

Newton Fraktale

Block 3, Computerpraktikum

Valentino Bergamotto

Momcilo Drljaca

Karoline Schüler

7. Februar 2024

Inhaltsverzeichnis

1 Das Programm	2
1.1 Aufgabenstellung	2
1.2 Installation	2
1.3 Funktionen und Arbeitsteilung	2
2 Eingabe und Beispieldaten	3
3 Newtonverfahren	6
3.1 Punktweise Implementierung mit Matrixmultiplikation	6
3.2 Simultane Implementierung mit Multiplikation der Komplexen Zahlen	8
3.3 Berechnung der Nullstellen aus den gegebenen Berechnungen	8
3.4 Finale Version des Verfahrens	9
3.5 Geschwindigkeit der finalen Version	10
4 Interaktion mit dem Plot	12
4.1 Allgemeines zur Klasse <code>Fractal</code>	12
4.2 Neuberechnung	12
4.3 Die Wahl der Farben	13
4.4 Zoom mit Mausrad und Verschieben des Plots	14
4.5 Zoom mit Rechteck	14
4.6 Interaktion über die Tastatur	15
4.7 Ändern der Pixel Anzahl	16
4.8 Automatisches Zoomen	17
5 Schnellere Berechnung	18
5.1 Motivation	18
5.2 GUI im Browser	18
5.3 Ableiten der Eingabe auf lokalem Python Server	18
5.4 GLSL	19
5.5 Anleitung	19
5.6 Zusammenfassung	20

1 Das Programm

1.1 Aufgabenstellung

Gegeben sei eine Funktion $f : \mathbb{C} \rightarrow \mathbb{C}$. Dann kann für einen gegebenen Punkt $a + bi \in \mathbb{C}$ mithilfe des (lokal konvergierenden) Newtonverfahren eine Nullstelle von f bestimmt werden.

Ordnet man für jeden Punkt in $\mathbb{R}^2 \cong \mathbb{C}$ als Farbe die berechnete Nullstelle und als Intensität die Anzahl an Iterationen bis das Newtonverfahren terminiert zu, so erhält man bei den meisten Funktionen f ein hübsches Fraktal.

Ziel des Projektes war es, das Newtonverfahren zu implementieren und mit den entstandenen Fraktalen ein bisschen herumzuexperimentieren.

1.2 Installation

Benötigte Module:

```
numpy, matplotlib, time, colorsys, warnings
```

Verwendung: Man führt die Datei `main.py` aus, um dort mithilfe der Konsole bei der Programmierung geleitet zu werden (siehe Abb. 1).

Dabei ist es für die Interaktion entscheidend, dass der Plot in einem eigenen Fenster geöffnet wird. Dies kann in den Einstellungen von Spyder entsprechend eingestellt werden. Im Folgenden eine Anleitung dazu:

Gehe zu „Tools“ → „Preferences“ → „IPython console“ → „Graphics“ → „Backend : Inline“, ändere „Inline“ zu „Automatic“, klicke „OK“.

1.3 Funktionen und Arbeitsteilung

Die Arbeitsteilung der folgenden Funktionen ist mit (V) für Valentino bzw. mit (M) für Momcilo und mit (K) für Karoline gekennzeichnet:

- Beispiel Funktionen mit gegebener Ableitung und Bezeichnung. (M)
- Eingabe von Funktion und weiteren benötigten Daten per Konsole (M) (siehe Kapitel 2)
- Implementierung des Newtonverfahrens (V) (siehe Kapitel 3)
- Interaktion mit dem Plot (K) (siehe Kapitel 4)
- Umsetzung des Newtonverfahrens in einer anderen, schnelleren Programmiersprache (M) (siehe Kapitel 5)

Weitere Arbeitsschritte sind

- Planung der Dateistruktur (K)
- Dokumentation (K)

2 Eingabe und Beispieldaten

Aufgrund veränderter Anforderungen und Möglichkeiten des implementierten Newton Verfahrens musste auch die Eingabe der Daten verändert werden.

Zu Beginn des Projektes erschien es sinnvoll, die Ableitungen der Funktionen, von denen die Fraktalstruktur bereits bekannt war (vgl. $f(z) = z^3 - 1$ usw.), zunächst zu harcoden, um die folgenden Module (siehe `fractal.py`) mit verschiedensten Funktionen testen zu können. Im Zuge dessen ist die Datei `data_collection.py` entstanden. Für die spätere Auswertung ergeben sich Lambda Funktionen als geschickteste Option, entsprechend wurden die Beispieldaten als solche implementiert. Im weiteren Verlauf des Projektes sollte auch die Möglichkeit für den Nutzer, Funktionen selbst einzugeben, implementiert werden. Hierfür existiert in SymPy die Möglichkeit, Text als SymPy- Expression zu parsen.

Die Eingabeaufforderung sieht in etwa wie folgt aus:

```
Welcome to our project
```

```
You can assemble the plot information by yourself.  
First you can chose whether you like to look at one  
of the provided examples or enter a function by yourself.  
Enter 'True' if you want to enter a function by yourself  
and enter 'False' if you want to use one of the provided examples.
```

```
> True
```

```
Please input your function that maps z to some complex  
value, i.e. for example z^3-1
```

```
> exp(cos(z))  
exp(cos(z))  
The function parsed!
```

```
Please wait now. Your fractal will be assembled.
```

```
Ready.
```

```
...
```

Abbildung 1: Konsolen-Output

Sämtliche Eingabemöglichkeiten für den Nutzer werden in dem `data_collection.py` Modul behandelt. Hierbei wurde sich dafür entschieden, dass der Nutzer per Konsole-Eingabe darüber entscheidet, ob eines der Beispiele als Funktion verwendet werden soll oder ob er selbst etwas eingeben möchte. Im weiteren Verlauf wird die (bei entsprechender vorheriger Entscheidung) gewählte Funktion als SymPy Ausdruck geparsst, abgeleitet und wieder zu einer Lambda-Funktion umgewandelt. Anschließend werden die Lambdafunktionen f und f' an die compute-Module (`newton.py`) weitergegeben und berechnet.

Zu den relevanten Beispelfunktionen, gehören natürlich solche, deren resultierendes Fraktal bereits bekannt ist (um testen zu können, ob die rechnenden Module korrekt funktionieren), aber auch

solche, die spezielle Fälle abhandeln (zum Beispiel unendlich viele Nullstellen [$f(z) = \sin(z)$]). Zu der Auswahl an Beispielfunktionen gehören die Einträge der folgenden Liste:

```
=====
Please chose the number of data. There are
1) z^3-1
2) z^3-1 with animation
3) 1/z+z^2
4) (z-1)^2*(z+1)*(z-i)
5) z^4-z^2+1
6) z^7-z
7) z^12-1
8) sin(z)
9) e^z-1
10) e^{(-2z)-1} with animation
11) sin(1/z)
12) sin(1/z) with animation
13) e^{(1/z)-1}
14) log(z)
```

Abbildung 2: Konsolen-Output Variante 2

Zwei Beispiele der resultierenden Fraktale, der erwähnten, interessanten Funktionen sind in den Abbildungen 3 und 4 dargestellt.

