De Fast Fourier Transformatie

Arthur van Dam

2 maart 1998

Inhoudsopgave

1 Fourier Transformatie; Een introductie **2**
1.1 Inleiding 2
1.2 Motivatie en probleemstelling 2

2 Fourier-transformatie nader bekeken **3**
2.1 De DFT 3
2.1.1 Het principe 3
2.1.2 De wiskundige achtergrond 3
2.2 De FFT 4
2.2.1 Het principe 4
2.2.2 De wiskundige achtergrond 5

3 Practicumgedeelte **7**
3.1 *JAVA*-implementatie 7
3.1.1 De DFT 7
3.1.2 De FFT 7
3.2 Nauwkeurigheid 8
3.3 Snelheid 8
3.3.1 De DFT 9
3.3.2 De FFT 9
3.4 Geheugengebruik 10
3.4.1 De DFT 11
3.4.2 De FFT 11

4 Voorbeeld-transformaties **13**
4.1 De sinusgolf 13
4.2 Zaagtand- en blokgolf 14

5 Conclusie **16**

A Listings **17**
A.1 DFT 17
A.2 FFT 18
A.3 class Complex 19

Hoofdstuk 1

Fourier Transformatie; Een introductie

1.1 Inleiding

Dit verslag geeft een beschrijving van het *CS1-practicum*, waarbij de *Discrete FourierTransformatie*(DFT) bestudeerd werd. De Fourier-methode is een onderwerp dat goed in het vakgebied van Computational Science past. Het is namelijk een wiskundige methode om een fysisch verschijnsel – in dit geval golven – te analyseren. Bovendien is het model pas echt praktisch wanneer het op computersystemen toegepast wordt. Het is dus een onderwerp dat drie vakgebieden samenbrengt. In de praktijk zijn vele toepassingen hiervan te bedenken:

- Het verwijderen van ruis uit radio- en satellietsignalen
- Het tot beelden verwerken van medische stralingsmetingen.
- Het verwerken van seismische trillings-metingen.

Tijdens het practicum hebben wij ons voornamelijk op de wiskundige achtergrond en de toepassingen in eigen *JAVA*-programma's geconcentreerd. Doel is uiteindelijk om zo ervaring op te doen in het combineren van verschillende vakgebieden.

1.2 Motivatie en probleemstelling

Tijdens dit practicum ging het ons voornamelijk om het 'herontdekken' van de DFT, het veelzijdig toepassen hiervan en het interpreteren van de resultaten. Wanneer de basis duidelijk is, worden nieuwe eisen gesteld. Bij bovenstaande voorbeelden is het niet verwonderlijk dat nauwkeurigheid zeer belangrijk is. Bij grote hoeveelheden gegevens speelt snelheid ook een belangrijke rol. Aan beide vereisten blijkt de *Fast Fourier Transformatie*(FFT) te voldoen. Deze hebben wij dan ook uitgebreid behandeld, waarna een goede vergelijking met de oorspronkelijke (langzame) Fourier-transformatie mogelijk was. In hoofdstuk 3 worden de twee modellen meteen naast elkaar gelegd. Wanneer in dit verslag gesproken wordt over de DFT wordt het oorspronkelijke – langzame – model bedoeld. Wanneer de FFT genoemd wordt – ook een DFT – wordt het snelle model bedoeld.

Hoofdstuk 2

Fourier-transformatie nader bekeken

2.1 De DFT

2.1.1 Het principe

Wat is nu het idee achter de Fourier-transformatie? Zoals gezegd gaat het hier over golf-fenomenen, in welke vorm dan ook. Satellietbeelden, spraak of operamuziek; allemaal bestaan ze uit een combinatie van basistrillingen. Door nu een golf-signaal op te splitsen in alle grondfrequenties, komen we veel meer te weten over de eigenschappen van dat signaal. De Fourier-transformatie beschrijft een wiskundige techniek om zo'n signaal te splitsen.

Het idee achter deze Fourier-transformatie is als volgt: Gegeven een periodieke functie f kunnen we een sinus- of cosinus-functie g opstellen, waarmee we f vermenigvuldigen. De integraal over deze nieuwe functie geeft ons informatie over de periodiciteit van de oorspronkelijke functie f. Is de f in vergelijking met g een constante functie, dan levert het produkt nagenoeg diezelfde g op, waardoor de integraal 0 levert. Wordt de periodieke f echter vrij goed door g benaderd, neemt de amplitude toe, en de evenwichtsstand schuift omhoog, waardoor de integraal een grote waarde oplevert. Als we dit voor zeer veel g's doen met verschillende frequenties, kunnen we de mate van voorkomen van alle frequenties in de functie f bepalen.

In bovenstaande gedeelte zijn we van een continue periodieke functie uitgegaan, maar zoals uit de voorbeelden in sectie 1.1 blijkt, wordt in de praktijk vrijwel altijd met een eindig aantal meetpunten gewerkt. Dit kan ook nauwelijks anders, aangezien Fourier-reeksen zich er niet voor lenen om met de hand uitgerekend te worden. De computer biedt hier uitkomst, maar werkt – als digitaal apparaat – alleen met eindige, discrete meetreeksen. Daarom zullen we de formules wat moeten aanpassen naar een vorm voor de DFT.

2.1.2 De wiskundige achtergrond

Laten we eerst een T-periodieke functie $f : \mathbb{R} \to \mathbb{C}$ nemen die stuksgewijs continu is. Deze functie f is te schrijven als een combinatie van sinussen en cosinussen, die we in het vervolg als complexe e-macht zullen voorstellen, naar Euler's formule, $e^{it} = \cos t + i \sin t$. De Fourier-reeks van f wordt hiermee:

$$\tilde{f}(t) = \sum_{k=-\infty}^{\infty} c_k e^{i2\pi kt/T} \tag{2.1}$$

waarin de coëfficiënten c_k gegeven worden door:

$$c_k = \frac{1}{T}\int_0^T f(t)e^{-i2\pi kt/T}dt \tag{2.2}$$

Deze c_k's geven aan in welke mate een frequentie in het signaal voorkomt en zijn dus in feite de amplitudes van de frequentiecomponenten.

