Diskrete Fouriertransformation

- Motivation: Gleichung der schwingenden Saite
- Crashkurs “Komplexe Zahlen”
- Anwendung: Mustererkennung, Signalverarbeitung (skizziert)
- Anwendung: Multiplikation von Polynomen
- Schnelle Implementierung mit *divide-and-conquer* (FFT)

Diskrete Fouriertransformation

Eine Saite der Länge 1 sei an ihren Enden fest eingespannt. Ihre Auslenkung an der Stelle x (wobei $0 \leq x \leq 1$) zum Zeitpunkt t (wobei $t \geq 0$) bezeichnen wir mit $u(x,t)$.

Unter vereinfachenden Annahmen gilt:

$$c^2 \frac{\partial^2}{\partial x^2} u(x,t) = \frac{\partial^2}{\partial t^2} u(x,t)$$

- c ist eine Konstante, in die Spannung und Masse der Saite eingehen.
- Die linke Seite entspricht der Krümmung der Saite an der Stelle x zum Zeitpunkt t; daraus resultiert eine Rückstellkraft.
- Die rechte Seite entspricht der Beschleunigung, also nichts anderes als Newtons Gesetz "Kraft ist Masse mal Beschleunigung".

Theorie der partiellen DGL $\Rightarrow$: ist $u(x,0)$ vorgegeben (Anfangsauslenkung), gilt $\frac{\partial u}{\partial t}(x,0) = 0$ (Saite wird nicht angeschoben) und gilt $u(0,t) = u(1,t) = 0$ (Saite ist an den Enden eingespannt), so ist $u(x,t)$ für alle $0 \leq x \leq 1$ und $t \geq 0$ eindeutig festgelegt (entsprechend der Anschauung!)

Aber wie rechnet man $u(x,t)$ aus $v(x) := u(x,0)$ aus??

Dieses Problem löste Fourier im 19.Jh.

Fouriers Lösung

Wenn $v(x) = u(x,0) = \sin(\pi k x)$, dann ist $u(x,t) = \sin(\pi k x)\cos(\pi c k t)$.

Beweis durch Nachrechnen:

$$c^2 \frac{\partial^2}{\partial x^2} u(x,t) = -c^2\pi^2 k^2 u(x,t) = \frac{\partial^2}{\partial t^2} u(x,t)$$

Falls $v(x) = \sum_{k=1}^{\infty} \alpha_k \sin(\pi k x)$ dann $u(x,t) = \sum_{k=1}^{\infty} c_k \sin(\pi k x)\cos(\pi c k t)$.

Fouriers geniale Feststellung: Jedes stetige v ist von dieser Form:

$$\alpha_k = \frac{2}{\pi k} \int_0^{\pi} v(x)\sin(\pi k x)\mathrm{d}x$$

Die Folge α_k ist die *Fouriertransformation* von v (genauer: Folge der Fourierkoeffizienten).

Diskrete Fouriertransformation

Gegeben sei eine Folge $a_0, a_1, \ldots, a_{n-1}$ reeller Zahlen. Z.B. Stützstellen einer Funktion.

Die *diskrete Fouriertransformation* (DFT) dieser Folge ist die Folge $y_0, y_1, \ldots, y_{n-1}$ *komplexer Zahlen* definiert durch

$$y_k = \sum_{j=0}^{n-1} e^{\frac{2\pi i}{n} jk} a_j$$

Zusammenhang mit Motivation: $e^{i\phi} = \cos(\phi) + i \sin(\phi)$ (EULERsche Formel). Die α_k können aus Imaginärteil der DFT näherungsweise bestimmt werden.

NB: Die Motivation mit „schwingender Saite“ ist nicht prüfungsrelevant.

Wir interessieren uns für die DFT als eigenständige Operation aus anderen (aber verwandten) Gründen.

Komplexe Zahlen

Menge der komplexen Zahlen $\mathbb{C}$ entsteht aus $\mathbb{R}$ durch formale Hinzunahme von $i = \sqrt{-1}$, also $i^2 = -1$.

Eine komplexe Zahl ist ein Ausdruck der Form $a + ib$ wobei $a, b \in \mathbb{R}$.

Addition: $(a_1 + ib_1) + (a_2 + ib_2) = (a_1 + a_2) + i(b_1 + b_2)$.

Multiplikation:
$(a_1 + ib_1)(a_2 + ib_2) = a_1a_2 + i(a_1b_2 + a_2b_1) + i^2b_1b_2 = (a_1a_2 - b_1b_2) + i(a_1b_2 + a_2b_1)$.

Division: $\frac{a_1+ib_1}{a_2+ib_2} = \frac{(a_1+ib_1)(a_2-ib_2)}{(a_2+ib_2)(a_2-ib_2)} = \frac{a_1a_2+b_1b_2+i(a_2b_1-a_1b_2)}{a_2^2+b_2^2}$

Bemerkung: a ist der Realteil der komplexen Zahl $a + ib$, die Zahl b ist ihr Imaginärteil. Man schreibt $a = \mathrm{Re}(a + ib)$ und $b = \mathrm{Im}(a + ib)$.

Zahlenebene

Man visualisiert komplexe Zahlen als Punkte oder Ortsvektoren. $a + ib$ entspricht dem Punkt (a, b).

