IGS:FF
Integrierte Gesamtschule
Franzsches Feld

Grünewaldstraße 12a - 38104 Braunschweig - T. 470 5850

Seminarfach – Informatik (Herr Heydecke)

Facharbeit der Schülerin/des Schülers

Jakob Rzeppa
mit dem Thema:

Numerische Berechnung der Nullstellen von Polynomfunktionen mit Java

Lehrerin / Lehrer: Marc Heydecke

Ausgabetermin des Themas: 25.09.2023
Abgabetermin der Facharbeit: 24.11.2023

Bewertung:

Datum, Unterschrift der Schülerin/des Schülers	Datum, Unterschrift der Lehrerin/des Lehrers

Die Facharbeit
Formvorschriften und Aufbau

Umfang (maximale Seitenangaben)

Einzelarbeit:	13-15 Textseiten in Maschinenschrift
Partnerarbeit (zu zweit):	19-21 Textseiten in Maschinenschrift

(Maschinenschrift: Arial, Calibri, Times New Roman)
Bei Partnerarbeiten muss das Thema so formuliert sein, dass inhaltlich selbständige und nicht nur arbeitsteilig abgrenzbare Unterthemen bearbeitet werden können und auf diese Weise die individuelle Einzelleistung erkennbar bleibt.

Formvorschriften

- **Din A 4-Blätter**, einseitig beschrieben, mit Heftungsrand (3cm), rechts: 3-4 cm, oben: 2,5 cm unten 2,5 cm
- nur in Maschinenschrift! (Schriftgrad 12)
- Zeilenabstand 1,5-fach
- **erste Seite: Deckblatt** mit Namen, Thema der Arbeit und Seminarfach
- **zweite Seite**: (Formblatt1), dritte Seite: (Formblatt2)
- **Inhaltsverzeichnis** (= Seite 1 gezählter Umfang)
- **Literaturverzeichnis** (zählt nicht mehr zum gezählten Umfang)
- **Erklärung am Ende der Arbeit** auf gesondertem Blatt: (zählt nicht zum gez. Umfang)
  „Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe."
  Datum, Unterschrift
- **Einverständnis zur Publikation**, wenn die Schülerin/der Schüler zustimmt.
  „Hiermit erkläre ich, dass ich damit einverstanden bin, wenn die von mir verfasste Facharbeit der schulinternen Öffentlichkeit zugänglich gemacht wird."

Aufbau/Anlage - Richtlinie

- **Einleitung**

  Problemstellung, Leitfrage, Abgrenzung des Themas, Nennung und Begründung der gewählten Arbeitsweisen und Methoden, Inhaltsübersicht, Erkenntnisabsicht

- **Hauptteil**

  Stand des Problems aufgrund der verwendeten Fachinformation, Beschreibung der eigenen Untersuchung in straffer Gliederung, Formulierung der Ergebnisse, kritische Reflexion der Methode und Ergebnisse, offen gebliebene Fragen, Widersprüche

- **Schluss, Fazit**

  Zusammenfassung und abschließende Überlegungen, Reflexion über das eigene methodische Vorgehen, Erkenntnisgewinn, Rückbezug auf die Leitfrage, die Problemstellung aus der Einleitung

- **Materialien im Anhang**

  Bilder, besondere Quellen, Kartenmaterial usw.

Bewertungskriterien

Die fachspezifischen Bewertungsmaßstäbe und Beurteilungskriterien sowie das Verhältnis der einzelnen Bewertungsmaßstäbe zueinander entsprechen den Grundsätzen für die Bewertung und Beurteilung von schriftlichen Arbeiten in der Sek II.
In folgenden Bereichen werden die Kenntnisse, Fähigkeiten und Fertigkeiten bewertet:

1) Formale Anlage

- eine nachvollziehbare Dokumentation anzufertigen
- einen Text formal und sprachlich korrekt und mit Sorgfalt anzulegen
- korrekt zu zitieren
- ein Literaturverzeichnis übersichtlich anzulegen und formale Regeln bei der Erstellung von Diagrammen u. ä. einzuhalten
- eine deutliche Gliederung anzufertigen

2) Methodische Durchführung

- Beherrschung der fachspezifischen Terminologie. Methoden und Arbeitstechniken
- Auswahl von themenbezogener Literatur
- sach- und problembezogener Einsatz von Zitaten, veröffentlichten Beweisen, Berechnungen, Statistiken, Bildmaterial.....
- sach- und problemgerechte Gliederung
- folgerichtige Argumentation
- zweckentsprechender Einsatz, bzw. Herstellung von Materialien
- Sachverhalte begrifflich präzise darstellen und das gewählte Vorgehen reflektieren

3) Inhaltliche Bewältigung

- auf dem durch das Thema begrenzte Sachgebiet selbständig zu Ergebnissen kommen (Richtigkeit des methodischen Vorgehens und der Anwendung fachspezifischer Verfahren)
- die Problemstellung richtig zu erfassen, zu analysieren und darzustellen
- Konzentration auf spezifische Aspekte des Themas, die Problembereiche differenziert und begründet zu beurteilen
- Entwicklung eines Lösungswegs zur Problemstellung
- eigene Projekte oder Versuche planen und durchführen und gewonnene Daten analysieren und bewerten
- zur logischen Verknüpfung der einzelnen Gedanken oder Beweisschritte, zu originellen oder kreativen Ergebnissen kommen
- kritische Reflexion hinsichtlich der angewandten Verfahren
- Entwicklung einer begründeten Stellungnahme
- *Anforderungsbereiche I, II, III müssen berücksichtigt werden*

Inhaltsverzeichnis

1 Einleitung

Mithilfe von Nullstellen kann viel über den Verlauf einer Polynomfunktion herausgefunden werden. Polynome ersten bis vierten Grades lassen sich mithilfe von Formeln wie der pq-Formel exakt berechnen. Für Polynome des Grades $n > 4$ gibt es keine allgemeine Formel zur Bestimmung der Nullstellen. Für diese muss auf einen anderen Teilbereich der Mathematik zurückgegriffen werden: die Numerik. In der Numerik versucht man durch Algorithmen und Annäherungen Nullstellen möglichst genau zu bestimmen. Händisch kann dies sehr lange dauern, jedoch kann mit Computern dieser Prozess extrem beschleunigt werden. Doch wie kann man Nullstellen von Polynomen mithilfe des Computers approximieren?

Ziel dieser Facharbeit ist es, eine Möglichkeit der Implementierung eines numerischen Verfahrens zu entwickeln und dieses darzustellen. Dafür wird zuerst ein Verfahren ausgesucht und ausführlich beschrieben. Dabei werden weiterführend die Herleitung, Probleme und Einschränkungen behandelt und das Verfahren wird anhand eines Beispiels verdeutlicht. Im letzten Teil der Facharbeit wird meine Implementierung eines Lösungsansatzes beschrieben und erklärt.
Dieses Thema habe ich gewählt, da ich mich seit der Einführung von Polynomfunktionen im Matheunterricht für das Bestimmen der Nullstellen von Polynomfunktionen höheren Grades interessiere.
Für die Implementierung des Verfahrens wird die Programmiersprache Java gewählt. Java bietet sehr gute Möglichkeiten für die objektorientierte Programmierweise und die Syntax ist leicht verständlich.
Um ein Verfahren auszuwählen, dieses zu erklären und zu implementieren, werde ich neben Fachliteratur und Informationen von unterschiedlichen Universitäten Informationen aus dem Internet nutzen.

2 Auswahl des Verfahren

2.1 Vorstellung der Verfahren

In der Numerik gibt es viele verschiedene Verfahren für die Nullstellenberechnung. Die bekanntesten Verfahren sind die Newton-Methode[1] und das Bisektionsverfahren bzw. der Zwischenwertsatz[2]. Beide Methoden sind jedoch nur für das Finden einer Nullstelle geeignet. Da das Ziel ist, alle Nullstellen eines Polynoms zu finden, kommen sie nicht infrage.
Wenn man alle Nullstellen finden möchte, muss man komplexere Verfahren in Betracht ziehen. Dabei bieten sich Weiterentwicklungen der Newton-Methode wie die Weierstraß-Iteration (Durand-Kerner-Methode)[3], das Aberth-Ehrlich-Verfahren[4] oder das Newton-Horner-Verfahren[5] an.

