Spectrum Analyse

Naam: Vak:

In deze opgave is het de bedoeling om op een heel eenvoudige manier een spectrum analyser te schrijven. Hiertoe implementeren we eerst de discrete Fourier transformatie (DFT).

Discrete Fourier Transformatie Gegeven een rij $x_0, \ldots, x_{N-1}$ van reëele coëfficiënten (die een signaal voorstellen), dan is de Fourier transformatie van deze rij eveneens een rij, maar van *complexe* coëficiënten $X_0, \ldots, X_{N-1}$, gedefiniëerd als volgt:

$$X_n = \sum_{k=0}^{N-1} x_k e^{-2\pi \frac{kn}{N} i} \in \mathbb{C} \tag{1}$$

voor $n = 0, 1, \ldots, N-1$.

Anderzijds, gegeven een rij van Fourier coëficiënten $X_0, \ldots, X_{N-1}$ dan is de inverse Fourier transformatie van deze rij een rij coëficiënten $x_0, \ldots, x_{N-1}$ zodat:

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{2\pi \frac{kn}{N} i} \in \mathbb{R} \tag{2}$$

Merk op dat deze definities gebruik maken van complexe getallen. Om te werken met complexe getallen gebruiken we in Haskell met het datatype `Complex` uit de module `Data.Complex`. Dit datatype is als volgt gedefinieerd:

```
data Complex a = a :+ a
```

Dit datatype stelt een complex getal voor door middel van zijn reëel en zijn imaginair deel. Een complex getal $a + bi$ schrijven we dus als `a :+ b`. Voor dit datatype zijn ook de gebruikelijke operaties zoals `+`,`-`,`*`,`/`,`exp`, enzovoorts gedefiniëerd.

Om het werken wat te vergemakkelijken definiëren we eerst twee hulpfuncties om een `Double` om te zetten naar een `Complex Double`

Opdracht 1 Schrijf twee functies `d2rc :: Double -> Complex Double` en `d2ic :: Double -> Complex Double` die een `Double` nemen en dit invullen in het reële, respectievelijk het imaginaire deel van een `Complex Double`.

```
> d2rc 2.0
2.0 :+ 0.0

> d2ic 2.0
0.0 :+ 2.0

> d2ic 2.0 + d2rc 2.0
2.0 :+ 2.0
```

Opdracht 2 Daarnaast gebruiken we een datatype om reële coëfficiënten te onderscheiden van Fourier coëfficiënten. Het type synoniem `type Signal = [Double]` is al voorgedefiniëerd.

Vul nu de definitie van het datatype `Fourier` aan zodat dit een lijst van `Complexe Double` coëfficiënten bevat. Vul ook de definities van `mkFourier` en `unFourier` aan, die een `Fourier` datatype aanmaken en afbreken.

```
> unFourier $ mkFourier [1 :+ 2, 3 :+ 4, d2ic 4]
[1.0 :+ 2.0,3.0 :+ 4.0,0.0 :+ 4.0]
```

Terzijde: Vlottende-komma getallen vergelijken

Het resultaat van een berekening met vlottende komma (floating-point) is haast nooit exact. Daarom kijke we naar de absolute waarde van het verschil van twee getallen, in plaats van getallen exact te vergelijken.

Opdracht 3 Gegeven is de klasse `AlmostEq` met daarin de methode `(~=) :: a -> a -> Bool` die `True` is als twee waarden bijna gelijk zijn.

Definieer nu:

1. Een instantie van `AlmostEq` voor `Double` zodat:

   $$a \sim= c \iff |a - c| < \varepsilon$$

   waar $\varepsilon = 10^{-14}$.

   Hint De functie `abs` kan hierbij van nut zijn. In Haskell schrijf je 10^{-14} als `1e-14`.

2. Een instantie van `AlmostEq` voor `Complex Double` zodat:

   $$a + bi \sim= c + di \iff |(a + bi) - (c + di)| < \varepsilon$$

   waar $\varepsilon = 10^{-14}$. Merk op dat $|(a + bi) - (c + di)|$ dus een reël getal is.[1]

   Hint De functie `magnitude` kan hierbij van nut zijn.

3. Een instantie van `AlmostEq a => AlmostEq [a]` die waar is als alle cöordinaten elementsgewijs bijna gelijk zijn en de lijsten een gelijke lengte hebben.

