Inhalt

5 Numerische Methoden der praktischen Mathematik (I)

5.1 Einführung

Zwischen Mathematik und Praxis (Technik, Physik, Ökonomie, ...) ist über die Jahrhunderte hinweg eine befruchtende Wechselwirkung zu beobachten:

Mit *Rechenhilfsmitteln* wurden zunächst nur *formelmäßig* lösbare Probleme mathematisch bearbeitet. Später, zur Zeit der *mechanischen Rechenmaschinen*, entwickelten vor allem Mathematiker Verfahren, welche mittels sogenannten *Rechenschemata* gelöst wurden. Heute gestatten die modernen *elektronischen Rechenanlagen* neue numerische Lösungsverfahren, die als *Programme* auf diesen Anlagen ausgeführt werden.

Rechentechnik	Algorithmen	Entwickler	Ausführer
Rechenhilfsmittel (Urzeit)	Formeln	Naturwissenschaftler	Naturwissenschaftler
Mechanischer Rechner (ab Ende 17. Jh.)	Rechenschemata	Naturwissenschaftler	Technische Rechner
Elektronische Datenverarbeitung (ab Mitte 20. Jh.)	Programme	Naturwissenschaftler	Programmierer

Fehlerfortpflanzung

- *Ausgangsgrößen* bei der Lösung praktischer Probleme mittels der Mathematik und der Rechentechnik sind *häufig* durch Messungen ermittelte Werte.

⇒ **Fehlerbehaftete Eingabewerte**, deren Fehlertoleranzen von der verwendeten Messtechnik abhängen (Landvermessung, Elektronenmikroskop).

- *Numerische Verfahren* sind oft Näherungsverfahren.

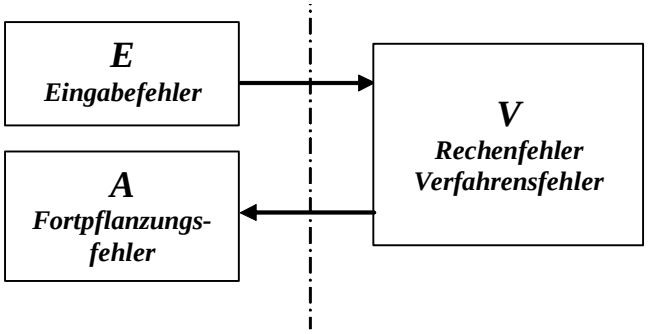
⇒ **Verfahrensfehler**, deren Größen von dem verwendeten Algorithmus abhängen.

- *Mathematische Hilfsmittel*, früher mechanische Rechenmaschinen, heute elektronische Rechenanlagen, arbeiten *ungenau*.

⇒ **Rechenfehler**, diese werden zwar mit den *immer genauer* arbeitenden Computer geringer, sind aber durch die Endlichkeit des Speichers, die Endlichkeit der Zahlendarstellung und die Endlichkeit der Rechenzeit *nicht* zu beseitigen.

Die Lösung eines Problems ist folglich mit *Eingabefehlern*, *Verfahrensfehlern* und *Rechenfehlern* behaftet. Die Korrektheit einer Lösung wird durch **Fehlerfortpflanzung** beeinflusst:

Fehlerbehaftete Eingabewerte werden mittels fehlerbehafteten Verfahren und ungenauer Rechnung verarbeitet.



Die **Numerik**, eine Teildisziplin der Mathematik, beschäftigt sich mit *Verfahren zur Lösung mathematischer Probleme*, deren *Fehlerfortpflanzung* und *Ergebnisauswertung*. Einem numerischen Verfahren liegt stets ein Algorithmus zugrunde, es enthält Kontrollen zur Fehlerfortpflanzung und entscheidet, in welchem Bereich eine Lösungsmethode verwendbare Ergebnisse liefert.

Bei der Auswahl eines Verfahrens sind Rechenzeit und Speicherökonomie zu beachten. Mathematisch elegante Lösungen sind manchmal rechentechnisch ungünstig. Man beurteilt ein numerisches Verfahren *streng nach ökonomischen Gesichtspunkten*:

Ein verwendetes numerisches Verfahren muss die notwendige Genauigkeit gewähren, aber unnötigen Rechenaufwand vermeiden.

5.2 Funktionen

Viele Algorithmen arbeiten mit reellen Funktionen $f(x)$ einer Veränderlichen. Da sehr oft Werteberechnungen von Funktionen notwendig sind, sollen verwendete Verfahren möglichst Zeit optimierend gestaltet werden. Nicht alle Funktionen lassen sich immer rechentechnisch exakt bestimmen. Wir werden hier *mathematisch exakte* Verfahren, welche *optimierende* Gesichtspunkte berücksichtigen, und auch *Näherungsverfahren* zur Funktionsberechnung kennenlernen.

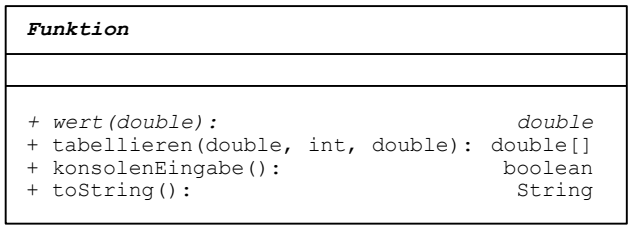
5.2.1 Funktionsklassen

Um den Funktionsbegriff systematisch zu modellieren, werden verschiedene, voneinander abhängige Funktionsklassen entwickelt. Zunächst definieren wir eine Klasse `Funktion` **abstrakt, d.h. ohne auf eine spezielle Funktion einzugehen.**
Die wichtigste Fähigkeit einer Funktion ist deren Wertberechnung bei einem gegebenen Argument:

Eine Funktion ordnet jedem Element *x* aus einem Definitionsbereich *D* eindeutig ein Element *y* aus einem Wertevorrat *W* zu.

Diese Abbildung soll in einer Methode `wert` realisiert werden. Da wir noch keine besonderen Funktionen betrachten, bleibt diese Methode ebenfalls **abstrakt, d.h. ohne auf eine spezielle Funktionswertberechnung einzugehen**. Unter der Voraussetzung, dass es solch eine Methode gibt, kann man eine weitere Methode `tabellieren` zum Erstellen einer Wertetabelle angeben. Schließlich wird eine Methode `toString` die Funktion als Zeichenkette in einer mathematisch üblichen Schreibweise darstellen. Eine Methode `konsolenEingabe` ermöglicht eine kontrollierte Tastatureingabe evtl. vorhandener Attribute.

Das folgende UML-Klassendiagramm beschreibt die Klasse `Funktion`, wobei *abstrakte* Bestandteile (Klassen und Methoden) *kursiv* dargestellt werden.