Zoals gezegd in sectie 2.1.1 is het wenselijk om formule 2.2 te discretiseren. Dit is te doen met behulp van de trapeziumregel voor integralen. Er kan nu een eindige som van 0 tot n opgesteld worden met $f(t_j)$ erin, waarin t_j gegeven wordt door: $t_j = j \cdot \frac{T}{n}$. Deze benadering levert:

$$\begin{aligned} c_k &= \frac{1}{T}\int_0^T f(t)e^{-i2\pi kt/T}dt \\ &\approx \frac{1}{T}\cdot\frac{T}{n}\left(\frac{f(0)}{2} + \sum_{j=1}^{n-1} f(t_j)e^{-i2\pi kt_j/T} + \frac{f(T)}{2}\right) \\ &= \frac{1}{n}\sum_{j=0}^{n-1} f(t_j)e^{-i2\pi jk/n}. \end{aligned} \tag{2.3}$$

In onze Fourier-context staan de c_k's in een vector $\mathbf{y}$ en staan de $f(t_j)$'s in een vector $\mathbf{x}$. De DFT van een vector $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$ wordt dan gegeven door de vector $\mathbf{y} = (y_0, \ldots, y_{n-1}) \in \mathbb{C}^n$ met

$$y_k = \sum_{j=0}^{n-1} x_j e^{-i2\pi jk/n}, \text{voor } 0 \le k \le n. \tag{2.4}$$

Voor het terugtransformeren van een Fourier-reeks moet een correctie van $\frac{1}{n}$ voor ieder element doorgevoerd worden, en de exponent wordt positief gemaakt. Hierna geldt de volgende formule voor de *backwardDFT*:

$$y_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j e^{i2\pi jk/n}, \text{voor } 0 \le k \le n. \tag{2.5}$$

In bovenstaande vergelijkingen stelt i het complexe getal voor, waarvoor geldt $i^2 = -1$.

2.2 De FFT

2.2.1 Het principe

In sectie 1.2 werd de FFT genoemd: een beduidend sneller model voor het berekenen van Fourier-reeksen. Waarop is dit nieuwe algoritme nu gebaseerd? Voor een Fourier-reeks moet in feite een $n \times n$-matrix F_n uitgerekend worden, waarvoor geldt: $\mathbf{y} = F_n \cdot \mathbf{x}$. Twee wetenschappers, Cooley en Tukey, ontdekten dat F_n een zogenaamde *ijle matrix* is en dus veel nullen bevat. Hierna bedachten ze dat ze een groot aantal berekeningen konden elimineren door dit probleem op te splitsen in twee deelproblemen, die erg veel op elkaar leken. Dit komt dus neer op het opsplitsen van de matrix in twee $\frac{n}{2} \times \frac{n}{2}$-matrices, en dit te herhalen tot $\frac{n}{2} = 1$.

Door dit recursieve algoritme reduceerde de looptijd $T(n)$ van de Fourier-transformatie van n^2 tot $n \log n$. Een tijdswinst die bij $n = 1024$ reeds een factor 100 bedraagt en bij toenemende n alleen maar groter wordt! Met de komst van de FFT ging een nieuwe wereld voor vele wetenschappers open en werd de Fourier-transformatie op zeer veel plaatsen toegepast.

De FFT dankt zijn succes ook aan het feit dat veel berekeningen in een Fourier-transformatie aan elkaar gelijk zijn. In sectie 2.2.2 zal blijken hoe dit zich in de formules uit. Er kan nu een recursief algoritme opgesteld worden dat van deze fomules gebruikt maakt.

Figuur 2.1: Pseudo-code voor het recursieve FFT algoritme. (Uit: [4])

```
RECURSIVEFFT(a)

1.  n ← length[a]          ▷n is a power of 2.
2.  if n = 1
3.      then return a
4.  ω_n ← e^{2πi/n}
5.  ω_n^{kj} ← 1
6.  a^[0] ← (a_0, a_2, ..., a_{n-2})
7.  a^[1] ← (a_1, a_3, ..., a_{n-1})
8.  y^[0] ← RECURSIVEFFT(a^[0])
9.  y^[1] ← RECURSIVEFFT(a^[1])
10. for k ← 0 to n/2 − 1
11.     do y_k ← y_k^[0] + ω_n^{kj} y_k^[1]
12.        y_{k+(n/2)} ← y_k^[0] − ω_n^{kj} y_k^[1]
13.        ω_n^{kj} ← ω_n^{kj} ω_n
14. return y               ▷ y is assumed to be a column vector.
```

Enkele aanvullende optimalisaties van de FFT worden in sectie 3.1.2 besproken.

2.2.2 De wiskundige achtergrond

Zoals gezegd is de FFT een recursief model. Een vector $\mathbf{y}$ wordt gesplitst in elementen op even en oneven posities. Van deze twee deelvectoren wordt ook weer de FFT bepaald door ze te splitsen. Dit gaat door tot $n = 1$, waarna de deelvectoren samengevoegd worden. Dit is mogelijk doordat formule 2.4 te schrijven is als:

$$y_k = \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{kj} x_{2j} + {\omega_n}^{k} \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{kj} x_{2j+1} \quad (2.6)$$

$$y_{k'} = \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{k'j} x_{2j} + {\omega_n}^{k'} \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{k'j} x_{2j+1} \quad (2.7)$$

waarin $0 \leq k < \frac{n}{2}$ en $k' = k + n/2$. Enig herschrijven maakt duidelijk dat:

$$\begin{aligned}
{\omega_{n/2}}^{k'j} &= {\omega_n}^{2(k'j)} \\
&= {\omega_n}^{2kj+nj} \\
&= {\omega_n}^{2kj} \cdot {\omega_n}^{nj} \\
\stackrel{{\omega_n}^n=1}{\longrightarrow} &= {\omega_n}^{2kj} \\
&= {\omega_{n/2}}^{kj}
\end{aligned}$$

waardoor geldt:

$$ {\omega_{n/2}}^{k'j} = {\omega_{n/2}}^{kj} \tag{2.8}$$

Ook blijkt dat:

$$\begin{aligned}
{\omega_n}^{k'} &= {\omega_n}^{k+n/2} \\
&= {\omega_n}^{k} \cdot {\omega_n}^{n/2}
\end{aligned}$$

en aangezien ${\omega_n}^{n/2} = 1$ geldt het volgende:

$$ {\omega_n}^{k} = -{\omega_n}^{k'} \tag{2.9}$$

Als we formule 2.8 en 2.9 invullen in formule 2.6 en 2.7 levert dit de volgende formules voor de FFT van een vector $\mathbf{x}$ op:

$$ y_k = \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{kj} x_{2j} + {\omega_n}^{k} \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{kj} x_{2j+1} \tag{2.10}$$

$$ y_{k'} = \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{kj} x_{2j} - {\omega_n}^{k} \sum_{j=0}^{\frac{n}{2}-1} {\omega_{n/2}}^{kj} x_{2j+1} \tag{2.11}$$

