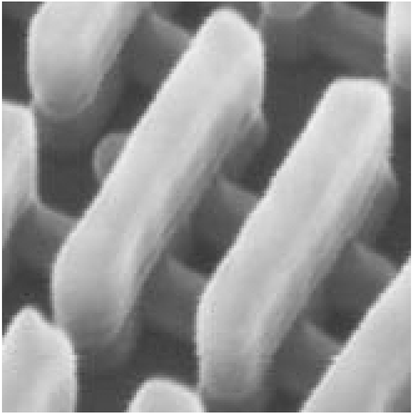
Digital representasjon

Dag Langmyhr
dag@ifi.uio.no

Hvordan lagre

- tall
- tekst
- bilder
- lyd

som **bit** i en datamaskin



Binære tall

For å bruke bit (0 og 1) som tall, må vi telle binært. Dette gjøres egentlig på samme måte som vi teller desimalt:

- Øk siste siffer med 1.
- Hvis vi ikke har flere siffer, sett til 0 og gjenta for sifferet til venstre.

Binært	0	1	10	11	100	101	110	111	1000	1001	1010	1011
Desimalt	0	1	2	3	4	5	6	7	8	9	10	11

Notasjon

Hvis det mulighet for tvil, skriver vi binære tall som 1001_2 og desimale tall som 1001_{10}.

Et matematisk blikk

I det desimale tallsystemet har posisjonene vekt
$1, 10, 100, 1000, \ldots = 10^0, 10^1, 10^2, 10^3, \ldots$

(Notasjonen a^n («a i n-te») betyr $\underbrace{a \times a \times \cdots \times a}_{n \text{ ganger}}$.)

I det binære tallsystemet har posisjonene vekt
$1, 2, 4, 8, \ldots = 2^0, 2^1, 2^2, 2^3, \ldots$

$$\begin{array}{ccccccc} & & 8 & 4 & 2 & 1 & \\ & & 2^3 & 2^2 & 2^1 & 2^0 & \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \\ 1011_2 & = & 1 & 0 & 1 & 1 & = 11_{10} \end{array}$$

Vi teller binært med grupper av bit, f eks **byte** som inneholder 8 bit:

0	0	0	0	0	0	0	0	$= 0_{10}$
0	0	0	0	0	0	0	1	$= 1_{10}$
0	0	0	0	0	0	1	0	$= 2_{10}$

$$\vdots$$

0	1	1	1	1	1	1	0	$= 126_{10}$
0	1	1	1	1	1	1	1	$= 127_{10} = 2^7 - 1$

Vi stopper når vi kommer til øverste bit.

Negative tall

Et negativt tall n lagres i 8 bit som $2^8 + n = 256 + n$:

1	1	1	1	1	1	1	1	$= -1_{10} = 256 - 1 = 255$
1	1	1	1	1	1	1	0	$= -2_{10}$
1	1	1	1	1	1	0	1	$= -3_{10}$
⋮								
1	0	0	0	0	0	0	1	$= -127_{10}$
1	0	0	0	0	0	0	0	$= -128_{10}$

Øverste bit («fortegnsbitet») angir om et tall er negativt.

Heltall i Java

Java tilbyr disse tallene:

byte	8 bit	−128 - +127
short	16 bit	−32 768 - +32 767
int	32 bit	−2 147 483 648 - +2 147 483 647
long	64 bit	−9 223 372 036 854 775 808 - +9 223 372 036 854 775 807

Hvorfor er dette viktig?

Overflyt.java

```
class Overflyt {
    public static void main(String arg[]) {
        int v = 1000000, v2 = v*v;

        System.out.println("v=" + v + " og v2=" + v2);
    }
}
```

gir dette resultatet

```
$ javac Overflyt.java
$ java Overflyt
v=1000000 og v2=-727379968
```

Hvor store tall kan vi da lagre?

IkkeOverflyt.java

```
class IkkeOverflyt {
    public static void main(String arg[]) {
        long v = 1000000, v2 = v*v;

        System.out.println("v=" + v + " og v2=" + v2);
    }
}
```

gir riktig svar:

```
$ javac IkkeOverflyt.java
$ java IkkeOverflyt
v=1000000 og v2=1000000000000
```

Hva så med Python?

