Inhalt

5 Numerische Methoden der praktischen Mathematik (II)

5.5 Numerische Integration

Die numerische Integration beschäftigt sich mit der angenäherten Berechnung *bestimmter Intergrale*.

$$F = \int_a^b f(x)dx$$

Grundgedanke

Die Funktion $f(x)$ wird durch ein oder mehrere **Interpolationspolynome** $P_r^j(x)$ approximiert, *r* Grad des Polynoms. Diese werden integriert und liefern einen Näherungswert für das Integral.

Ausgangspunkt

Gegeben sind $s = n+1 \geq 2$ **äquidistante Stützstellen** $x_0, x_1, \cdots, x_n$ mit $x_0 = a$, $x_n = b$ und $x_{i+1} = x_i + h$ für $i = 0,1,\cdots,n-1$. Für diese Stützstellen sind die Funktionswerte $y_0 = f(x_0)$, $y_1 = f(x_1), \cdots, y_n = f(x_n)$ bekannt. Für den Abstand *h* zwischen zwei Stützstellen ergibt sich folglich $h = \frac{b-a}{s-1}$.

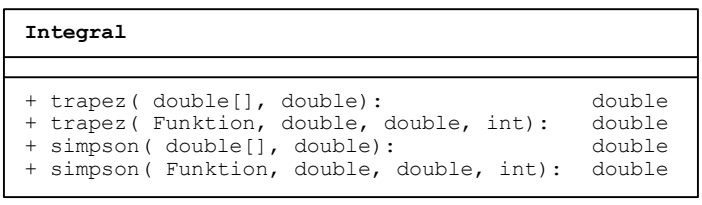
Prinzip

Durch benachbarte Punkte (x_i, y_i) werden *k + 1* Polynome gelegt, welche die Funktion in einem Bereich annähern. Durch Summieren der Integralwerte der Polynome in diesem Bereich wird ein Näherungswert für das gesuchte Integral berechnet:

$$F = \int_a^b f(x)dx \approx \sum_{j=0}^{k} \int_{a_j}^{b_j} P_r^j(x)dx \text{ mit } a_0 = a,\ b_k = b \text{ und } a_j = b_{j-1} \text{ für } 0 < j \leq k.$$

5.5.1 Klasse `Integral`

Eine Klasse `Integral` soll zur Integralberechnung zwei verschiedene numerische Verfahren (Trapezregel, Simpsonregel) aufnehmen. Ausreichend für die Interpolation sind die Stützstellen. Diese sind direkt gegeben oder werden mittels einer Funktion berechnet:



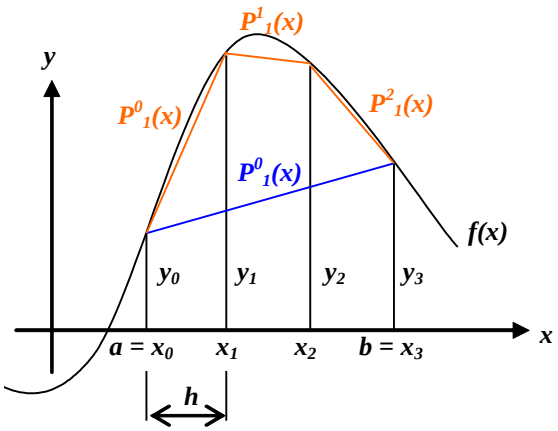
Integral.java

```
// Integral.java                                        MM 2012

/**
 * Integrieren mittels Interpolation.
 */
public class Integral
{
/**
 * Integrieren mittels linearer Interpolation
 * (Trapezregel).
 * @param y Funktionswerte aller Stuetzstellen
 * @param h Abstand zwischen den Stuetzstellen
 * @return Naeherungswert des Integrals
 */
  public double trapez( double[] y, double h){ . . . }

/**
 * Integrieren mittels linearer Interpolation
 * (Trapezregel).
 * @param fkt Funktion
 * @param a untere Integrationsgrenze
 * @param b obere Integrationsgrenze
 * @param s Anzahl der Stuetzstellen
 * @return Naeherungswert des Integrals
 */
  public double trapez
  ( Funktion fkt, double a, double b, int s)
  {
    double h = ( b - a) / ( s - 1);              // Abstand
    double[] y = fkt.tabellieren( a, s, h);

    return trapez( y, h);
  }

/**
 * Integrieren mittels quadratischer Interpolation
 * (Simpsonregel).
 * @param y Funktionswerte aller Stuetzstellen
 * @param h Abstand zwischen den Stuetzstellen
 * @return Naeherungswert des Integrals ,
 *         falls Simpsonvoraussetzung korrekt
 */
  public double simpson( double[] y, double h){ . . . }

/**
 * Integrieren mittels quadratischer Interpolation
 * (Simpsonregel).
 * @param fkt Funktion
 * @param a untere Integrationsgrenze
 * @param b obere Integrationsgrenze
 * @param s Anzahl der Stuetzstellen
```

```
   * @return Naeherungswert des Integrals ,
   *         falls Simpsonvoraussetzung korrekt
   */
    public double simpson
    ( Funktion fkt, double a, double b, int s)
    {
      double h = ( b - a) / ( s - 1);
      double[] y = fkt.tabellieren( a, s, h);

      return simpson( y, h);
    }
}
```

5.5.2 Trapezregel - lineare Interpolation

Approximation durch ein **lineares Interpolationspolynom** $P_1^j(x) = a_1 x + a_0$:



$s = 2$, $s = 4$ ⇒ je mehr Stützstellen, desto genauer die Näherung an die eigentliche Funktion.

1. **Zerlegung in Teilintervalle** $F = \int_{a=x_0}^{b=x_3} f(x)dx \approx \int_{x_0}^{x_1} P_1^0(x)dx + \int_{x_1}^{x_2} P_1^1(x)dx + \int_{x_2}^{x_3} P_1^2(x)dx$.

2. **Trapezflächenberechnung** $\int_{x_i}^{x_{i+1}} P_1^i(x)dx = \frac{h}{2}(y_i + y_{i+1})$

$$F \approx \frac{h}{2}(y_0 + y_1) + \frac{h}{2}(y_1 + y_2) + \frac{h}{2}(y_2 + y_3) = \frac{h}{2}(y_0 + 2y_1 + 2y_2 + y_3).$$

Allgemein

$[a,b]$ wird in n Teilintervalle $[x_0,x_1],[x_1,x_2],\cdots[x_{n-1},x_n]$ mit $x_0 = a$, $x_n = b$ zerlegt. In jedem dieser Teilintervalle wird die Funktion $f(x)$ durch eine Gerade angenähert. Einen Näherungswert für das bestimmte Integral der Funktion $f(x)$ erhält man durch Summieren der entsprechenden Trapezflächen:

Trapezregel: $F = \int_{a=x_0}^{b=x_n} f(x)dx \approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} P_1^i(x)dx = \frac{h}{2}\left(y_0 + 2y_1 + 2y_2 + \cdots + 2y_{n-1} + y_n\right)$.

Ergänzung in Integral.java

```
/**
 * Integrieren mittels linearer Interpolation
 * (Trapezregel).
 * @param y Funktionswerte aller Stuetzstellen
 * @param h Abstand zwischen den Stuetzstellen
 * @return Naeherungswert des Integrals
 */
  public double trapez( double[] y, double h)
  {
    int n = y.length -1;

    double ergebnis = ( y[ 0] + y[ n]) / 2;
    for( int i = 1; i < n; i++) ergebnis += y[ i];

    return h * ergebnis;
  }
```

Beispiel

Zu berechnen sei das bestimmte Integral $\int_0^1 \frac{dx}{1+x}$ durch lineare Interpolation mit sieben Stützstellen. $\Rightarrow\ n = 6,\ a = 0,\ b = 1,\ h = \frac{b-a}{n} = \frac{1}{6}$:

$0 \le i \le n$	0	1	2	3	4	5	6
$x_o = a, x_i = x_{i-1} + h$	0	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{5}{6}$	1
$y_i = f(x_i) = \frac{1}{1+x_i}$	1	$\frac{6}{7}$	$\frac{3}{4}$	$\frac{2}{3}$	$\frac{3}{5}$	$\frac{6}{11}$	$\frac{1}{2}$

Mathematische Lösung (TR): $\int_0^1 \frac{dx}{1+x} = \int_1^2 \frac{dt}{t}$ mit $t = 1 + x \Rightarrow \frac{dt}{dx} = 1$

$= \ln|t|_1^2 = \ln 2 - \ln 1 = \ln 2 - 0 = \ln 2 \approx$ **0.6931471805599453**

Trapezregel (TR): $\int_0^1 \frac{dx}{1+x} \approx \frac{1}{12}\left(1 + \frac{12}{7} + \frac{3}{2} + \frac{4}{3} + \frac{6}{5} + \frac{12}{11} + \frac{1}{2}\right) \approx$ **0.6948773448773448**

Programm *TrapezIntegralTest.java*:

7 Stützstellen **0.6948773448773449**

Das Ergebnis ist bedingt durch die *Verfahrensfehler* sehr ungenau und mit dem Taschenrechnerergebnis (bis auf die letzte Stelle) identisch.

