# Mandelbrotmenge (Apfelmännchen) - Theorie und Programmierung

*K.Achilles 2012-2018*

Das Thema Mandelbrotmenge bzw. "Apfelmännchen" gehört zum Oberthema

**"Chaos und Ordnung in dynamischen Systemen"** .

Es ist ein relativ neues Forschungsgebiete der Mathematik ( ab ca. 1980 ).

Namensgeber für die zu erforschende Menge ist der französisch-US-amerikanische Mathematiker **Benoît B. Mandelbrot** (1924-2010) .

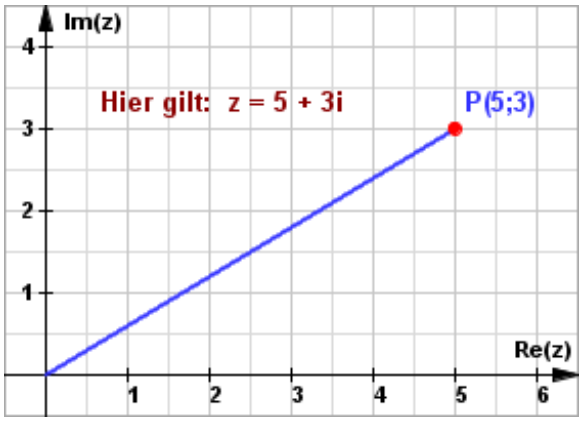
Die Mandelbrotmenge ist eine durch Iteration einer Zahlenfolge entstehende Punktmenge, die zu sehr interessanten, ja sogar schönen Bildern führt.

Ausgangspunkt ist die komplexe Iterationsformel $\mathbf{z_{n+1} = z_n^2 + c =: f(z_n)}$ .

Sowohl z als auch c sind komplexe Zahlen der Form $a + i \cdot b$ , wobei **a der Realteil** und **b der Imaginärteil** der Zahl ist (s. Grafik). Man schreibt hierfür auch: $a = Re(z)$ und $b = Im(z)$

Darstellung in der Gaußschen Zahlenebene:

Beispiel: $z = 5 + 3i$




Für jede komplexe Zahl $c = a + i \cdot b$ eines Ausschnitts aus der Gaußschen Zahlenebene wird die Zahlenfolge $\{z_n\}$ gemäß obiger Vorschrift ( $\mathbf{z_{n+1} = z_n^2 + c}$ ) auf **Konvergenz** geprüft, d.h. es wird untersucht, ob die Folge einen Grenzwert besitzt.

Dies bedeutet u.a., dass die Folge der Zahlen stets innerhalb eines endlichen Bereichs bleibt.
Besitzt die Zahlenfolge jedoch keinen Grenzwert, so wächst sie „über alle Schranken", sie „**divergiert**" .

Falls die Folge $\{z_n\}$ einen Grenzwert besitzt, so soll der Bildpunkt von c in der Gaußschen Zahlenebene gezeichnet werden.
Divergiert die Folge, so wird kein Punkt gezeichnet.

Das Gesamtbild aller derart gezeichneten Bildpunkte heißt **Apfelmännchen** bzw. **Mandelbrotmenge** , eine sog. „selbstähnliche" Figur (s.u.).

Da Konvergenz mit dem Computer nicht feststellbar ist, begnügen wir uns mit folgender Forderung:

*Die Folge $\{z_n\}$ sei konvergent, wenn ihr Betrag $|z| = \sqrt{(z_x^2 + z_y^2)}$ (das ist die Strecke vom Ursprung (0;0) bis zur Zahl $z = z_x + i \cdot z_y := (z_x , z_y)$ ) nach einer festen Anzahl von Iterationsschritten ( z.B. n = 500 Schritte ) die endliche Schranke 2 nicht überschreitet.*

Anmerkung:

Dass 2 als Schranke ausreicht, lässt sich übrigens mathematisch beweisen.

Rechenbeispiel für $c = -0{,}5 + 1{,}0 \cdot i$ , also $Re(z) = -0{,}5$ $Im(z) = 1{,}0$ :

Gestartet wird immer mit $z_0 = 0$ .

1.Schritt: $z_1 = 0^2+(-0{,}5+i) = -0{,}5+i$
2.Schritt: $z_2 = (-0{,}5+i)^2+(-0{,}5+i) = 0{,}25-1-i-0{,}5+i = -1{,}25$
3.Schritt: $z_3 = (-1{,}25)^2+(-0{,}5+i) = 1{,}5625-0{,}5+i = 1{,}0625+i$
4.Schritt: $z_4 = (1{,}0625+i)^2+(-0{,}5+i) = 1{,}12890625-1+2{,}125i-0{,}5+i = -0{,}37109375+3{,}125i$

Wie man leicht prüft, ist der Betrag dieser Zahl ( ≈3,1 ) bereits größer als 2, und somit gehört der Punkt (-0,5 ; 1,0) nicht zur Mandelbrotmenge.

Hinweis:

Eine grafische Darstellung der Iteration findet sich im Anhang des Artikels unter "**Mandelbrot-Orbits**".

*Der vorläufige Algorithmus für die Iteration der Folge:*

Vorgegeben sind: Iterationstiefe maxIt (z.B. 500) sowie $c = (Re(c) , Im(c))$
$z = 0$
$k = 0$
wiederhole
    $z = z^2 + c$
    $k = k+1$
bis $|z| > 2$ oder $k = maxIt$
falls $k = maxIt$ dann zeichne $P(Re(c),Im(c))$

Präzisierung des Algorithmus durch Verwendung der Regeln bei komplexen Zahlen:

Setzt man $z_x = Re(z)$, $z_y = Im(z)$, $c_x = Re(c)$, $c_y = Im(z)$ , so erhält man ( wegen $i^2 = -1$ ) :

$z^2 = (z_x + i \cdot z_y)^2 = z_x^2 - z_y^2 + i \cdot 2 \cdot z_x \cdot z_y$

$z := z^2 + c = z_x^2 - z_y^2 + i \cdot 2 \cdot z_x \cdot z_y + c_x + i \cdot c_y$, also

$z := z_x^2 - z_y^2 + c_x + i \cdot (2 \cdot z_x \cdot z_y + c_y)$

Außerdem gilt in der Menge der komplexen Zahlen: $|z| = \sqrt{(z_x^2 + z_y^2)}$ bzw. $|z|^2 = z_x^2 + z_y^2$

Anmerkung:

Zur Darstellung der Mandelbrotmenge genügt als Ausschnitt aus der Gaußschen Zahlenebene z.B.

**$-2{,}02 \le x \le 0{,}7$ ; $-1{,}2 \le y \le 1{,}2$** Verhältnis 2,72 : 2,4 bzw. 17:15

Als Auflösung wählen wir **459** Pixels (x-Bereich) mal **405** Pixels (y-Bereich), womit wir eine verzerrungsfreie Darstellung erhalten ( Verhältnis 17:15 ).
**Tipp:** Für den y-Bereich unbedingt eine ungerade Pixelzahl (hier: 405) wählen, damit es ein Pixel gibt, welches exakt in der Mitte liegt ! Bei 405 ist das Pixel Nr. 202 ( = 405 div 2 ) in der Mitte von 0 ... 404 . ( In den meisten Programmiersprachen beginnen Bereiche mit dem Index 0 statt 1, so auch in Java)

Höhere Auflösungen sind natürlich möglich, erhöhen jedoch die Rechenzeit !

Für jeden Punkt $C(c_x ; c_y)$ aus dem oben gewählten Ausschnitt wird dann iteriert.