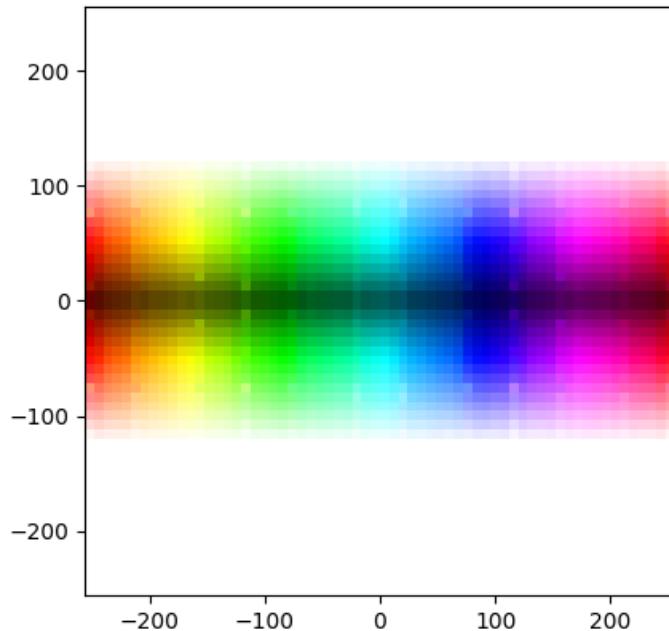


Abbildung 3: Beispiel $f(z) = \sin(z)$

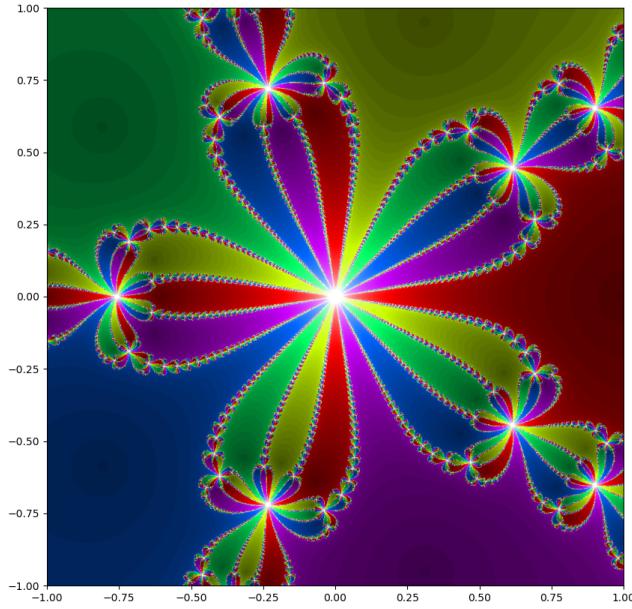


Abbildung 4: Beispiel $f(z) = z^5 - 1$

Alle anderen Eingabeaufforderungen (auch solche zum debugen der anderen Module) finden sich in der `data_collection.py` Datei wieder.

Nach der Ausgabe der Plots, wird in der Konsole noch auf Interaktionsmöglichkeiten hingewiesen. Diese erlauben zum Beispiel Zoomen oder Translation. Alle Interaktionsmöglichkeiten sind in der Konsolenausgabe beschrieben.

You can interact with the plot.
=====

Press 't' to toggle an infotextbox on and of.

You have two zoom options:

First you can zoom by drawing a rectangle.

Second you can zoom with the mouse wheel

and move the plot with drag and drop.

Use this option only when the calculation is fast!

You can switch the zoom options with 'z'.

Press 'b' to zoom back to the last zoom settings.

Press 'o' to zoom (a fix property) out.

Press 'r' to reset the zoom.

Klick (don't draw) on the slider to change the density of the pixels.

Abbildung 5: Konsolen-Output nach Ausgabe der Plots

Weitere Details zur Umsetzung der Interaktionsmöglichkeiten lassen sich in Kapitel 4 nachlesen.

3 Newtonverfahren

Das Newtonverfahren berechnet iterativ von einem gegebenem Punkt eine Nullstelle der gegebenen Funktion mittels Iterationsschritt

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Ändert sich durch den Iterationsschritt der Wert nur noch unterhalb einer gegebenen Toleranz, also gilt

$$|x_{n+1} - x_n| < tol$$

so bricht das Verfahren ab.

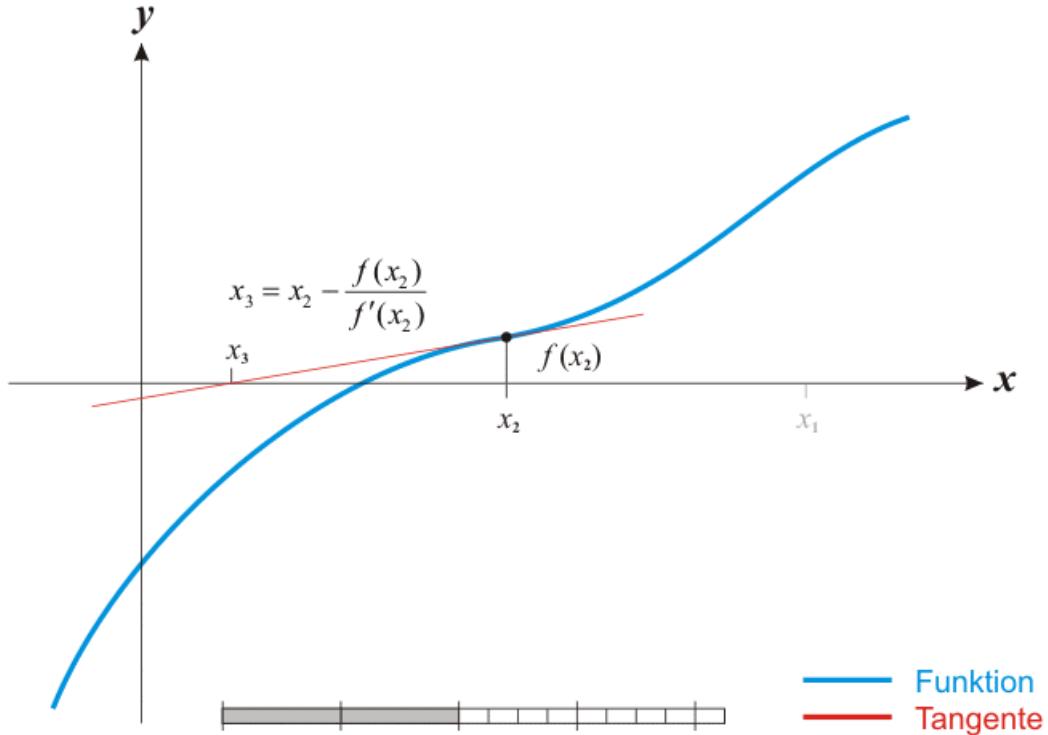


Abbildung 6: Beispiel Newtonverfahren für reelle Funktion [Newtonverfahren]

3.1 Punktweise Implementierung mit Matrixmultiplikation

Zu Beginn des Projekts wollten wir erst einmal ein funktionierendes Programm haben. Dabei haben wir keine Rücksicht auf Performance oder Sinnhaftigkeit der Implementierung genommen. Ein erster Prototyp des Newtonverfahrens war, das mehrdimensionale Verfahren für den \mathbb{R}^2 zu implementieren, um komplexe Zahlen zu simulieren. Hierbei muss man Funktionen wie $f(z) = z^2 + 1$ in Real- und Imaginärteil trennen und den $\frac{f(x_n)}{f'(x_n)}$ Teil durch eine Matrixmultiplikation mit der Inversen der Jacobi-Matrix von f ersetzen:

$$x_{n+1} = x_n - J(f) \cdot x_n$$

Der Einfachheit halber haben wir zu Beginn die Nullstellen noch selbst übergeben, um die ersten Tests schneller durchführen zu können.

```

def newton_approx(f, f_diff, start_point, zeroset, max_iterations, border):
    iterations = 0
    x = start_point
    while iterations < max_iterations:
        jacobi = np.array([[f_diff[0][0](x[0],x[1]),f_diff[0][1](x[0],x[1])],
                           [f_diff[1][0](x[0],x[1]),f_diff[1][1](x[0],x[1])]])
        jacobi = np.linalg.inv(jacobi)
        x = x - np.matmul(jacobi, f(x[0],x[1]))
        for i in zeroset:
            if np.linalg.norm(x, i) < border:
                return i, iterations
        iterations += 1
    return np.inf, iterations