20 Stützstellen	**0.6933202508885106**
80 Stützstellen	**0.6931571947801501**

⇒ *Je mehr Stützstellen*, desto genauer wird die Annäherung an die eigentliche Funktion, desto *besser* der Wert des Integrals.

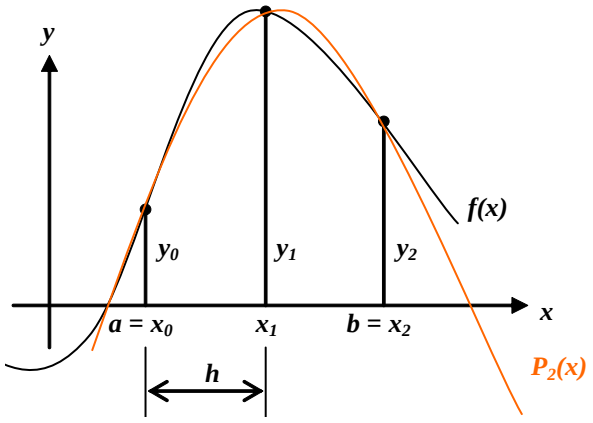
TrapezIntegralTest.java

```
// TrapezIntegralTest.java                              MM 2012

/**
 * Test der Trapezregel der Klasse Integral.
 */
public class TrapezIntegralTest
{
/**
 * Integral von 0 bis 1 ueber
 * Funktion ( 1.0 ) / ( 1.0 + 1.0 x^1 )
 */
  public static void main( String[] args)
  {
// Funktion
    Polynom zaehlerPolynom = new Polynom();
    double[] zaehler = new double[ 1];
    zaehler[ 0] = 1;
    zaehlerPolynom.setPolynom( zaehler);

    Polynom nennerPolynom = new Polynom();
    double[] nenner = new double[ 2];
    nenner[ 0] = 1;
    nenner[ 1] = 1;
    nennerPolynom.setPolynom( nenner);

    RationaleFunktion fkt = new RationaleFunktion();
    fkt.setRationaleFunktion
        ( zaehlerPolynom, nennerPolynom);

// Funktionsausgabe
    System.out.println();
    System.out.println( "f( x) = " + fkt);

// Mathematischer Wert
    System.out.println
    ( " Math.log( 2)            : " + Math.log( 2));

// Integral mit 7 Stuetzstellen
    Integral integral = new Integral();
    double erg = integral.trapez( fkt, 0, 1, 7);
    System.out.println
```

```
      ( " Trapez(  7 Stuetzstellen): " + erg);

  // Integral mit 20, 40, 60, 80 Stuetzstellen
      for( int i = 20; i < 100; i += 20)
      {
        erg = integral.trapez( fkt, 0, 1, i);
        System.out.println
        ( " Trapez( " + i + " Stuetzstellen): " + erg);
      }
    }
  }

  /* ------------------------------------------------------ */
                                          //  Testergebnisse
  /*
  f( x) = ( 1.0 ) / ( 1.0 + 1.0 x^1 )
   Math.log( 2):               0.6931471805599453
   Trapez(  7 Stuetzstellen): 0.6948773448773449
   Trapez( 20 Stuetzstellen): 0.6933202508885106
   Trapez( 40 Stuetzstellen): 0.6931882685712957
   Trapez( 60 Stuetzstellen): 0.6931651345260461
   Trapez( 80 Stuetzstellen): 0.6931571947801501
  */
```

5.5.3 Simpsonregel - quadratische Interpolation

Approximation durch Interpolation mit **Polynomen zweiten Grades** $\boxed{P_2^j(x) = a_2x^2 + a_1x + a_0}$.



1. **Parabelgleichung** $P_2(x) = a_2x^2 + a_1x + a_0$

   Gegeben sind drei äquidistante Stützstellen, durch die eine Parabel zu legen ist. Die Berechnung der Koeffizienten (a_0, a_1, a_2) der Parabel erfolgt mittels **Gauß-Eliminationsverfahren** unter Berücksichtigung, dass die Stützstellen der Funktion Punkte der Parabel sind:

$$\begin{aligned} a_0 + a_1 x_0 + a_2 x_0^2 &= y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 &= y_1 \\ a_0 + a_1 x_2 + a_2 x_2^2 &= y_2 \end{aligned} \quad \Rightarrow \quad \begin{aligned} a_0 &= y_0 - \frac{(y_1 - y_0)x_0}{h} + \frac{(y_2 + y_0 - 2y_1)x_1 x_0}{2h^2} \\ a_1 &= \frac{y_1 - y_0}{h} - \frac{y_2 + y_0 - 2y_1}{2h^2}(x_1 + x_0) \\ a_2 &= \frac{y_2 + y_0 - 2y_1}{2h^2} \end{aligned}$$

Sei $\Delta_1 = \frac{y_1 - y_0}{h}$ und $\Delta_2 = \frac{y_2 + y_0 - 2y_1}{2h^2}$.

$$\begin{aligned} P_2(x) = a_2 x^2 + a_1 x + a_0 &= \Delta_2 x^2 + (\Delta_1 - \Delta_2(x_1 + x_0))x + (y_0 - \Delta_1 x_0 + \Delta_2 x_0 x_1) \\ &= y_0 + \Delta_1(x - x_0) + \Delta_2(x - x_0)(x - x_1) \end{aligned}$$

Parabelgleichung: $P_2(x) = a_2 x^2 + a_1 x + a_0 = y_0 + \Delta_1(x - x_0) + \Delta_2(x - x_0)(x - x_1)$ mit
$\Delta_1 = \frac{y_1 - y_0}{h}$ und $\Delta_2 = \frac{y_2 + y_0 - 2y_1}{2h^2}$

2. Integralberechnung

$$\int_{x_0}^{x_2} P_2(x)dx = \int_{x_0}^{x_2} (y_0 + \Delta_1(x - x_0) + \Delta_2(x - x_0)(x - x_1))dx$$

$$\int_{x_0}^{x_2} y_0 dx = y_0 x\Big|_{x_0}^{x_2} = (x_2 - x_0)y_0 = 2hy_0$$

$$\int_{x_0}^{x_2} \Delta_1(x - x_0)dx = \Delta_1\left(\int_{x_0}^{x_2} x dx - \int_{x_0}^{x_2} x_0 dx\right) = 2h(y_1 - y_0)$$

$$\int_{x_0}^{x_2} \Delta_2(x - x_0)(x - x_1)dx = \Delta_2\left(\int_{x_0}^{x_2} x^2 dx + \int_{x_0}^{x_2} x_0 x_1 dx - \int_{x_0}^{x_2} (x_0 + x_1)x dx\right)$$

$$= \frac{1}{3}h(y_2 + y_0 - 2y_1)$$

$$\int_{x_0}^{x_2} P_2(x)dx = 2h\left(y_0 + y_1 - y_0 + \tfrac{1}{6}(y_2 + y_0 - 2y_1)\right) = \frac{h}{3}(y_0 + 4y_1 + y_2)$$

Keplersche Fassregel: $F = \int_{a=x_0}^{b=x_2} f(x)dx \approx \int_{x_0}^{x_2} P_2(x)dx = \frac{h}{3}(y_0 + 4y_1 + y_2)$.

Für 4 Teilintervalle benötigt man 2 Polynome zur Approximation der Funktion, $n = 4$:

$$F = \int_{a=x_0}^{b=x_4} f(x)dx \approx \int_{x_0}^{x_2} P_1^0(x)dx + \int_{x_2}^{x_4} P_1^1(x)dx$$
$$= \frac{h}{3}(y_0 + 4y_1 + y_2) + \frac{h}{3}(y_2 + 4y_3 + y_4)$$
$$= \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + y_4)$$

Allgemein

$[a,b]$ wird in n Teilintervalle $[x_0,x_1],[x_1,x_2],\cdots[x_{n-1},x_n]$ mit $x_0 = a$, $x_n = b$, n gerade, zerlegt. In jeweils für zwei aufeinanderfolgenden Teilintervalle wird die Keplersche Fassregel angewendet und das Integral durch Summieren der Ergebnisse näherungsweise berechnet:

Simpsonregel:

$$F = \int_{a=x_0}^{b=x_n} f(x)dx \approx \sum_{j=0}^{\frac{n-2}{2}} \int_{x_{2j}}^{x_{2j+2}} P_2^j(x)dx = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$

Ergänzung in Integral.java

```
/**
 * Integrieren mittels quadratischer Interpolation
 * (Simpsonregel).
 * @param y Funktionswerte aller Stuetzstellen
 * @param h Abstand zwischen den Stuetzstellen
 * @return Naeherungswert des Integrals,
 *         falls Simpsonvoraussetzung korrekt
 */
 public double simpson( double[] y, double h)
 {
   int n = y.length -1;
   double ergebnis = y[ 0] + y[ n];

   for( int i = 1; i < n; i += 2)
     ergebnis += 4 * y[ i];

   for( int i = 2; i < n - 1; i += 2)
     ergebnis += 2 * y[ i];

   return h / 3 * ergebnis;
 }
```

Beispiel

Berechnen des Integral $\int_0^1 \frac{dx}{1+x}$ durch quadratische Interpolation mit 7 Stützstellen (s.o.) ⇒ $n=6,\ a=0,\ b=1,\ h=\frac{1}{6}$:

Mathematische Lösung (TR): **0.693147180559453**

Simpsonregel (TR): $\int_0^1 \frac{dx}{1+x} \approx \frac{1}{18}\left(1+\frac{24}{7}+\frac{3}{2}+\frac{8}{3}+\frac{6}{5}+\frac{24}{11}+\frac{1}{2}\right) \approx$ **0.6931697931697931**

Programm *SimpsonIntegralTest.java*:

7 Stützstellen **0.6931697931697932**

Das Ergebnis ist mit dem Taschenrechnerergebnis (bis auf die letzte Stelle) identisch.