2.2 Vergleich der möglichen Verfahren

Wichtige Kriterien bei dem Vergleich von numerischen Verfahren sind die Geschwindigkeit und Einfachheit der Implementierung.
Die Weierstraß-Iteration und das Aberth-Ehrlich-Verfahren finden Nullstellen simultan, während das Newton-Horner-Verfahren die Nullstellen nacheinander findet. Aus diesem Grunde kommen sie mit weniger Iterationen aus. Das muss nicht immer bedeuten, dass diese Verfahren schneller sind. Jedoch können Prozesse parallel ausgeführt werden, was zu großen Verbesserungen der Performance führen kann.
Das Aberth-Ehrlich-Verfahren ist deutlich schneller als die Weierstraß-Iteration, benutzt jedoch eine komplexere Gleichung und ist deshalb schwieriger zu implementieren.
Das Newton-Horner-Verfahren ist schwierig zu implementieren, da es eine sehr hohe bis exakte Genauigkeit benötigt, um nach dem Finden einer Nullstelle diese von der Funktion zu entfernen.

[1] Prof. Dr. Bernd Engelmann: Numerische Mathematik, 2.4 Das Newton-Verfahren und seine Abkömmlinge (S.24)

[2] mathe-online.at: Zwischenwertsatz und Bisektionsverfahren, URL: `https://www.mathe-online.at/nml/materialien/innsbruck/bisektion/Bisektion.pdf`

[3] vgl. Wikipedia: Durand-Kerner method, URL: `https://en.wikipedia.org/wiki/Durand-Kerner_method` (Zuletzt aufgerufen: 29.10.2023)

[4] Wikipedia: Aberth method, URL: `https://en.wikipedia.org/wiki/Aberth_method` (Zuletzt aufgerufen: 18.11.2023)

[5] Eine Kombination aus der Newton-Methode und dem Horner-Verfahren.

2.3 Fazit

Aufgrund der zuvor genannten Schwächen und Stärken der einzelnen Methoden wähle ich die Weierstraß-Iteration. Dabei hat die Implementierbarkeit eine größere Rolle als die Geschwindigkeit gespielt, da die Geschwindigkeit nur bei sehr komplexen Polynomen wichtig ist und diese meist nicht vorkommen.

3 Weierstraß-Iteration

3.1 Funktionsweise

3.1.1 Bedingungen

Für die Weierstraß-Iteration muss ein normiertes univariates Polynom der Form $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ gegeben sein. Der Grad des Polynoms n muss größer als oder gleich zwei sein.[6]

3.1.2 Gleichungen für die Weierstraß-Iteration

Bei der Weierstraß-Iteration wird mit jeder Iteration jede Nullstelle in der Regel[7] etwas genauer. Dafür werden n Gleichungen für die n Nullstellen[8] $z_n, z_{n-1}, \ldots, z_1$ gebildet. Über diese wird iteriert, bis das Endkriterium erreicht ist. Dabei ist i die Anzahl der Iterationen und wird pro Iteration um 1 inkrementiert.

$$z_n^{(i+1)} = z_n^{(i)} - \frac{p(z_n^{(i)})}{\prod_{j=1;j\neq n}^{n}(z_n^{(i)} - z_j^{(i)})}$$

$$z_{n-1}^{(i+1)} = z_{n-1}^{(i)} - \frac{p(z_{n-1}^{(i)})}{\prod_{j=1;j\neq n-1}^{n}(z_{n-1}^{(i)} - z_j^{(i)})}$$

$$\ldots$$

$$z_1^{(i+1)} = z_1^{(i)} - \frac{p(z_1^{(i)})}{\prod_{j=1;j\neq 1}^{n}(z_1^{(i)} - z_j^{(i)})}$$

[6] vgl. Wikipedia: Weierstraß-(Durand-Kerner)-Verfahren, URL: `https://de.wikipedia.org/wiki/Weierstra-(Durand-Kerner)-Verfahren` (Zuletzt aufgerufen: 29.10.2023)

[7] vgl. Abschnitt 3.4.2

[8] Eine Funktion n-ten Grades hat nach dem Fundamentalsatz der Algebra genau n komplexe Nullstellen.

3.1.3 Startpunkte

Zunächst müssen alle Startpunkte $z_n^{(0)}, z_{n-1}^{(0)}, \ldots, z_1^{(0)} \in \mathbb{C}$ gesetzt werden. Diese können beliebig gewählt werden. Es muss jedoch beachtet werden, dass Startpunkte für komplexe Nullstellen einen imaginären Teil ungleich Null haben müssen. Außerdem müssen alle Startpunkte unterschiedlich sein, damit bei der ersten Iteration nicht durch Null dividiert wird. Denn bei der Gleichung

$$z_k^{(i+1)} = \frac{p(z_k^{(i)})}{\prod_{j=1;j\neq k}^{n}(z_k^{(i)} - z_j^{(i)})}$$

wird, wenn $z_k^{(i)}$ gleich einem anderen $z^{(i)}$ ist, durch Null dividiert. Die Startpunkte können allerdings auch die Anzahl der Iterationen beeinflussen. Deswegen macht es Sinn, möglichst nahe an den wahrscheinlichen Nullstellen anzufangen.
Eine der gängigsten Methoden ist, die Startpunkte in einem Kreis auf der komplexen Ebene zu verteilen. Dabei ist der Radius $r = \sqrt[n]{|a_0|}$ relativ genau. Jedoch ist es sehr ressourcenintensiv, Wurzeln zu berechnen. Deshalb kann auch ein ungenauerer Radius genommen werden: $r = |\frac{na_0}{2a_1}| + |\frac{a_{n-1}}{2n}|$. Auf dem Kreis werden alle Startpunkte gleichmäßig verteilt, indem der Kreis in n Abschnitte geteilt wird, an dessen Anfängen die Startpunkte liegen. Dafür wird das Inkrement $\theta = \frac{2\pi}{n}$ bzw. $\frac{360°}{n}$ benötigt. Zuletzt wird noch eine Verschiebung c durchgeführt, um keine reellen Startpunkte zu erhalten. Diese kann beliebig gewählt werden, solange keine reellen Zahlen nach der Verschiebung unter den Startpunkten sind. Bei der folgenden Implementierung wurde $c = \frac{\pi}{2n}$ gewählt. Um die Startpunkte $z_n^{(0)}, z_{n-1}^{(0)}, \ldots, z_1^{(0)}$ zu bestimmen, wird

$$z_k^{(0)} = r\cos((k-1)\theta + c) + r\sin((k-1)\theta + c)i \text{ für } k = 1, 2, \ldots, n$$

angewandt.[9]

3.1.4 Endkriterium

Um zu bestimmen, ob eine Nullstelle gefunden wurde, muss eine gewünschte Genauigkeit g (z. B. $0,0001$) gewählt werden. Diese wird dann mit dem Betrag der Differenz der letzten beiden Schritte der Weierstraß-Iteration verglichen. Wenn $|z_k^{(i-1)} - z_k^{(i)}| < g$ gilt, dann ist die Nullstelle z_k mit der Genauigkeit g gefunden. Dies wird für alle Null-

[9] vgl. Oscar Veliz: Durand-Kerner Method: Minute 5:40 (Veröffentlicht am 29.05.2019), URL: `https://www.youtube.com/watch?v=5JcpOj2KtWc` (Zuletzt aufgerufen: 30.10.2023)

stellen wiederholt. Falls alle Nullstellen genau genug sind, ist die Weierstraß-Iteration abgeschlossen.
Es kann jedoch auch vorkommen, dass die Weierstraß-Iteration nicht konvergiert. Darauf wird im Abschnitt 3.4.2 weiter eingegangen. Daher muss eine weitere Endbedingung gestellt sein: eine maximale Anzahl an Iterationen. Dabei muss man wissen, mit welche Polynome vorliegen. Sind diese sehr komplex, kann es vorkommen, dass die Weierstraß-Iteration weit über tausend Iterationen benötigt, um eine vernünftige Genauigkeit zu erreichen. In solchen Fällen muss man das Limit sehr hoch setzen. In meiner Implementierung wurde ein Limit von 1000 gesetzt, um für die meisten Polynome zuverlässig zu funktionieren.