```

Diese Implementierung berechnet für einen einzelnen Startpunkt die Nullstelle, gegen die dieser konvergiert. Die folgende Version berechnet die Nullstellen selbst und nimmt ein Meshgrid von Startpunkten entgegen. Das Verfahren wird aber immer noch punktweise angewendet und durch die Nullstellenberechnung ist es deutlich langsamer als die erste Version.

```

def newton_approx(f, f_diff, start_points, max_iterations, border):
    iterations = 0
    y_dim,x_dim = start_points[0].shape[0], start_points[0].shape[1]
    root_help = []
    roots = []
    point_help = np.zeros([y_dim,x_dim])
    for y in range(y_dim):
        for x in range(x_dim):
            point = np.array([start_points[0][y,x], start_points[1][y,x]])
            while iterations < max_iterations:
                jacobi = np.array([[f_diff[0][0](point[0],point[1]),
                                   f_diff[0][1](point[0],point[1])],
                                   [f_diff[1][0](point[0],point[1]),
                                   f_diff[1][1](point[0],point[1])]])
                jacobi = np.linalg.inv(jacobi)
                point = point - np.matmul(jacobi, f(point[0],point[1]))
                for i in root_help:
                    if np.linalg.norm(point, sum(i)/len(i)) < border:
                        i.append(point)
                    else:
                        root_help.append([point])
                iterations += 1
            point_help[y,x] = point
    for i in root_help:
        if len(i) > 9:
            roots.append(sum(i)/len(i))
    roots.append(np.inf)
    for y in range(y_dim):

```

```

    for x in range(x_dim):
        for i in range(len(roots)):
            if np.linalg.norm(point_help[x,y], roots[i]) < border:
                point_help[x,y] = i
                break
            else:
                point_help[x,y] = len(roots)
    return point_help, roots

```

3.2 Simultane Implementierung mit Multiplikation der Komplexen Zahlen

Ab hier war es wichtig, das Verfahren so effizient wie möglich zu implementieren. Der erste Schritt dazu war, von der Implementierung für den \mathbb{R}^2 zu einer Implementierung mit der python eigenen Klasse für Komplexe Zahlen zu wechseln.

```
point = point - f(point)/f_diff(point)
```

Mit der `vectorize()` Funktion von numpy kann man diesen Schritt nun nicht nur Punktweise, sondern für die gesamte Startmatrix gleichzeitig berechnen.

```
calculate_step = np.vectorize(lambda point: f(points)/f_diff(points))
points = points - calculate_step(points)
```

Dabei verliert man die Information über die Iterationsanzahl. Dies kann man beheben, indem man die vektorisierte Funktion nur für noch nicht terminierte Punkte ausführt. Das geschieht durch eine Hilfsmatrix von Bools, die die vektorisierte Funktion nur für die Punkte aufruft, an deren Index der Wert `true` ist. Mit einer weiteren Hilfsmatrix für die Iterationszahl, erhält man so die Information dafür.

```

points = grid[0]+1J*grid[1]
dim = points.shape[1]
iterations = np.zeros((dim, dim))
undone = np.ones((dim, dim)).astype(bool)
for i in range(max_iterations):
    points_old = points.copy()
    undone_old = undone.copy()
    points[undone] = points_old[undone]-calculate_step(points_old[undone])
    undone = np.logical_and(value!=np.Inf, np.abs(value-value_old)>=tolerance)
    iterations[np.logical_and(np.logical_not(undone), undone_old)] = i+1
    if not undone.any(): break
value[undone] = np.Inf
value[np.isnan(value)] = np.Inf

```

3.3 Berechnung der Nullstellen aus den gegebenen Berechnungen

Nun muss man noch die Nullstellen berechnen und den einzelnen Punkten ihre jeweiligen Nullstellen zuordnen. Dafür erstellen wir eine Datenmenge aller Punkte, die konvergieren und gruppieren diese, wenn ihr Abstand klein genug ist miteinander. Die Nullstellen sind dann der jeweilige Durchschnitt der unterschiedlichen Gruppen.

```

data = points[points!=np.Inf]
roots_set = []
while len(data)>0:
    mask = np.isclose(data, data[0], atol=10*tolerance)
    roots_set.append(data[mask])
    data = data[np.logical_not(mask)]
roots = np.sort_complex([np.average(r_set) for r_set in roots_set])

```

3.4 Finale Version des Verfahrens

Durch die Entwicklung unseres Verfahrens, Zusammenführen der einzelnen Schritte und Dokumentation erhalten wir unsere finale Version des Newtonverfahrens:

```

def newton_approximation(func, diff, grid, max_iterations, tolerance):
    """
    Newton approximation of roots for complex functions.
    Calculating the resluts simultanuously by using matrixmultiplication.

    Parameters
    -----
    func : function from C to C
        The function which generates the fractal.
    diff : function in one variable
        The derivative of func.
    grid: array-like of shape (2,dim,dim)
        Grid with all start points for the algorithm.
    max-iterations: int
        Number of iterations before the algorithm terminates.
        If it is exceeded, the function expects divergence.
    tolerance: float
        Required distance for the calculated root to the
        actual root in order to terminate.

    Returns
    -----
    roots : np.ndarray of float with shape (n+1,)
        Array of the roots of func.
        Inculding [np.Inf,np.Inf] as last entry for divergence.
    indexes : np.ndarray of int with shape (dim,dim)
        At each entry the index in roots of the root which will be
        approximated if the algorithem starts at the respective start
        point in grid.
    iterations : np.ndarray of int with shape (dim,dim)
        At each entry the number of iterations until the recursion terminates.
    """

    # Init iteration variables
    calculate_step = np.vectorize(lambda point: catch(
        lambda x:func(point)/x, diff(point), handle=np.Inf))

```

```

value = grid[0]+1J*grid[1]
dim = value.shape[1]
iterations = np.zeros((dim,dim))
undone = np.ones((dim,dim)).astype(bool)

# Iteration step for those, where done is False.
for i in range(max_iterations):
    value_old = value.copy()
    undone_old = undone.copy()
    value[undone] = value_old[undone]-calculate_step(value_old[undone])
    undone = np.logical_and(value!=np.Inf, np.abs(value-value_old)>=tolerance)
    iterations[np.logical_and(np.logical_not(undone),undone_old)] = i+1
    if not undone.any(): break
value[undone] = np.Inf
value[np.isnan(value)] = np.Inf

# Calculate the set of roots
data = value[value!=np.Inf]
roots_set = []
while len(data)>0:
    mask = np.isclose(data, data[0], atol=10*tolerance)
    roots_set.append(data[mask])
    data = data[np.logical_not(mask)]
roots = np.sort_complex([np.average(r_set) for r_set in roots_set])

# make last calculations
indexes = np.where(value==np.Inf,len(roots),0)
value_temp = np.kron(value[value!=np.Inf],np.ones_like(roots))
value_temp = value_temp.reshape(int(len(value_temp)/len(roots)),len(roots))
indexes[value!=np.Inf] = np.argmin(abs(value_temp-roots),axis=1).flatten()
roots = np.append(roots, np.Inf)

return roots, indexes, iterations

```

Die catch() Funktion ist hierbei nur dafür da, einem jeglichem Error (wie z.B zero division error) zu entgehen. Der Nachteil dabei ist, dass man auch eventuell relevante Fehlermeldungen unterdrückt.