21 Stützstellen **0.6931473746651163**

81 Stützstellen **0.6931471813225872**

⇒ *Je mehr Stützstellen*, desto genauer wird die Annäherung an die eigentliche Funktion, desto *besser* der Wert des Integrals.

⇒ Das mit der Simpsonregel entwickelte *Interpolationspolynom* liefert für dieses Beispiel eine *bessere* Annäherung der Ausgangsfunktion als das der Trapezregel und damit einen *besseren* Integralwert.

SimpsonIntegralTest.java

```
// SimpsonIntegralTest.java                                MM 2012

/**
 * Test der Simpsonregel der Klasse Integral.
 */
public class SimpsonIntegralTest
{
/**
 * Integral von 0 bis 1 ueber
 * Funktion ( 1.0 ) / ( 1.0 + 1.0 x^1 )
 */
  public static void main( String[] args)
  {
// Funktion
    Polynom zaehlerPolynom = new Polynom();
    double[] zaehler = new double[ 1];
    zaehler[ 0] = 1;
    zaehlerPolynom.setPolynom( zaehler);

    Polynom nennerPolynom = new Polynom();
    double[] nenner = new double[ 2];
    nenner[ 0] = 1;
    nenner[ 1] = 1;
    nennerPolynom.setPolynom( nenner);
```

```
    RationaleFunktion fkt = new RationaleFunktion();
    fkt.setRationaleFunktion
        ( zaehlerPolynom, nennerPolynom);

// Funktionsausgabe
    System.out.println();
    System.out.println( "f( x) = " + fkt);

// Mathematischer Wert
    System.out.println
    ( " Math.log( 2)              : " + Math.log( 2));

// Integral mit 7 Stuetzstellen
    Integral integral = new Integral();
    double erg = integral.simpson( fkt, 0, 1, 7);
    System.out.println
    ( " Simpson(  7 Stuetzstellen): " + erg);

// Integral mit 21, 41, 61, 81 Stuetzstellen
    for( int i = 21; i < 100; i += 20)
    {
      erg = integral.simpson( fkt, 0, 1, i);
      System.out.println
      ( " Simpson( " + i + " Stuetzstellen): " + erg);
    }
  }
}

/* ------------------------------------------------- */
                                        //  Testergebnisse
/*
f( x) = ( 1.0 ) / ( 1.0 + 1.0 x^1 )
 Math.log( 2)              : 0.693147180559453
 Simpson(  7 Stuetzstellen): 0.6931697931697932
 Simpson( 21 Stuetzstellen): 0.6931473746651163
 Simpson( 41 Stuetzstellen): 0.6931471927479559
 Simpson( 61 Stuetzstellen): 0.6931471829695383
 Simpson( 81 Stuetzstellen): 0.6931471813225872
*/
```

Zusammenfassend wurden zwei Näherungsverfahren zur Berechnung bestimmter Integrale besprochen. Da die in den Verfahren verwendete Polynominterpolation die zu integrierende Funktion ungenau darstellt, sind die Ergebnisse mit *Verfahrensfehlern* behaftet.

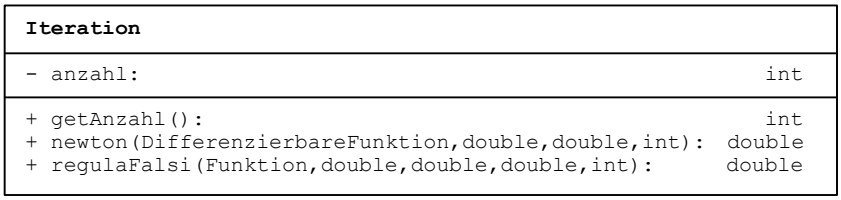
Beide Verfahren wurden in der Klasse `Integral` implementiert.

5.6 Nullstellenberechnung

Zur näherungsweisen *Bestimmung von Nullstellen stetiger Funktionen* $f(x)$ werden oft **Iterationsverfahren** verwendet. Ausgehend von *Anfangsnäherungen* für eine Nullstelle werden solange neue Näherungen berechnet, bis eine gewünschte *Genauigkeit* erreicht ist (**Iteration**). Diese Verfahren haben mit der Entwicklung der Computertechnik durch Rechengeschwindigkeit und Rechengenauigkeit an Bedeutung gewonnen.

5.6.1 Klasse `Iteration`

Eine Klasse `Iteration` soll zur Berechnung von Nullstellen einer Funktion zwei verschiedene numerische Verfahren (Newton, Regula falsi) aufnehmen. Die Anzahl der benötigten Iterationsschritte wird mitgezählt und nach einer Berechnung zur Verfügung gestellt:



Iteration.java

```
// Iteration.java                                        MM 2013

/**
 * Nullstelleniteration mittels Newton und Regula falsi.
 */
public class Iteration
{
/* ------------------------------------------------------ */
                                                  // Attribute
/**
 * Aktuelle Iterationstiefe.
 */
  private int anzahl = 0;

/* ------------------------------------------------------ */
                                                // get-Methode
/**
 * Liest Iterationstiefe.
 * @return anzahl
 */
  public int getAnzahl()
  {
    return anzahl;
  }

/* ------------------------------------------------------ */
```

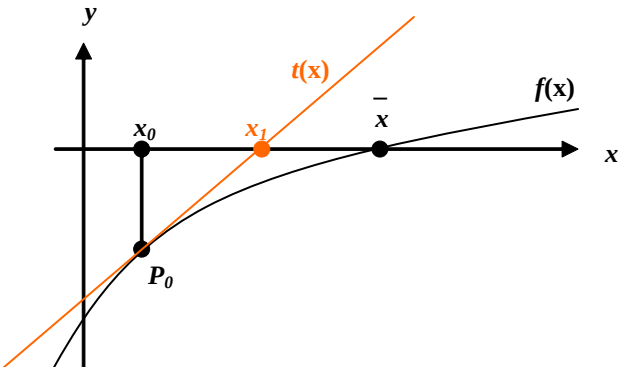
```
                                                    // Verfahren
/**
 * Nullstellenbestimmung mit Newton.
 * @param fkt differenzierbare Funktion
 * @param x0 Naeherung
 * @param eps Genauigkeit
 * @param max maximale Iterationstiefe
 * @return Naeherung einer Nullstelle,
 *         falls anzahl == max
 *         wurde die gewuenschte Genauigkeit nicht erreicht.
 */
  public double newton( DifferenzierbareFunktion fkt,
    double x0, double eps, int max){ . . . }

/**
 * Nullstellenbestimmung mit Ragula falsi.
 * @param fkt Funktion
 * @param x0 1. Naeherung (sgn( f( x0)) != sgn( f( x1)))
 * @param x1 2. Naeherung (sgn( f( x0)) != sgn( f( x1)))
 * @param eps Genauigkeit
 * @param max maximale Iterationstiefe
 * @return Naeherung einer Nullstelle,
 *         falls anzahl == max
 *         wurde die gewuenschte Genauigkeit nicht erreicht.
 */
  public double regulaFalsi( Funktion fkt,
    double x0, double x1, double eps, int max){ . . . }
}
```

5.6.2 Newton (Tangentenverfahren)

Das nach **Isaac Newton (1643-1727)** benannte Verfahren dient der Nullstellenbestimmung einer *differenzierbaren Funktion* $f(x)$.

Grundgedanke

Ausgehend von einer bereits vorhandenen Näherung x_0 wird im Punkt $P_0 = (x_0, f(x_0))$ an die Funktion $y = f(x)$ eine Tangente $t(x)$ gelegt. Der Schnittpunkt dieser Tangente mit der Abszisse liefert eine neue Näherung x_1. Der Vorgang wird so oft wiederholt, bis man eine geeignete Näherung einer Nullstelle $\bar{x}$ der Funktion $f(x)$ gefunden hat.



Ausgangspunkt

Eine *stetig differenzierbare* Funktion $f(x)$ und eine Näherung x_0.

1. Berechnen der Tangente $t(x)$ im Punkt P_0 an die Funktion $f(x)$:

Die Tangente $t(x)$ an die Funktion $y = f(x)$ durch den Punkt $P_0 = (x_0, y_0)$ mit $y_0 = f(x_0)$ wird mittels **Punktrichtungsgleichung** berechnet. Sei $y_0' = f'(x_0)$ die Steigung von $f(x)$ im Punkt P_0, so ist

$$y - y_0 = y_0'(x - x_0)$$

und

$$t(x) = y = y_0 + y_0'(x - x_0)$$

Tangente in P_0 an $f(x)$.

