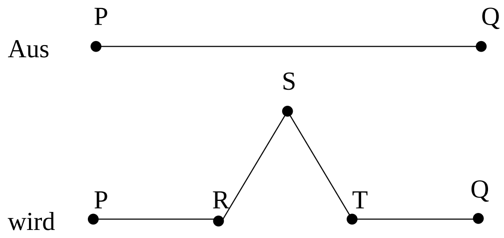
Kapitel 8

Fraktale

Die Erzeugung von Fraktal-Kurven erfolgt durch wiederholte Anwendung einer Transformation f auf eine Menge von Punkten. Charakteristisch für Fraktale ist ihre Selbstähnlichkeit (vgl. Stein im Gebirge, Blumenkohl, Broccoli).

8.1 Koch'sche Schneeflocke

Gegeben ein Polygon. Transformiere jede Polygonkante $\overline{PQ}$ zu einer Folge von Kanten $\overline{PR}$, $\overline{RS}$, $\overline{ST}$, $\overline{TQ}$:

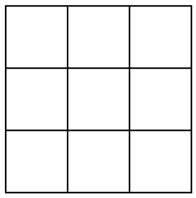


d.h., R und T dritteln die Kante $\overline{PQ}$, S ist eine 60°-Drehung des Knotens T um das Zentrum R gegen den Uhrzeigersinn.
Das so erhaltene Polygon kann erneut transformiert werden.

8.2 Fraktale Dimension

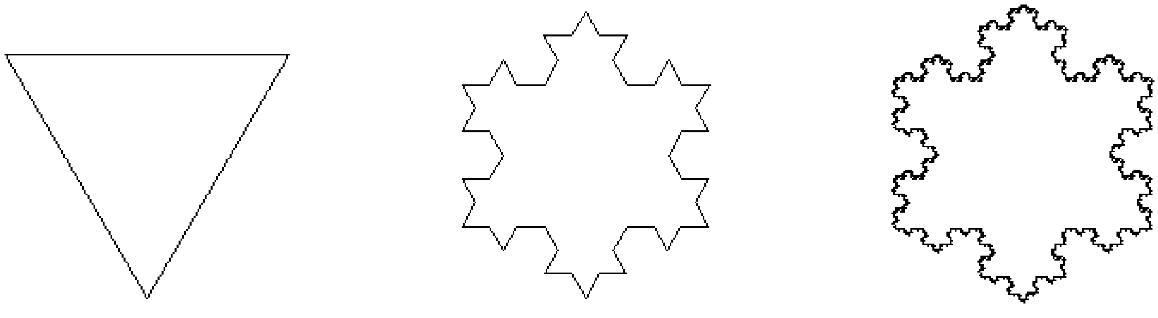
Ein selbstähnliches Objekt hat *Dimension* D, falls es in N identische Kopien unterteilt werden kann, die jeweils skaliert sind mit dem Faktor $r = \dfrac{1}{N^{\frac{1}{D}}}$.

Beispiel: Linie hat Dimension 1, denn sie besteht aus N Stücken der Größe $\frac{1}{N}$.
Quadratische Fläche hat Dimension 2, denn sie besteht aus N Flächen der Größe $\dfrac{1}{N^{\frac{1}{2}}}$,
z.B. 9 Teilflächen mit Kantenlängen skaliert um $r = \dfrac{1}{9^{\frac{1}{2}}} = \dfrac{1}{3}$



Sind N und r bekannt, läßt sich D bestimmen:
$r = \dfrac{1}{N^{\frac{1}{D}}} \Rightarrow r \cdot N^{\frac{1}{D}} = 1 \Rightarrow N^{\frac{1}{D}} = \dfrac{1}{r} \Rightarrow \dfrac{1}{D} \cdot \log(N) = \log(\dfrac{1}{r})$
$\Rightarrow D = \dfrac{\log(N)}{\log(\frac{1}{r})}$

Beispiel: Die Koch'sche Schneeflocke hat Dimension $D = \frac{\log(4)}{\log(3)} = 1.2618$, denn jeder Kantenzug besteht aus $N = 4$ Kopien, jeweils skaliert um den Faktor $r = \frac{1}{3}$.



