Selbstorganisation von Vektoren

Wolfhard Hövel
Wolfhard.Hoevel@ohm-hochschule.de

Kurzfassung

In einem zeitdiskreten dynamischen System werden im n-dimensionalen euklidischen Raum Orts-, Abstands- und Verschiebungsvektoren über das Skalarprodukt miteinander verknüpft. Der hier vorgestellte Algorithmus erzeugt vielfältige Strukturen, die zum Teil instabil sind und nach einer gewissen Lebensdauer in andere stabilere Attraktoren umspringen können. Zunächst betrachte man zwei Punkte (Ortsvektoren) im Raum. Der eine Punkt wird iterativ um einen Einheitsvektor verschoben und der andere ebenfalls um diesen Einheitsvektor, jedoch mit negativem Vorzeichen. Damit bleibt der „Schwerpunkt" dieses Punktepaars exakt erhalten. Die antiparallelen Verschiebungsvektoren liegen auf Parallelen, deren Abstand ebenfalls konstant gehalten wird. Damit ist auch das „Drehmoment" der beiden Punkte konstant. Nun können weitere Punkte hinzugefügt werden. Die Punkte lassen sich frei wählbar ebenfalls zu Punktepaaren zusammenfassen. Es entsteht also ein Graph, die Knoten des Graphen entsprechen den Punkten, die Kanten den gewählten Punktepaaren. Das ausführbare Programm Attractor.jar demonstriert die Vielzahl von Möglichkeiten, mit denen sich die Vektoren in Abhängigkeit von Dimension und gewählten Graphen spontan organisieren können.

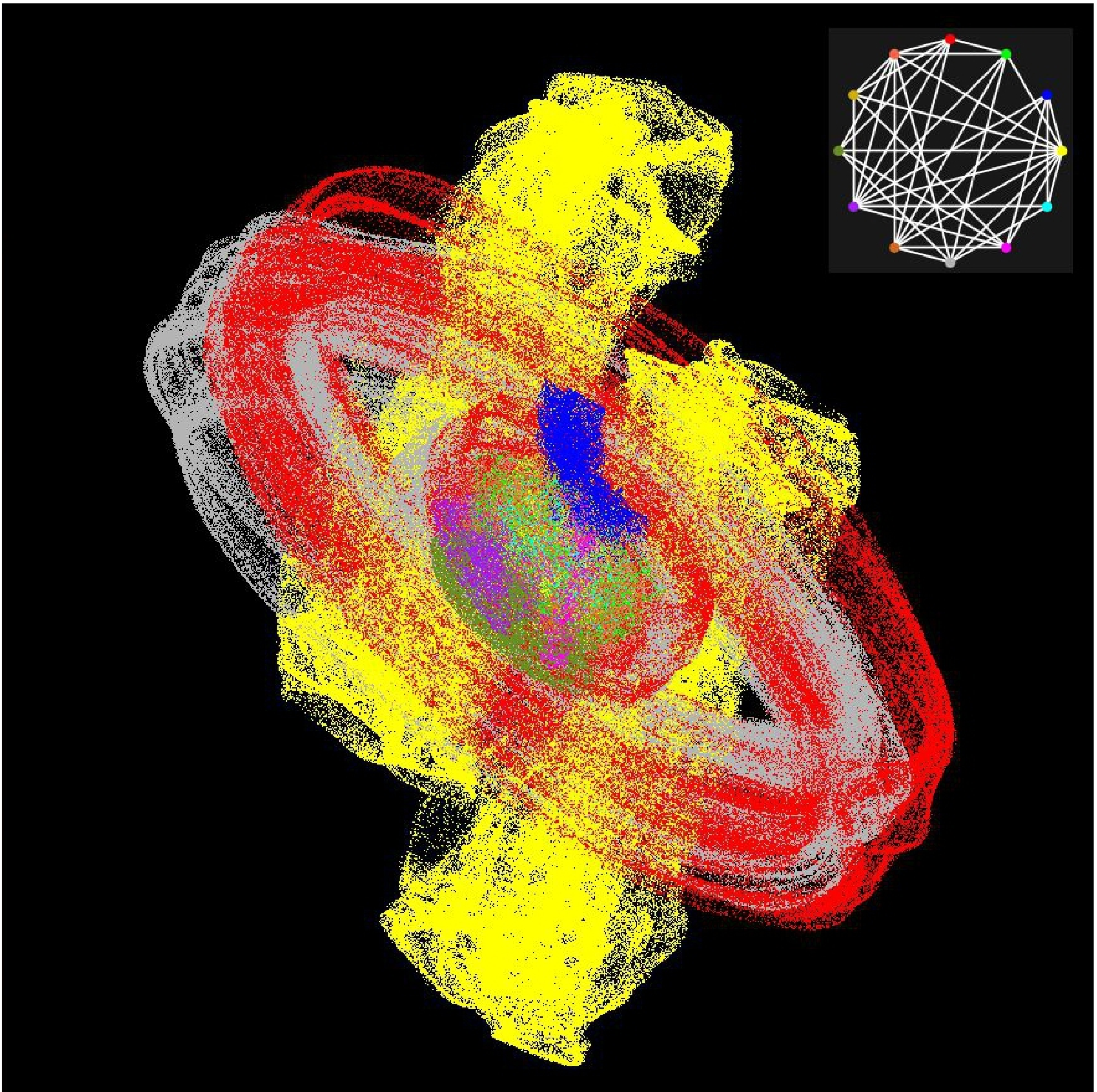

Bild 1: Beispiel für einen Attraktor.