**Algorithmus (Pseudocode):**

```
Algorithmus Mandelbrot
   imageBreite = 459   imageHoehe = 405
   xa = -2,02   xe = 0,7   ya = -1,2   ye = 1,2
   dx = (xe - xa) / (imageBreite-1)
   dy = (ye - ya) / (imageHoehe-1)
   maxIt = 500
   für sp=0 bis imageBreite-1 wiederhole
       cx = xa + sp * dx   // cx von links nach rechts
       für ze=0 bis  imageHoehe-1 wiederhole
           cy = ye - ze * dy   // cy von oben nach unten
           falls iterZahl(cx, cy, maxIt) = maxIt
               dann setzePunkt(sp,ze)
       ende ze
   ende sp

Algorithmus iterZahl(cx, cy, maxIt)
   zx = 0  zy = 0
   zaehler=0
   wiederhole  // iteriere
      // tmp verwenden, da die alten zx, zy im nächsten Schritt noch benötigt werden !
      tmp = zx*zx - zy*zy + cx
      zy = 2*zx*zy + cy
      zx = tmp
      zaehler = zaehler + 1
   bis (zx*zx + zy*zy > 4.0) oder (zaehler = maxIt)
   gib zaehler zurück
```

Hinweis für Geschwindigkeits-Interessierte:

Die Doppelschleife ( für sp ... ; für ze ... ) im Algorithmus "Mandelbrot" könnte man durch eine **schnellere Version** ersetzen, welche Multiplikationen vermeidet:

```
   cx = xa
   für sp=0 bis imageBreite-1 wiederhole
       cy = ye
       für ze=0 bis  imageHoehe-1 wiederhole
           falls iterZahl(cx, cy, maxIt) = maxIt
               dann setzePunkt(sp,ze)
           cy = cy - dy
       ende ze
       cx = cx + dx
   ende sp
```

Dies ist allerdings problematisch, da der Wert cy = 0 (x-Achse) übergangen wird (!), was an der Ungenauigkeit der Gleitpunktarithmetik von Programmiersprachen liegt.
Beispiel: Die 202-malige Berechnung von cy = cy - dy sollte cy = 1,2 - 202 * 2,4 / 404 = 0 liefern, jedoch berechnet die Java-Gleitpunktarithmetik den Wert `-1.0390993621101074E-15` .

Da aber cy = 0.0 im Bereich -2 ≤ cx ≤ 0 sehr wohl zur Mandelbrotmenge gehört (dies ist rechnerisch sehr leicht zu zeigen), sollte man die Iteration für cy = 0 separat abarbeiten:

```
       cy = 0   ze =  (int) (ye / (ye - ya) * (imageHoehe-1) ...
```

Liegt die x-Achse jedoch nicht im zu untersuchenden Bereich , z.B. y∈ [0,1 ; 1,4], dann muss eben doch der obige Algorithmus verwendet werden .

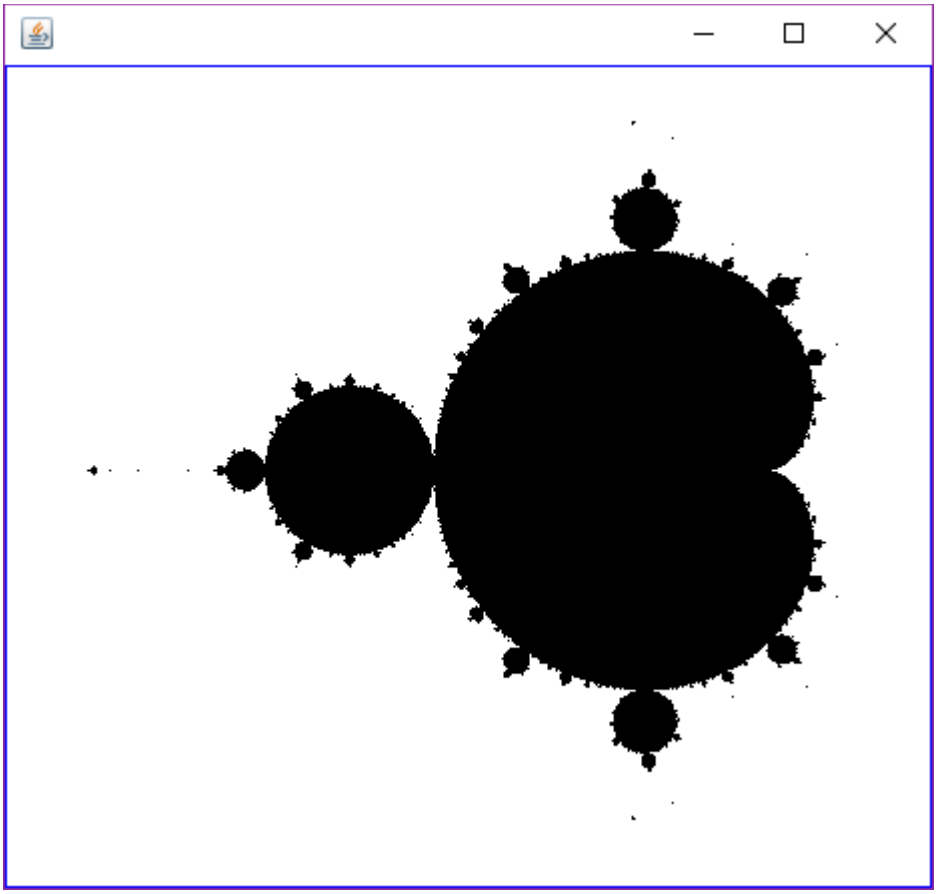
Programmierung der Mandelbrotmenge mit JAVA (auf einem JFrame):

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Insets;
import javax.swing.JFrame;

public class JFrameMandelbrot extends JFrame {  // Ac 04-2018
	public static void main(String[] args){
		new JFrameMandelbrot();
	}

	int imageBreite = 459, imageHoehe = 405;
	int frameBreite = imageBreite + 20, frameHoehe = imageHoehe + 45;

	public JFrameMandelbrot() { // Konstruktor
		setDefaultCloseOperation(EXIT_ON_CLOSE);
		setBounds(300, 350, frameBreite, frameHoehe);
		setVisible(true);
	}

	public void paint(Graphics g) {
		Insets ins = getInsets();
		int obRand = ins.top, untRand = ins.bottom, liRand = ins.left, reRand = ins.right;
		g.setColor(Color.BLUE);
		// Rahmen um das Grafikfenster zeichnen
		g.drawRect(liRand, obRand, frameBreite - liRand - reRand - 1, frameHoehe - obRand -
                                                                              untRand - 1);
		zeichneMandelbrotmenge(g, liRand, obRand);
	}

	public int iterZahl(final double cx, final double cy, int maxIt) {
	   // bestimmt Anzahl der Iterationen
		int zaehler = 0;
		double zx = 0.0, zy = 0.0, tmp;
		do {
			tmp = zx*zx - zy*zy + cx;
			zy = 2*zx*zy + cy;
			zx = tmp;
			zaehler = zaehler + 1;
		} while (zx*zx + zy*zy <= 4.0 && zaehler < maxIt);
		return zaehler;
	}

	public void zeichneMandelbrotmenge(Graphics g, int liRand, int obRand) {
		double xa = -2.02, xe = 0.7, ya = -1.2, ye = 1.2;  // Ratio 17:15
		double dx = (xe-xa)/(imageBreite-1), dy = (ye-ya)/(imageHoehe-1);
		int yHalbe = imageHoehe/2;
		double cx, cy;
		int maxIt = 500;
		cx = xa;
		g.setColor(Color.BLACK);
		long zeit = System.currentTimeMillis();
		for (int sp = 0; sp < imageBreite; sp++) {
			cy = ye; // von oben nach unten
			for (int ze = 0; ze < imageHoehe; ze++) {
				if (iterZahl(cx, cy, maxIt) == maxIt)
					g.drawLine(sp + liRand, ze + obRand, sp + liRand, ze + obRand);

				cy = cy - dy;
			} // for ze

			cy = 0; // cy = 0 separat behandeln !
			if (iterZahl(cx, cy, maxIt) == maxIt)
				g.drawLine(sp + liRand, yHalbe + obRand, sp + liRand, yHalbe + obRand);
			cx = cx + dx;
		} // for sp