Hovedregel: Python jonglerer med lagring av tall, så overflyt forekommer ikke.

Men: De fleste biblioteker man bruker i Python, er skrevet i C, og der har man de samme problemene som i Java.

Heksadesimal notasjon

Det er lett å gjøre feil når man jobber med binære tall:

11111110110111001011101010011000...
...01110110010101000011001000010000

Det er enklere å erstatte fire og fire binære sifre med **heksadesimale** sifre 0x**0**–0x**F**:

$$\underbrace{1111}_{0xF}\underbrace{1110}_{0xE}\underbrace{1101}_{0xD}\underbrace{1100}_{0xC}\underbrace{1011}_{0xB}\underbrace{1010}_{0xA}\underbrace{1001}_{0x9}\underbrace{1000}_{0x8}$$

$$\underbrace{0111}_{0x7}\underbrace{0110}_{0x6}\underbrace{0101}_{0x5}\underbrace{0100}_{0x4}\underbrace{0011}_{0x3}\underbrace{0010}_{0x2}\underbrace{0001}_{0x1}\underbrace{0000}_{0x0}$$

Oktal notasjon

Tidligere brukte man ofte **oktal** notasjon der man slår sammen tre og tre bit:

$$\underbrace{111}_{07}\underbrace{110}_{06}\underbrace{101}_{05}\underbrace{100}_{04}\underbrace{011}_{03}\underbrace{010}_{02}\underbrace{001}_{01}\underbrace{000}_{00}$$

(I Java skriver man oktale tall som **0*nnn***; i Python 2 skriver man dem også som **0*nnn***, men i Python 3 bruker man **0o*nnn***.)

I dag har hex-notasjonen overtatt.

Men hva med disse tallene?

- Andromedagalaksen er 24 029 742 100 000 000 000 km unna.
- $\pi = 3{,}14159265$
- Et H-atom er 0,000 000 096 mm stort.

Vi trenger altså ikke bare heltall men også tall som

- kan ha veldig små og veldig store verdier
- kan ha desimaler
- ikke behøver å være helt nøyaktige.

Løsningen i det desimale tallsystemet er å oppgi tallet med 10-erpotens:

- Andromedagalaksen er $2{,}4 \cdot 10^{19}$ km unna.
- $\pi = 3{,}14159265$
- Et H-atom er $9{,}6 \cdot 10^{-8}$ mm stort.

Så lagrer vi både den justerte tallverdien (9,6) og 10-erpotensen (−8).

På en datamaskin gjør vi det samme, men bruker vi 2-erpotenser i f eks float:

31	30 … 23	22 … 0
S	*2-erpotens*	*Tallverdien*

I Java har vi disse flyttallene:

			Minste	Største
float	32 bit	7 sifre	$1{,}2 \cdot 10^{-38}$	$3{,}4 \cdot 10^{38}$
double	64 bit	16 sifre	$2{,}2 \cdot 10^{-308}$	$1{,}8 \cdot 10^{308}$

Python har bare **double** men kaller den en **float**.

Vi kan konvertere mellom heltall og flyttall i Java:

Konvertering.java

```
class Konvertering {
    public static void main(String arg[]) {
        double f1 = 3.94, f2;
        int i;

        i = (int)f1;  f2 = (double)i / 2;
        System.out.println("i=" + i + " og f2=" + f2);
    }
}
```

Svaret blir: i=3 og f2=1.5

Det er tilsvarende i Python:

konvert.py

```
f1 = 3.94
i = int(f1)
f2 = float(i) / 2
print "i =", i, "og f2 =", f2
```

Svaret blir: i = 3 og f2 = 1.5

Er dette viktig?