Groot voordeel van deze formules is dat ze dezelfde termen bevatten. Deze hoeven derhalve slechts éénmaal uitgerekend te worden.

Hoofdstuk 3

Practicumgedeelte

3.1 *JAVA*-implementatie

Nu het wiskundige model in sectie 2.1.2 behandeld is en de verbetering daarvan in sectie 2.2.2, kunnen we de computer de beide modellen laten doorlopen.

3.1.1 De DFT

Voor de DFT hebben we een programma geschreven, dat de *forward-DFT*(fDFT) en de *backward-DFT*(bDFT) van een vector met complexe getallen kan bepalen met behulp van formules 2.4 en 2.5. Aangezien *JAVA* nog geen complexe getallen herkent, hebben we zelf een klasse `Complex.java` geschreven. De belangrijkste methodes hieruit staan in sectie A.3. De listing van de methode `DFT.java` staat in sectie A.1. Hierin staat ook verdere uitleg over de werking in de vorm van commentaar. Deze methode bevat geen versnellende trucs; de output-vector wordt in een lus gevuld en voor ieder element is een lus nodig langs alle elementen van de input-vector. Zoals verwacht, geeft de DFT-methode na heen- en terugtransformeren weer de bronvector. Toch is er wel sprake van een zekere onnauwkeurigheid, zoals in sectie 3.2 zal blijken.

3.1.2 De FFT

Ook ons FFT-programma kan *forward* en *backward* transformaties uitvoeren afhankelijk van een *boolean* parameter. De beschrijving van de FFT, zoals in sectie 2.2.2 was voor ons aanleiding om nog een verbetering aan te brengen.

In regel 13 van het algoritme in figuur 2.2.1 wordt ${\omega_n}^{jk}$ telkens vermenigvuldigd met ω_n. Deze complexe vermenigvuldiging kost niet alleen veel tijd (6 flops per keer), maar maakt de hele transformatie ook een stuk onnauwkeuriger. Dit kan gelukkig opgelost worden aangezien ω_n^{jk} voor verschillende waarden van n bij bepaalde k dezelfde waarde heeft. Door nu alle verschillende waarden vooraf te bereken en in een array op te slaan vermijden we die complexe vermenigvuldiging. Deze array kan in de hele recursie-boom gebruikt worden. Er is dan wel een correctie van de index nodig, aangezien door het opsplitsen n en daarmee ω_n verandert. Hoe kleiner n wordt, des te groter moet de stap tussen twee opeenvolgende indexen in de array zijn om de goede waarde van de nieuwe ω_n eruit te halen. Deze factor wordt in de listing in sectie A.2 gegeven door:

$$stride = \text{originele lengte/lengte huidige array} = \text{omegatab.length/n}$$

De in de DFT gebruikte Vector-objecten lijken flexibel, maar zijn in de praktijk een stuk trager. Daarom wordt in de FFT gebruikt gemaakt van arrays. Als in het vervolg over vectoren wordt gesproken, worden de wiskundige vectoren bedoeld. Staat er ergens 'Vector', dan wordt het `Vector`-object uit *JAVA* bedoeld.

Een andere –puur versnellende – aanpassing zou kunnen zijn, bij een array-langte van 4 al te stoppen en de resterende Fourier-reeks hiervan te laten uitrekenen met de matrix F_4, die met de hand berekend en vervolgens ingeprogrammeerd is.

Om het programma ook voor vectorlengtes die geen macht van 2 zijn, te laten werken, zou er de volgende aanpassing kunnen worden gemaakt: De FFT wordt recursief bepaald, maar bij iedere nieuwe aanroep wordt gecontroleerd of n een macht van 2 is (`if n%2 == 0...`). Zo ja, wordt de FFT-methode verder uitgevoerd, zo nee, wordt een DFT uitgevoerd op de vector die nog over is.

3.2 Nauwkeurigheid

Zoals gezegd is de nauwkeurigheid van de DFT- en FFT-methode belangrijk. Om deze te testen van de beide programma's heb ik een testprogramma geschreven, dat een afhankelijk van een meegegeven parameter een vector van bepaalde lengte maakt, met als waarden precies één periode van $\sin x$. Vervolgens wordt de fDFT bepaald, waarvan de output-vector als input voor de bDFT dient. De uitkomst van de bDFT wordt vergeleken met de bron-vector, waarna het gemiddelde en maximale absolute verschil bepaald wordt. Dit wordt exact hetzelfde gedaan bij de FFT. De resultaten zijn in tabel 3.1 opgenomen. In sectie 3.1.2 werd al

lengte	DFT		FFT	
n	Maximaal	Gemiddeld	Maximaal	Gemiddeld
16	$1.91 \cdot 10^{-14}$	$7.31 \cdot 10^{-15}$	$4.58 \cdot 10^{-15}$	$1.83 \cdot 10^{-16}$
256	$3.62 \cdot 10^{-13}$	$9.64 \cdot 10^{-14}$	$1.55 \cdot 10^{-15}$	$4.13 \cdot 10^{-16}$
2048	$4.53 \cdot 10^{-12}$	$9.66 \cdot 10^{-13}$	$2.52 \cdot 10^{-15}$	$4.91 \cdot 10^{-16}$
65536			$3.69 \cdot 10^{-15}$	$6.55 \cdot 10^{-16}$

Figuur 3.1: Afwijkingen Fourier-transformaties

gezegd dat het constant berekenen van ${\omega_n}^{kj}$ door een vermenigvuldiging met ω_n de nauwkeurigheid negatief beïnvloed. In de FFT-methode werd hiervoor dus een aanpassing gemaakt. Daarnaast worden in de FFT-methode überhaupt al minder berekeningen gemaakt dan in de DFT, waardoor er ook minder afrondingsfouten kunnen optreden. Uit de experimentele resultaten blijkt dat de verbeteringen in de FFT de moeite waard zijn. Vooral bij grote vectorlengtes is het verschil in nauwkeurigheid tussen de DFT en FFT groot. De ogenschijnlijk kleine afwijkingen, kunnen bij doorrekenen in grote problemen onverwachte fouten opleveren. De FFT geniet op dit punt dus duidelijk de voorkeur.