Addition entspricht dann der Vektoraddition.

Betrag: $|a + ib| = \sqrt{a^2 + b^2}$. Es gilt $|wz| = |w||z|$.

Man kann einer komplexen Zahl $z = a + ib$ ihren Winkel im Gegenuhrzeigersinn von der reellen Achse aus gemessen, zuordnen. Dieser Winkel ϕ heißt Argument von z, i.Z. $\arg(z)$. Es gilt $\phi = \texttt{atan2}(a, b)$. D.h. $\tan(\phi) = b/a$ und $b \leq 0 \Leftrightarrow |\phi| \geq \pi/2$.

Aus Betrag $r = |z|$ und Argument $\phi = \arg(z)$ kann man z rekonstruieren:

$$z = r\cos(\phi) + ir\sin(\phi)$$

Mit $e^{i\phi} = \cos(\phi) + i\sin(\phi)$ erhält man

$$z = re^{i\phi}$$

Dies ist die Polarform der komplexen Zahl z.

Es gilt: $(r_1 e^{i\phi_1})(r_2 e^{i\phi_2}) = r_1 r_2 e^{i(\phi_1+\phi_2)}$.

Beim Multiplizieren multiplizieren sich die Beträge und addieren sich die Argumente.

Einheitswurzeln

Sei $n \in \mathbb{N}$. Wir schreiben $\omega_n := e^{2\pi i/n}$.

Es gilt $\omega_n^n = 1$ und allgemeiner $(\omega_n^k)^n = 1$.

Die n (verschiedenen) Zahlen $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}$ heißen daher n-te Einheitswurzeln.

Sie liegen auf den Ecken eines regelmäßigen n-Ecks mit Mittelpunkt Null und einer Ecke bei der Eins.

Satz: Es gilt für $k = 0, \ldots, n-1$:

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \begin{cases} n, \text{ falls } k = 0 \\ 0, \text{sonst} \end{cases}$$

Beweis mit geometrischer Reihe oder durch Veranschaulichung in der Zahlenebene.

Satz: Ist n gerade, so sind die $n/2$-ten Einheitswurzeln gerade die Quadrate der n-ten Einheitswurzeln.

Beweis: Eine Richtung: $((\omega_n^k)^2)^{n/2} = 1$. Andere Richtung: $\omega_{n/2}^k = (\omega_n^2)^k$.

Einheitswurzeln und DFT

Definition der DFT: $y_k = \sum_{j=0}^{n-1} e^{\frac{2\pi i}{n} jk} a_j$.

Es ist $y_k = \sum_{j=0}^{n-1} \omega_n^{jk} a_j$.

Die DFT ergibt sich also als die Folge $A(1), A(\omega_n), A(\omega_n^2), \ldots, A(\omega_n^{n-1})$ wobei $A(x) = \sum_{j=0}^{n-1} a_j x^j$.

Translationsinvarianz

Satz: Sei $(a_j)_{j=0}^{n-1}$ eine Folge reeller Zahlen und $b_j = a_{(j+d) \bmod n}$. Sei $(y_k)_k$ die DFT von $(a_j)_j$ und $(z_k)_k$ die DFT von $(b_j)_j$. Es gilt $|y_k| = |z_k|$. D.h. der Betrag der DFT, das sog. Intensitätsspektrum, ist translationsinvariant.

Beweis: $z_k = \sum_j \omega_n^{jk} a_{j+d \bmod n} = \sum_{j'} \omega_n^{(j'-d)k} a_{j'k} = \omega_n^{-dk} y_k$. Und $|\omega_n^x| = 1$.

Anwendung: Um festzustellen, ob ein bestimmtes Muster irgendwo in der Folge a vorhanden ist, vergleiche man die DFT von a mit der DFT des Musters.
Insbesondere interessant bei 2D-DFT. Z.B. Schrifterkennung.

Inverse DFT

Gegeben sei eine Folge $y_0, \ldots, y_{n-1}$ komplexer Zahlen.

Die inverse DFT ist die Folge $a_0, \ldots, a_{n-1}$ definiert durch

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{-jk} y_k$$

Satz: Die inverse DFT ist tatsächlich die zur DFT inverse Operation.

Beweis: Wir verwenden das Kroneckersymbol $\delta_{jl} = \texttt{if } j = l \texttt{ then } 1 \texttt{ else } 0$.

Es genügt, die Behauptung für $a_j = \delta_{jl}$ für alle $l = 0, \ldots, n-1$ zu zeigen, da DFT, und inverse DFT linear sind und diese a's eine Basis bilden.

Sei also $a_j = \delta_{jl}$. Anwendung der DFT liefert

$$y_k = \omega_n^{kl}$$

Anwendung der inversen DFT liefert dann

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{-jk} \omega_n^{kl} = \frac{1}{n} \sum_{k=0}^{n-1} (\omega_n^k)^{l-j} = \delta_{lj}$$

DFT und Polynome

Ein Polynom mit Gradschranke n ist ein formaler Ausdruck der Form

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Die a_j heißen Koeffizienten des Polynoms A.

Man kann ein Polynom als Funktion auffassen (Polynomfunktion). Z.B.: $A(x) = 1 + 2x + 3x^2$ und $A(2) = 17$.