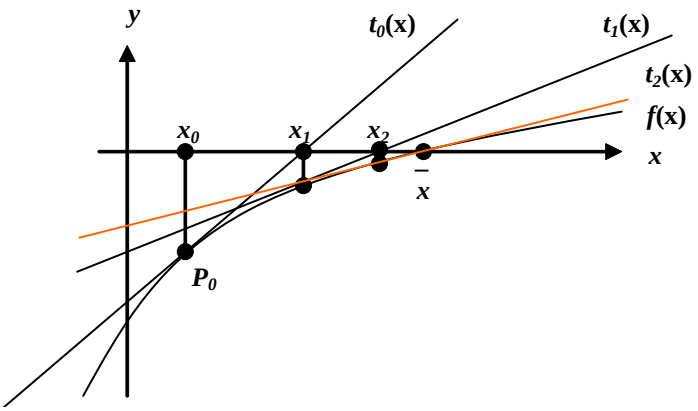
2. Berechnen der Nullstelle x_1 der Tangente $t(x)$:

$$t(x_1) = 0 \Rightarrow y_0 + y_0'(x_1 - x_0) = 0 \Rightarrow x_1 - x_0 = -\frac{y_0}{y_0'} \Rightarrow x_1 = x_0 - \frac{y_0}{y_0'}$$

$\Rightarrow$ ***Newtonnäherung:*** $\bar{x} \approx x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$.

3. Abbruch:

Ausgehend von einer Näherung x_0 wird eine Folge von Näherungen $x_1, x_2, \cdots, x_n$ ermittelt, die gegen $\bar{x}$ konvergiert ($y_i = f(x_i) \to 0$). Als Abbruchkriterium wählt man eine genügend kleine Größe $\varepsilon > 0$, so dass die Berechnung für $|f(x_i)| < \varepsilon$ abgebrochen wird.



Abbruch, falls $|f(x_i)| < \varepsilon$, ε Rechengenauigkeit.

Ergänzung in Iteration.java

```
/**
 * Nullstellenbestimmung mit Newton.
 * @param fkt differenzierbare Funktion
 * @param x0 Naeherung
 * @param eps Genauigkeit
 * @param max maximale Iterationstiefe
 * @return Naeherung einer Nullstelle,
 *         falls anzahl == max
 *         wurde die gewuenschte Genauigkeit nicht erreicht.
 */
```

```
public double newton( DifferenzierbareFunktion fkt,
  double x0, double eps, int max)
{
  double y0 = fkt.wert( x0), y1;
  anzahl = 0;
                              // Naeherung ist Nullstelle
  if( Math.abs( y0) < eps) return x0;

  do                                                 // Newton
  {
    // System.out.println( anzahl + ": " + x0);

    y1 = fkt.wertErsteAbleitung( x0);   // neue Naeherung
    x0 -= y0 / y1;
    y0 = fkt.wert( x0);

    if( y1 == 0) return x0;         // Division durch Null
    anzahl++;
                                                   // Abbruch
  } while(( Math.abs( y0) >= eps) && ( anzahl < max));

  return x0;
}
```

Da die Möglichkeit besteht, dass die Nullstellenfolge, zum Beispiel durch die Wahl eines ungünstigen Startwertes, nicht konvergiert und es zu einer unendlichen Schleife kommt, wird die Anzahl der Wiederholungen der Iteration durch eine Maximalanzahl eingeschränkt.

5.6.3 Anwendung 1 - Nullstellenbestimmung von Polynomen

Ein Polynom *r*-ten Grades hat höchstens *r* verschiedene Nullstellen. Gegeben sei ein Polynom $P_r(x) = a_r x^r + a_{r-1} x^{r-1} + \cdots + a_1 x + a_0$ und eine erste Näherung x_0. Als Ergebnis der Newtoniteration $\bar{x} \approx x_1 = x_0 - \frac{P_r(x_0)}{P_r'(x_0)}$ bekommt man in Abhängigkeit von der Startnäherung eine der *r* Nullstellen $\bar{x}_1$. Weitere Nullstellen kann man unter Umständen experimentell durch eine geeignete Änderung der Startnäherungen x_0 erhalten.

Mathematisch bietet sich ein anderes Vorgehen an. Um weitere Nullstellen $\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_r$ des Polynoms $P_r(x)$ zu berechnen, benutzt man folgendes Lösungsverfahren:
Ist $\bar{x}_1$ eine gefundene Nullstellen, so kann $P_r(x)$ mittels Polynomdivision ohne Rest durch $(x - \bar{x}_1)$ geteilt werden. Man erhält dadurch ein Polynom (*r*-1)-ten Grades und es gilt $P_r(x) = (x - \bar{x}_1) P_{r-1}(x)$. Nun bestimmt man eine Nullstelle $\bar{x}_2$ des Polynoms $P_{r-1}(x)$, welche auch Nullstelle von $P_r(x)$ ist. Das Verfahren kann fortgesetzt werden. Es determiniert, da die Anzahl der Nullstellen endlich ist.

$$P_r(x) = a_r x^r + a_{r-1} x^{r-1} + \cdots + a_1 x + a_0 = (x - \bar{x}_1)(x - \bar{x}_2) \cdots (x - \bar{x}_r)$$

Rechentechnisch besteht das Problem, dass man als Nullstellen Näherungen erhält. Bei der Division verwendet man diese fehlerbehafteten Werte. Als Divisionsergebnis erhält man ein fehlerhaftes Polynom. Es ergeben sich *Fortpflanzungsfehler* und eine weitere Nullstellenbestimmung wird sehr ungenau.

Beispiele

Das Polynom $P_3(x) = x^3 - 5x^2 + 7x - 3$ hat als Nullstelle $\bar{x}_1 = 3$. Die Division durch $(x-3)$ ergibt $P_2(x) = x^2 - 2x + 1$, so dass man das Polynom in der Form $P_3(x) = (x-3)(x^2 - 2x + 1)$ darstellen kann. Die *zweifache* Nullstelle $\bar{x}_2 = 1$ von $P_2(x)$ ist auch Nullstelle von $P_3(x)$ und es gilt $P_3(x) = (x-3)(x-1)^2$.

Ein Anwendungsprogramm soll nun unter Verwendung des Newtonverfahrens aus der Klasse `Iteration` Nullstellen für Polynome berechnen. Die Klasse `Polynom` stellt die für die Newtoniteration $\bar{x} \approx x_1 = x_0 - \frac{P_r(x_0)}{P_r'(x_0)}$ notwendigen Funktionsberechnungen zur Verfügung.

Das Programm liefert für das Polynom $P_3(x) = x^3 - 5x^2 + 7x - 3$ mit dem Startwert $x_0 = 4$ und der Genauigkeit von $\varepsilon = 10^{-15}$ nach 6 Schritten den exakten Wert der Nullstelle $\bar{x}_1 = 3$:

```
0: 4.0
1: 3.4
2: 3.1
3: 3.008695652173913
4: 3.0000746407911927
5: 3.000000005570624
 Newton (6): 3.0
```

$P_2(x) = x^2 - 2x + 1$ besitzt genau **eine Nullstelle** $\bar{x}_1 = 1$. Das Programm liefert mit dem Startwert $x_0 = 2$ und der gleichen Genauigkeit nach 25 Schritten einen genäherten Wert der Doppelnullstelle:

```
0: 2.0
1: 1.5
2: 1.25
...
22: 1.000000238418579
23: 1.0000001192092896
24: 1.0000000596046448
 Newton (25): 1.0000000298023224
```

$P_2(x) = x^2 - 2x + 0$ besitzt **zwei Nullstellen** $\bar{x}_1 = 0$ und $\bar{x}_2 = 2$. Das Programm liefert mit dem Startwert $x_0 = 1$ und unveränderter Genauigkeit als genäherten Wert ∞, da eine Division durch Null auftrat:

```
Berechnung vorzeitig abgebrochen! Ergebnis ungenau!
Newton (1000): Infinity
```

Mit den Startwerten $x_0 = 0.5$ und $x_0 = 3$ werden Näherungen beider Nullstellen gefunden.

$P_2(x) = x^2 - 2x + 2$ besitzt **keine Nullstelle**. Das Programm liefert mit dem Startwert $x_0 = 3$ und unveränderter Genauigkeit einen Wert. Dieser ist jedoch unbrauchbar, da nach 1000

Schritten keine Nullstelle mit der gewünschten Genauigkeit erreicht wurde und auch nie erreicht werden kann:

```
Berechnung vorzeitig abgebrochen! Ergebnis ungenau!
Newton (1000): -1.390804467448622
```

5.6.4 Anwendungsprogramm `PolynomNullstellen`

Das folgende Programm berechnet Nullstellen für Polynome. Es wird in Abhängigkeit des verwendeten Anfangswertes jeweils eine Näherung geliefert.

