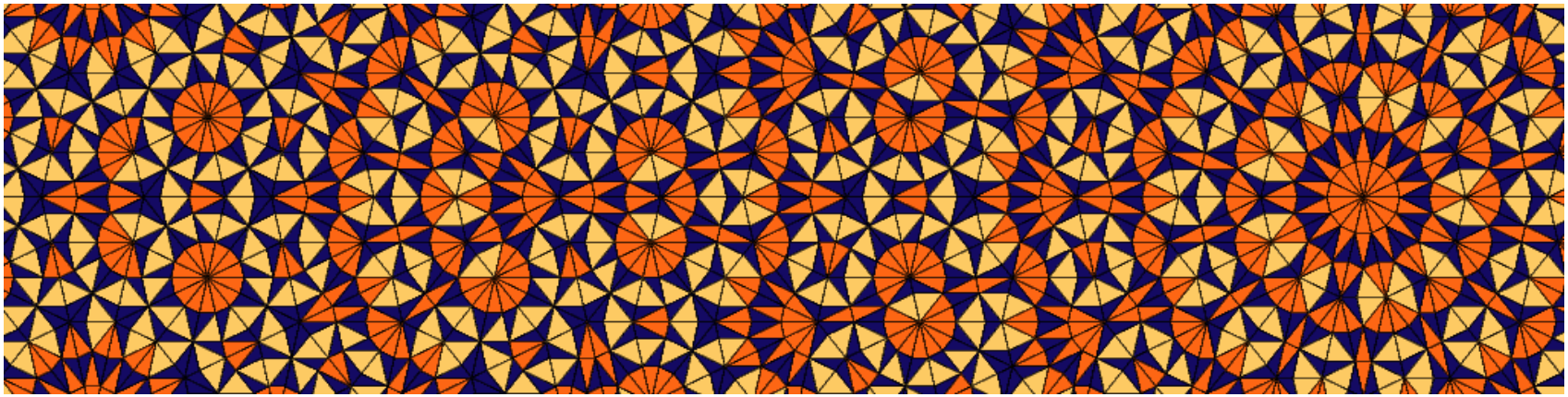
Mathematische Methoden der Biowissenschaften III

DFT, FFT, jpeg und schnelles Multiplizieren

Dirk Frettlöh
Technische Fakultät

12.2.2016

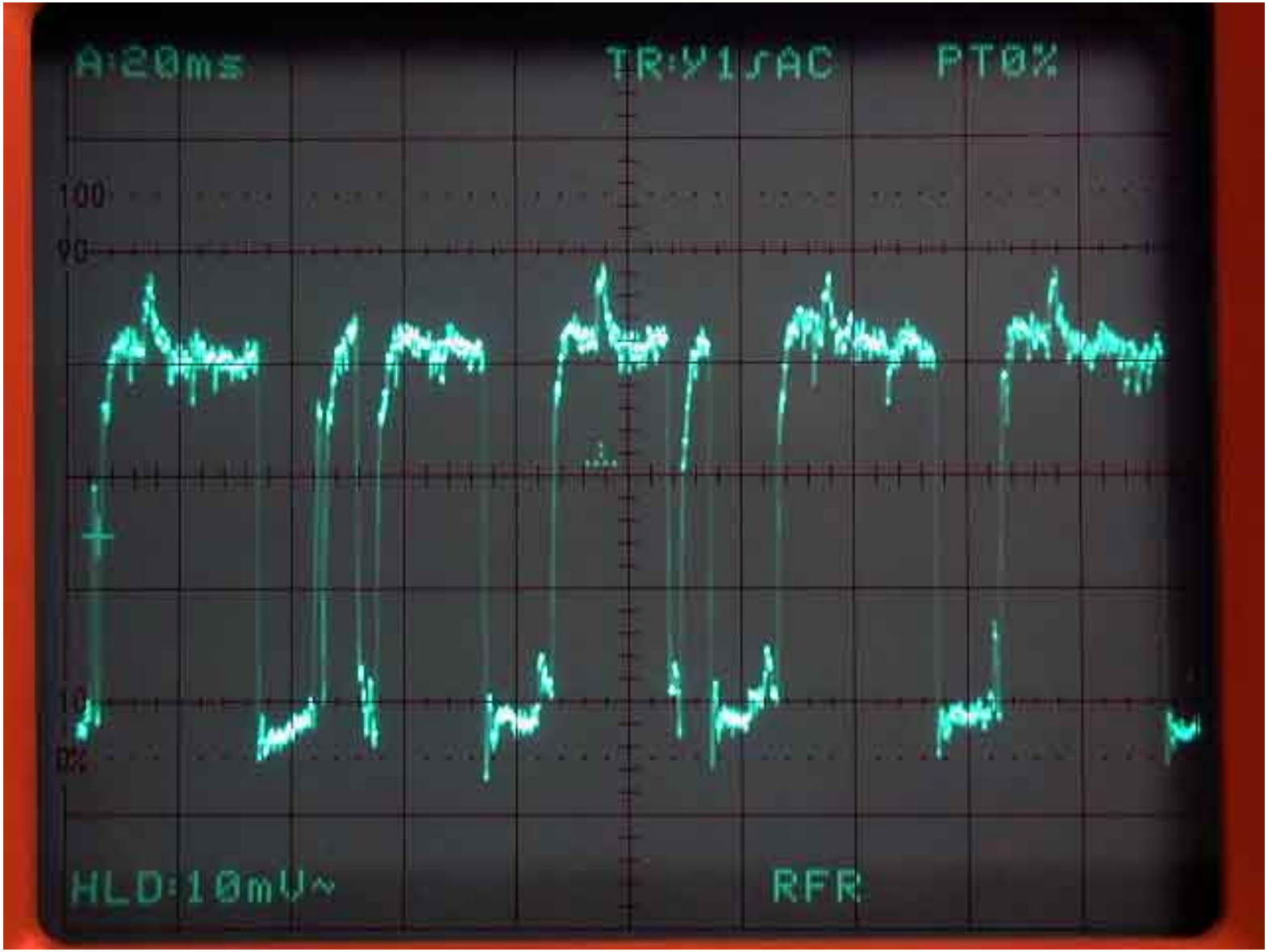


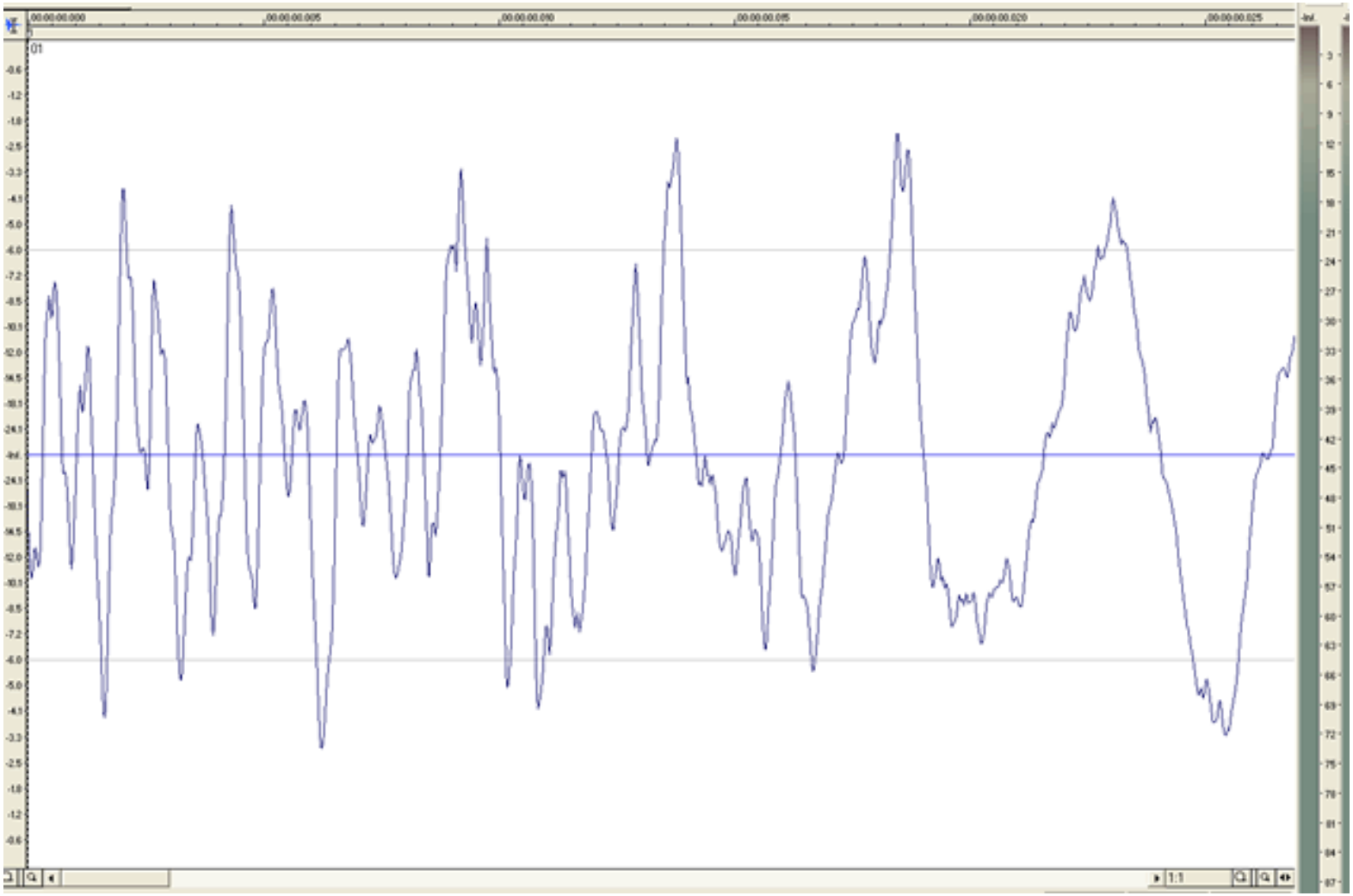
Nichts ist praktischer als eine gute Theorie.