FloatTest.java

```
class FloatTest {
    public static void main(String arg[]) {
        double a = 1.000000000000000000001,
               b = 1.000000000000000000008;

        System.out.println("a-b = " + (a-b));
    }
}
```

fungerer slik:

```
$ javac FloatTest.java
$ java FloatTest
a-b = 0.0
```

floattest.py

```
a = 1.000000000000000000001
b = 1.000000000000000000008
print "a-b =", a-b
```

fungerer slik:

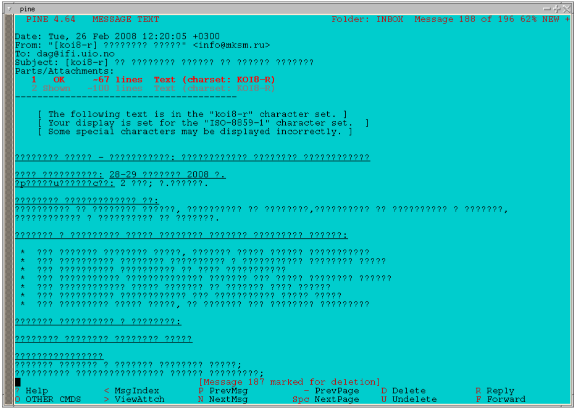
```
$ python floattest.py
a-b = 0.0
```

Tekster

Hvordan lagrer vi tegn i datamaskinen? Det er lett:

Sett opp en tabell over tegn vi trenger og gi hvert tegn et nummer.

Er det virkelig så enkelt?



Problemet

Hva hvis ikke alle bruker samme tabell?

ASCII

Første vellykkede forsøk på standardisering var ASCII (American standard code for information interchange) i 1963:

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- \+ ASCII ble etter hvert brukt av de fleste.
- \+ Det er lett å sjekke om et tegn er et siffer eller en bokstav.
- \+ Det er lett å konvertere fra liten til stor bokstav.
- – Mangler ÆØÅ og andre bokstaver og tegn.

Det siste ga opphav til et utall av lokale varianter der for eksempel **[\]{|}** ble erstattet av **ÆØÅæøå**. Dette førte til «Gj|vik-syndromet».

Latin-1

En klart bedre løsning ble **ISO 8859-1** (Latin-1).

Den (og varianter) er ennå i bruk.