PolynomNullstellen.java

```
// PolynomNullstellen.java                                MM 2013
import Tools.IO.*;                                  // Eingaben

/**
 * Nullstellenberechner für Polynome mit Newton,
 * Anwendung der Klasse Iteration.
 */
public class PolynomNullstellen
{
/**
 * Hauptprogramm,
 * startet Dialog zwischen Nutzer und Programm.
 */
  public static void main( String[] args)
  {
// Ueberschrift
    System.out.println();
    System.out.println( "Polynomnullstellen");

// Dialog
    char weiter = 'j';

    do
    {
// Neues Polynom
      System.out.println();
      Polynom p = new Polynom();
      if( p.konsolenEingabe())           // Eingabe korrekt
      {
// Polynomausgabe
        System.out.println();
        System.out.println( "p( x) = " + p);

        do
        {
// Eingabe der Startnullstelle
          double x0
          = IOTools.readDouble( " Anfangswert x0 = ");
          System.out.println( " " + x0);
```

```
            double eps = 1e-15;                 // Genauigkeit
            int max = 1000;    // Maximalzahl der Iterationen

  // Newtoniteration
            Iteration it = new Iteration();
            double erg = it.newton( p, x0, eps, max);

            int anzahl = it.getAnzahl();
            if( anzahl == max) System.out.println
              ( " Berechnung vorzeitig abgebrochen!" +
                " Ergebnis ungenau!");

            System.out.println
            ( " Newton (" + anzahl + "): " + erg);

  // Weiter
            weiter = IOTools.readChar
                     ( "Neuer Anfangswert (j/n)? ");
          } while( weiter == 'j');
        }

        weiter = IOTools.readChar( "Neues Polynom (j/n)? ");
      } while( weiter == 'j');

  // Programm beendet
      System.out.println();
      System.out.println( "Programm beendet");
    }
  }
```

5.6.5 Anwendung 2 - Wurzelberechnung

Wurzelwerte lassen sich mittels Nullstellenbestimmung ermitteln. Zur Berechnung von $x = \sqrt[k]{a}$ mit $k \in \mathrm{N}$, $k > 1$, $a \in \mathrm{R}$, $a > 0$ betrachtet man die Funktion

$$\boxed{f(x) = x^k - a = 0}.$$

⇒ Das Problem reduziert sich damit auf die Nullstellenbestimmung dieser Funktion. $f(x) = x^k - a$ mit $f'(x) = k \cdot x^{k-1}$ in die Newtonformel eingesetzt, liefert zur Berechnung einer neuen Näherung aus einer gegebenen

$$\boxed{x_1 = x_0 - \frac{x_0^k - a}{k \cdot x_0^{k-1}}}.$$

Als Startnäherung für das Newtonverfahren nimmt man gewöhnlich den Wurzelradikand *a*.

Beispiel
Speziell für $a=4,\ k=2 \Rightarrow f(x)=x^2-4 \quad f'(x)=2\cdot x$ und dem Anfangwert $x_0=4$ ergibt sich aus dem Newtonverfahren folgende Näherungen (TR):

i	x_i	$y_i = x_i^2 - 4$	$y_i' = 2 \cdot x_i$	$x_{i+1} = x_i - \frac{y_i}{y_i'}$
0	**4**	12	8	4 - 12/8 = 2.5
1	**2.5**	2.25	5	2.5 –2.25/5 = 2.05
2	**2.05**	0.2025	4.1	2.05 – 0.2025/4.1 = 2.0006097

Die Berechnung zeigt eine Verbesserung der Startnäherung. Dieses Verfahren konvergiert sehr schnell gegen die korrekte Nullstelle.

Für die Berechnung in einem Programm betrachten wir $f(x)=x^k-a$ als Polynom. Damit können die Funktionsberechnung und die Berechnung der ersten Ableitung über die entsprechende Wertberechnung von Polynomen erfolgen.
Unser Programm bricht bei einer Genauigkeit von *eps* = 0.000000000000001 im 6. Schritt die Berechnung mit dem exakten 2.0 Wert ab. Die Zwischenergebnisse der ersten drei Schritte entsprechen denen des Taschenrechners:

```
0: 4.0
1: 2.5
2: 2.05
3: 2.000609756097561
4: 2.0000000929222947
5: 2.000000000000002
 Newton (6): 2.0
```

5.6.6 Klasse `Wurzel`

Unter Verwendung der Newtoniteration kann man Wurzelwerte bestimmen. Als Startwert der Iteration verwendet man das Argument. Damit lässt sich eine neue Funktion zur Wurzelberechnung einführen:

Wurzel	
- exponent:	int
+ setWurzel(int):	boolean
+ konsolenEingabe():	boolean
+ getExponent():	int
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String
- myWurzel(double):	double

Wurzel.java

```
// Wurzel.java                                          MM 2013
import Tools.IO.*;                                   // Eingaben

/**
 * Wurzelfunktion f(x) = k. Wurzel( x), k > 0 aus N,
 * D = W = [ 0, +unendlich[.
 */
public class Wurzel extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
                                                    // Attribute
/**
 * Wurzelexponent.
 */
  private int exponent = 2;

/* ------------------------------------------------------ */
                                                 // set-Methoden
/**
 * Setzt Wurzelexponent.
 * @param k Wurzelexponent
 * @return true, bei erfolgreichem Eintrag
 *         false, k < 1
 */
  public boolean setWurzel( int k)
  {
    if( k < 1) return false;

    exponent = k;
    return true;
  }

/**
 * Wurzelexponenteingabe ueber Konsole.
 * @return true, bei erfolgreichem Eintrag
 *         false, k < 1
 */
  public boolean konsolenEingabe()
  {
// Eingabe des Wurzelexponenten
    int k
    = IOTools.readInteger( " Wurzelexponent k ( k > 0) = ");

// Setzen des Wurzelexponenten
    return setWurzel( k);
  }

/* ------------------------------------------------------ */
                                               // service-Methode
/**
 * Liest Wurzelexponent.
 * @return exponent Wurzelexponent
```

```
   */
    public int getExponent()
    {
      return exponent;
    }

  /**
   * Berechnen eines Funktionswertes als Nullstelle
   * der Funktion x ^ k - arg.
   * @param arg Argument
   * @return f( arg) falls definiert, NaN sonst
   */
    public double wert( double arg)
    {
  // Trivialfaelle
      if( arg < 0)  return Double.NaN;
      if( arg == 0) return 0;
      if( exponent == 1)   return arg;

  // Newtoniteration
      return myWurzel( arg);
    }

  /**
   * Berechnen eines Funktionswertes als Nullstelle
   * der Funktion x ^ k - arg mittels Newton.
   * @param arg Argument
   * @return f( arg)
   */
    private double myWurzel( double arg)
    {
  // Polynom x ^ k - arg
      double[] koeff = new double[ exponent + 1];
      koeff[ exponent] = 1;
      koeff[ 0] = -arg;
      for( int i = 1; i < exponent; i++) koeff[ i] = 0;

      Polynom polynom = new Polynom();
      polynom.setPolynom( koeff);
      // System.out.println( "Polynom " + polynom);

  // Newtoniteration
      Iteration it = new Iteration();
      return it.newton( polynom, arg, 1e-15, 1000);
    }

  /**
   * Berechnen der ersten Ableitung
   * f'(x) = ( k. Wurzel aus x) / ( x * k).
   * @param arg Argument
   * @return f'( arg)
   */
```

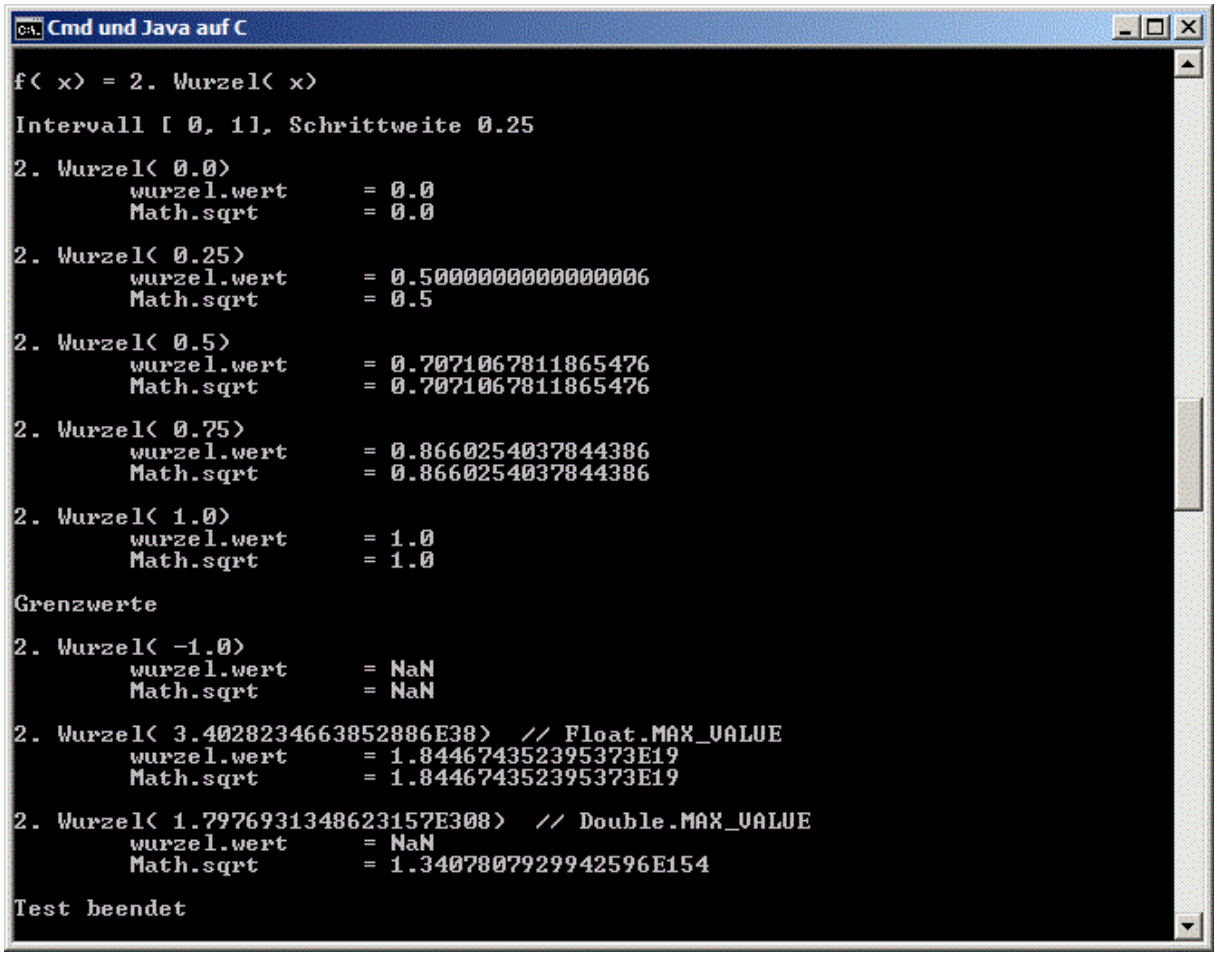
```
  public double wertErsteAbleitung( double arg)
  {
// Trivialfall
    if( arg == 0) return Double.NaN;

// Sonst
    return wert( arg) / ( arg * exponent);
  }

/* ------------------------------------------------- */
                                        // toString-Methode
/**
 * Darstellen der Wurzelfunktion.
 * @return Funktion in linearer Schreibweise
 */
  public String toString()
  {
    return "" + exponent + ". Wurzel( x)";
  }
}
```

