Technische Universität München
Institut für Informatik
Prof. Dr. Hans-Joachim Bungartz
Philipp Samfass
Michael Obersteiner

WiSe 2017 / 2018

Numerisches Programmieren

2. Programmieraufgabe: Bildinterpolation, Fouriertransformation

Motivation

Einige Anwendungen in der Bildbearbeitung, wie z.B. das Vergrößern oder Drehen von Bildern, führen zu einer Veränderung der Pixelmenge bzw. der Pixelstruktur. Daher müssen Inhalte des Bildes an Stellen zwischen Pixeln ausgewertet werden können. In dieser Programmieraufgabe sollen verschiedene Interpolationsverfahren umgesetzt und am Beispiel der Skalierung von Bildern getestet werden. Zusätzlich soll die schnelle (inverse) Fouriertransformation implementiert werden, die die Grundlage vieler moderner numerischer Verfahren ist.

Bildinterpolation

Werden Bilder im RGB-Farbraum dargestellt, enthält jedes Pixel Informationen zu den Grundfarben Rot, Grün und Blau. Für jedes dieser Farbkanäle lässt sich ein Bild als eine Funktion $f : [a, b] \times [c, d] \to [0, 1]$ auffassen, wobei $f(x, y)$ die Intensität der entsprechenden Farbe an der Stelle (x, y) angibt ($0 \mathrel{\widehat{=}}$ keine Farbe, $1 \mathrel{\widehat{=}}$ volle Intensität). Typischerweise steht uns das Bild jedoch nur in diskretisierter Form zur Verfügung, d.h. es wurde bereits in $n \times m$ (gleich große, einfarbige) Rechtecke bzw. Pixel unterteilt und wir kennen die Farbwerte der Pixel. Wir gehen im Folgenden davon aus, dass die Farbe eines Pixels dem Funktionswert in seinem Mittelpunkt entspricht. Außerdem nehmen wir an, dass die gegebenen Pixel Seitenlänge 1 und ihre Mittelpunkte ganzzahlige Koordinaten beginnend bei $(1, 1)$ haben. Somit ist $f : [0.5, n + 0.5] \times [0.5, m + 0.5] \to [0, 1]$ und wir kennen $f(i, j)$ $\forall i = 1 \dots n, j = 1 \dots m$. Abb. 1 zeigt die Pixel und ihre Mittelpunkte.

Wollen wir das Bild nun mit einer höheren Auflösung von $\tilde{n} \times \tilde{m}$ Pixeln darstellen, so unterteilen wir $[0.5, n + 0.5] \times [0.5, m + 0.5]$ in ein neues Raster aus $\tilde{n} \times \tilde{m}$

Rechtecken (s. Abb. 2). Die Farben der neuen Pixel sind die Werte von f in den Mittelpunkten der neuen Pixel. Um sie zu erhalten müssen wir für jeden der drei Farbkanäle f mit Hilfe der Werte aus dem alten Raster interpolieren.