ISO 8859–1

0 000 00	32 040 20	64 100 40 @	96 140 60 ‘	128 200 80	160 240 A0	192 300 C0 À	224 340 E0 à
1 001 01	33 041 21 !	65 101 41 A	97 141 61 a	129 201 81	161 241 A1 ¡	193 301 C1 Á	225 341 E1 á
2 002 02	34 042 22 "	66 102 42 B	98 142 62 b	130 202 82	162 242 A2 ¢	194 302 C2 Â	226 342 E2 â
3 003 03	35 043 23 #	67 103 43 C	99 143 63 c	131 203 83	163 243 A3 £	195 303 C3 Ã	227 343 E3 ã
4 004 04	36 044 24 $	68 104 44 D	100 144 64 d	132 204 84	164 244 A4 ¤	196 304 C4 Ä	228 344 E4 ä
5 005 05	37 045 25 %	69 105 45 E	101 145 65 e	133 205 85	165 245 A5 ¥	197 305 C5 Å	229 345 E5 å
6 006 06	38 046 26 &	70 106 46 F	102 146 66 f	134 206 86	166 246 A6 ¦	198 306 C6 Æ	230 346 E6 æ
7 007 07	39 047 27 ’	71 107 47 G	103 147 67 g	135 207 87	167 247 A7 §	199 307 C7 Ç	231 347 E7 ç
8 010 08	40 050 28 (	72 110 48 H	104 150 68 h	136 210 88	168 250 A8 ¨	200 310 C8 È	232 350 E8 è
9 011 09	41 051 29)	73 111 49 I	105 151 69 i	137 211 89	169 251 A9 ©	201 311 C9 É	233 351 E9 é
10 012 0A	42 052 2A *	74 112 4A J	106 152 6A j	138 212 8A	170 252 AA ª	202 312 CA Ê	234 352 EA ê
11 013 0B	43 053 2B +	75 113 4B K	107 153 6B k	139 213 8B	171 253 AB «	203 313 CB Ë	235 353 EB ë
12 014 0C	44 054 2C ,	76 114 4C L	108 154 6C l	140 214 8C	172 254 AC ¬	204 314 CC Ì	236 354 EC ì
13 015 0D	45 055 2D -	77 115 4D M	109 155 6D m	141 215 8D	173 255 AD -	205 315 CD Í	237 355 ED í
14 016 0E	46 056 2E .	78 116 4E N	110 156 6E n	142 216 8E	174 256 AE ®	206 316 CE Î	238 356 EE î
15 017 0F	47 057 2F /	79 117 4F O	111 157 6F o	143 217 8F	175 257 AF ¯	207 317 CF Ï	239 357 EF ï
16 020 10	48 060 30 0	80 120 50 P	112 160 70 p	144 220 90	176 260 B0 °	208 320 D0 Ð	240 360 F0 ð
17 021 11	49 061 31 1	81 121 51 Q	113 161 71 q	145 221 91	177 261 B1 ±	209 321 D1 Ñ	241 361 F1 ñ
18 022 12	50 062 32 2	82 122 52 R	114 162 72 r	146 222 92	178 262 B2 ²	210 322 D2 Ò	242 362 F2 ò
19 023 13	51 063 33 3	83 123 53 S	115 163 73 s	147 223 93	179 263 B3 ³	211 323 D3 Ó	243 363 F3 ó
20 024 14	52 064 34 4	84 124 54 T	116 164 74 t	148 224 94	180 264 B4 ´	212 324 D4 Ô	244 364 F4 ô
21 025 15	53 065 35 5	85 125 55 U	117 165 75 u	149 225 95	181 265 B5 µ	213 325 D5 Õ	245 365 F5 õ
22 026 16	54 066 36 6	86 126 56 V	118 166 76 v	150 226 96	182 266 B6 ¶	214 326 D6 Ö	246 366 F6 ö
23 027 17	55 067 37 7	87 127 57 W	119 167 77 w	151 227 97	183 267 B7 ·	215 327 D7 ×	247 367 F7 ÷
24 030 18	56 070 38 8	88 130 58 X	120 170 78 x	152 230 98	184 270 B8 ¸	216 330 D8 Ø	248 370 F8 ø
25 031 19	57 071 39 9	89 131 59 Y	121 171 79 y	153 231 99	185 271 B9 ¹	217 331 D9 Ù	249 371 F9 ù
26 032 1A	58 072 3A :	90 132 5A Z	122 172 7A z	154 232 9A	186 272 BA º	218 332 DA Ú	250 372 FA ú
27 033 1B	59 073 3B ;	91 133 5B [	123 173 7B {	155 233 9B	187 273 BB »	219 333 DB Û	251 373 FB û
28 034 1C	60 074 3C <	92 134 5C \	124 174 7C |	156 234 9C	188 274 BC ¼	220 334 DC Ü	252 374 FC ü
29 035 1D	61 075 3D =	93 135 5D]	125 175 7D }	157 235 9D	189 275 BD ½	221 335 DD Ý	253 375 FD ý
30 036 1E	62 076 3E >	94 136 5E ^	126 176 7E ~	158 236 9E	190 276 BE ¾	222 336 DE Þ	254 376 FE þ
31 037 1F	63 077 3F ?	95 137 5F _	127 177 7F	159 237 9F	191 277 BF ¿	223 337 DF ß	255 377 FF ÿ

Unicode

Men det er bare én løsning på sikt: en tegnkoding som omfatter alle skriftspråk i verden. Den heter **Unicode** og er nå stort sett ferdig.

- Unicode skal omfatte alle skriftspråk som brukes eller har vært brukt

π Я 音 æ∞

- Det er plass til drøyt 1 000 000 tegn.
- 110 187 tegn er foreløbig definert.
- Mer og mer programvare (som Java og Python) støtter Unicode.

UTF-8

Hvordan lagre Unicode-tekst uten å bruke for mye plass på disk eller over nettet? **UTF-8** bruker fra 1 til 4 byte til å lagre et tegn:

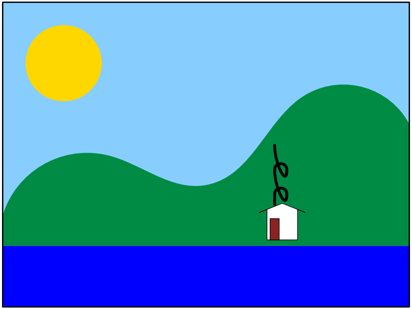
$	U+0024	00100100
¢	U+00A2	11000010 10100010
€	U+20AC	11100010 10000010 10101100
𐎄	U+10384	11110000 10010000 10001110 10000100

Unicode og UTF-8 er nå standard ved Ifi.