1. Der Algorithmus

Die kleinste Einheit des Rechenverfahrens ist ein Punktepaar. Die Punkte werden im n-dimensionalen Raum durch die Ortsvektoren $\mathbf{r_1} = (a_1, a_2, \ldots, a_n)$ und $\mathbf{r_2} = (b_1, b_2, \ldots, b_n)$ dargestellt. Vektoren sollen durch Fettdruck gekennzeichnet sein. Der Ursprung des kartesischen Koordinatensystems $\mathbf{O} = (0, 0, \ldots, 0)$ legt den Fußpunkt der Ortsvektoren fest. Die beiden Punkte des Punktepaars haben den Abstand

$$\mathbf{r_{12}} = \mathbf{r_2} - \mathbf{r_1} = (b_1 - a_1, b_2 - a_2, \ldots, b_n - a_n).$$

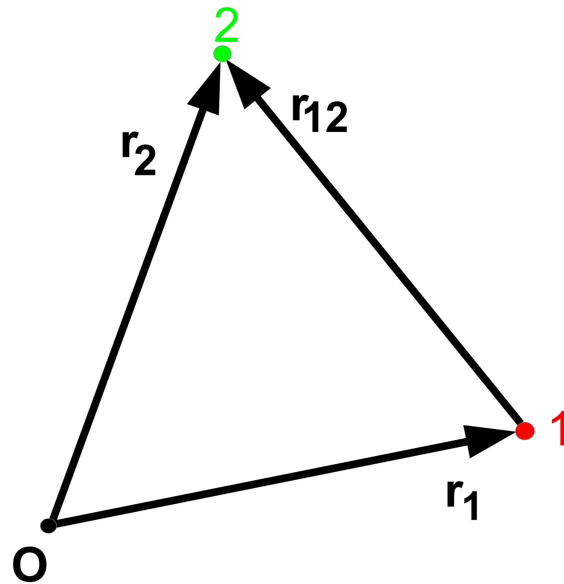

Bild 2: Abstandsvektor.

Nun werden die beiden Ortsvektoren $\mathbf{r}_1$ und $\mathbf{r}_2$ durch einen Verschiebungsvektor **dr** verändert. Zu $\mathbf{r}_1$ wird **dr** addiert und von $\mathbf{r}_2$ wird **dr** subtrahiert. Damit bleibt der Schwerpunkt der Punkte (**1**, **2**) konstant. Es ist sinnvoll aber nicht notwendig, dass für den Betrag von **dr** der Wert 1 gewählt wird, der Verschiebungsvektor sei also immer ein Einheitsvektor.

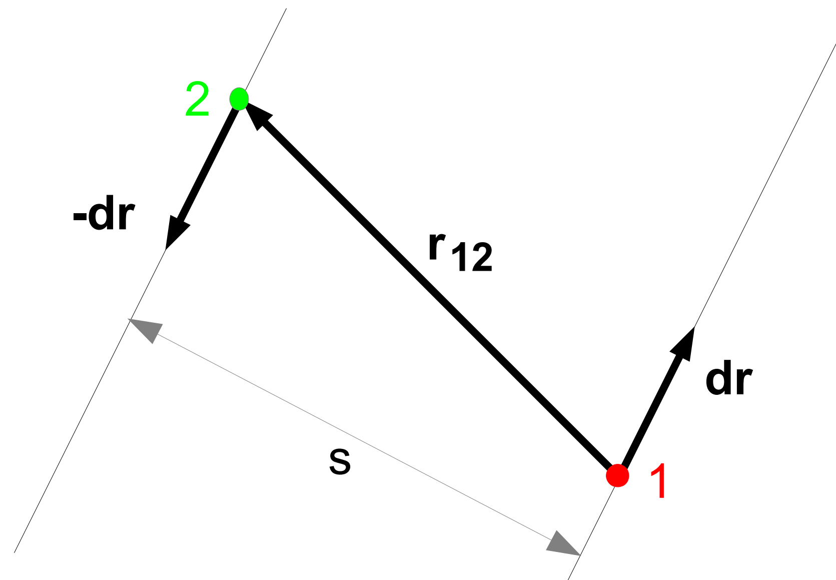

Bild 3: Verschiebung der Punkte.

Die beiden Punkte 1 und 2 bewegen sich gegenläufig auf zwei Parallelen mit dem Abstand s, somit bleibt das Drehmoment des Punktepaars ebenfalls konstant. Damit das System bei weiteren Iterationen nicht divergiert, muss eine obere Schranke eingeführt werden. Überschreitet der Betrag des Vektors $\mathbf{r}_{12}$ die frei wählbare Schranke e, wird die Richtung des Vektors **dr** derart geändert, dass das Drehmoment s konstant bleibt und das System nicht divergiert. Dabei muss immer $e \geq s$ gelten, da der Abstand der Punkte nicht kleiner als der Abstand der Parallelen sein kann. Die neue Richtung

des Verschiebungsvektors **dr** muss auf einer Tangente an einem Kreis, dessen Mittelpunkt mit dem Schwerpunkt des Punktepaars übereinstimmt und dessen Radius s/2 ist, liegen. Dann ist die Konstanz des Drehmoments gewährleistet. Der Verschiebungsvektor wird hier praktisch reflektiert.

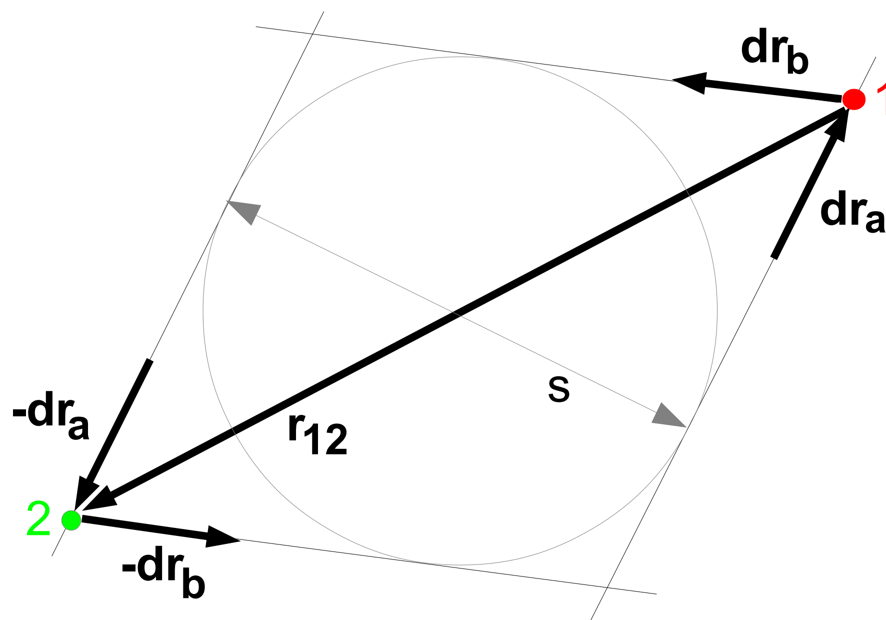

Bild 4: Reflexion.

