Inhalt

5 Numerische Methoden der praktischen Mathematik (II) 5-2
5.5 Numerische Integration 5-2
5.5.1 Klasse `Integral` 5-2
5.5.2 Trapezregel - lineare Interpolation 5-4
5.5.3 Simpsonregel - quadratische Interpolation 5-7
5.6 Nullstellenbestimmung 5-13
5.6.1 Newton (Tangentenverfahren) 5-13
5.6.2 Klasse `NewtonIteration` 5-14
5.6.3 Anwendung 1 - Wurzelberechnung 5-16
5.6.4 Klasse `Wurzel` 5-17
5.6.5 Anwendung 2 - Nullstellenbestimmung von Polynomen 5-21
5.6.6 Klasse `PolynomNullstellen` 5-22
5.6.7 Regula falsi (Sekantenverfahren) 5-23
5.6.8 Klasse `RegulaFalsiIteration` 5-25
5.6.9 Anwendung 1 - Wurzelberechnung 5-27
5.6.10 Anwendung 2 - Nullstellenbestimmung von Polynomen 5-28
5.6.11 Umkehrfunktionen als Nullstellenproblem 5-30

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 $s = n+1$ **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.

Prinzip

Durch benachbarte Punkte (x_i, y_i) werden 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 (*k* ... Anzahl der Polynome):

$$F = \int_a^b f(x)dx \approx \sum_{j=0}^{k} \int_{a_j}^{b_j} P_r^j(x)dx \text{ mit } a_0 = a \text{ und } b_k = b$$

5.5.1 Klasse `Integral`

Die folgende abstrakte Klasse beschreibt allgemein die Voraussetzungen für diese Verfahren:

Integral	
- f:	Funktion
- a:	double
- b:	double
- s:	int
+ setIntegral(Funktion,double,double,int):	void
+ getStuetzStellen():	double[]
+ getAbstand():	double
+ integrieren():	*double*

Integral.java

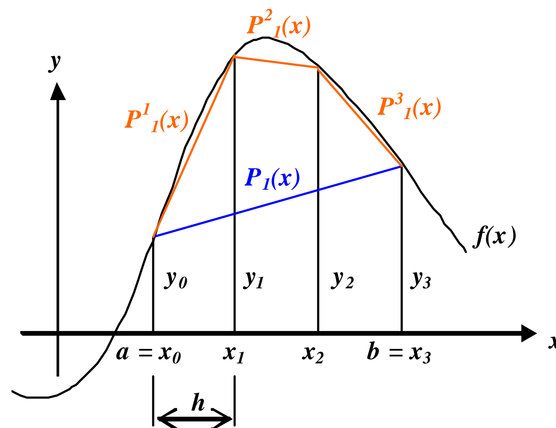
```
// Integral.java                                    MM 2010

/**
 * Integration mittels Interpolation, abstrakt.
 */
public abstract class Integral
{
```

```
/* -------------------------------------------------- */
                                        // Attribute
/**
 * Funktion.
 */
 private Funktion f = null;

/**
 * Untere Grenze.
 */
 private double a = 0;

/**
 * Obere Grenze.
 */
 private double b = 0;

/**
 * Anzahl der Stuetzstellen.
 */
 private int s = 1;

/* -------------------------------------------------- */
                                       // set-Methode
/**
 * Setzt Integral.
 * @param funktion Funktion
 * @param untereGrenze untere Integrationsgrenze
 * @param obereGrenze obere Integrationsgrenze
 * @param anzahl Anzahl der Stuetzstellen
 */
 public void setIntegral( Funktion funktion,
   double untereGrenze, double obereGrenze, int anzahl)
 {
   f = funktion;
   a = untereGrenze;
   b = obereGrenze;
   s = anzahl;
 }

/* -------------------------------------------------- */
                                       // get-Methode
/**
 * Berechnet Stuetzstellen.
 * @return Funktionswerte aller Stuetzstellen
 */
 public double[] getStuetzStellen()
 {
   return f.tabellieren( a, s, ( b - a) / ( s - 1));
 }

/**
```

```
     * Berechnet Abstand zwischen den Stuetzstellen.
     * @return h
     */
      public double getAbstand()
      {
        return ( b - a) / ( s - 1);
      }

   /* ------------------------------------------------ */
                                    // service-Methoden
   /**
     * Integrieren, abstract.
     * @return Wert des Integrals
     */
      public abstract double integrieren();
   }
```

5.5.2 Trapezregel - lineare Interpolation

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



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

1. **Zerlegung in Teilintervalle** $F = \int_{a=x_0}^{b=x_3} f(x)dx \approx \int_{x_0}^{x_1} P_1^1(x)dx + \int_{x_1}^{x_2} P_1^2(x)dx + \int_{x_2}^{x_3} P_1^3(x)dx$.

