

# Das Tricorn-Fractal

## ERA-Praktikum - Team 167

Nicolás Arteaga, Manuel Cárdenas, Erick Quintanar

# Inhalt

- Das Problem
- Unsere Überlegungen dazu
- Unsere Implementierung
- Ergebnisse

# Das Problem

- Multicorn Fractal, gegeben durch:

$$z_{i+1} = \overline{z_i}^m + c \quad (i > 0)$$

# Das Problem

- Multicorn Fractal, gegeben durch:

$$z_{i+1} = \overline{z_i}^m + c \quad (i > 0)$$

- Spezifischer, das Tricorn Fractal

# Das Problem

- Multicorn Fractal, gegeben durch:

$$z_{i+1} = \overline{z_i}^m + c \quad (i > 0)$$

- Spezifischer, das Tricorn Fractal: Multicorn mit **m = 2**

# Das Problem

- Multicorn Fractal, gegeben durch:

$$z_{i+1} = \overline{z_i}^m + c \quad (i > 0)$$

- Spezifischer, das Tricorn Fractal: Multicorn mit **m = 2**

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

$$z_{i+1} = \overline{z_i}^2 + c \ (i > 0)$$

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

- Zi in der **Komplexen Ebene** zu betrachten

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

- **Zi in der Komplexen Ebene zu betrachten**
- Man betrachte **Verhalten der Formel** für verschiedene **c**

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

- Zi in der **Komplexen Ebene** zu betrachten
- Man betrachte **Verhalten der Formel** für verschiedene **c** innerhalb vom **Benutzer** eingegeben Intervall:

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

- **Zi** in der **Komplexen Ebene** zu betrachten
- Man betrachte **Verhalten der Formel** für verschiedene **c** innerhalb vom **Benutzer** eingegeben Intervall:
  - Falls **Zi beschränkt**: Schwarz färben

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

- **Zi** in der **Komplexen Ebene** zu betrachten
- Man betrachte **Verhalten der Formel** für verschiedene **c** innerhalb vom **Benutzer** eingegeben Intervall:
  - Falls **Zi beschränkt**: Schwarz färben
  - Falls **Zi unberschränkt**: Farbscala

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

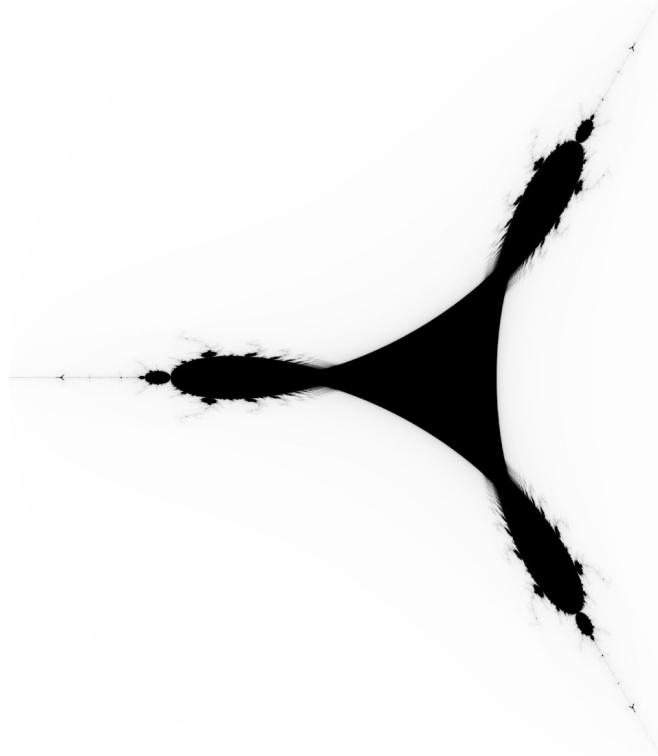
- **Zi** in der **Komplexen Ebene** zu betrachten
- Man betrachte **Verhalten der Formel** für verschiedene **c** innerhalb vom **Benutzer** eingegeben Intervall:
  - Falls **Zi beschränkt**: Schwarz färben
  - Falls **Zi unberschränkt**: Farbscala --> wie unbeschränkt. D.h. wie viele **Iterationen** gebraucht sind um zu merken, dass **Zi** unbeschränkt ist

