Apfelmännchen - Theorie und Programmierung

Das Thema "Apfelmännchen" gehört zum Oberthema

"Chaos und Ordnung in dynamischen Systemen" .

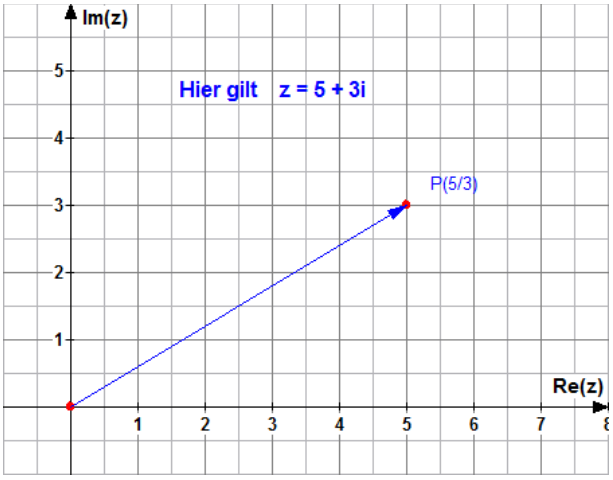
Es ist ein relativ neues Forschungsgebiete der Mathematik (ab ca. 1980).

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*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 Gaußsche Zahlenebene

Beispiel z = 5 + 3i



Für jede komplexe Zahl c=a+ib 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, 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.).

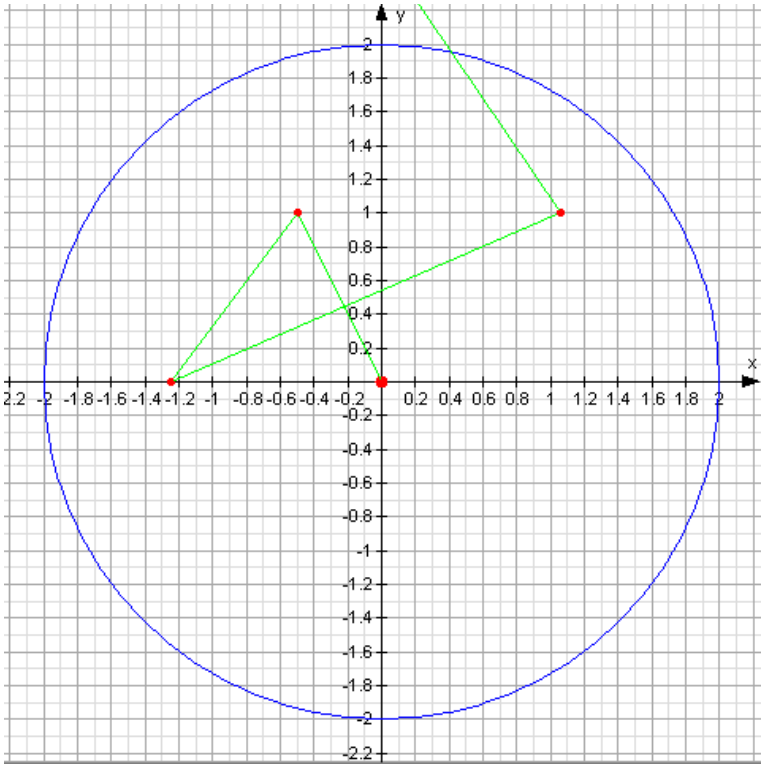
Da Konvergenz mit Hilfe des Computers schlecht feststellbar ist, begnügen wir uns mit folgender Forderung:
Die Folge sei konvergent, wenn ihr Betrag |z| (das ist die Strecke vom Ursprung (0;0) bis zur Zahl z = x+i*y := (zx,zy)) nach einer festen Zahl von Iterations-schritten (z.B. n=500 Schritte) die oben angegebene endliche Schranke 2 nicht überschreitet.
(Dass 2 als Schranke ausreicht lässt sich übrigens mathematisch beweisen.)
Anm.: Der Betrag berechnet sich zu $\sqrt{(zx^2+zy^2)}$

Rechenbeispiel für c = -0,5 + 1,0 * 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.