4. Een instantie voor `Fourier` die waar is als alle cöordinaten elementsgewijs bijna gelijk zijn.

   Hint Maak gebruik van de instantie voor `[a]`.

```
> 1.0 :+ (0 :: Double) ~= 1.0 :+ 0
True
> 1e-16 :+ (0 :: Double) ~= 0 :+ 0
True
> 1e-14 ~= (0.0 :: Double)
False
> [1.0, (0 :: Double)] ~= [1.0, 1e-16]
True
> [1.0, (0 :: Double)] ~= [1.0]
False
> mkFourier [1.0 :+ 0, 1e-16 :+ 0] ~= mkFourier [1.0 :+ 0, 0 :+ 0]
True
> mkFourier [1.0 :+ 0, 1e-16 :+ 0] ~= mkFourier [1.0 :+ 0, 1e-13 :+ 0]
False
```

Naïve DFT

In deze sectie moet je zowel de DFT als zijn inverse implementeren. Let bij het implementeren goed op dat je geen factoren zoals (-1) of $\frac{1}{N}$ vergeet.

[1] Dit noemen we de *magnitude* of absolute waarde van het complexe getal.

Opdracht 4 Implementeer de DFT door rechtstreeks gebruik te maken van Vergelijking 1. Doe dit door de implementatie van de functie `dft :: Signal -> Fourier` aan te vullen, zodat

$$\texttt{unFourier (dft [}x_0\texttt{,...,}x_{N-1}\texttt{]) = [}X_0\texttt{,...}X_{N-1}\texttt{]}$$

Hint Probeer zo veel mogelijk gebruik te maken van list-comprehensions. De functies `exp`, `pi` en `sum` kunnen zeker van pas komen.

```
> unFourier (dft [1,2,2,1]) ~= [6 :+ 0, (-1) :+ (-1), 0:+ 0, (-1) :+ 1]
True
> unFourier (dft [])
[]
```

Opdracht 5 Implementeer de inverse DFT door rechtstreeks gebruik te maken van Vergelijking 2. Doe dit door de implementatie van de functie `idft :: Fourier -> Signal` aan te vullen, zodat

$$\texttt{idft (mkFourier [}X_0\texttt{,...,}X_{N-1}\texttt{]) = [}x_0\texttt{,...}x_{N-1}\texttt{]}$$

Hint Probeer zo veel mogelijk gebruik te maken van list-comprehensions. De functies `exp`, `pi`, `sum` en `realPart` kunnen zeker van pas komen.

```
> idft (mkFourier [6 :+ 0, (-1) :+ (-1), 0:+ 0, (-1) :+ 1]) ~= [1,2,2,1]
True
> idft (dft [sin t | t <- [0,pi/4..2*pi]]) ~= [sin t | t <- [0,pi/4..2*pi]]
True
```

Spectrum Analyser

Een spectrum analyser is een programma dat de energieniveaus van het frequentiespectrum van een signaal toont. Concreet is het energieniveau van frequentie $\frac{n}{N}$ de magnitude van de Fourier coëfficiënt X_n.

Opdracht 6 Schrijf een functie `spectrumAnalyser :: Signal -> IO ()` die het frequentiespectrum van een signaal toont door voor coëfficiënt op een aparte regel een aantal '#'-symbolen uit te printen. Doe dit door de Fourier transformatie te berekenen, voor elke coëfficiënt de magnitude te bepalen en deze te herschalen zodat de grootste magnitude gelijk is aan 40. Het getal dat je dan bekomt, naar boven afgerond, is het aantal '#'-jes dat je moet afprinten.