		System.out.println("benötigte Zeit = " + (System.currentTimeMillis() - zeit) + " ms");
	}

}
```

Lässt man die Zeilen für cy = 0 weg, so sieht die Grafik so aus:

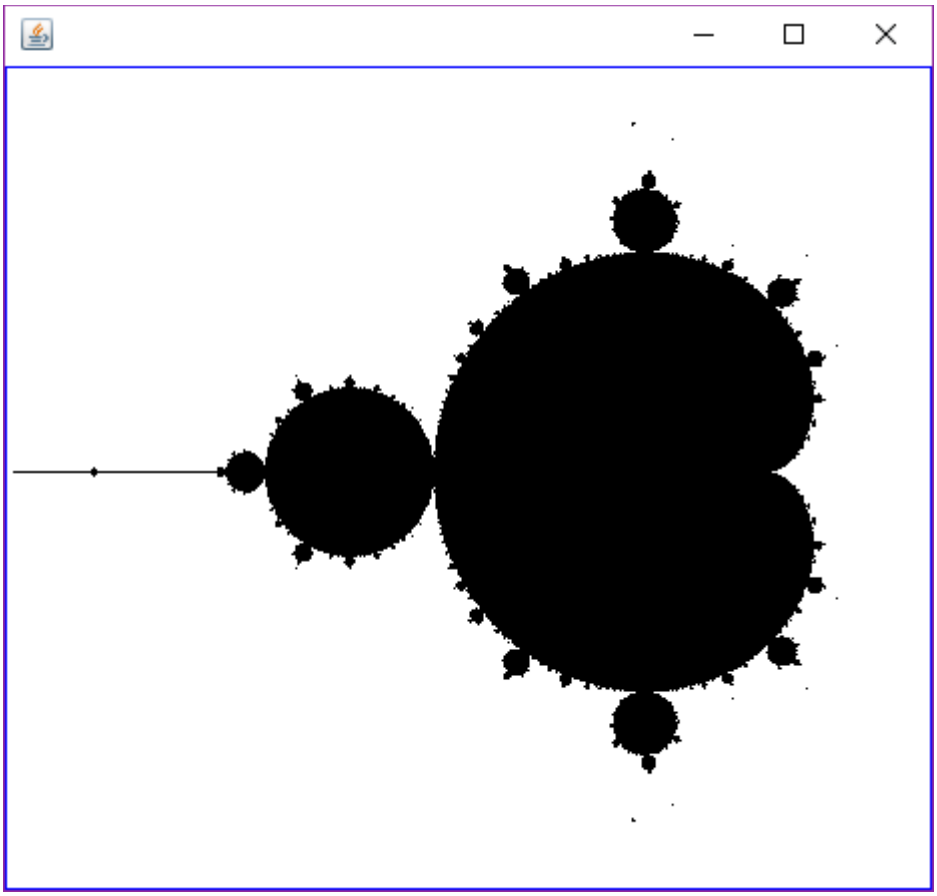
Offensichtlich ist die Linie für cy = 0.0 nicht richtig erfasst worden.

Dies liegt an der ungenauen Gleitpunktarithmetik von Java ( vgl. mit obigem Hinweis ).



Untersucht man die Linie für cy = 0.0 separat , so erhält man ein besseres Ergebnis.

Natürlich kann man auch den etwas langsameren Algorithmus verwenden, welcher Multiplikationen in den Schleifen verwendet !



**Anmerkung:** Die **Rechenzeit** für das Programm ist relativ lang; 2s auf einem Core i7-6700
Ein **JPanel** ist zur Darstellung der Grafik besser geeignet als ein JFrame !

Daher anschließend noch die **JPanel-Version**, die auch deutlich schneller ist.

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class JPanelMandelbrot extends JFrame {   // Ac 04-2018

	public static void main(String[] args) {
		new JPanelMandelbrot();
	}

	static int imageBreite = 459;
	static int imageHoehe = 405;
	int frameBreite = imageBreite + 30, frameHoehe = imageHoehe + 50;
	Leinwand malPanel = new Leinwand(imageBreite, imageHoehe);
	JPanel contentPane;

	public JPanelMandelbrot() {  // Konstruktor
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setSize(frameBreite, frameHoehe);
		setLocationRelativeTo(null);
		setTitle("Mandelbrotmenge mit JPanel");
		contentPane = new JPanel();
		setContentPane(contentPane);
		setVisible(true);
		malPanel.setPreferredSize(new Dimension(imageBreite, imageHoehe));
		contentPane.add(malPanel);
	}
}

class Leinwand extends JPanel {
   public Leinwand(int imageBreite, int imageHoehe) {
      setBackground(Color.WHITE);
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      long zeit = System.currentTimeMillis();
      Mandelbrot.zeichneMandelbrotmenge(g,JPanelMandelbrot.imageBreite,
                                                              JPanelMandelbrot.imageHoehe);
      System.out.println("benötigte Zeit = " + (System.currentTimeMillis() - zeit) + " ms");
   }
}

class Mandelbrot { // statische Klasse

	public static int iterZahl(final double cx, final double cy, int maxIt) {
		int zaehler = 0;
		double zx = 0.0, zy = 0.0, tmp;
		do {
		    tmp = zx*zx - zy*zy + cx;
		    zy = 2*zx*zy + cy;
		    zx = tmp;
		    zaehler = zaehler + 1;
		} while (zx*zx + zy*zy <= 4.0 && zaehler < maxIt);
		return zaehler;
	}

	public static void zeichneMandelbrotmenge(Graphics g, int imageBreite, int imageHoehe) {
		double xa = -2.02, xe = 0.7, ya = -1.2, ye = 1.2;  // Ratio 17:15
		final double dx = (xe-xa)/(imageBreite-1), dy = (ye-ya)/(imageHoehe-1);
		double cx, cy;
		int maxIt = 500;
		g.setColor(Color.BLACK);
		for (int sp = 0; sp < imageBreite; sp++) {
			cx = xa + sp * dx; // von links nach rechts
			for (int ze = 0; ze < imageHoehe; ze++) {
				cy = ye - ze * dy; // von oben nach unten
				if (iterZahl(cx, cy, maxIt) == maxIt)
					g.drawLine(sp, ze, sp, ze);
			}
		}
	}
}
```

## Optimierung des Algorithmus:

Das Erstellen der Bilder dauert in der Regel etwa eine Sekunde oder sogar weniger bei aktuellen Computern (Stand: 2018) . Dennoch können sich Optimierungen des Algorithmus lohnen, falls man

- die Iterationstiefe stark erhöhen möchte,
- Ausschnitte aus der Menge vergrößert darstellen will (deutlich erhöhte Rechenzeit !),
- die Mandelbrotmenge deutlich präziser darstellen will, also dx, dy verkleinert .
  Es zeigt sich nämlich, dass die Mandelbrotmenge wesentlich **stärker verästelt** ist, als es die üblichen Bilder suggerieren (vgl. Bilder weiter unten !) .

1) Liegt die Figur symmetrisch zur x-Achse, so kann man dies ausnutzen und die Rechenzeit halbieren.

