

Inhalt

5	Numerische Methoden der praktischen Mathematik (III).....	5-2
5.7	Zusammenfassung.....	5-2
5.8	Einige Anwendungsprogramme für Funktionen.....	5-4
5.8.1	Funktionen im Überblick	5-4
5.8.2	Werteberechnung einer Funktion.....	5-5
5.8.3	Wertetabelle einer Funktion.....	5-9
5.8.4	Integration von Funktionen.....	5-13
5.8.5	Nullstellenbestimmung von Funktionen	5-17

5 Numerische Methoden der praktischen Mathematik (III)

5.7 Zusammenfassung

1. **Eingabefehler** lassen sich in der Regel nur durch die Verbesserung der *Messtechniken verringern*, sind *nie* ganz zu vermeiden. Die Toleranz, in der diese Fehler liegen, entscheidet über die erreichbare Genauigkeit in den Ergebnissen.
2. Mathematisch exakte Verfahren können durch interne **fehlerbehaftete Datendarstellung** und dadurch bedingte **fehlerbehaftete Rechnungen** zu unexakten Ergebnissen führen. Das gilt schon bei der Addition sehr großer Zahlen mit sehr kleinen. Diese Fehler entstehen durch den Widerspruch in der „Unendlichkeit der Mathematik“ und der „Endlichkeit in der Rechentechnik“. Solche **Datendarstellungs- und Rechenfehler** sind auch bei hochentwickelter Computertechnik *nicht* auszuschließen.

Exp.java.

Das Programm lieferte für betragsmäßig große negative Argumente als Ergebnis negative Werte. Die Ursache liegt in der Fehlerfortpflanzung, die sich bei der Reihenentwicklung aus der immer kleiner werdenden Summe durch wechselnde Addition und Subtraktion mit fehlerbehafteten Summanden ergeben.

Sinus.java.

Das Programm lieferte für betragsmäßig große Argumente nicht verwertbare Ergebnisse. Große double-Werte sind wegen der geringen Dichte ihrer Darstellung als Maschinenzahlen mit Rundungsfehlern behaftet, welche sich in der Reihenentwicklung durch Potenzieren dieser zu „Monsterfehlern“ entwickeln.

3. Durch die Einschränkungen auf vorhandene Datentypen sind **Grenzen in den Rechenoperationen** gesetzt. Ergebnisse sind deshalb stets bzgl. ihrer Korrektheit abzuschätzen.

Summe.java.

Die Summenberechnung $s = \sum_{i=1}^n i$ liefert bei long-Zahlen für $n < 2^{32}$ exakte Ergebnisse,

da die Summe s die größte darstellbare long-Zahl $2^{63} - 1$ ($\hat{=}$ **Long.MAX_VALUE**) nicht überschreiten darf. Bei größeren Zahlen wird die Berechnung durch Runden ungenau. Die Wahl eines anderen Elementardatentyps bringt keine Verbesserung. Die Anzahl der exakt dargestellten Ziffern ist bei jedem Zahlentyp begrenzt und liegt bei anderen Datentypen unterhalb der Anzahl der Ziffern einer long-Zahl.

4. **Näherungsverfahren** sind mathematisch unexakte Verfahren. Sie ersetzen komplizierte Verfahren durch einfache und liefern Näherungslösungen. **Verfahrensfehler** können durch Verbesserung der Verfahren eingeschränkt, aber *nicht* vollständig aufgehoben werden.

Integral.java, TrapezIntegral.java, SimpsonIntegral.java.

Bei der bestimmten Integration werden die zu integrierenden Funktionen durch Polynome approximiert, welche sich rechentechnisch besser integrieren lassen. Eine Verbesserung der Ergebnisse konnte durch mehr Stützstellen oder einen höheren Grad des Interpolationspolynoms, also der Verbesserung des verwendeten Verfahrens, erreicht werden.

NewtonIteration.java, RagulaIteration.java.

