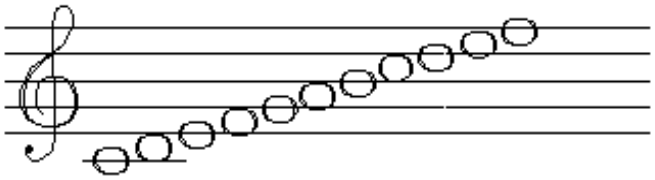
8 Hieman äänistä

Ääni on mekaanista aaltoliikettä, joka etenee palloaaltoina. Aaltoliikkeen taajuus ilmaistaan herzeinä (Hz).
Musiikkisävelet ilmaistaan kirjaimilla c, d, e, f, g, a, h. Eri sävelillä on eri taajuutensa. Siirryttäessä yksi oktaavi ylöspäin sävelen taajuus kaksinkertaistuu.



Seuraava taulukko esittää yksiviivaisen oktaavin sävelten taajuudet ja sävelen suhteen perussäveleeseen c.

Sävelkirjain	Taajuuden suhde c-säveleen	Taajuus/Hz
c	1	264
d	9/8	297
e	5/4	330
f	4/3	352
g	3/2	396
a	5/3	440
h	15/8	495
c	2	528

Perustietokoneen äänenlaatu on melkoisen heikko. Kaiutin antaa vain perussävelen, eikä lainkaan yläsäveliä, ja ääni kuulostaa monotoniselta. Juuri perussävel ja siihen liittyvät yläsävelet (joita molempia erilaiset instrumentit generoivat) muodostavat äänen soinnin. Näin sointuja ei siis voida generoida muutoin kuin määrittämällä soinnun nuotit peräkkäin annetuiksi (musiikkikielessä ns. arpeggio).
Perustietokoneella ei voida myöskään muuttaa äänen intensiteettiä. Kuitenkin voimme tehdä tietokoneesta soittimen määrittelemällä eri taajuuksia vastaamaan näppäinpainalluksia. Jotta jokin kappale voitaisiin soittaa automaattisesti, on lisäksi määriteltävä taukoviiveet ja kunkin sävelen kestoaika.
PC-tietokoneissa on prosessorin oheispiiri (8255), jonka kautta kaiutinta voi ohjata lähettämällä piirin porttiosoitteisiin arvoja. Tähän menettelyyn emme kuitenkaan tutustu, vaan lukija voi käyttää ohjelmointikielten lisäominaisuuksia äänten muodostamiseen.

Algoritmin voi laatia määrittelemällä ensin viiveitä esimerkiksi tyhjillä silmukoilla. Viiveiden pituus on tietokonekohtainen samoja parametreja käytettäessä, joten on ensin haettava esimerkiksi 0,5 sekunnin viiveen aikaansaavan silmukan laskuraja ja käytettävä sen kerrannaisia luomaan muun pituisia viiveitä. Jos 0,5 sekunnin ajatellaan vastaavan 1/4-nuottia, niin 1/2-nuottia vastaisi 2*0,5 sekuntia. Jotta ääni kuuluisi viiveen määrittelemän ajan, voidaan äänen generointiin yhdistää ehto (ohjelmalause), jolla tarkistetaan, onko esimerkiksi jokin sellainen ennalta määritetty näppäin painettuna, joka on määritetty muuttamaan perusviivettä. Voidaan esimerkiksi määrittää, että, jos ALT-näppäin on alhaalla samalla, kun valitaan tiettyä ääntä, viive kaksinkertaistuu jne. Samoin voidaan ohjelmassa määrittää, että SHIFT-näppäin alaspainettuna kaksinkertaistaa valitun taajuuden (eli tällöin siirrytään oktaavi ylöspäin). Näppäinten ja taajuuksien vastaavuudet voidaan järjestää helpoimmin ohjelmointikielen valintalauseella. Ohjelmaa kirjoitettaessa riittää, kun antaa taajuuden vain c-sävelelle ja käyttää muille sävelille ylläolevan taulukon kerrointa (käänteislukuna).
Äänen voi muodostaa esimerkiksi dos.h-kirjaston aliohjelmalla sound. Samassa kirjastossa on myös delay-aliohjelma. Mikäli käytetyn ohjelmointikielen taajuutta ei voida säädellä siten, että tiedettäisiin tarkka taajuusluku hertzeissä, kannattanee käyttää suoraan piiriä 8255, jonka portin 97 kautta käännetään kaiutin päälle (luku 79) ja pois sekä laskuripiiriä 8253, jonka perustaajuus on 1190000 Hz. Kaiutinta ohjaavaa piiriä 8255 käytetään siis ajastinpiirin 8253 kautta. Perustaajuutta jakamalla saadaan säveliä vastaavat taajuudet. Laskuripiiri on ensin asetettava oikeaan moodiin. Aiemmat piirien porttien tilat ja arvot tulee tallettaa ennen muuttamista ja palauttaa jälkeenpäin. Kiinnostuneen lukijan kannattaa syventyä tarkemmin piireihin ja niiden ohjaamiseen.