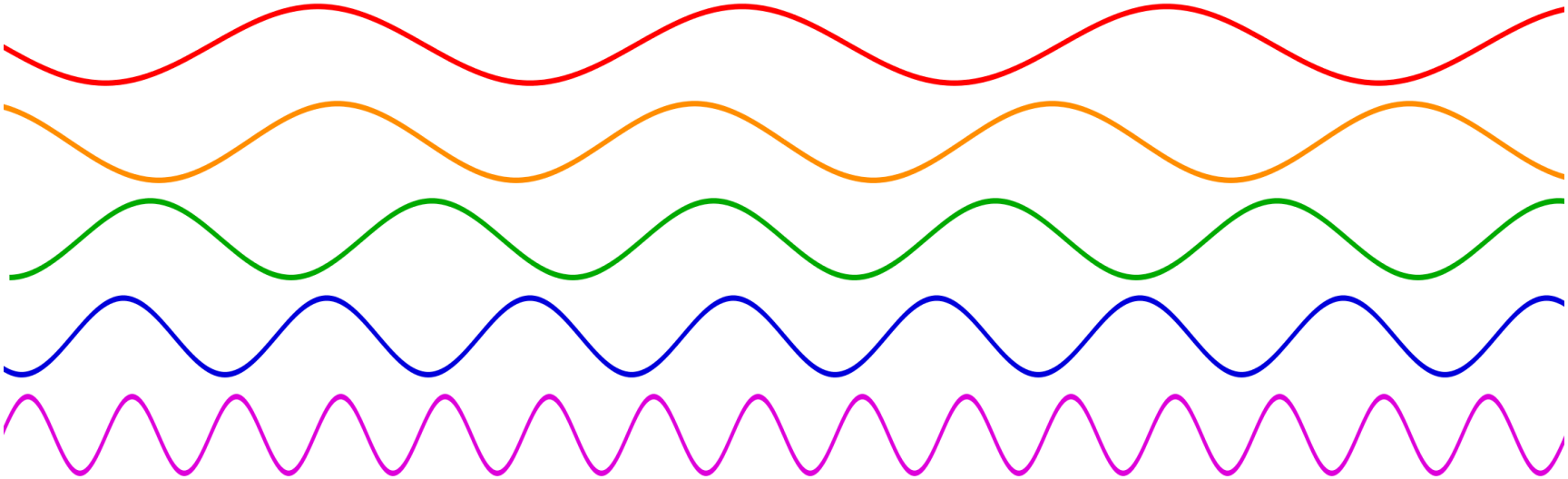
Bob der Baumeister

- Von Frequenzanalyse zu FT
- DFT, Schnelle FT ("FFT")
- Bildkompression mit jpeg
- Schnelle Multiplikation

Inspiration zu jpeg: Frequenzanalyse von Signalen.



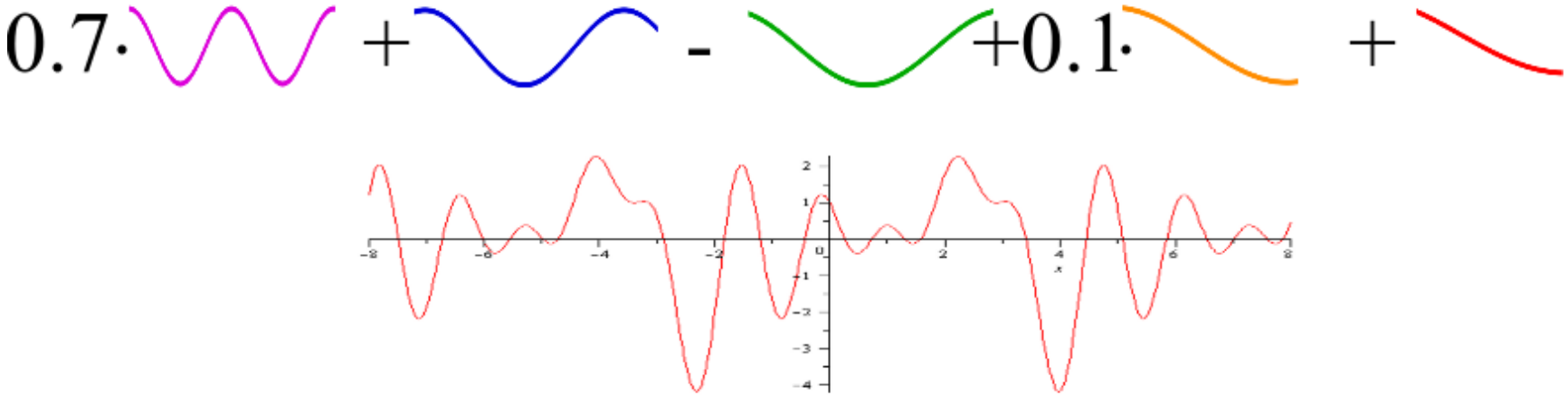




$\sin(x)$, $\sin(2x)$, $\sin(3x)$, $\cos(4x)$, $\sin(5x)$,

Ziel: Signal als Kombination "reiner" Töne darstellen. Reine Töne entsprechen Sinus- und Kosinusschwingungen $\sin(nx)$ bzw $\cos(nx)$ ($n \in \mathbb{N}$), also mit Periode $\frac{2\pi}{n}$.

Beispiel: Ein Signal als Kombination reiner Töne:



$$f(x) = \sin(x) + 0.1\sin(2x) - \sin(3x) + \cos(4x) + 0.7\sin(5x) + \cdots$$

Meist ist das Problem: Gegeben ein Signal f, wie bekommt man die Vorfaktoren? (hier: $1, 0.1, -1, 1, 0.7, \ldots$)

Allgemein ist das Problem also:

Wenn $f(x)$ eine gegebene Funktion ist, wie bestimmt man die a_n und b_n?

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos(nx) + b_n \sin(nx) \right)$$

Antwort: *Fourier-Reihe* von f. Koeffizienten

$$\boxed{a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(s) \cos(ns) ds \quad (n \in \mathbb{N}_0)}$$

$$\boxed{b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(s) \sin(ns) ds \quad (n \in \mathbb{N})}$$

(siehe Skript, oder Heuser: *Gewöhnliche Differentialgleichungen*)

Es kam auch dran:

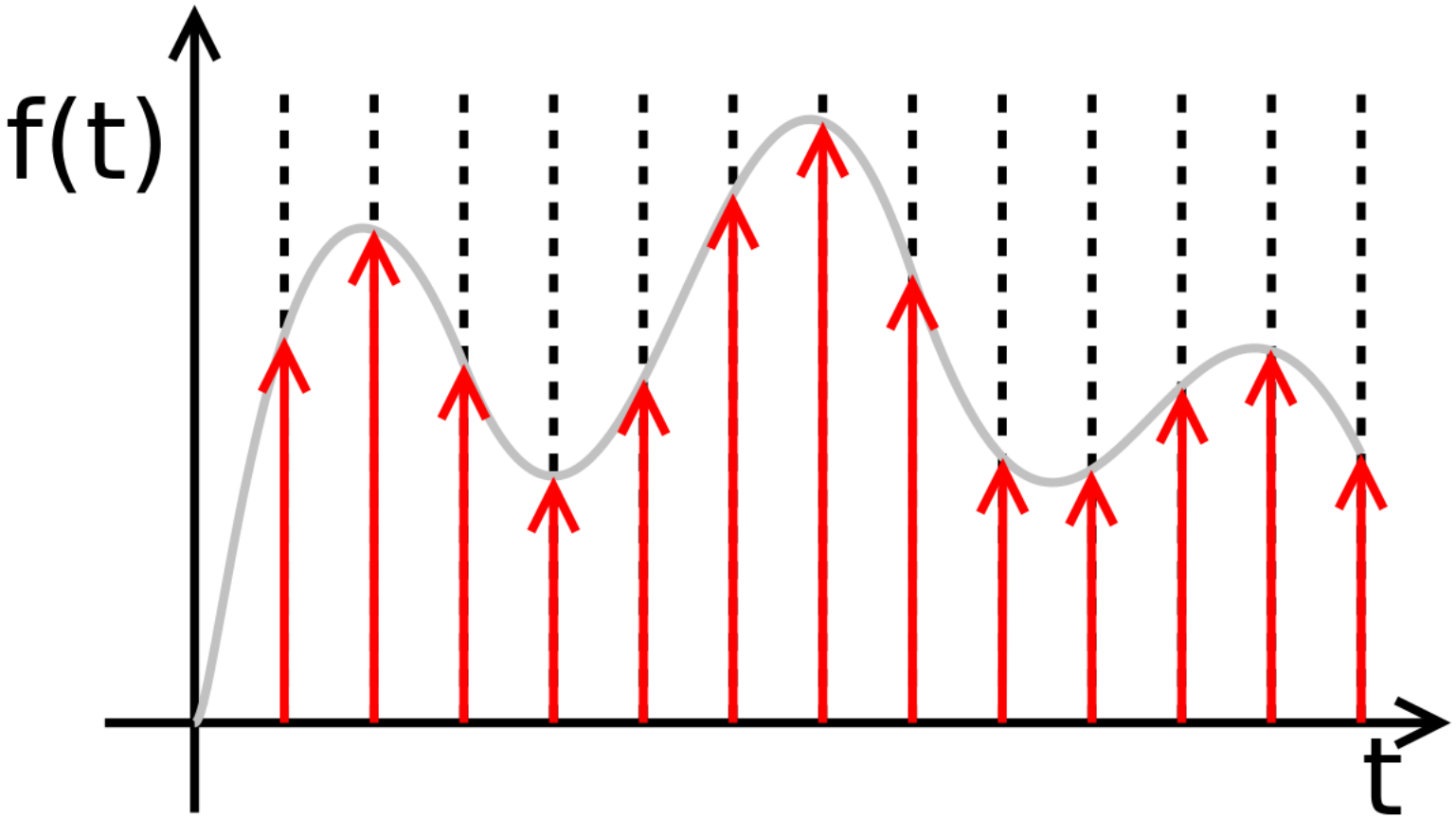
Riemann-Lebesgue-Lemma: Ist f stetig, dann gilt