Der Verschiebungsvektor $\mathbf{dr}_a$ geht ohne Änderung des Drehmomentes in den Vektor $\mathbf{dr}_b$ über. Die bisherige Betrachtung gilt nur für ein einziges Punktepaar. Interessant wir der Algorithmus jedoch erst dann, wenn weitere Punkte hinzugefügt werden. Zwischen den Punkten können beliebige neue Punktepaare gebildet werden. Die Vernetzung der Punkte lässt sich durch einen Graphen darstellen, wobei die Knoten des Graphen die Punkte sind und die Kanten des Graphen den Punktepaaren entsprechen. Gehört nun ein Punkt zu mehreren Punktepaaren (im Graphen laufen mehrere Kanten an einem Knoten zusammen), werden die entsprechenden Verschiebungsvektoren zu dem Ortsvektor addiert.
Wenn zum Beispiel ein dritter Punkt mit dem Punkt 1 ein neues Punktepaar bildet, wird der Punkt 1 zusätzlich verschoben und die Bedingung s = konstant gilt nicht mehr. Daher muss die Geometriebetrachtung erweitert werden (Bild 5). Der Verschiebungsvektor $\mathbf{dr}_a$ wird beim folgenden Iterationsschritt um den Punkt 1 soweit gedreht, bis er auf einer Tangente des s-Kreises liegt und in $\mathbf{dr}_b$ übergeht. Nun ist s = konstant wieder erfüllt. Zur Erleichterung der Berechnung werden einige Abkürzungen eingeführt:

$$|\mathbf{r}_{12}| = r,$$

$$|\mathbf{dr}_a| = |\mathbf{dr}_b| = 1.$$

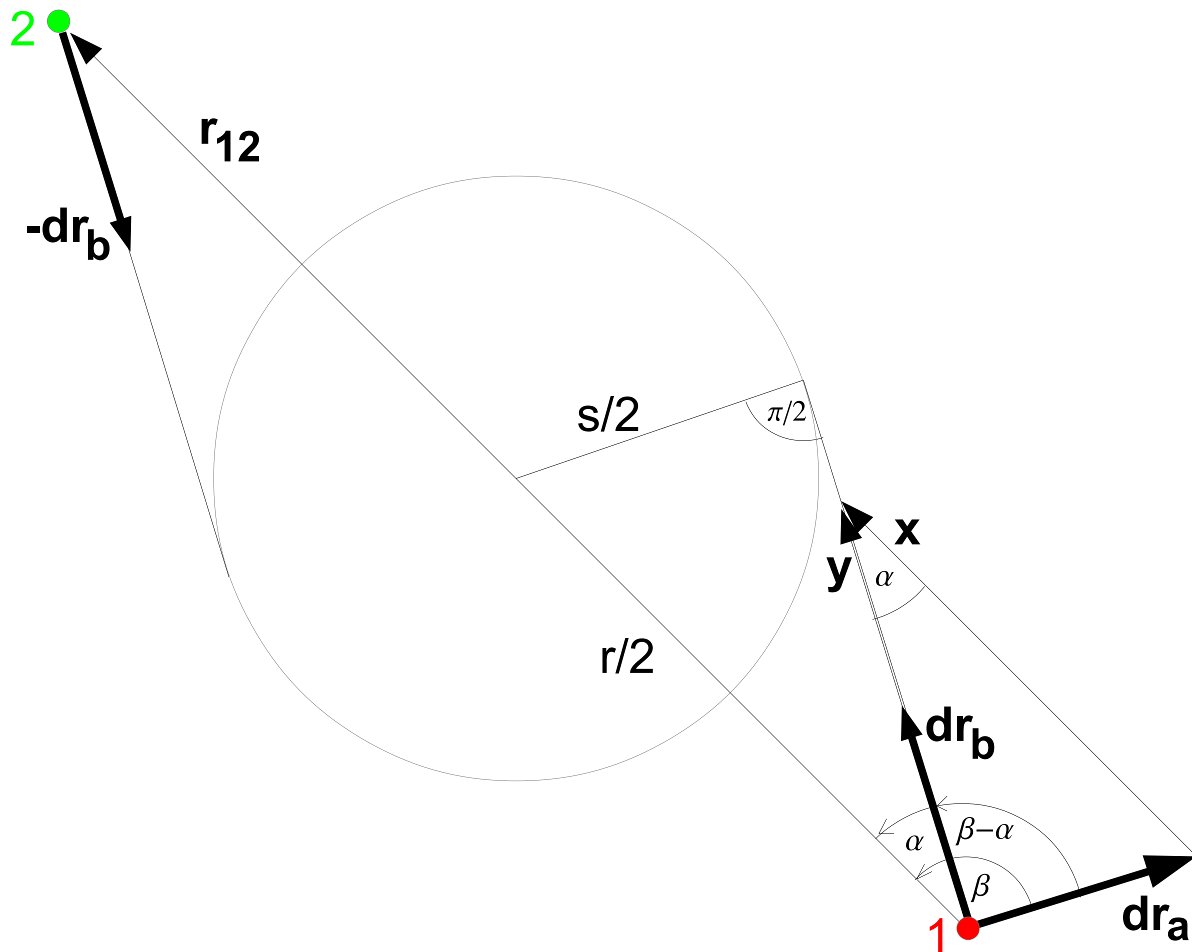

Bild 5: Ermittlung des neuen Verschiebungsvektors.

