Fast Fourier Transform

**Structuur (+ wat op de dia’s komt):**

**Dia 1: Titelblad**

Fast Fourier Transform (FFT)

Teamnaam

Namen

**Dia 2: Inleiding**

James Cooley & John Tukey (+ Fig 1)

DFT = traag!

Oplossing: FFT = snel!

Grafiek dia 4 (in klein) -> Erbij zeggen dat dit later wordt besproken

Toepassingen: noise cancelling, complexe DVGL … (\ldots)

**Dia 3: Waarom werkt de DFT zo traag en FFT zoveel sneller?**

Tabel met langs de ene kant DFT en de andere kant FFT

**DFT**  
formules: *Xk*=∑*n*=0*N*−1*xn*⋅*e*−*i* 2*π* *k* *n* / *N*   
 *xn*=1*N*∑*k*=0*N*−1*Xkei* 2*π* *k* *n* / *N*  
efficiëntie: O(N^2)  
vermenigvuldigingen: N^2  
optellingen: N(N-1)

**FFT**  
formules: …  
efficiëntie: O(N\*log\_2 ⁡N)  
vermenigvuldigingen: N/2\*log\_2 N  
optellingen: N\*log\_2 N

**Dia 4: Complexiteit van DFT vs FFT (tabel)**

Fig 2

**Dia 5: Complexiteit van DFT vs FFT (grafiek)**

Fig 3

**Dia 6: Hoe wordt dit opgelost in de FFT?**

Hoe O(\*N log\_2 ⁡ N ) verkrijgen?

Symmetrie cos en sin

**Dia 7: werking FFT**

**Dia 8: werking FFT**

**Dia 9: Het vlinderdiagram**

**Dia 10: De Python-code van Robbe**

**Dia 11: BRONVERMELDING !!!**

**???** Bronvermelding van gebruikte grafieken, tabellen, afbeeldingen nodig **???**

@misc{wikiFFT,

title={Fast Fourier transform},

URL= \\\texttt{https://nl.wikipedia.org/wiki/Fast\_Fourier\_transform},

datum={geraadpleegd op 30-11-2020}

}

@misc{wikiDFT,

title={Discrete fouriertransformatie},

URL= \\\texttt{https://nl.wikipedia.org/wiki/Discrete\_fouriertransformatie},

datum={geraadpleegd op 30-11-2020}

}

@misc{understandingAlgorithm,

title={Understanding the FFT Algorithm},

URL= \\\texttt{https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/},

datum={geraadpleegd op 23-11-2020}

}

@misc{uitlegFormulesDFT,

title={Discrete Fourier Transform (DFT)},

URL= \\\texttt{http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/dft.html},

datum={geraadpleegd op 04-12-2020}

}

@misc{uitlegFormulesFFT,

title={Fast Fourier Transform (FFT)},

URL= \\\texttt{http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html},

datum={geraadpleegd op 04-12-2020}

}

@misc{vlinderdiagram,

title={What is a Fast Fourier Transform (FFT)? The Cooley-Tukey Algorithm},

URL= \\\texttt{What is a Fast Fourier Transform (FFT)? The Cooley-Tukey Algorithm},

datum={geraadpleegd op 04-12-2020}

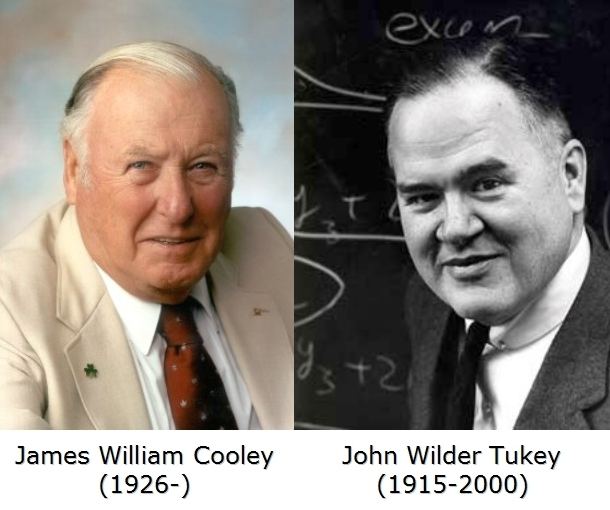
}

**Tekst:**

**Dia 1: Titelblad: Fast Fourier Transform (FFT)**

**Dia 2: Inleiding**

De Fast Fourier Transformatie, afgekort als FFT, is een algoritme die we te danken hebben aan James Cooley en John Tukey. De Discrete Fourier Transformatie die we onderzochten is een zeer trage methode om de Fourierontwikkeling te bestuderen. We gebruiken deze om bijvoorbeeld geluidsbestanden te comprimeren. In het dagelijkse leven is het natuurlijk niet zo praktisch dat dit zo traag werkt, daarom werd de FFT ontworpen die hetzelfde doet als de Discrete Fourier Transformatie maar in een veel kortere tijd. Doordat deze sneller werkt, heb je dus ook meer mogelijkheden. Want bij de DFT zal je sommige zaken niet kunnen oplossen omdat het gewoon té lang duurt. Zo kan je met de FFT bijvoorbeeld zeer complexe differentiaalvergelijkingen oplossen. Ook ruisonderdrukking (ook wel gekend als noise cancelling) wordt hiermee mogelijk gemaakt. (bv. In koptelefoons moet dat snel gebeuren.) Zelfs bijvoorbeeld gps-signalen en internet-verbinding rusten op het FFT-principe!