3.5 Geschwindigkeit der finalen Version

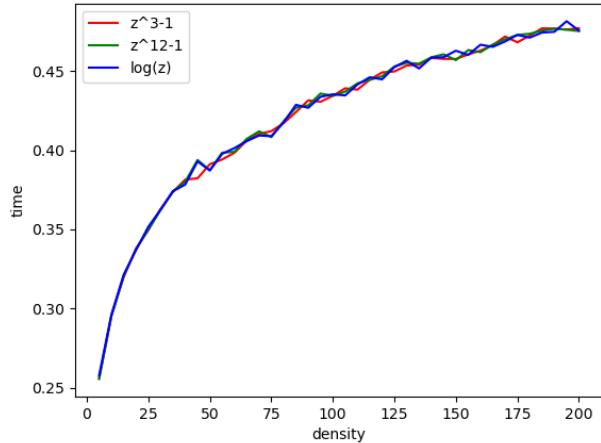
Nun ist es natürlich interessant zu wissen, wie verschiedene Eingaben die Geschwindigkeit des Verfahrens beeinflussen. Dazu haben wir für verschiedene Variablen repräsentativ jeweils den Durchschnitt von 30 Durchläufen genommen.

Die Daten wurden mit folgenden Systemspezifikationen berechnet:

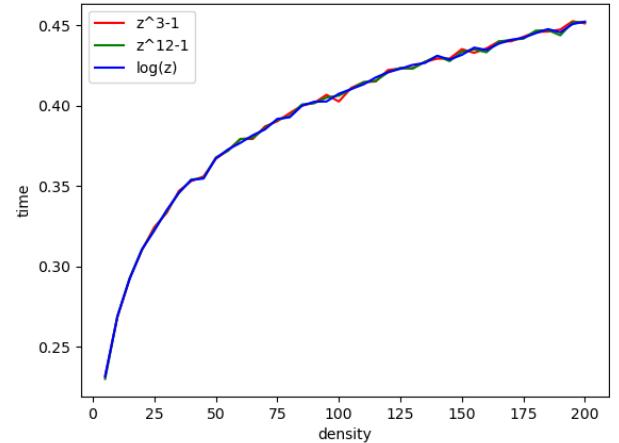
CPU: 12th Gen Intel(R) Core(TM) i7-12700K

GPU: NVIDIA GeForce TRX 4070Ti

RAM: 4800 MHz 2x16 GB



(a) Toleranz von 10^{-7}



(b) Toleranz von 10^{-4}

Hierbei kann man erkennen, dass das Verfahren für verschiedene Funktionen circa gleich schnell konvergiert, auch wenn $z^{12}-1$ vier mal so viele Nullstellen wie z^3-1 hat. Da das Verfahren quadratisch konvergiert, ist auch der geringe Zeitunterschied für unterschiedliche Abbruchtoleranzen nicht verwunderlich. Hier erkennt man den größten Unterschied bei kleiner density der Startpunktmatrix. Dieser nähert sich aber mit zunehmender density der null an. Ein auf den ersten Gedanken unerwartetes Ergebnis ist, dass die Laufzeit mit der density nicht quadratisch ansteigt, obwohl die Menge der Startpunkte mit der density ansteigt. Dies ist der Vektorisierung des Verfahrens und damit Parallelisierung der berechnung zu verdanken. Die älteren Versionen, die punktweise Berechnen. sind für eine höhere density deswegen auch praktisch nicht anwendbar.

4 Interaktion mit dem Plot

Abhängig davon, wie schnell die Berechnungen sind, hat der Nutzer verschiedene Möglichkeiten mit dem Plot zu interagieren.

4.1 Allgemeines zur Klasse Fractal

Da man die Daten - wie die Funktion und ihre Ableitung, das Plot-Fenster oder die Anzahl an Pixeln - für den Plot beliebig wählen können soll, diese für neue Berechnungen mit teils neuen Einstellungen noch zugreifbar sein sollen und der Plot für Interaktivität eigene Methoden braucht, wollten wir eine Klasse dafür erstellen.

Uns war es wichtig, das Bild gut untersuchen zu können, um die fraktalen Strukturen zu erkunden. Daher haben wir einige Methoden erschaffen, die es dem Nutzer ermöglichen in den Plot reinzu-zoomen.

Wir beschreiben hier im Bericht nur die einzelnen Möglichkeiten und Ideen dahinter sowie die relevanten Funktionen und Module die verwendet werden. Für die konkrete Umsetzung kann gerne die - recht ausführliche - Dokumentation der Klasse in Python selbst gelesen werden.

Die Klasse selbst wird mit einer Funktion $f : \mathbb{C} \rightarrow \mathbb{C}$, die vektorisierbar sein muss initialisiert. Also beispielsweise eine `lambda`-Funktion oder eine `numpy` eigene Funktion wie `numpy.sin`.

Gerne kann zudem die Ableitung der Funktion, mit Keyword `diff` - ebenfalls als vektorisierbare Funktion von \mathbb{C} nach \mathbb{C} - und eine Bezeichnung für die Funktion, mit Keyword `label`, eingegeben werden. Sind diese nicht gegeben, so wird versucht die Ableitung der gegebenen Funktion mit Hilfe von `sympy` zu berechnen (siehe Kapitel 2), was zum Absturz des Programms führen kann. Die Bezeichnung wird als String der Funktion gesetzt, was meistens jedoch sehr nichtssagende Bezeichnungen wie „`<function <lambda> at 0x000001A301C70E50>`“ ergibt.

Ebenso kann mit `pointer` ein `numpy` 2D-array eingegeben werden (siehe Kapitel 4.8). Ansonsten wird der `pointer` als `None` gesetzt.

Weitere mögliche Eingaben sind `density`, `max_iterations` und `tolerance`. Ersteres wird als die (anfängliche) Pixelzahl gesetzt und ist standardmäßig 64. Das haben wir als guten Kompromiss zwischen guter Darstellung und - auch für Interaktion geeignete - schneller Berechnungszeit. Zweiteres wird als die maximale Anzahl an Iterationen im Newton-Verfahren gesetzt und ist standardmäßig bei 128. Letzteres gibt die Toleranz an, die in `newton_approximation` als `tolerance` verwendet wird (siehe Kapitel 3.4).

Noch eine allgemeine Bemerkung zu der Klasse ist, dass wir uns dagegen entschieden haben der Klasse ein Attribut zu geben, dass die aktuellen Grenzen des Plots beinhaltet, und das dann gegebenfalls ersetzt wird. Stattdessen hat die Klasse das Attribut `lims`, dass alle bisherigen Grenzen, seit dem letzten Reset, beinhaltet. Die Berechnung des Plots erfolgt immer anhand der letzten Grenzen und eine Aktualisierung der Grenzen wird durch die Methode `set_lims` realisiert, die die neuen Grenzen `lims` hinten anfügt. Die Arbeit mit diesem Array an Grenzen hat den Vorteil, dass es dem Nutzer ermöglicht wird, zu vorherigen Zoom-Einstellungen zurückzukehren.

4.2 Neuberechnung

Wurden die Einstellungen des Plots, wie z.B. die Grenzen, die Pixelzahl oder der Toggler für die Infobox geändert, muss neu geplottet werden. Dies macht die Methode `update`. Sie löscht den alten

Plot, plotted entsprechend der Einstellungen die neuen Sachen und reinitialisiert den Regler (siehe Kapitel 4.7).