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

- **Zi** in der **Komplexen Ebene** zu betrachten
- Man betrachte **Verhalten der Formel** für verschiedene **c** innerhalb vom **Benutzer** eingegeben Intervall:
  - Falls **Zi beschränkt**: Schwarz färben
  - Falls **Zi unberschränkt**: Farbscala --> wie unbeschränkt. D.h. wie viele **Iterationen** gebraucht sind um zu merken, dass **Zi** unbeschränkt ist
- --> Entsteht eine **Figur**

# Die Figur

# Die Figur



Aus: [https://en.wikipedia.org/wiki/Tricorn\\_\(mathematics\)#/media/File:Tricorn.png](https://en.wikipedia.org/wiki/Tricorn_(mathematics)#/media/File:Tricorn.png)

# Praktischen Teil der Aufgabe

# Praktischen Teil der Aufgabe

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

# Praktischen Teil der Aufgabe

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

- `r_start`, `r_end`, `i_start` und `i_end` enthalten das Bereich wo das Bild gezeichnet werden soll

# Praktischen Teil der Aufgabe

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

- `r_start`, `r_end`, `i_start` und `i_end` enthalten das Bereich wo das Bild gezeichnet werden soll
- `res` enthält die Resolution des Bildes

# Praktischen Teil der Aufgabe

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

- `r_start`, `r_end`, `i_start` und `i_end` enthalten das Bereich wo das Bild gezeichnet werden soll
- `res` enthält die Resolution des Bildes
- `img` -> Pointer zur Speicher wo das Bild gebaut werden soll

# Inhalt

- ~~Das Problem~~

- Unsere Überlegungen dazu
- Unsere Implementierung
- Ergebnisse

# Unsere Überlegungen

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

# Unsere Überlegungen

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

- Erste Fragen:
  - Attribut **res**
  - Wann ist **Zi** beschränkt

# Unsere Überlegungen

- Funktion in Assembler

```
void multicorn(float r_start, float r_end, float i_start, float  
i_end, float res, unsigned char* img)
```

- Erste Fragen:
  - Attribut **res**
  - Wann ist **Zi** beschränkt
- Technische Entscheidungen

# Erste Fragen

- **Auflösung**

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes
    - multipliziert Breite und Länge

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes
    - multipliziert Breite und Länge
    - Skalierung von jedem Pixel:  $1/\text{res}$

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes
    - multipliziert Breite und Länge
    - Skalierung von jedem Pixel:  $1/\text{res}$
- **Wann ist Zi beschränkt**

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes
    - multipliziert Breite und Länge
    - Skalierung von jedem Pixel:  $1/\text{res}$
- **Wann ist Zi beschränkt**
  - **Zi** unbeschränkt --> Ausserhalb Bereich  $[-2, 2]$

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes
    - multipliziert Breite und Länge
    - Skalierung von jedem Pixel:  $1/\text{res}$
- **Wann ist Zi beschränkt**
  - **Zi** unbeschränkt --> Ausserhalb Bereich  $[-2, 2]$
  - **Zi** beschränkt: nicht unbeschränkt nach 100 Iterationen

# Erste Fragen

- **Auflösung**
  - Skalar des Bildes
    - multipliziert Breite und Länge
    - Skalierung von jedem Pixel:  $1/\text{res}$
- **Wann ist Zi beschränkt**
  - **Zi** unbeschränkt --> Ausserhalb Bereich  $[-2, 2]$
  - **Zi** beschränkt: nicht unbeschränkt nach 100 Iterationen
    - 100 --> gutes Mittelwert

# Technische Entscheidungen

- Eingabe

# Technische Entscheidungen

- **Eingabe**
  - Unkorrektes Codeverhalten soll vermieden werden
  - Auflösung darf nicht 0 sein

# Technische Entscheidungen

- **Eingabe**
  - Unkorrektes Codeverhalten soll vermieden werden
  - Auflösung darf nicht 0 sein
- **Ausgabe**

# Technische Entscheidungen

- **Eingabe**
  - Unkorrektes Codeverhalten soll vermieden werden
  - Auflösung darf nicht 0 sein
- **Ausgabe**
  - Bitmap: 24-bit BMP Datei --> Padding soll betrachtet werden