Koch'sche Schneeflocken mit 3, 48 und 3072 Polygonpunkten

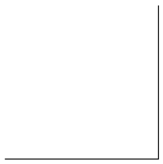
```
/****************************************************************************/
/*                                                                          */
/*                         Koch'sche Schneeflocke                           */
/*                                                                          */
/****************************************************************************/

private int koch_stern(         // wendet eine Koch-Iteration an auf
        Point poly_punkt[],     // das Polygon poly_punkt
        int poly_punkt_cnt)     // mit poly_point_cnt Punkten
                                // neue Anzahl Punkte wird zurueckgeliefert
{
        int i,j;
        double x0,x1,x2,x3,x4,y0,y1,y2,y3,y4;
        Point[] HPunkt = new Point[MAX_POINTS];
        j=0;

        for (i=0; i<poly_point_cnt; i++)
        {
                x0 = poly_punkt[i].x;
                y0 = poly_punkt[i].y;

                x4 = poly_punkt[(i+1)%(poly_point_cnt)].x;
                y4 = poly_punkt[(i+1)%(poly_point_cnt)].y;

                x1 = (x4-x0)/3+x0;
                y1 = (y4-y0)/3+y0;

                x3 = x4 - (x4-x0)/3;
                y3 = y4 - (y4-y0)/3;

                x2 = (int)(x1+ (x3-x1)*cos_60 - (y3-y1)*sin_60);
                y2 = (int)(y1+ (y3-y1)*cos_60 + (x3-x1)*sin_60);

                HPunkt[j]   = new Point();
                HPunkt[j+1]   = new Point();
                HPunkt[j+2]   = new Point();
                HPunkt[j+3]   = new Point();

                HPunkt[j]       = poly_punkt[i];
                HPunkt[j+1].x       = (int)x1;
                HPunkt[j+1].y       = (int)y1;
                HPunkt[j+2].x       = (int)x2;
                HPunkt[j+2].y       = (int)y2;
                HPunkt[j+3].x       = (int)x3;
                HPunkt[j+3].y       = (int)y3;
                j=j+4;
        }

        for (i=0; i<j; i++)                       // neu berechnetes Polygon
                poly_punkt[i] = HPunkt[i];        // kopieren

        return j;                                 // neue Anzahl Punkte zurueck
}
```

8.3 Lindenmayer-Systeme

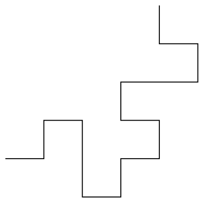
Eine nicht-grafische Beschreibung mancher Fraktale erfolgt mit Lindenmayer-Systemen, kurz L-Systems.
Ein Beispiel für ein L-System ist die quadratische Koch-Kurve:

$$\begin{array}{lccl} \text{Alphabet } \sum = \{r,u,l,d\} \text{ für right, up, left, down} & & & \\ \text{Regelmenge } f = \{ & r & \Rightarrow & r\ u\ r\ d\ d\ r\ u\ r, \\ & u & \Rightarrow & u\ l\ u\ r\ r\ u\ l\ u, \\ & l & \Rightarrow & l\ d\ l\ u\ u\ l\ d\ l, \\ & d & \Rightarrow & d\ r\ d\ l\ l\ d\ r\ d\ \} \end{array}$$

Ausgehend von einem Startwort w wird in jedem Iterationsschritt auf jedes Zeichen von w eine Regel aus f angewandt.
Sei $w = r\ u$. Dann ist $f(w) = r\ u\ r\ d\ d\ r\ u\ r\ u\ l\ u\ r\ r\ u\ l\ u$.