Das Testprogramm berechnet die Funktion $f(x)=\sqrt[k]{x}$ für $x\in[0,1]$ mit der Schrittweite 0.25 und vergleicht die Ergebnisse der Iteration mit denen, die `Math.sqrt( x)` liefert.

WurzelTest.java

```
// WurzelTest.java                                    MM 2013

/**
 * Test der Klasse Wurzel.
 */
public class WurzelTest
{
/**
 * Berechnet die 2.Wurzel( x) im Intervall [ 0, 1]
 * mit der Schrittweite 0.25 und Grenzwerte,
 * vergleicht Ergebnis mit Math.sqrt( x).
 */
  public static void main( String[] args)
  {
// Neue Wurzel
    Wurzel wurzel = new Wurzel();
    wurzel.setWurzel( 2);

// Wurzelausgabe
    System.out.println();
    System.out.println( "f( x) = " + wurzel);

// Wertetabelle
    System.out.println();
```

```
      System.out.println
      ( "Intervall [ 0, 1], Schrittweite 0.25");

      double x;

      for( x = 0; x <= 1; x += 0.25)
      {
        System.out.println();
        System.out.println( "2. Wurzel( " + x + ")");
        System.out.println
        ( "\twurzel.wert   \t= " + wurzel.wert( x));
        System.out.println
        ( "\tMath.sqrt     \t= " + Math.sqrt( x));
      }

  // Grenzwerte
      System.out.println();
      System.out.println( "Grenzwerte");

  // x < 0
      x = -1;
      System.out.println();
      System.out.println( "2. Wurzel( " + x + ")");
      System.out.println
      ( "\twurzel.wert   \t= " + wurzel.wert( x));
      System.out.println
      ( "\tMath.sqrt     \t= " + Math.sqrt( x));

  // x sehr gross
      x = Float.MAX_VALUE;
      System.out.println();
      System.out.println
      ( "2. Wurzel( " + x + ")   // Float.MAX_VALUE");
      System.out.println
      ( "\twurzel.wert   \t= " + wurzel.wert( x));
      System.out.println
      ( "\tMath.sqrt     \t= " + Math.sqrt( x));

      x = Double.MAX_VALUE;
      System.out.println();
      System.out.println
      ( "2. Wurzel( " + x + ")  // Double.MAX_VALUE");
      System.out.println
      ( "\twurzel.wert   \t= " + wurzel.wert( x));
      System.out.println
      ( "\tMath.sqrt     \t= " + Math.sqrt( x));

  // Programm beendet
      System.out.println();
      System.out.println( "Test beendet");
    }
  }
```

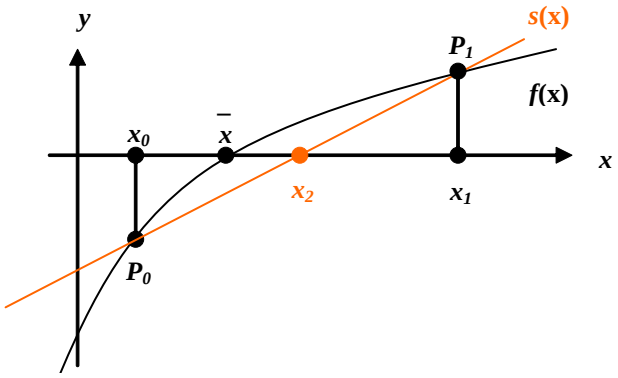

Cmd und Java auf C
f(x) = 2. Wurzel(x)
Intervall [0, 1], Schrittweite 0.25
2. Wurzel(0.0)
wurzel.wert = 0.0
Math.sqrt = 0.0
2. Wurzel(0.25)
wurzel.wert = 0.5000000000000006
Math.sqrt = 0.5
2. Wurzel(0.5)
wurzel.wert = 0.7071067811865476
Math.sqrt = 0.7071067811865476
2. Wurzel(0.75)
wurzel.wert = 0.8660254037844386
Math.sqrt = 0.8660254037844386
2. Wurzel(1.0)
wurzel.wert = 1.0
Math.sqrt = 1.0
Grenzwerte
2. Wurzel(-1.0)
wurzel.wert = NaN
Math.sqrt = NaN
2. Wurzel(3.4028234663852886E38) // Float.MAX_VALUE
wurzel.wert = 1.844674352395373E19
Math.sqrt = 1.844674352395373E19
2. Wurzel(1.797693134862315 7E308) // Double.MAX_VALUE
wurzel.wert = NaN
Math.sqrt = 1.3407807929942596E154
Test beendet

5.6.7 Regula falsi (Sekantenverfahren)

Mit dem Sekantenverfahren können Nullstellen *stetiger Funktionen* $f(x)$ berechnet werden.

Grundgedanke

Ausgehend von zwei bereits vorhandenen Näherungen x_0, x_1 mit $y_0 = f(x_0) < 0$ und $y_1 = f(x_1) > 0$, wird durch die Punkte $P_0 = (x_0, y_0)$ und $P_1 = (x_1, y_1)$ eine Sekante $s(x)$ gelegt. Der Schnittpunkt dieser Sekante mit der Abzisse liefert eine neue Näherung x_2. Der Vorgang wird so oft wiederholt, bis man eine geeignete Näherung einer Nullstelle $\bar{x}$ gefunden hat.



Ausgangspunkt

Eine *stetige* Funktion $f(x)$ und zwei Nullstellennäherung x_0 und x_1 mit $y_0 < 0$ und $y_1 > 0$.

1. **Berechnung der Sekante $s(x)$ von der Funktion $f(x)$ durch die Punkte P_0 und P_1:**
   Die Sekante $s(x)$ von $y = f(x)$ durch die Punkte $P_0 = (x_0, y_0)$ und $P_1 = (x_1, y_1)$ wird mittels **Zweipunktegleichung** berechnet, so ist

   $$(y - y_0) = m(x - x_0) \text{ mit } m = \frac{y_1 - y_0}{x_1 - x_0}$$

   und

   $$s(x) = y = y_0 + m(x - x_0)$$

   Sekante von $y = f(x)$ durch P_0 und P_1.

2. **Berechnung der Nullstelle x_2 der Sekante $s(x)$:**

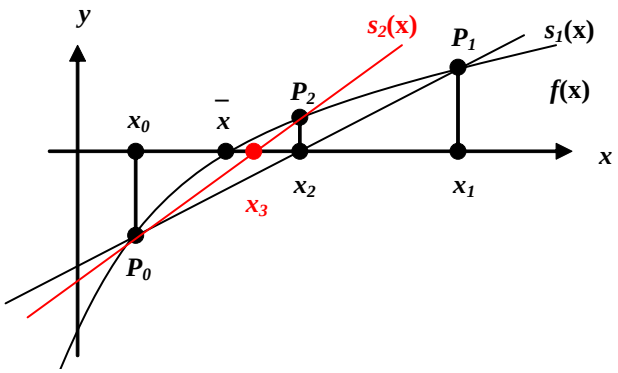
   $$s(x_2) = 0 \Rightarrow y_0 + m(x_2 - x_0) = 0 \Rightarrow x_2 = x_0 - \frac{y_0}{m}$$

   $\Rightarrow$

   Regula falsi: $\quad \bar{x} \approx x_2 = x_0 - y_0 \dfrac{x_1 - x_0}{y_1 - y_0}$.