$$\lim_{n\to\infty} a_n = 0, \qquad \lim_{n\to\infty} b_n = 0.$$

D.h.: nur die ersten (paar) a_n und b_n sind wichtig! Die anderen werden klein oder verschwinden. Das wird später wichtig!
Zunächst aber:

***

Computer kennen keine stetigen Funktionen und keine reellen Zahlen. Daher:



f ist beschrieben durch $f(t_0), f(t_1), f(t_2), \ldots, f(t_{N-1})$.

Diskrete Fouriertransformation (DFT)

Die *diskrete Fouriertransformation* (DFT) von $f = (f_0, f_1, \ldots, f_{N-1})$ ist $d = (d_0, d_1, \ldots, d_{N-1})$ mit

$$d_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j \cdot \left(\cos(-\frac{2\pi}{N} jk) + i \sin(-\frac{2\pi}{N} jk) \right) \quad (k = 0, 1, \ldots, N-1)$$

Das sind Vektoren!
Daher kann man die DFT mittels dieser Matrix ausrechnen (hier ist $\xi = e^{2\pi i/N} = \cos(\frac{2\pi}{N}) + i \sin(\frac{2\pi}{N})$).

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-1} \end{pmatrix} = \frac{1}{N} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \xi^{-1} & \xi^{-2} & \cdots & \xi^{-(N-1)} \\ 1 & \xi^{-2} & \xi^{-4} & \cdots & \xi^{-2(N-1)} \\ 1 & \xi^{-3} & \xi^{-6} & \cdots & \xi^{-3(N-1)} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & \xi^{-(N-1)} & \xi^{-2(N-1)} & \cdots & \xi^{-(N-1)^2} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

Fast Fourier Transform (FFT)

Insbesondere falls $N = 2^k$ (oder $N \approx 2^k$) ist es günstig, die DFT nicht mittels der Matrix zu berechnen (Aufwand $O(n^2)$), sindern mit einem Divide-and-Conquer-Ansatz.

Theorem

Die DFTs von $(f_0, f_1, \ldots, f_{N-1})$ *und* $(f_N, f_{N+1}, \ldots, f_{2N-1})$ *liefern die DFT von* $(f_0, f_N, f_1, f_{N+1}, f_2, f_{N+2}, \ldots, f_{N-1}, f_{2N-1})$ *(Länge 2N) so: Mit*

$$d_k = \frac{1}{2}\left(DFT(f_0, f_1, \ldots, f_{N-1}) + e^{-\pi ik/N} DFT(f_N, f_{N+1}, \ldots, f_{2N-1})\right)_k$$

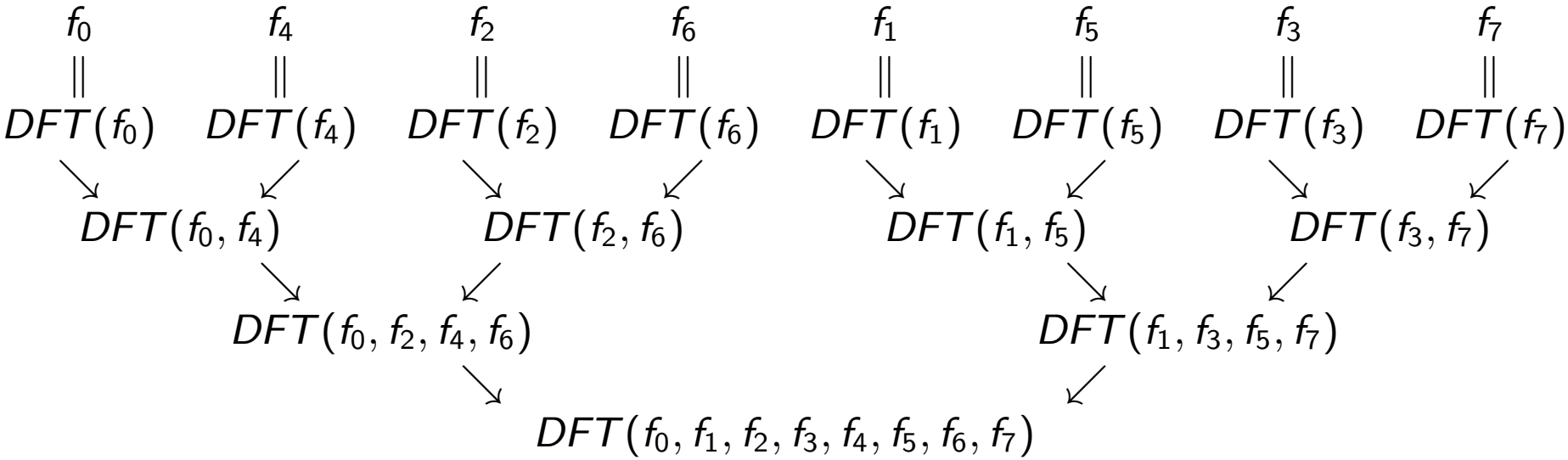
$$d_{N+k} = \frac{1}{2}\left(DFT(f_0, f_1, \ldots, f_{N-1}) - e^{-\pi ik/N} DFT(f_N, f_{N+1}, \ldots, f_{2N-1})\right)_k$$

(wobei $0 \leq k \leq N-1$*) ist dann*

$$DFT(f_0, f_N, f_1, f_{N+1}, f_2, f_{N+2}, \ldots, f_{N-1}, f_{2N-1}) = (d_0, d_1, \ldots, d_{2N-1})$$

.

Dieser Satz liefert einen Divide-and-Conquer-Algorithums zum Berechnen der DFT in $O(n \log n)$ Schritten. (Hier nur für $N = 2^k$.) Dazu muss das Problem aufgeteilt werden in das Berechnen zweier DFTs, die dann wieder aufgeteilt werden in 2 mal 2 usw.; bis jeweils N Stück DFTs der Länge 1 berechnet werden müssen. Die müssen dann in der richtigen Reihenfolge kombiniert werden. Hier das Schema (für $N = 8$):



Das einzige Problem ist nun, wie bringen wir die f_n in die richtige Anfangsreihenfolge? Die richtige Reihenfolge erhalten wir einfach durch Bitumkehr:

$$000 \to 000,\ 001 \to 100,\ 010 \to 010,\ 011 \to 110,\ 100 \to 001,\ \textit{usw}$$

Also (im Falle $N = 8$) muss an der 0-ten Stelle f_0 stehen, an der ersten Stelle f_4, an der zweiten f_2 usw. Natürlich hängt die Bitumkehr von $N = 2^k$ ab. Für $N = 16$ ergibt sich etwa

$$0000 \to 0000,\ 0001 \to 1000,\ 0010 \to 0100,\ 0011 \to 1100,\ 0100 \to 0010,\ \textit{usw}$$

Algorithmus FFT:

Sei $N = 2^q$. Sei $g = (g_0, g_1, \ldots, g_{N-1})$ der Vektor, den man durch Umnummerierung mittels Bitumkehr aus $f = (f_0, f_1, \ldots, f_{N-1})$ erhält.
Starte mit den Vektoren der Länge 1: $d^{[0,j]} = (g_j)$ $(j = 0, \ldots, N-1)$. Berechne in Schritt r $(r \geq 1)$ die 2^{q-r} Vektoren der Länge 2^r

$$d^{[r,0]}, d^{[r,1]}, \ldots, d^{[r,2^{q-r}-1]}.$$

aus den Vektoren im $r-1$-ten Schritt gemäß

$$d_k^{[r,j]} = \frac{1}{2}\left(d_k^{[r-1,2j]} + e^{-\pi ik/2^{r-1}} d_k^{[r-1,2j+1]}\right) \quad (1)$$