Dabei kann es sein, dass der Plot neu berechnet werden muss, was der Methode mittels der Boole `recalculate` mitgeteilt wird. Dann wird die Methode `color_newton` aufgerufen und die Berechnungszeit in der Methode als `calculation_time` gespeichert.

Die Methode `color_newton` erstellt, abhängig von den Grenzen und der Pixelzahl ein Gitter, wo jeder Punkt einen Pixel beschreibt. Dieses Gitter wird dann der Funktion `newton_approximation` von `newton` übergeben. Mit den Ergebnissen wird mithilfe von `set_roots` die Liste der Nullstellen erneuert und damit die anderen Ergebnisse von `newton_approximation` in den neuen Plot übersetzt.

4.3 Die Wahl der Farben

Das erste wichtige kreative Element bei diesem Projekt war die Wahl der Farben für die jeweiligen Nullstellen.

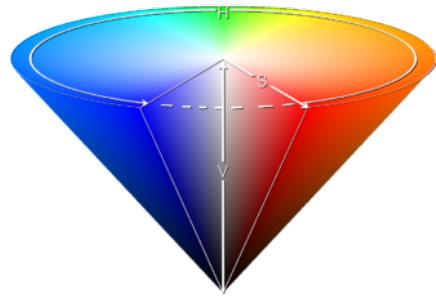
Da das Wählen der Farbe unabhängig von der Anzahl der Nullstellen funktionieren soll, und auch die Anzahl der Iterationen einen beliebigen positiven Wert - kleiner als eine Schranke - annehmen kann, ist es sinnvoll kontinuierliche Werte wählen zu können. Zudem haben wir an jedem Punkt zwei Informationen, weshalb es sinnvoll ist, einen mindestens zweidimensionalen Farbraum zu wählen.

Der HSV-Farbraum bietet diese Möglichkeiten. Die Umrechnung von HSV (resp. HSL) in RGB geht einfach mit der Funktion `hsv_to_rgb` (resp. `hls_to_rgb`) from `colorsys`.

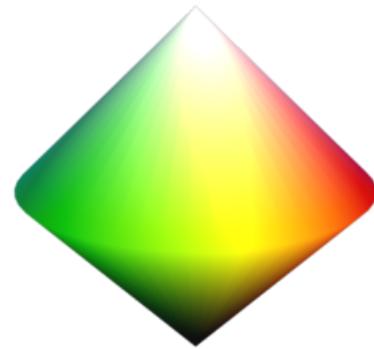
Damit kann dann der Liste der berechneten Nullstellen ein Array aus äquidistant verteilten Punkten zwischen 0 und (exklusive) 1 zugeordnet werden. Diese Zahlen entsprechen dem hue- (also Farb-)Wert. Die Intensität entspricht - mit einem Offset, sodass auch bei schneller Konvergenz die bestimmte Nullstelle noch zu erkennen ist - genau der Intensität.

Ein Problem des HSV-Farbraums für unsere Bedürfnisse ist, dass für die maximale Intensität die Farbe stark ist und es nicht weiß wird (siehe Abbildung 8a), was jedoch besser zur Intuition passen würde, dass da das Newton-Verfahren fast nicht konvergiert. Daher wählten wir den HSL-Farbraum (siehe Abbildung 8b), statt dem HSV-Farbraum, der sich ansonsten nicht vom HSV-Farbraum unterscheidet.

Leider sind die Farben unter Veränderung der Einstellungen recht instabil. Da bei neuer Berechnung des Plots auch die Liste der Nullstellen neu berechnet wird und dabei neue Nullstellen dazu kommen oder welche wegfallen oder die Nullstellen anders sortiert sind ändern sich damit auch



(a) HSV-Kegel [HSV-Farbraum]



(b) HSL-Doppelkegel [HSV-Farbraum]

Abbildung 8: Veranschaulichung der beiden Farträume

jedes Mal die Farbwerte.

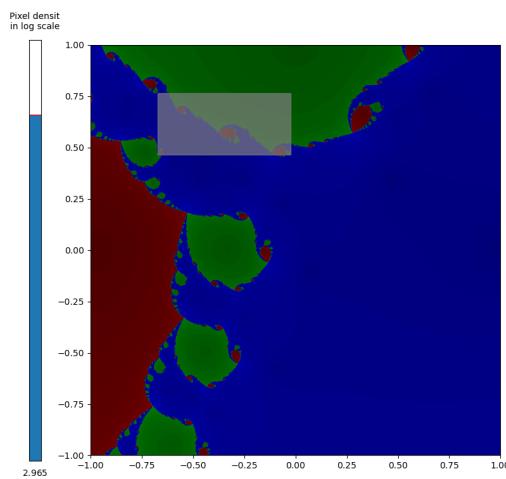
Diese Instabilität wird etwas durch das Anwenden der Funktion `numpy.sort_complex` in `newton_approximation` verringert werden. Damit haben (vor allem reelle) Nullstellen immer zu mindestens einen ähnlichen Farbwert. Etwas, was trotzdem leicht wackelt, ist das Wechseln der Farben von komplex konjugierten Nullstellen. Solche haben die meisten Eingabefunktionen, da sie reelle Koeffizienten (in der Taylorreihe) haben. Bei solchen komplex konjugierten Nullstellen unterscheidet sich die Reihenfolge in der Liste oft nur aufgrund von kleinen Rechenunterschieden. Die Nullstellen sind dann zwar immer noch nebeneinander in der Liste, aber bei Funktionen mit wenigen Nullstellen, fällt es trotzdem auf.

4.4 Zoom mit Mausrad und Verschieben des Plots

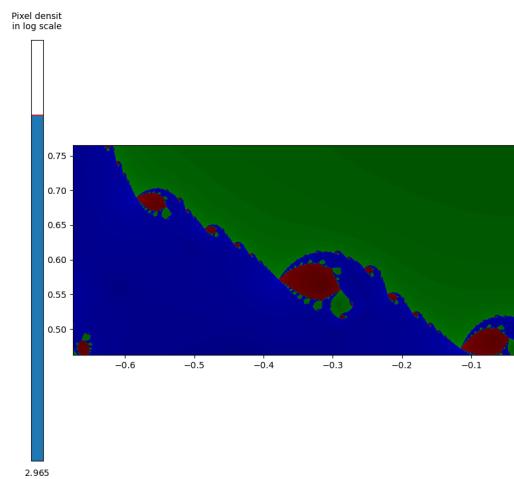
Wir sollten eine Möglichkeit basteln, in den Plot reinzuzoomen. Für uns ist das intuitiv die Möglichkeit, mit dem Mausrad zu zoomen. Daher haben wir mithilfe des `scroll_event` von `matplotlib.pyplot.connect` diese Methode implementiert. Dabei wird, falls der Mauszeiger innerhalb der Plot-Grenzen ist, zu dem Mauszeiger gezoomt. Ansonsten wird in die Mitte des Plots gezoomt.

Ebenso ist es für uns intuitiv den Plot verschieben zu können, um andere Bereiche anzuschauen. Dies haben wir mithilfe des `button_press_event`, ebenfalls von `matplotlib.pyplot.connect`, und weiteren Event-Interaktionen für drag-and-drop, desselben Moduls, realisiert. Dabei wird, falls in den Plot geklickt wird, erstens eine Funktion definiert, die bestimmt, wie sich die Grenzen des Plots, abhängig von der Bewegung des Mauszeigers (\rightarrow `motion_notify_event`), verändern. Zweitens wird eine Funktion definiert, die die Veränderung der Grenzen abbriicht, sobald die Maustaste losgelassen wird (\rightarrow `button_release_event`).