Hint De functie `magnitude` uit `Data.Complex`, `ceiling` en `maximum` kunnen hier van pas komen.

```
> spectrumAnalyser [1,2,2,1]
########################################
##########

##########
> spectrumAnalyser [1,2,1,2]
########################################

##############
```

```
> spectrumAnalyser [sin t + sin (2*t) | t <- [0,pi/4..2*pi-pi/4]]
#
########################################
########################################
#
#
#
########################################
########################################
```

Fast Fourier Transform (Cooley-Tukey)

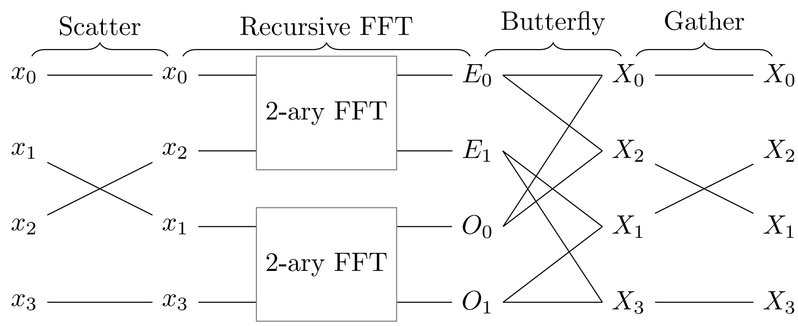
In de praktijk laat de performantie van de naïeve DFT, die rechtstreeks gebaseerd is op Vergelijkingen 1 en 2, te wensen over. De asymptotische complexiteit is namelijk $\mathcal{O}(N^2)$. Daarom gebruikt men meestal andere algoritmes, de zogenaamde *Fast Fourier Transforms* (FFT). Hiervan is het Cooley-Tukey algoritme wellicht het bekendste. De vorm die we hier gebruiken is specifiek voor het geval dat N *een macht van twee is* en gebruikt een recursieve strategie. Het splitst de coëfficiënten op in de even en oneven genummerde coëfficiënten en berekent van elke set apart de Fourier transformatie. Met deze coëfficiënten kunnen we dan de coëfficiënten van de volledige transformatie bepalen in een linear aantal stappen. Dit beperkt de tijdscomplexiteit tot $\mathcal{O}(Nlog\ N)$.

Het algoritme bestaat dus uit een aantal fasen (geïllustreerd in Figuur 1):

1. *Scatter*: Verdeel de coëfficiënten in even en oneven coëfficiënten $x_0, x_2, \ldots, x_{N-2}$ en $x_1, x_3, \ldots, x_{N-1}$.
2. *Recursive FFT*: Bepaal de Fourier transformaties van de even en oneven deellijsten, meer bepaald:
$$[E_0, \ldots, E_{\frac{N}{2}-1}] = \texttt{unFourier (fft [}x_0, x_2, \ldots, x_{N-2}\texttt{])}$$
$$[O_0, \ldots, O_{\frac{N}{2}-1}] = \texttt{unFourier (fft [}x_1, x_2, \ldots, x_{N-1}\texttt{])}$$
3. *Butterfly*: Combineer de even en oneven transformaties om de uiteindelijke coëfficiënten te bepalen volgens de volgende formule:
$$X_n = E_n + e^{-2\pi\frac{n}{N}i}O_n$$
$$X_{n+\frac{N}{2}} = E_n - e^{-2\pi\frac{n}{N}i}O_n$$
voor $n = 0, \ldots, \frac{N}{2} - 1$. De factor $e^{-2\pi\frac{n}{N}i}$ wordt ook wel de *twiddle* factor genoemd. Merk op dat E_n, O_n en de twiddle factor gedeeld worden tussen de berekening voor X_n en $X_{n+\frac{N}{2}}$.
4. *Gather*: Plaats de coëfficiënten terug in de juiste volgorde: De Butterfly fase plaatst X_n op index $2n$, en $X_{\frac{N}{2}+n}$ op index $2n+1$ voor $0 \leq n < \frac{N}{2}$. Rangschik de coëfficiënten zodat X_n op index n staat, voor $0 \leq n < N$.

Opdracht 7 Implementeer de functie `scatter :: [a] -> ([a],[a])` zodat de elementen op een even index in de eerste lijst terechtkomen, en die op een oneven index op de tweede lijst. Je mag er vanuit gaan dat de lijst een even aantal elementen bevat.

```
> scatter [0,1,2,3,4,5,6,7]
([0,2,4,6],[1,3,5,7])
> scatter [1,2,3,4,5,6,7,8]
([1,3,5,7],[2,4,6,8])
> scatter [False,True,False,True]
([False,False],[True,True])
> scatter [] :: ([Double],[Double])
([],[])
```



Figuur 1: Het Cooley-Tukey algoritme, voor $N = 4$.

