.d.). *WAVE Audio File Format*. Library of Congress. Læst december 17, 2021, fra https://www.loc.gov/preservation/digital/formats/fdd/fdd000001.shtml?loclr=blogsig

Library of Congress. (2017, februar 17). *RIFF (Resource Interchange File Format)*. Library of Congress. Læst december 17, 2021, fra https://www.loc.gov/preservation/digital/formats/fdd/fdd000025.shtml

lukicdarkoo. (2015). *Basic implementation of Cooley-Tukey FFT algorithm in Python*. gists - GitHub. Læst december 20, 2021, fra https://gist.github.com/lukicdarkoo/1ab6a9a7b24025cb428a

McLean, J. L. (Director). (2020). *Repetition 12* [Film]. YouTube. https://www.youtube.com/watch?v=O4EoX6Z2hrM

McLean, J. L. (Director). (2021, oktober 1.). Komplekse tal lektion 2 ved Absalon ingeniøruddannelse (Sæson Komplekse tal, Episode Lektion 2) [TV series episode]. Fra *Absalon ingeniøruddannelse*. YouTube. https://www.youtube.com/watch?v=TUPQSowrmDI

McLean, J. L. (Director). (2021, okt 1). Komplekse tal lektion 3 ved Absalon ingeniøruddannelse (Sæson Komplekse tal, Episode Lektion 3) [TV series episode]. Fra *Absalon ingeniøruddannelse*. YouTube. https://www.youtube.com/watch?v=jscO0YJwWOY

NTNU. (n.d.). *Forventning tolkning*. Institutt for matematisk fag. Læst december 18, 2021, fra https://wiki.math.ntnu.no/_media/tma4245/tema/begreper/forventningtolkning.svg?w=600&tok=19cde4

Sapp, C. (n.d.). *Microsoft WAVE soundfile format*. Soundfile. Læst december 2, 2021, fra http://soundfile.sapp.org/doc/WaveFormat/

3Blue1Brown (Director). (2018). *But what is the Fourier Transform? A visual introduction.* [Men hvad er Fourier Transformation? En visuel introduktion] [Film]. https://www.youtube.com/watch?v=spUNpyF58BY

webmatematik.dk. (2012). Polært koordinatsystem. *webmatematik*, *Andre koordinatsystemer*(Særligt for HTX). https://www.webmatematik.dk/lektioner/saerligt-for-htx/andre-koordinatsystemer/polaert-koordinatsystem

webmatematik.dk. (2012). Summationstegn. *webmatematik.dk*, *Statistik*(Matematik B). https://www.webmatematik.dk/lektioner/matematik-b/statistik/summationstegn

Wikipedia (Ed.). (n.d.). Eulers formel. *Dansk Wikipedia*. Wikipedia. Læst december 19, 2021, fra https://da.wikipedia.org/wiki/Eulers_formel

Xu, S. (Director). (2015). *The FFT Algorithm - Simple Step by Step* [FFT algoritmen - enkelt ét skridt ad gangen] [Film]. YouTube. https://www.youtube.com/watch?v=htCj9exbGo0

0xb01u. (2021, august 20). *pyFFT*. GitHub. Læst december 20, 2021, fra https://github.com/0xb01u/pyFFT/blob/master/Cooley-Tukey.py

Bilag

Bilag 1: Stereolyd, lydfil af den ikoniske lyd discord laver når en bruger deltager i et opkald.

- Filplacering: `Bilag\discord join.wav`

Bilag 2: Monolyd, lydfil over 250 Hz sinuskurve, som ligger halvvejs hen over 441 Hz sinuskurve.

- Filplacering: `Bilag\440 og 250 frekvens 441 samplerate.wav`

Bilag 3: Monolyd, 8 sekunder langt musikstykke, ved lav sampling rate.

- Filplacering: `Bilag\Musikinstrumenter.wav`

Bilag 4: Monolyd, ren lydfil på 441 Hz sinuskurve.

- Filplacering: `Bilag\440 frekvens 441 samplerate sinus.wav`