8.1 Morse-merkit

Tutkimme seuraavaksi morse-merkkejä ja lähetämme ne äänimerkkeinä. Käyttäjä voi täydentää ohjelman seuraavaa taulukkoa hyödyntäen:

A	B	C	D	E	F	G	H	I	J
.-	-…	-.-	-..	.	..-.	--.	….	..	.---
K	L	M	N	O	P	Q	R	S	T
-.-	.-..	--	-.	---	.--.	--.-	.-.	…	-
U	V	W	X	Y	Z	Å	Ä	Ö	
..-	…-	.--	-..-	-.--	--..	.--.	.-.-	---.	

Seuraavana on ohjelma morse-merkkien lähettämiseen.

Morse-merkit:

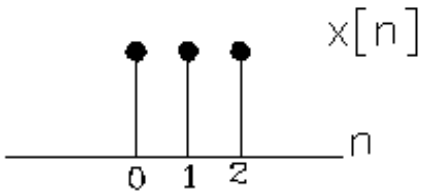
```
#include <iostream.h>
#include <dos.h>
#include <string.h>

class morse
{
char viesti[];
public:
morse (char sanoma[]);
void laheta();
void viiva();
void piste();
};

morse::morse(char sanoma[])
{
strcpy(viesti, sanoma);
}

void morse::viiva()
{
sound (100);
delay(600);
nosound();
delay(100);
}

void morse::piste()
{
sound (100);
delay(200);
nosound();
delay(100);
}

void morse::laheta()
{

int i;
for (i=0; i < strlen(viesti); i++)
{
if (viesti[i] == 's')
(piste(); piste(); piste();)

if (viesti[i] == 'o')
(viiva(); viiva(); viiva();)
```

```
}

}


void main()
{
morse merkit("sos");
merkit.laheta();
}
```

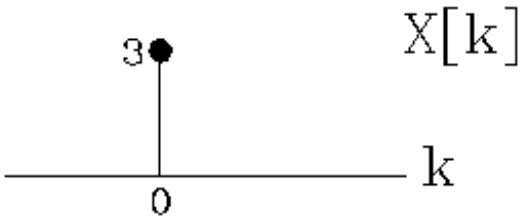
8.2 Diskreetti Fourier-muunnos (DFT, Discrete Fourier Transform)

Diskreetti Fourier-muunnos on diskreetti ja äärellisen pituinen taajuustason lukujono, joka on muodostettu vastaavasta aikatason diskreetistä signaalijonosta.

Seuraava kuva havainnollistaa aikatasossa äärellisen mittaista jonoa x[n], joka voi myös kuvata osaa jaksollisesta jonosta, jonka jaksonpituus on 3.



Vastaavaa DFT-muunnosta havainnollistetaan allaolevalla kuvalla:



8.2.1 DFT-muunnosparit

Äärellisten mittaisten sekvenssien DFT-muunnospari on:

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{kn}_N$$

ja

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)W^{-kn}_N$$

Kaavoissa $W_N = e^{-j2\pi/N}$

Valaisemme ensin muunnoksen tekemistä esimerkillä, jotta myös algoritmi avautuisi helpommin.

Esimerkki:

Tehdään ensin havainnollisuuden vuoksi muunnos X(k) -> x(n).

Olettakaamme, että meillä on X(k)-jono, jossa on 4 arvoa: 1, 3/4, 1/2, 1/4
Siis

X(0) = 1
X(1) = 3/4
X(2) = 1/2
X(3) = 1/4

Muunnoskaava oli seuraava:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)W^{-kn}_N$$

ja
$W_N = e^{-j2\pi/N}$

Nyt on siis:

N = 4 eli saadaan:

$$x(n) = 1/4 \sum_{k=0}^{3} X(k)\, e^{j2\pi nk/4}$$

Ensin muodostetaan koko jonoa koskeva lauseke, josta sitten lasketaan kukin x(n)-piste.