3.3 Snelheid

Bij een *goed* programma denken velen meteen aan een *snel* programma. Alhoewel in sectie 3.2 is aangetoond dat nauwkeurigheid van groot belang is, mag de snelheid van de modellen zeker niet onbeschouwd blijven. Voordat de werkelijke tijd gemeten wordt, is het interessant om

een voorspelling te doen. In sectie 3.3.1 en 3.3.2 worden de beide modellen geanalyseerd. De hardware is echter ook van wezenlijke invloed op de looptijd. Alle metingen voor dit practicum zijn verricht op een Sun SparcStation4 met 32 Mb RAM, onder Solaris 4.2.1. Het is gebruikelijk om het aantal floating point-operaties per seconde te meten (in MegaFlops/s). Om deze voor het gebruikte workstation te bepalen heb ik een testprogramma in een lus van 1 miljoen keer een complexe optelling, vermenigvuldiging en e-macht laten uitvoeren. Hiervan kon ik de processortijd meten met de Unix-utility **time**. De resultaten staan in tabel 3.2. Een

operatie	tijd(s)	flops/keer	MFlops/s
Optelling	6.35	2	0.315
Vermenigvuldiging	7.60	6	0.789
E-macht	13.45	22	1.636
Initialisatie	16.61	geen	nvt
		Gemiddeld:	0.913

Figuur 3.2: Flop-tijd metingen

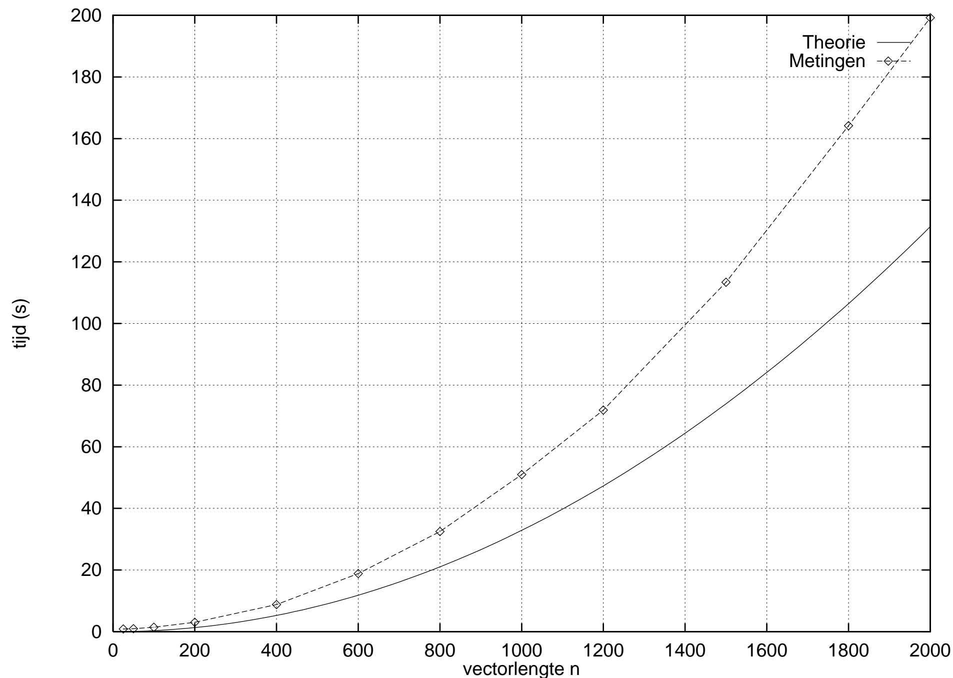
op het oog wat vreemde entry in deze tabel is de meting van het initialiseren van een Complex object. In de FFT wordt echter gebruik gemaakt van de methoden `retProd`, `retSum` en `retSub`, die een nieuw complex-object retourneren. Aangezien deze in een lus worden aangeroepen hebben ze veel invloed op de looptijd en moeten dus wel in rekening worden gebracht. Om in gelijke termen te kunnen spreken, zal ik net als bij de *echte* operaties de kosten van een initialisatie uitdrukken in flops. Dit komt uit op $16.61 \cdot 0.913 \approx 15.2$ denkbeeldige flops. Nu ik deze *externe factoren* bepaald heb, kan ik de modellen gaan analyseren.

3.3.1 De DFT

In de DFT (zie sectie A.1) wordt in de buitenste lus telkens een nieuw complex object geïnitialiseerd en een ander wordt op nul gezet. Het op nul zetten kost relatief zeer weinig tijd, maar het initialiseren kost in totaal ongeveer $15.2\,n$ flops. In de binnenste lus wordt een complexe e-macht, optelling en vermenigvuldiging uitgevoerd. Dit zorgt samen voor $n^2 \cdot (2 + 6 + 22) = 30\,n^2$ flops. Hiermee wordt de voorspelde tijd: $T(n) = 30\,n^2 + 15.2\,n$. Nu kunnen de gemeten waarden vergeleken worden met de voorspellingen. Voor de duidelijkheid zijn de waarden direct in een diagram uitgezet. Er is duidelijk een kwadratisch verband te zien in het verloop van de tijden. Bij toenemende n domineert de n^2-term over de n-term. De looptijd van de DFT is dan ook te schrijven als: $T(n) = \theta(n^2)$.