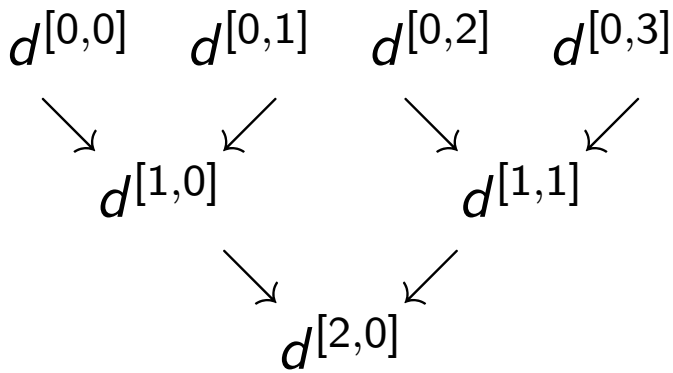
$$d_{2^{r-1}+k}^{[r,j]} = \frac{1}{2}\left(d_k^{[r-1,2j]} - e^{-\pi ik/2^{r-1}} d_k^{[r-1,2j+1]}\right) \quad (2)$$

Dabei läuft (innerste Schleife) $k = 0, 1, \ldots, 2^{r-1} - 1$, und (zweitinnerste Schleife) $j = 0, 1, \ldots, 2^{q-r} - 1$, sowie $r = 1, 2, \ldots, q$.

Beispiel: Hier ein (sehr einfaches) Beispiel für $N = 4$: Wir berechnen die DFT für den Vektor (Datensatz) $f = (8, -4, -8, 16)$. Bitumkehr liefert

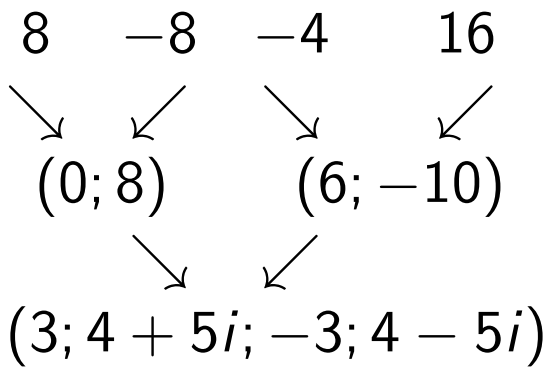
$$g_0 = f_0 = 8,\ g_1 = f_2 = -8,\ g_2 = f_1 = -4,\ g_3 = f_3 = 16$$

Das Schema ist



Dabei sind $d^{[1,0]}$ und $d^{[1,1]}$ Vektoren der Länge 2 (also $d^{[1,0]} = (d_0^{[1,0]}, d_1^{[1,0]})$ usw) und $d^{[2,0]}$ ist ein Vektor der Länge 4.

Die konkreten Werte:



Dabei berechnet sich z.B. $d^{[1,0]} = (d_0^{[1,0]}, d_1^{[1,0]})$ so:

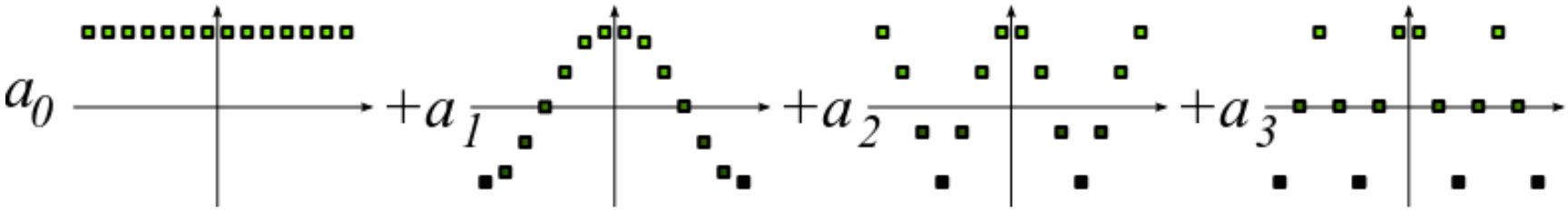
$$d_0^{[1,0]} = \frac{1}{2}(d_0^{[0,0]} + e^{-\pi i 0} d_0^{[0,1]}) = \frac{1}{2}(8 - 8) = 0$$
$$d_1^{[1,0]} = \frac{1}{2}(d_0^{[0,0]} - e^{-\pi i 0} d_0^{[0,1]}) = \frac{1}{2}(8 - (-8)) = 8,$$

...und $d^{[2,0]} = (d_0^{[2,0]}, d_1^{[2,0]}, d_2^{[2,0]}, d_3^{[2,0]})$ aus $d^{[1,0]} = (0, 8)$ und $d^{[1,1]} = (6, -10)$ so:

$$
\begin{aligned}
d_0^{[2,0]} &= \frac{1}{2}(d_0^{[1,0]} + e^{-\pi i 0} d_0^{[1,1]}) = \frac{1}{2}(0 + 6) = 3 \\
d_1^{[2,0]} &= \frac{1}{2}(d_1^{[1,0]} + e^{-\pi i/2} d_1^{[1,1]}) = \frac{1}{2}(8 - i(-10)) = 4 + 5i, \\
d_2^{[2,0]} &= \frac{1}{2}(d_0^{[1,0]} - e^{-\pi i 0} d_0^{[1,1]}) = \frac{1}{2}(0 - 6) = -3 \\
d_3^{[2,0]} &= \frac{1}{2}(d_1^{[1,0]} - e^{-\pi i/2} d_1^{[1,1]}) = \frac{1}{2}(8 + i(-10)) = 4 - 5i.
\end{aligned}
$$

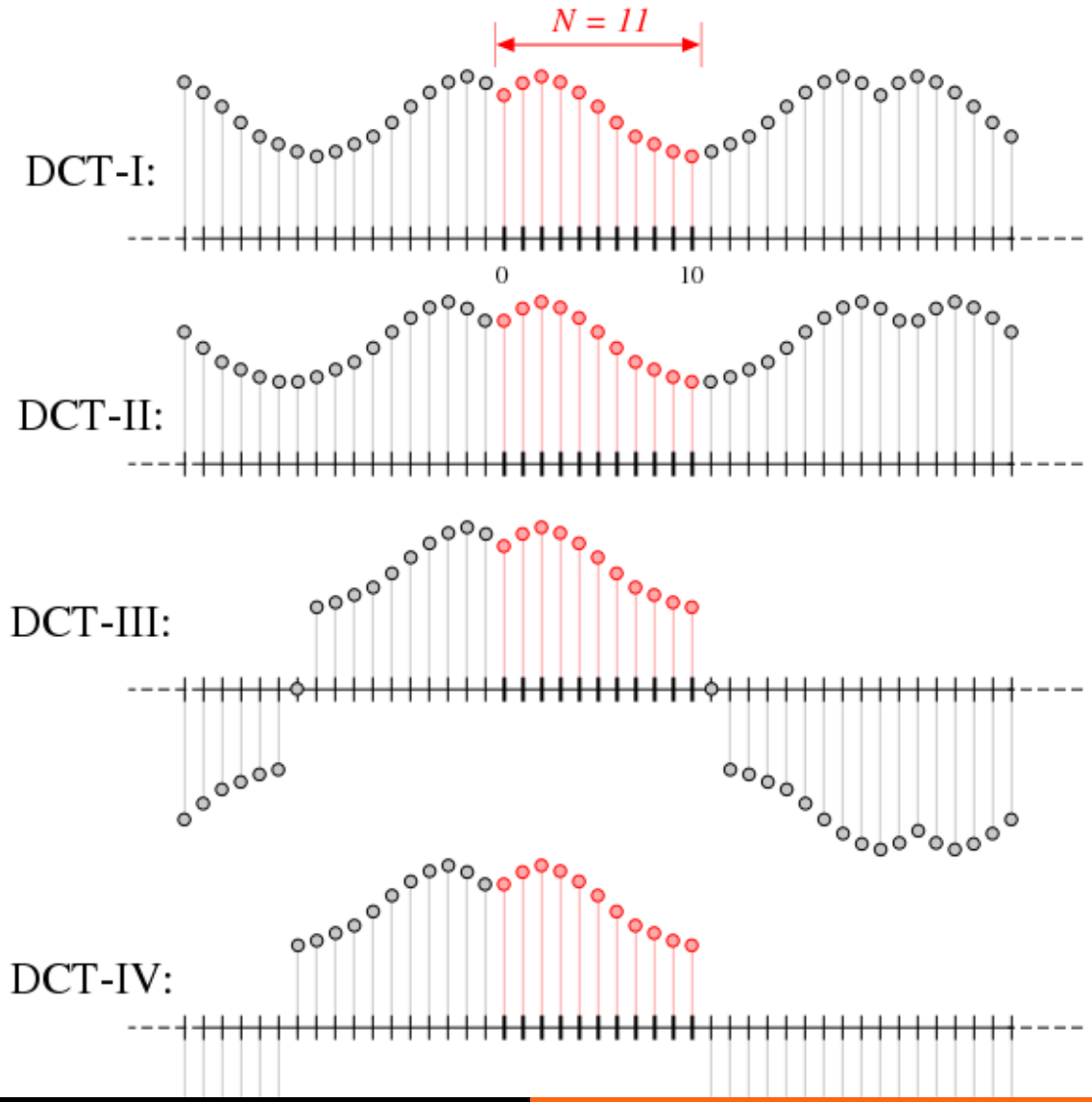
Anwendung: Bildkompression

Gleiche Idee wie bei Fourierreihen: Diskretes Signal (Ton, Bildzeile...) darstellen als Summe von Vektoren (mit Kosinussen, "Diskrete Kosinustransformation" (DCT))



Die ersten paar (hier: 4) Terme liefern eine gute Näherung. Speichere also nur 4 Werte statt 14. Rechtfertigung: Riemann-Lebesgue-Lemma.