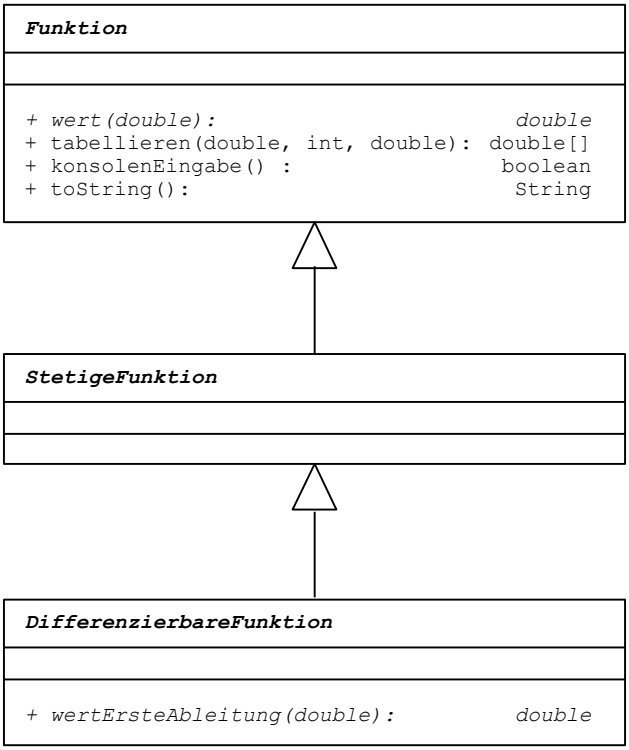
Funktion.java

```
// Funktion.java                                        MM 2013

/**
 * Funktion f( x), abstract.
 */
public abstract class Funktion
```

```
{
/* ------------------------------------------------------ */
                                          // service-Methoden
/**
 * Berechnen eines Funktionswertes, abstract.
 * @param arg Argument
 * @return f( arg)
 */
  public abstract double wert( double arg);

/**
 * Tabellieren von Funktionswerten.
 * @param x0 Startwert
 * @param n Anzahl der Werte
 * @param h Schrittweite
 * @return Wertetabelle
 */
  public double[] tabellieren( double x0, int n, double h)
  {
// Trivialfall, Tabelle aus nur einem Wert
    if( n < 1) n = 1;
    if( n == 1) h = 0;

// Tabelle
    double[] tabelle = new double[ n];

    double arg = x0;
    for( int i = 0; i < n; i++)
    {
      tabelle[ i] = wert( arg);
      arg += h;
    }

    return tabelle;
  }

/**
 * Eingabe von Parametern einer Funktion ueber Konsole,
 * falls welche existieren.
 * @return true, bei erfolgreicher Eingabe
 */
  public boolean konsolenEingabe()
  {
    return true;
  }

/**
 * Darstellen einer Funktion.
 * @return Funktion in linearer Schreibweise
 */
  public String toString()
  {
```

```
      return "";
    }
  }
```

Spezielle Funktionen sind stetige und stetig differenzierbare Funktionen. Von den letzteren kann die erste Ableitung berechnet werden. Da sie die Funktionalität der Funktionen erweitern, werden sie von der allgemeinen Klasse `Funktion` abgeleitet.



StetigeFunktion.java

```
// StetigeFunktion.java                                    MM 2013

/**
 * Stetige Funktion f( x), abstract,
 * besitzt stets einen Funktionswert.
 */
public abstract class StetigeFunktion extends Funktion
{
}
```

DifferenzierbareFunktion.java

```
// DifferenzierbareFunktion.java                           MM 2013

/**
 * Differenzierbare Funktion f( x), abstract.
 */
public abstract class DifferenzierbareFunktion
  extends StetigeFunktion
```

```
{
/* ------------------------------------------------------ */
                                              // service-Methode
/**
 * Berechnen der ersten Ableitung, abstract.
 * @param arg Argument
 * @return f'( arg)
 */
  public abstract double wertErsteAbleitung( double arg);
}
```

5.2.2 Beispiel „Lineare Funktionen - Geraden“

Um eine Klasse für eine spezielle Funktion einzuführen, muss man zunächst wissen, ob es sich um eine unstetige, stetige oder stetig differenzierbare Funktion handelt. Oft benötigt man zusätzlich noch Attribute, die die Funktion eindeutig bestimmen. Die Methoden `wert`, `toString` und `konsolenEingabe` sind entsprechend der Funktion zu spezifizieren. Die Methode `wertErsteAbleitung` ist im Fall einer differenzierbaren Funktion zu überschreiben.

Wir führen eine Klasse `Gerade` ein. Diese ist eine stetig differenzierbare Funktion und wird deshalb von der Klasse `DifferenzierbareFunktion` abgeleitet. Eine Gerade ist eindeutig durch zwei Werte definiert, dem linearen Glied, hier das Attribut `m`, und dem absoluten Glied, hier das Attribut `n`. Eine Methode `setGerade` setzt deren Werte. Für die Geradenberechnungen sind die Methoden `wert` und `wertErsteAbleitung` festzulegen. Eine Methode `toString` gibt die Gerade in einer mathematisch üblichen Schreibweise als Zeichenkette aus, eine Methode `konsolenEingabe` ermöglicht die Tastatureingabe der Attribute.

Gerade	
- m:	double
- n:	double
+ setGerade(double, double):	boolean
+ konsolenEingabe():	boolean
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String

Gerade.java

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

/**
 * Geradenfunktion g( x) = mx + n, D = W = R.
 */
public class Gerade extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
```

```
                                                    // Attribute
  /**
   * Anstieg m.
   */
    private double m = 0;

  /**
   * Absolutglied n.
   */
    private double n = 0;

  /* ------------------------------------------------- */
                                                // set-Methoden
  /**
   * Setzt Koeffizienten m und n.
   * @param anstieg Anstieg
   * @param absolut Absolutglied
   * @return true, bei erfolgreichem Eintrag
   */
    public boolean setGerade( double anstieg, double absolut)
    {
      m = anstieg;
      n = absolut;

      return true;
    }

  /**
   * Geradeneingabe ueber Konsole.
   * @return true, bei erfolgreicher Eingabe
   */
    public boolean konsolenEingabe()
    {
  // Eingabe der Geradenkoeffizienten
      double anstieg
      = IOTools.readDouble( " Anstieg      m = ");
      double absolut
      = IOTools.readDouble( " Absolutglied n = ");

  // Setzen der Geradenkoeffizienten
      return setGerade( anstieg, absolut);
    }

  /* ------------------------------------------------- */
                                            // service-Methoden
  /**
   * Berechnen eines Funktionswertes.
   * @param arg Argument
   * @return g( arg)
   */
    public double wert( double arg)
    {
```

```
    return m * arg + n;
  }

/**
 * Berechnen einer ersten Ableitung.
 * @param arg Argument
 * @return g'( arg)
 */
  public double wertErsteAbleitung( double arg)
  {
    return m;
  }

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

Wir betrachten nun zwei Anwendungen für die Klasse `Gerade`. Im Ersten Programm werden Funktionsberechnungen für Geraden durchgeführt und im zweiten Programm werden Geraden tabelliert.