2) Ersetzt man Quadrate wie zx*zx durch zxq , so lässt sich Rechenzeit einsparen:

```
zx = 0   zy = 0   zxq = 0   zyq = 0
zaehler=0
wiederhole   // iteriere
   zy = 2*zx*zy + cy
   zx = zxq - zyq + cx
   zxq = zx*zx
   zyq = zy*zy
   zaehler = zaehler + 1
bis (zxq + zyq > 4.0) oder (zaehler = maxIt)
```

3) Ein spürbare Verbesserung besteht darin, vor der Iteration einen sog. **Kreis-Kardioiden-Test** einzubauen, der überprüft, ob der zu untersuchende Punkt (cx ; cy) innerhalb eines Kreises bzw. einer Kardioide liegt und somit zur Mandelbrotmenge gehört.

Es lässt sich nämlich zeigen, dass "im Wesentlichen"

- der "Kopf" der Mandelbrotmenge ein **Kreis** mit M(-1;0) und r = 0,25 ist mit der Gleichung $(x+1)^2 + y^2 = 0{,}25^2$ . Der Kreis liegt im x-Bereich $-1{,}25 \le x \le -0{,}75$ .

P(cx ; cy) liegt im Kreis, wenn gilt: **$(cx+1)^2 + cy^2 \le 1/16$**

- das nierenförmige Hauptgebiet (der "Apfel") eine **Kardioide** (Herzkurve) ist mit der Parameterdarstellung:

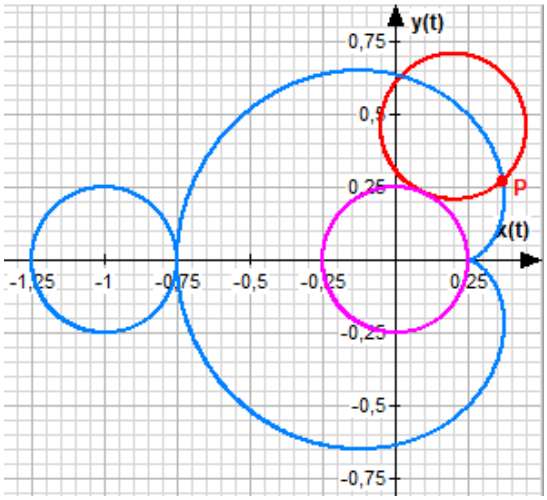
$x(t) = [2\cos(t) - \cos(2t)] / 4$
$y(t) = [2\sin(t) - \sin(2t)] / 4$ ; jeweils für $t \in [0 ; 2\pi]$

In algebraischer Form:
$[ x^2 + y^2 - (1/4)^2 ]^2 = 4 \cdot (1/4)^2 \cdot [ (x-1/4)^2 + y^2 ]$

P(cx ; cy) liegt innerhalb der Kardioide, wenn gilt:

**$16rs \le 5s - 4cx + 1$**
mit $r = cx^2 + cy^2$ und $s = \mathrm{sqrt}(r - cx/2 + 1/16)$

Der Rand dieser Kardioide ist diejenige Bahn, die ein Punkt P auf einem Rad mit Radius r = 0,25 beschreibt, welches auf einem weiteren Rad mit demselben Radius und dem Mittelpunkt M(0;0) abrollt.




Fällt die Überprüfung positiv aus, so muss die zeitraubende Iteration nicht mehr durchgeführt werden. Dies spart eine Menge an Rechenzeit ein !

```
// Algorithmus zur Ermittlung des zaehlers (mit Kreis-Kardioiden-Test)

   zx = 0.0   zy = 0.0
   zxq = 0.0   zyq = 0.0
   zaehler = 0
   cyq =  cy*cy
   falls cx > -0.75
      dann // Kardioidentest durchführen
         r = cx*cx + cyq
         s = wurzel(r - 0.5*cx + 0.0625)
         falls 16*r*s <= 5*s - 4*cx + 1  // Kardioidentest erfolgreich
            dann gib maxIt zurück
   sonst falls cx >= -1.25 und (cx+1)*(cx+1) + cyq <= 0.0625 // Kreistest erfolgreich
      dann gib maxIt zurück
   wiederhole // "iteriere" durchführen; Wert für zaehler ermitteln
                zy = 2*zx*zy + cy
                zx = zxq - zyq + cx
                zxq = zx*zx
                zyq = zy*zy
                zaehler = zaehler + 1
   bis (zxq + zyq > 4.0) oder (zaehler = maxIt)

   falls zaehler = maxIt dann gehört der Punkt zur Mandelbrotmenge
```

## Weitere Eigenschaften der Mandelbrotmenge:

Die Mandelbrotmenge ist zusammenhängend, es gibt also keine Inseln in der Menge.

Der **Flächeninhalt** der Mandelbrotmenge wurde auf **1,5065918849** geschätzt .

Methode:

Es wird ein hochaufgelöstes Bild der Mandelbrotmenge erzeugt und die Anzahl der Pixels gezählt.
"Hochaufgelöst" bedeutet hierbei, dass 1 000 000 x 1 000 000 oder mehr Pixels verwendet werden.

In den meisten Darstellungen wird das Apfelmännchen nur sehr ungenau dargestellt.
Daher erfolgt jetzt eine Darstellung mit ungleich **präziserer Betrachtung**, was natürlich sehr viel Rechenzeit kostet. Die Berechnung erfolgt hier auf "Intel Core i7-6700" .

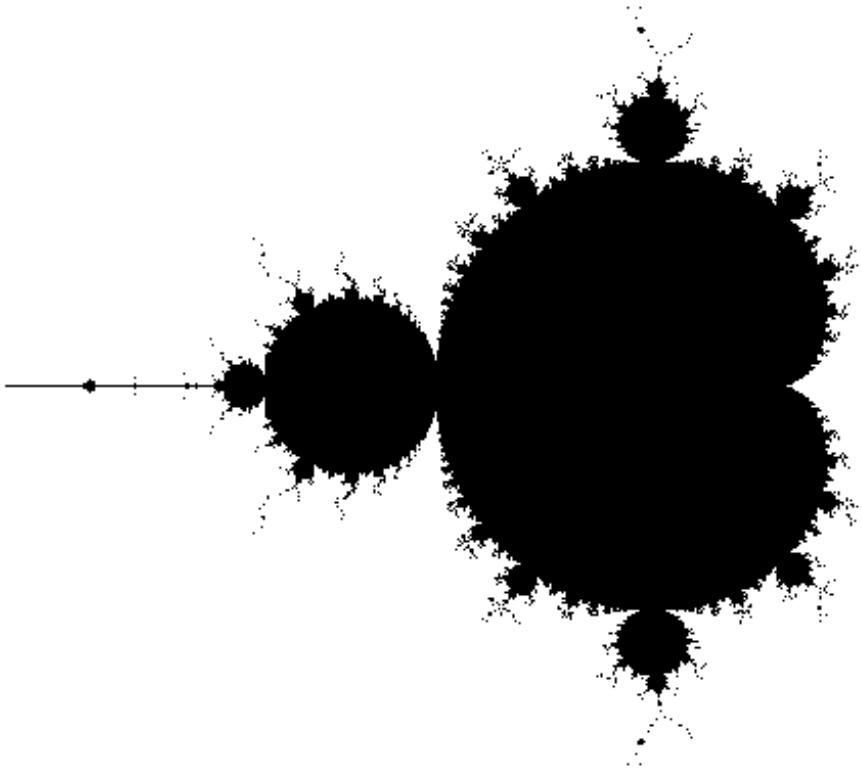
**Die Vorgehensweise:**

Die Auflösung wird (rechnerintern) in der Horizontalen sowie der Vertikalen um den Faktor k vervielfacht, insgesamt also rechnerintern eine $k^2$-fache Auflösung . Anschließend wird auf die Auflösung der Bilddarstellung (hier: 459 x 405) "heruntergerechnet" ! x-Bereich: [ -2,02 ; 0,7 ]