Bilder

Jeg har laget en vakker tegning:

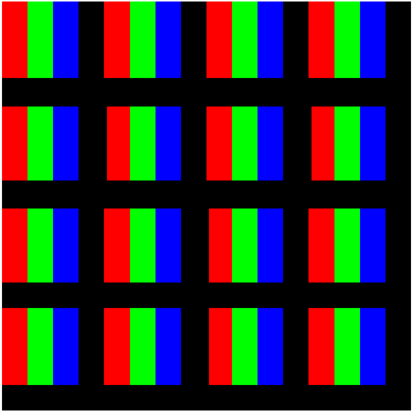
Hvordan kan jeg lagre den som bit i en datamaskin?



På en fargeskjerm består hvert bildepunkt («pixel») av tre farger

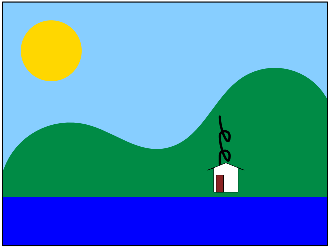
- Rød
- Grønn
- Blå

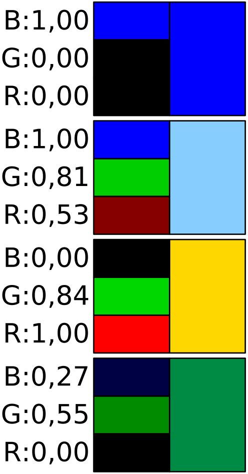
som kan lyse sterkt eller svakt.

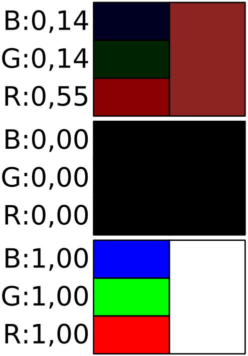


(Utskrift på papir bruker **CMYK** (Cyan, Magenta, Yellow, blacK) i stedet.)

Hvilke farger trenger vi?

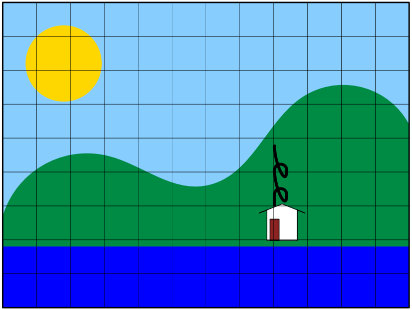






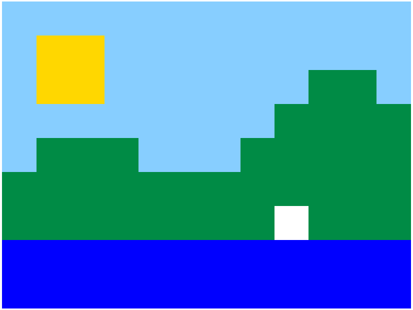
Da er det bare å legge et rutenett på bildet og notere mengden R, G og B i hver rute. Om vi bruker 1 byte til hver fargemengde, får vi:

135 206 255
135 206 255
135 206 255
:
0 0 255

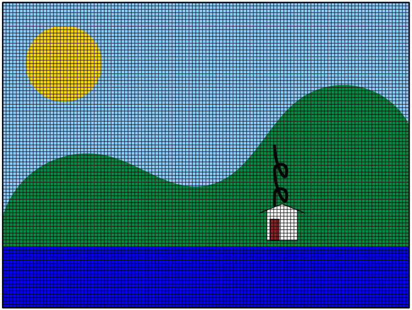


Hva er et bilde?

. . . og bildet er lagret:

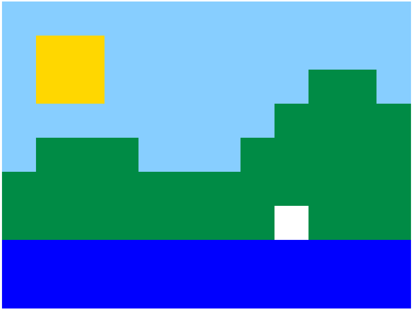


Vi bør nok prøve med et mer finmasket rutenett:



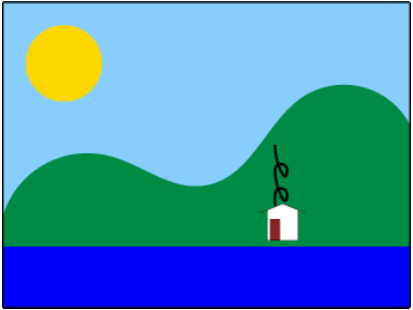
Hva er et bilde?