Hornerschema

Sei $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ ein Polynom mit Gradschranke n und x_0 eine Zahl.

Bestimmung von $A(x_0)$ nach HORNER mit $\Theta(n)$ Rechenoperationen:

$$A(x_0) = a_0 + x_0(a_1 + x_1(a_2 + \cdots + x_0(a_{n-2} + x_0 a_{n-1}) \dots)$$

Interpolation

Ein Polynom ist durch seine Polynomfunktion eindeutig bestimmt, d.h., man kann die Koeffizienten aus der Polynomfunktion ausrechnen.

Genauer: Sei $A(x_k) = y_k$ für n verschiedene Stützstellen $x_0, \ldots, x_{n-1}$.

Dann gilt nach Lagrange:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

Ausmultiplizieren liefert das gewünschte Polynom.

Bemerkung: Dies gilt nur bei den reellen Zahlen, oder genauer bei Körpern der Charakteristik 0. Im zweielementigen Körper induzieren x^2 und x dieselben Polynomfunktionen.

Addition und Multiplikation von Polynomen

Polynome kann man addieren und multiplizieren:

Die Summe zweier Polynome $A(x) = \sum_{j=0}^{n-1} a_j x^j$ und $B(x) = \sum_{j=0}^{n-1} b_j x^j$ mit Gradschranke n ist das Polynom $C(x) = \sum_{j=0}^{n-1} c_j x^j$ wobei $c_j = a_j + b_j$.

Das Produkt zweier Polynome $A(x) = \sum_{j=0}^{n-1} a_j x^j$ und $B(x) = \sum_{j=0}^{n-1} b_j x^j$ mit Gradschranke n ist das Polynom $C(x) = \sum_{j=0}^{2n-1} c_j x^j$ mit Gradschranke $2n$ wobei $c_j = \sum_{l=0}^{j} a_l b_{j-l}$.

Z.B.: $(1 + 2x + 3x^2)(4 + 5x + 6x^2) = 1 + 13x + 25x^2 + 27x^3 + 18x^4$.

Sind Polynome als Koeffizientenliste repräentiert, so erfordert das Ausrechnen der Summe $O(n)$ Operationen; das Ausrechnen des Produkts erfordert $O(n^2)$ Operationen.

Punkt-Wert Repräsentation

Man kann Polynome auch als Punkt-Wert Liste repräsentieren, also als Liste von Paaren

$$((x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}))$$

wobei die x_k paarweise verschieden sind.

Z.B. $A(x) = 1 + 2x + 3x^2$ repräsentiert durch $((0, 1), (1, 6), (-1, 2))$.

Punkt-Wert Repräsentation

Sind $((x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}))$

und $((x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1}))$

zwei solche Repräsentationen zu denselben Stützstellen, so erhält man Repräsentationen für Summe und Produkt als

$$((x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1}))$$

und

$$((x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{n-1}, y_{n-1} y'_{n-1}))$$

D.h. Addition und Multiplikation von Polynomen in Punkt-Wert Repräsentation erfordert $O(n)$ Operationen.

Aber die Auswertung von Polynomen in Punkt-Wert Repräsentation erfordert $O(n^2)$ Operationen (*there ain't no such thing as a free lunch*).

Multiplikation mit DFT

Erinnerung: Die DFT einer Liste $(a_0, \ldots, a_{n-1})$ ist die Liste $(y_0, \ldots, y_{n-1})$ wobei

$$y_k = A(\omega_n^k)$$

und

$$A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

Die DFT von $(a_0, \ldots, a_{n-1})$ ist die Punkt-Wert Repräsentation des Polynoms $\sum_{j=0}^{n-1} a_j x^j$ zu den Stützstellen $\omega_n^0, \ldots, \omega_n^{n-1}$, also den n-ten Einheitswurzeln.

Um zwei Polynome in Koeffizientenform zu multiplizieren kann man ihre DFTs punktweise multiplizieren und dann die inverse DFT anwenden.

Multiplikation mit DFT

- Gegeben zwei Polynome A, B mit Gradschranke n als Koeffizientenlisten
- Fülle mit Nullen auf um formell Polynome mit Gradschranke $2n$ zu erhalten.

$$a = (a_0, \ldots, a_{2n-1}), \quad b = (b_0, \ldots, b_{2n-1})$$

- Bilde jeweils DFT der beiden expandierten Koeffizientenlisten:
  $(y_0, \ldots, y_{2n-1}) = \mathrm{DFT}_{2n}(a)$, $(z_0, \ldots, z_{2n-1}) = \mathrm{DFT}_{2n}(b)$,
- Multipliziere die DFTs punktweise: $w_k = y_k z_k$ für $k = 0, \ldots, 2n-1$,
- Transformiere zurück um Koeffizientenliste von $C = AB$ zu erhalten:
  $(c_0, \ldots, c_{2n-1}) = \mathrm{DFT}^{-1}(w_0, \ldots, w_{2n-1})$.

Leider braucht das Ausrechnen der DFT $\Theta(n^2)$ Rechenoperationen, sodass keine unmittelbare Verbesserung eintritt.

Mit *divide and conquer* kann man aber die DFT in $\Theta(n \log n)$ ausrechnen!