# Technische Entscheidungen

- **Eingabe**
  - Unkorrektes Codeverhalten soll vermieden werden
  - Auflösung darf nicht 0 sein
- **Ausgabe**
  - Bitmap: 24-bit BMP Datei --> Padding soll betrachtet werden
  - Speicherallokation:

# Technische Entscheidungen

- **Eingabe**
  - Unkorrektes Codeverhalten soll vermieden werden
  - Auflösung darf nicht 0 sein
- **Ausgabe**
  - Bitmap: 24-bit BMP Datei --> Padding soll betrachtet werden
  - Speicherallokation:
    - Platz für alle Pixeln (+ Padding)
    - Darf in C gemacht werden

# Technische Entscheidungen

- Calling Convention

# Technische Entscheidungen

- **Calling Convention**
  - Zwei Dateien in Assembler: multicore.S, tricore.S  
--> Kommunikation

# Technische Entscheidungen

- **Calling Convention**
  - Zwei Dateien in Assembler: multicore.S, tricore.S  
--> Kommunikation
  - Werte werden durch die Register weitergegeben

# Technische Entscheidungen

- **Calling Convention**

- Zwei Dateien in Assembler: multicore.S, tricore.S  
--> Kommunikation
- Werte werden durch die Register weitergegeben
- Callee safe + temporär

# Inhalt

- ~~Das Problem~~
- ~~Unsere Überlegungen dazu~~
- Unsere Implementierung
- Ergebnisse

# Unsere Implementierung

# Unsere Implementierung

## 1. Eingabe wird verarbeitet

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert
3. Es wird durch Speicher iteriert

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert
3. Es wird durch Speicher iteriert
  - a. Für jedes Pixel (=Wert fur c) --> beschränkt oder nicht

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert
3. Es wird durch Speicher iteriert
  - a. Für jedes Pixel (=Wert fur c) --> beschränkt oder nicht
  - b. Pixel wird gefärbt

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert
3. Es wird durch Speicher iteriert
  - a. Für jedes Pixel (=Wert fur C) --> beschränkt oder nicht
  - b. Pixel wird gefärbt

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert
3. Es wird durch Speicher iteriert
  - a. Für jedes Pixel (=Wert fur C) --> beschränkt oder nicht
  - b. Pixel wird gefärbt

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

# Unsere Implementierung

1. Eingabe wird verarbeitet
2. Platz für BMP in Speicher wird allokiert
3. Es wird durch Speicher iteriert
  - a. Für jedes Pixel (=Wert fur C) --> beschränkt oder nicht
  - b. Pixel wird gefärbt

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

## Parallelisierung mit SIMD

# Berechnung von

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

in **C**

```
int tricorn(float* z, float* c) {
    int MAX_ITERATIONS = 100;

    int i = 0; //iterations
    while (i < MAX_ITERATIONS)
    {
        squareConjugate(z);
        complexAddition(z, c);
        //printf("r:%f, i:%f \n", *z, *(z+1));
        i = i+1;
        if (check(z)) {
            //printf("unstable i:%i\n", i);
            return i;
        }
    }
    return 0;
}
```

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

# Und In Assembler (mit SIMD)

```
--  
22 tricorn:  
23  
24     # First Function Iteration C=0  
25     mov r8,0  
26     cvtsi2ss xmm2, r8  
27     cvtsi2ss xmm3, r8  
28  
29     mov rdx, 100          # max iteration number  
30     mov rcx, 0           # loop counter  
31  
32 start_loop:  
33     cmp rcx, rdx  
34     jge end_loop  
35
```

```
36 squareConjugate:          # squares and conjugates z
37     mov r8, -1
38     cvtsi2ss xmm4, r8
39     mulps xmm3, xmm4    # zImaginary negative -> conjugate
40
41     # (x, y)* i(x', y') -> (xx' - yy') + i(xy' + x'y) -> square
42     movups xmm4, xmm2   # helper registers
43     movups xmm5, xmm3   # helper registers
44
45     mulps xmm4, xmm4   # xx'
46     mulps xmm5, xmm5   # yy'
47     subps xmm4, xmm5   # (xx' - yy')
48
49     movups xmm5, xmm4   # xmm5 as temp
50     movups xmm4, xmm2
51     movups xmm2, xmm5   # xmm0 = (xx' - yy')
52
53     movups xmm5, xmm3
54     mulps xmm4, xmm5   # xy'
55     addps xmm4, xmm4   # (xy' + x'y), since x=x' and y=y'
56     movups xmm3, xmm4   # xmm1 = (xy' + x'y)
```