 Fig 1: Foto van James Cooley (links) en John Tukey (rechts)

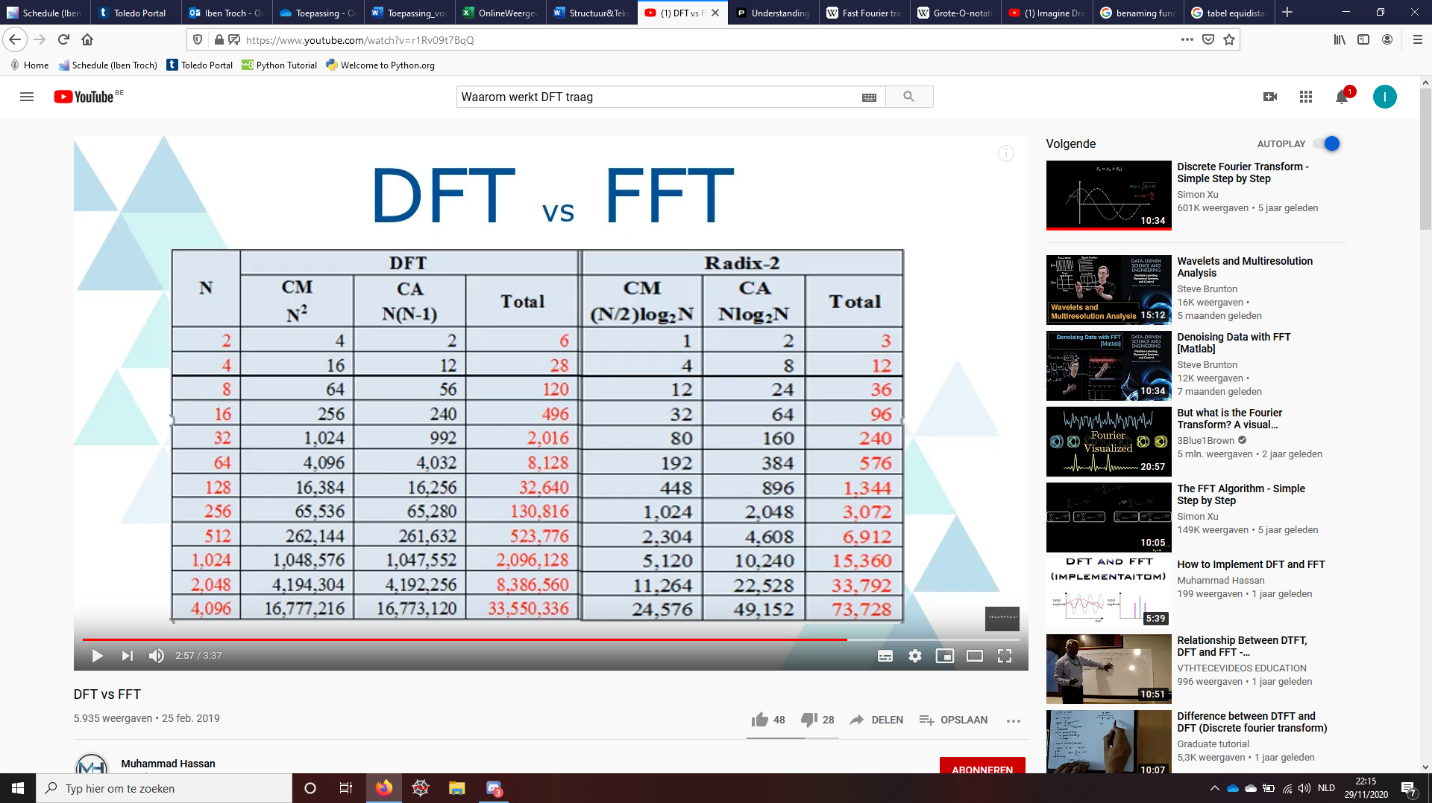
**Dia 3: Waarom werkt de DFT zo traag en FFT zoveel sneller?**