Auflösung:
$32^2$ - fach

Iterationstiefe:
10000

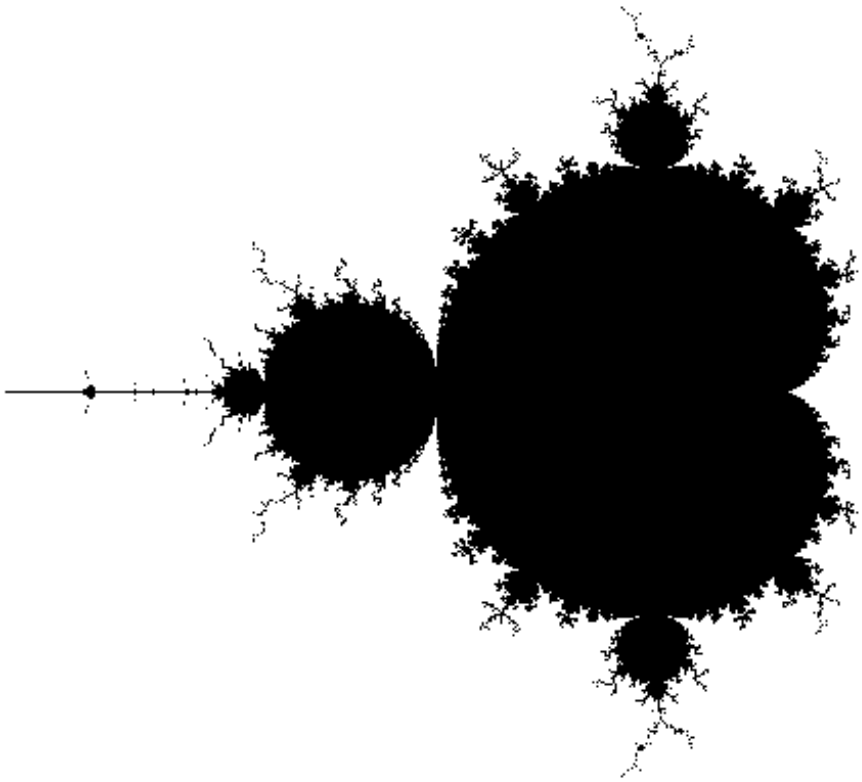
Rechenzeit :
70 s



Auflösung:
$128^2$ - fach

Iterationstiefe:
10000

Rechenzeit :
18 min 50 s



## Farbdarstellung der Mandelbrotmenge:

Untersucht man für divergente Folgen, wie nahe der Zähler der maximalen Iterationszahl maxIt gekommem ist und zeichnet für entsprechende Bereiche des ermittelten Zählers einen farbigen Punkt (cx ; cy) in die Gaußsche Zahlenebene, so erhält man eine interessante **farbige Darstellung** für das Apfelmännchen (s. Bild unten).

Anmerkung: Die farbigen Punkte gehören nicht zur Mandelbrotmenge, sondern sie markieren lediglich Bereiche mit mehr oder weniger starker Divergenz !

Die Zuordnung der farbigen Punkte gemäß der Anzahl der Iterationen (zaehler) kann z.B. folgendermaßen geschehen:

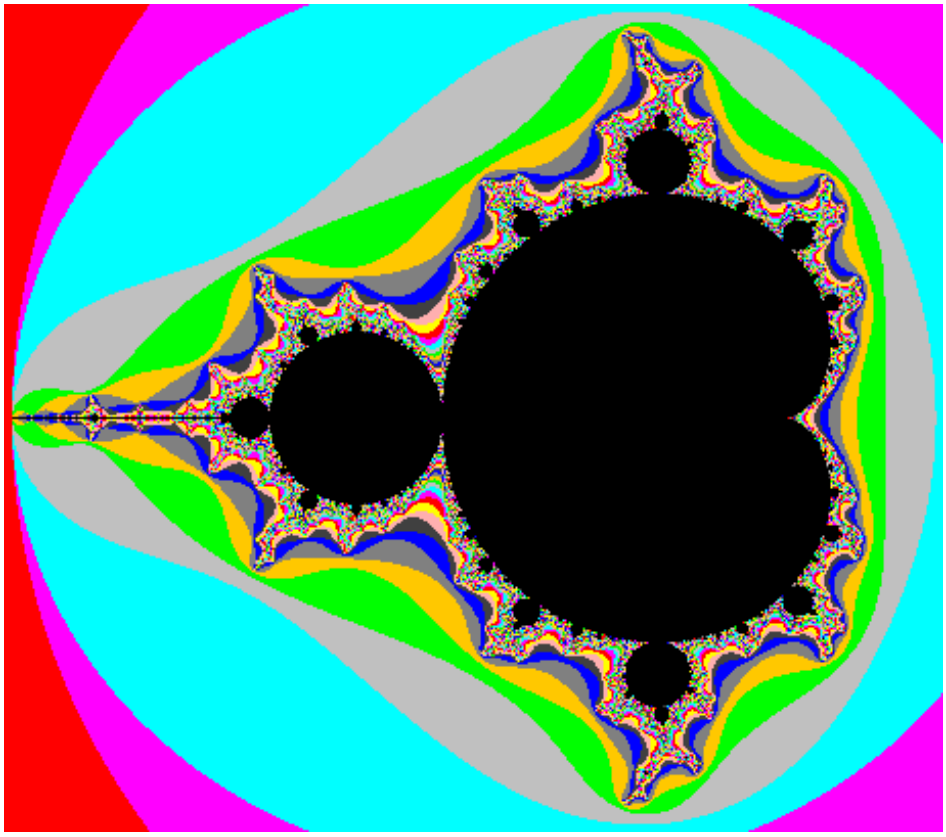
Man definiert ein Feld (array) von Farben , z.B.

```
final Color[] farbFeld = {
                Color.YELLOW,
                Color.RED,
                Color.MAGENTA,
                Color.CYAN,
                Color.LIGHT_GRAY,
                Color.GREEN,
                Color.ORANGE,
                Color.GRAY,
                Color.BLUE,
                Color.DARK_GRAY,
                Color.PINK
};
```

und weist die Farben über ein Modulo-Rechnung ( "%" in Java ) zu:

```
if (zaehler == maxIt)
   zeichnePixel(sp, ze, pixFarbe);
else
   zeichnePixel(sp, ze, farbFeld[zaehler % farbFeld.length]);
```

Farbige Darstellung mit so genannten " Höhenlinien " :