Schnelle Fouriertransformation (FFT)

Sei n bis auf weiteres eine Zweierpotenz.

$A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$.

Es ist $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$ wobei

$$A^{[0]}(z) = a_0 + a_2 z + a_4 z^2 + \cdots + a_{n-2} z^{n/2-1}$$

$$A^{[1]}(z) = a_1 + a_3 z + a_5 z^2 + \cdots + a_{n-1} z^{n/2-1}$$

Beispiel:

$$A(x) = 1 + 2x + 3x^2 + 4x^3$$
$$A^{[0]}(z) = 1 + 3z$$
$$A^{[1]}(z) = 2z + 4z^2$$

Also gilt unter Beachtung von $\omega_n^{2k} = \omega_{n/2}^k$:

$$A(\omega_n^k) = A^{[0]}(\omega_{n/2}^k) + \omega_n^k A^{[1]}(\omega_{n/2}^k)$$

Für $k < n/2$ sind die beiden rechtsstehenden Polynomauswertungen selbst DFTs, aber zur halben Größe.

Für $k \geq n/2$ passiert nichts neues, da $\omega_{n/2}^{k+n/2} = \omega_{n/2}^k$.

Nur der Vorfaktor ω_n^k ändert sich: $\omega_n^{k+n/2} = -\omega_n^k$.

Schnelle Fouriertransformation (FFT)

Schnelle Fouriertransformation (FFT)

RECURSIVE-FFT(a)

1 $n \leftarrow length[a]$

2 **if** $n = 1$

3 **then return** a

4 $\omega_n \leftarrow e^{2\pi i/n}$

5 $\omega \leftarrow 1$

6 $a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n/2-2})$

7 $a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n/2-1})$

8 $y^{[0]} \leftarrow$ RECURSIVE-FFT$(a_0, a_2, \ldots, a_{n/2-2})$

9 $y^{[1]} \leftarrow$ RECURSIVE-FFT$(a_1, a_3, \ldots, a_{n/2-1})$

10 **for** $k \leftarrow n/2 - 1$ **do**

11 $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$

12 $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$

13 $\omega \leftarrow \omega\omega_n$

14 **return** y

Schnelle Fouriertransformation (FFT)

- Nach dem Vorhergesagten ist klar, dass y die DFT von a ist.
- Die Laufzeit $T(n)$ von RECURSIVE-FFT erfüllt $T(n) = 2T(n) + \Theta(n)$, also $T(n) = \Theta(n \log n)$.
- In der Praxis rechnet man die DFT iterativ von unten her aus (vgl. dynamische Programmierung), indem man sich überlegt für welche Arrays rekursive Aufrufe stattfinden.
- Man kann die FFT auch gut parallelisieren.
- Zur Multiplikation von Polynomen mit ganzen Koeffizienten empfiehlt sich die modulare DFT.

DFT in 2D

Sei $a_{jk}, j = 0, \ldots, n-1, k = 0, \ldots, n-1$ eine Matrix von Zahlen. Die DFT von a ist die Matrix $y_{uv}, u = 0, \ldots, n-1, v = 0, \ldots, n-1$ definiert durch

$$y_{uv} = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \omega_{2n}^{ju+kv} a_{jk}$$

Translationsinvarianz des Intensitätsspektrums gilt jetzt bezüglich Verschiebung in j und k Richtung.

Schnelle inverse DFT

Erinnerung: Die inverse DFT einer Liste $(y_0, \ldots, y_{n-1})$ ist die Liste $(a_0, \ldots, a_{n-1})$ wobei

$$a_j = \frac{1}{n} Y(\omega_n^{-j})$$

und

$$Y(x) = y_0 + y_1 x + \cdots + y_{n-1} x^{n-1}$$

Aber $\omega_n^{-j} = \omega_n^{n-j}$. Man erhält die inverse DFT von y aus der DFT von y durch Umgruppieren und Dividieren durch n:

Schnelle inverse DFT

INVERSE-FFT(y)

1 $n \leftarrow \mathit{length}[y]$

2 $a \leftarrow$ RECURSIVE-FFT(y)

3 **for** $j \leftarrow 1$ **to** $n/2 - 1$ **do**

4 $\quad$ *Swap* $a_j \leftrightarrow a_{n-j}$

5 **for** $j \leftarrow 0$ **to** $n - 1$ **do**

6 $\quad$ $a_j \leftarrow a_j/n$

7 **return** a

Faltung

Seien $a = (a_0, \ldots, a_{n-1})$ und $b = (b_0, \ldots, b_{n-1})$ zwei Folgen von je $2n$ Zahlen wobei $a_k = b_k = 0$ für $k \geq n$.

Die Faltung (engl.: *convolution*) $a \star b$ ist die Folge von $2n$ Zahlen definiert durch

$$(a \star b)_k = \sum_{j=0}^{k} a_j b_{k-j}$$

Beachte: $a \star b$ ist die Koeffizientenfolge des Produkts der durch a und b repräsentierten Polynome.

Satz: $a \star b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a)\mathrm{DFT}_{2n}(b))$ wobei das Produkt komponentenweise zu verstehen ist.

Anwendung der Faltung

Faltung mit $b = (-1, 2, -1, 0, 0, 0, 0, \ldots, 0)$ hebt Sprünge hervor.