De DFT wordt uitgevoerd aan de hand van N equidistante punten via de formules *Xk*=∑*n*=0*N*−1*xn*⋅*e*−*i* 2*π* *k* *n* / *N en xn*=1*N*∑*k*=0*N*−1*Xkei* 2*π* *k* *n* / *N.* Deze methode werkt heel erg traag omdat er een efficiëntie is van O(N^2). Dit noemen we de grote-O-notatie en geeft een benadering van de tijdscomplexiteit van een algoritme. Zoals we allemaal weten stijgt de grafiek van de grafiek van de kwadratische functie redelijk snel. In de onderstaande tabel kan je nog eens duidelijk zien hoe snel de waarden oplopen. M.a.w. hoe snel het aantal berekeningen die je computer moet uitvoeren oploopt. De FFT heeft namelijk een efficiëntie van O ( N log\_2 ⁡ N ). Als we de grafiek van deze functie vergelijken met de kwadratische functie, zien we dat deze al een stuk minder snel stijgt.  
Om beter in te zien waarom de DFT zo traag werkt kijken we even naar de formules. We zien dat we ‘N’ complexe vermenigvuldigingen hebben en dat we ‘N-1’ optellingen hebben. Dit voor elke waarde van k, dus hebben we N^2 vermenigvuldigingen en N(N-1) optellingen. Meestal in de praktijk gebruikt men een groot aantal equidistante punten, waardoor je kan inzien dat dit allemaal zeer lang zal duren. In tegenstelling tot de DFT, moet de FFT veel minder berekeningen uitvoeren. Hierbij hebben we (N/2\*log\_2 N) vermenigvuldigingen en (N\*log\_2 N) optellingen.

Ook in de tabel zien we een aanzienlijk verschil. (dia 4)

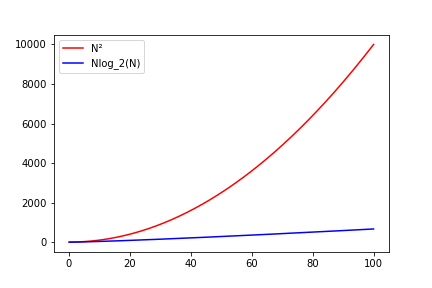
(Op de volgende dia staan de grafieken van de functies en de tabel.)

**Dia 4: Complexiteit van DFT vs FFT (tabel)**

Fig 2: optellingen en vermenigvuldigingen

Hier zien we een tabel waarbij we de tijdscomplexiteit illustreren. (N is het aantal equidistante punten.)

**Dia 5: Complexiteit van DFT vs FFT (grafiek)**

Fig 3: Plot waarop we het verschil in complexiteit vergelijken tussen de DFT (rood) en FFT (blauw). 

Hier zien we dat de complexiteit van de DFT kwadratisch stijgt volgens stijgende n, terwijl de complexiteit van de FFT bij benadering lineair toeneemt. Dit bevestigt ons vermoeden in verband met de efficiëntie van de Fast Fourier Transformatie ten opzichte van die van de Discrete Fourier Transformatie. De voornaamste reden om hiermee rekening te houden is uiteraard dat geluidsbestanden heel erg veel waarden bevatten. Als een geluidsbestand bijvoorbeeld 262.144 waarden bevat zal de complexiteit van de DFT van de grootteorde O(262.144^2) zijn, terwijl de FFT van de grootteorde O(250.000log\_2(262.144)) zal zijn. De tijdscomplexiteit van de DFT is dus maar liefst O(262.144^2)/O262.144log(262.144)) ~= 12.000 keer groter bij een geluidsbestand van 262.144 datapunten. (262.144 omdat dit de 18de macht van 2 is)