3.3.2 De FFT

In de FFT wordt aan het begin een tabel met alle waarden van $\omega_{n/2}^{kj}$ aangemaakt in de vorm van een array. Dit kost $n/2 \cdot 12 = 6n$ flops(2 vermenigvuldigingen en een sinus en een cosinus). Dan wordt de recursie-boom opgebouwd. In de *Merge-lus* aan het eind wordt een complex product, som en verschil berekend, waarvoor respectievelijk 6, 2 en 2 flops staan. Deze drie methoden maken iedere keer een nieuw Complex object aan, waarin de uitkomst wordt gezet. Dit kost in flops omgerekend ongeveer 15.2 flops per keer. De tijd hiervoor wordt dan gegeven



Figuur 3.3: Tijdsmetingen DFT

door:

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) + \frac{n}{2} \cdot 55.6 \\ &= 4T(\frac{n}{4}) + 2 \cdot \frac{n}{4} \cdot 55.6 + \frac{n}{2} \cdot 55.6 \\ &\vdots \\ &= n \cdot T(1) + n \cdot \log_2 n \cdot 27.8 \\ T(n) &= 27.8 \cdot n \log_2 n \end{aligned} \tag{3.1}$$

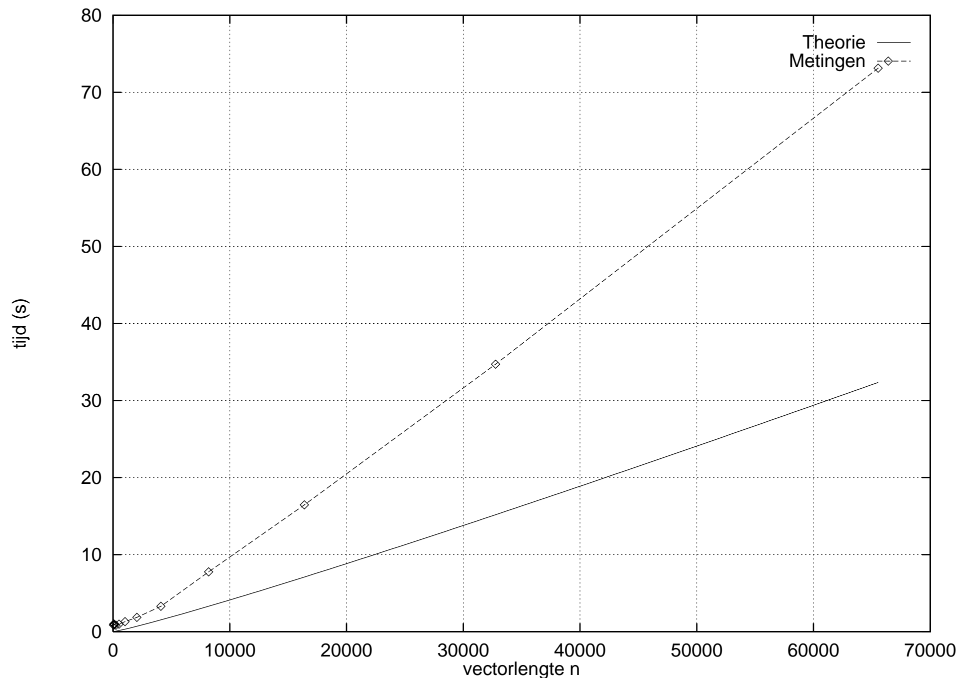
Wanneer de kosten van het aanmaken van de tabel worden meegeteld, levert dit uiteindelijk:

$$T(n) = 27.8 \cdot n \log_2 n + 6\, n \tag{3.2}$$

In figuur 3.4 zijn alle waarden weer opgenomen.

3.4 Geheugengebruik

Geheugengebruik is in *JAVA* moeilijk te bepalen. In tegenstelling tot *C* kan in *JAVA* het geheugen niet expliciet beheerd worden. Door naar de gebruikte variabelen te kijken, is een ruwe schatting te maken.



Figuur 3.4: Tijdsmetingen FFT

3.4.1 De DFT

In de DFT worden 2 Vectoren gebruikt – de meegegeven `source` en het vector object zelf –, elk ter lengte n. Deze vectoren bevatten elk n Complex-objecten, die op hun beurt weer twee *doubles* bevatten. Een *double* neemt 8 bytes geheugenruimte in beslag, dus het totale geheugengebruik van de DFT komt uit op $2\,n \cdot 2 \cdot 8 = 32\,n$ bytes.

3.4.2 De FFT

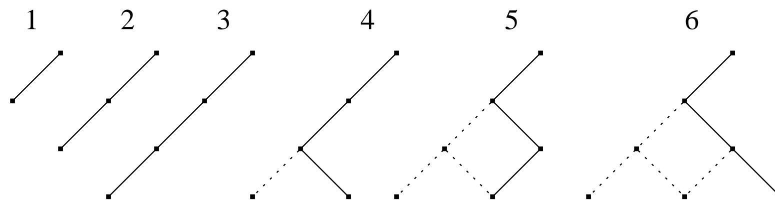
Bij de FFT kan niet zomaar van twee vectoren gesproken worden. Een vector wordt namelijk pas ingevuld als zijn twee helften zijn ingevuld. Bij iedere aanroep vande recursieve methode worden er drie nieuwe vectoren aangemaakt: `output`, ter lengte n,`odd` en `even`, elk ter lengte $n/2$.[1] In totaal zijn er dan dus $2 \cdot (n + n/2 + n/2) = 4\,n$ *doubles* in gebruik. Door splitsing wordt de vectorlengte steeds korter, waardoor die deelvector minder geheugen in beslag neemt. De *parent* van die halve vector maakt echter ook nog gebruik van het geheugen en de *parent* daarvan ook weer etcetera. De maximale hoeveelheid geheugen is dus in gebruik, wanneer het diepste punt van de recursie-boom is bereikt. Dit idee is voor het linker gedeelte grafisch uitgebeeld in figuur 3.5.

De bronvector die éénmalig aan de top van de recursieboom wordt meegegeven, levert ook nog eens extra lengte n en de tabel met waarden van $\omega_n/2^{kj}$ heeft ook nog lengte $n/2$. De lengte van alle arrays kan dan gesommeerd worden:

$$n_{tot} = (2\,n + n + n/2 + \ldots + 1) + n + n/2 \approx 5\frac{1}{2}\,n \qquad (3.3)$$

[1] De namen zijn afkomstig uit de *source-code* in sectie A.2.