Der Vektor **y** wird aus der Summe der Vektoren $\mathbf{dr_a}$ und **x** gebildet, wobei **x** parallel zu $\mathbf{r_{12}}$ liegt

$$\boldsymbol{y}=\boldsymbol{dr}_a+\boldsymbol{x} \quad .$$

Der Vektor **x** ist gleich dem Einheitsvektor in Richtung von $\mathbf{r_{12}}$ multipliziert mit dem Betrag von **x**

$$\boldsymbol{x}=\frac{\boldsymbol{r}_{12}}{|\boldsymbol{r}_{12}|}x=\frac{\boldsymbol{r}_{12}}{r}x \quad \text{mit} \quad |\boldsymbol{x}|=x \quad .$$

Eine wichtige Rolle spielt das Skalarprodukt k, das im euklidischen Raum den Winkel β zwischen zwei Vektoren definiert. Zunächst notieren wir die Vektoren $\mathbf{r_{12}}$ und $\mathbf{dr_a}$ in Komponentenschreibweise

$$\boldsymbol{r}_{12}=(r_1, r_2, \ldots, r_n)$$
$$\boldsymbol{dr}_a=(dr_1, dr_2, \ldots, dr_n)$$

und damit

$$k=\boldsymbol{r}_{12}\cdot\boldsymbol{dr}_a=r_1\cdot dr_1+r_2\cdot dr_2+\ldots+r_n\cdot dr_n=|\boldsymbol{r}_{12}|\cdot|\boldsymbol{dr}_a|\cdot\cos\beta \quad .$$

Mit dem Skalarprodukt lässt sich der Betrag eines Vektors berechnen

$$|\boldsymbol{r}_{12}|=\sqrt{\boldsymbol{r}_{12}\cdot\boldsymbol{r}_{12}}=\sqrt{r_1^2+r_2^2+...+r_n^2}=r \quad .$$

Für den Winkel zwischen den Vektoren $\mathbf{r}_{12}$ und $\mathbf{dr}_a$ ergibt sich wegen $|\mathbf{r}_{12}| = r$ und $|\mathbf{dr}_a| = 1$

$$\cos\beta=\frac{k}{|\boldsymbol{r}_{12}|\cdot|\boldsymbol{dr}_a|}=\frac{k}{r}$$

und weiter mit $\sin^2\beta + \cos^2\beta = 1$

$$\sin\beta=\sqrt{1-cos^2\beta}=\sqrt{1-\frac{k^2}{r^2}} \quad .$$

Aus dem rechtwinkligen Dreieck in Bild 5 kann entnommen werden

$$\sin\alpha=\frac{s}{r} \quad .$$

Nun betrachte man das Dreieck, das aus den Vektoren $\mathbf{dr}_a$, $\mathbf{x}$ und $\mathbf{y}$ gebildet wird. Für dieses lautet der Sinussatz wegen $|\mathbf{x}| = x$ und $|\mathbf{dr}_a| = 1$

$$\frac{x}{\sin(\beta-\alpha)}=\frac{1}{\sin\alpha} \quad .$$

Hieraus wird der Skalar x mit $\sin(\beta-\alpha) = \sin\beta \cos\alpha - \cos\beta \sin\alpha$ berechnet

$$x=\frac{\sin(\beta-\alpha)}{\sin\alpha}=\frac{\sin\beta\cos\alpha-\cos\beta\sin\alpha}{\sin\alpha} \quad .$$

Mit

$$\cos\alpha=\sqrt{1-sin^2\alpha}=\sqrt{1-\frac{s^2}{r^2}}$$

und Einsetzen können die Winkelfunktionen eliminiert werden

$$x=\frac{\sqrt{1-\frac{k^2}{r^2}}\sqrt{1-\frac{s^2}{r^2}}-\frac{k}{r}\frac{s}{r}}{\frac{s}{r}}$$

$$x=\frac{r}{s}\left(\sqrt{\left(1-\frac{k^2}{r^2}\right)\left(1-\frac{s^2}{r^2}\right)}-\frac{k}{r^2}s\right)$$

$$x=-\frac{k}{r}+\frac{r}{s}\sqrt{\left(1-\frac{k^2}{r^2}\right)\left(1-\frac{s^2}{r^2}\right)} \quad .$$

Für die Länge des Vektors **x** ergibt sich

$$x = -\frac{k}{r} + \sqrt{\left(\frac{r^2}{s^2} - 1\right)\left(1 - \frac{k^2}{r^2}\right)}$$

und seine Richtung stimmt mit dem Vektor $\mathbf{r}_{12}$ überein

$$\boldsymbol{x} = x \frac{\boldsymbol{r}_{12}}{|\boldsymbol{r}_{12}|} \quad .$$

Durch Addition der Vektoren $\mathbf{dr}_a$ und **x** erhält man den Vektor **y,** dessen Richtung mit dem neuen Verschiebungsvektor $\mathbf{dr}_b$ übereinstimmt. Der Vektor **y** braucht nur noch auf die Länge 1 normiert zu werden

$$\boldsymbol{y} = \boldsymbol{dr}_a + \boldsymbol{x} = \boldsymbol{dr}_a + x \frac{\boldsymbol{r}_{12}}{|\boldsymbol{r}_{12}|}$$

$$\boldsymbol{dr}_b = \frac{\boldsymbol{y}}{|\boldsymbol{y}|} = \frac{\boldsymbol{dr}_a + \boldsymbol{x}}{|\boldsymbol{dr}_a + \boldsymbol{x}|} \quad .$$

2. Beispielprogramm AttracSimple

Werden die obigen Gleichungen in einen Algorithmus umgesetzt, führt dies zu einem recht einfachen Programm. Dies soll im folgenden Beispiel demonstriert werden. Der Graph besteht aus den Punkten (Knoten) 1, 2 und 3. Zwischen ihnen wurden zwei Punktepaare (Kanten) 12 und 13 gewählt.

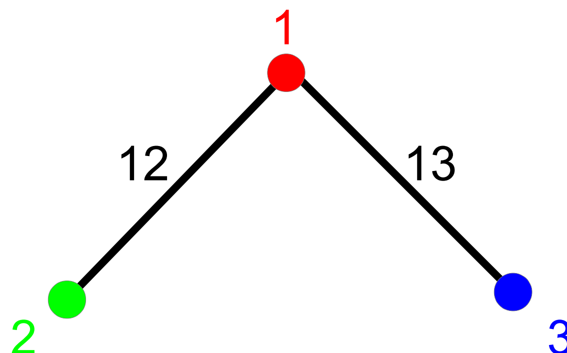

Bild 6: Graph für AttracSimple.