GeradenBerechner.java

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

/**
 * Berechnet Geraden.
 */
public class GeradenBerechner
{
/**
 * Geradenberechner:
 * Geradeneingabe, Wertberechnung, Ergebnisausgabe.
 */
  public static void main( String[] args)
  {
// Ueberschrift
    System.out.println();
    System.out.println( "Geradenberechner");

// Dialog
    char weiter = 'j';
```

```
    do
    {
// Neue Gerade
      System.out.println();
      System.out.println( "Geradenkoeffizienten:");
      Gerade g = new Gerade();
      g.konsolenEingabe();

// Geradenausgabe
      System.out.println();
      System.out.println( "g( x) = " + g);

      do
      {
// Argumenteingabe
        System.out.println();
        System.out.println( "Wertberechnung");
        double x0 = IOTools.readDouble( " x0 = ");

// Wertberechnung
        double y0 = g.wert( x0);
        double y1 = g.wertErsteAbleitung( x0);

// Ausgabe
        System.out.print( " g( " + x0 + ")  = " + y0);
        System.out.println
        ( "\t\tg'( " + x0 + ") = " + y1);

        System.out.println();

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

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

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


/* ------------------------------------------------- */
                                          // Testbeispiel
/*
  g( x) = 3.0x + 2.0

  Wertberechnung x0 = 3
  g( 3) = 11.0
  g'( 3) = 3.0
```

```
*/
```

GeradenTabulator.java

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

/**
 * Tabelliert Geraden.
 */
public class GeradenTabulator
{
/**
 * Geradentabulator:
 * Geradeneingabe, Eingabe der Tabellenparameter,
 * Berechnen und Ausgabe der Tabelle.
 */
  public static void main( String[] args)
  {
// Ueberschrift
    System.out.println();
    System.out.println( "Geradentabulator");

// Dialog
    char weiter = 'j';
    do
    {
// Neue Gerade
      System.out.println();
      System.out.println( "Geradenkoeffizienten:");
      Gerade g = new Gerade();
      g.konsolenEingabe();

// Geradenausgabe
      System.out.println();
      System.out.println( "g( x) = " + g);

      do
      {
// Eingabe der Tabellenparameter
        System.out.println();
        System.out.println( "Wertbereich");

        double x0
        = IOTools.readDouble( " Startargument x0 = ");
        int n
        = IOTools.readInteger( " Anzahl n (n > 0) = ");
        double h = 0;
        if( n > 1)
        h = IOTools.readDouble( " Schrittweite h   = ");
```

```
  // Wertetabelle
        double[] tabelle = g.tabellieren( x0, n, h);

  // Tabellenausgabe
        double arg = x0;
        for( int i = 0; i < tabelle.length; i++)
        {
          System.out.println
          ( " g( " + arg + ")\t= " + tabelle[ i]);
          arg += h;
        }

        System.out.println();

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

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

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


/* ------------------------------------------------- */
                                          // Testbeispiel
/*
  g( x) = 3.0x + 2.0

  Wertbereich
   Startargument x0 = 0
   Anzahl n (n > 0) = 5
   Schrittweite h   = 0.25

   g( 0.0)  = 2.0
   g( 0.25) = 2.75
   g( 0.5)  = 3.5
   g( 0.75) = 4.25
   g( 1.0)  = 5.0

*/
```

5.3 Polynomberechnung

Polynome spielen in der Numerik eine besondere Rolle. Oft verwendet man sie als *Interpolationspolynome* zur angenäherten Darstellung einer Funktion. Die Funktion selbst muss dabei nicht bekannt sein. Man benötigt diskret gegebenen Funktionswerte, welche zum Beispiel durch Messwerte ermittelt wurden, legt durch diese ein oder mehrere Polynome und approximiert damit eine Funktion, welche diese Werte durchläuft. Eine Funktionsberechnung kann so näherungsweise mittels der Berechnung der Polynome ausgeführt werden.

$$P_r(x) = a_r x^r + a_{r-1} x^{r-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{r} a_i x^i \quad \text{mit } a_r \neq 0$$

Wir behandeln ein Rechenzeit optimierendes Verfahren zur Polynomberechnung.

5.3.1 Hornerschema

Grundgedanke

Durch geeignetes *Ausklammern* verhindert man das zeitaufwendige Berechnen von Potenzen in einem Polynom und spart damit Rechenzeit.

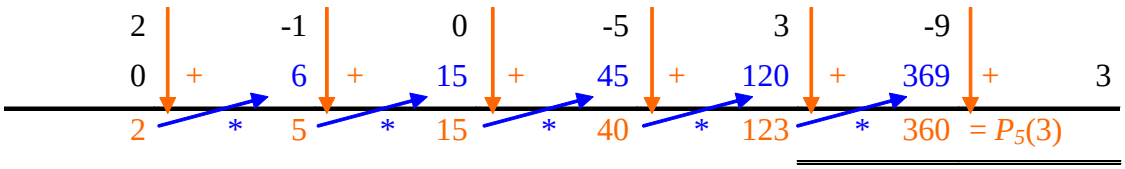
Beispiel

Gegeben sei das Polynom 5. Grades $P_5(x) = 2x^5 - x^4 - 5x^2 + 3x - 9$, gesucht sei der Wert des Polynoms an der Stelle $x_0 = 3$, also $P_5(3)$. Nun formt man das Polynom durch Ausklammern so um, dass keine Potenzen mehr vorhanden sind:

$$P_5(x) = 2x^5 - x^4 - 5x^2 + 3x - 9 = ((((2x-1)x+0)x-5)x+3)x-9$$

Dieses umgestellte Polynom gestattet eine sehr schnelle Berechnung, auch mit dem Taschenrechner.

Das hierfür verwendete *Rechenschema* trägt den Namen **Hornerschema**, nach **William Georg Horner (1786 - 1837)**, und sieht folgendermaßen aus:



Probe

$$P_5(3) = 486 - 81 - 45 + 9 - 9 = 360\,.$$

Allgemein

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

Hornerschema für die Polynomwertbestimmung an der Stelle x_0

Durch dieses Verfahren wird der Rechenaufwand wesentlich verringert:

Polynomberechnung ohne Hornerschema:

# Mult	$= \sum_{i=1}^{r} i = \frac{r(r+1)}{2}$	
# Add	$= r$	
# Mult + # Add	$= \frac{r^2+3r}{2}$	

Polynomberechnung mit Hornerschema:

# Mult	$= r$	
# Add	$= r$	+1 (Anfangsaddition mit 0)
# Mult + # Add	$= 2r$	+1 (Anfangsaddition mit 0)

Rechentechnische Umsetzung des Hornerschemas

Ein Polynom ist durch seine Koeffizienten eindeutig bestimmt. Deshalb werden diese als Attribute abgespeichert. Die Wertberechnung erfolgt durch das Hornerschema.

```
private double[] koeffizienten;

public double wert( double arg)
{
  double erg = 0;
  for( int i = koeffizienten.length - 1; i >= 0; i--)
    erg = erg * arg + koeffizienten[ i];
  return erg;
}
```