$$z_{i+1} = \boxed{\overline{z_i}^2} + c \quad (i > 0)$$

# Unsere Implementierung

```
58    complexAddition:  
59        addss xmm2, xmm0  
60        addss xmm3, xmm1
```

$$z_{i+1} = \overline{z_i}^2 + c \quad (i > 0)$$

```
64 check:                      # checks if unstable
65     mov r8,-2                  # constants we compare with, if number out
66     cvtsi2ss xmm5,r8          # of interval [-2, 2] it is unstable
67     mov r8,2
68     cvtsi2ss xmm4,r8
69
70     ucomiss xmm2,xmm5      # cmp -2 < x -> false
71     jb unstable
72
73     ucomiss xmm2,xmm4      # cmp 1 > x -> false
74     ja unstable
75
76     ucomiss xmm3,xmm5      # cmp -1 < x -> false
77     jb unstable
78
79     ucomiss xmm3,xmm4      # cmp 1 > x -> false
80     ja unstable
81
82     mov rax,0                  # not unstable
83     jmp finish_check
```

```
64 check:                      # checks if unstable
65     mov r8,-2                  # constants we compare with, if number out
66     cvtsi2ss xmm5,r8          # of interval [-2, 2] it is unstable
67     mov r8,2
68     cvtsi2ss xmm4,r8
69
70     ucomiss xmm2,xmm5      # cmp -2 < x -> false      85    unstable:
71     jb unstable
72
73     ucomiss xmm2,xmm4      # cmp 1 > x -> false      86    |   mov rax,rcx
74     ja unstable
75
76     ucomiss xmm3,xmm5      # cmp -1 < x -> false      87    |
77     jb unstable
78
79     ucomiss xmm3,xmm4      # cmp 1 > x -> false      88    finish_check:
80     ja unstable
81
82     mov rax,0                 # not unstable           89    |   cmp rax, 0
83     jmp finish_check        90    |   je start_loop
```

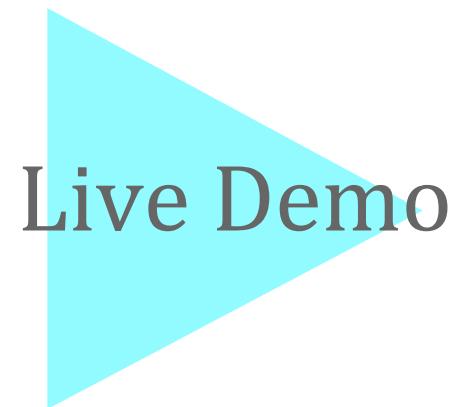
# Falls unbeschränkt

```
95    mov al, cl          # Scalaring red color
96    imul ax,10
97    add al,60
98    mov dl, cl          # Scalaring green color
99    imul dx,10
100   add dl,100
101   imul cx,5          # Scalaring blue color
102   neg cx
103   add cl, 220
104
105  mov byte ptr [rdi], cl
106  mov byte ptr [rdi+1], dl
107  mov byte ptr [rdi+2], al
108
109  ret
```

# Falls beschränkt

```
110    end_loop:  
111  
112        xor al, al  
113        xor dl, dl  
114        xor cl, cl  
115        mov byte ptr [rdi], cl  
116        mov byte ptr [rdi+1], dl  
117        mov byte ptr [rdi+2], al  
118        ret
```