Lausekkeeksi saadaan siis seuraava:
Ensin ensimmäistä X(k)-arvoa vastaava lausekkeen arvo havainnollisuuden vuoksi tarkemmin.

$$1/4(X(0)\, e^{j2\pi n0/4}) = 1/4 (1 * e^{0}) = 1/4 * 1*1 = 1/4 \ (1)$$

Muut arvot (ja x(n)-jonon lauseke) saadaan samalla tavalla sijoittamalla:

$$x(n) = 1/4\{1 + X(1)\, e^{j2\pi n1/4} + X(2)\, e^{j2\pi n2/4} + X(3)\, e^{j2\pi n3/4} \}$$

=>

$$\mathbf{x(n) = 1/4\{1 + 3/4\, e^{j2\pi n1/4} + 1/2\, e^{j2\pi n2/4} + 1/4\, e^{j2\pi n3/4} \}}$$

Tästä lausekkeesta saadaan laskettua jono x(0) ... x(3) sijoittamalla järjestyksessä arvot 0...3 lausekkeeseen ja laskemalla kukin arvo. Lauseketta voi tosin sieventää hyödyntämällä Eulerin kaavaa ($e^{i\varphi} = \cos\varphi + i\sin\varphi$) ja vastaavasti kaavoja $\sin\varphi = 1/2i(e^{i\varphi} - e^{-i\varphi})$ ja $\cos\varphi = 1/2(e^{i\varphi} + e^{-i\varphi})$. Erityisesti viimeinen kaava näyttää olevan hyödyllinen siksi, että siinä esiintyy imaginääriyksikkö ainoastaan e:n eksponenttina.

Esimerkin vuoksi lasketaan jonon x(n) ensimmäinen arvo x(0) lausekkeesta:
$x(n) = 1/4\{1\, e^{j2\pi n0/4} + 3/4\, e^{j2\pi n1/4} + 1/2\, e^{j2\pi n2/4} + 1/4\, e^{j2\pi n3/4} \}$
Saadaan siis:

$$x(0) = 1/4\{1 + 3/4\, e^{j2\pi 01/4} + 1/2\, e^{j2\pi 02/4} + 1/4\, e^{j2\pi 03/4} \}$$

$$= 1/4 (1+3/4+1/2+1/4) = 5/8$$

Huomaamme, että arvojen x(1) ... x(3) laskennassa imaginääriyksikkö j säilyy e:n eksponentissa. Kompleksiluvuilla operointi sisältyy valitettavasti harvoihin ohjelmointikieliin. Eräs perinteinen ohjelmointikieli, jossa voidaan esittää kompleksitietoa on *Fortran*. Siinä kompleksitieto ilmaistaan kompleksiluvun *likiarvona* ja se esitetään sekä reaali- että imaginääriosan suhteen reaalisena. Muissa

ohjelmointikielissä (esim. C++) on mahdollisuus käyttää kirjastoja kuten C++:n MATH.H ja COMPLEX.H, joiden avulla selvitään kompleksilukujen laskemisesta.

Muunnamme edellä kehitetyn x(n)-lausekkeen ensin sinejä ja kosineja sisältävään muotoon.

Lausekkeemme oli:

$$x(n) = 1/4\{1 + 3/4\, e^{j2\pi n1/4} + 1/2\, e^{j2\pi n2/4} + 1/4\, e^{j2\pi n3/4}\}$$

Ja Eulerin kaava taas oli: $e^{i\varphi} = \cos\varphi + i\sin\varphi$

Saamme siis:

$$x(n) = 1/4(1 + 3/4(\cos2\pi n1/4 + j\sin2\pi n1/4) + 1/2(\cos2\pi n2/4 + j\sin2\pi n2/4) + 1/4(\cos2\pi n3/4 + j\sin2\pi n3/4)$$

Ja sievennettynä:

$$x(n) = 1/4(1 + 3/4(\cos\pi n1/2 + j\sin\pi n1/2) + 1/2(\cos\pi n + j\sin\pi n) + 1/4(\cos\pi n3/2 + j\sin\pi n3/2)$$

Koska esimerkkinä oli X(k)-jonon muuntaminen x(n)-jonoksi, myös algoritmi tähän voidaan esittää ensin.