5.3.2 Erweitertes Hornerschema

Aus dem Hornerschema folgt, dass $b_0 = a_r$, $b_{i+1} = a_{r-i-1} + b_i\, x_0$ für $i = 1, ..., r$ und $b_r = P_r(x_0)$.
Umgeformt ergibt sich $a_r = b_0$ und $a_{r-i-1} = b_{i+1} - b_i\, x_0$.
Setze $j = r - i - 1$, so ist $a_j = b_{r-j} - b_{r-j-1}\, x_0$ und wegen $a_r = b_0$ und $b_r = P_r(x_0)$ folgt:

$$\begin{aligned}
P_r(x) &= a_r x^r + a_{r-1}x^{r-1} + a_{r-2}x^{r-2} + \cdots + a_1 x + a_0 \\
&= b_0 x^r + (b_1 - b_0 x_0)x^{r-1} + (b_2 - b_1 x_0)x^{r-2} + \cdots + (b_{r-1} - b_{r-2}x_0)x + (b_r - b_{r-1}x_0) \\
&= (x - x_0)b_0 x^{r-1} + (x - x_0)b_1 x^{r-2} + \cdots + (x - x_0)b_{r-2}x + (x - x_0)b_{r-1} + b_r \\
&= (x - x_0)(b_0 x^{r-1} + b_1 x^{r-2} + \cdots + b_{r-2}x + b_{r-1}) + b_r
\end{aligned}$$

$$\Rightarrow \quad P_r(x) = (x - x_0)(b_0 x^{r-1} + b_1 x^{r-2} + \cdots + b_{r-2}x + b_{r-1}) + P_r(x_0)$$

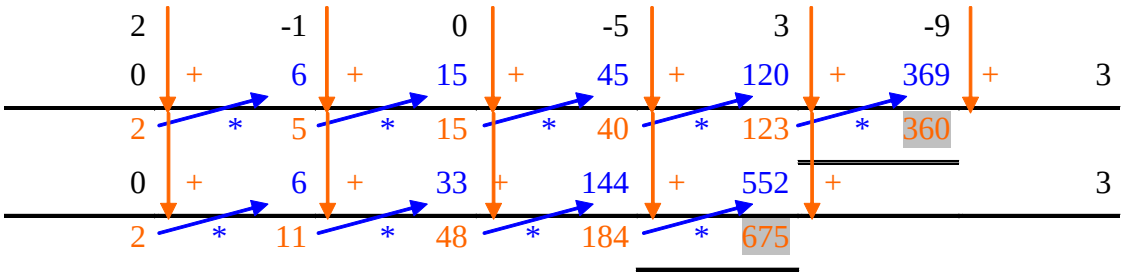
$$\Rightarrow \quad \frac{P_r(x) - P_r(x_0)}{x - x_0} = p_r(x) \text{ mit } p_r(x) = b_0 x^{r-1} + b_1 x^{r-2} + \cdots + b_{r-2}x + b_{r-1}$$

$\Rightarrow$ $p_r(x)$ kann als Differenzenquotient zur Berechnung der 1. Ableitung an der Stelle $x = x_0$ genutzt werden:

$$P'(x_0)=\lim_{x\to x_0}\frac{P(x)-P(x_0)}{x-x_0} \quad \text{mit} \quad P'(x)=b_0x^{r-1}+b_1x^{r-2}+\cdots+b_{r-2}x+b_{r-1}$$

Beispiel

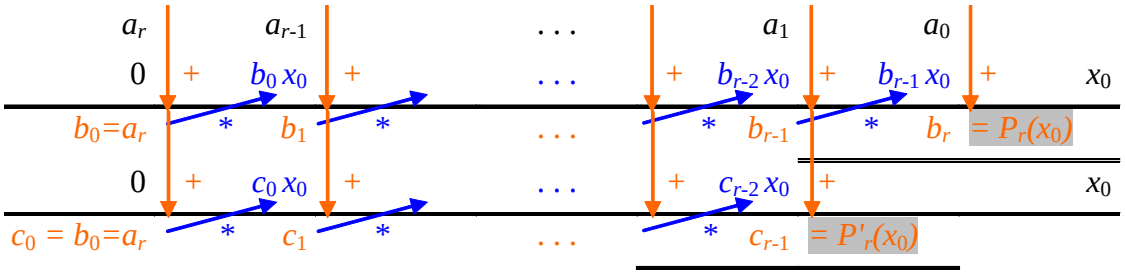
$P_5(x)=2x^5-x^4-5x^2+3x-9$, gesucht $P_5'(3)$.



Probe

$P_5'(x)=10x^4-4x^3-10x+3,\ P_5'(3)=810-108-30+3=675.$

Erweitertes Hornerschema für die Polynomwertbestimmung und für die Berechnung der 1. Ableitung an der Stelle x_0:



Rechentechnische Umsetzung des erweiterten Hornerschemas

```
public double wertErsteAbleitung( double arg)
{
  double erg = 0, erg1 = 0;
  for( int i = koeffizienten.length - 1; i > 0; i--)
  {
    erg = erg * arg + koeffizienten[ i];
    erg1 = erg1 * arg + erg;
  }
  return erg1;
}
```

5.3.3 Klasse `Polynom`

Zum Abschluss fassen wir die vollständige Klasse `Polynom` zusammen. Ein Polynom ist eine differenzierbare Funktion und wird von der Klasse `DifferenzierbareFunktion` abgeleitet. Es wird eindeutig durch seine Koeffizienten definiert. Diese werden als Feld abgelegt. Eine Methode `setPolynom` setzt deren Werte. Für die Polynomberechnungen sind

die Methoden `wert` und `wertErsteAbleitung` festzulegen. Eine Methode `toString` gibt das Polynom in einer mathematisch üblichen Schreibweise als Zeichenkette aus, eine Methode `konsolenEingabe` ermöglicht die Tastatureingabe der Koeffizienten.

Polynom.java

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

/**
 * Ganze rationale Funktion (Polynom r-ten Grades),
 * f(x) = ar*x^r + ... a2*x^2 + a1*x + a0, D = W = R.
 */
public class Polynom extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
                                                  // Attribute
/**
 * Polynomkoeffizienten.
 */
  private double[] koeffizienten = null;

/* ------------------------------------------------------ */
                                               // set-Methoden
/**
 * Setzt Polynomkoeffizienten fest.
 * @param koeff Koeffizienten des Polynoms
 * @return true, bei erfolgreichem Eintrag
 *         false, falls keine Koeffizienten existieren
 */
  public boolean setPolynom( double[] koeff)
  {
    if( koeff == null || koeff.length == 0)
    {
      koeffizienten = null;
      return false;
    }
    koeffizienten = koeff;
    return true;
  }

/**
 * Polynomeingabe ueber Konsole.
 * @return true, bei erfolgreichem Eintrag
 */
  public boolean konsolenEingabe()
  {
// Eingabe der Polynomkoeffizienten
    System.out.println( "Polynomeingabe");
```

```
      int grad =
      IOTools.readInteger( " Grad des Polynoms (>=0): ");

      double[] koeff = null;
      if( grad >= 0)
      {
        koeff = new double[ grad + 1];
        for( int i = 0; i < grad + 1; i++)
          koeff[ i] =
          IOTools.readDouble( " " + i + ". Koeffizient: ");
      }

  // Setzen der Polynomkoeffizienten
      return setPolynom( koeff);
    }

  /* -----------------------------------------------------*/
                                          // service-Methoden
  /**
   * Berechnen eines Funktionswertes nach HORNER.
   * @param arg Argument
   * @return f( arg)
   */
    public double wert( double arg)
    {
      double erg = 0;
      for( int i = koeffizienten.length - 1; i >= 0; i--)
        erg = erg * arg + koeffizienten[ i];
      return erg;
    }