Grafische Darstellung
der obigen Iteration:



Der vorläufige Algorithmus für die Iteration der Folge:
Vorgegeben ist die Iterationstiefe maxIt (z.B. 500)
z(0):=0 k:=0
wiederhole
  z(k+1):=z(k)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 zx:=Re(z), zy:=Im(z), cx:=Re(c), cy:=Im(z) , so erhält man:
$z^2 = (zx+i*zy)^2 = zx^2-zy^2+i*2*zx*zy$
$z(k+1) = zx^2-zy^2+i*2*zx*zy+cx+i*cy$, also $z(k+1) = zx^2-zy^2+cx + i*(2*zx*zy+cy)$

Außerdem gilt in der Menge der komplexen Zahlen:
$|z|=\sqrt{(zx^2+zy^2)}$, d.h. $|z|^2 = zx^2+zy^2$

Anmerkung:
Zur Darstellung der Mandelbrotmenge genügt als Ausschnitt aus der Gaußschen Zahlenebene z.B. **-2,05<=x<=0,75 ; -1,25<=y<=1,25**
Als Auflösung wählen wir **476 (x-Bereich) mal 425 (y-Bereich)**, womit wir eine verzerrungsfreie Darstellung erhalten (Verhältnis 112:100).
Für jeden Punkt (cx,cy) aus dem oben gewählten Ausschnitt wird dann iteriert.

Algorithmus (Pseudocode):

```
ImageBreite=476   ImageHoehe=425    maxIt=500
xa=-2.05   xe=0.75   ya=-1.25   ye=1.25
dx=(xe-xa)/ImageBreite   dy=(ye-ya)/ImageHoehe
cx=xa  // von links nach rechts abarbeiten
für Sp=0 bis ImageBreite wiederhole
   cy=ye  // Beginn am oberen Rand
   für z=0 bis  ImageHoehe wiederhole
      zx=0  zy=0  zaehler=0
      wiederhole   // iteriere
         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)
      falls zaehler = maxIt dann setzePunkt(Sp,Ze)
      cy=cy-dy
   ende  // für Ze
   cx=cx+dx
ende  // für Sp
```

Erweiterungen und Hinweise:

1) Obiger Algorithmus ist insofern ungünstig, als er in der Regel den Wert cy=0 (x-Achse) übergeht ! Dies liegt an der Ungenauigkeit der Gleitpunktarithmetik von Programmiersprachen. Addiert man nämlich dy fortlaufend zu cy = -1.2 , so ergibt sich nach 200 Iterationen nicht gleich 0, sondern z.B. 1.0390993621101074E-15 (JAVA-Compiler).
Da aber cy=0 im Bereich -2<=cx<=0 sehr wohl zur Mandelbrotmenge gehört (dies ist rechnerisch sehr leicht zu zeigen), muss man bei der Iteration mit cy=0 beginnen und sich dann nach oben bzw. unten „vorarbeiten".
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 .

Modifizierter Algorithmus:

```
cy=0   für Ze=y0Pkt bis ImageHoehe wiederhole  ...  cy=cy-dy
cy=0   für Ze=y0Pkt ab bis 0 wiederhole  ....    cy=cy+dy
```

2) Das Erstellen der Bilder dauert in der Regel mehrere Sekunden bei aktuellen Computern (ca. 2012), so dass sich **Optimierungen** des Algorithmus lohnen. Beispielsweise lassen sich die oben eingefärbten Iterationsanweisungen ersetzen durch eine optimierte Fassung (zxq, zyq sparen Rechnungen ein):

```
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)
```

Ein weitere Verbesserung besteht darin, vor der Iteration einen sog. Kreis-Zykloiden-Test einzubauen:

```
zx=0.0  zy:=0.0 zxq=0.0  zyq=0.0   zaehler=0
r=cx*cx+cy*cy    s=wurzel(r-0.5*cx+0.0625)
falls (16*r*s > 5*s-4*cx+1) und ((cx+1)*(cx+1)+cy*cy > 0.0625)
   dann wiederhole
      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)
   sonst zaehler=maxIt
 falls zaehler=maxIt dann .....
```

3) Untersucht man für divergente Folgen, wie nahe der Zähler der maximalen Iterationszahl maxIt gekommem ist und zeichnet für entsprechende Bereiche des ermittelten Zählers (z.B. [maxIt-10;maxIt-1], [maxIt-20;maxIt-11] etc.) einen farbigen Punkt (cx;cy) in die Gaußsche Zahlenebene, so erhält man eine interessante farbige Darstellung für das Apfelmännchen.