Es gibt für DCT verschiedene Wahlmöglichkeiten: Punkte $0, \frac{1}{N}, \frac{2}{N}, \ldots \frac{N-1}{N}, 1$, oder je um $\frac{1}{2N}$ verschoben. Sowie: soll es nach rechts gerade oder ungerade weitergehen.



DCT I: $$a_k = \frac{1}{2}(f_0+(-1)^k f_{N-1})+\sum_{n=1}^{N-2} f_n \cos\left(\frac{\pi}{N-1} nk\right) \qquad k = 0,\ldots,N-1.$$

DCT II: $$a_k = \sum_{n=0}^{N-1} f_n \cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right) \qquad k = 0,\ldots,N-1.$$

DCT III: $$a_k = \frac{1}{2} f_0 + \sum_{n=1}^{N-1} f_n \cos\left(\frac{\pi}{N} n\left(k+\frac{1}{2}\right)\right) \qquad k = 0,\ldots,N-1.$$

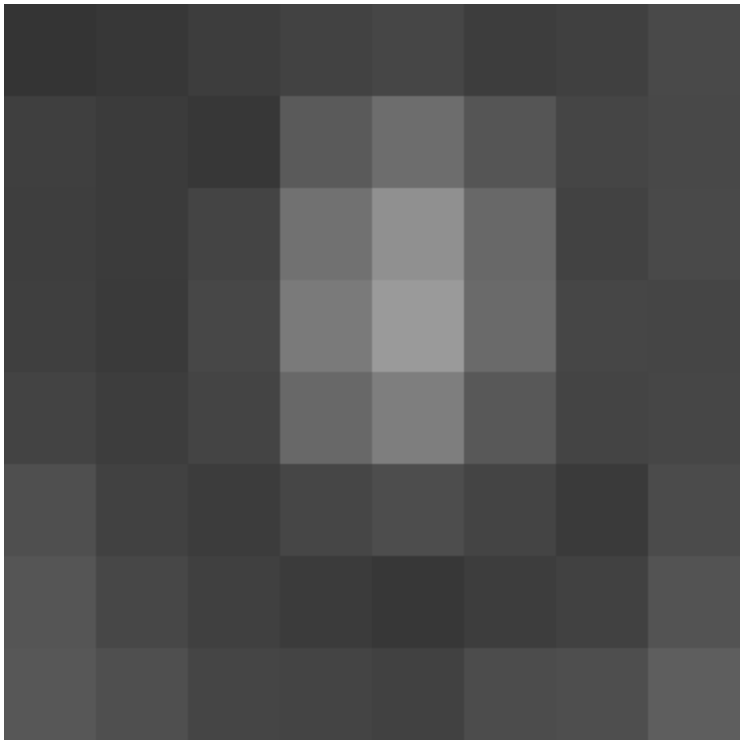
DCT IV: $$a_k = \sum_{n=0}^{N-1} f_n \cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)\left(k+\frac{1}{2}\right)\right) \qquad k = 0,\ldots,N-1.$$

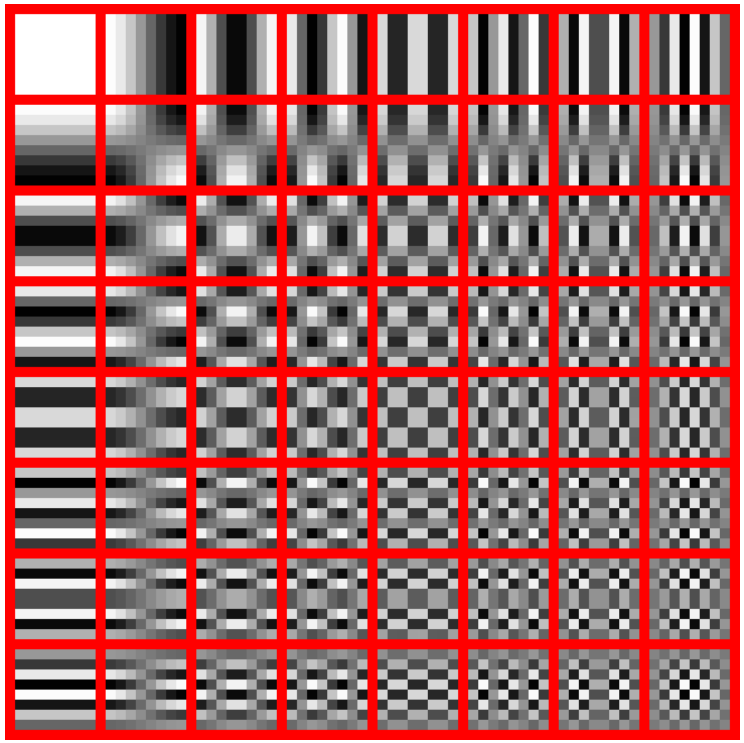
...eigentlich brauchen wir aber gleich die zweidimensionale DCT:

$$
\begin{aligned}
a_{k_1,k_2} &= \sum_{n_1=0}^{N_1-1}\left(\sum_{n_2=0}^{N_2-1} f_{n_1,n_2}\cos\left[\frac{\pi}{N_2}\left(n_2+\frac{1}{2}\right)k_2\right]\right)\cos\left[\frac{\pi}{N_1}\left(n_1+\frac{1}{2}\right)k_1\right] \\
&= \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} f_{n_1,n_2}\cos\left[\frac{\pi}{N_1}\left(n_1+\frac{1}{2}\right)k_1\right]\cos\left[\frac{\pi}{N_2}\left(n_2+\frac{1}{2}\right)k_2\right].
\end{aligned}
$$

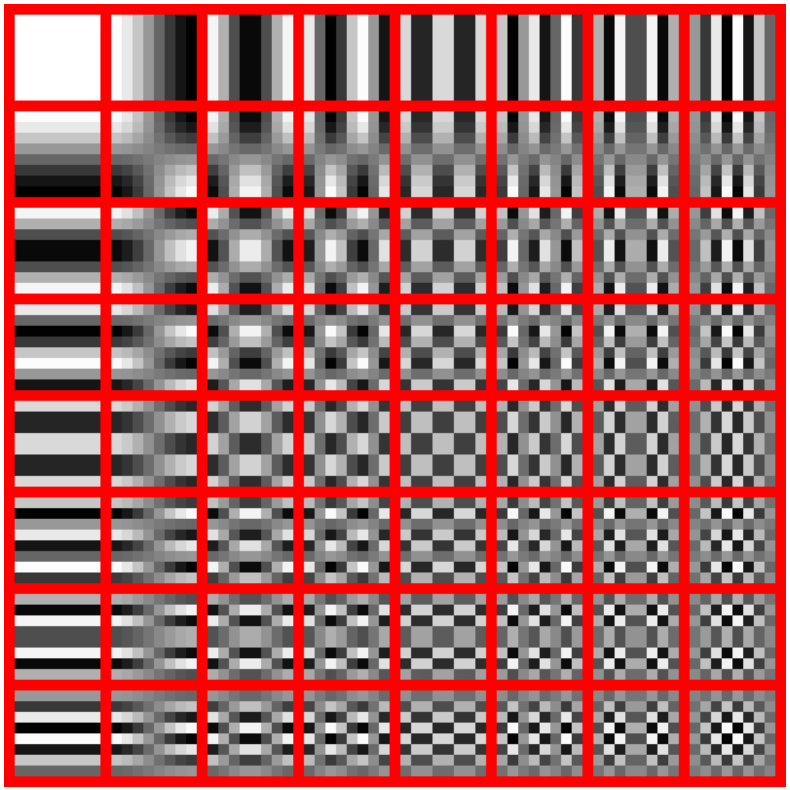
Funktionsweise jpeg:

- Bearbeite jeden der drei Farbwerte einzeln (RGB bzw. YCbCr)
- Unterteile das Bild in 8×8 Felder (u. links: ein solches Feld)
- 2D DCT für jedes einzelne Feld (Linearkombination der 64 8×8-Felder unten rechts)
- Quantisieren der DCT (übernächste Folie)
- Run-length coding, dann Huffman coding





DCT: Die Matrix rechts zeigt den Vorfaktor für das entsprechende Feld links ("wie viel" vom entsprechenden Feld links wir hinzuaddieren)



$$\begin{bmatrix} -415 & -30 & -61 & 27 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -3 & 4 & -1 \\ 0 & 0 & -1 & -1 & 0 & 1 & 2 \end{bmatrix}$$