Das Java-Programm AttracSimple zeigt das Prinzip für drei Dimensionen, andere Dimensionen sind einfach durch Änderung der Anzahl der Vektorkomponenten in diesem Quelltext möglich. Der Parameter e legt die maximale Länge der Abstandsvektoren zwischen den Punkten fest. Die Größe s bestimmt den Abstand der Parallelen auf denen die antiparallelen Verschiebungsvektoren eines Punktepaars liegen (Bild 3) und damit auch das Drehmoment. Dann werden die Anfangswerte für die Ortsvektoren der Punkte eingegeben. Für jedes Punktepaar muss außerdem ein erster Verschiebungsvektor definiert werden. Dieser muss nicht normiert sein, da er automatisch immer in

einen Einheitsvektor umgerechnet wird.

```
package attracsimple;

public class AttracSimple {
  public static void main(String[] args) {

    // Parameters
    double e = 5.0, s = 3.0;   // e >= s, s > 0

    double radi, dr, x;

    //Initial conditions (x, y, z)
    double x1 = 0.01, y1 = -0.02, z1 = 0.04;
    double x2 = -0.04, y2 = -0.05, z2 = 0.01;
    double x3 = 0.05, y3 = -0.04, z3 = -0.01;
    double rq12, r12, scalar12, dx12 = 1.0, dy12 = -1.0, dz12 = 1.0;
    double rq13, r13, scalar13, dx13 = -1.0, dy13 = 1.0, dz13 = -1.0;
    int N = 15000; //steps
```

Nun beginnt die Iterationsschleife. Nach Berechnung des Abstandes r_{12} zwischen den Punkten 1 und 2 wird das Skalarprodukt $\mathbf{r}_{12}\cdot\mathbf{dr}_{12}$ berechnet. Dann wird abgefragt, ob die obere Schranke für den Punkteabstand erreicht ist und ob der Betrag von $\mathbf{r}_{12}$ größer als s ist. Falls diese Bedingungen erfüllt sind, wird die Länge des Hilfsvektors **x** berechnet.

```
    //Iteration loop---------------------------------------------------------
    for(int i = 0; i < N; i++) {
      rq12 = (x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1);
      r12 = Math.sqrt(rq12);
      scalar12 = dx12*(x2-x1)+dy12*(y2-y1)+dz12*(z2-z1);
      if(r12 > e && r12 > s) {
        radi = (rq12/(s*s)-1.0)*(1.0-(scalar12*scalar12)/rq12);
        if(radi < 0.0) radi= 0.0;
        x = -scalar12/r12+Math.sqrt(radi);
```

Zunächst werden die Komponenten des Vektors **y** bestimmt und daraus durch Normierung auf die Länge 1 der neue Verschiebungsvektor.

```
        dx12 = dx12+x*(x2-x1)/r12;
        dy12 = dy12+x*(y2-y1)/r12;
        dz12 = dz12+x*(z2-z1)/r12;
        dr = Math.sqrt(dx12*dx12+dy12*dy12+dz12*dz12);
        dx12 = dx12/dr;
        dy12 = dy12/dr;
        dz12 = dz12/dr;
      }
```

Die gleiche Rechnung wird für das Punktepaar 1 und 3 durchgeführt. Hier wäre es natürlich sinnvoll gleiche Programmteile zu einer Methode oder Funktion zusammenzufassen, wegen besserer Übersicht wurde hier aber darauf verzichtet.

```
      rq13 = (x3-x1)*(x3-x1)+(y3-y1)*(y3-y1)+(z3-z1)*(z3-z1);
      r13 = Math.sqrt(rq13);
      scalar13 = dx13*(x3-x1)+dy13*(y3-y1)+dz13*(z3-z1);
      if(r13 > e && r13 > s) {
        radi = (rq13/(s*s)-1.0)*(1.0-(scalar13*scalar13)/rq13);
        if(radi < 0.0) radi = 0.0;
        x = -scalar13/r13+Math.sqrt(radi);
        dx13 = dx13+x*(x3-x1)/r13;
        dy13 = dy13+x*(y3-y1)/r13;
        dz13 = dz13+x*(z3-z1)/r13;
        dr = Math.sqrt(dx13*dx13+dy13*dy13+dz13*dz13);
        dx13 = dx13/dr;
        dy13 = dy13/dr;
        dz13 = dz13/dr;
      }
```

Nun sind die beiden neuen Verschiebungsvektoren bekannt. Damit werden die Ortsvektoren der Punkte geändert. Da beim Punkt 1 zwei Punktepaare berücksichtigt werden müssen, werden die Vektoren $\mathbf{dr}_{12}$ und $\mathbf{dr}_{13}$ zu $\mathbf{r}_1$ addiert. Zu den Ortsvektoren $\mathbf{r}_2$ und $\mathbf{r}_3$ werden die jeweiligen antiparallelen Verschiebungsvektoren addiert. Dann können die neuen Punkte ausgegeben werden.

```
    //Superposition
    x1 = x1+dx12+dx13;
    y1 = y1+dy12+dy13;
    z1 = z1+dz12+dz13;
    x2 = x2-dx12;
    y2 = y2-dy12;
    z2 = z2-dz12;
    x3 = x3-dx13;
    y3 = y3-dy13;
    z3 = z3-dz13;
    //Position of node1, node2 and node3
    System.out.format("%10f %10f %10f %10f %10f %10f %n",
    x1, y1, x2, y2, x3, y3);
    }
  }
}
```

Im Anhang ist das Programm AttracSimple nochmals zusammenhängend wiedergegeben.