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

5) Die weiter oben erwähnte Selbstähnlichkeit zeigt sich, wenn man Bereiche aus der Mandelbrotmenge vergrößert. Man erhält dann zur Originalfigur ähnliche Figuren. Dies wird ganz unten anhand der Bilder gezeigt.

6) 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** bezeichnet !

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

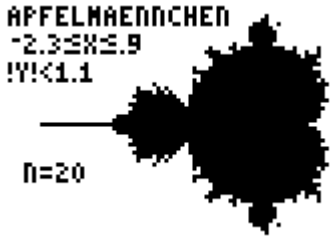
Wegen dx = dy kann man auf F verzichten, denn es ist F = E.
Ebenso kann man wegen ya = -ye auf C verzichten, denn C = -D.

Das TI-Basic-Programm sieht dann so aus:

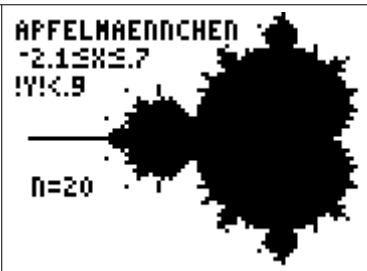
```
Input „XA=",A : Input „XE=",B
Input"MAXIT=",N
(B-A)/94 STO E
31E STO D
FnOff:AxesOff:ClrDraw:DispGraph
Text(1,1,"APFELMAENNCHEN")
Text(8,1,A,"≤X≤",B)
Text(15,1,!Y!<round(D,1))
Text(40,5,"N=",N)
A STO X
For(I,0,94)
0 STO Y
For(K,0,31)
0 STO L:0 STO M:0 STO Z:0 STO Q:0 STO P
Repeat (Q+P>4) or (Z=N)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

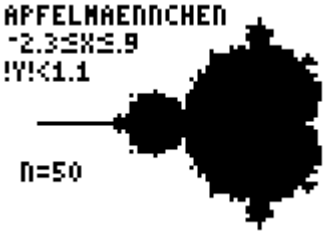
ca. 10 min



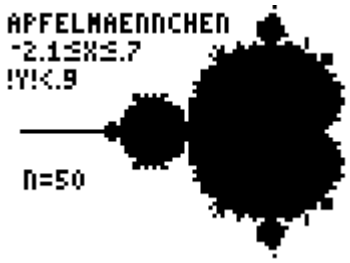
ca. 15 min



ca. 18 min



ca. 28 min



ca. 33 min

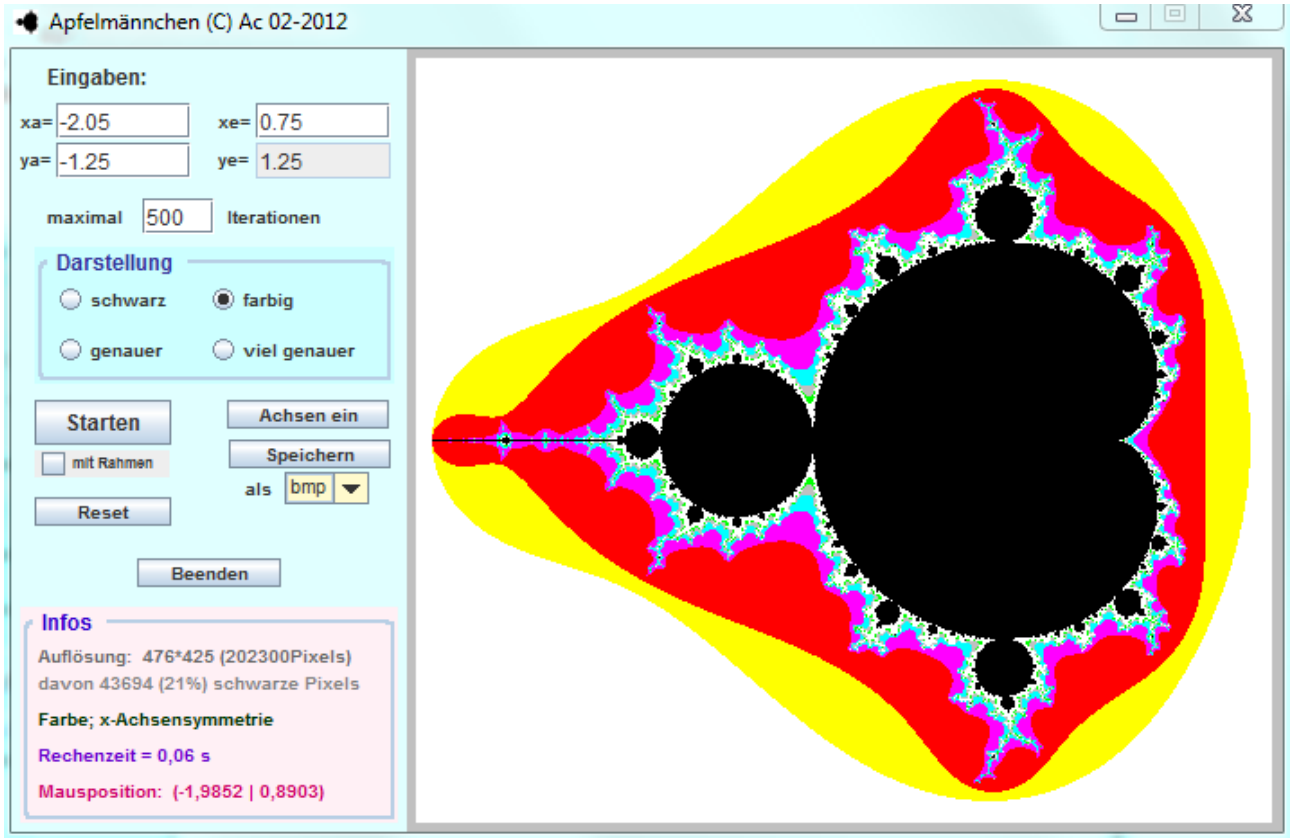
Wie man sieht, setzt die magere Auflösung einige Grenzen.

Verwendet wurde der TI83plus Silver Edition.

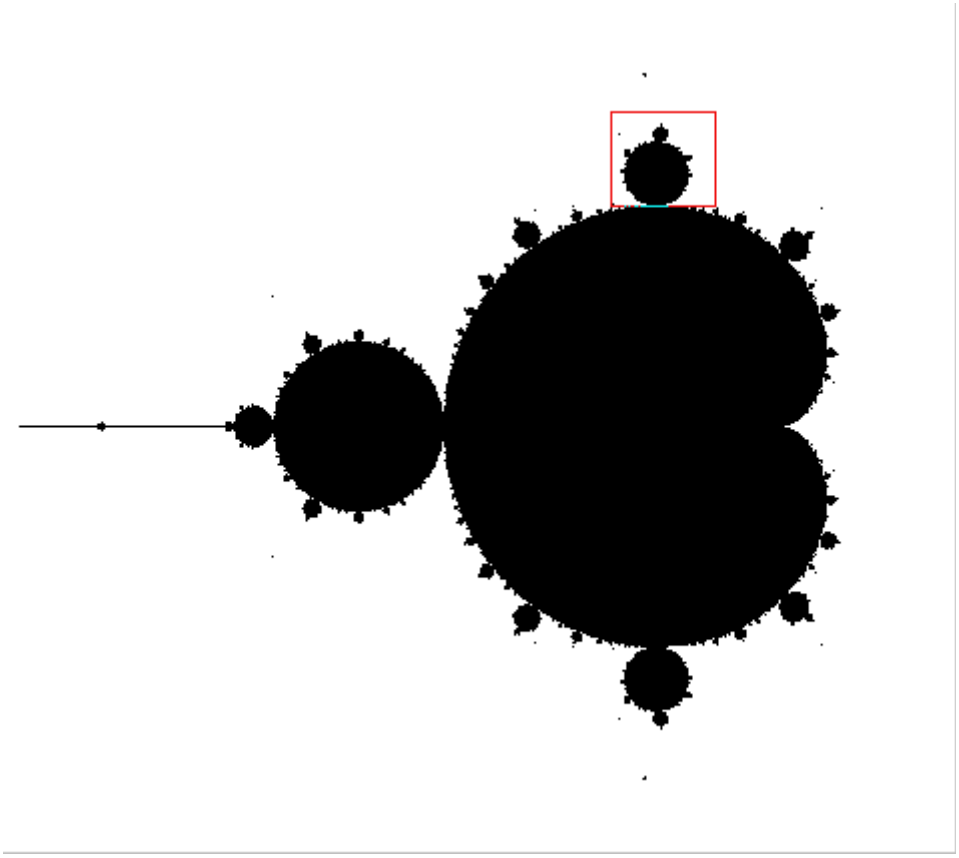
Programmierung mit JAVA (auf einem Panel) / Programmauszug:

```
public int iter(double cx, double cy) {
   // Iteration mit Kreis- und Zykloidentest
   int zaehler = 0;
   double zx = 0.0, zy = 0.0, zxq = 0.0, zyq = 0.0, r, s;
   r = cx * cx + cy * cy;     s = Math.sqrt(r - 0.5 * cx + 0.0625);
   if ((16.0 * r * s > 5.0 * s - 4.0 * cx + 1)&&
                           ((cx + 1.0) * (cx + 1.0) + cy * cy > 0.0625))
      do {
         zy = 2.0 * zx * zy + cy;    zx = zxq - zyq + cx;
         zxq = zx * zx;  zyq = zy * zy;
         zaehler++;
      } while ((zxq + zyq <= 4.0) && (zaehler < maxIt));
      else zaehler = maxIt;
   return zaehler;
}


public void zeichne(Graphics g) { // Apfelmännchen plotten
   // g ist die jeweilige Grafikumgebung von panelPlot oder bufImg
   int imageBreite = maxX, imageHoehe = maxY;
   int sp, ze, zaehler;
   int yHalbe = imageHoehe / 2, y0Pkt = (int) (ye / (ye - ya) * imageHoehe);
   double cx, cy;
   double dx = (xe - xa) / imageBreite, dy = (ye - ya) / imageHoehe;
                  . . .