  /**
   * Berechnen einer Ableitung nach erweitertem HORNER.
   * @param arg Argument
   * @return f'( arg)
   */
    public double wertErsteAbleitung( double arg)
    {
      double erg = 0, erg1 = 0;
      for( int i = koeffizienten.length - 1; i > 0; i--)
      {
        erg = erg * arg + koeffizienten[ i];
        erg1 = erg1 * arg + erg;
      }
      return erg1;
    }

  /* ------------------------------------------------------ */
                                          // toString-Methode
  /**
   * Darstellung eines Polynoms.
   * @return Funktion in linearer Schreibweise
```

```
   */
    public String toString()
    {
      String str = "" + koeffizienten[ 0];
      for( int i = 1; i < koeffizienten.length; i++)
        str += " + " + koeffizienten[ i] + " x^" + i;
      return str;
    }
}
```

5.3.4 Klasse `RationaleFunktion`

$$f(x) = \frac{P_r(x)}{P_s(x)} = \frac{\sum_{i=0}^{r} a_i x^i}{\sum_{i=0}^{s} b_i x^i}$$

Rationale Funktionen unterteilt man in *ganzrationale* ($s = 0$, $r > 0$: *Polynome*) und *gebrochen rationale* ($s > 0$; $r < s$ *echte*, $r \geq s$ *unechte*) Funktionen. Im zweiten Fall sind diese nicht notwendig stetig. Folglich werden sie direkt von der Klasse `Funktion` abgeleitet. Zähler und Nenner sind Polynome. Sie werden als Attribute aufgenommen. Die `set`-Methode übergibt diese. Für die Wertberechnung `wert`, die Methode `konsolenEingabe` und die Funktionsdarstellung `toString` werden die entsprechenden Methoden für das Zähler- und Nennerpolynom der Klasse `Polynom` herangezogen.

RationaleFunktion	
- zaehler: - nenner:	Polynom Polynom
+ setRationaleFunktion(Polynom, Polynom): + konsolenEingabe(): + wert(double): + toString():	boolean boolean double String

RationaleFunktion.java

```
// RationaleFunktion.java                                   MM 2013

/**
 * Rationale Funktion (Polynom / Polynom),
 * D = R \ <Nennerpolynomnullstellen>, W = R.
 */
public class RationaleFunktion extends Funktion
{
/* ------------------------------------------------------ */
                                                  // Attribute
/**
 * Zaehler.
```

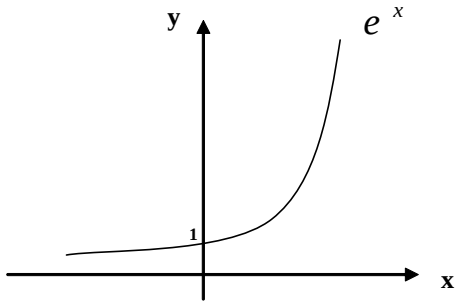
```
   */
    private Polynom zaehler = null;

  /**
   * Nenner.
   */
    private Polynom nenner = null;

  /* ------------------------------------------------- */
                                              // set-Methoden
  /**
   * Setzt Zaehler- und Nennerpolynom.
   * @param z Zaehlerpolynom
   * @param n Nennerpolynom
   * @return true, bei erfolgreichem Eintrag
   *         false, falls keine Polynome existieren
   */
    public boolean setRationaleFunktion( Polynom z, Polynom n)
    {
      if( z == null || n == null)
      {
        zaehler = null;
        nenner = null;
        return false;
      }

      zaehler = z;
      nenner = n;
      return true;
    }

  /**
   * Eingabe einer rationalen Funktion ueber Konsole.
   * @return true, bei erfolgreichem Eintrag
   */
    public boolean konsolenEingabe()
    {
  // Eingabe der Zaehler- und Nennerpolynome
      System.out.println( "Zaehler");
      Polynom z = new Polynom();
      if( !z.konsolenEingabe()) z = null;

      System.out.println( "Nenner");
      Polynom n = new Polynom();
      if( !n.konsolenEingabe()) n = null;

  // Setzen des Zaehler- und Nennerpolynoms.
      return setRationaleFunktion( z, n);
    }

  /* ------------------------------------------------- */
                                          // service-Methode
```

```
/**
 * Berechnen eines Funktionswertes Polynom / Polynom
 * unter Beruecksichtigung von Nennernullstellen.
 * @param arg Argument
 * @return f( arg), falls
 *         n == 0 und z < 0:  Double.NEGATIVE_INFINITY
 *         n == 0 und z == 0: Double.NaN
 *         n == 0 und z > 0:  Double.POSITIVE_INFINITY
 */
  public double wert( double arg)
  {
    double z = zaehler.wert( arg);
    double n = nenner.wert( arg);