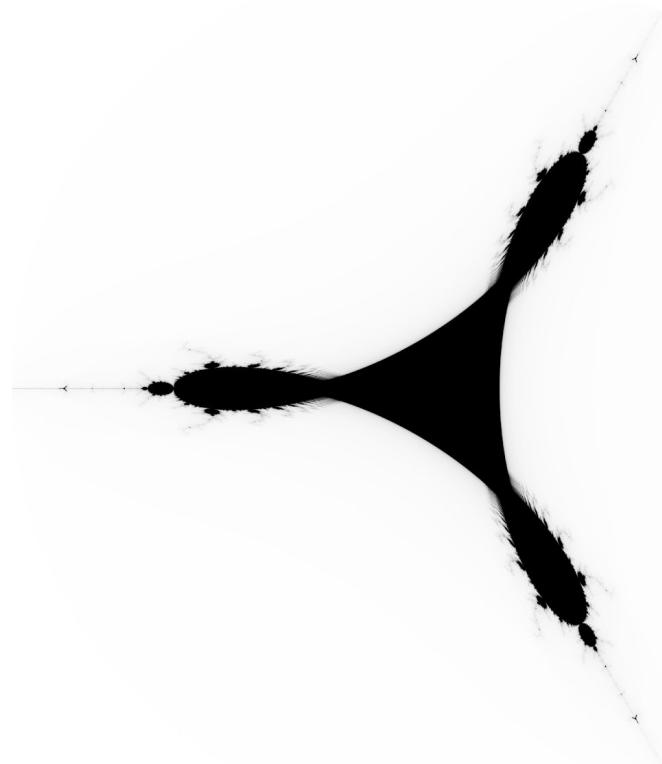
# Live Demo



# Inhalt

- ~~Das Problem~~
- ~~Unsere Überlegungen dazu~~
- ~~Unsere Implementierung~~
- Ergebnisse