**Dia 6: Hoe wordt dit opgelost in de FFT?**

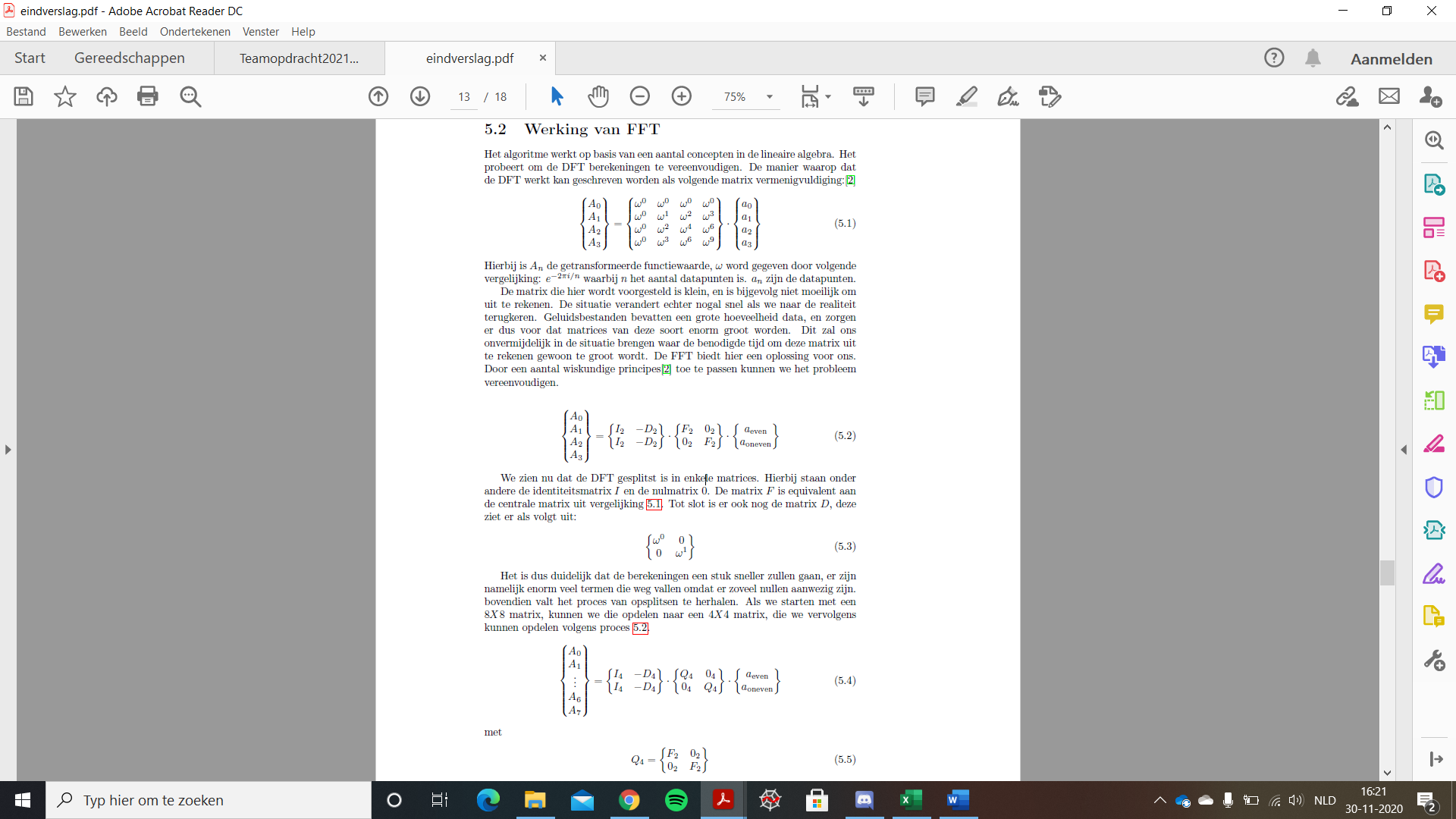
De vraag is nu hoe die efficiëntie van O( N\*log\_2 ⁡ N ) wordt verkregen.

Het FFT-algoritme van James Cooley en John Tukey maakt gebruik van de periodiciteit en de symmetrie van sinus- en cosinusfuncties. Daardoor kunnen heel wat datapunten weggelaten worden, omdat heel veel van de datapunten dezelfde informatie dragen. Het komt er uiteindelijk op neer dat een fouriertransformatie van lengte N wordt opgesplitst in twee fouriertransformaties met lengte N/2. Door dit principe recursief toe te passen, dat wil zeggen dat je de fouriertransformatie met lengte N/2 opnieuw in twee splitst indien mogelijk enz. , bekom je uiteindelijk een veel minder ingewikkelde berekening. Om deze simplificatie uit te kunnen voeren, moet het aantal datapunten een macht van 2 zijn, maar omdat het FFT-algoritme goed in mekaar zit, bekomt het, na het vervangen van het ‘teveel’ aan datapunten door nullen, nog steeds een zeer goede benadering. Je blijft deze simplificatie doorvoeren tot je een groot aantal zeer simpele matrices bekomt. Dit doet men aan de hand van enkele toepassingen van de lineaire algebra, waar we omwille van de moeilijkheid niet verder op in gaan. We kunnen wel een globaal overzicht geven van de manier waarop de FFT in eerste instantie als matrixproduct geschreven kan worden en ook hoe de simpelere matrices bekomen worden, en dit overzicht volgt in de rest van de slides.

