

## Beilage zur Serie 6

### Trigonometrische Interpolation und schnelle Fourier-Transformation

#### Trigonometrische Interpolation

Wir möchten eine Funktion interpolieren, von der wir wissen, dass sie  $2\pi$ -periodisch ist, d.h.

$$f(x) = f(x + 2\pi k) \text{ für alle } x \in [0, 2\pi) \text{ und } k \in \mathbb{Z},$$

Dazu seien eine gerade Anzahl von  $n = 2m$  Stützpunkte  $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$  gegeben, wobei die Knoten  $\{x_k\}_{k=0}^{n-1}$  äquidistant verteilt seien mit

$$x_k := \frac{2\pi k}{n}, \quad \text{und} \quad y_k := f(x_k) \quad \text{für} \quad k = 0, \dots, n-1.$$

Eine weitverbreitete Methode ist, ein Interpolationspolynom der Form

$$P(x) = \frac{a_0}{2} + \sum_{\ell=1}^{m-1} [a_\ell \cos(\ell x) + b_\ell \sin(\ell x)] + \frac{a_m}{2} \cos(mx) \quad (1)$$

durch die Stützstellen aufzustellen, das die Periodizität der Funktionen  $\sin(\ell x)$  und  $\cos(\ell x)$  ausnutzt. Durch einen Wechsel in das Komplexe können wir die Koeffizienten  $a_\ell, b_\ell \in \mathbb{R}$  mit Hilfe des *trigonometrischen Polynoms*

$$\Pi(x) = \beta_0 + \beta_1 e^{ix} + \beta_2 e^{2ix} + \dots + \beta_{n-1} e^{(n-1)ix}$$

mit  $\beta_\ell \in \mathbb{C}$  und

$$\Pi(x_k) \stackrel{!}{=} y_k \quad \text{für alle} \quad k = 0, 1, \dots, n-1$$

bestimmen. Mit Hilfe der Beziehung

$$e^{ix} = \cos(x) + i \sin(x)$$

findet man (siehe Vorlesung “Einführung in die Numerik”), dass

$$a_0 = 2\beta_0, \quad a_m = 2\beta_m, \quad a_\ell = \beta_\ell + \beta_{n-\ell}, \quad b_\ell = i(\beta_\ell - \beta_{n-\ell}), \quad \ell = 1, \dots, m-1.$$

Beachte, dass zwar  $P(x_k) = \Pi(x_k) = y_k$  für  $k = 0, \dots, n-1$  gilt, im Allgemeinen aber  $P(x) \neq \Pi(x)$  ist, wenn  $x \neq x_k$ . Durch Nachrechnen kann man zeigen, dass die Koeffizienten  $\beta_\ell$  gegeben sind durch

$$\beta_\ell = \frac{1}{n} \sum_{m=0}^{n-1} y_m \omega_n^{-m\ell}, \quad \ell = 0, 1, \dots, n-1, \quad (2)$$

wobei  $\omega_n := e^{2\pi i/n}$  die  $n$ -te komplexe Einheitswurzel bezeichnet.

## Schnelle Fourier-Transformation

Alle Koeffizienten  $\beta_\ell$  gemäss (2) zu berechnen hat einen Aufwand von  $\mathcal{O}(n^2)$ , was für viele Anwendungen (z.B. in der digitalen Signalverarbeitung) zu teuer ist. Solche Anwendungen benutzen stattdessen die *schnelle Fourier-Transformation*, die nur einen Aufwand von  $\mathcal{O}(n \log_2 n)$  hat.

Schreibt man

$$\mathbf{T}_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \omega_n^2 & \dots & \omega_n^{(n-1)} \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{n-1} \end{bmatrix} \quad \text{und} \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix},$$

so ergibt sich aus (2), dass

$$\boldsymbol{\beta} = \frac{1}{n} \mathbf{T}_n^* \mathbf{y}.$$

Zur Erinnerung:  $\mathbf{T}_n^*$  bedeutet, dass die Matrix  $\mathbf{T}_n$  transponiert und komplex konjugiert wird. Für eine komplexe Zahl der Form  $a + ib$  ist die *komplexe Konjugation* definiert als  $\overline{a + ib} := a - ib$ , d.h. das Vorzeichen des Imaginärteils wird umgekehrt. Es gilt damit  $e^{ix} = \overline{e^{-ix}}$ . In MATLAB für eine Matrix  $\mathbf{B}$  wird der Befehl `conj(B)` zur komplexen Konjugation, `B'` zum transponieren und konjugieren der Matrix in einer Operation und `B.'` zum transponieren der Matrix genutzt.