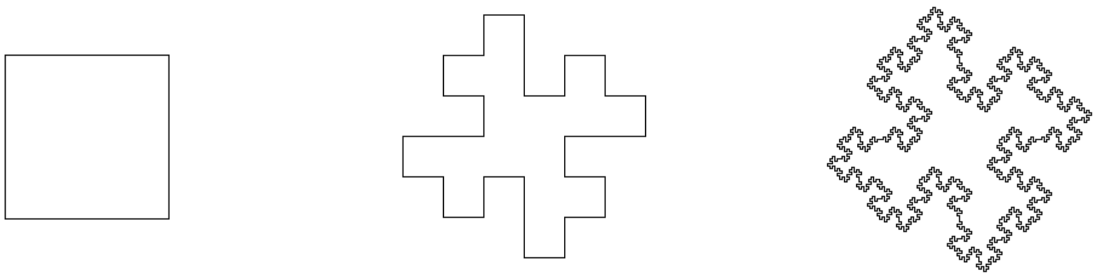


w



$f(w)$

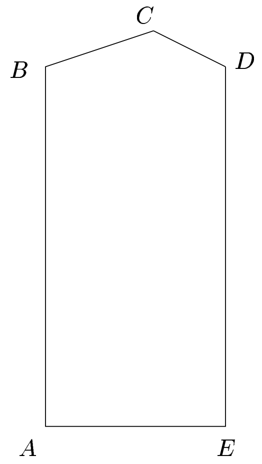
Ist nach n Iterationen das Wort $f^n(w)$ entstanden, so kann es, zusammen mit dem Parameter n (erforderlich für die Schrittweite), einer Ausgabeprozedur übergeben werden.
Die quadratische Koch-Kurve hat die Dimension $D = \frac{\log(8)}{\log(4)} = 1.5$, denn jeder Kantenzug besteht aus $N = 8$ Kopien, jeweils skaliert um den Faktor $r = \frac{1}{4}$.



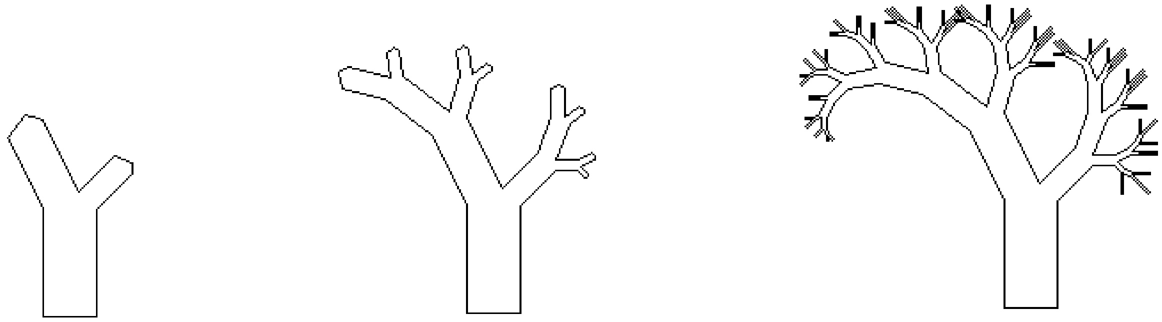
L-System mit 4, 32 und 16384 Polygonpunkten

8.4 Baumstrukturen

Gegeben ein "Ast", repräsentiert durch ein 5-seitiges Polygon:



An den Kanten $\overline{BC}$ bzw. $\overline{CD}$ können nun weitere "Äste" als Polygone angesetzt werden, die bzgl. des gegebenen Polygons mit 0.75 bzw. 0.47 skaliert sind. Durch wiederholtes Anstückeln entsteht ein Baum mit immer feinerer Verästelung. Statt dieses deterministischen Verfahrens kann man z.B. die Lage des Punktes C von einem Zufallsparameter beeinflussen lassen.