    return z / n;
  }

/* ------------------------------------------------------ */
                                          // toString-Methode
/**
 * Darstellung einer rationalen Funktion.
 * @return Funktion in linearer Schreibweise
 */
  public String toString()
  {
    return "( " + zaehler + " ) / ( " + nenner + " )";
  }
}
```

5.4 Reihenentwicklungen

Einige in einem Bereich der reellen Zahlen stetig differenzierbare Funktionen lassen sich als Grenzwerte konvergenter Potenzreihen entwickeln, so zum Beispiel Exponentialfunktion und Winkelfunktionen.
Da diese Reihen unendlich sind und nur endliche Algorithmen verarbeitet werden, muss die Berechnung geeignet abgebrochen werden. Deshalb sind *Verfahrensfehler* zu erwarten. Da nur endliche Datendarstellungen verarbeitet werden, erhalten wir mit *Rechenfehler* belastete Näherungswerte für die Funktionsberechnungen. Die dabei entstehenden *Fortpflanzungsfehler* dürfen sich nur unwesentlich auf das Ergebnis auswirken, ansonsten wird eine Fehlerbehandlung notwendig oder das verwendete Verfahren ist unbrauchbar.

5.4.1 Exponentialfunktion



Reihenentwicklung: $$f(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Konvergenzbereich: $|x| < \infty$

Algorithmus für eine Reihenentwicklung allgemein

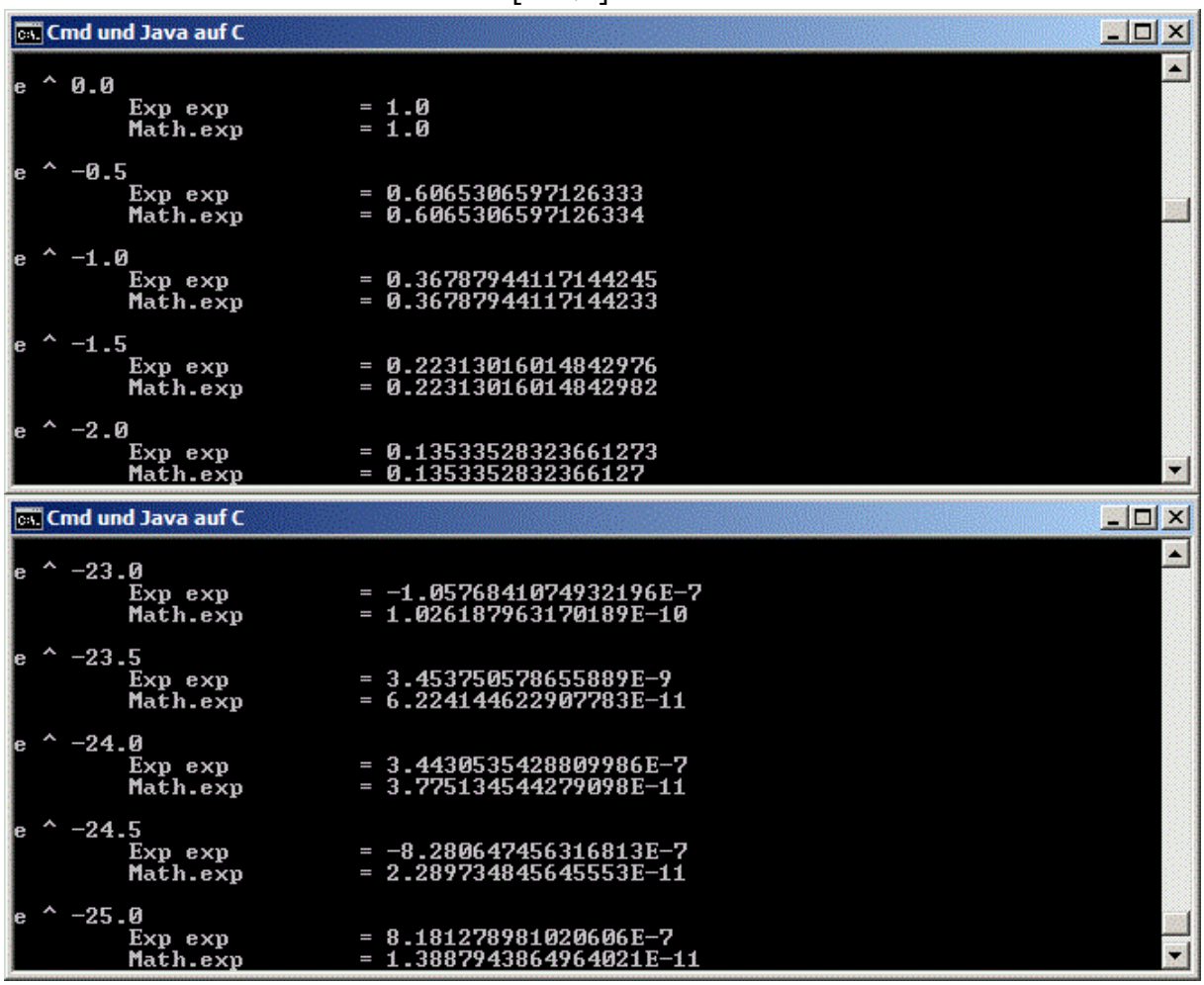
Um die Berechnung zu optimieren, bestimmt man den neuen Summanden aus dem alten und addiert ihn zur bis dahin berechneten Summe. Man wiederholt das Aufsummieren solange, bis der neue Summand so klein ist, dass er keinen Beitrag zur Summe bildet, d.h. die Summe sich nicht mehr verändert (*Rechnergenauigkeit*).

Rechentechnische Umsetzung der Reihenentwicklung

```
private double myExp( double arg)
{
  double x = arg;
  double alteSumme, neueSumme = 1, summand = 1;
  int i = 0;

  do                                  // Reihenentwicklung
  {
    summand *= x / ++i;                   // neuer Summand

    alteSumme = neueSumme;                   // neue Summe
    neueSumme += summand;

  } while( neueSumme != alteSumme);
  return alteSumme;
}
```

In einem Testprogramm wurden Funktionswerte mit der Reihenentwicklung berechnet und die Ergebnisse anschließend mit der Klassenmethode `exp` der Klasse `Math` verglichen.

Ausschnitt aus der Wertetabelle für $x \in [-25{,}0]$:



Fehlerbehandlung

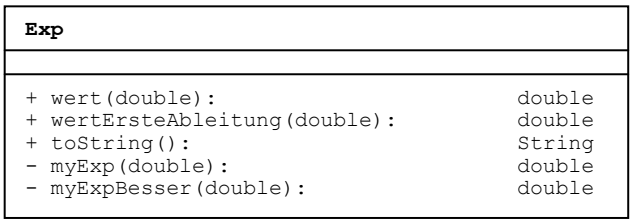
Für negative *x* wird der Wert der Funktion teilweise sogar negativ. Diese Fehler entstehen als *Rechenfehler und deren Fortpflanzung*, bedingt durch die Endlichkeit der Zahlendarstellung. Es ist eine Ergebniskorrektur notwendig:

$$e^x = \begin{cases} e^x, & x \geq 0 \\ \dfrac{1}{e^{|x|}}, & x < 0 \end{cases}.$$

```
private double myExpBesser( double arg)
{
  if(arg < 0) return 1 / myExp( - arg);
  return myExp(arg);
}
```

5.4.2 Klasse `Exp`

Die neuen Klasse `Exp` wird von der Klasse `DifferenzierbareFunktion` abgeleitet und hat den folgenden Aufbau:



Die Werteberechnung erfolgt unter Verwendung der Potenzreihe $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. Die Reihenentwicklung und deren Verbesserung werden in zwei `private`-Methoden vorbereitet.

Exp.java

```
// Exp.java                                                MM 2013

/**
 * Exponentialfunktion f(x) = e ^ x,
 * D = R, W = ] 0, +unendlich[.
 */
public class Exp extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
                                          // service-Methoden
/**
 * Berechnen eines Funktionswertes e ^ x.
 * @param arg Argument
 * @return e^arg
 */
  public double wert( double arg)
  {
// Trivialfaelle
    if( arg == 0) return 1;

// einfache Reihenentwicklung
//  return myExp( arg);
// verbesserte Reihenentwicklung
    return myExpBesser( arg);
  }