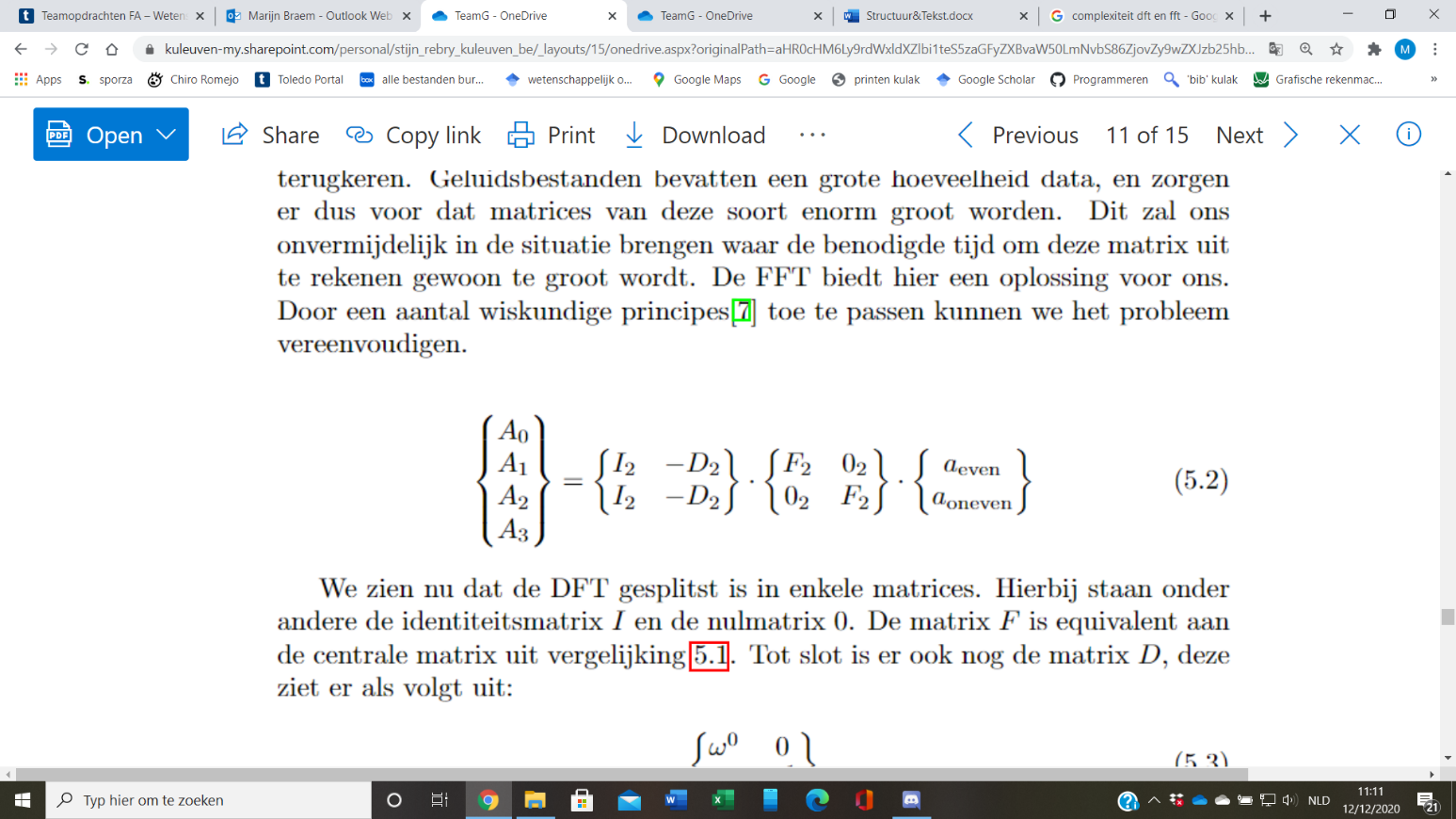
**Dia 7&8: Werking FFT (5.2)**

Om de FFT te snappen, moeten we eerst ook weten hoe de DFT in mekaar zit.