3. Beispielprogramm AttracDemo

Als zweites Beispiel zeigt das Java-Programm AttracDemo, wie man mit der Klasse „Vector“ alle benötigten Vektoroperationen definiert. Um Anfangsbedingungen festzulegen, wird ein Zufalls-vektor konstruiert. Dann folgen die Operationen

Multiplikation eine Vektors mit einem Skalar
Summe zweier Vektoren
Differenz zweier Vektoren

Skalarprodukt zweier Vektoren
Betrag eines Vektors
Einheitsvektor eines Vektors
Reflexion eines Verschiebungsvektors

Damit lassen sich beliebig weitere Knoten und Kanten zum Algorithmus einfügen. Im Beispiel besteht der Graph aus drei Knoten und drei Kanten. Zunächst werden die drei Abstandsvektoren berechnet. Dann wird abgefragt, ob die oberen Schranken für die Punkteabstände erreicht sind und neue Verschiebungsvektoren bestimmt werden müssen. Aus Sicherheitsgründen wird noch geprüft, ob der Abstand der Punkte größer als s ist. Jetzt kann für jeden Ortsvektor durch Überlagerung der entsprechenden Verschiebungsvektoren die neue Lage der Punkte ermittelt werden. In diesem Beispiel sind jeweils zwei Verschiebungsvektoren zu berücksichtigen, da im Graphen in jedem Knoten zwei Kanten münden. Die farbigen Punkte 0, 1 und 2 werden auf dem Bildschirm nach jedem Iterationsschritt ausgegeben. Im Anhang ist das Programm AttracDemo nochmals zusammenhängend wiedergegeben.

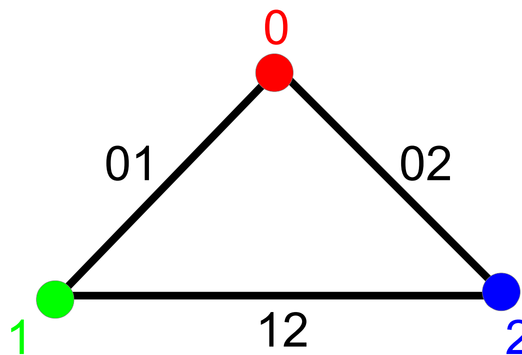

Bild 7: Graph für AttracDemo.

```
/===========================================================
//Algorithm with vectors
    for( int i = 0; i < n; ++i){
      r01 = r1.minus(r0);
      r02 = r2.minus(r0);
      r12 = r2.minus(r1);
      //Reflection?
      if (r01.mag() > Vector.e && r01.mag() > Vector.s) dr01 = r01.reflect(dr01);
      if (r02.mag() > Vector.e && r02.mag() > Vector.s) dr02 = r02.reflect(dr02);
      if (r12.mag() > Vector.e && r12.mag() > Vector.s) dr12 = r12.reflect(dr12);
      // Superposition
      // node 0 (red)
      r0 = r0.plus(dr01);
      r0 = r0.plus(dr02);
      // node 1 (green)
      r1 = r1.minus(dr01);
      r1 = r1.plus(dr12);
      // node 2 (blue)
      r2 = r2.minus(dr02);
      r2 = r2.minus(dr12);
//===========================================================
```

4. Das Programm Attractor.jar

Im beigefügten ausführbaren Programm Attractor.jar sind maximal 12 Knoten und 66 Kanten möglich. Im Anhang ist die Bedienung des Programms über die Benutzeroberfläche näher erläutert. Die Dimension n kann für n >1 beliebig gewählt werden. Die Parameter e und s haben einen großen Einfluss darauf, ob sich Attraktoren bilden können. Es sind eine ganze Reihe von Graphen aufrufbar, jedoch können zusätzlich beliebige Graphen durch Zufallszahlen erzeugt und untersucht werden. Interessant ist auch das dynamische Verhalten der Strukturen, das mit den Einstellungen „Strobe“ und „Period“ besser beobachtet werden kann.

5. Selbstorganisation

Der beschriebene Algorithmus verknüpft Orts-, Abstands- und Verschiebungsvektoren miteinander. Dabei stellt sich heraus, dass sich die Vektoren in vielen Fällen zu Strukturen selbst organisieren. Selbst bei Graphen mit 12 Ortsvektoren und 33 Abstands- und Verschiebungsvektoren in der 7. Dimension lässt sich dieser Effekt feststellen. Im angeführten Beispiel sind immerhin 547 Vektorkomponenten miteinander vernetzt und bilden die verschiedenartigsten Attraktoren. Dabei scheint es, dass Selbstorganisation für höhere Dimensionen leichter eintreten kann.
Die Attraktoren haben unterschiedliche Strukturen. Die erzeugte Punktwolke kann sich auf mehr oder weniger viele feste Punkte zusammenziehen. Die „Spitzen“ der Ortsvektoren laufen manchmal auf geschlossenen Kurven oder beschreiben Flächen. Ebenso sind diffuse Strukturen möglich. Diese scheinen weniger stabil zu sein, da sie nach vielen Iterationsschritten in einfachere Gebilde umspringen können.

6. Anhang

6.1 Listing AttracSimple

```
package attracsimple;

public class AttracSimple {
  public static void main(String[] args) {

    // Parameters
    double e = 5.0, s = 3.0;   // e >= s, s > 0

    double radi, dr, x;

    //Initial conditions (x, y, z)
    double x1 = 0.01, y1 = -0.02, z1 = 0.04;
    double x2 = -0.04, y2 = -0.05, z2 = 0.01;
    double x3 = 0.05, y3 = -0.04, z3 = -0.01;
    double rq12, r12, scalar12, dx12 = 1.0, dy12 = -1.0, dz12 = 1.0;
    double rq13, r13, scalar13, dx13 = -1.0, dy13 = 1.0, dz13 = -1.0;
    int N = 15000; //steps

    //Iteration loop---------------------------------------------------------
    for(int i = 0; i < N; i++) {
      rq12 = (x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1);
```

```
        r12 = Math.sqrt(rq12);
        scalar12 = dx12*(x2-x1)+dy12*(y2-y1)+dz12*(z2-z1);
        if(r12 > e && r12 > s) {
          radi = (rq12/(s*s)-1.0)*(1.0-(scalar12*scalar12)/rq12);
          if(radi < 0.0) radi= 0.0;
          x = -scalar12/r12+Math.sqrt(radi);
          dx12 = dx12+x*(x2-x1)/r12;
          dy12 = dy12+x*(y2-y1)/r12;
          dz12 = dz12+x*(z2-z1)/r12;
          dr = Math.sqrt(dx12*dx12+dy12*dy12+dz12*dz12);
          dx12 = dx12/dr;
          dy12 = dy12/dr;
          dz12 = dz12/dr;
        }
        rq13 = (x3-x1)*(x3-x1)+(y3-y1)*(y3-y1)+(z3-z1)*(z3-z1);
        r13 = Math.sqrt(rq13);
        scalar13 = dx13*(x3-x1)+dy13*(y3-y1)+dz13*(z3-z1);
        if(r13 > e && r13 > s) {
          radi = (rq13/(s*s)-1.0)*(1.0-(scalar13*scalar13)/rq13);
          if(radi < 0.0) radi = 0.0;
          x = -scalar13/r13+Math.sqrt(radi);
          dx13 = dx13+x*(x3-x1)/r13;
          dy13 = dy13+x*(y3-y1)/r13;
          dz13 = dz13+x*(z3-z1)/r13;
          dr = Math.sqrt(dx13*dx13+dy13*dy13+dz13*dz13);
          dx13 = dx13/dr;
          dy13 = dy13/dr;
          dz13 = dz13/dr;
        }
      //Superposition
      x1 = x1+dx12+dx13;
      y1 = y1+dy12+dy13;
      z1 = z1+dz12+dz13;
      x2 = x2-dx12;
      y2 = y2-dy12;
      z2 = z2-dz12;
      x3 = x3-dx13;
      y3 = y3-dy13;
      z3 = z3-dz13;
      //Position of node1, node2 and node3
      System.out.format("%10f %10f %10f %10f %10f %10f %n",
      x1, y1, x2, y2, x3, y3);
      }
    }
  }
```