In 2D: *edge detection*.

Faltung mit $b = (1, 2, 1, 0, 0, 0, 0, 0, \ldots, 0)$ verwischt.

In 2D: *blurring*.

Die Operation $c \mapsto \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(c)/\mathrm{DFT}_{2n}(b))$ macht die Faltung mit b rückgängig.

In 2D: *sharpening*

Die Faltung hat auch zahlreiche Anwendungen bei der Audiosignalverarbeitung.

Geometrische Algorithmen

- Entscheidungsprobleme:
  - Schneiden sich zwei Geraden / Strecken / Kreise / Objekte?
  - Liegt ein Punkt im Inneren eines Dreiecks / Polygons / Kugel / Objekts?
  - Bilden n Punkte ein konvexes Polygon / Polyeder?
  - Liegen Punkte auf einer Geraden / in einer Ebene ?
  - Ist ein Winkel spitz oder stumpf?
- Berechnungsprobleme:
  - Konvexe Hülle
  - Projektion
  - Triangulierung
- Anwendungen:
  - Grafik (Spiele, VR, ...)
  - CAD/CAM
  - Robotik

Punkte, Vektoren, Strecken, Geraden

Ein Punkt in der Ebene ist durch sein Koordinatenpaar gegeben: $p = (x, y)$.

Wir identifizieren Punkte mit ihren Ortsvektoren.

Vektoren (und somit Punkte) kann man (komponentenweise) addieren, subtrahieren und mit Skalaren multiplizieren.

Gerade durch zwei Punkte in der Ebene:

$$p_1 p_2 = \{\alpha p_1 + (1 - \alpha) p_2 \mid \alpha \in \mathbb{R}\}$$

Strecke zwischen zwei Punkten in der Ebene:

$$\overline{p_1 p_2} = \{\alpha p_1 + (1 - \alpha) p_2 \mid 0 \leq \alpha \leq 1\}$$

Ein Punkt in $\overline{p_1 p_2}$ heißt auch konvexe Kombination von p_1 und p_2.

Eine Punktmenge M ist konvex, wenn für je zwei Punkte $p_1, p_2 \in M$ die gesamte Verbindungsstrecke in M liegt, also $\overline{p_1 p_2} \subseteq M$.

Kreuzprodukt

Das Kreuzprodukt zweier Vektoren $p_1 = (x_1, y_1)$ und $p_2 = (x_2, y_2)$ ist der Skalar

$$p_1 \times p_2 = x_1 y_2 - x_2 y_1$$

Es gilt:

$$p_1 \times p_2 = - \, p_2 \times p_1$$

Es gilt auch:

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$

Der Betrag von $p_1 \times p_2$ ist die Fläche des von p_1 und p_2 aufgespannten Parallelogramms.

Das Vorzeichen von $p_1 \times p_2$ ist positiv, wenn p_2 aus p_1 durch Drehen nach links (und Skalieren) entsteht.

Das Vorzeichen von $p_1 \times p_2$ ist negativ, wenn p_2 aus p_1 durch Drehen nach rechts (und Skalieren) entsteht.

Das Kreuzprodukt $p_1 \times p_2$ ist Null, wenn p_1 und p_2 linear abhängig sind.

Nach links oder nach rechts?

Gegeben sind drei Punkte p_0, p_1, p_2.

Man bewegt sich entlang $\overline{p_0 p_1}$ von p_0 nach p_1 und dann entlang $\overline{p_1 p_2}$ nach p_2.

Man soll feststellen ob man bei p_1 nach links oder nach rechts geht.

Man geht nach links, wenn $(p_1 - p_0) \times (p_2 - p_0) > 0$.

Man geht nach rechts, wenn $(p_1 - p_0) \times (p_2 - p_0) < 0$.

Man geht geradeaus oder rückwärts, wenn $(p_1 - p_0) \times (p_2 - p_0) = 0$.

Feststellen, ob sich zwei Strecken schneiden

Wir untersuchen jetzt die Frage, ob sich zwei gegebene Strecken in einem Punkt schneiden.

Zu einer Strecke $\overline{p_1 p_2}$ bilden wir die *bounding box* (begrenzendes achsenparalleles Rechteck):

Linke untere Ecke der *bounding box*: $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$, wobei
$\hat{x}_1 = \min(x_1, x_2), \hat{y}_1 = \min(y_1, y_2)$.

Rechte obere Ecke der *bounding box*: $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$, wobei
$\hat{x}_2 = \max(x_1, x_2), \hat{y}_2 = \max(y_1, y_2)$.

Oberflächlicher Test

Zwei Strecken $\overline{p_1p_2}$ und $\overline{p_3p_4}$ haben sicher keinen Punkt gemeinsam, wenn ihre *bounding boxes* disjunkt sind, d.h., wenn

$$(\hat{x}_2 \geq \hat{x}_3) \wedge (\hat{x}_4 \geq \hat{x}_1) \wedge (\hat{y}_2 \geq \hat{y}_3) \wedge (\hat{y}_4 \geq \hat{y}_1) \qquad (\star)$$

falsch ist.

Hier sind $(\hat{x}_3, \hat{y}_3)$ und $(\hat{x}_4, \hat{y}_4)$ die linke untere / rechte obere Ecke der *bounding box* von $\overline{p_3p_4}$.