Käytämme kaavoja:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)W^{-kn}{}_N$$

jossa
$W_N = e^{-j2\pi/N}$

Teemme Eulerin kaavan mukaisen muunnoksen heti alkuun, jolloin kaava saa muodon:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)W_N{}^{-kn}$$

==>

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)\, e^{j2\pi nk/N}$$

==>

$$x(n) = 1/N \sum_{k=0}^{N-1} [X(k)\, (\cos 2\pi nk/N + j\sin 2\pi nk/N)]$$

Seuraava ohjelma generoi DFT:n kaavaa käyttäen:

DFT:

```
Annetaan signaalien määrä N
Annetaan signaalit X(k) taulukkoon X[0...N-1)
Esitellään taulukko x[0...N-1]

n = 0 to N-1
    Summa = 0
    k = 0 to N-1
        Summa = Summa + 1/N [X[k] (cos2πnk/N + jsin2πnk/N)]
    next k
    x[n] = Summa
next n
```

Vastaavalla tavalla (muuttamalla algoritmia hieman) saadaan itse DFT eli aikatason äärellisen, diskreetin signaalijonon disktreetti Fourier-muunnos.

Muunnos DFT-jonosta aikatason jonoksi:

```
#include <iostream.h>
#include <math.h>
#include <complex.h>

const double pi =     3.14159265;


// DFT:n muuntaminen muunnoskaavalla
// Annettu X(k)-jono, josta lasketaan x(n)-jono
// X(k) on taajuustason DFT-jono
```

```
// x(n) on vastaava aikatason diskreetti-jono



void main()
{
int k, n;
double X[] = {1, 0.75, 0.5, 0.25, 0, 0.25, 0.5, 0.75};
int N = 8;
double x[8];

n =1;
for (n = 0; n < N; n++)
{

double lause = 0.0;
double lause2 = 0.0;
for (k=0; k < N; k++)
{
// Jos käytät ylempää laskutapaa, käytä complex-tyyppiä jäsenille
//lause = lause + (1.0/N * X[k]) * pow(exp(1),
real(complex(0,2*pi*n*k/N) ) );

// Tämä laskee reaalisen arvon oikein eikä herjaa complex-
yhteensopivuudesta
lause2 = lause2 + (1.0/N * X[k]) * ((cos(2*pi*n*k/N)) +
(real(complex(0,sin(2*pi*n*k/N)) ) ) );
}

//cout << real(lause) << "\n";

cout << lause2 << "\n";   // OK
cout << "\n";
}
// Tulee: 0.5,  0,2133883476483, 0, 0,036611652, 0, 0,036611652, 0,
0,2133883476483

}
```

Seuraavassa on muunnos aikatason jonosta taajuustason jonoksi (samat arvot kuin edellisessä tuloksessa, mutta toiseen suuntaan):

DFT-jonoksi:

```
#include <iostream.h>
#include <math.h>
#include <complex.h>
```

```
/* Muunnetaan aikatason diskreetti jono x[n] taajuustason
DFT-jonoksi  */

const long double pi =     3.14159265;

void main()
{
int k, n;
complex x[] = {0.5,  0.2133883476483, 0, 0.036611652, 0,
0.036611652, 0, 0.2133883476483};


int N = 8;


for (k = 0; k < N; k++)
{
complex lause = 0.0;

for (n=0; n < N; n++)
{
lause = lause + (x[n]) * pow(exp(1), (complex(0,-2*pi*n*k/N) ) );
}

cout << real(lause) << "\n";

}

// Tulee X(k) = 1, 0.75, 0.5, 0.25, 0, 0.25, 0.5, 0.75

}
```

8.2.2 Nopeampi muunnoksen laskenta

Kun käytämme kaavaa

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{kn}_{N}$$

suoraan, tarvitsemme N kompleksista kertolaskua ja N-1 kompleksista yhteenlaskua kutakin näytettä kohti. Tällöin kertolaskuja tulee N^2 kappaletta ja yhteenlaskuja N(N-1) kappaletta.

Nopeampia laskentatapoja saadaan hyödyntämällä termin W^{kn}_N symmetrisyyttä ja jaksollisuutta.

Kaavoissa $W_N = e^{-j2\pi/N}$

Symmetrisyys:

$W^{k(n-N)}_N = W^{-kn}_N$

Jaksollisuus:

$W^{kn}_N = W^{k(n+N)}_N = W^{(k+N)n}_N$

Nopeammissa muunnosten laskutavoissa hyödynnetään edellä mainittuja ominaisuuksia ja niiden avulla ohjelman tehokkuus on luokkaa NlogN.

Jos tutkit vaikkapa edellisten muunnosohjelmien antamia tuloksia, näet että samat arvot toistuvat.