6.2 Listing AttracDemo

```
package attracdemo;

public class Vector {
  public double[] vect;
  public static int dim = 4; //dim > 1
  public static int s = 1; //s > 0
  public static int e = 4; //e >= s

  //Constructor
  public Vector( double[] vect) {
    this.vect = vect;
  }

  //Returns a random vector (initial conditions)
  public Vector(double c){
    this.vect = new double[dim];
    for (int i = 0; i < dim; i++)
      this.vect[i] = c*(Math.random()-0.5);
  }

  // vector times scalar
  public Vector mult(double b){
    double[] temp = new double[dim];
    Vector a = this;
    for (int i = 0; i < dim; i++)
      temp[i] = b * a.vect[i];
    Vector c = new Vector(temp);
    return c;
  }

  //Sum of two vectors
  public Vector plus(Vector b){
    double[] temp = new double[dim];
    Vector a = this;
    for (int i = 0; i < dim; i++)
      temp[i] = a.vect[i] + b.vect[i];
    Vector c = new Vector(temp);
    return c;
  }
  //Difference of two vectors
  public Vector minus(Vector b){
    double[] temp = new double[dim];
    Vector a = this;
    for (int i = 0; i < dim; i++)
      temp[i] = a.vect[i] - b.vect[i];
    Vector c = new Vector(temp);
    return c;
  }
```

```
    // Dot product of two vectors
    public double dot(Vector b){
        Vector a = this;
        double c = 0.0;
        for (int i = 0; i < dim; i++)
            c = c + a.vect[i] * b.vect[i];
        return c;
    }

    //Returns the magnitude of this vector.
    public double mag() {
        return Math.sqrt(this.dot(this));
    }

    //Returns the unit vector of this vector.
    public Vector unit() {
        double[] tmp = new double[dim];
        double scalar = this.mag();
        if (scalar > 0.0)
            scalar = 1.0 / scalar;
        for (int i = 0; i < dim; i++)
        tmp[i] = scalar * this.vect[i];
        return new Vector(tmp);
    }

    //Reflection at the boundary
    public Vector reflect(final Vector dre) {
        double r, rq, scalar, radi, x;
        rq = this.dot(this);
        if (rq < 1.0E-15) rq = 1.0E-15;
        r = Math.sqrt(rq);
        scalar = this.dot(dre);
        radi = (rq / (s*s) - 1.0)
              * (1.0 - scalar*scalar / rq);
        if (radi < 0) radi = 0;
        x = -scalar / r + Math.sqrt(radi);
        Vector dra = dre.plus(this.mult(x / r)).unit();
        return dra;
    }
}

package attracdemo;

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

```
public class AttracDemo {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override public void run() {
        createAndShowGUI();
      }
    });
  }

  private static void createAndShowGUI() {
    JFrame f = new JFrame("AttracDemo");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.add(new MyPanel());
    f.pack();
    f.setVisible(true);
  }
}

class MyPanel extends JPanel {
  public static Vector r0 = new Vector(0.1);
  public static Vector r1 = new Vector(0.1);
  public static Vector r2 = new Vector(0.1);
  public static Vector r01 = new Vector(0.0);
  public static Vector r02 = new Vector(0.0);
  public static Vector r12 = new Vector(0.0);
  public static Vector dr01 = new Vector(1.0);
  public static Vector dr02 = new Vector(1.0);
  public static Vector dr12 = new Vector(1.0);
  Color color0 = new Color( 255, 0, 0 );
  Color color1 = new Color( 0, 255, 0 );
  Color color2 = new Color( 0, 0, 255 );

  public MyPanel() {
    setBackground(Color.BLACK);
  }

  @Override
  public Dimension getPreferredSize() {
    return new Dimension(800,800);
  }
  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    //Paint specifications
    double zoom = 100.0;
    int x0 = 400;
    int y0 = 400;
    int n = 300000; //Number of pixels
```