Ist die Aussage $(\star)$ wahr so schneiden sich die *bounding boxes*; die Strecken können natürlich trotzdem disjunkt sein.

Entwurfsprinzip: *bounding box*

Oft muss man entscheiden, ob in einer Menge geometrischer Objekte welche überlappen, oder sich überdecken.

Beispiel: Szenenwiedergabe (*rendering*).

Es bietet sich an, für alle Objekte *bounding boxes* mitzuführen, oder zu berechnen. Durch schnelle Tests auf den *bounding boxes* kann man die meisten Fälle von vornherein ausschließen und muss (möglicherweise aufwendige) exakte Tests nur auf wenigen Objekten durchführen.

Feststellen, ob sich zwei Strecken schneiden

Gegeben sind zwei Strecken $\overline{p_1p_2}$ und $\overline{p_3p_4}$. Wir wollen überprüfen, ob $\overline{p_1p_2} \cap \overline{p_3p_4} \neq \emptyset$.

- Zunächst überprüft man, ob sich die *bounding boxes* schneiden.
- Falls nein, so schneiden sich die Strecken auch nicht.
- Falls ja, so schneiden sich die Strecken genau dann, wenn $\overline{p_1p_2}$ die Gerade p_3p_4 schneidet und $\overline{p_3p_4}$ die Gerade p_1p_2 schneidet.
- Die Strecke $\overline{p_3p_4}$ schneidet p_1p_2 genau dann, wenn sich die Vorzeichen von $(p_3 - p_1) \times (p_2 - p_1)$ und $(p_4 - p_1) \times (p_2 - p_1)$ unterscheiden, oder eines oder beide Null ist (sind).

Beachte: Der Fall dass alle vier Punkte auf einer Geraden liegen, die Strecken aber nicht überlappen, wird durch den „oberflächlichen Test“ ausgeschlossen.

Überstreichen

Gegeben ist eine Menge S von n Strecken. Wir wollen feststellen, ob $a \cap b \neq \emptyset$ für zwei $a, b \in S$ mit $a \neq b$.

Wir verwenden das Entwurfsprinzip „Überstreichung“ (engl.: *sweeping*).

Eine gedachte senkrechte Gerade überstreicht dabei die gegebenen Daten von links nach rechts.

Dabei wird an Ereignispunkten nach Maßgabe des bisher Gesehenen eine dynamische Datenstruktur verändert.

Das Endresultat ergibt sich nach Überstreichen der gesamten Eingabe aus dem erreichten Zustand der Datenstruktur.

Anordnung der Strecken

Vereinfachende Annahme: Keine Strecke senkrecht, höchstens zwei Strecken schneiden sich in einem Punkt.

Seien a, b Strecken in S und x eine Position der Überstreichungsgeraden. Wir schreiben $a <_x b$ wenn sowohl a, als auch b, die Gerade durch x schneiden und zwar a weiter unten als b.

$$a <_x b \Leftrightarrow \exists y_1, y_2 . y_1 < y_2 \wedge (x, y_1) \in a \wedge (x, y_2) \in b$$

Schneiden sich a und b, so gibt es eine Position der Überstreichungsgeraden x_0, sodass a, b in x_0 unmittelbar aufeinanderfolgen.

Wie stellt man fest, ob $a <_x b$?

- Zunächst schauen wir, ob a und b von der Geraden durch x getroffen werden. Das erfordert nur Größenvergleiche reeller Zahlen. Wenn nicht, so sind a und b bzgl. $<_x$ unvergleichbar.
- Nun können wir die Schnittpunkte der duch a, b definierten Geraden mit der Geraden durch x bestimmen und die y-Koordinaten vergleichen. Durchmultiplizieren mit Hauptnenner vermeidet Divisionen.
- Alternativ können Kreuzprodukte verwendet werden, siehe Übung.

Feststellen, ob sich irgendzwei Strecken in S schneiden

Um festzustellen, ob sich irgendzwei Strecken in S schneiden...

- ...halten wir in einer Datenstruktur die Ordnung $<_x$ in der aktuellen Position x der Überstreichungsgeraden fest;
- der Zustand der Datenstruktur wird nur verändert an x-Koordinaten von Anfangs- oder Endpunkten von Strecken in S; die Ereignispunkte sind also diese x-Koordinaten;
- werden an einem Ereignispunkt x zwei Strecken unmittelbar aufeinanderfolgend (in der Ordnung $<_x$), so überprüfen wir, ob sie sich schneiden;
- falls das passiert, so gibt es zwei sich schneidende Strecken in S,
- passiert es nie, so schneiden sich keine zwei Strecken in S.

Verwendete Datenstruktur

- An den Ereignispunkten verändert sich die Ordnung $<_x$ nur dadurch, dass Strecken eingefügt oder entfernt werden.
- Die relative Position von Strecken zueinander ändert sich nicht, denn das könnte nur passieren, wenn sie sich schneiden und das würden wir (hoffentlich) rechtzeitig merken.
- Wir können daher die Ordnung $<_x$ in einem Rot-Schwarz-Baum speichern, der die folgenden Operationen unterstützen muss:
  - Insert(a, x): Einfügen der Strecke a gemäß $<_x$, falls a von der Geraden durch x geschnitten wird.
  - Above(a): Liefert den unmittelbaren Nachfolger von a in der durch den Baum definierten Ordnung (falls er existiert; Nil sonst.)
  - Below(a): Liefert den unmittelbaren Vorgänger von a in der durch den Baum definierten Ordnung.
  - Delete(a): Löscht die Strecke a aus der Datenstruktur.