12×9 piksler



324 byte

348×257 piksler



363 682 byte

Kan vi lagre bildet på mindre plass?

Fargetabell

Vi bruker 3 byte = 24 bit til hvert piksel, men vi har bare 7 ulike farger i bildet. Da kan vi sette opp en tabell

0	0 0 255	mørkeblå
1	135 206 255	lyseblå
2	255 214 0	gul
3	0 140 69	skoggrønn
4	140 36 36	brun
5	0 0 0	svart
6	1 1 1	hvit

og da trenger vi bare 3 bit per pixel.

$$363\,682 \text{ byte} \Rightarrow 45\,483 \text{ byte}$$

«Run-length»-koding

I mange rasterbilder er det ofte mange like piksler på rad. Da kan vi lagre fargen og hvor mange slike piksler det er på rad.

363 682 byte ⇒ 45 483 byte ⇒ 2 917 byte

Formater som PNG og GIF bruker slike teknikker.

Og om vi er villig til å godta en reduksjon i kvaliteten?

For fotografier gjelder dette:

- På fotografier er det sjelden brå overganger.
- Vi mennesker kan bare skjelne et begrenset antall nyanser.
- Vi søker automatisk etter mønstre.

JPEG-formatet

JPEG benytter dette til å lage en forenklet versjon av bildet.

Ekte rasterbilde	40,7 MB
JPEG 100%	5,5 MB
JPEG 50%	0,94 MB
JPEG 25%	0,61 MB
JPEG 10%	0,34 MB
JPEG 5%	0,24 MB

(Dette er komprimering med **tap**. Det er umulig å komme tilbake til det opprinnelige bildet.)

Originalbildet 100%

Kvalitet 50%

Kvalitet 25%

Kvalitet 10%

Kvalitet 5%

En sammenligning

Kvalitet 100%

Kvalitet 5%

Vektorgrafikk

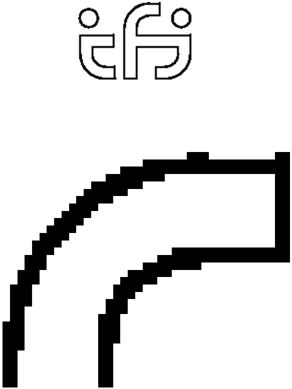
Det er også mulig å lagre bilder som linjer og kurver:

ifi-logo.eps

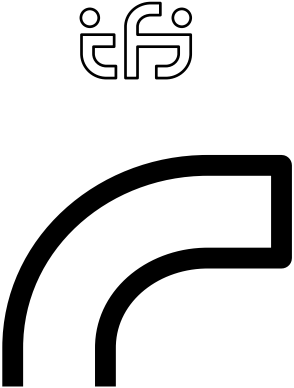
```
   :
newpath
38.4094 2.83464 moveto
38.4094 15.44878 lineto
28.3464 15.44878 lineto
21.03242 15.44878 14.93857
21.1286 14.93857 28.3464 curveto
14.93857 35.77315 lineto
36.56685 35.77315 lineto
36.56685 48.38728 lineto
2.32443 48.38728 lineto
2.32443 28.3464 lineto
2.32443 14.16295 14.0667
2.83464 28.3464 2.83464 curveto
closepath stroke
   :
```

Men hva med vektorer?

Raster



Vektor



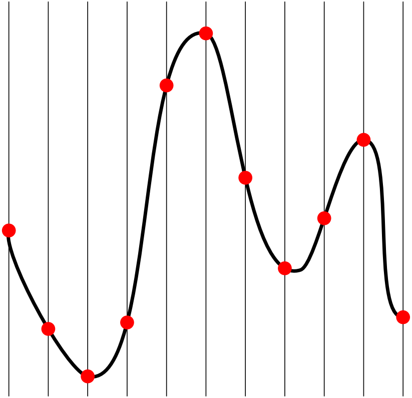
Anbefalinger

- Bruk vektorgrafikk om mulig: SVG, EPS, PDF.
- For fotografier bruk JPEG i så god kvalitet som mulig.
- For annen rastergrafikk bruk PNG med så mange piksler som mulig.