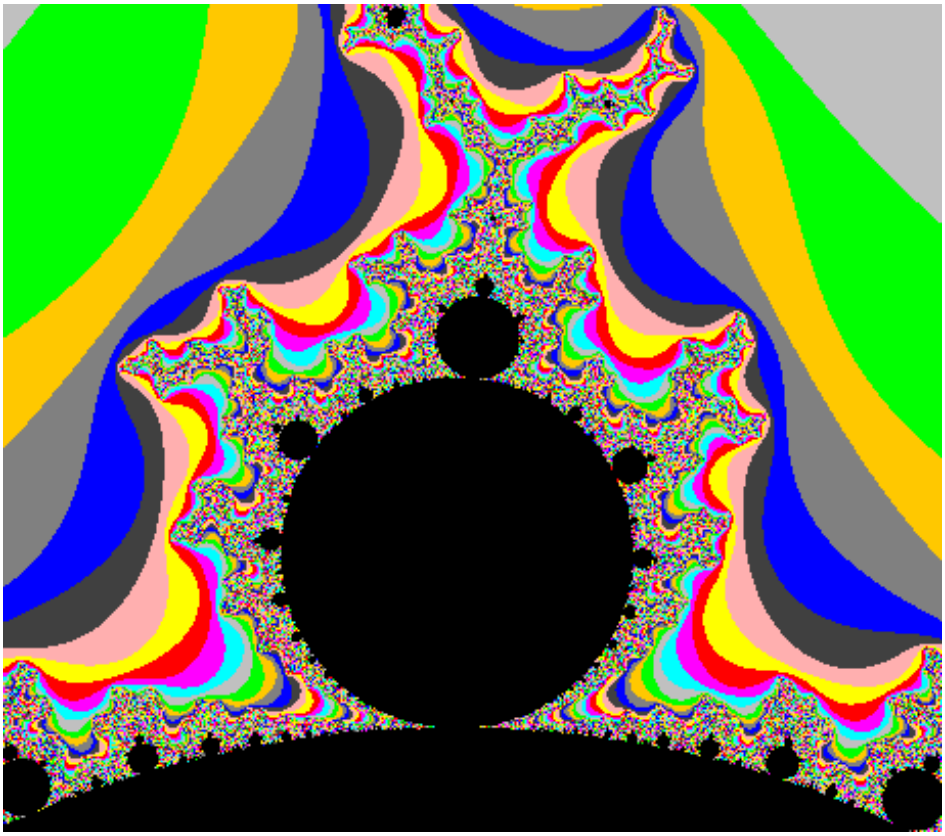
## Weitere Betrachtungen:

1) Ausschnittsvergrößerungen zeigen die Selbstähnlichkeit der Mandelbrotmenge .

Die Selbstähnlichkeit der Mandelbrotmenge bei Ausschnitts-vergrößerungen

x: [ -0,368996 ; 0,135808]
y: [ 0,594059 ; 1,039474]

Iterationstiefe = 1000



2) Setzt man den Startwert von z ungleich Null, so ergeben sich interessante Verformungen.

# Juliamengen

Gibt man die komplexe Zahl c konstant vor und betrachtet jetzt für jede komplexe Zahl z der Gaußschen Zahlenebene die Folge $\{z_n\}$ gemäß der Vorschrift $z_{n+1} = z_n^2 + c$ , so entstehen Bilder, die man als **Juliamengen** (benannt nach **Gaston Maurice Julia**) bezeichnet !

- liegt der Punkt (cx; cy) innerhalb des Apfelmännchens, so ist die Juliamenge zusammenhängend
- liegt (cx; cy) außerhalb des Apfelmännchens, so ist die Juliamenge unzusammenhängend
- interessant sind Punkte (cx; cy) auf dem Rand des Apfelmännchens

Da die Startwerte z(zx ; zy) in der Iterationsschleife gemäß der Gleichung $z = z^2 + c$ jedesmal wieder verändert werden, muss der Algorithmus "Mandelbrotmenge" so angepasst werden, dass zu Beginn jeder Schleife für die Spalten und die Zeilen zx und zy die jeweiligen Koordinaten des betreffenden Punktes der Gaußschen Zahlenebene erhalten.
Wir müssen also vor jeder Iterationsschleife setzen: zx = xa + sp * dx sowie zy = ye - ze * dy
Dies wird im folgenden Java-Programm berücksichtigt.

```
public void zeichneJuliamenge(Graphics g, int liRand, int obRand) {
      double xa = -1.6, xe = 1.6, ya = -1.43, ye = 1.43; // Werte variieren !
      double dx = (xe-xa)/(imageBreite-1), dy = (ye-ya)/(imageHoehe-1);
      double zx, zy, tmp;
      double  cx = -0.7, cy = 0.1; // Werte variieren !
      int maxIt = 200; // Wert variieren !
      int zaehler;
      g.setColor(Color.BLACK);
      for (int sp = 0; sp < imageBreite; sp++) {
          for (int ze = 0; ze < imageHoehe; ze++) {
              zx = xa + sp * dx; // von links nach rechts
              zy = ye - ze * dy; // von oben nach unten
              zaehler = 0;
              do {
                 tmp = zx*zx - zy*zy + cx;
                 zy = 2*zx*zy + cy;
                 zx = tmp;
                 zaehler = zaehler + 1;
              } while (zx*zx + zy*zy <= 4.0 && zaehler < maxIt);
              if (zaehler == maxIt)
                 g.drawLine(sp + liRand, ze + obRand, sp + liRand, ze + obRand);
            } // for ze
      } // for Sp
}
```

**Juliamenge** zu

cx = -0,7
cy = 0,05

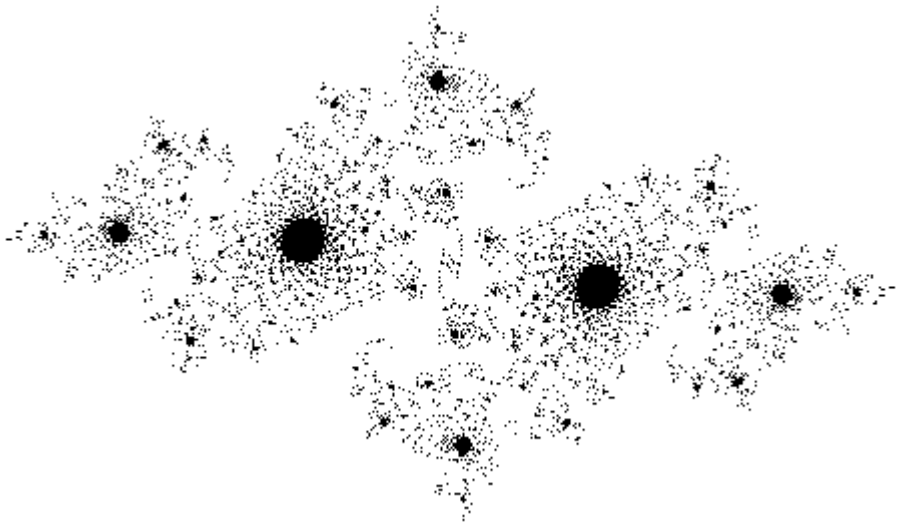
```
xa = -1,6
xe = 1,6

ya = -1,43
ye = 1,43

maxIt = 200
```

Beispiel aus dem obigen Java-Programm

**Juliamenge** zu

cx = -0,73967
cy = 0,15778

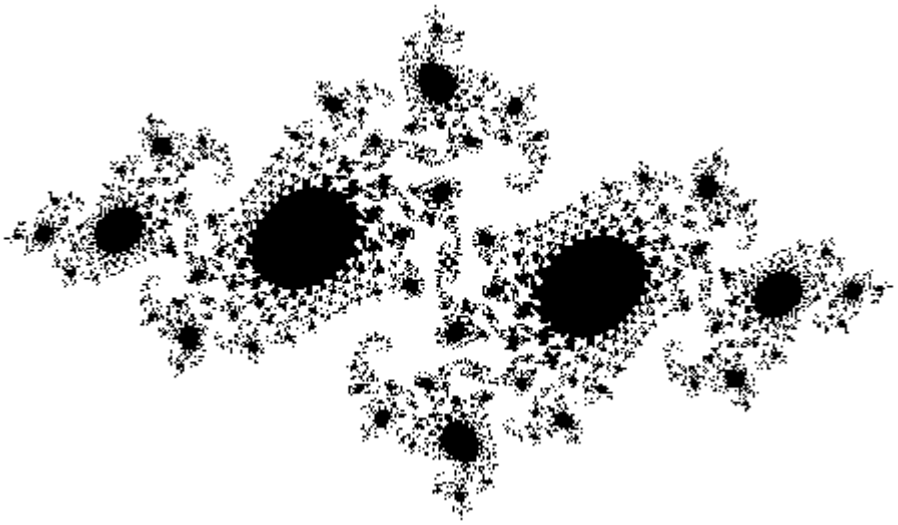
```
xa = -1,6
xe = 1,6

ya = -1,0
ye = 1,0

maxIt = 200
```