/**
 * Berechnen eines Funktionswertes durch Reihenentwicklung:
 * e ^ x = 1 + x + ( x ^ 2) / 2! + ( x ^ 3) / 3! ...
 * @param arg Argument
 * @return e^arg
```

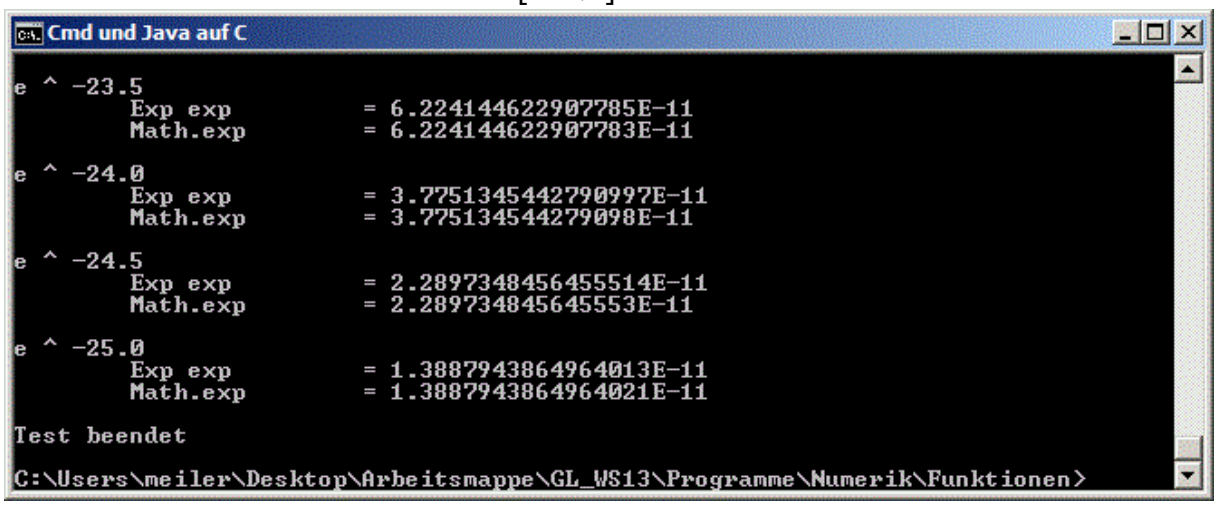
```
   */
    private double myExp( double arg)
    {
      double x = arg;
      double alteSumme, neueSumme = 1, summand = 1;
      int i = 0;
      do                                  // Reihenentwicklung
      {
//      System.out.println
//      ( i + ": " + neueSumme + " " + summand);
        summand *= x / ++i;                    // neuer Summand

        alteSumme = neueSumme;                    // neue Summe
        neueSumme += summand;
      } while(( neueSumme != alteSumme) && ( i < 1000));

      return alteSumme;
    }

  /**
   * Verbesserte Exponentialfunktion:
   * Fuer x < 0 berechne e^x = 1 / (e^-x).
   * @param arg Argument
   * @return e^arg verbessert
   */
    private double myExpBesser( double arg)
    {
      if( arg < 0) return 1 / myExp( -arg);
      return myExp( arg);
    }

  /**
   * Berechnen der ersten Ableitung f'( x) = e ^ x.
   * @param arg Argument
   * @return f'( arg)
   */
    public double wertErsteAbleitung( double arg)
    {
      return wert( arg);
    }

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

Das Testprogramm berechnet die Funktion e^x für gegebene Argumente *x*, vergleicht die Ergebnisse der verbesserten Reihenentwicklung mit denen die `Math.exp( x)` liefert.

ExpTest.java

```
// ExpTest.java                                          MM 2013

/**
 * Test der Klasse Exp.
 */
public class ExpTest
{
/**
 * Berechnet fuer gegebene Argumente e ^ x,
 * vergleicht Ergebnis mit Math.exp( x).
 */
  public static void main( String[] args)
  {
// Neue Funktion
    Exp fkt = new Exp();

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

// Wertevergleich
    System.out.println();
    System.out.println
    ( "Vergleich von 0 bis -25, Schrittweite -0.5");

    for( double x = 0; x > -25.5; x -= 0.5)
    {
      System.out.println();
      System.out.println( "e ^ " + x);
      System.out.println
      ( "\tExp exp    \t= " + fkt.wert( x));
      System.out.println
      ( "\tMath.exp   \t= " + Math.exp( x));
    }

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

Ausschnitt aus der Wertetabelle für $x \in [-25,0]$:



5.4.3 Winkelfunktionen

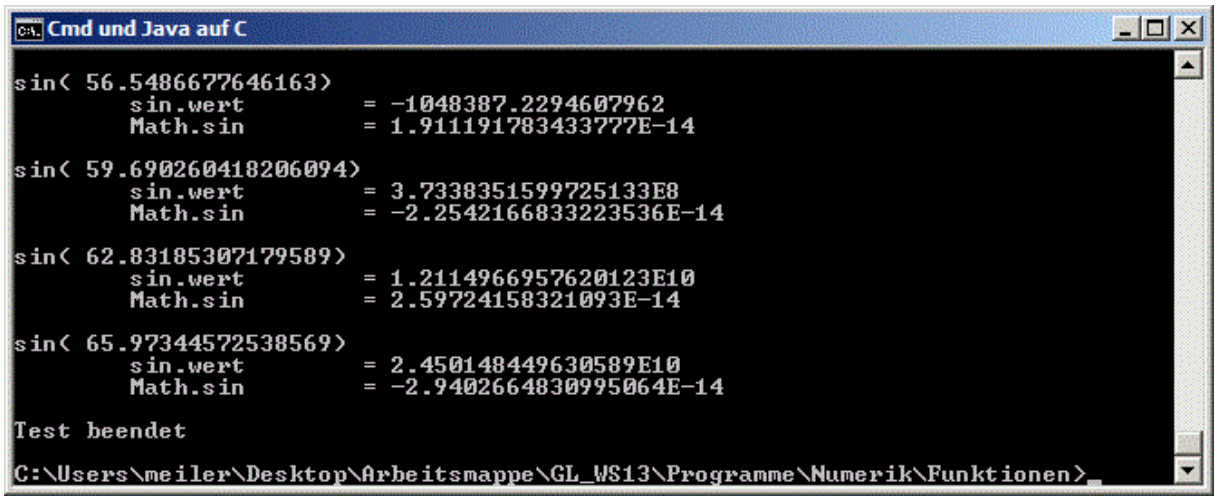
Reihenentwicklung: $f(x) = \sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$

Konvergenzbereich: $|x| < \infty$

Rechentechnische Umsetzung der Reihenentwicklung

```
private double mySin( double arg)
{
  double x = arg;
  double alteSumme, neueSumme = x, summand = x;
  int j = 1;

  do                                  // Reihenentwicklung
  {
    summand *= -x * x / ++j;          // neuer Summand
    summand /= ++j;

    alteSumme = neueSumme;                // neue Summe
    neueSumme += summand;

  } while( neueSumme != alteSumme);