3.2 Beispiel

Im Folgenden wird die Weierstraß-Iteration an einer Polynomfunktion vierten Grades beispielhaft durchgeführt. Dabei wird die Genauigkeit im Beispiel auf vier Nachkommastellen begrenzt, was zu Ungenauigkeiten kleiner als $0,0001$ führen kann.

Gleichungen Für die Funktion

$$p(x) = x^4 + 4x^3 - 2x^2 + 3x - 4 = (x - z_1)(x - z_2)(x - z_3)(x - z_4)$$

sind die Nullstellen $z_1, z_2, z_3, z_4 \in \mathbb{C}$ gesucht. Für jede der Nullstellen wird, wie zuvor beschrieben, eine Gleichung gebildet:

$$z_1^{(i+1)} = z_1^{(i)} - \frac{p(z_1^{(i)})}{(z_1^{(i)} - z_2^{(i)})(z_1^{(i)} - z_3^{(i)})(z_1^{(i)} - z_4^{(i)})}$$

$$z_2^{(i+1)} = z_2^{(i)} - \frac{p(z_2^{(i)})}{(z_2^{(i)} - z_1^{(i)})(z_2^{(i)} - z_3^{(i)})(z_2^{(i)} - z_4^{(i)})}$$

$$z_3^{(i+1)} = z_3^{(i)} - \frac{p(z_3^{(i)})}{(z_3^{(i)} - z_1^{(i)})(z_3^{(i)} - z_2^{(i)})(z_3^{(i)} - z_4^{(i)})}$$

$$z_4^{(i+1)} = z_4^{(i)} - \frac{p(z_4^{(i)})}{(z_4^{(i)} - z_1^{(i)})(z_4^{(i)} - z_2^{(i)})(z_4^{(i)} - z_3^{(i)})}$$

Startpunkte Zuvor müssen allerdings die Startpunkte $z_1^{(0)}, z_2^{(0)}, z_3^{(0)}, z_4^{(0)}$ bestimmt werden. Dabei ist der Radius des Kreises $r = |\frac{na_0}{2a_1}| + |\frac{a_{n-1}}{2na_n}| = \frac{19}{6}$, der Abstand zwischen

den Startpunkten $\theta = \frac{2\pi}{n} = \frac{1}{2}\pi$ und die Verschiebung $c = \frac{\pi}{2n} = \frac{1}{8}\pi$. Mit diesen Werten können die Startpunkte berechnet werden.
$z_1^{(0)} = r\cos(c) + r\sin(c)i = 2,9256 + 1,2118i$
$z_2^{(0)} = r\cos(\theta + c) + r\sin(\theta + c)i = -1,2118 + 2,9256i$
$z_3^{(0)} = r\cos(2\theta + c) + r\sin(2\theta + c)i = -2,9256 - 1,2118i$
$z_4^{(0)} = r\cos(3\theta + c) + r\sin(3\theta + c)i = 1,2118 - 2,9256i$

Ausführung

Iteration	z_1	z_3	z_2	z_4
0	2,9256 + 1,2118i	-1,2118 + 2,9256i	-2,9256 - 1,2118i	1,2118 - 2,9256i
1	1,2993 + 0,8722i	-1,8873 + 2,0075i	-3,405 - 0,7665i	-0,0069 - 2,1133i
2	1,0806 + 0,5889i	-1,1784 + 0,4767i	-4,1287 + 0,3433i	0,2265 - 1,4089
3	0,7884 + 0,3817i	-0,2138 + 0,7292i	-4,6806 - 0,0989	0,1061 - 1,012i
4	0,6759 - 0,234i	0,0243 + 0,9804i	-4,613 - 0,0021i	-0,0872 - 0,9549i
5	0,8621 + 0,0197i	-0,1294 + 0,9945i	-4,6149	-0,1177 - 1,0141i
6	0,842 + 0,0004i	-0,1134 + 1,0078i	-4,6149	-0,1136 - 1,0082i
7	0,8421	-0,1136 + 1,0081i	-4,6149	-0,1136 - 1,0081i
8	0,8421	-0,1136 + 1,0081i	-4,6149	-0,1136 - 1,0081i

Endkriterium Zwischen Iteration sieben und acht ist die Differenz zwischen den beiden Werten für jede Nullstelle kleiner als die Genauigkeit $0,0001$. Damit ist das Endkriterium erreicht und die Weierstraß-Iteration abgeschlossen. Die Nullstellen der Polynomfunktion $p(x) = x^4+4x^3-2x^2+3x-4$ sind bei ca. $0,8421; -4,6149; -0,1136-1,0081i; -0,1136+1,0081$.

Probe Zur Überprüfung kann jeder der Werte in $p(x)$ eingesetzt werden.

$$p(0,8421) \approx 0, p(-4,6149) \approx 0, p(-0,1136 - 1,0081i) \approx 0, p(-0,1136 + 1,0081i) \approx 0$$

Daraus kann geschlossen werden, dass alle vier Werte Annäherungen der Nullstellen der Polynomfunktion $p(x)$ sind.
Weiterhin kann aus dem Fundamentalsatz der Algebra hergeleitet werden, dass alle Nullstellen von $p(x)$ gefunden wurden, da n gleich der Anzahl der gefundenen Nullstellen ist.
$\Rightarrow$ Die Weierstraß-Iteration war für $p(x) = x^4 + 4x^3 - 2x^2 + 3x - 4$ erfolgreich.

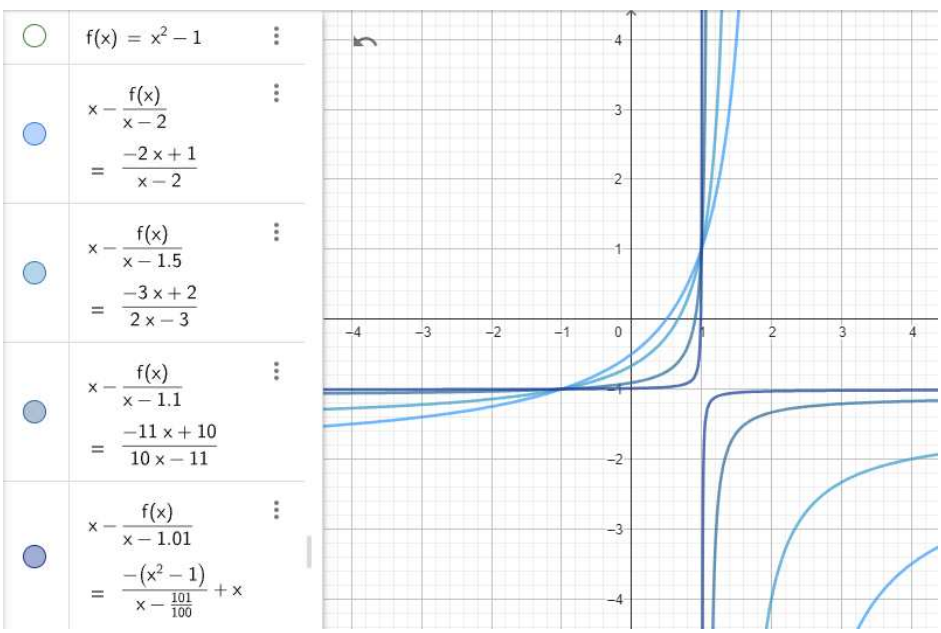
3.3 Herleitung

3.3.1 Methode

Auf die Herleitung der Methode wird der Einfachheit halber nur mit reellen Zahlen eingegangen. Jedoch kann die Herleitung mit wenigen Veränderungen auch auf komplexe Zahlen übertragen werden.