**Juliamenge** zu

cx = -0,73967
cy = 0,15778

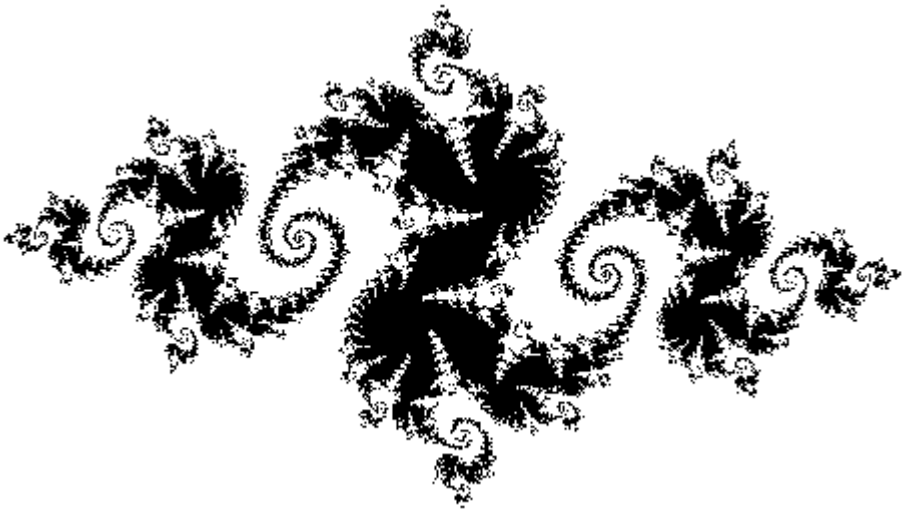
```
xa = -1,6
xe = 1,6

ya = -1,0
ye = 1,0

maxIt = 100

Beispiel wie
oben, nur
kleineres
maxIt !
```



**Juliamenge** zu

cx = -0,777
cy = 0,116

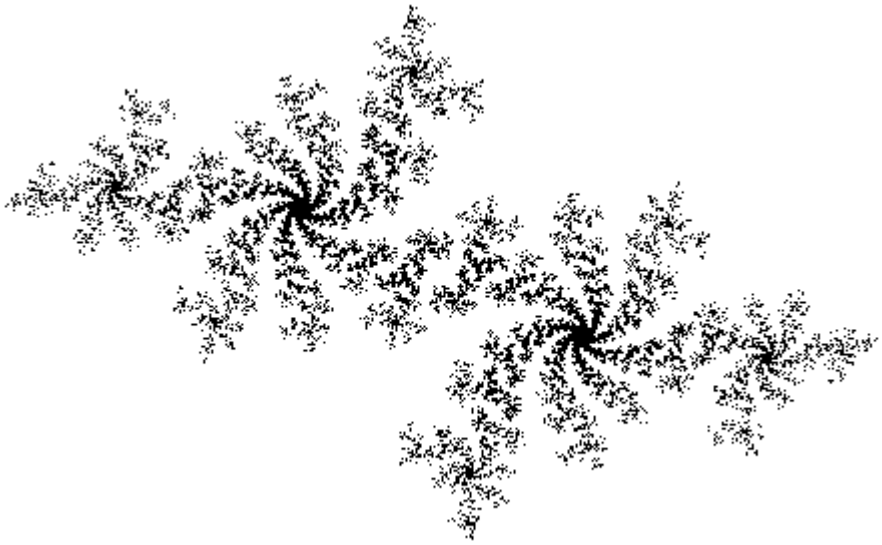
```
xa = -1,6
xe = 1,6

ya = -1,0
ye = 1,0

maxIt = 200
```



**Juliamenge** zu

cx = -0,65175
cy = 0,4185

```
xa = -1,6
xe = 1,6

ya = -1,0
ye = 1,0

maxIt = 70
```



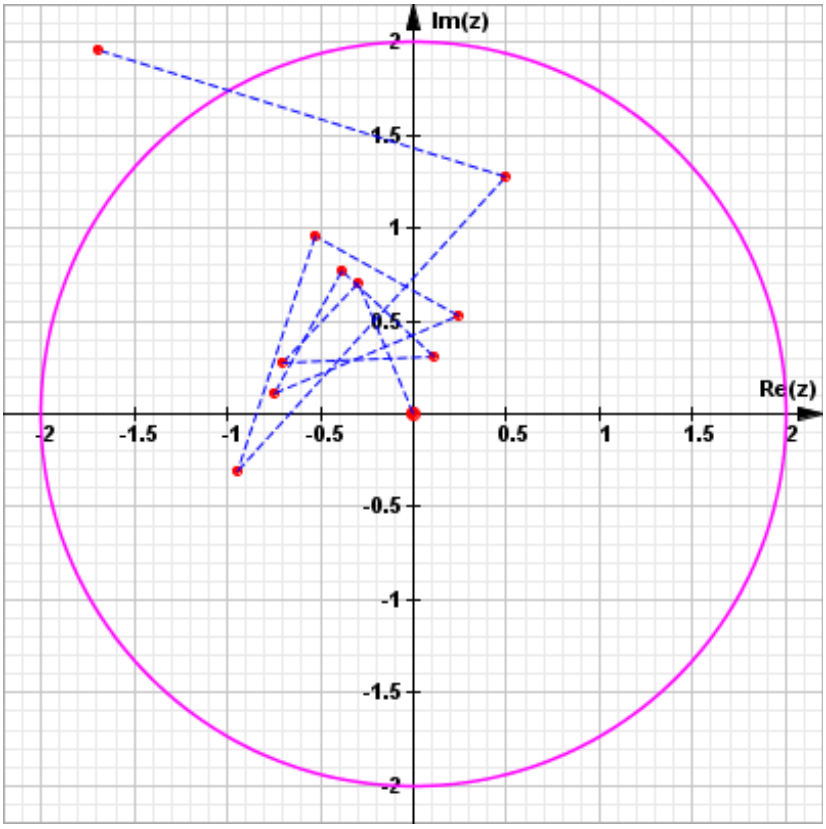
Anhang 1: **Mandelbrot-Orbits**

Grafische Darstellung der Iteration $z = z^2 + c$ mit vorgegebenem c = (cx ; cy)

Beispiel: cx = -0,3 cy = 0,7

| zx | zy |
|---|---|
| 0 | 0 |
| -0,300 | 0,700 |
| -0,700 | 0,280 |
| 0,112 | 0,308 |
| -0,382 | 0,769 |
| -0,745 | 0,112 |
| 0,242 | 0,533 |
| -0,526 | 0,958 |
| -0,942 | -0,307 |
| 0,492 | 1,279 |
| -1,692 | 1,959 |
| ... | ... |