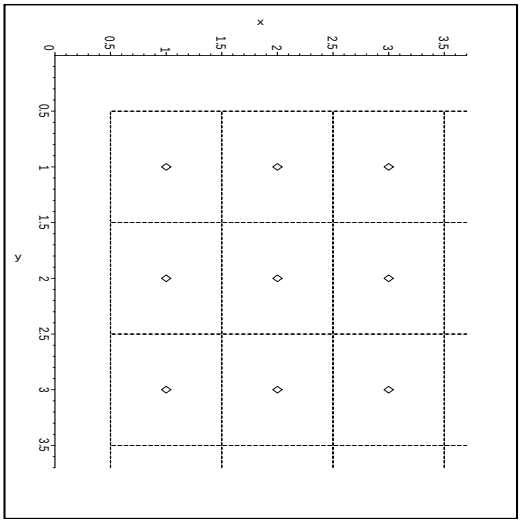

Abb. 1: altes Pixelraster

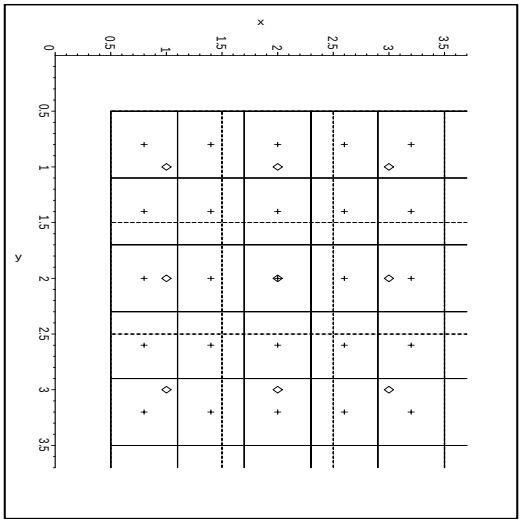

Abb. 2: altes und neues Pixelraster

2D-Interpolation

Für die Bildinterpolation werden Verfahren zur Interpolation im Zweidimensionalen benötigt. Diese Verfahren lassen sich dabei gewöhnlich einfach auf den eindimensionalen Fall zurückführen. Das soll nun am Beispiel der Polynominterpolation veranschaulicht werden.

Analog zum Vorgehen in 1D lassen sich auch zweidimensionale Interpolationspolynome konstruieren. Die Aufgabenstellung ist hierbei zu vorgegebenen Stützstellen (x_i, y_j) und zugehörigen Werten f_{ij} $i, j = 0 \dots n$ (s. Abb. 3) ein Polynom p vom Grad n in x und y zu finden, sodass $p(x_i, y_j) = f_{ij}$ $\forall i, j$ (Abb. 6).

Die Auswertung des 2D-Interpolationspolynoms p an einer Stelle (x^*, y^*) lässt sich durch „Festhalten“ einer Koordinate auf den eindimensionalen Fall zurückführen. Wir halten zunächst die x-Koordinate fest und definieren die Hilfspolynome

$$\tilde{p}_i(y) := p(x_i, y)$$

Die $\tilde{p}_i$ sind (eindimensionale) Polynome n-ten Grades in y und somit durch die Interpolationsbedingung $\tilde{p}_i(y_j) = p(x_i, y_j) = f_{ij}$ $\forall j$ eindeutig festgelegt (Abb. 4). Insbesondere lässt sich somit durch 1D-Interpolation $p(x_i, y^*) = \tilde{p}_i(y^*)$ bestimmen. Als nächstes halten wir nun die y-Koordinate fest und definieren

$$\widehat{p}(x) := p(x, y^*)$$

$\widehat{p}$ ist ein Polynom n-ten Grades in x und durch die Interpolationsbedingung $\widehat{p}(x_i) = p(x_i, y^*) = \tilde{p}_i(y^*)\ \forall i$ festgelegt (Abb. 5). Auswerten von $\widehat{p}$ an der Stelle x^* liefert also schließlich das gesuchte $p(x^*, y^*) = \widehat{p}(x^*)$.

Das Endergebniss (Abb. 6) sieht aus wie das *Interims Hörsaal* Gebäude neben dem FMI in Garching.