Vom Baum-Algorithmus erzeugter Baum mit 11, 47 bzw. 3071 Polygonpunkten

```
/*****************************************************************************/
/*                                                                           */
/*                               Baum                                        */
/*                                                                           */
/*****************************************************************************/

public void baum()              // wendet eine Iteration auf die Ansatzstuecke an
        //Point ansatz[],       // alte und neue Folge von Ansatzstuecken
        //int   anzahl,         // Anzahl der Anzsatzstueckendpunkte
        //Point poly_punkt[],   // alte und neue Folge von Polygonecken
        //int   poly_point_cnt  // Anzahl der Polygonecken
{
        int i,j,k,p;
        double a,b,l,si_de,co_de;
        Point P;
        Point Q;
        Point A;
        Point B;
        Point C;
        Point[] ast = new Point[3];
        Point[] Hpunkt = new Point[MAX_POINTS];
        Point[] Hansatz = new Point[MAX_POINTS];

        DoublePoint[] schema = new DoublePoint[3];

        ast[0] = new Point();
        ast[1] = new Point();
        ast[2] = new Point();

        schema[0] = new DoublePoint();
        schema[1] = new DoublePoint();
        schema[2] = new DoublePoint();

        schema[0].x = 0.0;                   // generiere Schema
        schema[0].y = 2.0;
        schema[1].x = 2.0/3.0;
        schema[1].y = 7.0/3.0;
        schema[2].x = 1.0;
        schema[2].y = 2.0;

        j=0;                                 // Index fuer neue Ansatzstuecke
        k=0;                                 // Index fuer neue Polygonpunkte
        for (p=0; p<anzahl; p=p+2)
        {
                P = ansatz[p];
                Q = ansatz[p+1];

                a     = P.y - Q.y;           // bestimme l, sinus, cosinus
                b     = Q.x - P.x;
                l     = Math.sqrt(a*a + b*b);
                si_de = a/l;
                co_de = b/l;
```

```
        for (i=0; i<3; i++)           // fuehre Skalierung und Rotation aus
        {
            ast[i].x = P.x + (int)(l*schema[i].x*co_de-l*schema[i].y*si_de);
            ast[i].y = P.y - (int)(l*schema[i].y*co_de+l*schema[i].x*si_de);
        }

        A = ast[0];                   // kreiere neue Ast-Punkte
        B = ast[1];
        C = ast[2];

        if (anzahl>2) del_line(P, Q); // loesche Ansatzstueck

        set_line(P,A);                // zeichne Figur P,A,B,C,Q
        set_line(A,B);
        set_line(B,C);
        set_line(C,Q);
                                      // speichere neue Ansatzstuecke
        Hansatz[j]          = new Point(A);
        Hansatz[j+1]        = new Point(B);
        Hansatz[j+2]        = new Point(B);
        Hansatz[j+3]        = new Point(C);
        j=j+4;
    }

    i=0;                              // Index fuer altes Polygon
    k=0;                              // Index fuer neues Polygon
    for (p=0; p<anzahl; p=p+2)        // Index fuer alte Ansatzstuecke
    {
        while  ((poly_punkt[i].x !=ansatz[p].x) ||
                (poly_punkt[i].y !=ansatz[p].y))
                                      // uebernimm unveraenderte Polygonpkte
            Hpunkt[k++] = poly_punkt[i++];

        Hpunkt[k]   = ansatz[p];      // uebernimm neue Polygonpunke
        Hpunkt[k+1] = Hansatz[2*p];
        Hpunkt[k+2] = Hansatz[2*p+1];
        Hpunkt[k+3] = Hansatz[2*p+3];
        Hpunkt[k+4] = ansatz[p+1];
        k+=4;
        i++;
    }

    while (i<poly_point_cnt) Hpunkt[k++]=poly_punkt[i++];

    for (i=0; i<j; i++)
        ansatz[i] = Hansatz[i];       // neu berechnete Ansatzstuecke kopieren
    anzahl=j;

    for (i=0; i<k; i++)
        poly_punkt[i] = Hpunkt[i];    // neu berechnete Polygonpunkte kopieren
    poly_point_cnt=k;
}
```

8.5 Mandelbrot-Menge

Sei z eine komplexe Zahl mit Realteil $z.re$ und Imaginärteil $z.im$:

```
public class Komplex
{
        double re;
        double im;

...
```

Das Quadrat einer komplexen Zahl z hat den Realteil $z.re^2 - z.im^2$ und den Imaginärteil $2 \cdot z.re \cdot z.im$. Der Betrag einer komplexen Zahl c sei $|c| = \sqrt{c.re^2 + c.im^2}$.
Betrachte die Funktion $f : \mathbb{C} \to \mathbb{C}, f(z) = z^2 + c$ für festes c.

```
Komplex f(Komplex z)
{
        double tmp;

        tmp   = z.re;
        z.re  = z.re * z.re - z.im * z.im + c.re;
        z.im  = 2 * tmp * z.im + c.im;

        return z
}
```