4.5 Zoom mit Rechteck



(a) graues Rechteck über dem neuen Bereich



(b) Der neue Bereich

Abbildung 9: Zoomoption mit Ziehen eines Rechtecks

Da jene Möglichkeiten der Interaktion für mittel schnelle Berechnungen stocken und für langsame

Berechnungen (mit mehr Pixeln) nicht sinnvoll sind, haben wir auch eine Methode implementiert, mit der man auch bei längeren Rechenzeiten des Newton-Verfahrens in den Plot rein zoomen kann. Die Benutzung ist nicht viel komplizierter. Man klickt mit der linken Maustaste irgendwo in den Plot für eine Ecke, zieht den Mauszeiger zur gewünschten gegenüberliegenden Ecke und lässt dort die Maustaste los. Während man den Mauszeiger zieht, wird ein graues Rechteck über den Plot - ohne diesen neu zu berechnen - gezeichnet. Die programmiertechnische Umsetzung ist dabei sehr verwandt mit des drag-and-drop aus dem vorherigen Kapitel.

4.6 Interaktion über die Tastatur

Da es natürlich eine Möglichkeit geben soll, zwischen den unterschiedlichen Zoomoptionen zu wechseln und man auch beim langsamen Berechnen irgendwie rauszoomen will, haben wir Methoden implementiert, die auf das Drücken bestimmter Tastatur-Tasten reagieren. Dafür verwendeten wir wieder `matplotlib.pyplot.connect` und dieses Mal das `key_press_event`. In der Methode `switch_zoom` selbst, wird dann zwischen den unterschiedlichen Tasten unterschieden und die entsprechende Veränderung vorgenommen.

Wird „z“ gedrückt, so wird zwischen den unterschiedlichen Zoomoptionen gewechselt. Dabei ist die Standardeinstellung am Anfang der Zoom mit Rechteck, damit auch bei langsam Berechnungen der Zoom direkt funktioniert.

Ist die gedrückte Taste „b“, so wird auf den vorherigen Stand gezoomt. Beim erneuten Drücken von „b“ wird weiter zurück gezoomt (siehe Beschreibung im letzten Teil von Kapitel 4.1).

Bei „o“ wird mit den festen Faktor 2 rausgezoomt, was vor allem bei Funktionen wie $\sin(z)$ oder $\exp(z) - 1$ praktisch ist, da dort in den Standardgrenzen nur die erste Nullstelle 0 zu sehen ist (siehe Abbildung 10).

Schließlich wird, falls „r“ gedrückt wird, der Zoom zu den Anfangsgrenzen ($x_{min} = -1 = y_{min}$ und $x_{max} = 1 = y_{max}$) zurückgesetzt.

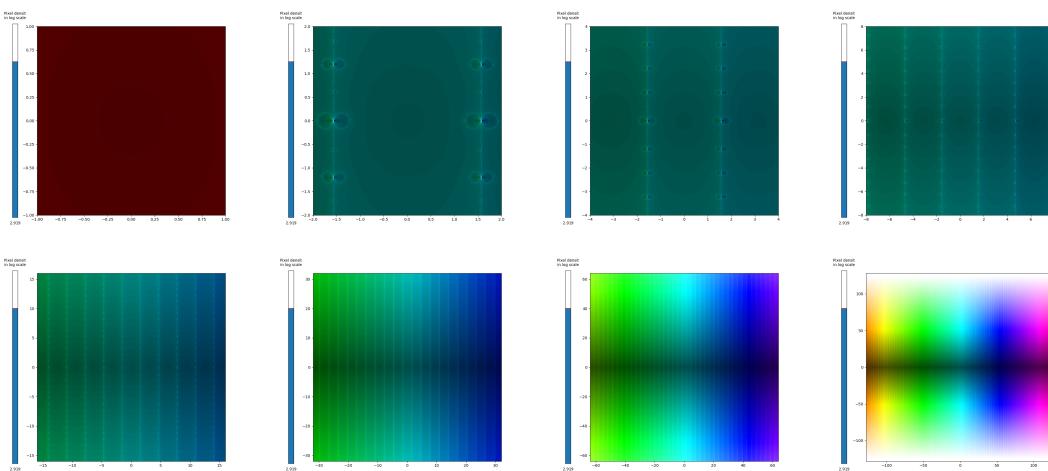


Abbildung 10: Rauszoomen bei $\sin(z)$. Zwischen jedem Bild wurde einmal „o“ gedrückt.

In derselben Methode - es muss Dieselbe sein, da es nur eine Methode für das `key_press_event` geben kann - haben wir zudem einen Toggler auf „t“ für eine Infobox (Siehe Abbildung 11) hinzugefügt. Diese Infobox liefert Daten über den Plot: die Funktion, die Anzahl der Pixel, die benötigte Rechenzeit, die Anzahl der (berechneten) Nullstellen und die Nullstellen selbst mit Farbwert. Die

Informationen dafür werden in der Methode `get_info_str` der Klasse gesammelt und über den bestehenden Plot ausgegeben, ohne dass dieser neu berechnet werden muss. Wurden viele unterschiedliche Nullstellen berechnet, so werden nur die ersten 40 ausgegeben.