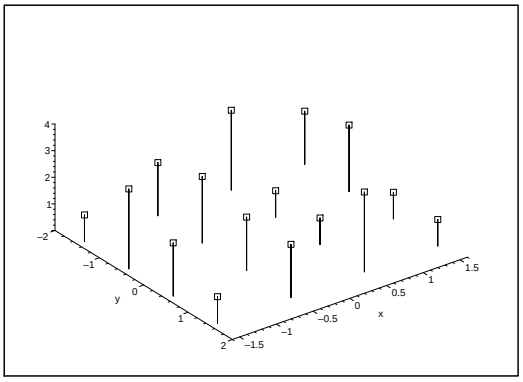

Abb. 3: Stützpunkte

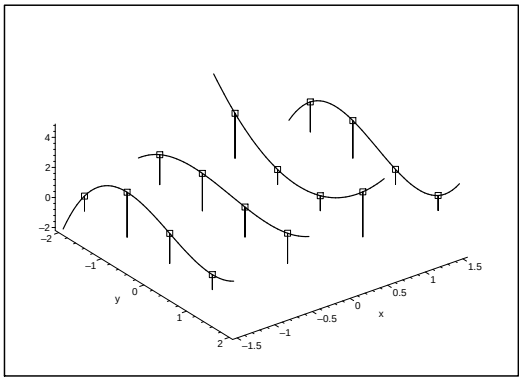

Abb. 4: Die Polynome $\tilde{p}_i$

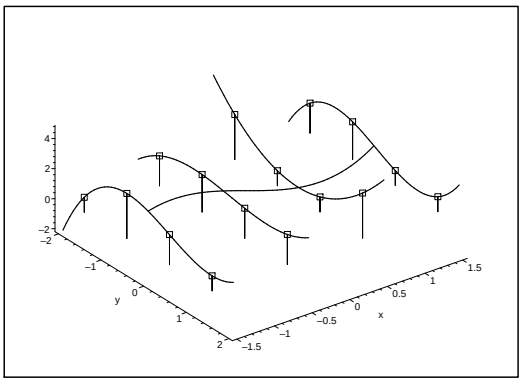

Abb. 5: $\tilde{p}_i$ und $\widehat{p}$ für $y^* = 0$

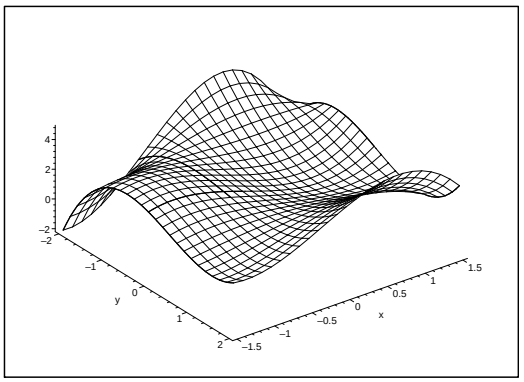

Abb. 6: vollst. Interpolationspolynom

Verschiedene Interpolationsverfahren

In diesem Abschnitt werden nun einige Interpolationsverfahren vorgestellt, d.h. Verfahren, die eine gegebene Menge an Stützpunkten (x_i, y_i) interpoliert und an weiteren Stellen auswertbar ist. In der Bildinterpolation üblich sind die Verfahren Nearest Neighbor, die bilineare (2D lineare Interpolation) und die bikubische Interpolation. Bikubische Verfahren können auf verschieden Splinekonzepten basieren, und werden gewöhnlich nur auf Grundlage des 4x4 Pixelfeldes um den gesuchten Punkt aufgebaut. In dieser Programmieraufgabe wird der kubische

Spline-Ansatz gewählt und auf das gesamte Bild bezogen. Zusätzlich soll sich die Polynominterpolation als Verfahren zur Bildinterpolation versuchen.

Nearest Neighbor

Nearest Neighbor ist ein sehr einfach umzusetzendes Verfahren. Dem gesuchten Punkt wird dabei der naheliegenste Pixel zugeordnet.

Lineare Interpolation

Hierbei wird stückweise zwischen zwei Stützpunkten linear interpoliert, d.h. zwei benachbarte Stützpunkte werden mit einer Geraden verbunden.

Newton-Interpolation

Als Umsetzung der Polynominterpolation dient hier die Newton-Interpolation. Wie aus der Vorlesung bekannt, lassen sich Polynome in Newton-Basis darstellen:

$$p(x) = [x_0]f + [x_0, x_1]f \cdot (x - x_0) + \ldots + [x_0, \cdots, x_n]f \cdot \prod_{i=0}^{n-1}(x - x_i) \ .$$

Die Koeffizienten des Interpolationspolynoms in dieser Darstellung berechnen sich einfach mit Hilfe der dividierten Differenzen. Sie folgen dem Aufbau

$$[x_i]f = y_i$$
$$[x_i, \cdots, x_{i+k}]\, f = \frac{[x_{i+1}, \ldots, x_{i+k}]f - [x_i, \ldots, x_{i+k-1}]f}{x_{i+k} - x_i}$$

und lassen sich in einem Dreiecksschema anordnen:

x_i	$i \setminus k$	0		1		2		$\cdots$
x_0	0	$[x_0]f$	$\rightarrow$	$[x_0, x_1]f$	$\rightarrow$	$[x_0, x_1, x_2]f$	$\rightarrow$	$\cdots$
			$\nearrow$		$\nearrow$			
x_1	1	$[x_1]f$	$\rightarrow$	$[x_1, x_2]f$	$\rightarrow$	$\vdots$		
			$\nearrow$					
x_2	2	$[x_2]f$	$\rightarrow$	$\vdots$				
$\vdots$	$\vdots$	$\vdots$						

Damit kann man in der ersten Zeile direkt die Koeffizienten $[x_0, \cdots, x_k]f$ ablesen.

Ist ein Polynom

$$p(x) = a_0 + a_1 \cdot (x - x_0) + \ldots + a_n \cdot \prod_{i=0}^{n-1} (x - x_i)$$

einmal zur Newton-Basis aufgebaut, lässt es sich sehr effizient mit einem Verfahren ähnlich dem Horner-Schema an einer Stelle x auswerten. Dazu bringt man das Polynom durch Ausklammern der mehrfach auftretenden $(x - x_i)$ in die Form

$$p(x) = a_0 + (x - x_0) \cdot \Big(a_1 + (x - x_1) \cdot \Big(\ldots \Big(a_{n-1} + (x - x_{n-1}) \cdot a_n\Big) \ldots \Big)\Big)$$

und erhält damit automatisch einen iterativen Algorithmus mit einer Laufzeit von $O(n)$.