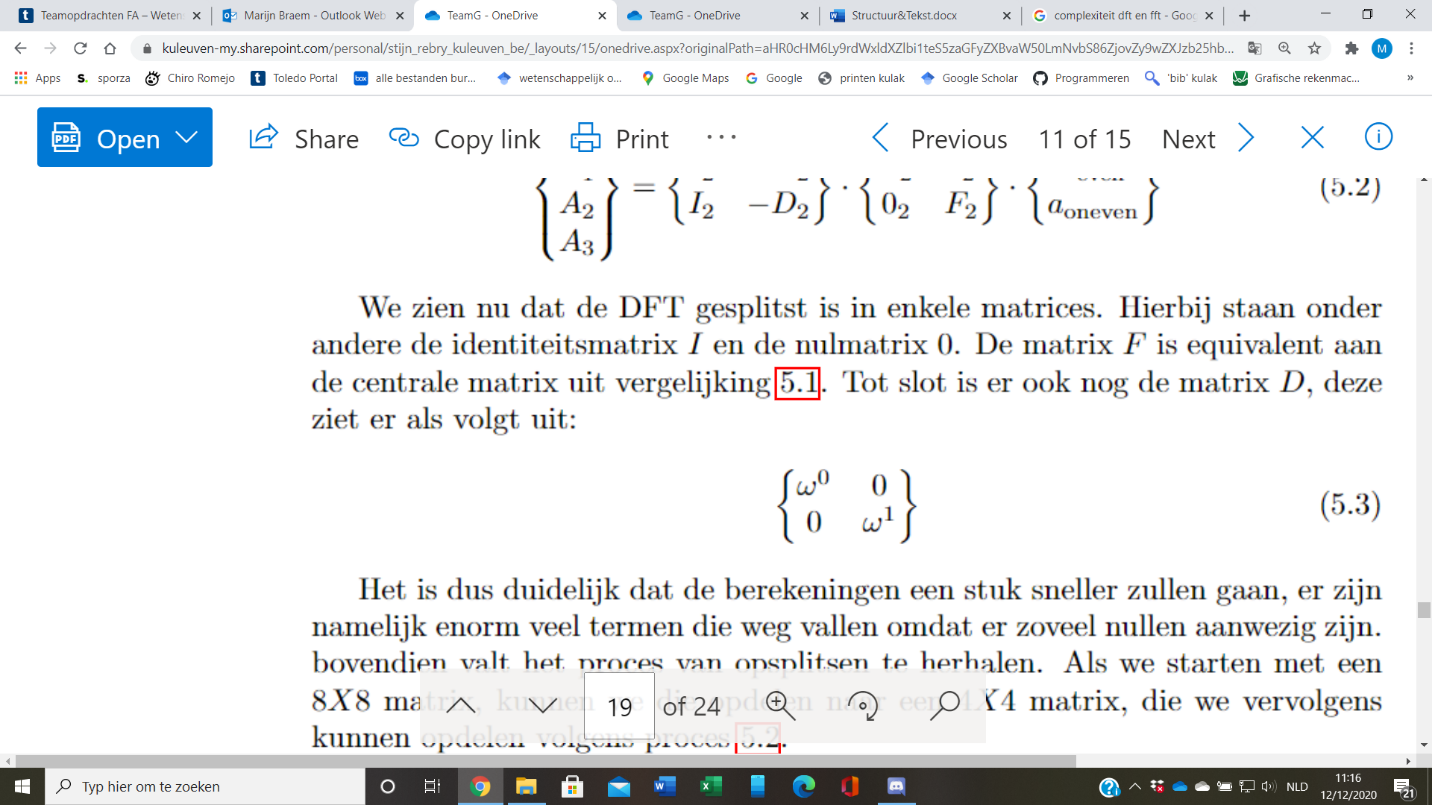
Het algoritme werkt op basis van een aantal concepten in de lineaire algebra. Het probeert om de DFT-berekeningen te vereenvoudigen. De manier waarop dat de DFT werkt kan geschreven worden als volgende matrixvermenigvuldiging:



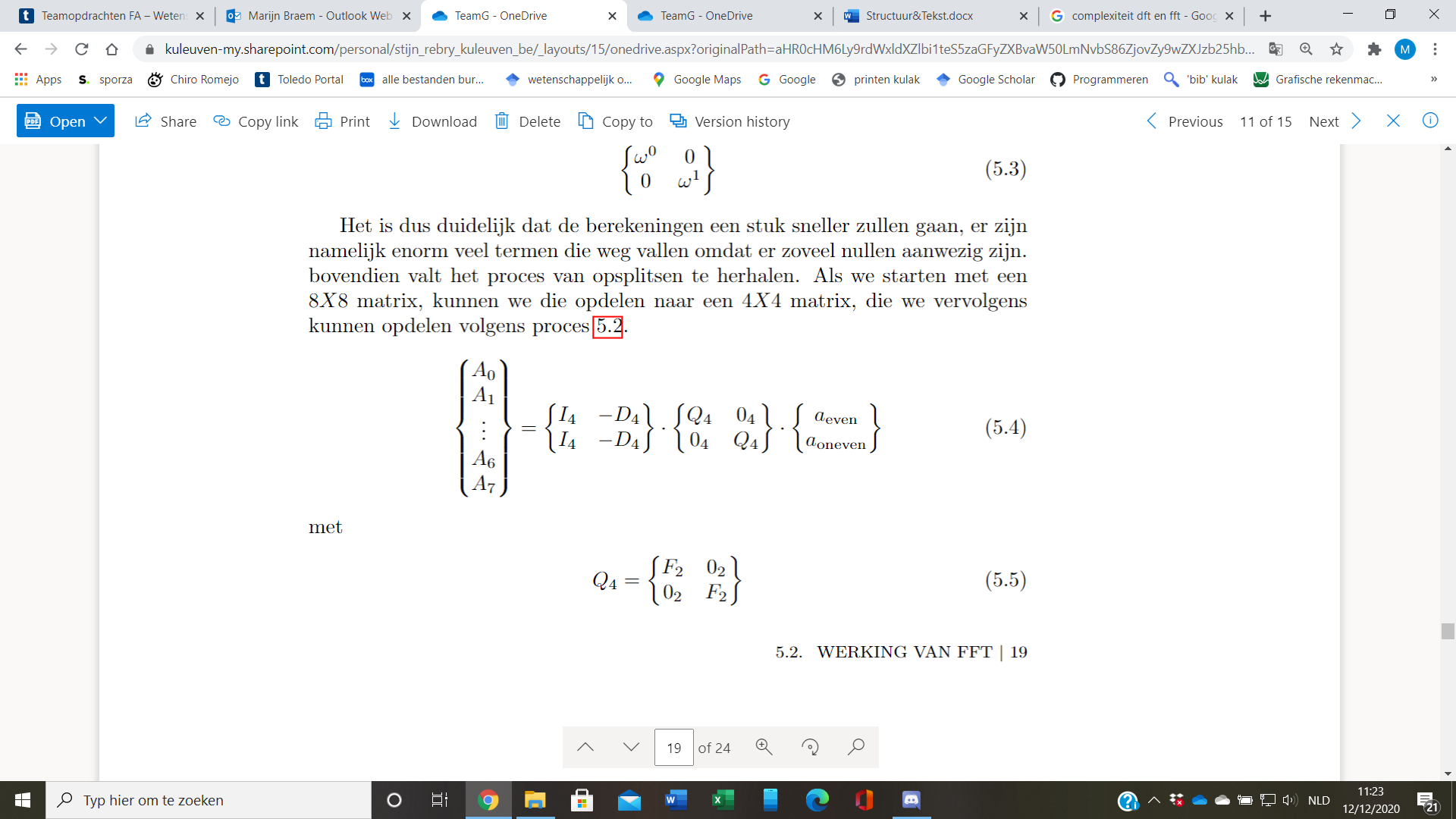
Hierbij is An de getransformeerde functiewaarde, ω wordt gegeven door volgende vergelijking: e−2πi/n waarbij n het aantal datapunten is. a\_n zijn de datapunten. De matrix die hier wordt voorgesteld is klein, en is bijgevolg niet moeilijk om uit te rekenen. De situatie verandert echter nogal snel als 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.