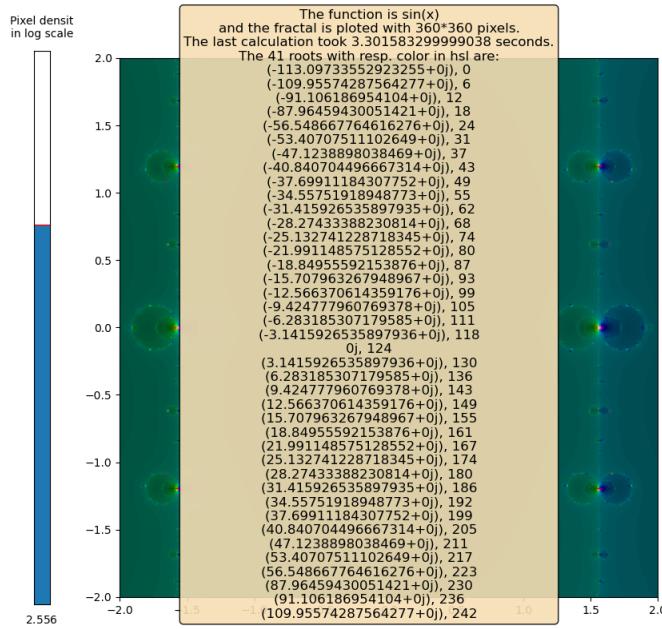


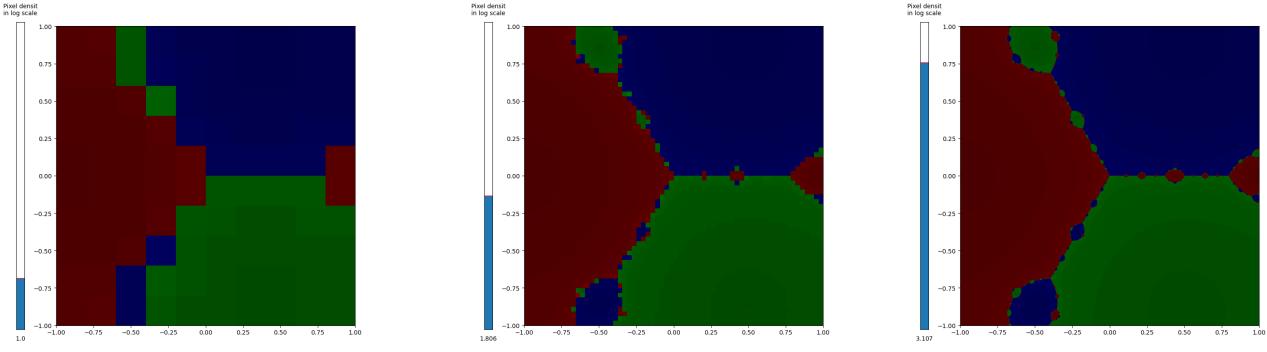
Abbildung 11: Infobox von $\sin(z)$

4.7 Ändern der Pixel Anzahl

Nun haben wir viele Möglichkeiten die Grenzen des Plots zu ändern, die angenehmer sind, wenn die Pixelzahl klein ist, da dann die Berechnung schneller ist und man daher nicht so lange warten muss. Jetzt hat man seine Wunsch Grenzen erreicht aber sieht nur grobe Pixel. Die Lösung: Ein Regler, mit dem man die Pixelzahl ändern kann!

Realisiert wird dieser Regler durch das Attribut `slider` der Klasse `Fractal`, dass in der Methode `update` (siehe Kapitel 4.2) immer wieder reinitialisiert wird und als Update-Funktion die Methode `slider_update` der Klasse hat.

Zwei Sachen sind bei der Benutzung des Reglers zu beachten. Erstens sollte auf den Regler nur geklickt und nicht daran gezogen werden, auch wenn das bei anderen Reglern üblich ist. Zieht man am Regler, so wird kontinuierlich der Plot neuberechnet, was nicht funktioniert, da die Rechenzeit teilweise sehr lange wird. Zweitens ist der Regler in logarithmischer Skala. Das ist erstmal praktisch, da damit sowohl Pixelzahlen von 10 und 100 aber auch 1000 gut unterschieden werden können. Jedoch wird für größere Pixelzahlen die Rechenzeit schnell größer (siehe Kapitel 3.5). Das noch mit der logarithmischen Skala verbunden bewirkt, dass im oberen Teil des Reglers ein kleiner Höhenunterschied einen großen Unterschied in der Rechenzeit bewirken kann.



(a) 10 Pixel

(b) 64 Pixel

(c) 1280 Pixel

Abbildung 12: $1/z + z^2$ mit unterschiedlichen Pixel-Dichten

4.8 Automatisches Zoomen

Als Letztes hatten wir noch die Idee, dass es doch recht nervig sein kann, die ganze Zeit reinzuzoomen. Es wäre doch viel praktischer, wenn das Programm das selbst tun würde. Für so einen automatischen Zoom muss man dem Programm natürlich erst einmal sagen, wohin es denn zoomen soll. Standardmäßig in die Mitte (oder zu einem anderen festen Punkt) zu zoomen ist dabei recht schnell langweilig, da der Plot leicht eintönig wird: Man kommt leicht in den Konvergenzbereich einer Nullstelle und sieht damit nur noch eine Farbe. Oder bei Funktionen der Form $z^n - 1$ für ein $n \in \mathbb{N}$) hat man nur eine fraktale Struktur, nämlich in die Mitte zeigende Streifen. Daher sollte der Punkt zu dem gezoomt wird mit der Funktion selbst der Klasse übergeben werden. Zwar sind die anderen Zoomoptionen unsinnig während dem automatischen Zoom, aber wir wollen sie noch in irgendeiner Form behalten. Also es soll auch möglich sein, doch keinen automatischen Zoom zu haben.

Die Lösung ist das Attribut `pointer` der Klasse `Fractal`. Dieses ist ein Punkt innerhalb von $[-1, 1] \times [-1, 1] \subseteq \mathbb{R}^2$ oder `None`. Der Grund für die Wahl von $\text{pointer} \in [-1, 1] \times [-1, 1]$ ist, dass für die Interaktion und den Plot selbst, die Ebene als Ausschnitt von \mathbb{R}^2 betrachtet wird, nur für die `newton_approximation` selbst wird in \mathbb{C} gerechnet. Zudem es ist ein komisches Zoom-Gefühl wenn zu einem Punkt außerhalb des Sichtbereichs gezoomt wird. Für die Wahl von `pointer` muss dann im konkreten Fall ein bisschen mit den Werten rumgespielt werden, um einen Punkt zu finden, an den sich das Reinzoomen lohnt.

Ist nun `pointer==None`, z.B. weil bei der Initialisierung des Objektes kein `pointer` eingegeben wurde, so haben wir das alte System, ohne automatischen Zoom.

Ist `pointer` gegeben, so wird mithilfe der Methode `kino` automatisch gezoomt. Dies passiert, indem die Zeitdifferenz zwischen Initialisierung des Fractals und der aktuellen Zeit als Zoomfaktor genommen wird und - wie beim Zoomen mit Mausrad - zum `pointer` gezoomt wird. Da in gleichen Zeitabständen um den gleichen Faktor gezoomt werden soll, wird die Zeitdifferenz noch - mit einem kleinen Vorfaktor - in den Exponenten genommen.

Da der automatische Zoom nicht mehr sinnvoll ist, sobald das Plot-Fenster geschlossen ist und dann nur unnötig Rechenkapazität frisst, wird der Zoom beendet, sobald das Fenster geschlossen wurde. Dies wird durch die Methode `isVisible` realisiert, welche `False` zurückgibt, wenn das Fenster geschlossen wurde - ansonsten `True` - und nur wenn der Wert `True` ist, wird weiter gezoomt.

5 Schnellere Berechnung

Im letzten Teil wird auf eine kreative Idee eingegangen, welche vorsieht, die ursprüngliche Aufgabe effizient umzusetzen. Da der Arbeitsaufwand sich als sehr hoch herausgestellt hat, wird hier nur auf die erzielten Erfolge eingegangen, welche allerdings bereits vielversprechende Resultate sind und im Wesentlichen die geringere Rechenzeit demonstrieren.

5.1 Motivation

Dieser kreative Teil der Aufgabe sollte im besten Fall nach Beendigung des Projektes veröffentlicht werden, um möglichst vielen Leuten interaktiven Zugang zu dem Thema zu gewähren. Darum wurde das gesamte Projekt in etwas abgeschwächter Form nochmal als Web-App geschrieben. Diese Browser-Variante des Projektes ist im Wesentlichen unabhängig von der Hauptversion (in Python) und soll nur zeigen, dass die Verwendung offener Standards (siehe Abschnitt 5.4) einen Vorteil bei der Berechnung von parallelisierbaren Prozessen bieten kann. Ein solcher offener Standard ist die Shader-Sprache GLSL, welche zum 3. März 2011 auch in vielen Browsern unter dem Namen WEB GL adaptiert wurde. Seitdem ist es möglich, Berechnungen mit der Grafikeinheit durchzuführen und so zum Beispiel bei den Newton Fraktalen, pro Pixel einen Prozess gleichzeitig laufen zu lassen, der diesen entsprechend einfärbt.

5.2 GUI im Browser

Die Anforderungen an diese Version der Aufgabe sind geringer, da im Wesentlichen nur ein Laufzeitunterschied beobachtet werden soll. Entsprechend ist hier nur die Translation des Bildes per Drag and Drop mit der Maus umgesetzt worden.

5.3 Ableiten der Eingabe auf lokalem Python Server

Achtung, für diesen Teil muss die Bibliothek `flask` zur Verfügung stehen!