Man kann Nullstellen von Funktionen mit unterschiedlichen Näherungsverfahren berechnen. Das Newtonverfahren konvergiert sehr schnell, kann aber nur auf

differenzierbare Funktionen angewandt werden. Für stetige, *nicht* differenzierbare, Funktionen bietet sich das Verfahren *Ragula falsi* an, welches langsamer zu einem Ergebnis führt. Wendet man die Verfahren auf Funktionen mit mehreren Nullstellen an, so findet man u. U. *nicht* alle Nullstellen der Funktion. Die Wahl der Anfangsnäherung ist für das Ergebnis ausschlaggebend. Mathematische Verfahren, wie zum Beispiel die Polynomdivision bei einem schon gefundenen Fehler, sind oft unbrauchbar, da der gefundene Fehler eine Näherung darstellt, also nicht exakt ist.

5. Mathematisch elegante Verfahren sind rechentechnisch *nicht* immer die günstigsten. Rechentechnisch optimale Verfahren verbrauchen wenig Rechenzeit. Eine **Rechenzeitminimierung** erreicht man durch Minimierung der Anzahl der verwendeten Operationen.

Polynom.java

Bei der Wertberechnung eines Polynoms vom Grad r ohne Horner-Schema benötigt man $\frac{r^2 + 3r}{2}$ Additionen und Multiplikationen, das sind zum Beispiel bei $r = 10$ insgesamt 65

Operationen. Durch die Verwendung des **Horner-Schemas** konnte dieser Rechenaufwand auf $2r$ Additionen und Multiplikationen verringert werden, bei $r = 10$ sind das nur 20 Operationen, also auf weniger als ein Drittel.

6. Die **modulare Programmierung** gestattet eine mehrfache Verwendung von bereits getesteter Programmbestandteile, hier Klassen und ihre Methoden. In sich abgeschlossene Programmteile werden in einer Klasse zusammengefasst. Diese können getrennt vom Anwendungsprogramm ausgetestet, jederzeit wieder verwendet und leicht durch andere ausgetauscht oder auch erweitert werden.

Funktion.java

Das Modul bildet die Basis aller Funktionen und legt fest, dass jede Funktion ihren Wert berechnen kann. Neue Funktionen können jederzeit als Ableitung dieser Basisklasse aufgenommen werden und damit ein bereits vorhandenes Funktionssystem erweitern.