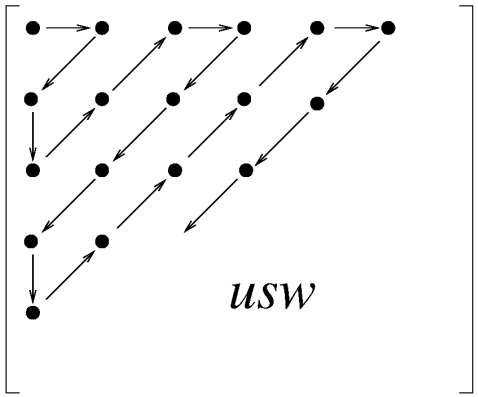
Quantisieren:

$$\begin{bmatrix} -415 & -30 & -61 & 27 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -3 & 4 & -1 \\ 0 & 0 & -1 & -1 & 0 & 1 & 2 \end{bmatrix}$$ mit $$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

runden liefert

$$\begin{bmatrix} -26 & -3 & -6 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Speichere das in dieser Reihenfolge



Dann **Run-length-coding** (Blöcke von 0en!), dann **Huffman-coding** (häufige Zeichen mit wenig Bits)
Klingt kompliziert. Klappt sehr gut:

Komprimierungsfaktor $0,38$

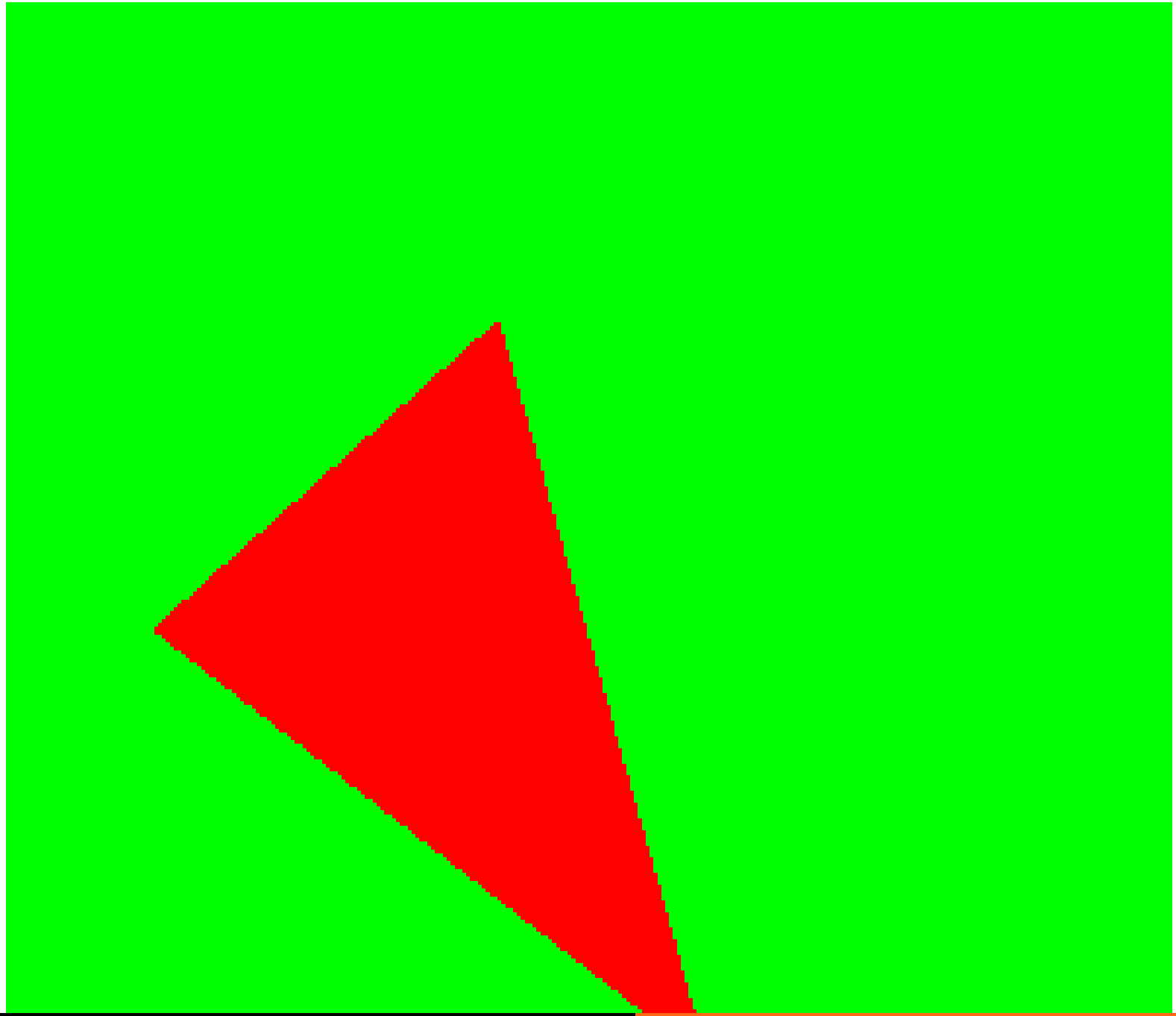
Komprimierungsfaktor $0,07$

Komprimierungsfaktor $0,04$

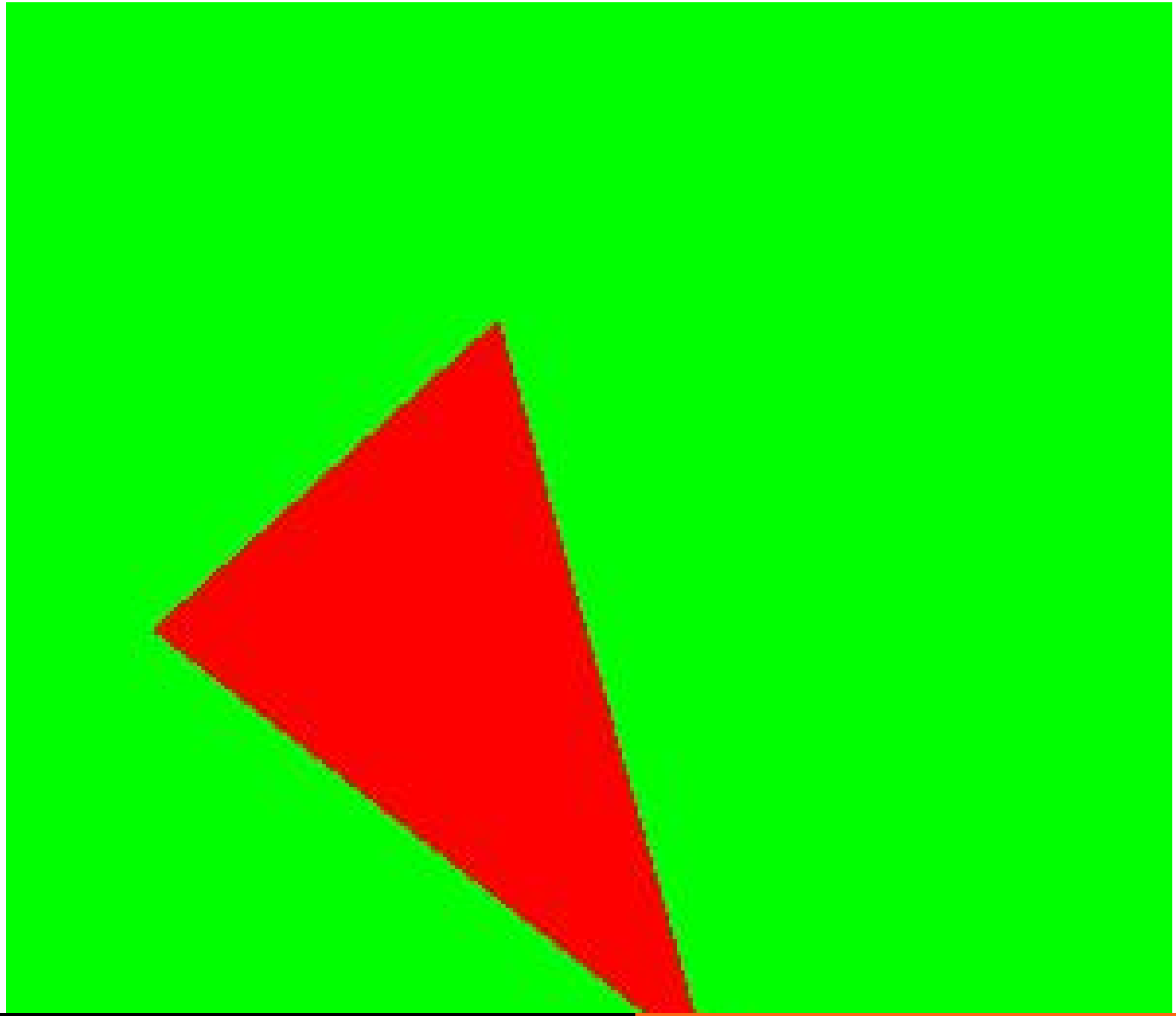
Komprimierungsfaktor $0,02$

Komprimierungsfaktor $0,007$

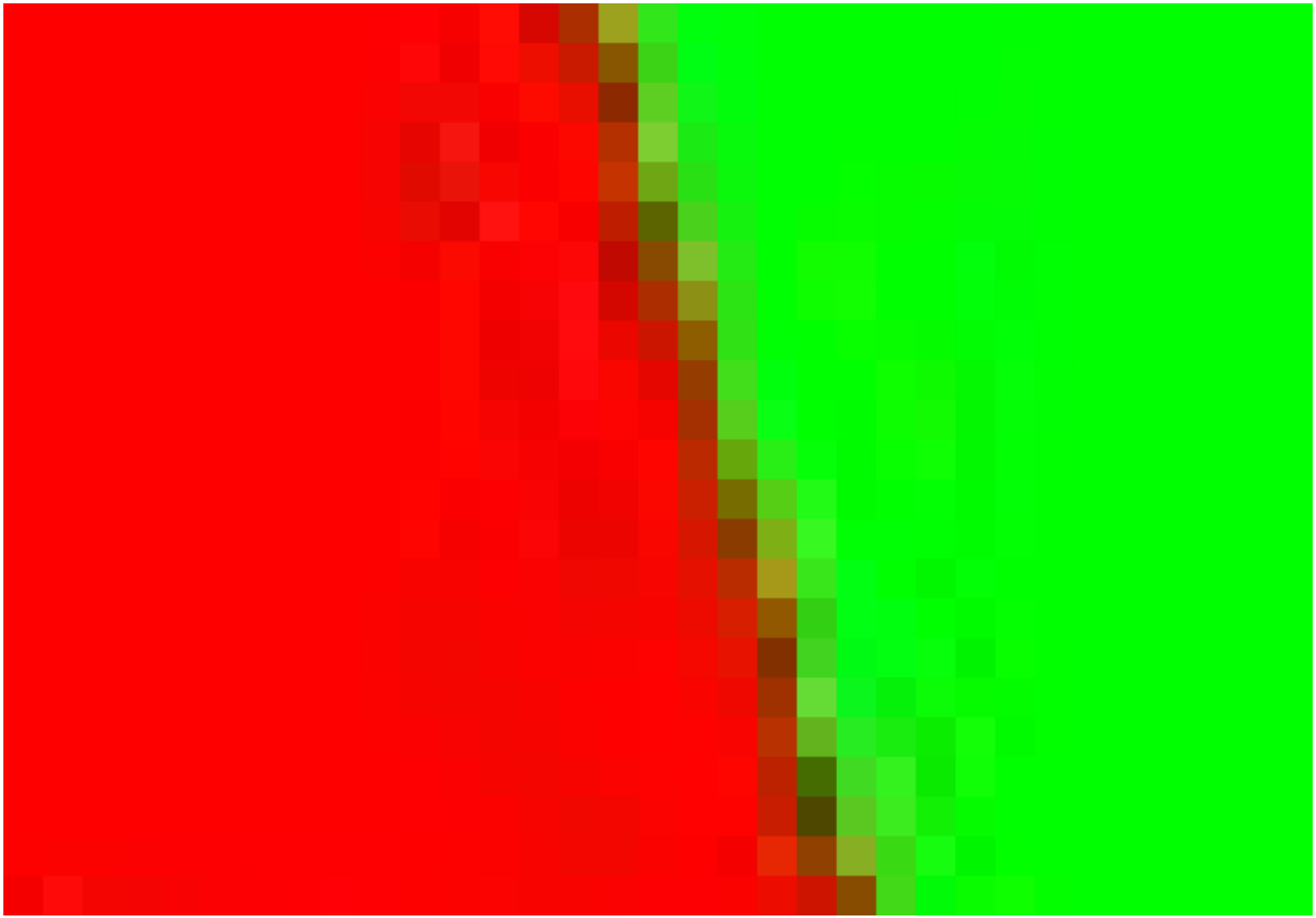
jpeg ist nicht gut bei scharfen Kanten und wenigen Farben. png:



Dasselbe Bild als jpeg:

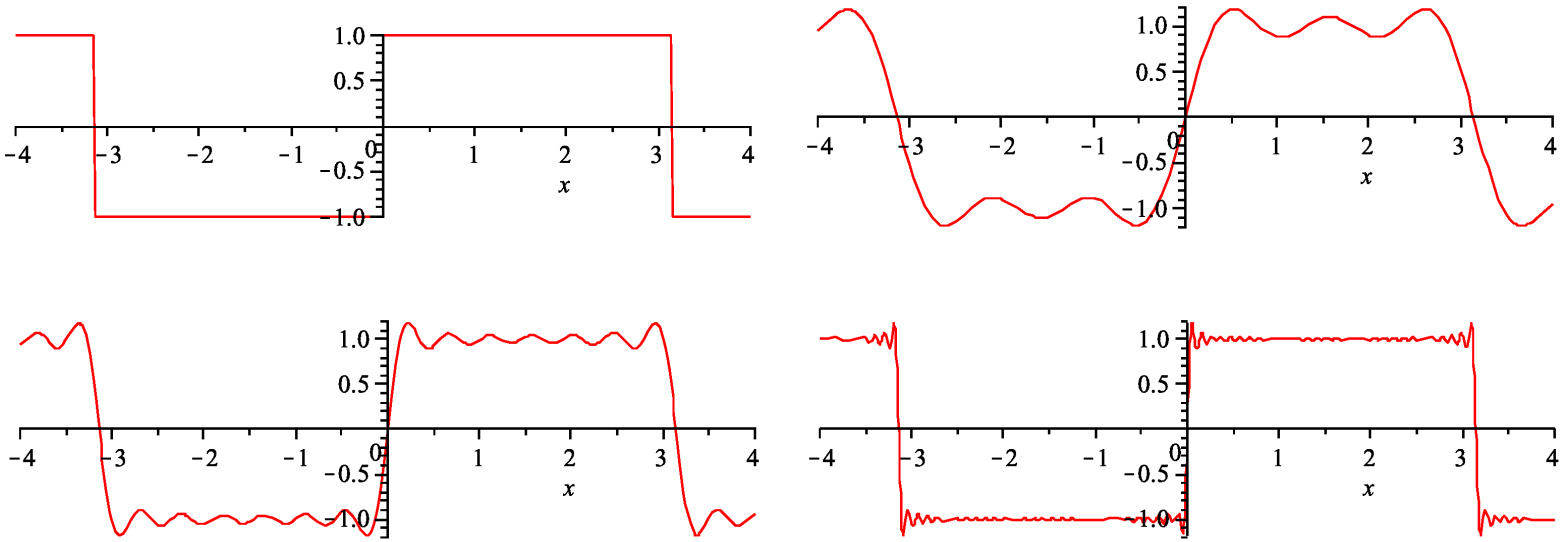


Ausschnittsvergrößerung desselben Bilds:



Gibbssches Phänomen:

Das kann man mittels der Theorie der Fourierreihen verstehen:



Oben links: Originalsignal f (Rechteckkurve), oben rechts: die ersten vier Terme der Fourierreihe.
Unten links: die ersten 16 Terme, unten rechts: die ersten 30 Terme.
Egal wie viele Terme man hinzunimmt, es gibt (beweisbar) immer einen Ausschlag um ca 15% zu weit nach oben.

Eine weitere Anwendung der schnellen Fouriertransformation:
Multiplizieren zweier Zahlen mit n Stellen in Zeit $O(n \log n)$.

Normaler Algorithmus:

	123	·	456	
		6	12	18
	5	10	15	
4	8	12		
5	6	0	8	8

Laufzeit?
3 mal 3 Multiplikationen, 5 Additionen, bzw. allgemein:

n^2 Multiplikationen und ca. $2n - 1$ Additionen

Zur einfacheren Darstellung des über 40 Jahre alten und immer noch besten[1] Algorithmus von **Schönhage-Strassen**.

Dazu zunächst Mal:

Multiplikation von Polynomen:

$$
\begin{aligned}
&(x^3 + x^2 + x + 1)(x^3 + x^2 + 1) \\
&= x^6 + x^5 + x^4 + x^3 + x^5 + x^4 + x^3 + x^2 + x^3 + x^2 + x + 1 \\
&= x^6 + 2x^5 + 2x^4 + 3x^3 + 2x^2 + x + 1
\end{aligned}
$$

[1] seit 2007 gibt es theoretisch schneller, aber dieser ist der Standard für alles ab ca 50 000 Dezimalstellen.