# Ergebnisse



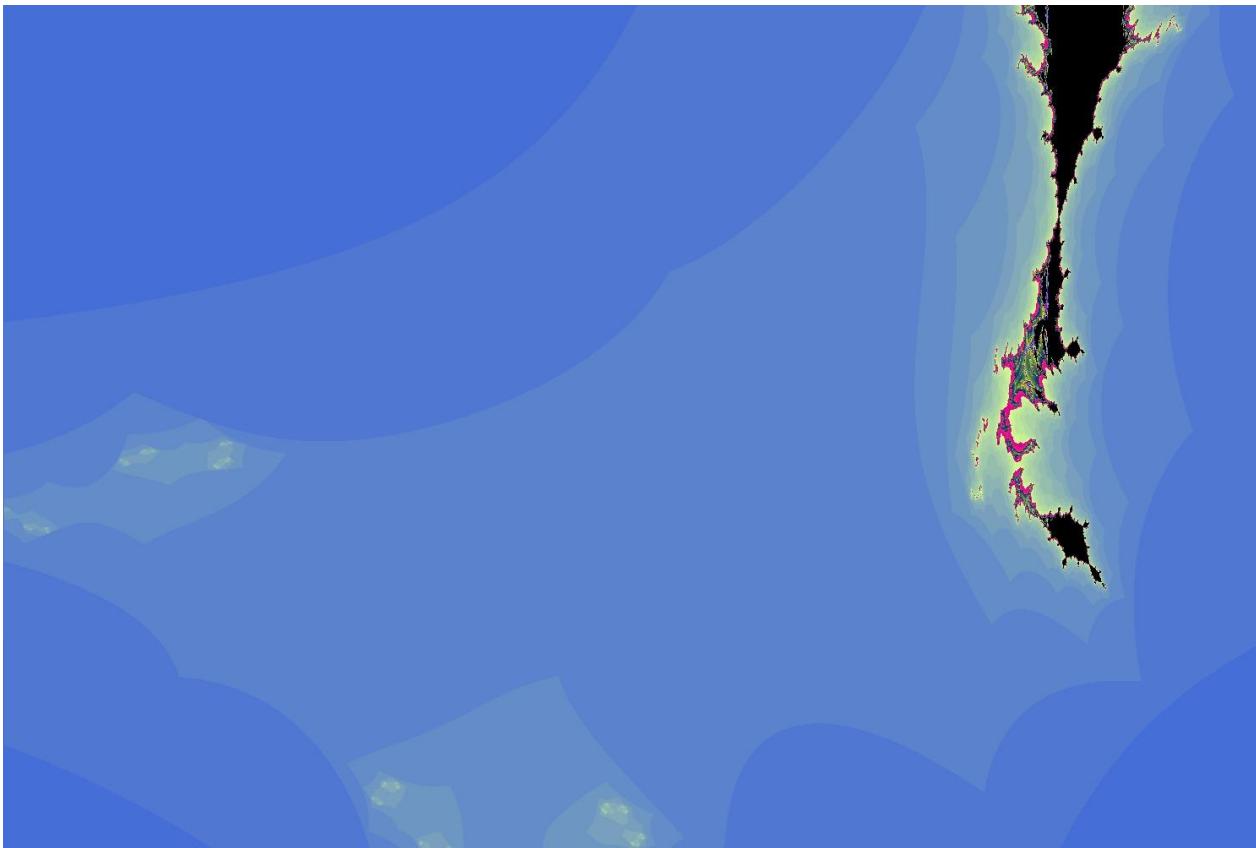
# Ergebnisse



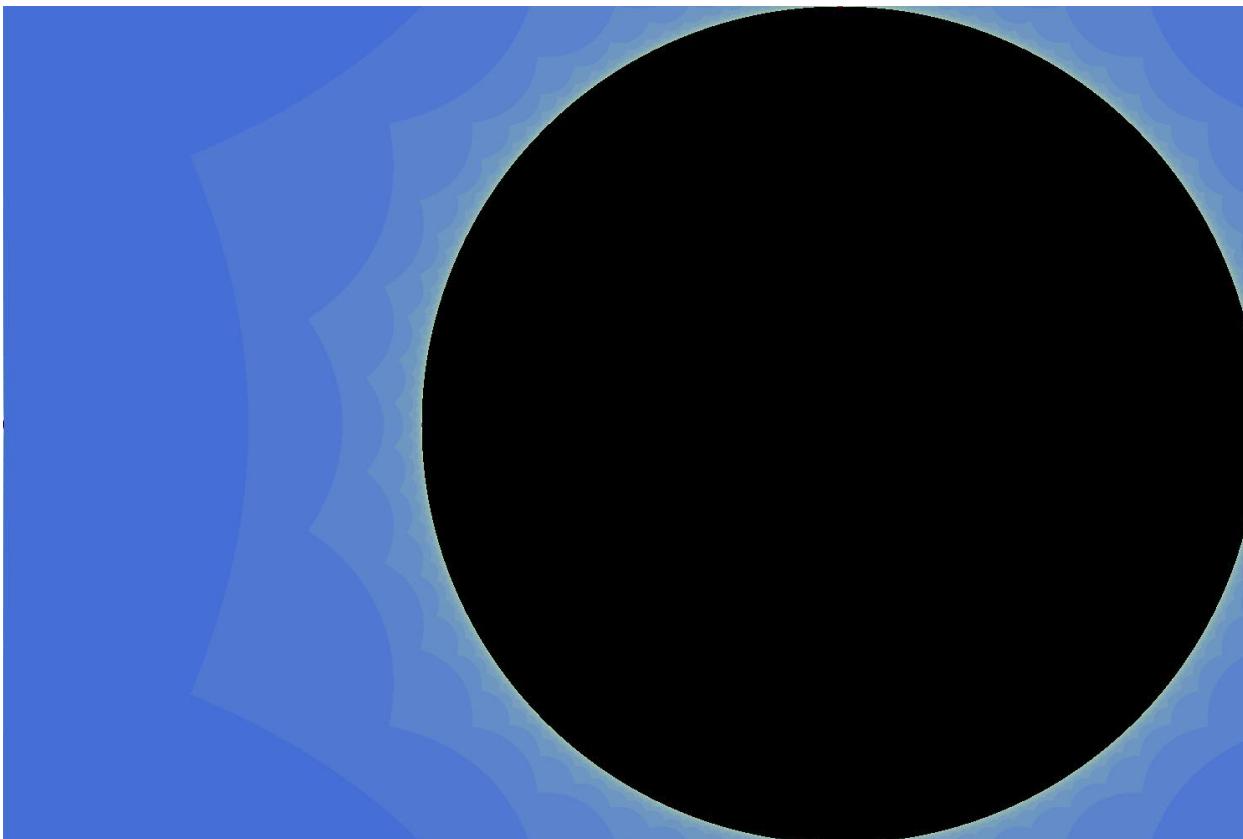
# Ergebnisse

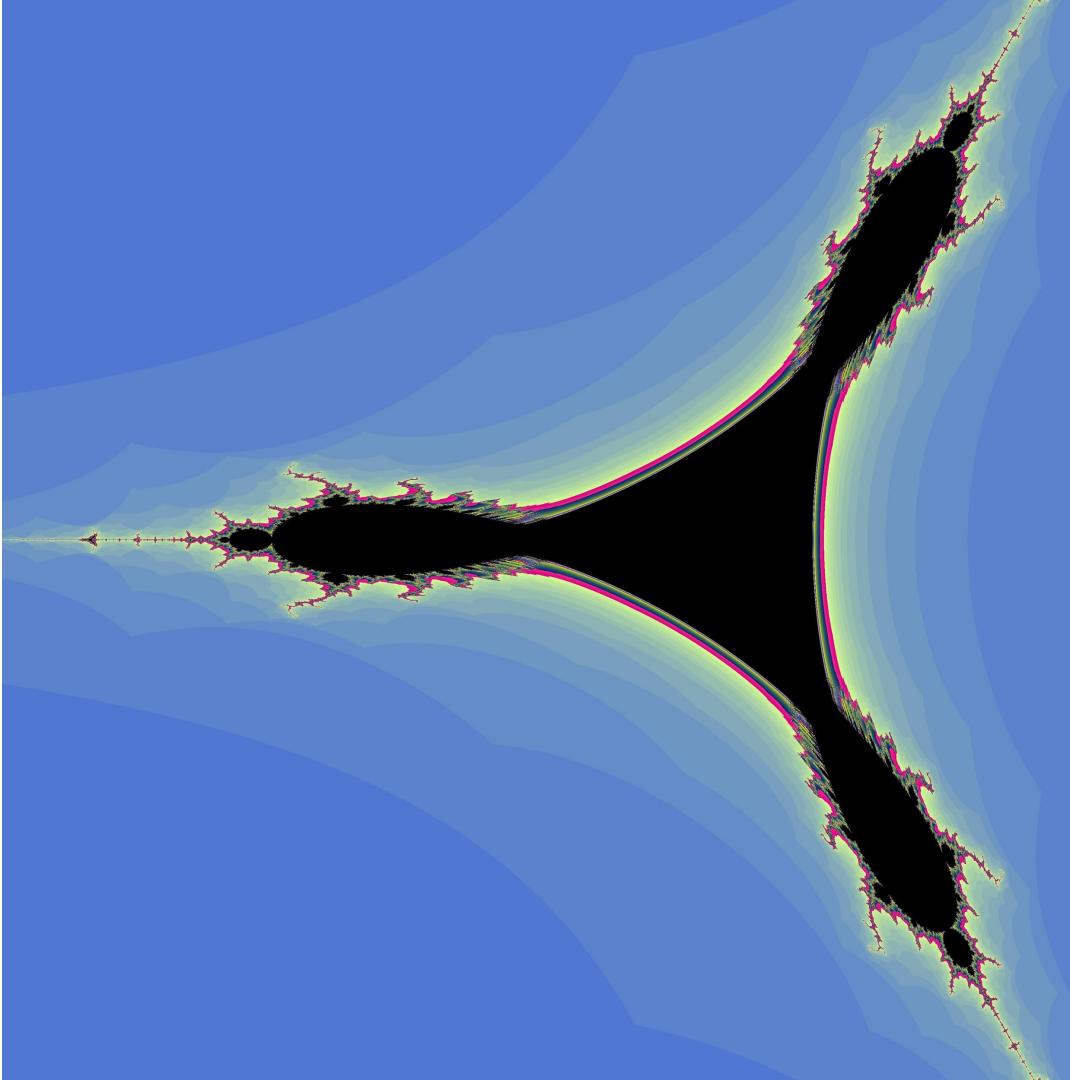


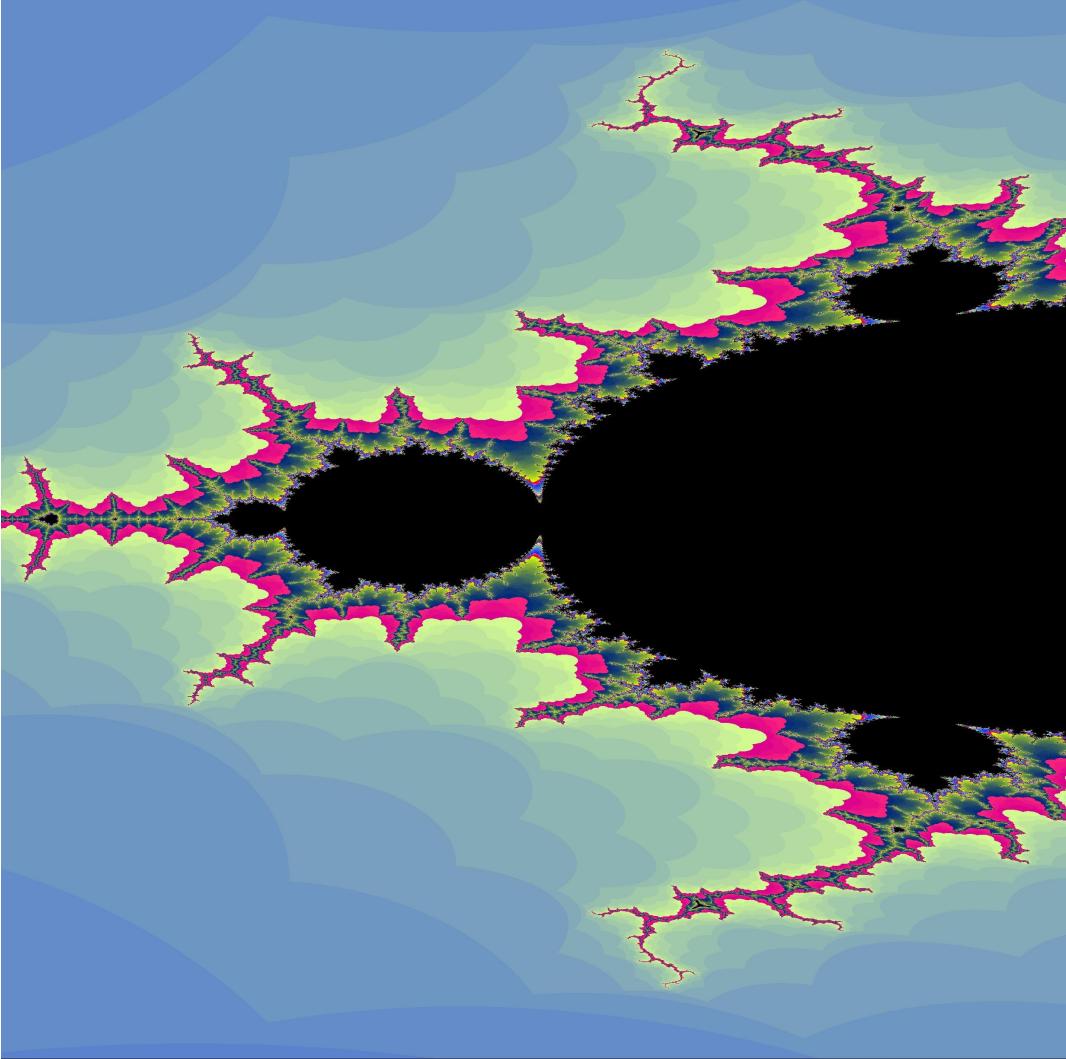
# Ergebnisse



# Ergebnisse







# Ergebnisse

Tabelle 1: Proz. Intel i5-5200U, Zeit für jede Implementierung

	250	500	750	1000	2500	5000
<b>C</b>	0.0454124	0.157775	0.3436602	0.605639	3.7216696	14.798096
<b>ASM</b>	0.0650032	0.2352244	0.5176214	0.9119462	5.649717	22.7363514
<b>ASM_SimD</b>	0.03399	0.1066764	0.2285984	0.3981316	2.4923508	9.6546974

# Ergebnisse

Tabelle 1: Proz. Intel i5-5200U, Zeit für jede Implementierung

	250	500	750	1000	2500	5000
C	0.0454124	0.157775	0.3436602	0.605639	3.7216696	14.798096
ASM	0.0650032	0.2352244	0.5176214	0.9119462	5.649717	22.7363514
ASM_SimD	0.03399	0.1066764	0.2285984	0.3981316	2.4923508	9.6546974

- SIMD lohnt sich --> schneller als unsere mit O3 optimierte Implementierung