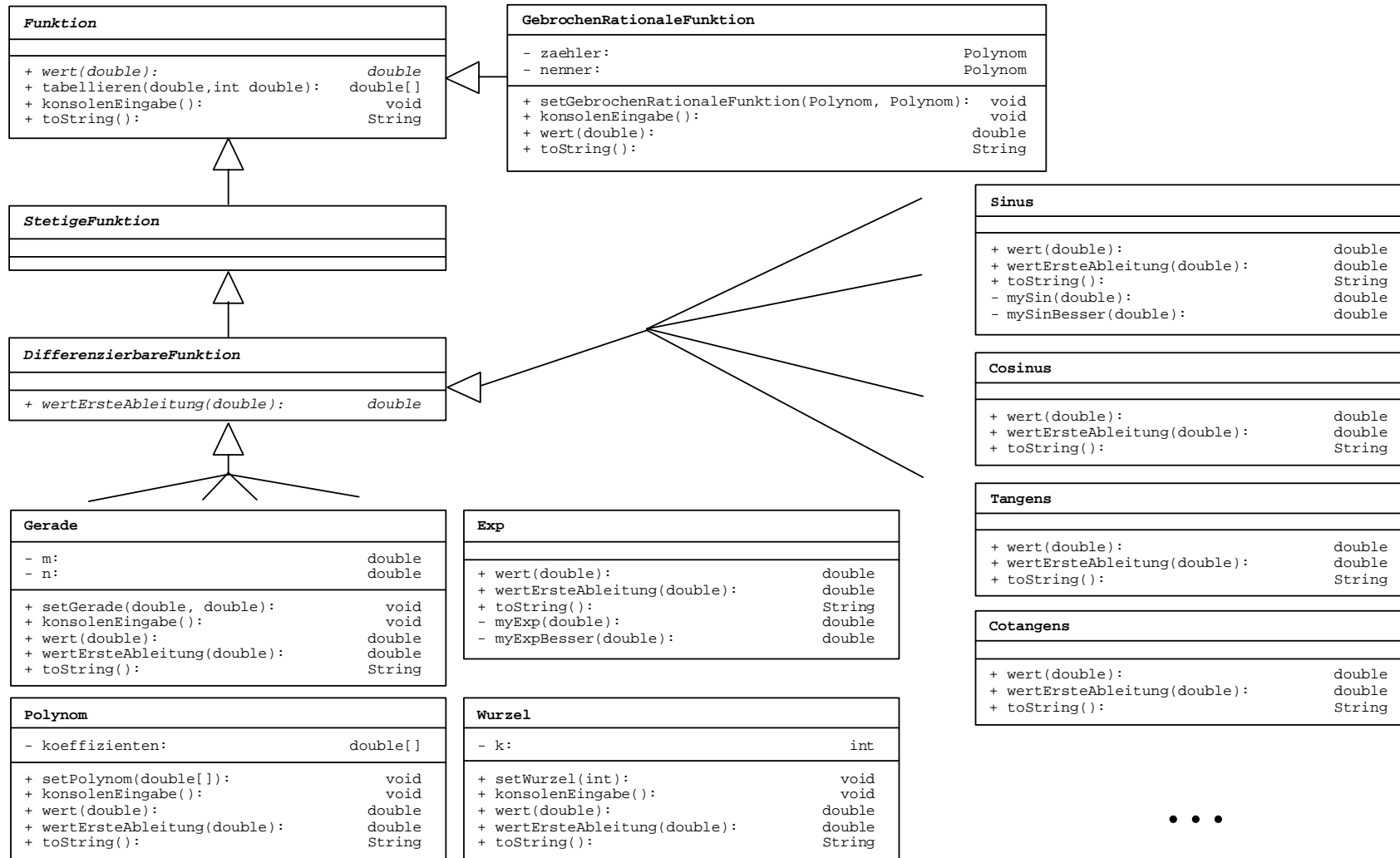
FunktionBerechner.java FunktionTabulator.java,

FunktionsIntegrator.java, NullstellenIterator.java.

Anwendungsprogramme können bereits zur Verfügung stehende Funktionen einbinden, ohne diese neu zu entwickeln. Dazu reicht der *Bytecode* dieser aus.

5.8 Einige Anwendungsprogramme für Funktionen

5.8.1 Funktionen im Überblick



5.8.2 Werteberechnung einer Funktion

In diesem Programm kann der Wert einer Funktion an einer Stelle x_0 und deren 1. Ableitung, falls diese differenzierbar ist, unter Verwendung der in der Vorlesung implementierten Funktionsklassen berechnet werden.

FunktionsBerechner	
- differenzierbar:	boolean
+ dialog():	void
- funktionsAuswahl():	Funktion
- berechneWert(Funktion):	void
+ main(String[]):	static void