Es wird zu erwarten sein, dass die Newton-Interpolation bei vielen Stützstellen Schwierigkeiten hat und **große Fehler** produziert. Warum können sich sogar bei der Bildskalierung mit dem Faktor 1 schlechte Ergebnisse ergeben? Wieso findet man in diesem Fall oft links oben noch einen richtig berechneten Ausschnitt?

Kubische Splines

Bei den kubischen Splines gehen wir vereinfacht von äquidistanten Stützstellen mit Intervallbreite h aus. Stückweise zwischen zwei benachbarten Stützstellen x_i und x_{i+1} werden Polynome vom Grad 3 eingefügt, sodass die gesamte Funktion zweifach stetig differenzierbar ist. Kubische Splines lassen sich folgendermaßen aufbauen und auswerten:

Zunächst werden die Ableitungen y'_i an den Stützstellen x_i ermittelt. Als Randbedingung sind dazu die beiden äußersten Ableitungen y'_0 und y'_n gegeben (in unserem Fall der Bildinterpolation setzten wir beide auf 0), alle weiteren erhält man durch lösen des Gleichungssystems

$$\begin{pmatrix} 4 & 1 & & \\ 1 & 4 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 4 \end{pmatrix} \begin{pmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_{n-2} \\ y'_{n-1} \end{pmatrix} = \frac{3}{h} \begin{pmatrix} y_2 - y_0 - \frac{h}{3} y'_0 \\ y_3 - y_1 \\ \vdots \\ y_{n-1} - y_{n-3} \\ y_n - y_{n-2} - \frac{h}{3} y'_n \end{pmatrix}.$$

Aufgrund seiner tridiagonalen Struktur lässt sich dieses System mit einer Laufzeit von $O(n)$ (Thomas-Algorithmus) lösen, statt der für die Gaußelimination üblichen $O(n^3)$. Mit diesen Ableitungen können nun die stückweisen Polynome aufgebaut werden.

Sollen nun die Splines an einer Stelle x ausgewertet werden, kann man wie folgt vorgehen:

i) Finde das Intervall $I = [x_i, x_{i+1}]$ mit $x \in I$ und damit das entsprechende Polynom.

ii) Transformiere x in das Intervall $[0, 1]$ mit Hilfe von:

$$t = t_i(x) = \frac{x - x_i}{h} \text{ mit } h = x_{i+1} - x_i$$

iii) Werte nun das Polynom q zur Hermite-Basis in t aus:

$$q(t) = y_i \cdot H_0(t) + y_{i+1} \cdot H_1(t) + h \cdot y_i' \cdot H_2(t) + h \cdot y_{i+1}' \cdot H_3(t)$$

$$\begin{aligned} \text{mit } H_0(t) &= 1 - 3t^2 + 2t^3 \\ H_1(t) &= 3t^2 - 2t^3 \\ H_2(t) &= t - 2t^2 + t^3 \\ H_3(t) &= -t^2 + t^3. \end{aligned}$$

Schnelle (inverse) Fourier-Transformation

Aus der Vorlesung und den Tutorübungen (insb. Aufgabenblatt 5) ist die diskrete und die diskrete inverse Fourier-Transformation bereits bekannt. Da die Implementierung der diskreten (inversen) Fourier-Transformation eine Laufzeit von $O(n^2)$ hat, wird sie in vielen Programmen durch die schnelle (inverse) Fourier-Transformation mit einer Laufzeit von $O(n \log n)$ ersetzt.

In dieser Programmieraufgabe beschänken wir uns auf die schnelle inverse Fourier-Transformation ($IFFT$). Den Pseudocode dazu finden Sie in der Vorlesung.

Das Programmgerüst enthält bereits eine Implementierung der diskreten Fourier-Transformation. Zum testen Ihres Codes können Sie sich folgende Gleichung zunutze machen:

$$IFFT(DFT(v)) = v$$

Für das Verständnis der Fourier-Transformation sind komplexe Zahlen essentiell.

Komplexe Zahlen

Komplexe Zahlen erweitern den Zahlenbereich der reellen Zahlen, sodass die Gleichung $x^2 = -1$ lösbar wird. Die Lösung dieser Gleichung ist die Zahl i, also $i^2 = -1$. Die Zahl i wird auch als imaginäre Einheit bezeichnet.