Visualisierung Für die Herleitung der Methode macht es Sinn, den Graphen einer einzelnen Gleichung mit immer genaueren restlichen Nullstellen zu visualisieren. Dabei ist $z_k^{(i)} = x$ und $z_k^{(i+1)} = y$:

$$g(x) = y = x - \frac{p(x)}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})} \tag{1}$$



Dabei sieht man, dass es sich, wenn die restlichen Nullstellen nicht genau sind, um eine gebrochen rationale Funktion handelt. Das kann auch an der Gleichung gesehen werden, da diese mit ein wenig umstellen aus einem Polynom geteilt durch ein anderes

Polynom besteht:

$$\begin{aligned} g(x) = y &= x - \frac{p(x)}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})} \\ &= \frac{x}{1} - \frac{\prod_{j=1}^{n}(x - z_j^{(i)})}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})} \\ &= \frac{x \cdot \prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})} - \frac{\prod_{j=1}^{n}(x - z_j^{(i)})}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})} \\ &= \frac{x \cdot \prod_{j=1;j\neq k}^{n}(x - z_j^{(i)}) - \prod_{j=1}^{n}(x - z_j^{(i)})}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})} \end{aligned}$$

Wenn die restlichen Nullstellen gefunden wurden, handelt es sich um eine waagrechte lineare Funktion. Das liegt daran, dass alle linearen Faktoren des Polynoms außer dem der gesuchten Nullstelle gefunden wurden und nun von dem Polynom dividiert werden. Dabei bleibt $x - (x + z_k^{(i)})$ über. Die beiden x fallen weg und es bleibt der konstante Wert $z_k^{(i)}$ über. Daraus folgt, dass die Funktion waagrecht zu der x-Achse ist.

Bedeutung für die Weierstraß-Iteration Das Annähern an eine Nullstelle mit jeder Iteration kann man sich mit zwei miteinander verknüpften Annäherungen vorstellen. Einerseits das Annähern an die Nullstelle mit der jeweiligen Gleichung für die Nullstelle und andererseits das Annähern der Funktion $g(x)$ an die Waagrechte $y = z_k$, mit immer genaueren restlichen Nullstellen.

Annäherung an die Waagrechte $y = z_k$ Wie in der Abbildung zuvor zu sehen, nährt sich die Funktion mit genaueren Nullstellen an die Funktion $y = z_k$ an. Das hat zur Folge, dass die einzelnen Schritte der Weierstraß-Iteration schneller genau werden. Dabei dient die Annäherung an $y = z_k$, da die Gleichung statt der Nullstelle genauer wird, mehr als Beschleunigung der Methode, während das Annähern an die Nullstelle mithilfe der Gleichung die eigentliche Annäherung bringt.

Annähern an die Nullstelle mithilfe der Gleichung Für jedes $z_k^{(i)}$, welches weit genug von den Definitionslücken entfernt ist, gilt, dass $g(z_k^{(i)})$ näher an z_k als $z_k^{(i)}$ ist.

Für die Herleitung diese Aussage muss zuerst gezeigt werden, dass $g(x)$ eine waag-

rechte Asymptote besitzt. Dafür kann man sich die zuvor hergeleitete Gleichung

$$g(x) = y = \frac{x \cdot \prod_{j=1;j\neq k}^{n}(x - z_j^{(i)}) - \prod_{j=1}^{n}(x - z_j^{(i)})}{\prod_{j=1;j\neq k}^{n}(x - z_j^{(i)})}$$

angucken. Da die Polynome im Zähler normiert und vom gleichen Grad sind, besitzen beide den Term $a_n x^n$ mit $a_n = 1$. Werden diese Polynome nun miteinander subtrahiert, fällt dieser n-te-Term weg, da er bei beiden Polynomen gleich ist. Somit ist der Grad des Differenzpolynoms $n - 1$.

$$g(x) = y = \frac{\text{Polynom vom Grad } n-1}{\text{Polynom vom Grad } n-1}$$

Da der Nenner und Zähler den gleichen Grad haben, hat $g(x)$ eine waagrechte Asymptote.[10]

Wegen dieser waagrechten Asymptote ist $|g(x)| < |x|$ für jedes $x \in \mathbb{R}$ mit ausreichendem Abstand zu den Definitionslücken wahr. Außerdem gilt $|g(x)| > |g(z_k)|$ für jedes $x \in \mathbb{R}$ mit den gleichen Bedingungen wie zuvor und $g(x)$ hat das gleiche Vorzeichen wie z_k mit ausreichend Abstand zu den Definitionslücken. Das kann man an der Lage der waagrechten Asymptote sehen. Diese kann man ausrechnen, indem man die Faktoren vor der höchsten Potenz im Zähler durch den Faktor der höchsten Potenz im Nenner teilt[10]. Da das Nennerpolynom normiert ist, ist die waagrechte Asymptote gleich des Faktors vor der höchsten Potenz im Zähler. Der Betrag dieser ist größer als $|z_k|$ und hat das gleiche Vorzeichen wie z_k. Daraus folgt, dass jedes $z_k^{(i)}$, welches weit genug von den Definitionslücken entfernt ist, in $g(x)$ eingesetzt einen Wert zurückgibt, welcher näher an z_k als $z_k^{(i)}$ ist.

Schluss Das bedeutet für die Weierstraß-Iteration, dass in den meisten Fällen mit jeder Iteration $z_k^{(i+1)}$ näher an z_k als $z_k^{(i)}$ ist. Außerdem wird in den meisten Fällen mit jeder Iteration $g(x)$ genauer. Da für die Nullstelle z_k die Gleichung $g(z_k) = z_k$ gilt und mit einem höheren i der Wert $z_k^{(i)}$ meist genauer wird konvergiert die Weierstraß-Iteration meistens. In dem Abschnitt 3.4.2 wird mehr auf die Fälle eingegangen, bei denen die Weierstraß-Iteration nicht konvergiert.

[10] vgl. studimup.de: Asymptoten berechnen und erkennen, URL: `https://www.studimup.de/abitur/analysis/asymptoten/#waagerecht` (Zuletzt aufgerufen: 04.11.2023)

3.3.2 Wählen der Startpunkte

Alle Nullstellen einer Polynomfunktion befinden sich auf der abgeschlossen Kreisscheibe $B(0, M) := \{x \in \mathbb{C} \big| |x| \leq M\}$ mit $M := 1 + \max\{\frac{|a_j|}{|a_n|} \big| \ j = 0, \ldots, n-1\}$.[11] Um möglichst genaue Startpunkte zu bekommen, kann man diese auf einer Kreislinie innerhalb der Kreisscheibe anordnen.

Dabei gibt es keinen „besten“ Standard für das Wählen des Radius. Jeder Radius ist für manche Funktionen näher und andere weiter von den Nullstellen entfernt. Je nach den Anforderungen kann man einen eigenen Radius für die Kreislinie nehmen. In meiner Implementierung habe ich mich deshalb für $r = |\frac{na_0}{2a_1}| + |\frac{a_{n-1}}{2n}|$ [9] entschieden. Dieser Radius ist relativ ressourcensparend und einfach zu implementieren.

3.4 Probleme und Einschränkungen

3.4.1 Normiertes Polynom

Für die Weierstraß-Iteration muss die gegebene Polynomfunktion $p(x)$ in der Form $\prod_{i=1}^{n}(x - z_i)$ dargestellt werden können. Da $\prod_{i=1}^{n}(x - z_i)$ ausmultipliziert, in jedem Fall ein normiertes Polynom ergibt, muss $p(x)$ für die Weierstraß-Iteration ein normiertes Polynom sein. Ist dies nicht der Fall, wird es unmöglich, $p(x)$ in der Form $\prod_{i=1}^{n}(x - z_i)$ darzustellen. Daraus folgt, dass man die Methode ohne Umwege nicht ausführen kann. Allerdings gibt es eine einfache Lösung für dieses Problem. Wenn man eine Funktion gleich Null setzt, kann man beide Seiten mit einer beliebigen Zahl multiplizieren. Dabei verändert sich das Ergebnis nicht, da $0m = 0 \ \ \forall m \in \mathbb{C}$ ist. Daraus folgt, dass sich die Nullstellen eines Polynoms, wenn dieses mit einer beliebigen Zahl multipliziert wird, nicht verändern.