Schreiben wir die Polynome als Koeffizientenvektoren, wird aus $1 + x + x^2 + x^3$ mal $1 + 0 \cdot x + x^2 + x^3$ gleich $1 + x + 2x^2 + 2x^2 + 3x^3 + 2x^4 + 2x^5 + x^6$ dieses:

$$\begin{aligned} f \otimes g &= (1,1,1,1,0,0,0,0) \otimes (1,0,1,1,0,0,0,0) \\ &= (1,1,2,3,2,2,1,0) = h. \end{aligned}$$

Was ist das $\otimes$? Sind f und g die Koeffizientenvektoren, dann ist Eintrag Nummer 0 von h (also h_0) gleich $g_0 \cdot f_0$. Weiter ist $h_1 = f_0 \cdot g_1 + f_1 \cdot g_0$, $h_2 = f_0 \cdot g_2 + f_1 \cdot g_1 + f_2 \cdot g_0$ usw.
Allgemein ist Eintrag Nummer n von $f \otimes g$:

$$h = f \otimes g, \quad h_n = \sum_{k=0}^{N-1} f_k g_{n-k} \qquad (0 \leq n \leq N-1)$$

Obacht: es tauchen $g_{-1}, g_{-2}, \ldots$ auf. Kein Problem: Vereinbaren wir $g_{-1} := g_{N-1}$, $g_{-2} := g_{N-2}$ usw für diese. Machen wir unsere Vektoren lang genug, so sorgen die vielen führenden Nullen dafür, dass kein Fehler passiert (nachprüfen!)

Also

$$h = f \otimes g, \quad h_n = \sum_{k=0}^{N-1} f_k g_{n-k} \qquad (0 \leq n \leq N-1)$$

In der (diskreten) Fouriertheorie gibt es den Begriff der Faltung:

$$f * g(n) = \frac{1}{N} \sum_{k=0}^{N-1} f_k g_{n-k} \qquad (0 \leq n \leq N-1)$$

Also ist $\otimes$ dasselbe wie "Faltung mal N". Noch besser, es gibt einen **Faltungssatz:**

$$DFT(f * g) = N \cdot DFT(f) \cdot DFT(g)$$

Was heißt das zweite "mal"? Eintragsweise malnehmen!

Fakt: Da f ein Koeffizientenvektors eines Polynoms $p(x)$ war, ist $N\cdot\mathrm{DFT}(f)$ der Vektor der Funktionswerte $(p(1), p(\xi^{-1}), p(\xi^{-2}), , \ldots, p(\xi^{-(N-1)}))$.

Setzen wir der Einfachheit halber $\zeta = \xi^{-1}$. Dann ist $N\cdot\mathrm{DFT}(f)$ der Vektor der Funktionswerte $(p(1), p(\zeta), p(\zeta^2), , \ldots, p(\zeta^{N-1}))$.

Bsp: $p(x) = 1 + 2x - x^3$.

$$\mathrm{DFT}(p) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \zeta & \zeta^2 & \zeta^3 \\ 1 & \zeta^2 & \zeta^4 & \zeta^6 \\ 1 & \zeta^3 & \zeta^6 & \zeta^9 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 1+2-1 \\ 1+2\zeta-\zeta^3 \\ 1+2\zeta^2-\zeta^6 \\ 1+2\zeta^3-\zeta^9 \end{pmatrix}.$$

Damit kann man sich überlegen:

- $N \cdot (f * g)$ ist "Polynom mal Polynom" (f mal g)
- $N \cdot \mathrm{DFT}(f * g)$ ist dann der Vektor der Funktionswerte von $f * g$, also $(f \cdot g)(1), (f \cdot g)(\zeta), \ldots (f \cdot g)(\zeta^{N-1})$. Also $f(1) \cdot g(1), f(\zeta) \cdot g(\zeta), \ldots f(\zeta^{N-1}) \cdot g(\zeta^{N-1})$.

Die DFT lässt sich auch umdrehen: IDFT. Berechnet sich fast genau wie die DFT (insbesondere auch schnell: FFT).

Matrix IDFT:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \xi^1 & \xi^2 & \ldots & \xi^{(N-1)} \\ 1 & \xi^2 & \xi^4 & \ldots & \xi^{2(N-1)} \\ 1 & \xi^3 & \xi^6 & \ldots & \xi^{3(N-1)} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & \xi^{(N-1)} & \xi^{2(N-1)} & \ldots & \xi^{(N-1)^2} \end{pmatrix}$$

Idee: Berechne

$$N \cdot (f * g) = N \cdot IDFT\big(DFT(f * g)\big) = N \cdot IDFT\big(N \cdot DFT(f) \cdot DFT(g)\big)$$
$$= N^2 \cdot IDFT\big(DFT(f) \cdot DFT(g)\big)$$

Aufwand: Insgesamt $O(N \log N)$.

$$N \cdot (f * g) = N^2 \cdot IDFT\big(DFT(f) \cdot DFT(g)\big)$$

- Drei (I)DFTs: 3 mal $O(N \log N)$ Operationen.
- Eintragsweise multiplizieren: N Operationen.
- Evtl. im Ergebnis alles nochmal "mal N^2": N Operationen.

Also können wir nun zwei Polynome (vom Grad $\leq N/2$) multiplizieren mit Aufwand $O(N \log N)$ statt $O(N^2)$.

Also auch Binärzahlen...

...denn: Statt des Koeffizientenvektors eines Polynoms betrachte den Vektor der Binärdarstellung der Länge $2N$ zweier Zahlen mit der Länge N. (Man überlege sich: wenn $(p_0, p_1, p_2, \ldots)$ die Binärdarstellung ist, und p das Polynom zu diesem Koeffizientenvektor, was ist dann $p(2)$?)

Beispiel:

$$15 = (1,1,1,1,0,0,0,0), \quad 5 = (1,0,1,0,0,0,0,0)$$

Wir wenden den Algorithmus an:

$$15 \cdot 5 = (1,1,2,2,1,1,0,0)$$

Das ist so keine Binärzahl, aber Abarbeiten der Überträge (von links nach rechts) liefert die korrekte Binärzahl:

$$(1,1,2,2,1,1,0,0) \to (1,1,0,3,1,1,0,0) \to (1,1,0,1,2,1,0,0)$$
$$\to (1,1,0,1,0,2,0,0) \to (1,1,0,1,0,0,1,0) = 75 = 15 \cdot 5$$

Es gibt noch Nachteile:

- Bisher benutzten wir bei der DFT komplexe Einheitswurzeln: Lösungen der Gleichung $x^N = 1$ in $\mathbb{C}$. Also $x = e^{2\pi i/N}$. Daher
  - komplexe Zahlen
  - nicht immer ganzzahlig

Der Algorithmus von Schönhage-Strassen benutzt statt komplexer Einheitswurzeln (in $\mathbb{C}$) Einheitswurzeln in dem *Ring* $\{0, 1, \ldots, N-1\}$ mit $+ \mod N$ und $\cdot \mod N$: Lösungen von

$$x^n = 1 \mod N$$

Dadurch rechnet man ganzzahlig (und reell).

Man wählt außerdem $N = 2^k + 1$, dann sind die Einheitswurzeln alle von der Form 2^m. "Mal Einheitswurzel" ist dann billig: shift der Binärzahl.
Außerdem ist $\mod 2^k + 1$ auch billig.

Laufzeit des Schönhage-Strassen-Algorithmus zur Multiplikation zweier Zahlen mit n Binärstellen ist $O(n \log n \log\log n)$.

Für Zahlen mit 1024 Bit bereits besser als "normale" Multiplikation: Statt ca. $1024^2 = 1.048.576$ Operationen für "naive" Multiplikation brauchen wir drei DFTs der Länge 2048: weniger als 180.000 Operationen.

Der Schönhage-Strassen-Algorithmus war von 1971 bis 2007 der effizienteste bekannte Algorithmus zur Multiplikation großer Zahlen; 2007 veröffentlichte Martin **Fürer** *eine Weiterentwicklung des Algorithmus mit einer noch niedrigeren asymptotischen Komplexität.*
Diese Komplexität stellt eine Verbesserung sowohl gegenüber dem "naiven" aus der Schule bekannten Algorithmus der Laufzeit $O\left(n^2\right)$ als auch gegenüber dem 1962 entwickelten **Karatsuba-Algorithmus** *mit einer Laufzeit von $O\left(n^{\log_2(3)}\right)$ sowie dessen verbesserter Variante, dem* **Toom-Cook-Algorithmus** *mit $O(n^{1+\varepsilon})$ Laufzeit dar." (wikipedia)*

Karatsuba:
Grundidee sehr einfach:

$$\begin{aligned}(ax+b)(cx+d) &= acx^2 + (ad+bc)x + bd \\ &= acx^2 + ((a+b)(c+d) - ac - bd)x + bd\end{aligned}$$

Allgemein (in Basis B, z.B. $B = 2$ oder $B = 10$):

$$x = x_1 B^m + x_0, y = y_1 B^m + y_0$$

Produkt: $xy = (x_1 B^m + x_0)(y_1 B^m + y_0) = z_2 B^{2m} + z_1 B^m + z_0$
mit $z_2 = x_1 y_1, z_1 = x_1 y_0 + x_0 y_1, z_0 = x_0 y_0$. Oder

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

In jedem Divide-and-Conquer-Schritt drei Multiplikationen statt vier. Gesamtzahl Multiplikationen daher nicht $n^{\log_2 4} = n^2$, sondern $n^{\log_2 3} \approx n^{1,585}$.

Karatsuba und Toom-Cook sind auch Divide-and-Conquer-Algorithmen.

In der Praxis ist Schönhage-Strassen ab etwa $n = 2^{2^{15}}$ bis $2^{2^{17}}$ (also ca 10 000 bis 40 000 Dezimalstellen) besser als Karatsuba und Tom-Cook.

"Fürer's algorithm currently only achieves an advantage for astronomically large values and is not used in practice." (wikipedia)