Net als bij de DFT is n_{tot} het aantal complexe objecten dat in gebruik is. Deze bevatten twee *doubles*, die op hun beurt weer 8 *bytes* gebruiken, zodat het totale geheugengebruik uitkomt op $2 \cdot 8 \cdot 5\frac{1}{2}\,n = 88\,n$ *bytes*. Als de uitkomst vergeleken wordt met de DFT, blijkt dat de FFT meer geheugen nodig heeft bij een gegeven vectorlengte n. Dit is een nadeel, maar het hangt van de situatie af of dit opweegt tegen de grote voordelen van de snelheid en nauwkeurigheid.



Figuur 3.5: Doorlopen van de recursieboom

Hoofdstuk 4

Voorbeeld-transformaties

Ter verduidelijking van de werking van de Fourier-transformatie, is het interessant te kijken naar het effect hiervan op enkele standaardfuncties. Soms is het zelfs mogelijk om de uitkomst te voorpellen, maar dat wordt al snel onmogelijk. In sectie 2.1.2 werd al gezegd dat de getransformeerde $\mathbf{y}$ van $\mathbf{x}$ de amplitudes van de verschillende frequentiecomponenten weergeeft. Bij een letterlijke sinus- of cosinusfunctie is dit nog met de hand te bepalen. In sectie 4.1 wordt dit voor een sinus gedaan. In sectie 4.2 wordt geen uitwerking meer gegeven, alleen de grafieken worden afgebeeld.

4.1 De sinusgolf

Als voorbeeld wordt hier $f(x) = \sin(2\pi\, x)$ genomen. Fourier-transformatie werkt met periodieke functies van de vorm zoals die in formule 2.4 staat weergegeven. De exponent van de complexe e-macht geeft de periodiciteit van de functie weer. Bij $\sin(2\pi\, x)$ ligt het dus voor de hand dat $k = 1$ wordt gekozen. Als dit uitgeschreven wordt met Euler's formule, blijkt dat $k = -1$ ook een noodzakelijke bijdrage levert. In formules is dit eenvoudig te zien:

$$k = 1: \qquad e^{-i\,2\pi\,x} = \cos(2\pi\,x) + i\,\sin(2\pi x) \tag{4.1}$$

$$k = -1: \qquad e^{i\,2\pi\,x} = \cos(2\pi\,x) - i\,\sin(2\pi x) \tag{4.2}$$

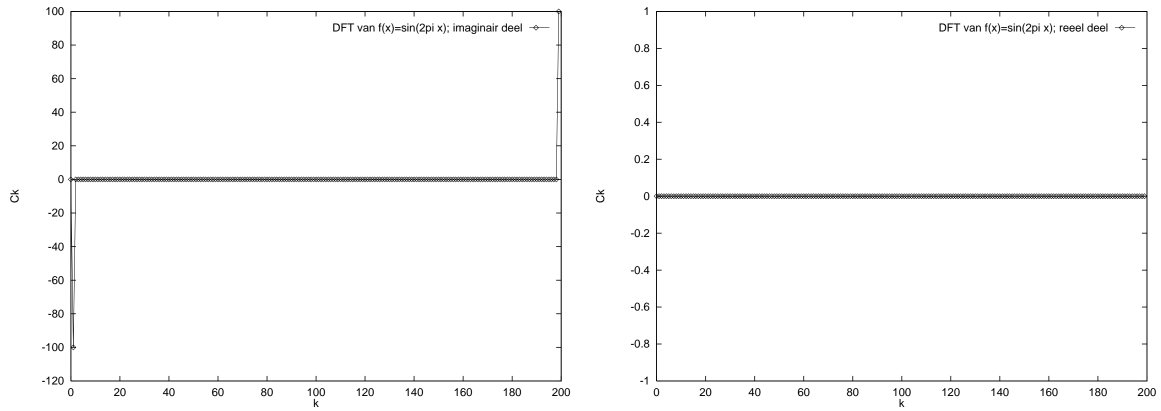
$$(4.1) - (4.2): \quad e^{i\,2\pi\,x} - e^{-i\,2\pi\,x} = 2\,i\,\sin(2\pi x) \tag{4.3}$$

De doelfunctie is $\sin(2\pi x)$ dus moet formule 4.3 met $\frac{1}{2\,i}$ worden vermenigvuldigd. Dit betekent dat formule 4.1 met $\frac{1}{2\,i}$ en formule 4.2 met $-\frac{1}{2\,i}$ moet worden vermenigvuldigd. Aangezien de factor $\frac{1}{n}$ bij bovenstaande berekening nog niet was meegenomen, moet daar nu nog wel een correctie voor worden gedaan. Dit levert uiteindelijk de gevraagde constanten:

$$\begin{aligned} c_{-1} &= -1 \cdot \frac{1}{2\,i} \cdot n = \frac{n}{2} \cdot i \\ c_{-1} &= 1 \cdot \frac{1}{2\,i} \cdot n = -\frac{n}{2} \cdot i \end{aligned}$$