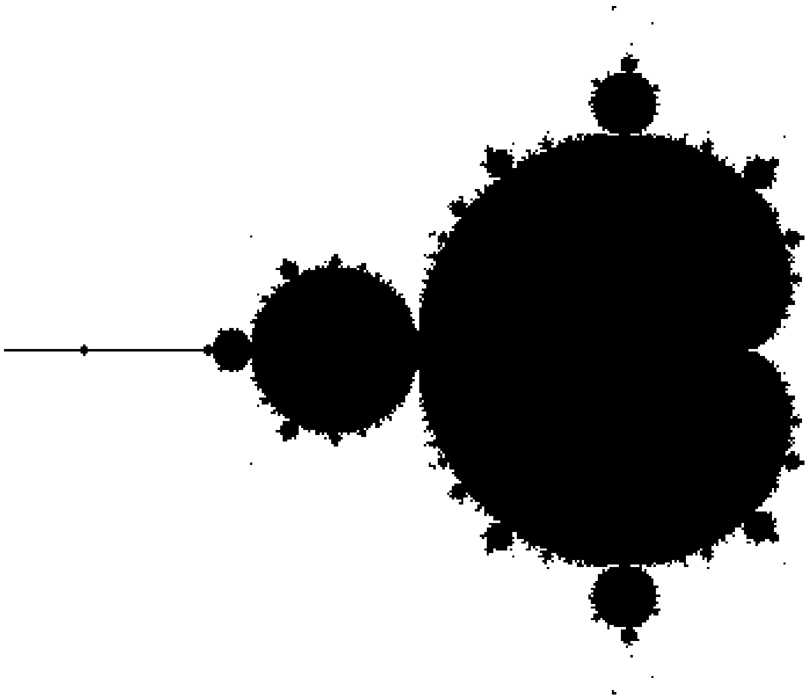
Betrachte die Folge $z, f(z), f^2(z), f^3(z), \ldots$ beginnend bei $z = 0$ für festes c.
Für manche c wächst der Betrag beliebig, für manche c konvergiert er zu einem Fixpunkt.
Die *Mandelbrotmenge* besteht aus solchen komplexen Zahlen c, die beim Startwert $z = 0$ zur Konvergenz führen.
Um die Mandelbrotmenge grafisch darzustellen, definiert man

$$\text{farbe}(c) := \begin{cases} \text{schwarz,} & \text{falls die zu } c \text{ gehörende Folge konvergiert} \\ \text{weiß,} & \text{falls die zu } c \text{ gehörende Folge nicht konvergiert} \end{cases}$$

Ordne jedem Pixel $(p.x, p.y)$ des Bildschirms eine komplexe Zahl zu wie folgt. Die linke obere Ecke bezeichne die komplexe Zahl `start` (z.B. $-2.15, 2.15$). Die rechte obere Ecke bezeichne die komplexe Zahl `ende` (z.B. $0.85, 2.15$)
Dann ergibt sich bei 300 Pixeln pro Zeile eine Schrittweite von $(0.85 + 2.15)/300 = 0.01$.
Somit läßt sich ein Koordinatenpaar $(p.x, p.y)$ umrechnen durch den folgenden Konstruktor.

```
public Komplex(Point p, Komplex start, double schritt)
// bildet die komplexe Zahl zum Pixel p mit linker/oberer Ecke start
// und Schrittweite schritt
{
        this.re = start.re + schritt * (double)p.x;
        this.im = start.im - schritt * (double)p.y;
}
```

Sei $(p.x, p.y)$ das zur komplexen Zahl c gehörende Pixel. Dann färbe es mit farbe(c).



Mandelbrotmenge: $-2.2 \le z.re \le 0.6$, $-1.2 \le z.im \le 1.2$, 100 Iterationen

Zur Implementation:

Sobald während des Iterierens der Betrag von $z > 2 \Rightarrow$ Folge konvergiert definitiv nicht.
Falls Betrag von z nach z.B. 100 Iterationen noch $< 2 \Rightarrow$ Folge konvergiert vermutlich.
D.h., bei Erreichen einer vorgegebenen Iterationenzahl wird das Pixel, welches der komplexen Zahl c entspricht, schwarz gefärbt, da c vermutlich zur Konvergenz führt.

```
void mandel ()
{
     Point p;
     int zaehler;
     Komplex c, z;

     for (p.x = 0; p.x < WIDTH;  p.x++)
     for (p.y = 0; p.y < HEIGHT; p.y++)
     {
          zaehler = 0;
          c       = new Komplex(p, start, schritt);
          z       = new Komplex(c);

          while ((betrag (z) < 2.0) && (zaehler++ < max_iter))

             z = f(z);

          if (betrag(z) < 2.0) set_pixel(p);
      }
}
```

Bemerkung: Bei Erhöhung der Iterationszahl können einige Pixel weiß werden, da sich herausgestellt hat, daß durch weitere Iterationen die Betragsgrenze 2 überschritten wird. Um die Zahlen c, die nicht zur Konvergenz geführt haben, weiter zu klassifizieren, setzt man