Pseudocode für das Verfahren

```
TEST-INTERSECT(S)
1   (p_1, ..., p_k) ← Sortierung der Endpunkte in S nach x-Koordinaten und
                      bei gleicher x-Koordinate nach y-Koordinaten.
2   Erzeuge leeren Rot-Schwarz-Baum für Strecken
3   for i ← 1 to n/2 − 1 do
4       x ← x-Koordinate von p_i
5       if p_i = Linker Endpunkt der Strecke a then
6           INSERT(a, x)
7           if INTERSECT(a, ABOVE(a)) ∨ INTERSECT(a, BELOW(a)) then return TRUE
8       else
9           if INTERSECT(ABOVE(a), BELOW(a)) then return TRUE
10          DELETE(a)
11  return FALSE
```

Pseudocode für das Verfahren

Beachte: In Zeile 1 müssen Punkte, die mehrfach als Endpunkte auftreten, auch mehrfach aufgeführt werden.

Am besten erreicht man das, indem man p als Paar eines Punktes und einer Strecke implementiert.

Nur dann geht auch das Auffinden von a in $O(1)$.

Die Prozedur INTERSECT(a, b) soll entscheiden, ob sich a und b schneiden. Ist eines der beiden Argumente NIL, so soll FALSE herauskommen.

Verifikation

Die folgenden Invarianten gelten jeweils nach Zeile 10:

- Die Datenstruktur enthält genau die Ordnung $<_x$ eingeschränkt auf die bisher gesehenen Punkte und abzüglich derer, deren rechter Endpunkt x ist.
- Keine zwei Strecken schneiden sich in einem Punkt mit x-Koordinate kleiner oder gleich x
- Schneiden sich zwei im Baum befindliche Strecken nach x, so folgen sie nicht unmittelbar aufeinander.

Wir liefern nur dann TRUE zurück, wenn tatsächlich eine Überschneidung vorliegt.

Wir liefern nur dann FALSE zurück, wenn wir alle Punkte in der for-Schleife verarbeitet haben. in diesem Falle implizieren die Invarianten, dass keine Überschneidungen vorliegen.

Konvexe Hülle

Die konvexe Hülle einer Punktmenge M ist die kleinste konvexe Punktmenge, die M umfasst.

Zur Erinnerung: X ist konvex, wenn mit $x, y \in X$ auch $\overline{xy} \subseteq X$.

Ist M eine endliche Menge, so ist die konvexe Hülle ein (konvexes) Polygon, dessen Ecken eine Teilmenge von M bilden.

Das Problem „Bestimmung der konvexen Hülle“ besteht darin, zu gegebener (endlicher) Punktmenge M der Größe n eine Liste von Punkten $p_1, \ldots, p_h$ zu berechnen, sodass die p_i die Ecken der konvexen Hülle von M in der richtigen Reihenfolge sind.

Manchmal verzichtet man auf die Reihenfolge.

Anwendungen der komplexen Hülle

- Größter Durchmesser. Um die am weitesten entfernt liegenden Punkte einer Liste von Punkten zu finden, kann man die Suche auf die konvexe Hülle beschränken. Den größten Durchmesser eines konvexen Polygons kann man in Zeit $O(n)$ bestimmen.
- Oberflächlicher Test für Überlappung. Überlappen die konvexen Hüllen nicht, so auch nicht die Punktmengen.

Bestimmung der konvexen Hülle

- **Inkrementelles Verfahren.** Sukzessives Hinzufügen von Punkten zur konvexen Hülle der bisher gesehenen Punkte. Wenn naiv implementiert: $\Theta(n^2)$.
- **Graham's scan.** Man ordnet die Punkte nach Polarwinkel und überstreicht dann mit einer rotierenden Halbgeraden. Als Datenstruktur kommt ein Keller zur Anwendung: $O(n \log n)$.
- **Jarvis' march.** (auch *gift wrapping*) man „umwickelt" die Punktmenge von unten her links und rechts: $O(nh)$ wobei h die Größe der konvexen Hülle ist.
- **Divide-and-conquer.** Komposition der konvexen Hülle zweier konvexer Polygone: $\Theta(n \log n)$.
- **Bestes Verfahren:** Ähnlich wie Medianfindung in Linearzeit: $O(n \log h)$.

Graham's scan

- Wir wählen irgendeinen Punkt p_0 aus M und sortieren die übrigen nach Polarwinkel: $p_1, \ldots, p_{n-1}$.
- Wir verarbeiten die Punkte in dieser Reihenfolge, also $p_0, p_1, p_2, \ldots$.
- Wir verwenden einen Keller (*stack*) dessen Inhalt immer gerade die konvexe Hülle der bisher gesehenen Punkte ist.
- Bildet ein neuer Punkt p mit den obersten beiden Punkten auf dem Keller eine Linkskurve, so wird p auf den Keller gelegt.
- Das gleiche gilt, wenn der Keller weniger als zwei Elemente enthält.
- Bildet ein neuer Punkt p mit den obersten beiden Punkten auf dem Keller eine Rechtskurve oder liegt geradeaus, so entfernen wir Punkte vom Keller bis eine Linkskurve vorliegt.
- Sind alle Punkte verarbeitet, so enthält der Keller die Ecken der konvexen Hülle.