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:

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

TrapezIntegral.java

```
/**
 * Integration mittels linearer Interpolation
 * (Trapezregel).
 */
public class TrapezIntegral extends Integral
{
/* ------------------------------------------------- */
                                    // service-Methode
/**
 * Integrieren mittels Trapezregel,
 * h/2 ( y[0] + 2y[1] + ... + 2y[n-1] + y[n]).
 * @return Wert des Integrals
 */
  public double integrieren()
  {
    double[] stuetzStellen = getStuetzStellen();
    int n = stuetzStellen.length - 1;
    double h = getAbstand();

    double ergebnis
    = ( stuetzStellen[ 0] + stuetzStellen[ n]) / 2;

    for( int i = 1; i < n; i++)
      ergebnis += stuetzStellen[ 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
$y_i = f(x_i)$	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 $t = 1 + x \Rightarrow \frac{dt}{dx} = 1$

$= \ln|t|\Big|_1^2 = \ln 2 - \ln 1 = \ln 2 - 0 = \ln 2 \approx$ **<u>0.6931</u>471805599453**

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 *TrapezIntegral.java*:

7 Stützstellen **0.6948773448773449**

Das Ergebnis ist durch bedingt 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.

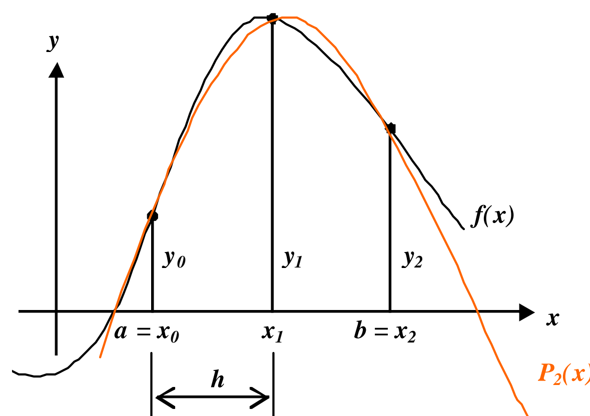
TrapezIntegralTest.java

```
// TrapezIntegralTest.java                              MM 2010

/**
 * Test der Klasse TrapezIntegral
 */
public class TrapezIntegralTest
{
/**
 * Integral von 0 bis 1 ueber
 * Funktion ( 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
 */
  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( "Funktion " + fkt);
        System.out.println
        ( " Math.log( 2):              " + Math.log( 2));

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

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

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:

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

$$\begin{aligned} P_2(x) &= a_2 x^2 + a_1 x + a_0 \\ &= \left(\frac{y_2 + y_0 - 2y_1}{2h^2} \right) x^2 \\ &+ \left(\frac{y_1 - y_0}{h} - \frac{y_2 + y_0 - 2y_1}{2h^2}(x_1 + x_0) \right) x \\ &+ \left(y_0 - \frac{(y_1 - y_0)x_0}{h} + \frac{(y_2 + y_0 - 2y_1)x_1 x_0}{2h^2} \right) \\ &= y_0 + \frac{y_1 - y_0}{h}(x - x_0) + \frac{y_2 + y_0 - 2y_1}{2h^2}(x - x_0)(x - x_1) \\ &= y_0 + \Delta_1 (x - x_0) + \Delta_2 (x - x_0)(x - x_1) \end{aligned}$$

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

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

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

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

$$\begin{aligned} \int_{x_0}^{x_2} \Delta_2 (x - x_0)(x - x_1) dx &= \Delta_2 \left(\int_{x_0}^{x_2} x^2 dx + \int_{x_0}^{x_2} x_0 x_1 dx - \int_{x_0}^{x_2} (x_0 + x_1) x dx \right) \\ &= \frac{1}{3} h \Delta'' \text{ mit } \Delta'' = (y_2 + y_0 - 2y_1) \end{aligned}$$

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

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

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:

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

Simpsonregel:

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

SimpsonIntegral.java

```java
// SimpsonIntegral.java                                    MM 2010

/**
 * Integration mittels quadratischer Interpolation
 * (Simpsonregel) bei ungerader Stuetzstellenanzahl.
 */
public class SimpsonIntegral extends Integral
{
/* ------------------------------------------------------ */
                                          // service-Methode
/**
 * Integrieren mittels Simpsonregel,
 * h/3(y[0]+4y[1]+2y[2]+ ... +2y[n-2]+4y[n-1]+y[n]).
 * @return Wert des Integrals
 */
  public double integrieren()
  {
    double[] stuetzStellen = getStuetzStellen();
    int n = stuetzStellen.length - 1;
    double h = getAbstand();

    double ergebnis
    = stuetzStellen[ 0] + stuetzStellen[ n];

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

    for( int i = 2; i < n - 1; i += 2)
      ergebnis += 2 * stuetzStellen[ 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): **0.6931471805599453**

Simsonregel (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 *SimpsonIntegral.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.

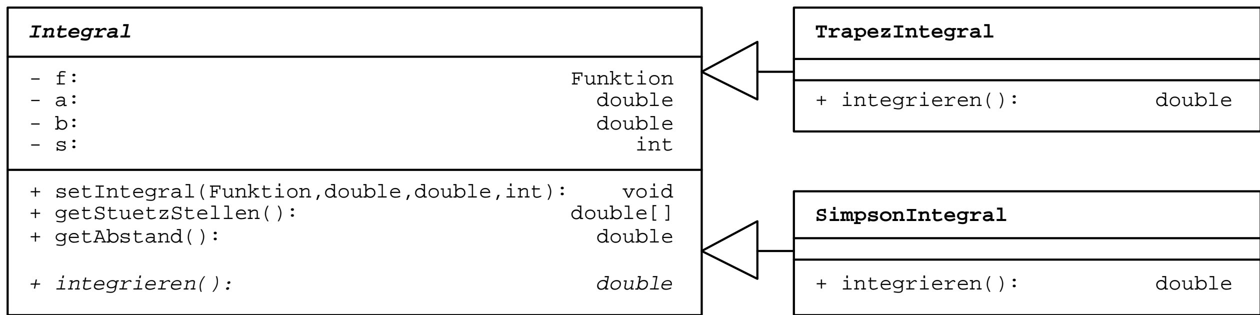
⇒ Das mit der Simpsonregel entwickelte *Interpolationspolynom* liefert eine *bessere* Annäherung der Ausgangsfunktion als das der Trapezregel und damit einen *besseren* Integralwert.

SimpsonIntegralTest.java

```
// SimpsonIntegralTest.java                          MM 2010

/**
 * Test der Klasse SimpsonIntegral.
 */
public class SimpsonIntegralTest
{
/**
 * Integral von 0 bis 1 ueber
 * Funktion ( 1.0 ) / ( 1.0 + 1.0 x^1 ).
 *
 *  Math.log( 2):               0.6931471805599453
 *  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
 */
  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( "Funktion " + fkt);
      System.out.println
      ( " Math.log( 2):               " + Math.log( 2));

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

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

Zusammenfassend wurden hier zwei Näherungsverfahren zur Berechnung bestimmter Integrale besprochen, welche beide von der abstrakten Klasse `Integral` abgeleitet wurden:


Integral
- f: Funktion
- a: double
- b: double
- s: int
+ setIntegral(Funktion,double,double,int): void
+ getStuetzStellen(): double[]
+ getAbstand(): double
+ integrieren(): double
TrapezIntegral
+ integrieren(): double
SimpsonIntegral
+ integrieren(): double

5.6 Nullstellenbestimmung

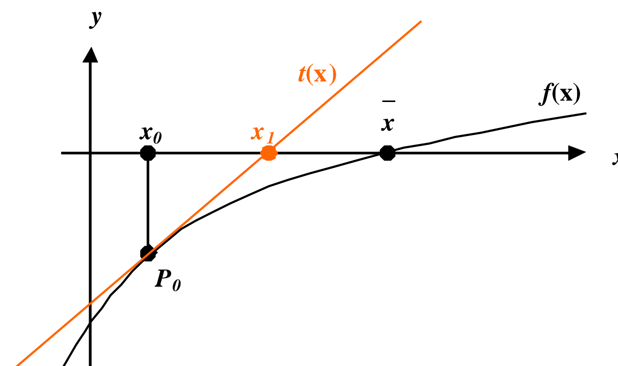
Zur näherungsweisen *Bestimmung von Nullstellen stetiger Funktionen* $f(x)$ werden oft **Iterationsverfahren** verwendet. Ausgehend von einer *Anfangsnäherungen* für eine Nullstelle werden solange neue Näherungen berechnet, bis eine gewünschte *Genauigkeit* erreicht ist (**Iteration**). Diese Verfahren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

5.6.1 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 $P_0 = (x_0, f(y_0))$ an die Funktion $y = f(x)$ eine Tangente $t(x)$ gelegt. Der Schnittpunkt dieser Tangente mit der Abzisse 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