Somit kann man das Polynom, dessen Nullstellen man finden möchte, durch a_n teilen und bekommt ein normiertes Polynom mit den gleichen Nullstellen. Mit diesem kann man dann die Weierstraß-Iteration durchführen.

[11] vgl. „*Sei* $P(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0$ *ein komplexes Polynom vom Grad* $n \geq 1$. *Dann liegen alle Nullstellen von* $P(z)$ *in der abgeschlossenen Kreisscheibe* $B(0, M) := \{z \in \mathbb{C} \big| |z| \leq M\}$, *wobei* $M := 1 + \max\{\frac{|a_j|}{|a_n|} \big| \ j = 0, \ldots, n-1\}$.“ Humboldt-Universität zu Berlin: Helga Baum: Nullstellen komplexer Polynome (6.3. Erster Satz von Cauchy), URL: `https://didaktik.mathematik.hu-berlin.de/user/fehlingerl/Helga.pdf` (Zuletzt aufgerufen: 30.10.2023)

3.4.2 Keine generelle Konvergenz

Die Weierstraß-Iteration ist nicht generell konvergent. Das bedeutet, dass für bestimmte Startpunkte mancher Polynomfunktionen die Methode bei Iterationen gegen unendlich keinen festen Wert anstrebt. Dabei verfängt sich die Weierstraß-Iteration in periodischen Zyklen[12]. Es ist nicht generell bestimmt, bei welchen Startpunkten und Polynomen die Weierstraß-Iteration nicht konvergiert. Beispielsweise hat das Polynom $x^3+x+177$ eine offene Menge an Startpunkten, bei denen die Weierstraß-Iteration nicht konvergiert.[6] Da dies allerdings, besonders wenn man vernünftige Startpunkte wählt, sehr selten ist, kann es meist vernachlässigt werden.

4 Implementierung

Im Folgenden wird auf die Implementierung der Weierstraß-Iteration in Java eingegangen. Dabei werden die Klassen „Complex“, „Poly“ und „Weierstrass“ implementiert, um komplexe Zahlen, Polynomfunktionen und den Hauptteil der Weierstraß-Iteration darzustellen.

4.1 Komplexe Zahlen (Complex)

In Java sind komplexe Zahlen in keinem Grunddatentyp oder erweiterten Datentyp enthalten. Deshalb habe ich mich dazu entschieden, eine eigene Klasse „Complex“ zu schreiben.

Attribute

Die Attribute „real“ und „imaginary“ repräsentieren den reellen und imaginären Teil der komplexen Zahl.

Konstruktor

Der Konstruktor initialisiert eine komplexe Zahl mit einem gegebenen Real- und Imaginärteil.

[12] d. h. die Weierstraß-Iteration wiederholt sich immer wieder in dem gleichen Abstand.

Methoden

public double abs() berechnet und gibt den Betrag der komplexen Zahl zurück.
public Complex round(int accuracyDecimalPlaces) rundet den Real- und Imaginärteil auf die angegebene Anzahl von Dezimalstellen und gibt eine neue komplexe Zahl zurück.
@Override public String toString() gibt eine lesbare Repräsentation der komplexen Zahl als String zurück.

Statische Methoden

Die statischen Methoden führen die grundlegenden Rechenoperationen Addition, Subtraktion, Multiplikation, Division und Potenzierung mit komplexen Zahlen durch.

4.2 Polynomfunktion (Poly)

Um Polynomfunktionen darzustellen, wird die Klasse „Poly“ benutzt. In dieser werden die Koeffizienten und der Grad des Polynoms gespeichert und für die Weierstraß-Iteration wichtige Methoden implementiert.

Attribute

In dem Array „coefficients“ von Gleitkommazahlen (double) sind die Koeffizienten des Polynoms enthalten. Der Index repräsentiert den Exponenten des zugehörigen Terms. Der Grad des Polynoms wird als Integer in dem Attribut „degree“ gespeichert.

Konstruktor

Der Konstruktor „Poly(double... coefficients)“, erstellt ein Polynom mit den gegebenen Koeffizienten.

Methoden

public void normalise() normiert das Polynom. Hierbei wird jeder Koeffizient durch den führenden Koeffizienten a_n des Polynoms geteilt. Die Nullstellen des Polynoms ändern sich nicht, aber die Darstellung des Polynoms ändert sich.
public Complex solve(Complex val) berechnet den Funktionswert des Polynoms für einen gegebenen komplexen Wert „val“. Dabei wird das einfache Horner-Schema[13] verwendet,

[13] k-achilles.de: Das HORNER-Schema, URL: `https://www.k-achilles.de/algorithmen/HORNER-Schema.pdf` (Zuletzt aufgerufen: 18.11.2023)

um die Auswertung effizient durchzuführen.
@Override public String toString() überschreibt die „toString()“-Methode, um eine lesbare Darstellung des Polynoms auszugeben. Die Methode berücksichtigt verschiedene Fälle einschließlich der Anzeige von Termen mit positiven oder negativen Koeffizienten und der Auslassung von Termen mit Koeffizienten gleich null.

4.3 Weierstraß-Iteration (Weierstrass)

In der Klasse „Weierstrass“ befindet sich der Hauptteil der Weierstraß-Iteration und „Entry-Point“ des Programms. Alle Funktionen in der Klasse sind statisch. Die Funktion „weierstrass(Poly p)“ enthält das Framework der Weierstraß-Iteration, während spezifische Aufgaben in den anderen Funktionen ausgelagert sind.

calc(Poly p, Complex[] roots, int index)

Die „calc-Methode“ berechnet den nächsten Wert für eine Nullstelle des Polynoms unter Verwendung der Weierstraß-Iterationsformel[14]. Dabei werden drei Parameter entgegengenommen: das Polynom „p“, für das die Nullstellen berechnet werden, ein Array von komplexen Zahlen „roots“, das die aktuellen Näherungen der Nullstellen enthält, und der Index der Nullstelle, die in dieser Iteration aktualisiert wird.
Zunächst werden der Zähler („numerator“) und der Nenner („denominator“) des Weierstrass-Korrekturterms berechnet. Dann wird dieser Korrekturterm von der aktuellen Nullstelle subtrahiert, um die nächste Näherung zu erhalten.

startingPoints(Poly p, int accuracyDecimalPlaces)

Die „startingPoints“-Methode generiert anhand der Eigenschaften des Polynoms anfängliche Schätzungen für die Nullstellen. Sie nimmt als Parameter das Polynom „p“ entgegen und gibt ein Array von komplexen Zahlen zurück, das die initialen Schätzungen der Nullstellen enthält.
Die Methode verwendet die zuvor beschriebenen Berechnungen, um den Radius und den Winkel für die Platzierung der Anfangsschätzungen auf einem Kreis in der komplexen Ebene zu bestimmen. Die generierten Startpunkte werden als Ausgangspunkte für die Weierstraß-Iteration verwendet.