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 diagonaalmatrix D, deze ziet er als volgt uit:



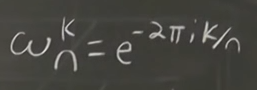
Het is dus duidelijk dat de berekeningen een stuk sneller zullen gaan, er zijn namelijk enorm veel termen die weg vallen omdat er zoveel nullen aanwezig zijn. Bovendien valt het proces van opsplitsen te herhalen. Als we starten met een 8X8 matrix, kunnen we die opdelen naar een 4X4 matrix, die we vervolgens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

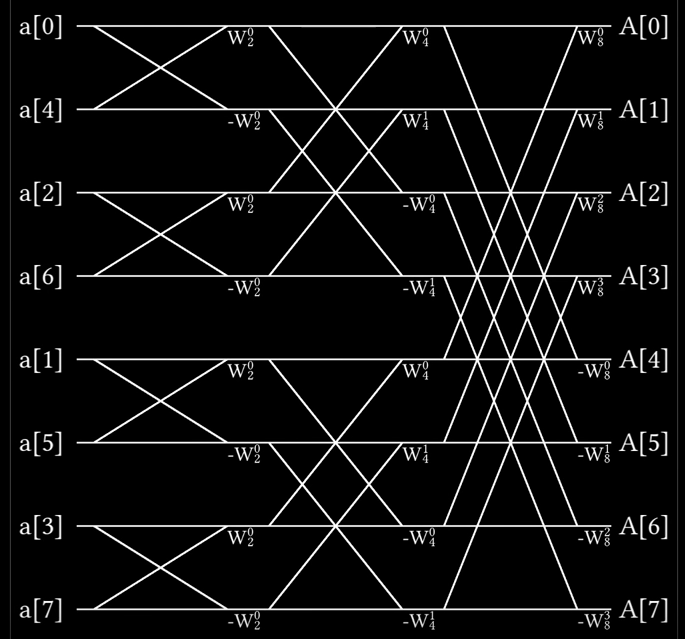


Wat extra verduidelijking: Je zou je kunnen afvragen waarom de FFT zoveel sneller werkt als je de data gewoon opdeelt in steeds kleinere deeltjes. Want dan moet je namelijk nog steeds alle nodige bewerkingen uitvoeren op al die deeltjes, waardoor je dus zou denken dat het evenveel tijd in beslag neemt. In 5.4 kan je namelijk zien dat ook de nulmatrices van orde veranderen. Dit betekent dat de bewerkingen gemakkelijker worden, en dus zal je computer minder rekentijd vereisen voor het toepassen van het FFT-algoritme. Dit wordt, naast de screenshot hierboven, ook uitgelegd in onderstaande youtube video’s.