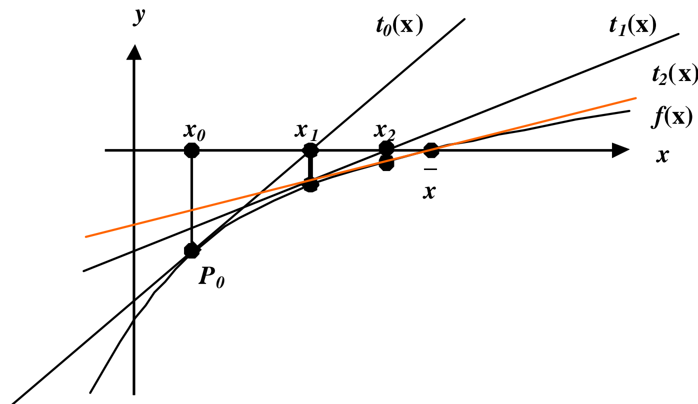
Eine *stetig differenzierbare* Funktion $f(x)$ und eine Näherung x_0.

1. **Berechnung 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. **Berechnung 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:*** $$\bar{x} \approx x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

3. Abbruch:

Ausgehend von einer Näherung x_0 wird eine Folge von Näherungen $x_1, x_2, \cdots, x_n$ 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.

NewtonIteration.java

```
 ...

   public double iterieren()
   {
     anzahl = 0;
     do
     {
       x0 -= f.wert( x0) / f.wertErsteAbleitung( x0);
       anzahl++;
     } while(( Math.abs( f.wert( x0)) >= eps)
                 && ( anzahl < maxAnzahl));
     return x0;
   }

 ...
```

5.6.2 Klasse `NewtonIteration`

NewtonIteration	
- f:	DifferenzierbareFunktion
- x0:	double
- maxAnzahl:	int
- anzahl:	int
- eps :	double
+ setNewtonIteration (DifferenzierbareFunktion,double,double,int):	void
+ getAnzahl():	int
+ iterieren():	double

NewtonIteration.java

```
// NewtonIteration.java                              MM 2010

/**
 * Interation zur Nullstellenbestimmung durch Newton.
 */
public class NewtonIteration
{
/* ------------------------------------------------- */
                                             // Attribute
/**
 * Funktion.
 */
  private DifferenzierbareFunktion f = null;

/**
 * Naeherung.
 */
  private double x0 = 0;

/**
 * Maximale Iterationstiefe.
 */
  private int maxAnzahl = 10;

/**
 * Aktuelle Iterationstiefe.
 */
  private int anzahl = 0;

/**
 * Genauigkeit.
 */
  private double eps = 0.0001;

/* ------------------------------------------------- */
                                            // set-Methode
/**
 * Setzt Attribute für Newton.
 * @param fkt differenzierbare Funktion
 * @param xx0 Naeherung
 * @param e Genauigkeit
 * @param max maximale Iterationstiefe
 */
  public void setNewtonIteration
  ( DifferenzierbareFunktion fkt, double xx0, double e,
    int max)
  {
    f = fkt;
    x0 = xx0;
    eps = e;
```

```
      maxAnzahl = max;
      anzahl = 0;
    }

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

  /* -------------------------------------------------- */
                                     // service-Methoden
  /**
   * Iteration.
   */
    public double iterieren()
    {
      anzahl = 0;
      do
      {
//      System.out.println( anzahl + ": " + x0);
        x0 -= f.wert( x0) / f.wertErsteAbleitung( x0);
        anzahl++;
      } while(( Math.abs( f.wert( x0)) >= eps)
                && ( anzahl < maxAnzahl));
      return x0;
    }
  }
```

5.6.3 Anwendung 1 - 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 auf die Nullstellenbestimmung dieser Funktion. $f(x) = x^k - a$ und $f'(x) = k \cdot x^{k-1}$ in die Newtonformel eingesetzt, liefert zur Berechnung der einer Näherung $x_1 = x_0 - \frac{x_0^k - a}{k \cdot x_0^{k-1}}$. 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 mit einem Programm betrachten wir $f(x) = x^k - a$ als Polynom. Damit kann die Funktionsberechnung und die Berechnung der ersten Ableitung über die entsprechenden 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
```

5.6.4 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	
- k:	int
+ setWurzel(int):	void
+ konsolenEingabe() :	void
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String

Wurzel.java

```
// Wurzel.java                                    MM 2010
import Tools.IO.*;

/**
 * Wurzelfunktion f(x) = k. Wurzel( x),
```

```
 * x >=0 aus R, k > 1 aus N.
 */
public class Wurzel
  extends DifferenzierbareFunktion
{
/* ---------------------------------------------------- */
                                           // Attribute
/**
 * Wurzelexponente k.
 */
  private int k = 2;

/* ---------------------------------------------------- */
                                         // set-Methode
/**
 * Setzt k fuer k. Wurzel( x).
 * @param exponent Wurzelexponent k
 */
  public void setWurzel( int exponent)
  {
    k = exponent;
  }