[14] vgl. Abschnitt 3.1.2

isAccurateEnough(Complex[] newRoots, Complex[] oldRoots, double accuracy)

Die „isAccurateEnough“-Methode überprüft, ob die Nullstellen ausreichend genau konvergiert sind. Sie vergleicht den Unterschied zwischen den neuen Nullstellen und den alten Nullstellen mit einer vorgegebenen Genauigkeitsgrenze („accuracy“).
Die Methode gibt wahr zurück, wenn alle Nullstellen innerhalb der Genauigkeitsgrenze konvergiert sind, andernfalls gibt sie falsch zurück.

weierstrass(Poly p)

Die „Weierstrass“-Methode ist die Hauptmethode der Weierstraß-Iteration. Sie nimmt ein Polynom entgegen und gibt ein Array von komplexen Zahlen mit Näherungen der Nullstellen des Polynoms zurück. Die Methode kann in sechs Schritte unterteilt werden.

a) Polynom normieren: Zuerst wird das Polynom normiert, um die Weierstraß-Iteration möglich zu machen. Dafür wird die Methode „normalise()“ der Polynomfunktion aufgerufen.

b) Genauigkeit und maximale Iterationen festlegen: Darauffolgend wird die Genauigkeit mit 10^{-10} und maximale Anzahl an Iterationen mit 1000 festgelegt.

c) Initialisierung der Nullstellen: Als nächstes werden mithilfe der zuvor behandelten Funktion „startingPoints(Poly p)“ die Annährungen der Nullstellen in einem Array initialisiert.

d) Durchführen der Weierstraß-Iteration: Jetzt kann eine Schleife gestartet werden, die solange läuft, bis die Konvergenz erreicht ist oder die maximale Anzahl von Iterationen. In dieser wird die Funktion „calc(Poly p, Complex[] roots, int index)“ für alle Nullstellen bei jeder Iteration ausgeführt.

e) Probe: Nach Abschluss der Iteration wird jede gefundene Nullstelle in das Polynom eingesetzt und ausgegeben, um die Genauigkeit zu überprüfen.

f) Rückgabe der Nährungen der Nullstellen: Zuletzt wird das endgültige Array der berechneten Wurzeln zurückgegeben.

4.4 Input/Output

Der Input erfolgt mithilfe der Bibliothek „java.util.Scanner“ in der „main“-Methode. Dabei wird für jeden Term des Polynoms nach dem Koeffizienten gefragt. Es wird bei dem 0-ten Term begonnen und in aufsteigender Reihenfolge weitergefragt. Wenn der gewünschte Grad erreicht ist, kann der Nutzer „done“ eingeben und die Weierstraß-Iteration wird mit dem gegebenen Polynom ausgeführt.

Darauffolgend wird zuerst die Polynomfunktion zur Kontrolle ausgegeben. Während die Weierstraß-Iteration ausgeführt wird, werden alle Näherungen an die Nullstellen für diese Iteration angegeben. Zuletzt wird die Probe aller Nullstellen ausgegeben. Bei dieser sollte, falls die Weierstraß-Iteration funktioniert hat, „0.0“ für jede Nullstelle herauskommen.[15]

5 Fazit

Um die Nullstellen einer Polynomfunktion n-ten Grades mithilfe des Computers zu finden, kann die Weierstraß-Iteration genutzt werden. Dabei werden n Gleichungen für n Nullstellen gebildet. Diese nehmen die vorherigen Approximationen der Nullstellen und geben meist eine genauere Nullstelle zurück. Sobald die Annäherungen die gewünschte Genauigkeit haben, ist die Weierstraß-Iteration beendet. Um zu testen, ob die Methode erfolgreich war, können alle Näherungen in das Polynom eingesetzt werden. In manchen Fällen konvergiert die Weierstraß-Iteration nicht. Aus diesem Grund muss ein Limit an Iteration angegeben sein.

Mit der Implementierung der Weierstraß-Iteration in Java habe ich eine Möglichkeit, Nullstellen von Polynomen mithilfe des Computers zu approximieren, entwickelt. Es gibt allerdings auch andere Verfahren, die andere Stärken und Schwächen haben. In meiner Implementierung habe ich den Fokus auf die Lesbarkeit des Codes statt auf die Optimierung der Performance gelegt. Trotz diesen Einschränkungen können die Nullstellen der meisten Polynome mithilfe meiner Implementierung in relativ wenig Iterationen gefunden werden.
Auf Basis meiner Ergebnisse währe es nun interessant, dieses Verfahren anderen gegenüberzustellen und zu vergleichen. Des Weiteren kann die Weierstraß-Iteration und

[15] vgl. Abschnitt 6.1

meine Implementierung noch bezüglich der Geschwindigkeit optimiert werden.

Im Rahmen meiner Facharbeit habe ich einen tiefen Einblick in die wissenschaftliche mathematische Arbeitsweise erlangt. Durch die Bearbeitung der gewählten Problemstellung kenne ich mich nun besser mit der Numerik aus. Im Nachhinein währe es ein Vorteil gewesen, Python statt Java zu benutzen, da in Python komplexe Zahlen direkt eingebaut sind. Das hätte Zeit gespart, hat jedoch keinen Einfluss auf die Qualität des Ergebnisses. Des Weiteren habe ich nur wenige Informationen zu der Weierstraß-Iteration finden können. Die meisten Bücher über die Numerik behandeln andere Verfahren. Aus diesem Grund musste ich zu einem Teil auf eher unseriösere Quellen zurückgreifen. Diese habe ich durch Vorwissen und den Vergleich mit anderen Quellen überprüft und Webseiten, bei denen Fehler aufgefallen sind, nicht benutzt. Während des Schreibens der Facharbeit ist mir klar geworden, wie komplex dieses Thema wirklich ist. Aus diesem Grund musste ich manche Herleitungen und Erklärungen kürzen. An diesen Stellen bietet es sich an, sich weiter mit dem Thema zu beschäftigen.

6 Anhang

6.1 Input/Output Beispiel

```
1  PS C:\Development\Facharbeit\src> java Weierstrass
2  Please input a coefficient for the 0th Term or 'done':
3  3
4  Please input a coefficient for the 1th Term or 'done':
5  -4
6  Please input a coefficient for the 2th Term or 'done':
7  2
8  Please input a coefficient for the 3th Term or 'done':
9  7
10 Please input a coefficient for the 4th Term or 'done':
11 -3
12 Please input a coefficient for the 5th Term or 'done':
13 done
14
15 polynom: 1.0x^(4) - 2.3333333333333335x^(3) - 0.6666666666666666x
       ^(2) + 1.3333333333333333x - 1.0
```

```
16
17 iteration: 0
18 i = 0: 1.6552841624 + 0.6856411497i
19 i = 1: -0.6856411497 + 1.6552841624i
20 i = 2: -1.6552841624 - 0.6856411497i
21 i = 3: 0.6856411497 - 1.6552841624i
22
23 iteration: 1
24 i = 0: 1.8539472642 + 0.5118992343i
25 i = 1: 0.1470889953 + 1.0987293667i
26 i = 2: -0.8341324115 - 0.3650474203i
27 i = 3: 1.1664294853 - 1.2455811807i
28
29 iteration: 2
30 i = 0: 2.1268684556 - 0.0084935444i
31 i = 1: 0.0818868737 + 0.5809472615i
32 i = 2: -0.845802451 - 0.1407098505i
33 i = 3: 0.9703804549 - 0.4317438666i
34
35 iteration: 3
36 i = 0: 2.5309652599 - 0.0492014993i
37 i = 1: 0.3267937002 + 0.3464210299i
38 i = 2: -0.9060995604 + 0.0202659974i
39 i = 3: 0.3816739337 - 0.317485528i
40
41 iteration: 4
42 i = 0: 2.4517265689 - 0.0019358547i
43 i = 1: 0.436328963 + 0.5486123919i
44 i = 2: -0.9656563518 - 0.002268555i
45 i = 3: 0.4109341533 - 0.5444079821i
46
47 iteration: 5
48 i = 0: 2.4512903802 - 2.09016E-5i
49 i = 1: 0.4208896496 + 0.5020943496i
50 i = 2: -0.9576001405 - 2.65727E-5i
51 i = 3: 0.418753444 - 0.5020468753i
52
53 iteration: 6
54 i = 0: 2.451294556 - 8.1E-9i
55 i = 1: 0.4197084882 + 0.499965846i
```

```
i = 2: −0.9573689688 + 7.97E−8i
i = 3: 0.419699258 − 0.4999659176i

iteration: 7
i = 0: 2.4512945632
i = 1: 0.4197037322 + 0.4999622586i
i = 2: −0.9573686941
i = 3: 0.4197037321 − 0.4999622586i

iteration: 8
i = 0: 2.4512945632
i = 1: 0.4197037321 + 0.4999622586i
i = 2: −0.9573686941
i = 3: 0.4197037321 − 0.4999622586i

——— 1.0x^(4) − 2.3333333333333335x^(3) − 0.6666666666666666x^(2) +
    1.3333333333333333x − 1.0 ———
Probe: f(2.4512945632) = 0.0
Probe: f(0.4197037321 + 0.4999622586i) = 0.0
Probe: f(−0.9573686941) = 0.0
Probe: f(0.4197037321 − 0.4999622586i) = 0.0
```