FunktionsBerechner.java

```
// FunktionsBerechner.java
import Tools.IO.*;

/**
 * Programm zum Berechnen von Funktionswerten.
 */
public class FunktionsBerechner
{
    /* ----- */
    // Attribut

    /**
     * Speichert Differenzierbarkeit.
     */
    private boolean differenzierbar = false;

    /* ----- */
    // Dialog

    /**
     * Funktionsberechner:
     * Funktionseingabe, Wertberechnung.
     */
    public void dialog()
    {
        // Ueberschrift
        System.out.println();
        System.out.println( "Funktionsberechner" );

        // Dialog
        char weiter = 'j';
        do
        {
            // Neue Funktion
            Funktion fkt = funktionsAuswahl();
            if( fkt == null) break;
            fkt.konsolenEingabe();
        }
    }
}
MM 2010 // Eingaben
```

```
// Funktionsausgabe
    System.out.println();
    System.out.println( "Funktion " + fkt);

    do
    {
// Werteberechnug
        System.out.println();
        berechneWert( fkt);

        System.out.println();

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

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

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

/* ----- */
// Funktion

/**
 * Funktionsauswahl.
 * @return Funktion
 */
private Funktion funktionsAuswahl()
{
// Menue aller Funktionen
    System.out.println( "Funktionsauswahl");
    System.out.println
    ( " Gerade          ... 1");
    System.out.println
    ( " Polynom         ... 2");
    System.out.println
    ( " Wurzel-Funktion ... 3");
    System.out.println
    ( " e^x - Funktion  ... 4");
    System.out.println
    ( " a^x-Funktion    ... 5");
    System.out.println
    ( " sin-Funktion    ... 6");
    System.out.println
    ( " cos-Funktion    ... 7");
    System.out.println
    ( " tan-Funktion    ... 8");
    System.out.println
```

```
( " cot-Funktion          ... 9");
System.out.println
( " arctan-Funktion     ... 10");
System.out.println
( " ln-Funktion         ... 11");
System.out.println
( " log-Funktion        ... 12");
System.out.print
( " gebrochenrationale Funktion ... 13");

int eingabe = IOTools.readInteger(": ");

// Festlegen der Funktion
Funktion fkt = null;
switch( eingabe)
{
    case 1: differenzierbar = true;
            fkt = new Gerade(); break;
    case 2: differenzierbar = true;
            fkt = new Polynom( ); break;
    case 3: differenzierbar = true;
            fkt = new Wurzel(); break;
    case 4: differenzierbar = true;
            fkt = new Exp(); break;
    case 5: differenzierbar = true;
            fkt = new Power(); break;
    case 6: differenzierbar = true;
            fkt = new Sinus(); break;
    case 7: differenzierbar = true;
            fkt = new Cosinus( ); break;
    case 8: differenzierbar = true;
            fkt = new Tangens(); break;
    case 9: differenzierbar = true;
            fkt = new Cotangens(); break;
    case 10: differenzierbar = true;
            fkt = new Arctan(); break;
    case 11: differenzierbar = true;
            fkt = new Ln(); break;
    case 12: differenzierbar = true;
            fkt = new LogA(); break;
    case 13: differenzierbar = false;
            fkt = new RationaleFunktion(); break;
    default: System.out.println();
            System.out.println( "Fehlerhafte Eingabe!");
}

return fkt;
}

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

```
* Berechnen einer Funktion und deren 1. Ableitung
* an einer Stelle x0.
* @param fkt Funktion
*/
private void berechneWert( Funktion fkt)
{
// Argumenteingabe
    System.out.println();
    System.out.println( "Wertberechnung");
    double x0 = IOTools.readDouble( " x0 = ");

// Wertberechnung
    double y0 = fkt.wert( x0);

// Ausgabe
    System.out.print( " f( " + x0 + ") = " + y0);

// Wertberechnung 1. Ableitung
    if( differenzierbar)
    {
        DifferenzierbareFunktion dfkt
        = (DifferenzierbareFunktion)fkt;
        double y1 = dfkt.wertErsteAbleitung( x0);
        System.out.println( "\t\tf'( " + x0 + ") = " + y1);
    }
}

/* ----- */
// Programm
/**
 * Hauptprogramm, startet Funktionsberechnung.
 */
public static void main( String[] args)
{
// Berechner erzeugen
    FunktionsBerechner berechner
    = new FunktionsBerechner();

// Berechner starten
    berechner.dialog();
}

/* ----- */
// Testbeispiele
/*
 * Funktion
 *  $-9.0 + 3.0 x^1 + -5.0 x^2 + 0.0 x^3 + -1.0 x^4 + 2.0 x^5$ 
 *  $P( 3.0) = 360.0$        $P'( 3.0) = 675.0$ 
 *
 * Funktion
 *  $-1.0 + 1.0 x^1 + 1.0 x^2 + 3.0 x^3 + 0.0 x^4$ 

```

```

*   + -2.0 x^5 + 1.0 x^6
*   P( -1.0)  = -1.0           P'( -1.0) = -8.0
*   P( 1.0)   = 3.0           P'( 1.0)  = 8.0
*/

```

5.8.3 Wertetabelle einer Funktion

Tabelliert werden Funktionen unter Verwendung der in der Vorlesung implementierten Funktionsklassen.

FunktionsTabulator	
+ dialog():	void
- funktionsAuswahl():	Funktion
- erzeugeTabelle(Funktion):	void
+ main(String[]):	static void

FunktionsTabulator.java