/**
 * Wurzelexponenteingabe ueber Konsole.
 */
  public void konsolenEingabe()
  {
// Eingabe der Geradenkoeffizienten
    System.out.println( "Wurzeleingabe");

    int k = 2;
    do
    {
       k = IOTools.readInteger
       ( " Wurzelexponent k ( k > 1) = ");
    } while( k < 2);

// Setzen des Wurzelexponenten
    setWurzel( k);
  }

/* ---------------------------------------------------- */
                                     // service-Methode
/**
 * Berechnen eines Funktionswertes
 * als Nullstelle der Funktion x ^ k - arg.
 * @param arg Argument
 * @return f( arg)
 */
  public double wert( double arg)
  {
```

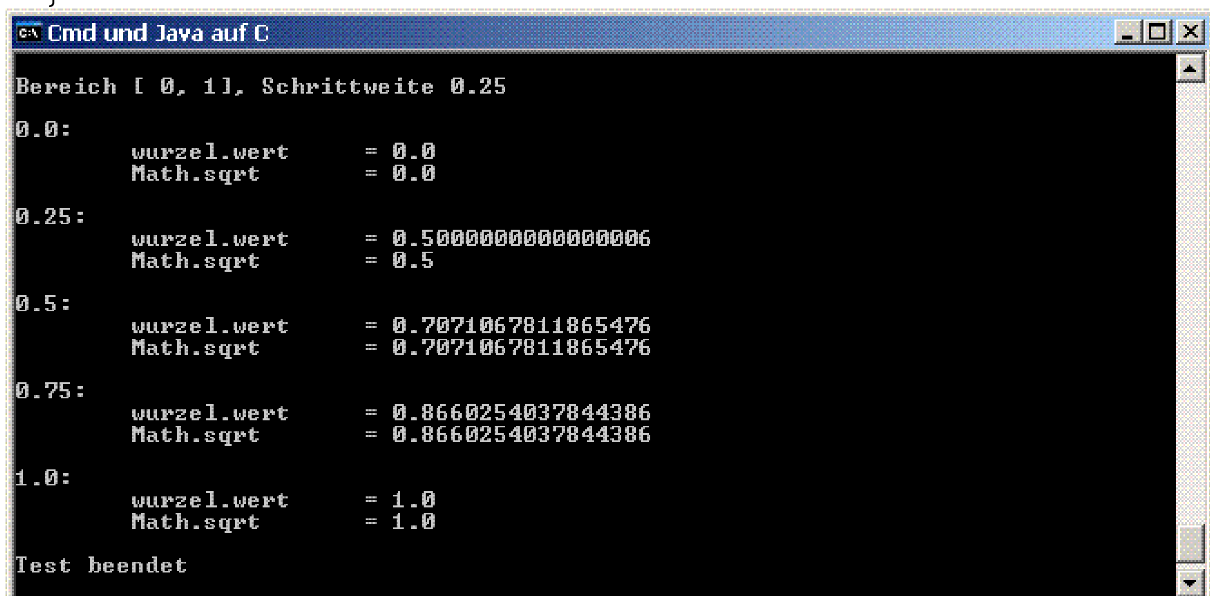
```
    if( arg == 0) return 0;

    double[] koeff = new double[ k + 1];
    koeff[ 0] = -arg;
    koeff[ k] = 1;
    for( int i = 1; i < k; i++) koeff[ i] = 0;

    Polynom polynom = new Polynom();
    polynom.setPolynom( koeff);

// Newtoniteration
    NewtonIteration newton = new NewtonIteration();
    newton.setNewtonIteration( polynom, arg, 1e-15, 50);

// Nullstellenberechnung
    return newton.iterieren();
  }

/**
 * Berechnen einer ersten Ableitung
 * f'(x) = k. Wurzel aus x / ( x * k).
 * @param arg Argument
 * @return Wert der ersten Ableitung an der Stelle arg
 */
  public double wertErsteAbleitung( double arg)
  {
    if( arg == 0) return 0;
    return wert( arg) / ( arg * k);
  }

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

Das Testprogramm berechnet für gegebene Argumente x die Funktion $f(x) = \sqrt[k]{x}$, vergleicht die Ergebnisse der Iteration mit denen die `Math.sqrt( x)` liefert.

WurzelTest.java

```
// WurzelTest.java                           MM 2010

/**
 * Test der Klasse Wurzel.
 */
public class WurzelTest
```

```
{
/**
 * Berechnet die Funktion f( x) = 2.Wurzel( x) fuer
 * gegebene x, 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( "  " + wurzel);

// Wertetabelle
    System.out.println();
    System.out.println
    ( "Bereich [ 0, 1], Schrittweite 0.25");

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

    System.out.println();
    System.out.println( "Test beendet");

    System.out.println();
  }
}
```



5.6.5 Anwendung 2 - 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.

Beispiel

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 $P_3(x) = (x - 3)(x - 1)^2$.