3. **Abbruch:**

Ausgehend von zwei Näherung x_0, x_1 wird eine Folge von Näherungen $x_1, x_2, \cdots, x_n$ berechnet, die gegen $\bar{x}$ konvergiert ($y_i = f(x_i) \to 0$). Als Abbruchkriterium wählt man eine genügend kleine Größe $\varepsilon > 0$, so dass das Verfahren für $|f(x_i)| < \varepsilon$ beendet wird.



Abbruch falls $|f(x_i)| < \varepsilon$, ε Rechengenauigkeit.

Der Vorteil dieses Verfahrens gegenüber der Newtoniteration besteht darin, dass man auf die Berechnung der ersten Ableitung verzichten kann. Der Nachteil besteht in der langsameren Konvergenz.

Ergänzung in Iteration.java

```
/**
 * Nullstellenbestimmung mit Ragula falsi.
 * @param fkt Funktion
 * @param x0 1. Naeherung (sgn( f( x0)) != sgn( f( x1)))
 * @param x1 2. Naeherung (sgn( f( x0)) != sgn( f( x1)))
 * @param eps Genauigkeit
 * @param max maximale Iterationstiefe
 * @return Naeherung einer Nullstelle,
 *         falls anzahl == max
 *         wurde die gewuenschte Genauigkeit nicht erreicht.
 */
 public double regulaFalsi( Funktion fkt,
   double x0, double x1, double eps, int max)
 {
   double y0 = fkt.wert( x0);
   double y1 = fkt.wert( x1);
                                          // Vorzeichentest
   if( Math.signum( y0) == Math.signum( y1))
   {
     anzahl = max;
     return x0;
   }

   anzahl = 0;
                              // Naeherungen ist Nullstelle
   if( Math.abs( y0) < eps) return x0;
   if( Math.abs( y1) < eps) return x1;

   do                                       // Regula falsi
   {
     // System.out.print  ( " x0: " + x0);
     // System.out.println( "\tx1: " + x1);
```

```
                                                  // neue Naeherung
      double x = x0 - ( x1 - x0) / ( y1 - y0) * y0;
      if( y1 == y0) return x;          // Division durch Null

      double y = fkt.wert( x);
                                                 // Vorzeichen
      if( Math.signum( y) == Math.signum( y1))
      {
        x1 = x0;
        y1 = y0;
      }
      x0 = x;
      y0 = y;

      anzahl++;                                    // Abbruch
    } while(( Math.abs( y0) >= eps) && ( anzahl < max));

    return x0;
  }
```

5.6.8 Anwendung 1 - Nullstellenbestimmung von Polynomen

Das Polynom $P_3(x) = x^3 - 5x^2 + 7x - 3$ hat als Nullstellen $\bar{x}_1 = 3$ und $\bar{x}_2 = 1$ und es gilt: $P_3(x) = (x-3)(x^2 - 2x + 1) = (x-3)(x-1)^2$.
Das Verfahren liefert mit dem Startwerten $x_0 = 4$ und $x_1 = 2$ und der Genauigkeit von *eps* = 0.000000000000001 nach 62 Schritten einen Näherungswert der Nullstelle $\bar{x}_1 = 3$:

```
x0: 4.0                    x1: 2.0
x0: 2.2                    x1: 4.0
x0: 2.404255319148936      x1: 4.0
x0: 2.588498402555911      x1: 4.0
x0: 2.7345021468888673     x1: 4.0
x0: 2.837659798424266      x1: 4.0
...
x0: 2.999999999999997      x1: 4.0
x0: 2.9999999999999982     x1: 4.0
x0: 2.999999999999999      x1: 4.0
Regula falsi(62): 2.9999999999999996
```

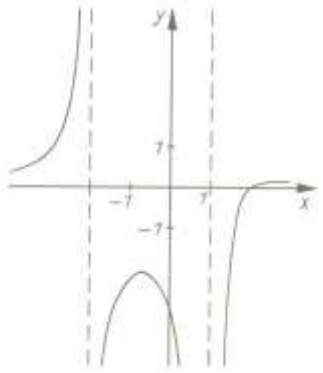
Das Restpolynom $P_2(x) = x^2 - 2x + 1$ lässt sich nicht mit Regula falsi berechnen, da die Anfangsbedingung für die Startnäherungen x_0, x_1 mit $y_0 = f(x_0) < 0$ und $y_1 = f(x_1) > 0$ nicht erfüllt werden kann, denn für alle $x \in R$ gilt: $P_2(x) = x^2 - 2x + 1 \geq 0$.

Da Polynome stetig differenzierbar sind, sollte man grundsätzlich das schneller konvergierende Newtonverfahren verwenden.

5.6.9 Anwendung 2 – Nullstellenbestimmung rationaler Funktionen

Gebrochen rationale Funktionen allgemein sind in der Regel nicht differenzierbar. Das Newtonverfahren ist nicht anwendbar. Die Berechnung der Nullstellen erfolgt deshalb grundsätzlich mit dem Sekantenverfahren.

Betrachtet man zum Beispiel die rechts grafisch dargestellte Funktion $f(x)=(3(x-2))/((x-1)^2(x+2))=(3x-6)/(x^3-3x+2)$.
Diese Funktion hat zwei Polstellen. Bei geeigneten Startwerten findet man in der Regel die Nullstelle, solange man während der Berechnung nicht auf eine der Pole stößt.



Die Anfangswerte liegen auf dem Kurvenstück der Nullstelle:

```
Anfangswert x0 =  1.5
Anfangswert x1 =  2.5
Regula falsi (138): 2.0000000000000013
```

Die Anfangswerte liegen auf benachbarten Kurvenstücken, erreichen trotzdem die Nullstelle:

```
Anfangswert x0 =  -1.0
Anfangswert x1 =  2.5
Regula falsi (7): 2.0
```

Die Anfangswerte laufen in einen der Pole (hier `-2`).

```
Anfangswert x0 =  -2.5
Anfangswert x1 =  1.5
Regula falsi (115): NaN
```

5.6.10 Anwendungsprogramm `RationaleFunktionNullstellen`

Das folgende Programm berechnet Nullstellen für rationale Funktionen allgemein. Bei der Nullstellenbestimmung von Polynomen entspricht dem Zählerpolynom das zu berechnende Polynom. Das Nennerpolynom wird auf 1 gesetzt.

RationaleFunktionNullstellen.java

```
// RationaleFunktionNullstellen.java                MM 2013
import Tools.IO.*;                                // Eingaben

/**
 * Nullstellenberechner für rationale Funktionen
 * mit Regula falsi,
 * Anwendung der Klasse Iteration.
 */
public class RationaleFunktionNullstellen
{
/**
 * Hauptprogramm,
 * startet Dialog zwischen Nutzer und Programm.
 */
  public static void main( String[] args)
  {
```

```
// Ueberschrift
    System.out.println();
    System.out.println( "Nullstellen rationaler Funktionen");

// Dialog
    char weiter = 'j';
    do
    {
// Neue Funktion
      System.out.println();
      RationaleFunktion f = new RationaleFunktion();
      if( f.konsolenEingabe())           // Eingabe korrekt
      {
// Funktionsausgabe
        System.out.println();
        System.out.println( "f( x) = " + f);

        do
        {
// Eingabe der Startnullstellen
          double x0
          = IOTools.readDouble( " Anfangswert x0 = ");
          System.out.println( " " + x0);

          double x1
          = IOTools.readDouble( " Anfangswert x1 = ");
          System.out.println( " " + x1);

          if( Math.signum( f.wert( x0))
              != Math.signum( f.wert( x1)))
          {
            double eps = 1e-15;              // Genauigkeit
            int max = 1000;  // Maximalzahl der Iterationen

// Newtoniteration
            Iteration it = new Iteration();
            double erg
            = it.regulaFalsi( f, x0, x1, eps, max);

            int anzahl = it.getAnzahl();
            if( anzahl == max) System.out.println
              ( " Berechnung vorzeitig abgebrochen!" +
                " Ergebnis ungenau!");

            System.out.println
            ( " Regula falsi (" + anzahl + "): " + erg);
          }
          else
          {
            System.out.println
            ( " Berechnung mit Regla falsi nur moeglich,");
            System.out.println
```

```
            ( " falls signum( f( x0)) != signum( f( x1)).");
          }

// Weiter
          weiter
          = IOTools.readChar( "Neue Anfangswerte (j/n)? ");
        } while( weiter == 'j');
      }

      weiter = IOTools.readChar( "Neue Funktion (j/n)? ");
    } while( weiter == 'j');

// Programm beendet
    System.out.println();
    System.out.println( "Programm beendet");
  }
}
```

5.6.11 Funktionsberechnung als Nullstellenproblem

Die Wurzelfunktion können wir als Nullstellenproblem eines Polynoms lösen und setzen im Programm die Polynomnullstellenberechnung ein.