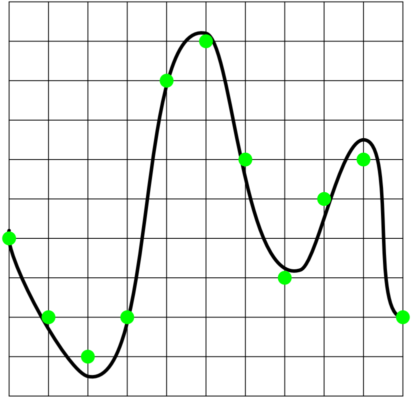
Hvordan lagre lyd

Lyd er bølger i luft, men de kan overføres som strøm:

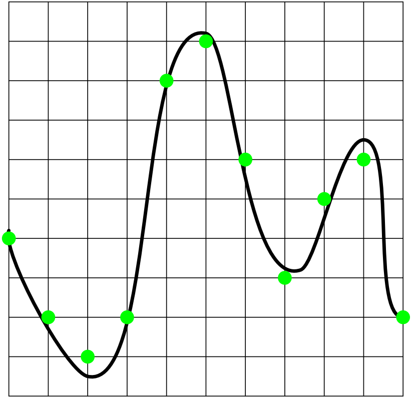
Vi kan lagre lyden ved å måle strømmen med jevne mellomrom:



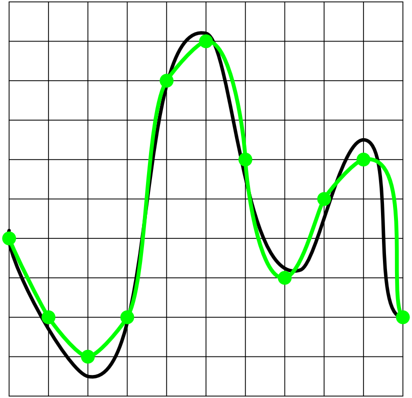
Men for å lagre digitalt må vi måle styrken i faste intervaller:



Med disse målingene



kan vi gjenskape lyden (men ikke helt nøyaktig):



Kvaliteten på lyden blir da avhengig av

- hvor ofte vi måler
- hvor mange trinn vi benytter til målingen

CD

En vanlig CD har 74 minutter spilletid med god lyd:

- 44 100 målinger («samples») per sekund
- $2^{16} = 65\,536$ intervaller (dvs 2 byte)
- 2 kanaler
- + 37% feilkorreksjonsdata

En CD må derfor ha plass til 783 MB.

Er det mulig å spare plass?

- Høyre og venstre kanal er stort sett nesten like. Det er lurere å lagre venstre kanal samt forskjellen.
- I stedet for å lagre hver måling med sin verdi, holder det å lagre forskjellen.

  Men: På grunn av feil og spoling må man av og til lagre den ekte verdien.

Enda mer plass kan vi spare om vi tar hensyn til hvordan vi mennesker hører:

- Vi kan ikke høre lyder under 20 Hz og over 20 000 Hz.
- Om vi hører en kraftig lyd med én frekvens, hører vi ikke litt svakere lyder med noe høyere frekvens.
- Etter å ha hørt en sterk lyd, hører vi dårligere en tid etterpå (inntil 0,2 s).

Moderne standarder som

MP2 (brukt i DAB)

MP3

AAC (brukt i DAB+, YouTube, iTunes, . . .)

utnytter dette og tillater til dels sterk komprimering:

Ekte CD	1410 kb/s
MP2	ca 256 kb/s
MP3	ca 160 kb/s
AAC	ca 128 kb/s

men kvaliteten går ned om det komprimeres ennå mer.

Oppsummering

- Alt er bit i en datamaskin.
- Det finnes ulike typer tallverdier, og programmereren må velge riktig.
- Det er mange ulike tegnkodinger å forholde seg til (ennå).
- Rasterbilder og vektorbilder er nyttige til hvert sitt formål.
- Lydkoding med MP2, MP3 og AAC er blitt standard, men vi bør velge rett kvalitet.