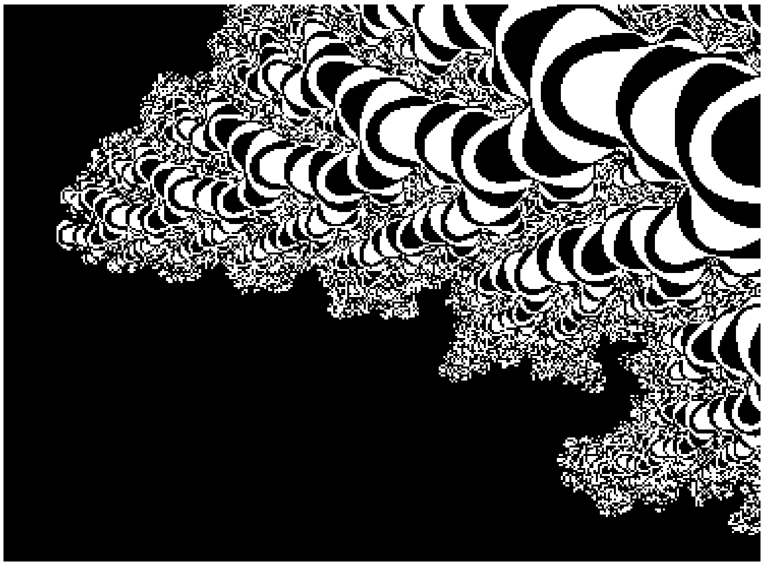
$$\text{farbe}(c) := \begin{cases} \text{schwarz,} & \text{falls konvergiert} \\ \text{weiß,} & \text{falls Iterationszahl nach Überschreiten von Betrag 2 gerade} \\ \text{schwarz,} & \text{falls Iterationszahl nach Überschreiten von Betrag 2 ungerade} \end{cases}$$

Also ergibt sich:

```
if (betrag < 2.0)        set_pixel (p); else
if ((zaehler %  2) != 0) set_pixel (p);
```

Bei Farbbildschirmen gibt es folgende Möglichkeit, die Divergenzgeschwindigkeit der Folge für ein c zu veranschaulichen:
Es seien die Farben `farbe[0], ..., farbe[NUM_COL-1]` vorhanden

```
set_col_pixel (p, farbe [zaehler % NUM_COL])
```



Vom Mandelbrot-Algorithmus erzeugtes Bild mit Modulo 2 und 100 Iterationen
$-1.2548 \leq Re \leq -1.2544$, $0.3816 \leq Im \leq 0.3822$

```
/****************************************************************************/
/*                    Mandelbrotmenge                                         */
/****************************************************************************/
void  mandel(                   // berechnet und zeichnet Mandelbrotmenge ab
        Komplex start,          // linker oberer Ecke start der komplexen Flaeche
        double  schritt,        // mit Schrittweite schritt und
        int     max_iter,       // Iterationsanzahl max_iter (muss gerade sein)
        int     fall)           // gemaess Fall fall
                                // Abbruch erfolgt, wenn der Betrag groesser als 2 ist
                                // oder wenn Iterationszahl erreicht ist
{
    Point p = new Point();
    int zaehler ;
    Komplex c;
    Komplex z;
    double tmp;

    for (p.x = 0; p.x < WIDTH;  p.x++)
    for (p.y = 0; p.y < HEIGHT; p.y++)
    {
        c = new Komplex(p, start, schritt);
        z = new Komplex(0.0, 0.0);
        zaehler = 0;

        while( (q_betrag(z) < 4.0)  && (zaehler++ < max_iter) ) {    // berechne f(z)
                tmp    = z.re;
                z.re   = z.re * z.re - z.im * z.im + c.re;
                z.im   = 2 * tmp * z.im + c.im;
        }

        switch (fall) {

        case 1: if (q_betrag(z) < 4.0) set_pixel(p); break;          // schwarz

        case 2: if (q_betrag(z) < 4.0) set_pixel(p); else            // schwarz
                if ((zaehler %2) != 0) set_pixel(p); break;     // mit modulo

        case 3: if(zaehler == max_iter+1)
                set_col_pixel(p,farbe[0]); else                 // Mandelbrotmenge selbst
                set_col_pixel(p,farbe[zaehler%NUM_COL]); break; // farbe modulo
        }
    }
}
```