Die Abbildung  $\mathbf{y} \mapsto \boldsymbol{\beta}$  wird als diskrete Fourier-Transformation bezeichnet, während ihre Umkehrung  $\boldsymbol{\beta} \mapsto \mathbf{y} = \mathbf{T}_n \boldsymbol{\beta}$  Fourier-Synthese genannt wird. Aus der Symmetrie von  $\mathbf{T}_n$  kann man folgern, dass  $\boldsymbol{\beta} = \frac{1}{n} \overline{\mathbf{T}_n \mathbf{y}}$  gilt, weshalb man die Fourier-Transformation mit Hilfe der Fourier-Synthese und vice versa berechnen kann. Die Fourier-Synthese entspricht der Auswertung des trigonometrischen Polynoms  $\Pi(x)$  an den äquidistanten Stellen  $x_k$ .

Das Matrix-Vektor-Produkt  $\mathbf{T}_n \mathbf{y}$  lässt sich besonders effizient mithilfe einer *Divide-and-Conquer-Strategie* berechnen, vorausgesetzt die Anzahl der Stützstellen ist eine Zweierpotenz, d.h.  $n = 2^j$ ,  $j \in \mathbb{N}$  (siehe Vorlesung "Einführung in die Numerik"). Dabei geht man nach folgendem Algorithmus vor:

### Algorithmus (schnelle Fourier-Transformation).

**Input:** Funktionsauswertungen  $\mathbf{y} \in \mathbb{R}^n$  zu Stützstellen  $\{2\pi k/n\}_{k=0}^{n-1}$ ,  $n = 2^j$ ,  $j \in \mathbb{N}$

**Output:** Koeffizientenvektor  $\boldsymbol{\gamma} \in \mathbb{C}^n$

Ist  $n = 2$ , so bilde

$$\boldsymbol{\gamma} = \mathbf{T}_2 \mathbf{y} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \mathbf{y}.$$

Andernfalls setze  $n := n/2$  und

1. berechne die Matrix-Vektor-Produkte

$$\mathbf{a} = \mathbf{T}_n \begin{bmatrix} y_0 \\ y_2 \\ \vdots \\ y_{2n-2} \end{bmatrix} \quad \text{und} \quad \mathbf{b} = \mathbf{T}_n \begin{bmatrix} y_1 \\ y_3 \\ \vdots \\ y_{2n-1} \end{bmatrix}$$

wiederum mit der schnellen Fourier-Transformation,

2. berechne  $\mathbf{c} \in \mathbb{R}^n$  für jedes  $k = 0, 1, \dots, n-1$  als  $\mathbf{c}_k := e^{\pi i k/n} \mathbf{b}_k$ ,

3. setze

$$\begin{bmatrix} \gamma_0 \\ \gamma_1 \\ \vdots \\ \gamma_{n-1} \end{bmatrix} = \mathbf{a} + \mathbf{c} \quad \text{und} \quad \begin{bmatrix} \gamma_n \\ \gamma_{n+1} \\ \vdots \\ \gamma_{2n-1} \end{bmatrix} = \mathbf{a} - \mathbf{c}.$$

Sobald man den Koeffizientenvektor  $\boldsymbol{\gamma}$  bestimmt hat, ist  $\boldsymbol{\beta} = \frac{1}{n} \overline{\boldsymbol{\gamma}}$ . Beachte, dass wenn  $\mathbf{y} \in \mathbb{C}^n$ ,  $\mathbf{y}$  zuerst komplex konjugiert werden muss, bevor man die Fourier-Transformation berechnet.

## Rekursive Funktionen in MATLAB

Der Algorithmus für die schnelle Fourier-Transformation ruft sich im ersten Schritt (Punkt 1 im Algorithmus oben) zweimal selbst auf, d.h. er ist *rekursiv*. In diesem Schritt benutzt man einfach wie gewohnt die Funktion, als ob man sie von einem Skript her aufrufen würde. Als Beispiel für eine rekursive Funktion sei die folgende gegeben, welche rekursiv  $n!$  berechnet:

```
function n_fac = factorial(n)

    if n == 1
        n_fac = 1;
    else
        n_fac = n*factorial(n-1);
    end

end
```

---

Allgemeine Informationen zum Praktikum befinden sich auf der Webseite  
<http://cm.dmi.unibas.ch/teaching/praktikumII/praktikumII.html>