Andere Funktionen lassen sich als Nullstellenprobleme ihrer Umkehrfunktionen bestimmen:

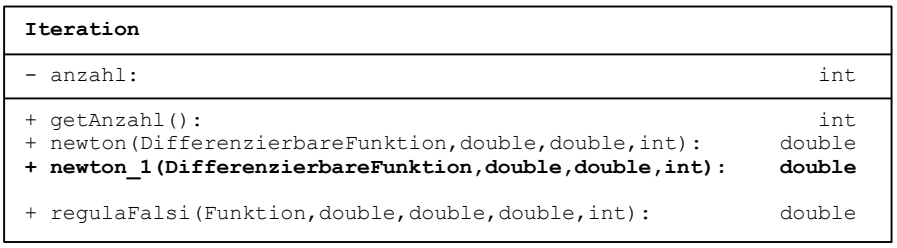
Zur Berechnung der Funktionswerte einer Funktion $x = g(a)$ betrachtet man das Nullstellenproblem der Umkehrfunktion $f(x) = g^{-1}(x) - a = 0$.

Hier einige Beispiele, in denen die Umkehrfunktion stetig differenzierbar ist und das Nullstellenproblem mit dem Newton-Verfahren gelöst werden kann:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{g^{-1}(x_0) - a}{\left(g^{-1}\right)'(x_0)}$$

Funktion		Nullstellenproblem		Newton	
Wurzel	$x = \sqrt[k]{a}$	$f(x) = x^k - a$	$f'(x) = k \cdot x^{k-1}$	$x_1 = x_0 - \frac{x_0^k - a}{k \cdot x_0^{k-1}}$	$x_1 = x_0 - \frac{P_k(x_0)}{P_k'(x_0)}$
Natürlicher Logarithmus	$x = \ln a$	$f(x) = e^x - a$	$f'(x) = e^x$	$x_1 = x_0 - \frac{e^{x_0} - a}{e^{x_0}}$	Verbesserung notwendig
Arcussinus	$x = \arcsin a$, $\lvert a \rvert < 1$	$f(x) = \sin x - a$	$f'(x) = \cos x$	$x_1 = x_0 - \frac{\sin x_0 - a}{\cos x_0}$	
Arcustangens	$x = \arctan a$	$f(x) = \tan x - a$	$f'(x) = 1 + \tan^2 x$	$x_1 = x_0 - \frac{\tan x_0 - a}{1 + \tan^2 x_0}$	Verbesserung notwendig

Eine Aufnahme einer neuen Methode `newton_1` zur Nullstellenberechnung mittels Newton für Funktionen des Typs $f(x) = g^{-1}(x) - a$ in die Klasse `Iteration` ermöglicht zahlreiche weitere Berechnungen von Funktionen in unserer Funktionenhierarchie:



Ergänzung in Iteration.java

```
/**
 * Nullstellenbestimmung mit Newton
 * fuer Wertberechnung einer Funktion
 * mittels ihrer Umkehrfunktion.
 * @param fkt differenzierbare Umkehrfunktion
 * @param arg Argument, wird als 1. Naeherung verwendet
 * @param eps Genauigkeit
 * @param max maximale Iterationstiefe
 * @return Naeherung einer Nullstelle,
 *         falls anzahl == max
 *         wurde die gewuenschte Genauigkeit nicht erreicht.
 */
 public double newton_1( DifferenzierbareFunktion fkt,
   double arg, double eps, int max)
 {
   double x0 = arg, y0 = fkt.wert( x0) - arg, y1;
   anzahl = 0;
                                // Naeherung ist Nullstelle
   if( Math.abs( y0) < eps) return x0;

   do                                              // Newton
   {
     // System.out.println( anzahl + ": " + x0);

     y1 = fkt.wertErsteAbleitung( x0);   // neue Naeherung
     x0 -= y0 / y1;
     y0 = fkt.wert( x0) - arg;

     if( y1 == 0) return x0;        // Division durch Null
     anzahl++;
                                                 // Abbruch
   } while(( Math.abs( y0) >= eps)&& ( anzahl < max));

   return x0;
 }
```

Hier als Beispiel die Berechnung des natürlichen Logarithmus als Nullstellenproblem der Exponentialfunktion.

Ln.java

```
// Ln.java                                                MM 2013

/**
 * Logarithmusfunktion, f(x) = ln( x), Basis e,
 * D = ] 0, +unendlich[, W = R.
 */
public class Ln extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
                                          // service-Methode
/**
 * Berechnen des Funktionswertes.
 * @param arg Argument
 * @return ln( arg)
 */
  public double wert( double arg)
  {
// Trivialfaelle
    if( arg < 0)  return Double.NaN;
    if( arg == 0) return Double.NEGATIVE_INFINITY;
    if( arg == 1) return 0;

// Newtoniteration
    return myLn( arg);
  }

/**
 * Berechnen eines Funktionswertes als Nullstelle der
 * Umkehrfunktion e ^ x - arg mittels Newton.
 * @param arg Argument
 * @return ln( arg)
 */
  private double myLn( double arg)
  {
    Iteration it = new Iteration();
    Exp exp = new Exp();

    return it.newton_1( exp, arg, 1e-15, 1000);
  }

/**
 * Berechnen der ersten Ableitung f'(x) = 1 / x.
 * @param arg Argument
 * @return f'( arg)
 */
  public double wertErsteAbleitung( double arg)
```

```
  {
// Trivialfall
    if( arg == 0) return Double.POSITIVE_INFINITY;

// Sonst
    return 1 / arg;
  }

/* ------------------------------------------------- */
                                        // toString-Methode
/**
 * Darstellen der Funktion ln.
 * @return Funktion in linearer Schreibweise
 */
  public String toString()
  {
    return "ln( x)";
  }
}
```

Für große x ($x \geq 710$) liefert die Berechnung kein korrektes Ergebnis. Deshalb ist auch hier eine Verbesserung des Algorithmus notwendig.

Weitere Funktionen lassen sich natürlich auch durch Anwendung der aus der Mathematik bekannten Grundbeziehungen berechnen, wie zum Beispiel:

$$\log_b x = \frac{\ln x}{\ln b} \text{ für } b \in R, b > 0, b \neq 1, x > 0$$
$$b^x = e^{x \ln b} \text{ für } b \in R, b > 0, b \neq 1$$
$$\arccos(x) = \frac{\pi}{2} - \arcsin(x) \text{ für } |x| < 1$$

Hierzu als Beispiel die Berechnung des allgemeinen Logarithmus:
Unter Verwendung des Nullstellenproblems wurde in der oben eingeführten der Klasse `Ln` der natürlichen Logarithmus berechnet. Damit kann unter Verwendung der mathematischen Beziehung $\log_b x = \frac{\ln x}{\ln b}$ der Logarithmus beliebiger Basis b ermittelt werden.

LogB.java

```
// LogB.java                                          MM 2013
import Tools.IO.*;                                 // Eingaben

/**
 * Logarithmusfunktion f(x) = logb( x),
 * Basis b aus R, b > 0, b >< 1,
 * D = ] 0, +unendlich[, W = R.
```

```
 */
public class LogB extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
                                                  // Attribute
/**
 * Basis b.
 */
  private double b = 2;

/* ------------------------------------------------------ */
                                                // set-Methode
/**
 * Setzt b fuer logb( x).
 * @param basis Basis b
 * @return true, bei erfolgreichem Eintrag
 *         false, b <= 0 oder b == 1
 */
  public boolean setBasis( double basis)
  {
    if( b <= 0 || b == 1) return false;
    b = basis;
    return true;
  }

/**
 * Basiseingabe ueber Konsole.
 * @return true, bei erfolgreichem Eintrag
 *         false, b <= 0 oder b == 1
 */
  public boolean konsolenEingabe()
  {
// Eingabe der Geradenkoeffizienten
    System.out.println( "Basiseingabe");

    double b = IOTools.readDouble
       ( " Basis b ( b > 0, b >< 1) = ");

    return setBasis( b);
  }

/* ------------------------------------------------------ */
                                            // service-Methode
/**
 * Berechnen eines Funktionswertes der Funktion
 * logb( x) = ln( x) / ln( b).
 * @param arg Argument
 * @return f( arg)
 */
  public double wert( double arg)
  {
// Trivialfaelle
```

```
      if( arg < 0)  return Double.NaN;
      if( arg == 0)
        if( b > 1) return Double.NEGATIVE_INFINITY;
        else return Double.POSITIVE_INFINITY;
      if( arg == 1) return 0;

  // Sonst
      Ln ln = new Ln();
      return ln.wert( arg) / ln.wert( b);
    }

  /**
   * Berechnen der ersten Ableitung
   * f'( x) = 1 / ( x * ln( b)).
   * @param arg Argument
   * @return f'( arg)
   */
    public double wertErsteAbleitung( double arg)
    {
  // Trivialfall
      if( arg == 0) return Double.POSITIVE_INFINITY;

  // Sonst
      Ln ln = new Ln();
      return 1 / (arg * ln.wert( b));
    }

  /* ------------------------------------------------------ */
                                          // toString-Methode
  /**
   * Darstellen der Logarithmusfunktion.
   * @return Funktion in linearer Schreibweise
   */
    public String toString()
    {
      return "log( x) zur Basis " + b;
    }
  }
```

Weitere Beispiele finden Sie im Netz.