Die notwendige Funktionalität für die Newtoniteration $\bar{x} \approx x_1 = x_0 - \frac{P_r(x_0)}{P_r'(x_0)}$ stellt die Klasse `Polynom` zur Verfügung. Unser Programm liefert für das Polynom folgende Nullstellen:

$P_3(x) = x^3 - 5x^2 + 7x - 3$ berechnet 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
```

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

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

5.6.6 Klasse `PolynomNullstellen`

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

PolynomNullstellen.java

```
// PolynomNullstellen.java                              MM 2010
import Tools.IO.*;                                   // Eingaben

/**
 * Nullstellenberechner für Polynome.
 */
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
      Polynom polynom = new Polynom();
      polynom.konsolenEingabe();

// Polynomausgabe
      System.out.println();
      System.out.println( "Polynom " + polynom);

      do
      {
// Eingabe der Startnullstelle
```

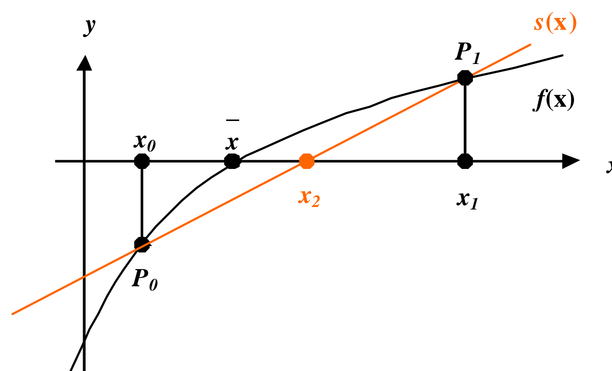
```
            double x0
            = IOTools.readDouble( "Anfangswert x0 = ");

    // Newtoniteration
            double eps =  1e-15;                // Genauigkeit
            int max = 30;// Maximale Anzahl der Iterationen

            NewtonIteration newton = new NewtonIteration();
            newton.setNewtonIteration
            ( polynom, x0, eps, max);

    // Nullstellenberechnung
            double erg = newton.iterieren();
            System.out.println
            ( " Newton (" + newton.getAnzahl() + "): " + erg);

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

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

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

5.6.7 Regula falsi (Sekantenverfahren)

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

Grundgedanke

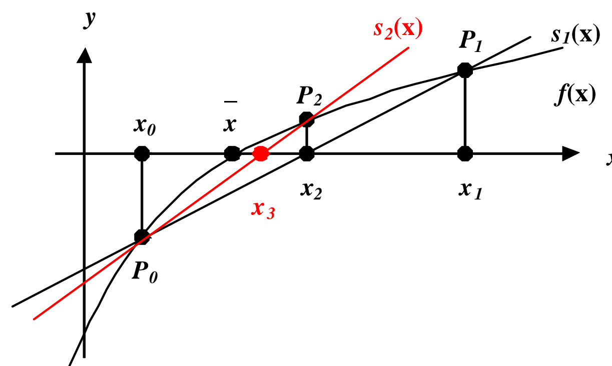
Ausgehend von zwei bereits vorhandenen Näherungen x_0, x_1 mit $y_0 = f(x_0) < 0$ und $y_1 = f(x_1) > 0$, wird durch die Punkte $P_0 = (x_0, y_0)$ und $P_1 = (x_1, y_1)$ eine Sekante $s(x)$ gelegt. Der Schnittpunkt dieser Sekante mit der Abzisse liefert eine neue Näherung x_2. Der Vorgang wird so oft 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
$$(y - y_0) = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$
   und
$$s(x) = y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$
   Sekante von $y = f(x)$ durch P_0 und P_1.
2. **Berechnung der Nullstelle x_2 der Sekante $s(x)$:**
$$s(x_2) = 0 \Rightarrow y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0) = 0 \Rightarrow x_2 - x_0 = -y_0 \frac{x_1 - x_0}{y_1 - y_0}$$
   $\Rightarrow$ ***Regula falsi:*** $\bar{x} \approx x_2 = x_0 - y_0 \dfrac{x_1 - x_0}{y_1 - y_0}$.
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.