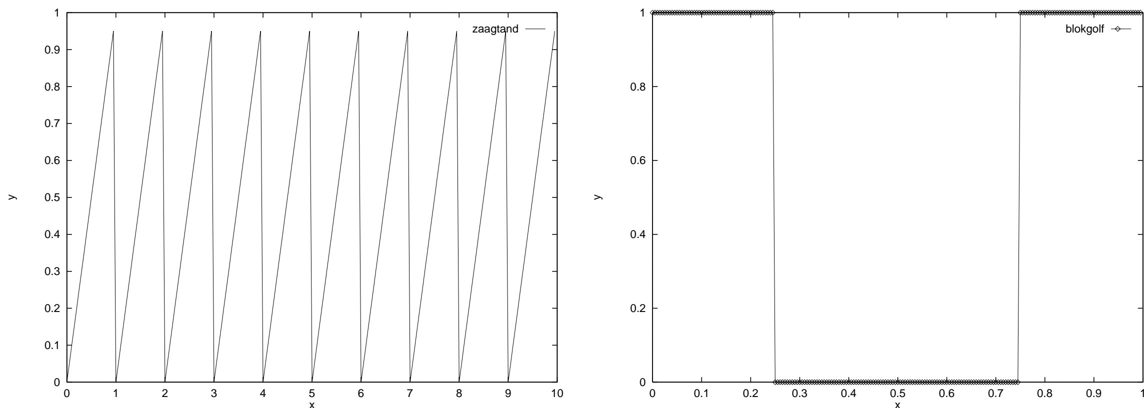
De negatieve frequentie bij $k = -1$ lijkt misschien wat vreemd, maar de Fourier-transformatie zet het negatieve gedeelte links van de oorsprong, aan de achterkant van het domein, geheel rechts van de oorspronkelijke functie. De uitkomst van de Fourier-transformatie wordt gesplitst in een reëel en imaginair deel. De voorspelling is dus dat de reëele reeks allemaal nullen

bevat, terwijl de imaginaire reeks een piek omlaag heeft bij $k = 1$, en één omhoog bij $k = n-1$. Voor $n = 200$ staat de Fourier-transformatie in figuur 4.1.

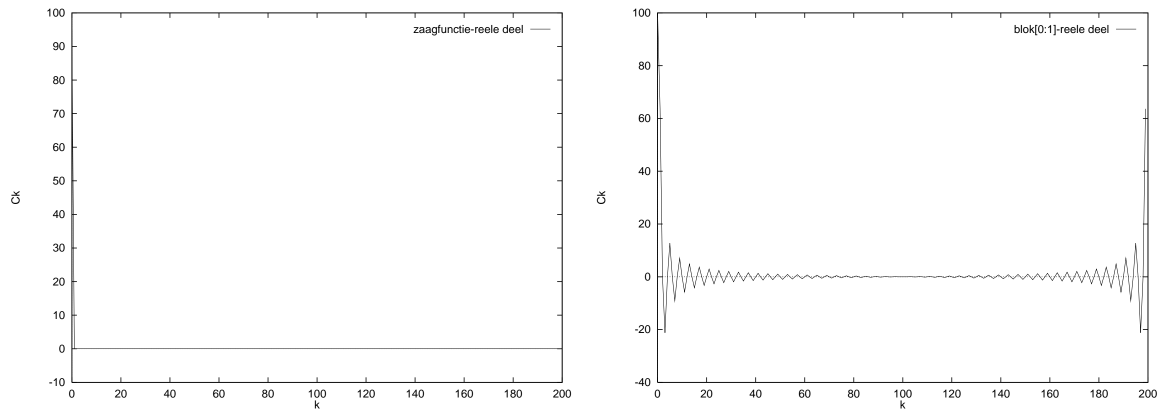


Figuur 4.1: Fourier-transformatie van $f(x) = \sin(2\pi x)$.

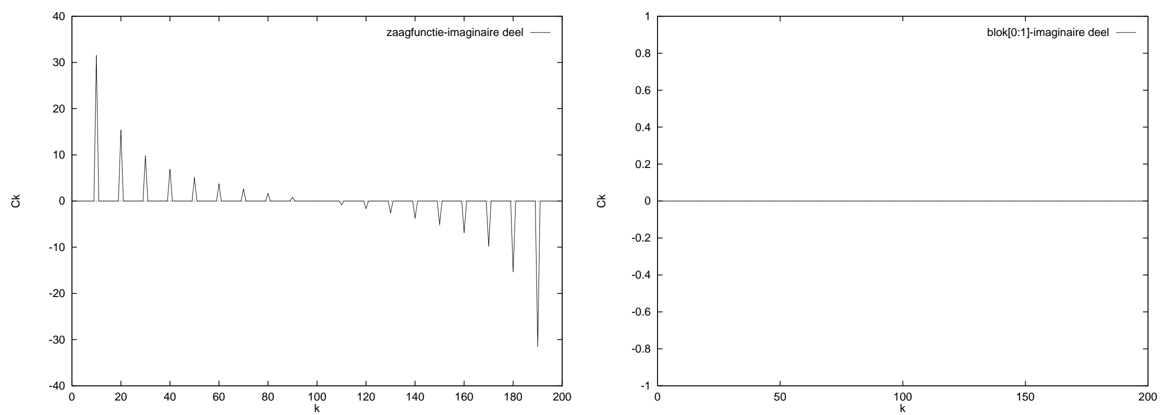
4.2 Zaagtand- en blokgolf



Figuur 4.2: De oorspronkelijke zaagtand- en blokgolffunctie.



Figuur 4.3: Reële deel van DFT van zaagtand en blokgolf.



Figuur 4.4: Imaginaire deel van DFT van zaagtand en blokgolf.

Hoofdstuk 5

Conclusie

Aan het eind van dit verslag een *overall view* ter vergelijking van beide modellen. Gebleken is dat de FFT niet zomaar een 'recht-toe-recht-aan'-model is. Er is over nagedacht en dit resulteerde in een enorme tijdswinst ten opzichte van het oude model. Daarnaast was de FFT ook een stuk nauwkeuriger dan de DFT, doordat er minder berekeningen worden uitgevoerd. Hierbij moet wel vermeld worden, dat het contrast tussen de beide modellen in sectie 3.2 niet helemaal reëel is. Bei de FFT werd namelijk ook nog gebruik gemaakt van een tabel met voorberekende waarden. Deze had echter ook in de DFT gebruikt kunnen worden. Uit [5] blijkt dat, wanneer beide modellen gebruik maken van een tabel, het verschil in nauwkeurigheid ongeveer een factor $1\frac{1}{2}$ à 2 bedraagt. Toch blijft dit de moeite waard.