  return alteSumme;
}
```

Ausschnitt aus der Wertetabelle für $x = k \cdot \pi$, $k \in [0,20] \Rightarrow \sin(x) = 0$

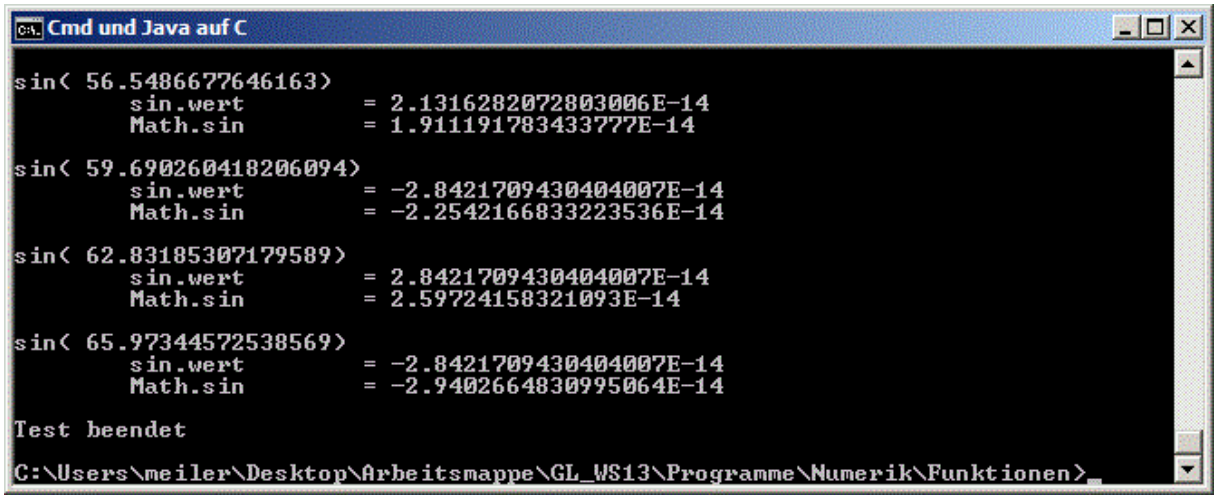


Fehlerbehandlung

Die Ergebnisse weichen wesentlich von den korrekten Funktionswerten ab: $\sin(x) \in [-1,1]$. Auch hier entstehen *Rechenfehler und deren Fortpflanzung*. Diese fehlerbehaftete Funktionswerte treten für sehr große Argumente *x* auf. Der Grund liegt in deren geringen Darstellungsdichte als Maschinenzahl und lassen sich durch eine Transformation des Wertes *x* auf das dichtere Intervall $x \in \left[0, \frac{\pi}{2}\right]$ beheben.

```
  private double mySinBesser( double arg)
  {
    double x = arg;
    int s = 1;                                    // Vorzeichen
// x < 0
    if( x < 0)
    {
      x = -x;
      s = -s;
    }
// x > PI
    if( x > Math.PI)
    {
      int temp = ( int)( x / Math.PI);
      if( temp % 2 != 0) s = -s;        // Vorzeichenwechsel
      x = x - temp * Math.PI;
    }
// x > PI/2
    if( x > Math.PI/2) x = Math.PI - x;

// Trivialfaelle
    if( x == 0 ) return 0;
    if( x == Math.PI / 2) return s;

    return s * mySin( x);
  }
```

Ausschnitt aus der Wertetabelle für $x = k \cdot \pi,\ k \in [0,20] \Rightarrow \sin(x) = 0$



Weitere trigonometrische Funktionen ließen sich analog durch ihre Reihenentwicklungen berechnen. Besser ist die Anwendung ausgewählter Grundbeziehungen zwischen den Winkelfunktionen, wie zum Beispiel

$$\cos(x) = \sin\left(\frac{\pi}{2} + x\right) \text{ für } |x| < \infty,$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \text{ für } |x| < \frac{\pi}{2} \text{ und}$$

$$\cot(x) = \frac{\cos(x)}{\sin(x)} \text{ für } 0 < x < \pi.$$

5.4.4 Klasse `Sinus`

Die Werteberechnung erfolgt unter Verwendung der Potenzreihe: $\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$

Sinus	
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String
- mySin(double):	double
- mySinBesser(double):	double

Sinus.java

```
// Sinus.java                                          MM 2013

/**
 * Sinusfunktion f(x) = sin( x), D = R, W = [ 1, -1].
 */
public class Sinus extends DifferenzierbareFunktion
{
```

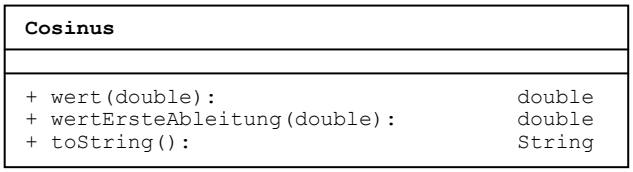
```
/* ------------------------------------------------- */
                                        // service-Methoden
/**
 * Berechnen eines Funktionswertes.
 * @param arg Argument
 * @return sin( arg)
 */
  public double wert( double arg)
  {
// einfache Reihenentwicklung
//  return mySin( arg);
// verbesserte Reihenentwicklung
    return mySinBesser( arg);
  }

/**
 * Berechnen eines Funktionswertes durch Reihenentwicklung:
 * sin x = x - ( x ^ 3) / 3! + ( x ^ 5) / 5! ...
 * @param arg Argument
 * @return sin( arg)
 */
  private double mySin( double arg)
  {
    double x = arg;
    double alteSumme, neueSumme = x, summand = x;
    int j = 1;

    do                                  // Reihenentwicklung
    {
      summand *= -x * x / ++j;             // neuer Summand
      summand /= ++j;

      alteSumme = neueSumme;                  // neue Summe
      neueSumme += summand;
    } while(( neueSumme != alteSumme) && ( j < 1000));

    return alteSumme;
  }

/**
 * Verbesserte Sinusfunktion:
 * Transformationdes Arguments in das Intervall [0,PI/2].
 * @param arg Argument
 * @return sin( arg) verbessert
 */
  private double mySinBesser( double arg)
  {
    double x = arg;
    int s = 1;                                 // Vorzeichen
// x < 0
    if( x < 0)
    {
```

```
        x = -x;
        s = -s;
      }
// x > PI
      if( x > Math.PI)
      {
        int temp = ( int)( x / Math.PI);
        if( temp % 2 != 0) s = -s;       // Vorzeichenwechsel
        x = x - temp * Math.PI;
      }
// x > PI/2
      if( x > Math.PI/2) x = Math.PI - x;

// Trivialfaelle
      if( x == 0 ) return 0;
      if( x == Math.PI / 2) return s;

      return s * mySin( x);
    }

/**
 * Berechnen der ersten Ableitung
 * f'( x) = cos( x) = sin( PI/2 + x).
 * @param arg Argument
 * @return f'( arg)
 */
  public double wertErsteAbleitung( double arg)
  {
    return wert( Math.PI / 2 + arg);
  }

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

5.4.5 Klasse `Cosinus`

Die Wertberechnung erfolgt unter Verwendung der Klasse `Sinus` und der Beziehung:

$$\cos(x) = \sin\left(\frac{\pi}{2} + x\right)$$



Cosinus.java

```
// Cosinus.java                                          MM 2013

/**
 * Cosinusfunktion f(x) = cos( x),
 * D = R, W = [ 1, -1].
 */
public class Cosinus extends DifferenzierbareFunktion
{
/* ------------------------------------------------------ */
                                          // service-Methode
/**
 * Berechnen eines Funktionswertes durch
 * Phasenverschiebuung: cos( x) = sin( PI/2 + x).
 * @param arg Argument
 * @return cos( arg)
 */
  public double wert( double arg)
  {
    Sinus sin = new Sinus();
    return sin.wert( Math.PI/2 + arg);
  }

/**
 * Berechnen der ersten Ableitung f'(x) = -sin( x).
 * @param arg Argument
 * @return f'( arg)
 */
  public double wertErsteAbleitung( double arg)
  {
    Sinus sin = new Sinus();
    return -sin.wert( arg);
  }

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

Ausschnitt aus der Wertetabelle für $x = k \cdot \pi,\ k \in [0,20] \Rightarrow \cos(x) = \pm 1$