Alle komplexen Zahlen lassen sich in der Form $a + b \cdot i$ darstellen, wobei a und b reelle Zahlen sind. Komplexe Zahlen lassen sich außerdem (ähnlich wie reelle Zahlen auf einem Zahlenstrahl) in der komplexen Ebene eintragen (siehe Abbildung 7). Dabei entspricht der reelle Anteil a dem Wert auf der "x-Achse" und der imaginäre Anteil dem Wert auf der "y-Achse".

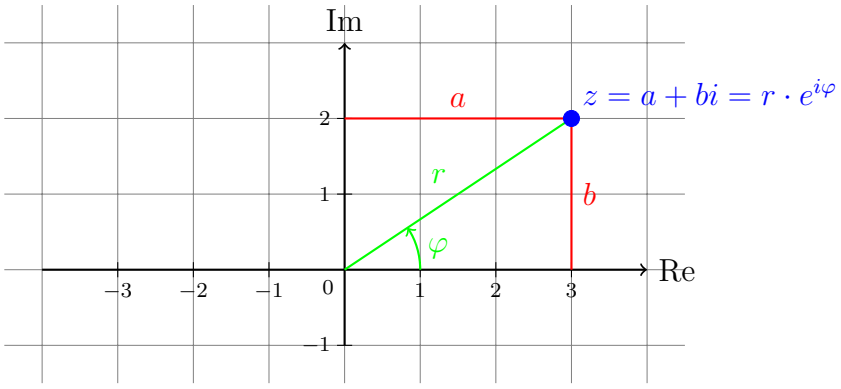

Abb. 7: Visualisierung einer komplexen Zahl z.

Neben der Form $a + bi$ lässt sich jede komplexe Zahl auch eindeutig in Polarkoordinaten $r \cdot e^{i\varphi}$ darstellen. Aus der komplexen Ebene ist diese Darstellung leicht Zahl ersichtlich: r entspricht dem Abstand zum Ursprung – dem Radius – und φ dem Winkel zur positiven reellen Achse.

Mit der folgenden Formel lassen sich aus den Polarkoordinaten einer komplexen Zahl z der reelle Teil a und der imaginäre Teil b berechnen:

$$z = r \cdot e^{i\varphi} = r \cdot (\cos(\varphi) + i \cdot \sin(\varphi))$$

In komplexen Zahlen lassen sich "Wurzeln von Eins" darstellen, die für die Fouriertransformation wichtig sind. Die n-ten Wurzeln von Eins sind die komplexen Zahlen $z_k = e^{2\pi i k/n}$, $0 \leq k < n$. Für diese gilt:

$$1 = z_k^n$$

Rechenoperationen

Addition

$$x + y = (a + bi) + (c + di) = (a + c) + (b + d)i$$

Subtraktion

$$x - y = (a + bi) - (c + di) = (a - c) + (b - d)i$$

Multiplikation [1]

$$x \cdot y = (a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$$

Division

$$\frac{x}{y} = \frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

Potenz mit ganzzahligem Exponenten

Mit dem Satz von Moivre (*Mathematik für Physiker, Kerner u. Wahl, 2013*) lassen sich leicht ganzzahlige Potenzen von komplexen Zahlen in Polarkoordinaten berechnen:

$$z^n = r^n \cdot (e^{i\varphi})^n = r^n \cdot (\cos\varphi + i\sin\varphi)^n = r^n \cdot (\cos(n\varphi) + i\sin(n\varphi)) = r^n \cdot e^{in\varphi}$$

Achtung: Auf Grund der Periodizität von sin und cos lässt sich diese Formel nicht ohne weiteres auf rationale oder reelle Exponenten erweitern!

[1] Aus dieser Formel können Sie leicht die Definition $i^2 = -1$ ableiten.