Näissä muunnosmenettelyissä jaetaan DFT-jono lyhyemmiksi DFT-jonoiksi juuri symmetrisyytä ja jaksollisuutta hyödyntäen. Yleensä jono puolitetaan useamman kerran, jolloin laskentojen määrä putoaa dramaattisesti. Muuntamisen syöttöjonon määrän onkin usein oltava parillinen luku tai yleisimmin kakkosen potenssi.

Nopeampia muunnostapoja kutsutaan Fast Fourier Transform (FFT) -menetelmiksi.

Yleisesti käytetty FFT-muunnosmenettely on ns. decimation-on-time.
Siinä haetaan parillisia ja parittomia x[n]-jonon alkioita vastaavat arvot erikseen ja tehdään niille N/2-mittainen DFT-muunnos. Menettelyn takana on ns. hajota-ja-hallitse -toimintatapa. Menettelyä käytetään muun muassa polynomien kertolaskuissa.

Koko näytejono jaetaan ensin kahdeksi osaksi, joissa toisessa ovat parillisten indeksien näytteet ja toisessa parittomien indeksien näytteet. Tuloksena on kaksi jonoa, joissa kummassakin on N/2 jäsentä.
Seuraavaksi voidaan nuo N/2-mittaiset jonot jakaa uudelleen parillisiin ja parittomiin näytteisiin, jolloin saadaan 4 kpl N/4-mittaisia näytteitä.
Tätä jakamista jatketaan, kunnes näytteiden määrä on 2, joka voidaan laskea yhdellä kompleksisella kertolaskulla.

Koko muunnoksen laskenta päätyy lopulta erillisten 2 alkion pituisten jonojen laskentaan. Noita jonoja kutsutaankin nimellä butterfly (perhonen).

Ennen FFT:n laskentaa tulee näytteet asettaa vielä tiettyyn järjestykseen. Tässä järjestelyssä voidaan hyödyntää sitä tietoa, että näytteet ovat bit reversed (käännetyt bitit) -järjestyksessä.

Uusi indeksi, jonka mukaan laskentajärjestys määräytyy, syntyy, kun alkuperäisen jonon indeksin bittijono käännetään peilikuvakseen (esimerkiksi indeksiä 001 vastaa uusi indeksi 100).

Seuraavana on ohjelma, joka laskee aikatason arvot taajuustason näytteistä:

FFT:

```
#include <iostream.h>
#include <math.h>
#include <complex.h>


complex x[] ={0.5,  0.2133883476483, 0, 0.036611652,
              0, 0.036611652, 0, 0.2133883476483};
// aikatason jonoksi tulee:
// {1, 0.75, 0.5, 0.25, 0, 0.25, 0.5, 0.75};

const long double pi = 3.1415926;
int maara();
void fft(int N, complex x[]);
void kaanna(int N, complex x[]);

void main()
{

int N = 8;

/*complex temp[8];  Bittien käännön havainnollistaminen
temp[0] = x[0];
temp[4] = x[1];
temp[2] = x[2];
temp[6] = x[3];
temp[1] = x[4];
temp[5] = x[5];
temp[3] = x[6];
temp[7] = x[7];   */

int e;

for (e = 0; e < N; e++)
cout << x[e] << "\n";
cout << "\n";
```

```
kaanna(N, x);
fft(N, x);
for (e = 0; e < N; e++)
cout << x[e] << "\n";
cout << "\n";


}

void fft(int N, complex x[])
{
int tila = 1, leveys;
int S, M, R, jarj = N;

complex t1, t2;
double a;

for (M = 0; jarj != 1; M++)
jarj = (jarj >> 1);

for (int s = 1; s <= M; s++)
{
tila = pow(2,s);
S = N/tila;
leveys = tila/2;

for (int p = 0; p <= (leveys - 1); p++)
{
R = S * p;
a = 2 * pi * R/N;
t1 = complex(cos(a), -sin(a));

for (int o = p; o <= N-2; o = o + tila)
{
int b = o + leveys;
t2 = x[b] * t1;
x[b] = x[o] - t2;
x[o] = x[o] + t2;
}
}
}

}


void kaanna(int N, complex x[])
{
complex temp[8];
int M, jarj = N;
```

```
for (M = 0; jarj != 1; M++)
jarj = (jarj >> 1);

for (int i = 0; i < N; i++)
{
int ind1 = 0;
int ind2 = i;

for (int j = 0; j <= M-1; j++)
{
ind1 = ind1 + ((( 1 << j ) & ind2) ? (1 << (M-1-j)) : 0);
}

temp[ind1] = x[i];
}

for (i=0; i < N; i++)
{
x[i] = temp[i];
cout << x[i] << "\n";
}


}
```