```

// FunktionsTabulator.java
import Tools.IO.*;
// MM 2010
// Eingaben

/**
 * Programm zum Tabellieren von Funktionen.
 */
public class FunktionsTabulator
{
    /* ----- */
    // Dialog

    /**
     * Funktionstabulator:
     * Funktionseingabe, Wertetabelle.
     */
    public void dialog()
    {
        // Ueberschrift
        System.out.println();
        System.out.println( "Funktionstabulator");

        // Dialog
        char weiter = 'j';
        do
        {
            // Neue Funktion
            Funktion fkt = funktionsAuswahl();
            if( fkt == null) break;
            fkt.konsolenEingabe();

            // Funktionsausgabe
            System.out.println();
        }
    }
}

```

```
        System.out.println( "Funktion " + fkt);

        do
        {
// Wertetabelle
            System.out.println();
            erzeugeTabelle( fkt);

            System.out.println();

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

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

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

/* ----- */
// Funktion

/**
 * Funktionsauswahl.
 * @return Funktion
 */
private Funktion funktionsAuswahl()
{
// Menue aller Funktionen
    System.out.println( "Funktionsauswahl");
    System.out.println
    ( " Gerade                ... 1");
    System.out.println
    ( " Polynom                ... 2");
    System.out.println
    ( " Wurzel-Funktion        ... 3");
    System.out.println
    ( " e^x - Funktion         ... 4");
    System.out.println
    ( " a^x-Funktion           ... 5");
    System.out.println
    ( " sin-Funktion           ... 6");
    System.out.println
    ( " cos-Funktion           ... 7");
    System.out.println
    ( " tan-Funktion           ... 8");
    System.out.println
    ( " cot-Funktion           ... 9");
    System.out.println
    ( " arctan-Funktion        ... 10");
```

```

System.out.println
( "  ln-Funktion          ... 11");
System.out.println
( "  log-Funktion        ... 12");
System.out.print
( "  gebrochenrationale Funktion ... 13");

int eingabe = IOTools.readInteger(": ");

// Festlegen der Funktion
Funktion fkt = null;
switch( eingabe)
{
    case 1: fkt = new Gerade(); break;
    case 2: fkt = new Polynom( ); break;
    case 3: fkt = new Wurzel(); break;
    case 4: fkt = new Exp(); break;
    case 5: fkt = new Power(); break;
    case 6: fkt = new Sinus(); break;
    case 7: fkt = new Cosinus( ); break;
    case 8: fkt = new Tangens(); break;
    case 9: fkt = new Cotangens(); break;
    case 10: fkt = new Arctan(); break;
    case 11: fkt = new Ln(); break;
    case 12: fkt = new LogA(); break;
    case 13: fkt = new RationaleFunktion(); break;
    default: System.out.println();
        System.out.println( "Fehlerhafte Eingabe!");
}

return fkt;
}

/* ----- */
// Tabellieren
/**
 * Erzeugen und Ausgeben einer Wertetabelle.
 * @param fkt Funktion
 */
private void erzeugeTabelle( Funktion fkt)
{
// Eingabe der Tabellenparameter
System.out.println( "Wertbereich");

double x0
= IOTools.readDouble( " Startargument x0 = ");

int n = 1;
do
{
    n = IOTools.readInteger( " Anzahl n (n > 0) = ");
} while( n < 1);

```

```
double h
= IOTools.readDouble( " Schrittweite h = ");

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

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

/* ----- */
// Programm
/**
 * Hauptprogramm, startet Tabellieren von Funktionen.
 */
public static void main( String[] args)
{
// Tabulator erzeugen
    FunktionsTabulator tabulator
    = new FunktionsTabulator();