6.2 Programmcode

6.2.1 Complex.java

```java
/**
 * The Complex class represents complex numbers and provides basic
    operations
 * such as addition, subtraction, multiplication, division, and
    exponentiation.
 */
class Complex {
    private double real; // real part
    private double imaginary; // imaginary part

    /**
     * Constructor for initializing a complex number with a real
        and an imaginary part.
     *
```

```
12      * @param real The real part of the complex number.
13      * @param im The imaginary part of the complex number.
14      */
15     Complex(double real, double imaginary) {
16         this.real = real;
17         this.imaginary = imaginary;
18     }
19
20     /**
21      * Calculates the absolute value of the complex number.
22      *
23      * @return The absolute value of the complex number.
24      */
25     public double abs() {
26         return Math.sqrt(Math.pow(real, 2) + Math.pow(imaginary, 2)
               );
27     }
28
29     /**
30      * Rounds the real and imaginary parts of the complex number to
            the specified number
31      * of decimal places.
32      *
33      * @param accuracyDecimalPlaces The number of decimal places
            for rounding.
34      * @return                      A new complex number with
            rounded real and imaginary parts.
35      */
36     public Complex round(int accuracyDecimalPlaces) {
37         // shove the numbers after the decimal point in front of
               the decimal point and cut of the rest
38         double factor = Math.pow(10, accuracyDecimalPlaces);
39                 double r1 = Math.round(this.real * factor);
40         double i1 = Math.round(this.imaginary * factor);
41
42         // shove the numbers after the decimal point back to where
               they belong
43         double r2 = (double) r1 / factor;
44         double i2 = (double) i1 / factor;
45
```

```
46                  return new Complex(r2, i2);
47      }
48
49      /**
50       * Returns a string representation of the complex number.
51       *
52       * @return The string representation of the complex number.
53       */
54      @Override
55      public String toString() {
56          if (imaginary == 0) return real + "";
57          if (real == 0) return imaginary + "i";
58          if (imaginary <  0) return real + " - " + (-imaginary) + "i
                ";
59          return real + " + " + imaginary + "i";
60      }
61
62      // ----- Static methods for complex number operations -----
63
64      public static Complex plus(Complex a, Complex b) {
65          double re = a.real + b.real;
66          double im = a.imaginary + b.imaginary;
67          return new Complex(re, im);
68      }
69
70      public static Complex minus(Complex a, Complex b) {
71          double re = a.real - b.real;
72          double im = a.imaginary - b.imaginary;
73          return new Complex(re, im);
74      }
75
76      public static Complex multiply(Complex a, Complex b) {
77          double re = a.real * b.real - a.imaginary * b.imaginary;
78          double im = a.real * b.imaginary + a.imaginary * b.real;
79          return new Complex(re, im);
80      }
81
82          public static Complex divide(Complex a, Complex b) throws
                ArithmeticException {
83          double re = (a.real * b.real + a.imaginary * b.imaginary) /
```

```
                (b.real * b.real + b.imaginary * b.imaginary);
84          double im = (a.imaginary * b.real - a.real * b.imaginary) /
                (b.real * b.real + b.imaginary * b.imaginary);
85          return new Complex(re, im);
86      }
87
88      /**
89       * Raises a complex number to a power.
90       *
91       * @param c                     The complex number.
92       * @param exponent              The exponent (a non-negative
             integer).
93       * @return                      The result of the
             exponentiation.
94       * @throws ArithmeticException  if a negative exponent is
             passed.
95       */
96      public static Complex power(Complex c, int exponent) throws
            ArithmeticException {
97          if (exponent < 0) { throw new ArithmeticException("Negative
                exponentiation isn't implemented"); }
98          if (exponent == 0) {
99              return new Complex(1, 0);
100         }
101         Complex product = c;
102         for (int i = 2; i <= exponent; i++) {
103             product = Complex.multiply(product, c);
104         }
105         return product;
106     }
107 }
```

6.2.2 Poly.java

```
1  /**
2   * The "Poly" class represents a polynomial with real coefficients
       and provides methods
3   * for normalization, solving for complex values and generating a
       string representation.
4   */
5  class Poly {
```

```
6           /**
7            * Array containing coefficients of the polynomial, where
                the index represents the power of x.
8            * For example, coefficients[2] represents the coefficient
                of x^2.
9            */
10          public double[] coefficients;
11
12          /**
13           * The degree of the polynomial, which is the highest power
                 of x with a non-zero coefficient.
14           */
15          public int degree;
16
17          /**
18           * Constructs a polynomial with the given coefficients.
19           */
20          Poly(double... coefficients) {
21                  this.coefficients = coefficients;
22                  this.degree = coefficients.length - 1;
23          }
24
25          /**
26           * Normalizes the polynomial by dividing each coefficient
                by the leading coefficient.
27           * The roots of the polynomial won't change.
28           */
29          public void normalise() {
30                  for (int i = 0; i < this.coefficients.length; i++)
                        {
31                          this.coefficients[i] = this.coefficients[i]
                                / this.coefficients[degree];
32                  }
33          }
34
35          /**
36           * Solves the polynomial for a given complex value.
37           *
38           * @param val   The complex value for which the polynomial
                is evaluated.
```

```
39          * @return              The complex result of evaluating
                the polynomial for the given value.
40          */
41         public Complex solve(Complex val) {
42                 Complex result = new Complex(0, 0);
43                 for (int i = 0; i < coefficients.length; i++) {
44                         Complex c = Complex.power(val, i);
45                         Complex monomial = Complex.multiply(new
                               Complex(coefficients[i], 0), c);
46                         result = Complex.plus(result, monomial);
47                 }
48                 return result;
49         }
50
51         /**
52          * Generates a string representation of the polynomial.
53          *
54          * @return A string representing the polynomial in a human-
                readable form.
55          */
56         @Override
57         public String toString() {
58                 if (coefficients.length == 0) {
59                         return "";
60                 }
61
62                 StringBuilder out = new StringBuilder();
63
64                 for (int i = coefficients.length - 1; i >= 0; i--)
                       {
65                         if (coefficients[i] == 0) {
66                                 continue;
67                         }
68
69                         if (out.length() > 0) {
70                                 out.append(coefficients[i] >= 0 ? "
                                       ␣+␣" : "␣-␣");
71                         }
72
73                         double absCoefficient = Math.abs(
```

```
                    coefficients[i]);
74                  if (i > 1) {
75                      out.append(absCoefficient).append("
                            x^(").append(i).append(")");
76                  } else if (i == 1) {
77                      out.append(absCoefficient).append("
                            x");
78                  } else {
79                      out.append(absCoefficient);
80                  }
81              }
82
83              return out.toString();
84          }
85  }
```