Bovengenoemde verbeteringen leidden ertoe, dat de FFT wijdverbreid werd toegepast in de wetenschap en industrie en de medische wereld. De toepassingen, genoemd in sectie 1.1 brengen veel data met zich mee, waardoor een snel model vereist is. Een interessante toepassing van de FFT, is het filteren van geluid. Dit is mogelijk door een geluidssignaal – dat als getallenreeks wordt meegegeven – heen te transformeren. De ongewilde frequenties, moeten dan in de getransformeerde vector op $(0, 0)$ worden gezet. Dit moet wel symmetrisch om het midden van de reeks gebeuren, want de Fourier-transformatie heeft het linker gedeelte van de reeks, rechts achteraan gezet. Wanneer de vector nu teruggetransformeerd wordt, en de getallenreeks naar een geluidssignaal geconverteerd wordt, is het oorspronkelijke geluidssignaal weer terug, met uitzondering van de verwijderde frequenties. Een dergelijk programma is leuk om mee te experimenteren, maar ook praktische toepassingen zijn te bedenken. Een hinderlijke bromtoon kan op deze manier eenvoudig uit het geluid gefilterd worden.

Er bleek echter ook, dat de FFT meer geheugen gebruikt dan de DFT. Dit kan een probleem vormen, wanneer de hardware dit niet aankan. Tijdens dit practicum, was dit echter geen probleem. In deze, zich op computergebied razendsnel ontwikkelende tijd zal dit steeds minder een probleem zijn. De FFT zal dan ook nog vele malen zijn dienst bewijzen in de meest uiteenlopende toepassingen.

Bijlage A

Listings

A.1 DFT

```
public void DFT(Vector source, boolean forward)
{
   double phi;
   int n = source.size();
   if(forward) phi = -2*Math.PI/n;    //phi afhankelijk van fDFT of bDFT
   else        phi =  2*Math.PI/n;

   Complex temp_pow = new Complex (0.0,0.0);
   Complex temp_sum = new Complex (0.0,0.0);

   for (int k = 0; k < n; k++)
   {
       for (int j = 0; j < n; j++)
       {
           temp_pow.real = 0.0;        //exponent van e-macht bepalen.
           temp_pow.complex = (phi*k*j);
           temp_pow.pow();             //berekenen van e^(+/-2*i*j*k/n)

           temp_pow.product( (Complex) source.elementAt(j));
           temp_sum.sum(temp_pow);
       }
       if(forward)                    //correctie voor bDFT
            addElement(new Complex(temp_sum.real, temp_sum.complex));
       else addElement(new Complex(temp_sum.real/n, temp_sum.complex/n));
       temp_sum.clear();               //'resetten' voor hergebruik in zelfde lus
   }
}
```

A.2 FFT

```
class RecursiveFFT
{
    static Complex omegatab[];
    public static Complex[] startFFT(Complex[] source, boolean forward)
    {
        int n = source.length;
        int halfn = n/2;
        int t = -2;
        if(!forward)                    //correctie exponent voor bDFT
            t = 2;
        double theta = t*Math.PI/n;
        omegatab = new Complex[n/2];

        for(int i = 0; i < halfn; i++) //tabel met machten van omega vullen
            omegatab[i] = new Complex(Math.cos((double)i*theta), Math.sin((double)i*theta));
        Complex output[] = RecursiveFFT(source);
        if(!forward)                    //correctie (1/n) voor bDFT
            for(int i = 0; i < n; i++)
            {
                output[i].real /= n;
                output[i].complex /= n;
            }
        return output;
    }

    public static Complex[] RecursiveFFT(Complex[] source)
    {
        int n = source.length;
        int halfn = n/2;
        if(n == 1) return source;       //diepste punt van recursie-boom

        int stride = (2*omegatab.length)/n;
        Complex output[] = new Complex[n];
        Complex even[]   = new Complex[halfn];
        Complex odd[]    = new Complex[halfn];

        int pos = 0;
        for(int i = 0; i < halfn; i++) //source splitsen in even en oneven
        {
            even[i] = source[pos];
            odd[i]  = source[pos+1];
            pos += 2;
        }

        even = RecursiveFFT(even);      //los deelproblemen op
        odd  = RecursiveFFT(odd);
        Complex temp = new Complex();   //om omega x X[2j+1] in op te slaan

        for(int k = 0; k < halfn; k++) //Merge-lus
        {
            temp            = omegatab[k*stride].retProd(odd[k]);
```

```
            output[k]       = temp.retSum(even[k]);
            output[k+halfn] = temp.retSub(even[k]);
        }
        return output;
    }
}
```

A.3 class Complex

```
class Complex
{
    double real, complex;

    public Complex()
    {
    }

    public Complex(double r, double c)
    {
        real = r;
        complex = c;
    }

    public void sum(Complex other)    //(a+bi)+(c+di) = (a+c)+(b+d)i
    {
        real    +=other.real;
        complex +=other.complex;
    }

    public void product(Complex other)//(a+bi)(c+di) = (ac-bd)+(ad+bc)i
    {
        double temp_real = real;       //real-waarde moet nog bekend blijven.
        real = real*other.real - complex*other.complex;
        complex = complex*other.real + temp_real*other.complex;
    }

    public void pow()                  //e^(a+bi) = e^a*cos(b) + e^a*i*sin(b)
    {
        double temp_real = real;
        real = Math.exp(temp_real)*Math.cos(complex);
        complex = Math.exp(temp_real)*Math.sin(complex);
    }

    public Complex retSum(Complex b)   //retourneer som in nieuw object
    {
        return new Complex(real+b.real, complex+b.complex);
    }

    public Complex retSub(Complex b)   //retourneer verschil in nieuw object
    {
        return new Complex(real-b.real, complex-b.complex);
    }
```

```
public Complex retProd(Complex b)  //retourneer produkt in nieuw object
{
    return new Complex(real*b.real - complex*b.complex
                       complex*b.real + real*b.complex);
}

public void clear()                //zet huidig object op 0+0i
{
    real = 0.0;
    complex = 0.0;
}
```

Bibliografie

[1] Rob H. Bisseling. *Handleiding Computational Science Practicum*. 1997

[2] Rob H. Bisseling. *Parallel Scientific Computation*, pagina 75-76. 1998

[3] Barry A. Cipra. The FFT: Making Technology Fly. *Siam News*, pagina 1 & 23. Mei 1993.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*, hoofdstuk 32. The MIT Press, 1990.

[5] Verslag FFT, http://www.students.cs.ruu.nl/people/mastijnm/CS/verslag/front.html, 1997