```
//=============================================================
//Algorithm with vectors
    for( int i = 0; i < n; ++i){
      r01 = r1.minus(r0);
      r02 = r2.minus(r0);
      r12 = r2.minus(r1);
      //Reflection?
      if (r01.mag() > Vector.e && r01.mag() > Vector.s) dr01 = r01.reflect(dr01);
      if (r02.mag() > Vector.e && r02.mag() > Vector.s) dr02 = r02.reflect(dr02);
      if (r12.mag() > Vector.e && r12.mag() > Vector.s) dr12 = r12.reflect(dr12);
      // Superposition
      // node 0 (red)
      r0 = r0.plus(dr01);
      r0 = r0.plus(dr02);
      // node 1 (green)
      r1 = r1.minus(dr01);
      r1 = r1.plus(dr12);
      // node 2 (blue)
      r2 = r2.minus(dr02);
      r2 = r2.minus(dr12);
//=============================================================
    //Draw Pixels
      g.setColor(color0);
      g.drawLine(x0 + (int)(zoom*(r0.vect[0])), y0 - (int)(zoom*(r0.vect[1])),
      x0 + (int)(zoom*(r0.vect[0])), y0 - (int)(zoom*(r0.vect[1])));
      g.setColor(color1);
      g.drawLine(x0 + (int)(zoom*(r1.vect[0])), y0 - (int)(zoom*(r1.vect[1])),
      x0 + (int)(zoom*(r1.vect[0])), y0 - (int)(zoom*(r1.vect[1])));
      g.setColor(color2);
      g.drawLine(x0 + (int)(zoom*(r2.vect[0])), y0 - (int)(zoom*(r2.vect[1])),
      x0 + (int)(zoom*(r2.vect[0])), y0 - (int)(zoom*(r2.vect[1])));
    }
  }
}
```

6.3 Benutzeroberfläche für Attractor.jar

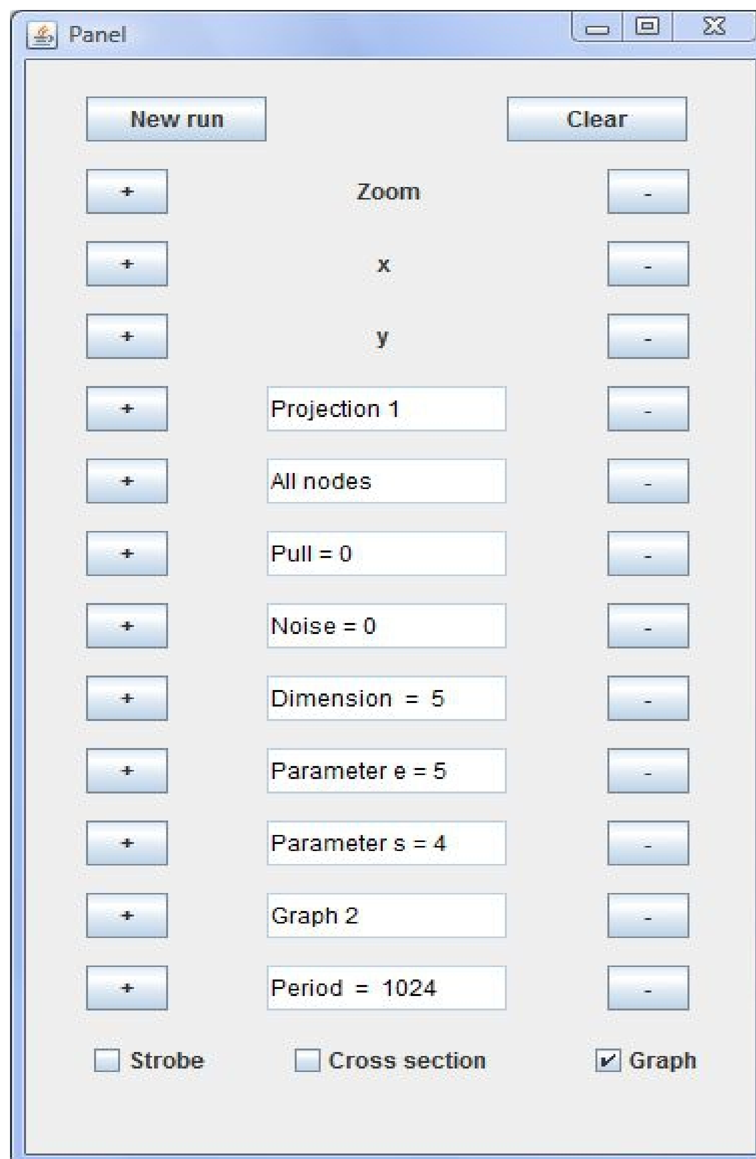

Bild 8: Benutzeroberfläche für Attractor.jar

New run
Die Anfangsbedingungen werden zufällig neu gesetzt.

Clear
Der Bildschirminhalt wird gelöscht.

Zoom
Der Bildschirminhalt wird gelöscht und um den Faktor 1,2 vergrößert neu gezeichnet oder entsprechend verkleinert.

x
Verschiebung des Bildes in x – Richtung.

y
Verschiebung des Bildes in y – Richtung.

Projection
Der Attraktor wird aus verschiedenen Richtungen auf den Bildschirm projiziert. Die Anzahl der Projektionen hängt von der gewählten Dimension ab.

All nodes
Alle Punkte (Ortsvektoren) oder nur einzelne werden gezeichnet. In diesem Beispielprogramm sind maximal 12 Punkte möglich.

Pull
Die einzelnen Punktepaare werden mit wachsender Intensität auseinandergezogen. Dies erfolgt durch einen Vektor, der in die 45° - Richtung wirkt. Dabei können andersartige Attraktoren entstehen.

Noise
Die Punkte werden mit wachsender Intensität zufällig im Raum verschoben. Mit diesem überlagerten Rauschen kann die Stabilität des Attraktors geprüft werden.

Dimension
Die Dimension n des Raums kann für n > 1 beliebig eingestellt werden.

Parameter e
Der Parameter e beschränkt den maximalen Abstand der Punkte eines Punktepaars.

Parameter s
Der Parameter s bestimmt den Abstand der Parallelen, auf denen die antiparallelen Verschiebungsvektoren eines Punktepaars liegen.

Graph i
Hier kann man vorgegebene Graphen, für die Attraktoren gefunden wurden, auswählen. Für Graph i = 1 werden neue Graphen fortlaufend zufällig erzeugt. Wird ein neuer Attraktor gefunden, stoppt die Suche automatisch.

Period = i
Es werden i Iterationen durchgeführt und dann die berechneten Pixel gezeichnet.

Strobe
Der Bildschirm wird vor jeder Periode gelöscht. Mit Period und Strobe kann das dynamische Verhalten der Attraktoren untersucht werden.

Cross section
Aus der Struktur wird eine dünne Scheibe herausgeschnitten und gezeichnet. Die Anzahl der möglichen Darstellungen hängt von der gewählten Dimension ab.

Graph
Der verwendete Graph, der die Kopplung der Knoten (Ortsvektoren) bestimmt, wird angezeigt.