RegulaFalsiIteration.java

```
  . . .

    public double iterieren()
    {
      double y0 = f.wert( x0);            // Funktionswerte
      double y1 = f.wert( x1);

      anzahl = 0;
      do
```

```
      {
        double x = x0 - ( x1 - x0) / ( y1 - y0) * y0;
        double y = f.wert( x);

        if( Math.signum( y) == Math.signum( y1))
        {
          x1 = x0;
          y1 = y0;
        }
        x0 = x;
        y0 = y;

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

      return x0;
    }

  ...
```

5.6.8 Klasse `RegulaFalsiIteration`

RegulaFalsiIteration	
- f:	StetigeFunktion
- x0:	double
- x1 :	double
- maxAnzahl:	int
- anzahl:	int
- eps :	double
+ setRegulaFalsiIteration (StetigeFunktion,double,double,int):	void
+ getAnzahl():	int
+ iterieren():	double

RegulaFalsiIteration.java

```
  // RegulaFalsiIteration.java                          MM 2008

  /**
   * Interation zur Nullstellenbestimmung
   * durch Regula Falsi.
   */
  public class RegulaFalsiIteration
  {
  /* ------------------------------------------------- */
                                               // Attribute
  /**
   * Funktion.
   */
    private Funktion f = null;
```

```
/**
 * Erste Naeherung.
 */
 private double x0 = 0;

/**
 * Zweite Naeherung.
 */
 private double x1 = 0;

/**
 * Maximale Iterationstiefe.
 */
 private int maxAnzahl = 10;

/**
 * Aktuelle Iterationstiefe.
 */
 private int anzahl = 0;

/**
 * Genauigkeit.
 */
 private double eps = 0.0001;

/* ------------------------------------------------- */
                                           // set-Methode
/**
 * Setzt Attribute füe Ragula falsiIteration.
 * @param fkt stetige Funktion
 * @param xx0 1. Naeherung
 * @param xx1 2. Naeherung
 * @param e Genauigkeit
 * @param max maximale Iterationstiefe
 * @return false, falls Voraussetzung nicht erfüllt.
 */
 public boolean setRegulaFalsiIteration
 ( StetigeFunktion fkt, double xx0, double xx1, double e,
   int max)
 {
   if( Math.signum( fkt.wert( xx0))
       == Math.signum( fkt.wert( xx1))) return false;
   f = fkt;
   x0 = xx0;
   x1 = xx1;
   eps = e;
   maxAnzahl = max;
   anzahl = 0;
   return true;
 }
```

```
/* ----------------------------------------------- */
                                     // get-Methode
/**
 * Liest Anzahl der Durchgaenge.
 * @return anzahl
 */
  public int getAnzahl()
  {
    return anzahl;
  }

/* ----------------------------------------------- */
                                 // service-Methoden
/**
 * Iteration.
 */
  public double iterieren()
  {
    double y0 = f.wert( x0);          // Funktionswerte
    double y1 = f.wert( x1);

    anzahl = 0;
    do
    {
     // System.out.print  ( " x0: " + x0);
     // System.out.println( "\tx1: " + x1);

      double x = x0 - ( x1 - x0) / ( y1 - y0) * y0;
      double y = f.wert( x);

      if( Math.signum( y) == Math.signum( y1))
      {
        x1 = x0;
        y1 = y0;
      }
      x0 = x;
      y0 = y;

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

    return x0;
  }
}
```

5.6.9 Anwendung 1 - Wurzelberechnung

Berechnung von $x = \sqrt[k]{a}$ mit $k \in \mathrm{N}$, $k > 1$, $a \in \mathrm{R}$, $a > 0$ durch Nullstellenbestimmung von $f(x) = x^k - a$. Als Anfangsnäherungen empfehlen sich $x_0 = 1$ und $x_1 = a$, da gilt:

$f(x) = x^k - a$	$0 < a < 1$	$a > 1$
$f(1) = 1 - a$	+	-
$f(a) = a^k - a$	-	+

Beispiel

Speziell für $a = 4$, $k = 2 \Rightarrow f(x) = x^2 - 4$ und den Anfangwerten $x_0 = 1$, $x_1 = 4$ ergibt sich aus Regula falsi auf 6 Stellen gerundet folgende Näherungen (TR):

i	x_i	x_{i+1}	$y_i = x_i^2 - 4$	$y_{i+1} = x_{i+1}^2 - 4$	$x_{i+2} = x_i - y_i \frac{x_{i+1} - x_i}{y_{i+1} - y_i}$	$y_{i+2} = x_{i+2}^2 - 4$
0	**1**	**4**	-3	12	1.6	-1.44
1	1.6	4	-1.44	12	1.8571429	-0,55102
2	1.8571429	4	-0.55102	12	1.951295	-0.19244
3	1.951295	4	-0.19244	12	1.9836308	-0.0652087
4	1.9836308	4	-0.0652087	12	**1.9943517**	**-0.0225613**

Die Berechnung zeigt eine Verbesserung der Startnäherung. Allerdings konvergiert dieses Verfahren sehr langsam gegen die korrekte Nullstelle.