6.2.3 Weierstrass.java

```
1   import java.util.ArrayList;
2   import java.util.List;
3   import java.util.Scanner;
4
5   /**
6    * Weierstrass
7    *
8    * This class implements the Weierstrass-Iteration for finding
         roots of a polynomial.
9    */
10  public class Weierstrass {
11      /**
12       * @param p      The polynomial for which roots are being
             calculated.
13       * @param roots  The current approximation of roots.
14       * @param index  The index of the root being updated in the
             iteration.
15       * @return       The more accurate root.
16       */
17      private static Complex calc(Poly p, Complex[] roots, int index)
            {
18          // numerator
19          Complex numerator = p.solve(roots[index]);
```

```
20
21          // denominator
22          List<Complex> factors = new ArrayList<Complex>();
23          for (int i = 0; i < roots.length; i++) {
24              if (i == index) {
25                  continue;
26              }
27              factors.add(Complex.minus(roots[index], roots[i]));
28          }
29          // multiply all the linear factors
30          Complex denominator = new Complex(1, 0);
31          for (int i = 0; i < factors.size(); i++) {
32              denominator = Complex.multiply(denominator, factors.get
                    (i));
33          }
34
35          // correction term
36          Complex correctionTerm = Complex.divide(numerator,
                denominator);
37
38          return Complex.minus(roots[index], correctionTerm);
39      }
40
41      /**
42       * Generate initial guesses (starting points) for the roots
             based on the
43       * properties of the polynomial.
44       *
45       * @param p The polynomial for which roots are being calculated
             .
46       * @return  Array of initial guesses for the roots.
47       */
48      private static Complex[] startingPoints(Poly p, int
            accuracyDecimalPlaces) {
49          // check so that there is no devision by zero
50          double t1 = (p.coefficients[1] == 0) ? 1 : p.coefficients
                [1];
51          double t2 = (p.coefficients[p.degree] == 0) ? 1: p.
                coefficients[p.degree];
52
```

```
53          double radius = Math.abs((p.degree * p.coefficients[0]) /
                (2 * t1)) + Math.abs(p.coefficients[p.degree - 1] / (2 *
                p.degree * t2));
54          double theta = 2 * Math.PI / p.degree;
55          double offset = Math.PI / (2 * p.degree);
56          Complex[] roots = new Complex[p.degree];
57
58          System.out.println("iteration: 0");
59          for (int i = 0; i < p.degree; i++) {
60              roots[i] = new Complex(radius * Math.cos(i * theta +
                    offset), radius * Math.sin(i * theta + offset));
61              System.out.println("i = " + i + ": " + roots[i].round(
                    accuracyDecimalPlaces));
62          }
63          System.out.println();
64          return roots;
65      }
66
67      /**
68       * Check if the roots have converged to a sufficiently accurate
             solution.
69       *
70       * @param newRoots   The updated roots in the current iteration
             .
71       * @param oldRoots   The roots from the previous iteration.
72       * @param accuracy   The desired accuracy for convergence.
73       * @return           True if roots have converged; otherwise,
             false.
74       */
75      private static boolean isAccurateEnough(Complex[] newRoots,
            Complex[] oldRoots, double accuracy) {
76          boolean result = false;
77          for (int i = 0; i < newRoots.length; i++) {
78              if (Complex.minus(newRoots[i], oldRoots[i]).abs() <
                    accuracy) {
79                  result = true;
80              } else {
81                  result = false;
82              }
83          }
```

```
84          return result;
85      }
86
87      /**
88       * Find the roots of the polynomial using the Weierstrass-
             Iteration.
89       *
90       * @param p The polynomial for which roots are being calculated
             .
91       * @return  Array of roots.
92       */
93      public static Complex[] weierstrass(Poly p) {
94                  // for the Weierstrass-Iteration a_n has to be
                       equal to one -> devide every a by a_n (the
                       polynomial won't be the same, but the roots stay
                        the same)
95          p.normalise();
96
97          // The accuracy of the Weierstrass-Iteration
98          double accuracy = Math.pow(10, -10);
99          int accuracyDecimalPlaces = 10;
100         double maxIterations = 1000;
101
102         System.out.println("polynom: " + p.toString());
103         System.out.println();
104
105         // init roots -> starting points
106         Complex[] roots = startingPoints(p, accuracyDecimalPlaces);
107
108         boolean running = true;
109         int ctr = 0;
110         while (running) {
111             // init the array for the new roots
112             Complex[] newRoots = new Complex[roots.length];
113
114             System.out.println("iteration: " + ++ctr);
115
116             // calc for every root
117             for (int i = 0; i < roots.length; i++) {
118                 newRoots[i] = calc(p, roots, i);
```

```
119             }
120
121             for (int i = 0; i < roots.length; i++) {
122                 System.out.println("i = " + i + ": " + newRoots[i].
                        round(accuracyDecimalPlaces));
123             }
124             System.out.println();
125
126             // check if change is less than eqsilon
127             if (isAccurateEnough(newRoots, roots, accuracy) || ctr
                    >= maxIterations) {
128                 running = false;
129             }
130
131             roots = newRoots;
132         }
133
134         // Probe
135         System.out.println("---- " + p + " ----");
136         for (Complex r : roots) {
137             System.out.println("Probe: f(" + r.round(
                        accuracyDecimalPlaces) + ") = " + p.solve(r).round(
                        accuracyDecimalPlaces));
138         }
139
140         return roots;
141     }
142
143     public static void main(String[] args) {
144         List<Double> coefficientsList = new ArrayList<Double>();
145
146         Scanner scanner = new Scanner(System.in);
147
148         boolean running = true;
149         int i = 0;
150         while (running) {
151             System.out.println("Please input a coefficient for the "
                        + i + "th Term or 'done':");
152             String input = scanner.next();
153             if (!input.equals("done")) {
```

```
154                     try {
155                         coefficientsList.add(Double.parseDouble(input))
                                ;
156                     } catch (NumberFormatException e) {
157                         System.err.println("You need to input a valid
                                number!");
158                         i--;
159                     }
160                 } else {
161                     running = false;
162                 }
163                 i++;
164             }
165             System.out.println();
166             scanner.close();
167
168             double[] coefficients = new double[coefficientsList.size()
                    ];
169             for (int j = 0; j < coefficientsList.size(); j++) {
170                 coefficients[j] = coefficientsList.get(j);
171             }
172             Poly polynom = new Poly(coefficients);
173             weierstrass(polynom);
174         }
175     }
```

6.3 Literaturverzeichnis

Humboldt-Universität zu Berlin: Helga Baum: Nullstellen komplexer Polynome (6.3. Erster Satz von Cauchy), URL: `https://didaktik.mathematik.hu-berlin.de/user/fehlingerl/Helga.pdf` (Zuletzt aufgerufen: 30.10.2023)

k-achilles.de: Das HORNER-Schema, URL: `https://www.k-achilles.de/algorithmen/HORNER-Schema.pdf` (Zuletzt aufgerufen: 18.11.2023)

mathe-online.at: Zwischenwertsatz und Bisektionsverfahren, URL: `https://www.mathe-online.at/nml/materialien/innsbruck/bisektion/Bisektion.pdf` (Zuletzt aufgerufen: 15.11.2023)

Oscar Veliz: Durand-Kerner Method: Minute 5:40 (Veröffentlicht am 29.05.2019), URL: `https://www.youtube.com/watch?v=5JcpOj2KtWc` (Zuletzt aufgerufen: 30.10.2023)

Prof. Dr. Bernd Engelmann: Numerische Mathematik, 2.4 Das Newton-Verfahren und seine Abkömmlinge (S.24)

studimup.de: Asymptoten berechnen und erkennen, URL: https://www.studimup.de/abitur/analysis/asymptoten/#waagerecht (Zuletzt aufgerufen: 04.11.2023)

Wikipedia: Aberth method, URL: https://en.wikipedia.org/wiki/Aberth_method (Zuletzt aufgerufen: 18.11.2023)

Wikipedia: Durand-Kerner method, URL: https://en.wikipedia.org/wiki/Durand-Kerner_method (Zuletzt aufgerufen: 29.10.2023)

Wikipedia: Weierstraß-(Durand-Kerner)-Verfahren, URL: https://de.wikipedia.org/wiki/Weierstra-(Durand-Kerner)-Verfahren (Zuletzt aufgerufen: 29.10.2023)

6.4 Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.

Datum, Unterschrift

Hiermit erkläre ich, dass ich damit einverstanden bin, wenn die von mir verfasste Facharbeit der schulinternen Öffentlichkeit zugänglich gemacht wird.

Datum, Unterschrift