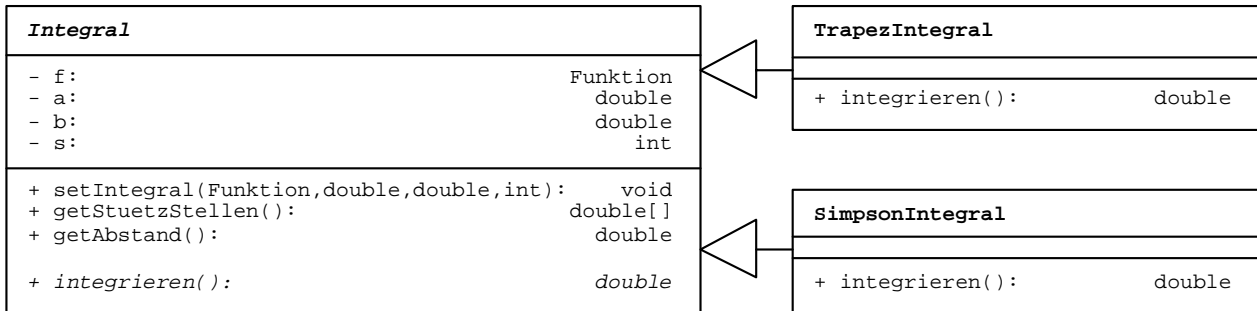
// Tabulator starten
    tabulator.dialog();
}

/* ----- */
// Testbeispiel
/*
 * Funktion
 *  $8.0 + 4.0 x^1 + -10.0 x^2 + -5.0 x^3 + 2.0 x^4 + 1.0 x^5$ 
 *
 * Wertbereich
 * Startargument x0 = -2.5
 * Anzahl n (n > 0) = 11
 * Schrittweite h = 0.5
 * F( -2.5) = -5.90625
 * F( -2.0) = 0.0
 * F( -1.5) = -1.09375
 * F( -1.0) = 0.0
 * F( -0.5) = 4.21875
 * F( 0.0) = 8.0
 * F( 0.5) = 7.03125
 * F( 1.0) = 0.0
 * F( 1.5) = -7.65625
 */
```

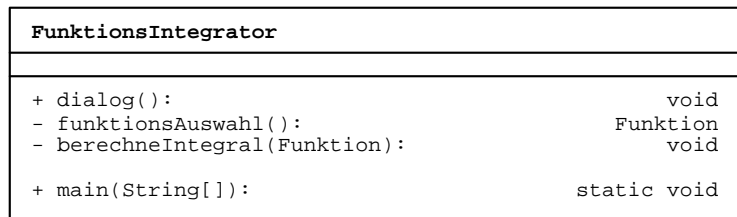
```
* F( 2.0) = 0.0
* F( 2.5) = 53.15625
*/
```

5.8.4 Integration von Funktionen

In der Vorlesung wurden zwei Integrationsverfahren behandelt. Beide Verfahren approximieren die zu integrierende Funktion als Polynom. Im Trapezverfahren wird ein lineares und im Simpsonverfahren ein quadratisches Polynom integriert.



Das Programm integriert Funktionen unter Verwendung der in der Vorlesung implementierten Funktionsklassen wahlweise mittels der behandelten Integrationsverfahren.



FunktionsIntegrator.java

```
// FunktionsIntegrator.java
import Tools.IO.*;
// Eingaben

/**
 * Programm zur Integrieren von Funktionen.
 */
public class FunktionsIntegrator
{
    /* ----- */
    // Dialog

    /**
     * Funktionsberechner:
     * Funktionseingabe, Wertberechnung.
     */
    public void dialog()
    {
        // Ueberschrift
        System.out.println();
        System.out.println( "Funktionsintegrator");
    }
}

```

```
// Dialog
    char weiter = 'j';
    do
    {
// Neue Funktion
        Funktion fkt = funktionsAuswahl();
        if( fkt == null) break;
        fkt.konsolenEingabe();

// Funktionsausgabe
        System.out.println();
        System.out.println( "Integral von " + fkt);

        do
        {
// Integralberechnug
            System.out.println();
            berechneIntegral( fkt);

            System.out.println();

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

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

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

/* ----- */
// Funktion

/**
 * Funktionsauswahl.
 * @return Funktion
 */
private Funktion funktionsAuswahl()
{
// Menue aller Funktionen
    System.out.println( "Funktionsauswahl");
    System.out.println
    ( " Gerade          ... 1");
    System.out.println
    ( " Polynom         ... 2");
    System.out.println
    ( " Wurzel-Funktion ... 3");
    System.out.println
    ( " e^x - Funktion  ... 4");
```