Zur Verifikation muss man sich nur überlegen, dass die angekündigte Invariante tatsächlich erhalten wird.

Laufzeit von Graham's scan

- Das Verarbeiten des i-ten Punktes kann bis zu $i-2$ Operationen erfordern. Laufzeit $\Theta(n^2)$??
- Natürlich nicht. Jeder Punkt wird höchstens einmal auf den Keller gelegt und höchstens einmal wieder entfernt. Es finden also insgesamt nur $O(n)$ Kelleroperationen statt. (Aggregatmethode)
- Darüberhinaus sind $O(n)$ Kreuzprodukte und Vergleiche vorzunehmen.
- Die Laufzeit wird vom Sortierprozess dominiert: $O(n \log n)$.

Jarvis' march

- Wir befestigen einen Faden am Punkt mit der niedrigsten y Koordinate.
- Wir wickeln das eine Ende des Fadens rechts herum, das andere links herum, solange bis sich die beiden Enden oben treffen.
- Der nächste Eckpunkt ist jeweils der mit dem kleinsten Polarwinkel in Bezug auf den als letztes Hinzugefügten.
- Der Polarwinkelvergleich kann allein mit Kreuzprodukten ermittelt werden.

Laufzeit $O(nh)$.

Besser als Graham's scan wenn $h = o(\log n)$.

Verallgemeinerbar auf 3D.

Punkte mit kleinstem Abstand

Wir wollen aus einer Menge P von n Punkten die zwei Punkte mit minimalem Abstand bestimmen.

Abstand: von (x_1, y_1) und (x_2, y_2):

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Interessiert man sich nur für Abstandsvergleiche, dann kann man auch das Abstandsquadrat verwenden. Man braucht dann keine Quadratwurzel.

Anwendungsbeispiel (in 3D): Flugsicherung. Die jeweils am nächsten beieinanderliegenden Flugzeuge seien rot zu kennzeichnen.

Mit *divide and conquer* können wir diese Aufgabe mit $O(n \log n)$ Operationen lösen.

Divide and conquer Verfahren für minimalen Abstand

- Ist $n \leq 3$ so bestimmen wir die nächstliegenden Punkte direkt.
- Ist $n > 3$ so teilen wir P durch eine senkrechte Gerade l in zwei Teile P_L und P_R sodass gilt: $|P_L| = \lceil n/2 \rceil$ und $|P_R| = \lfloor n/2 \rfloor$ und $P = P_L \cup P_R$ und P_L, bzw. P_R enthält nur Punkte, die links von l oder auf l, bzw. rechts von l oder auf l liegen.
- Seien δ_L und δ_R die minimalen Punktabstände in P_L, bzw. P_R bestimmt durch rekursive Anwendung des Verfahrens. Weiter sei $\delta = \min(\delta_L, \delta_R)$.
- Ist der minimale Punktabstand in P selbst kleiner als δ, so wird er realisiert durch Punkte $p_L \in P_L$ und $p_R \in P_R$. Die Punkte p_L und p_R müssen daher in einem Streifen der Breite 2δ um l liegen.

Durchsuchen des Streifens

- Die y-Koordinaten von p_L und p_R unterscheiden sich auch um höchstens δ, d.h. p_L und p_R (so es sie gibt) müssen in einem $2\delta \times \delta$ Rechteck um l liegen.
- Solch ein Rechteck enthält höchstens 8 Punkte: Die $\delta \times \delta$-Hälfte, die links von l liegt enthält höchstens vier Punkte (alle Punkte in P_L haben ja Abstand $\geq \delta$), ebenso die $\delta \times \delta$-Hälfte rechts von l. Also 8 Punkte insgesamt.
- Sortiert man die Punkte im 2δ-Streifen um l nach aufsteigenden y-Koordinaten, so muss man jeden Punkt immer nur mit den 7 nächstgrößeren vergleichen. Findet man unter diesen keinen, der näher als δ liegt, dann auch nicht später.

Die Laufzeit dieses Verfahrens erfüllt $T(n) = 2T(n/2) + O(n \log n)$.

Da ja für das Aufteilen nach x-Koordinaten sortiert werden muss und für das Mischen nach y-Koordinaten.

Also $T(n) = O(n(\log n)^2)$

Optimierung der Laufzeit

Man sortiert die Punkte am Anfang nach x und nach y Koordinaten.

D.h. man erzeugt zwei Listen X und Y jeweils der Länge $|P|$. Die Liste X (bzw. Y enthält die Punkte von P nach X-Koordinaten (bzw. y-Koordinaten) sortiert.

Aus diesen kann man in linearer Zeit ebensolche Listen gewinnen, die sich auf Teilmengen von P beziehen.

Damit erhält man die Laufzeit $O(n \log n)$ für das Verfahren.