Für die Berechnung mit einem Programm betrachten wir wiederum $f(x) = x^k - a$ als Polynom. Unser Programm bricht bei einer Genauigkeit von *eps* = 0.000000000000001 im 33. Schritt die Berechnung mit einem Näherungswert von 2.0 Wert ab:

```
  x0: 4.0                     x1: 1.0
  x0: 1.5999999999999996      x1: 4.0
  x0: 1.857142857142857       x1: 4.0
...
  x0: 1.9999999999999936      x1: 4.0
  x0: 1.9999999999999978      x1: 4.0
  x0: 1.9999999999999993      x1: 4.0
  Regula (33): 1.9999999999999998
```

5.6.10 Anwendung 2 - 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 $P_3(x) = (x-3)(x^2 - 2x + 1) = (x-3)(x-1)^2$.

Unser Programm 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 (62): 2.9999999999999996
```

Das Restpolynom $P_2(x) = x^2 - 2x + 1$ lässt sich nicht mit Regula Falsi berechnen, da die Anfangbedingung 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 $P_2(x) = x^2 - 2x + 1 \geq 0$.

Beide Anwendungen wurden mit dem folgenden Testprogramm berechnet.

RegulaFalsiIterationTest.java

```
// RegulaFalsiIterationTest.java                        MM 2010

/**
 * Test der Klasse RagulaFalsiIteration.
 */
public class RegulaFalsiIterationTest
{
/**
 * Zwei Beispiele:
 * 2. Wurzel aus 4
 * Polynom -3.0 + 7.0 x^1 + -5.0 x^2 + 1.0 x^3.
 */
  public static void main( String[] args)
  {
// Regulafalsiiteration
    double eps =  1e-15;                   // Genauigkeit
    int max = 100;   // Maximale Anzahl der Iterationen

// Beispiel 1: Wurzelberechnung
    double[] koeff1 = new double[ 3];
    koeff1[ 0] = -4;
    koeff1[ 1] = 0;
    koeff1[ 2] = 1;
    Polynom polynom1 = new Polynom();
    polynom1.setPolynom( koeff1);
    System.out.println( "\n" + polynom1);

// Regula falsi
    RegulaFalsiIteration regula1
    = new RegulaFalsiIteration();
    regula1.setRegulaFalsiIteration
    ( polynom1, 4, 1, eps, max);

// Nullstellenberechnung
    double erg1 = regula1.iterieren();
    System.out.println
    ( " Regula (" + regula1.getAnzahl() + "): " + erg1);
```

```
/* ------------------------------------------------ */
// Beispiel 2: Nullstellen eines Polynoms
    double[] koeff2 = new double[ 4];
    koeff2[ 0] = -3;
    koeff2[ 1] = 7;
    koeff2[ 2] = -5;
    koeff2[ 3] = 1;
    Polynom polynom2 = new Polynom();
    polynom2.setPolynom( koeff2);
    System.out.println( "\n" + polynom2);

// Regula falsi
    RegulaFalsiIteration regula2
    = new RegulaFalsiIteration();
    regula2.setRegulaFalsiIteration
    ( polynom2, 4, 2, eps, max);

// Nullstellenberechnung
    double erg2 = regula2.iterieren();
    System.out.println
    ( " Regula (" + regula2.getAnzahl() + "): " + erg2);
  }
}
```

5.6.11 Umkehrfunktionen als Nullstellenproblem

Die Wurzelfunktion konnten wir als Nullstellenproblem mittels der Potenzfunktion lösen. Weitere Umkehrfunktionen lassen sich analog als Nullstellenprobleme berechnen. Hier einige Beispiele:

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	$x = \ln x$	$f(x) = e^x - a$	$f'(x) = e^x$	$x_1 = x_0 + \frac{a}{e^{x_0}} - 1$	Verbesserung notwendig
Arcussinus	$x = \arcsin x$, $\lvert x \rvert < 1$	$f(x) = \sin x - a$	$f(x) = \cos x$	$x_1 = x_0 + \frac{\sin x_0 - a}{\cos x_0}$	
Arcustangens	$x = \arctan x$	$f(x) = \tan x - a$	$f(x) = 1 + \tan^2 x$	$x_1 = x_0 + \frac{\tan x_0 - a}{1 + \tan^2 x_0}$	Verbesserung notwendig

Weitere Funktionen lassen sich durch die Anwendung von ausgewählter Grundbeziehungen berechnen, wie zum Beispiel:

$$a^x = e^{x \ln a} \text{ für } a \in R, a > 0, a \neq 1,$$

$$\log_a x = \frac{\ln x}{\ln a} \text{ für } a \in R, a > 0, a \neq 1, x > 0,$$

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x) \text{ für } |x| < 1,$$