8.6 Julia-Menge

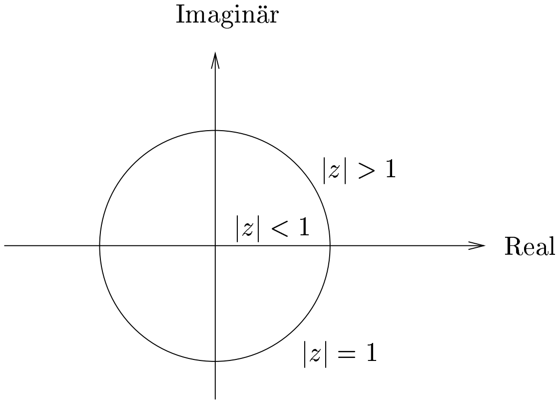
Gegeben eine komplexe Funktion $f : \mathbb{C} \to \mathbb{C}$, z.B. $f(z) = z^2$. Wähle Startwert z_0. Betrachte die Folge $z_0, {z_0}^2, z_0^4, z_0^8, \ldots$. Es gilt:

a) für $|z_0| < 1$ werden die erzeugten Zahlen immer kleiner und konvergieren zum Nullpunkt,

b) für $|z_0| > 1$ werden die erzeugten Zahlen immer größer und laufen gegen unendlich,

c) für $|z_0| = 1$ bleiben die Zahlen auf dem Einheitskreis um den Ursprung, der die Gebiete a) und b) trennt.

Die Menge c) wird *Julia-Menge* genannt. Sie ist invariant bzgl. der komplexen Funktion $f(z)$ und punktsymmetrisch zum Ursprung.



Julia-Kurve für $f(z) = z^2$

Es gilt: Julia-Menge für $f(z) = z^2 + c$ ist genau dann zusammenhängend, wenn c in der Mandelbrotmenge für $f(z)$ liegt.

Zur Implementation:

1. Möglichkeit: Sei c gegeben. Jeder Wert z der komplexen Fläche wird überprüft, ob er als Startwert zur Konvergenz führt:

```
void  julia_voll(         // berechnet Julia-Menge als gefuelltes Gebiet
        Komplex c,        // fuer komplexes c,
        Komplex start,    // die linke obere Ecke start der komplexen Flaeche,
        double schritt,   // Schrittweite schritt
        int max_iter)     // mit vorgegebener Iterationszahl max_iter
{
        Point p = new Point();
        Point q = new Point();          // Symmetrischer Punkt
        int zaehler ;
        Komplex z;

        for (p.x = 0; p.x <= WIDTH; p.x++)
        for (p.y = 0; p.y <= HEIGHT/2;  p.y++)
        // Wegen Symmetrie nur bis zur Haelfte laufen
```

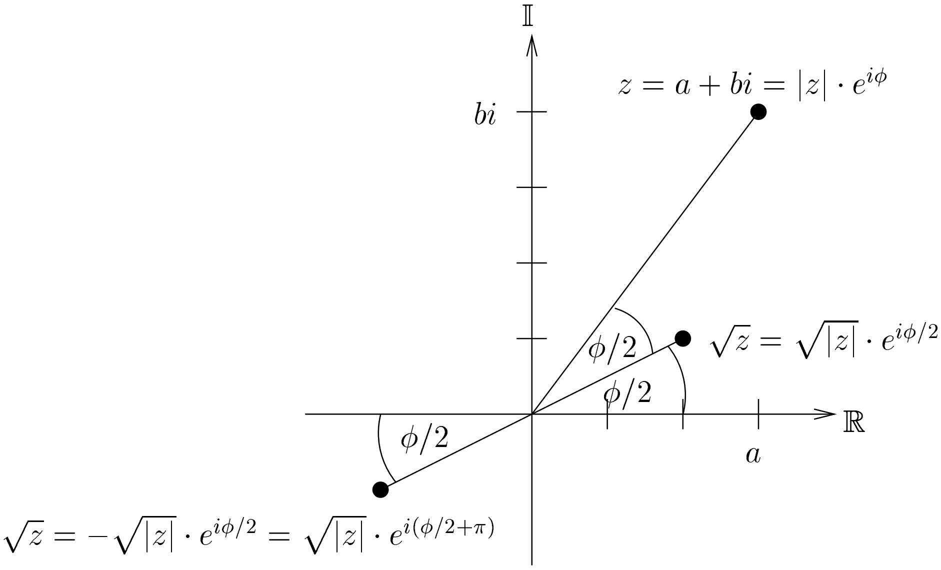
```
        {
                z = new Komplex(p, start, schritt);

                q.x = WIDTH  - p.x;     // Symmetrie nutzen
                q.y = HEIGHT - p.y;

                zaehler = 0;
                while ((q_betrag(z) < 4.0) && (zaehler++ < max_iter))
                {
                        z = f(z,c);
                }

                if (q_betrag(z) < 4.0)
                {
                        set_pixel(p);
                        set_pixel(q);
                }
        }
}
```