Implementeer ook de functie `gather :: [a] -> [a]` die de elementen in de lijst weer op de juiste plaats zet. Merk op dat de *Gather*-fase rechtstreeks geïmplementeerd kan worden door `scatter` uit te voeren, en de twee lijsten die je krijgt samen te voegen. Dit komt omdat de X_n met $n \geq \frac{N}{2}$, die dus in de tweede helft van de lijst horen te staan, op oneven indices in de lijst staan. Visueel blijkt dit uit Figuur 1: de lijnen in de Scatter en Gather fasen zijn identiek. Je mag er dus ook vanuit gaan dat aantal elementen even is.

```
> gather [0,4,1,5,2,6,3,7]
[0,1,2,3,4,5,6,7]
```

Opdracht 8 Implementeer de functie `twiddle :: Int -> Int -> Complex Double -> Complex Double -> [Complex Double]` zodat

$$\texttt{twiddle}\ N\ n\ E_n\ O_n\ \texttt{=}\ \texttt{[}E_n + e^{-2\pi\frac{n}{N}i}O_n\texttt{,}\ \ E_n - e^{-2\pi\frac{n}{N}i}O_n\texttt{]}$$

```
> twiddle 16 0 (1 :+ 0) ((-1) :+ 1)
[0.0 :+ 1.0,2.0 :+ (-1.0)]
> -- note: exp (0 :+ (-2*pi*1/4)) = -i
> twiddle 4 1 1 ((-1) :+ 1)
[2.0 :+ 1.0,1.1102230246251565e-16 :+ (-1.0)]
```

Opdracht 9 Implementeer de functie `butterfly :: Int -> Fourier -> Fourier -> [Complex Double]` die gegeven N en de Fourier transformatie van de even en de oneven elementen, de butterfly stap uitvoert. Bijvoorbeeld voor $N = 8$ (voor de leesbaarheid is hier de `mkFourier` weggelaten bij de argumenten van `butterfly`):

$$\texttt{butterfly 8 [}E_0\texttt{,}E_1\texttt{,}E_2\texttt{,}E_3\texttt{] [}O_0\texttt{,}O_1\texttt{,}O_2\texttt{,}O_3\texttt{] = [}X_0\texttt{,}X_4\texttt{,}X_1\texttt{,}X_5\texttt{,}X_2\texttt{,}X_6\texttt{,}X_3\texttt{,}X_7\texttt{]}$$

Hint Pas de functie `twiddle` elementsgewijs toe, en voeg de resultaten samen. Hierbij kan de functie `zipWith3` van pas komen.

```
> butterfly 4 (mkFourier [3 :+ 0, (-1) :+ 0]) (mkFourier [3 :+ 0, 1 :+ 0])
[6.0 :+ 0.0,0.0 :+ 0.0,(-0.9999999999999999) :+ (-1.0),(-1.0) :+ 1.0]
```

Opdracht 10 Implementeer de functie `fft :: Signal -> Fourier` die de verschillende fasen van de FFT na elkaar uitvoert: Gebruik eerst `scatter` om het invoersignaal op te splitsen, voer dan de FFT recursief uit, gebruik `butterfly` om de signalen opnieuw te combineren en plaats de coëfficiënten daarna terug in de juiste volgorde met `gather`.

Je mag er vanuit gaan dat het aantal invoercoëfficiënten een strikt positieve macht van twee is. Merk op dat als je de FFT recursief uitvoert je uiteindelijk een signaal met slechts één coëfficiënt x_k bekomt. Je kan dan uit de definitie van de DFT opmaken dat $X_k = x_k + 0i$ (zie ook het eerste voorbeeldje hieronder).

```
> unFourier (fft [1])
[1.0 :+ 0.0]
> unFourier (fft [1,2,2,1]) ~= unFourier (dft [1,2,2,1])
True
> idft (fft [1,2,2,1]) ~= [1,2,2,1]
True
> dft [sin t | t <- [0,pi/4..2*pi-pi/4]] ~= fft [sin t | t <- [0,pi/4..2*pi-pi/4]]
True
```

Optioneel Pas `fft` aan zodat het algoritme werkt voor eender welke invoerlengte: als N even is, gebruik dan het FFT algoritme, val anders terug op `dft`.

Veel Succes