**Dia 9: Het vlinderdiagram**

We kunnen deze methode ook op een andere manier illustreren, nl. aan de hand van een vlinderdiagram. Hierdoor zal je waarschijnlijk een beter zicht krijgen op hoe dit allemaal mogelijk is. Het komt erop neer dat we een **array van elementen** hebben die we op een strategische manier **door elkaar halen**. Ik zal er niet verder op ingaan welke methodes hiervoor worden gebruikt. Dan **combineren** we deze elementen tot paren. Daarna tot paren van paren. En daarna combineer je deze nog eens. Al deze combinaties noemen we **vlinders**. Elke vlinder stelt een zeer kleine **DFT** voor. Zo vermijden we uiteindelijk een heel aantal onnodige matrixbewerkingen.

Op deze afbeelding is . Bij de eerste vlinders is ’n’ gelijk aan 2, want we combineren 2 gekende punten. Nu blijkt dat de ’omega’ bij het ene punt gelijk is aan de negatieve omega van het andere punt. Dit door de koppels op een tactische manier (zoals eerder vermeld) te kiezen. Als we dan kijken naar de combinaties van deze paren zien we dat dit fenomeen zich opnieuw voordoet. Zo zullen we **uiteindelijk maar 7 vlinders**, ofwel DFT’s, moeten berekenen. Want de informatie die de koppels geven, blijkt gelijk te zijn. Dit is een stuk minder dan dat we er normaal zouden moeten berekenen. (**???** In plaats van 12 **???**)



<https://www.youtube.com/watch?v=XtypWS8HZco&t=163s>

<https://www.youtube.com/watch?v=XtypWS8HZco>

**Dia 10: Implementatie**

Implementatie:

Eerst en vooral nemen we een geluidsbestand in. Daarna passen we het FFT algoritme toe.

Daarna staat er op de dia’s dat we een manipulatie doen, maar dit is niet zo, het staat er enkel ter illustratie. Het is namelijk zo dat we tijdens deze fase, enorm gemakkelijk veranderingen kunnen doen aan de grafiek van de transformatie, om zo dingen weg te werken als ruis.

Daarna passen we de RFFT toe, dat is de reverse FFT. Dit zet de getransformeerde functiewaarden terug om naar een “geluidsbestand”. In realiteit moeten we nog de output van de RFFT ‘encoden’ naar een geluidsbestand.

**Conclusie:**

Om te concluderen maakt de fourieranalyse deel uit van ons dagelijks leven in vele opzichten. We hebben 2 soorten Fouriertransformaties: de DFT en de FFT. Eerst werd de DFT uitgevonden, maar als snel merkte men dat deze methode veel te traag werkt. Daarom was er nood aan een sneller algoritme, dus men vond de FFT uit. Deze is gebaseerd op de DFT. De DFT is gemakkelijker voor ons om te begrijpen, maar computers hebben moeite met de vele berekeningen die ze moeten uitvoeren. Daarom was er dus dringend nood aan iets nieuws.

* dagelijks leven
* 2 soorten fouriertransformaties
* verschil in uitvoeringssnelheid
* FFT is gebaseerd op DFT

**Youtube-filmpjes die de theorie duidelijk uitleggen:**

* Dia 6&7: <https://www.youtube.com/watch?v=E8HeD-MUrjY&t=64s>
* <https://www.youtube.lcom/watch?v=toj_IoCQE-4> vervolg
* <https://www.youtube.com/watch?v=spUNpyF58BY> (vanaf 12:10) ter aanvulling met

<https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/>

* <https://www.youtube.com/watch?v=XtypWS8HZco> (uitleg algoritme cooley tukey)

**Bronnen:** ✔ = in bronvermelding

<https://nl.wikipedia.org/wiki/Fast_Fourier_transform> ✔

<https://nl.wikipedia.org/wiki/Discrete_fouriertransformatie> ✔

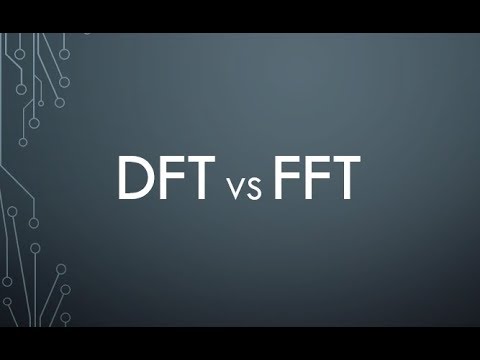
<https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/> ✔

<http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/dft.html> ✔

<http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html> ✔

<https://www.algorithm-archive.org/contents/cooley_tukey/cooley_tukey.html> ✔

[DFT vs FFT](https://www.youtube.com/watch?v=r1Rv09t7BqQ)



Antwoord Stijn over bronvermelding:

Bronvermelding in presentatie: zeker, dat hoort in elke vorm van wetenschappelijke verslaggeving! YouTube-filmpje: als je het toont, vermelding bij dat filmpje en/of als je er enkel concepten uit gehaald zijn bij de andere referenties