Es gilt: Der Rand der berechneten Fläche stellt die Julia-Menge für c dar.

2. Möglichkeit: *Inverse Iteration Method*:
Aus $f(z) = z^2 + c \Rightarrow f^{-1}(z) = \pm\sqrt{z-c}$

Sei $z = a + b \cdot i$ eine komplexe Zahl. Es gilt:

$$\sqrt{z} = \begin{cases} \pm\left(\sqrt{\frac{|z|+a}{2}} + i \cdot \sqrt{\frac{|z|-a}{2}}\right) & \text{falls } b \geq 0, \\ \pm\left(\sqrt{\frac{|z|+a}{2}} - i \cdot \sqrt{\frac{|z|-a}{2}}\right) & \text{falls } b < 0 \end{cases}$$



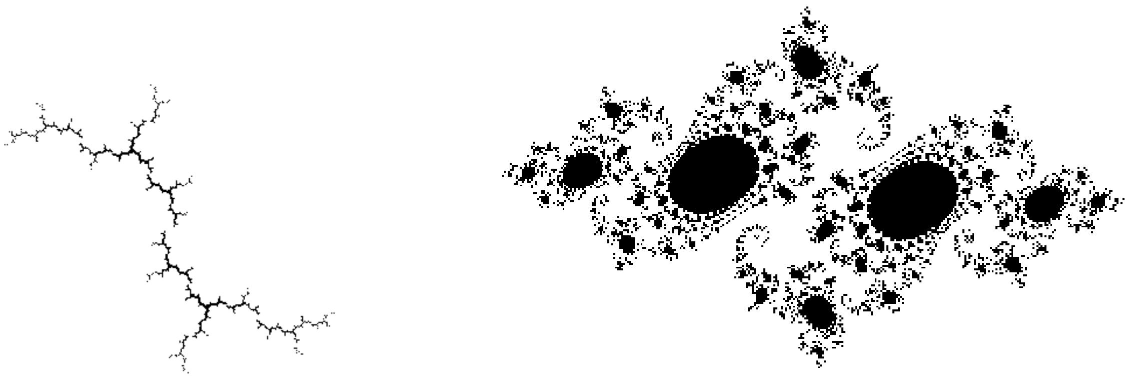
Realisiere f^{-1} durch

```
Komplex backward_random (Komplex z, Komplex c)
{
     /* z = z - c */
     /* bestimme r mit r * r = z */
     /* wuerfel das Vorzeichen von r */
     return (r);
}
```

Die Wahl des Vorzeichens bei dieser inversen Iteration erfolgt zufällig.

Ausgehend von $z = 1+0i$ werden zunächst 50 Iterationen $z := f^{-1}(z)$ durchgeführt, um die berechneten Punkte nahe genug an die Julia-Kurve heranzuführen. Iteriere danach weiter $z := f^{-1}(z)$ und zeige jeweils die berechneten z:

```
void julia_rueck(          // berechnet Julia-Menge
        Komplex c,         // fuer komplexes c
        Komplex start,     // linke obere Ecke der komplexen Flaeche
        double schritt,    // Schrittweite
        int anzahl)        // durch anzahl Rueckwaertsspruenge
{
        int k;
        Komplex z = new Komplex(1.0, 0.0);   // Startposition

        for (k=0; k<50; k++)         // tastet sich an die Julia-Kurve heran
            z = backward_random(z,c);


        for (k=0; k < anzahl; k++) {        // bestimme anzahl Vorgaenger
            z = backward_random(z,c);
            set_pixel(z.getPoint(start, schritt)); // und zeichne sie
        }

}
```



Vom Julia-Algorithmus erzeugte Bilder mit $c = i$ und $c = 0.746 + 0.112i$