   g.setColor(Color.black); // nur Mandelbrotmenge
   for (sp = 0; sp <= imageBreite; sp++) {
      cy = 0.0; // auf der x-Achse
      for (ze = yHalbe; ze <= imageHoehe; ze++) {
         if (iter(cx, cy) == maxIt) {
            g.drawLine(sp, ze, sp, ze);
            g.drawLine(sp, imageHoehe - ze, sp, imageHoehe - ze);
         }
         cy = cy - dy;
      } // for ze
      cx = cx + dx;
   } // for Sp
```

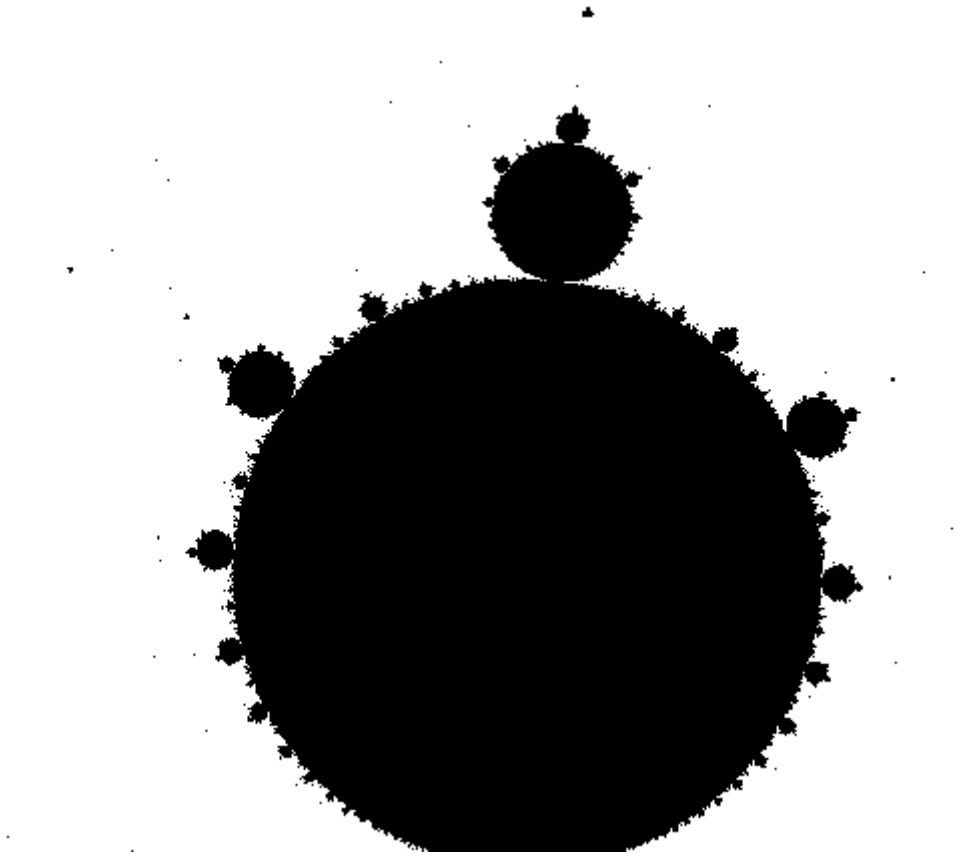
JAVA-Screenshot:



Apfelmännchen „pur" mit JAVA7:

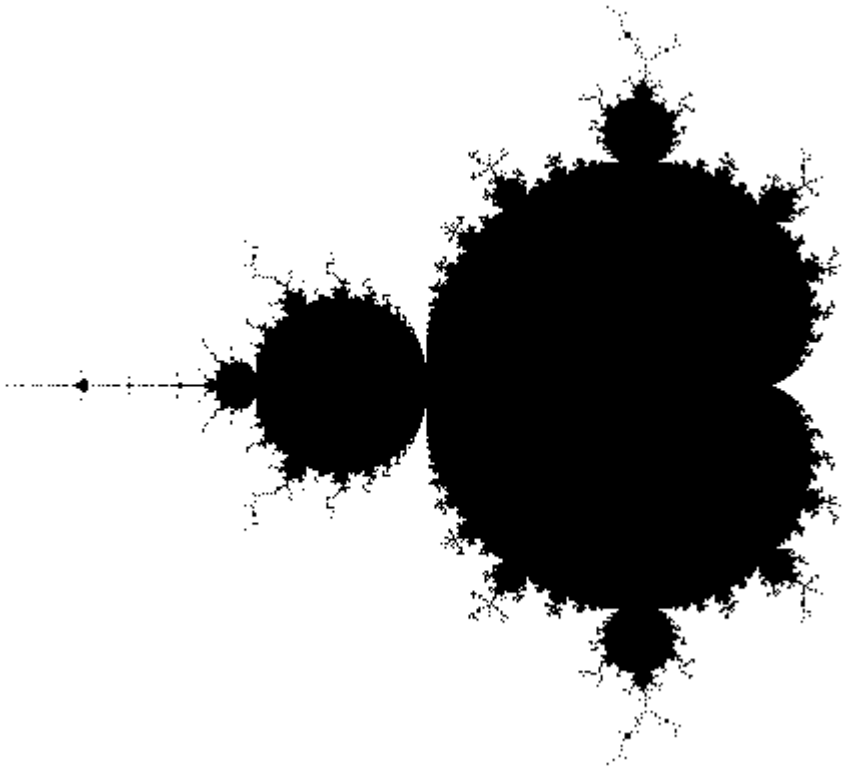


Der Ausschnitt mit x∈[-0,29;0.01], y∈[0,65;0,93] zeigt deutlich die Selbstähnlichkeit:



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:

Erweiterte Version mit größerer Genauigkeit (65^2-fache Rechnung) JAVA7: 44 sec



Erweiterte Version mit größerer Genauigkeit (513^2-fache Rechnung) JAVA7: 2670 sec