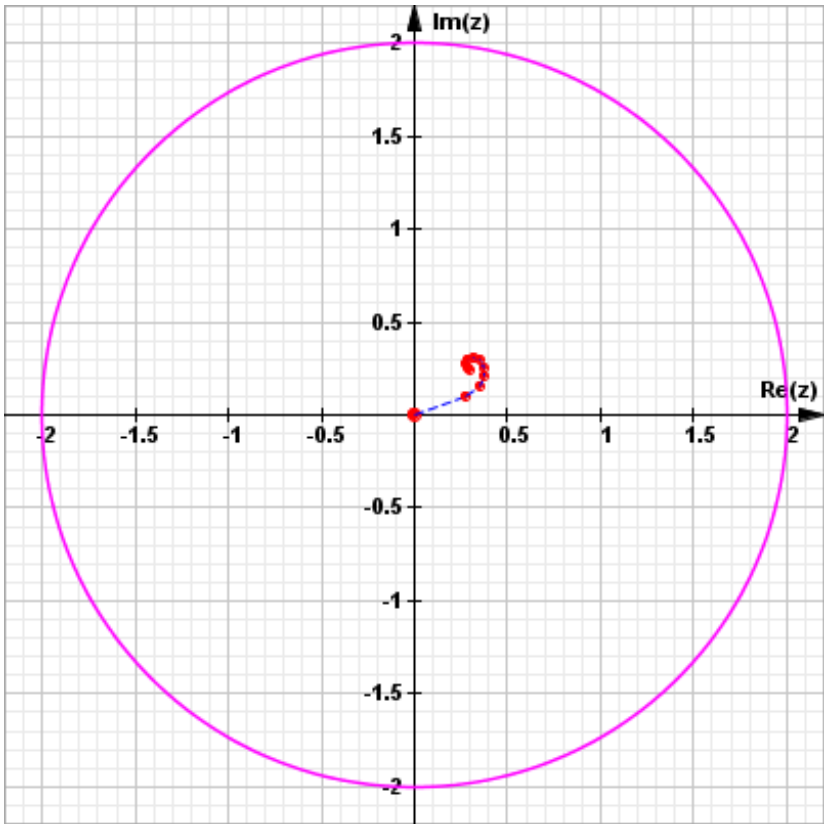
Kreis mit Radius 2 verlassen.
Offensichtlich Divergenz !




Beispiel: cx = 0,28 cy = 0,1

| zx | zy |
|---|---|
| 0 | 0 |
| 0,280 | 0,100 |
| 0,348 | 0,156 |
| 0,377 | 0,209 |
| 0,379 | 0,257 |
| 0,357 | 0,295 |
| 0,321 | 0,311 |
| 0,286 | 0,299 |
| 0,272 | 0,271 |
| 0,281 | 0,248 |
| 0,297 | 0,239 |
| ... | ... |

Konvergenz !



## Anhang 2: Programmierung mit TI83/84-BASIC:

Da das TI-Basic nur einen Buchstaben pro Variable zulässt, müssen erst (willkürliche) Zuweisungen der Variablen zu den Buchstaben A,B, ... erfolgen:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | **xa** | **B** | **xe** | **C** | **ya** | **D** | **ye** | **E** | **dx** | **X** | **cx** |
| **L** | **zx** | **M** | **zy** | **Q** | **zxq** | **P** | **zyq** | **F** | **dy** | **Y** | **cy** |
| **N** | **maxIt** | **Z** | **zaehler** | **I** | **Sp** | **K** | **Ze** | | | | |

Der TI hat 95 Pixels in der Waagerechten und 63 Pixels in der Senkrechten. Also sind die Pixelabstände: 94( waagerecht) 62 (senkrecht). Die Nummerierung beginnt jeweils bei 0.

Will man die Rechenzeit nicht unnötig verlängern, so sollte man die Symmetrie zur x-Achse nutzen und möglichst mit ganzen Zahlen iterieren (dies sind die Pixelkoordinaten von 0 bis 94 sowie von 0 bis 62). Wir rechnen dann von cy = 0 bis cy = ye. Die Ergebnisse für cy **∈ [ya;0]** erhalten wir automatisch wegen der x-Achsen-Symmetrie. Ferner ist eine Gleichsetzung von dx und dy nötig, um eine verzerrungsfreie Darstellung zu erhalten.
ya und ye werden dann ebenfalls automatisch berechnet, d.h. sie müssen nicht eingetippt werden. Die folgende rechnerische Umformung liefert die Zusammenhänge:

(xe-xa)/(ye-ya) = 94/62 . Mit ya = -ye folgt:
(xe-xa)/(2ye) = 94/62 und somit (xe-xa)/94 = ye/31

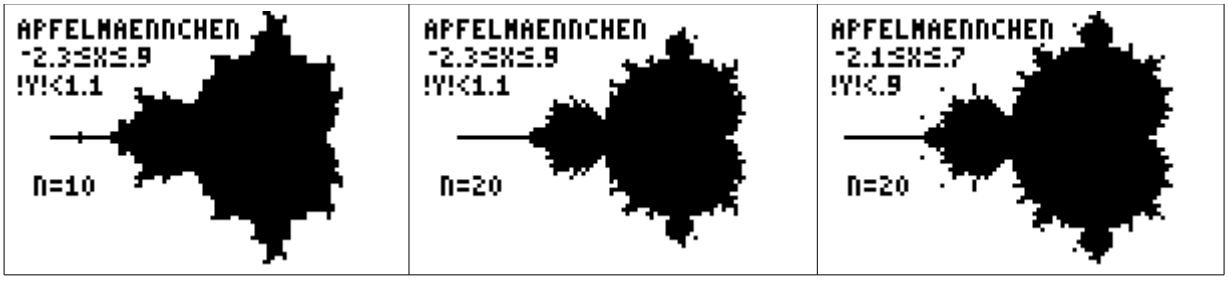
*Wir erhalten dann ( eingegeben wurden xa und xe ):*
***dx = dy = (xe-xa)/94 ye = 31*(xe-xa)/94 = 31*dx ya = -ye***

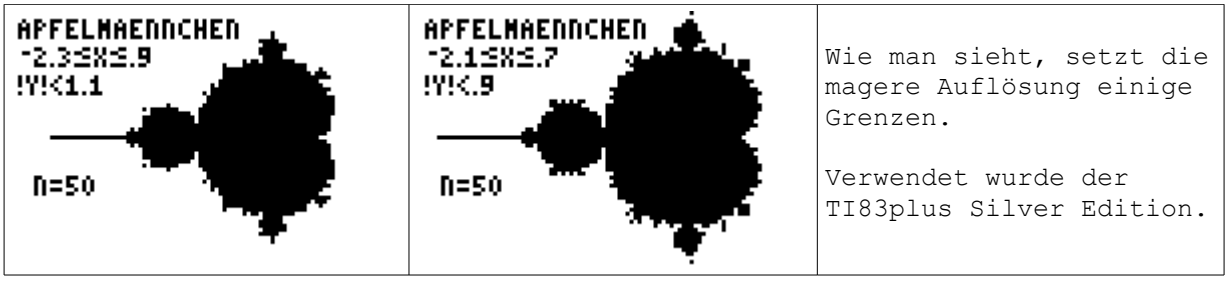
Wegen dx = dy kann man auf F verzichten, denn es ist F = E.
Ebenso kann man wegen ya = -ye auf C verzichten, denn C = -D.

Das TI-Basic-Programm sieht dann so aus:

```
Input „XA=",A : Input „XE=",B
Input"MAXIT=",N
(B-A)/94 STO E
31E STO D
FnOff:AxesOff:ClrDraw:DispGraph
Text(1,1,"APFELMAENNCHEN")
Text(8,1,A,"≤X≤",B)
Text(15,1,!Y!<round(D,1))
Text(40,5,"N=",N)
A STO X
For(I,0,94)
0 STO Y
For(K,0,31)
0 STO L:0 STO M:0 STO Z:0 STO Q:0 STO P
Repeat (Q+P>4) or (Z=N)
2LM+Y STO M
Q-P+X STO L
L² STO Q
M² STO P
Z+1 STO Z
End
If (Z=N)
Then
Pxl-On(31+K,I)
Pxl-On(31-K,I)
End
Y+E STO Y
End
X+E STO X
End
```

**Hier einige Beispielgraphen:** Anmerkung: !Y! bedeutet: „Betrag von Y"




Wie man sieht, setzt die magere Auflösung einige Grenzen.

Verwendet wurde der TI83plus Silver Edition.