Beschreibung des Programmgerüsts

Das auf der Vorlesungsseite zur Verfügung gestellte Programmgerüst enthält:

- `NearestNeighbour`, `LinearInterpolation`, `NewtonPolynom`, `CubicSpline`:
  Diese Klassen implementieren alle das Interface `InterpolationMethod` und bieten Funktionalitäten zum Aufbau (`init()`) und zur Auswertung (`evaluate()`) der entsprechenden Interpolationsmethode.
- `TridiagonalMatrix`:
  Diese Klasse beschreibt eine Tridiagonalmatrix und ermöglicht das Lösen (`solveLinearSystem(double[] b)`) von linearen Gleichungssystemen mit Tridiagonalgestalt und rechter Seite b. Eine Tridiagonalmatrix besitzt nur Einträge ungleich Null in der Diagonalen und den beiden Nebendiagonalen.
- `Picture`:
  Diese Klasse verwaltet ein Bild im RGB-Format. Sie ermöglicht das Manipulieren und Skalieren (`scale()`) des Bildes. Für die Bearbeitung dieser Programmieraufgabe ist ein Verständnis dieser Klasse **nicht** erforderlich.
- `ImageViewer`:
  Diese Klasse enthält eine `main()`-Methode und implementiert eine einfache Benutzeroberfläche zum Laden und Skalieren von Bildern mit verschiedenen Interpolationsmodi.
- `InterpolationsPlotter`:
  Diese Klasse enthält eine `main()`-Methode und implementiert eine einfache Benutzeroberfläche zur Darstellung von Interpolationsfunktionen.
- `Complex`:
  Die Klasse repräsentiert eine komplexe Zahl. Gespeichert wird die Zahl als Realteil und Imaginärteil. Außerdem werden alle wichtigen Operatoren (add, sub, mul, power) bereit gestellt sowie Funktion zum Konvertieren von bzw. in Polarkoordinaten.
- `DFT/IFFT`:
  Die Klassen enthalten Funktionen für die diskrete Fourier-Transformation und schnelle inverse Fourier-Transformation.
- `InterpolationsPlotter`:
  Diese Klasse enthält eine `main()`-Methode und implementiert eine einfache Benutzeroberfläche zur Darstellung von Interpolationsfunktionen.

Aufgaben

Im Folgenden werden die zu implementierenden Methoden aufgelistet. Details finden Sie jeweils in den Kommentaren zu den einzelnen Methoden. Überlegen Sie sich zu allen Methoden Testbeispiele und testen Sie unbedingt jede einzelne Methode direkt nach der Erstellung. Sie können zusätzlich die zur Verfügung gestellte Testklasse `Test` verwenden. Es ersetzt aber nicht das Verwenden eigener Testbeispiele.

- Programmieren Sie in der Klasse `LinearInterpolation` den gegebenen Methodenrumpf `evaluate`.
- Programmieren Sie in der Klasse `NewtonPolynom` die gegebenen Methodenrümpfe `computeCoefficients`, `addSamplingPoint` und `evaluate`.
- Programmieren Sie in der Klasse `CubicSpline` die gegebenen Methodenrümpfe `computeDerivatives` und `evaluate`.
- Programmieren Sie in der Klasse `Complex` die gegebenen Methodenrümpfe `add`, `sub`, `mul` und `fromPolar`.
- Programmieren Sie in der Klasse IFFT den gegebenen Methodenrumpf `ifft`. Den Pseudocode dazu finden Sie in den Vorlesungsfolien.

Formalien und Hinweise

- Das Programmgerüst erhalten Sie auf den Webseiten zur Vorlesung.
- Ergänzen Sie das Programmgerüst bitte **nur an den dafür vorgegebenen Stellen!** Falls Sie die Struktur der Programme an anderer Stelle verändern, können wir sie evtl. nicht mehr testen.
- Beseitigen Sie vor Abgabe Ihres Programms alle Ausgaben an die Konsole und reichen Sie bis zum **1. Dezember 2017, 12:00 Uhr** über Moodle ein.
- Reichen Sie nur die Dateien `LinearInterpolation.java`, `NewtonPolynom.java`, `CubicSpline.java`, `Complex.java` und `IFFT.java` ein. Die Zuordnung zum Package `dft` muss nicht entfernt werden.
- Bitte laden Sie Ihre java-Dateien als flaches tgz-Archiv hoch. Der Dateiname ist beliebig wählbar, bei der Erweiterung muss es sich jedoch um .tgz oder .tar.gz handeln.

  Ein solches Archiv können Sie beispielsweise mit dem Linux-Tool tar erstellen, indem Sie die laut Aufgabenstellung zu bearbeitenden java-Dateien in ein sonst leeres Verzeichnis legen und dort anschließend den Befehl

```
> tar cvvzf numpro_aufg2.tgz *.java
```

ausführen.

- Wir empfehlen Ihnen, die Programme unter Linux (Rechnerhalle) zu testen.

Bitte beachten Sie: *Alle Abgaben, die nicht den formalen Kriterien genügen, werden grundsätzlich mit 0 Punkten bewertet!*