Da nicht jede Programmiersprache über Bibliotheken wie SymPy verfügt, muss entweder ein Computer Algebrasystem, welches die gewünschten Aufgaben erfüllen kann, selbst geschrieben werden oder es muss mit Modulen (z.B. in Python geschrieben) kommuniziert werden, die diesen Aufgabenteil erfüllen können. Darum wurde in diesem Fall auf einen lokalen Python Server zurückgegriffen, welcher mit SymPy die erforderlichen Imaginär- und Realteile der Funktion $\frac{f}{f'}$

$$\frac{f(z)}{f'(z)} = \frac{1}{u_x^2 + v_x^2} \cdot \{(u \cdot u_x + v \cdot v_x) + i(u \cdot u_y + v \cdot v_y)\} \quad (1)$$

berechnet und als String zurückschickt. Dieser String wird dann geparsst. Hierbei mussten einige Module selbst geschrieben werden, zum Beispiel eines, welches die Syntax für Hochzahlen, der im String zurückgegebenen Funktion, von z.B. " $3a^3 + 5b^{(-10+3a)}$ " zu " $pow(3a, 3) + pow(5b, -10 + 3a)$ " umwandelt. Ebenso mussten alle Zahlen als Kommazahlen geschrieben werden (also z.B. " $3.3a^3 + 5b^{(-10+3a)}$ " wird zu " $pow(3.3a, 3.0) + pow(5.0b, -10.0 + 3.0a)$ "). Diese Module können separat getestet werden, siehe Abschnitt 5.5

5.4 GLSL

Die Sprache GLSL erlaubt im Wesentlichen die Erstellung von Vertices (Abstrakte Größe), und eine Zuordnung einer Funktion, welche für diesen Vertex ausgeführt werden soll. Das Laden und Erstellen der Vertices in den sogenannten Buffer, nimmt einige Sekunden in Anspruch, allerdings kann daraufhin relativ schnell gerechnet werden. Ein wesentlicher Vorteil von diesem Standard ist, dass er auf allen Betriebssystemen und Geräten, welche über einen modernen Browser verfügen, verstanden wird (also auch auf mobilen Geräten).

5.5 Anleitung

Um sich selbst von den Möglichkeiten der parallelen Berechnungen zu überzeugen, muss zunächst der Python Server `\newton_c++\momo#sPyServer.py` gestartet werden (das geht zum Beispiel über die Konsole durch Navigieren in den entsprechenden Ordner und einen Startbefehl der Datei mit Python/Python3). Dies startet einen lokalen Python Server, der statische Files senden kann und als API dient, welche Funktionen f als String als Input und den Imaginär-/Realteil von f/f' als Output hat. Die Seite kann aufgerufen werden, indem man in die Browser Suchleiste: `http://localhost:5000/` eingibt (falls die GUI auftauchen soll). Alternativ kann auch `http://localhost:5000/req` eingegeben werden, um eine Seite aufzurufen, die demonstriert, wie die Syntax umgewandelt wird. Nun können die folgenden Daten zum Testen eingegeben werden:

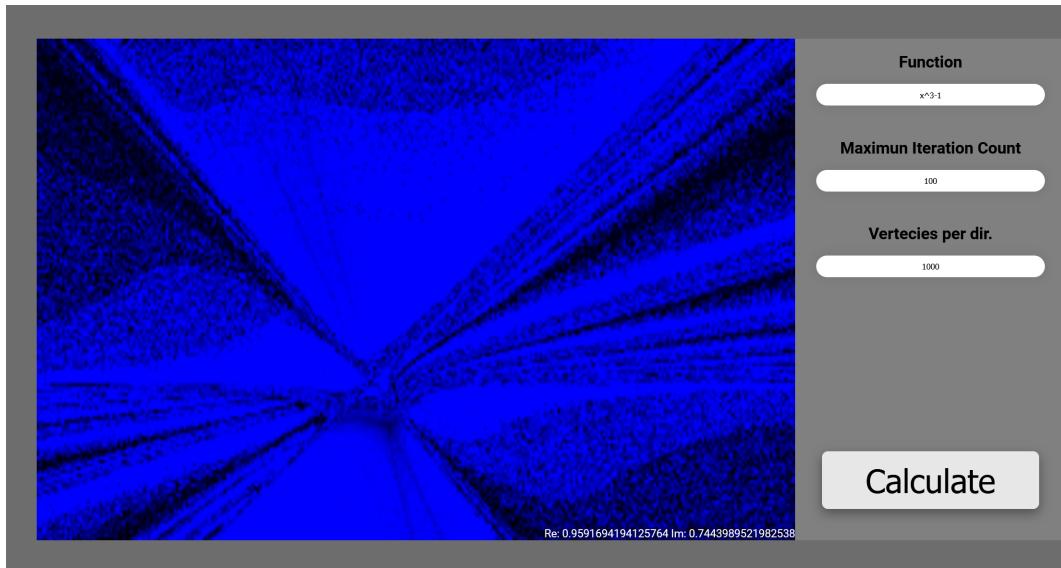


Abbildung 13: Webanwendung, Beispiel eingaben (1000 Vertices)

Um nur die Umwandlung der Syntax zu testen, können zum Beispiel die Daten aus Abbildung 14 eingegeben werden (es muss entsprechend im Browser zur richtigen Seite navigiert werden)

```

x^3-1
re: (a**3 - 3*a*b**2 + b*(3*a**2 - b**2) - 1)/(6*(a**2 - b**2))
im: a*b*(-a**3 + 3*a*b**2 - b*(3*a**2 - b**2) + 1)/(3*(a**2 - b**2)**2)

SYNTAX re: (pow(a,3) - 3*a*pow(b,2) + b*(3*pow(a,2) - pow(b,2)) - 1)/(6*(pow(a,2) - pow(b,2)))
SYNTAX im: a*b*(-pow(a,3) + 3*a*pow(b,2) - b*(3*pow(a,2) - pow(b,2)) + 1)/(3*pow((pow(a,2) - pow(b,2)),2))

SYNTAXX re: (pow(a,3.0) - 3.0*a*pow(b,2.0) + b*(3.0*pow(a,2.0) - pow(b,2.0)) - 1.0)/(6.0*(pow(a,2.0) - pow(b,2.0)))
SYNTAXX im: a*b*(-pow(a,3.0) + 3.0*a*pow(b,2.0) - b*(3.0*pow(a,2.0) - pow(b,2.0)) + 1.0)/(3.0*pow((pow(a,2.0) - pow(b,2.0)),2.0))

JUST RELOAD THE PAGE TO SEE NEW RESULTS

```

Abbildung 14: Webanwendung, Beispiel eingaben für Syntax Umwandlung

5.6 Zusammenfassung

Zusammenfassend lässt sich zum letzten Teil sagen, dass die Ausgabe nicht ganz korrekt ist, aber innerhalb von etwa einer Sekunde circa 1000 mal 1000 Pixel berechnen kann. Vermutlich liegt die fehlerhafte Ausgabe an einem Fehler im Code, der in der Zeit nicht behoben werden konnte. Trotzdem ist klar zu erkennen wie viel schneller die parallelen Berechnungen laufen als bei Hintereinanderausführung.

Literatur

[HSV-Farbraum] „HSV-Farbraum – Wikipedia.“ Accessed February 15, 2024.
<https://de.wikipedia.org/wiki/HSV-Farbraum>.

[Newtonverfahren] „Newtonverfahren – Wikipedia.“ Accessed February 15, 2024.
<https://de.wikipedia.org/wiki/Newtonverfahren>.