```

System.out.println
( "  a^x-Funktion      ...  5");
System.out.println
( "  sin-Funktion     ...  6");
System.out.println
( "  cos-Funktion     ...  7");
System.out.println
( "  tan-Funktion     ...  8");
System.out.println
( "  cot-Funktion     ...  9");
System.out.println
( "  arctan-Funktion  ... 10");
System.out.println
( "  ln-Funktion      ... 11");
System.out.print
( "  log-Funktion     ... 12");

int eingabe = IOTools.readInteger(": ");

// Festlegen der Funktion
Funktion fkt = null;
switch( eingabe)
{
    case 1: fkt = new Gerade(); break;
    case 2: fkt = new Polynom( ); break;
    case 3: fkt = new Wurzel(); break;
    case 4: fkt = new Exp(); break;
    case 5: fkt = new Power(); break;
    case 6: fkt = new Sinus(); break;
    case 7: fkt = new Cosinus( ); break;
    case 8: fkt = new Tangens(); break;
    case 9: fkt = new Cotangens(); break;
    case 10: fkt = new Arctan(); break;
    case 11: fkt = new Ln(); break;
    case 12: fkt = new LogA(); break;
    default: System.out.println();
             System.out.println( "Fehlerhafte Eingabe!");
}

return fkt;
}

/* ----- */
// Integrieren

/**
 * Integralberechnung.
 * @param fkt Funktion
 */
private void berechneIntegral( Funktion fkt)
{
// Eingabe der Integralparameter
double a, b;

```

```
do
{
    System.out.println();
    a = IOTools.readDouble( " Untere Grenze a = ");
    System.out.println( a );
    b = IOTools.readDouble( " Obere Grenze b = ");
    System.out.println( b );
} while( a >= b );

// Eingaben der Stuetzstellen
int s;
do
{
    System.out.print
    ( " Anzahl der Stuetzstellen s > 1, " );
    s = IOTools.readInteger( "s = ");
    System.out.println( s );
} while( s < 2 );

// Trapez
TrapezIntegral trapez = new TrapezIntegral();
trapez.setIntegral( fkt, a, b, s );

// Integralberechnung
double ergTrapez = trapez.integrieren();
System.out.println( " Trapez: F = " + ergTrapez );

// Simpson
if( s % 2 != 0 )
{
    SimpsonIntegral simpson = new SimpsonIntegral();
    simpson.setIntegral( fkt, a, b, s );

// Integralberechnung
double ergSimpson = simpson.integrieren();
System.out.println( " Simpson: F = " + ergSimpson );
}
}

/* ----- */
// Programm
/**
 * Hauptprogramm, startet Integration von Funktionen.
 */
public static void main( String[] args )
{
// Integrator erzeugen
FunktionsIntegrator integrator
    = new FunktionsIntegrator();

// Integrator starten
integrator.dialog();
```

```

    }
}

/* ----- */
// Testwerte
// Integral von 0 bis 1 ueber ( 1.0 + 1.0 x^1 )
// Stuetzstellen: 5
// Trapez : 1.5
// Simpson: 1.5

// Integral von 0 bis 1 ueber ( 1.0 ) / ( 1.0 + 1.0 x^1 )
// Stuetzstellen: 3
// Trapez : 0.7083333333333333
// Simpson: 0.6944444444444443
// Stuetzstellen: 7
// Trapez : 0.6948773448773449
// Simpson: 0.6931697931697932

```

5.8.5 Nullstellenbestimmung von Funktionen

In der Vorlesung wurden zwei Verfahren zur Nullstellenbestimmung von Funktionen behandelt. Beide Verfahren sind Iterationsverfahren, die zu einer bereits vorhandenen Nullstellennäherungen eine bessere sucht. Das Newtonverfahren konvergiert schneller an die eigentliche Lösung, verlangt aber eine differenzierbare Funktion. Das Verfahren Regula falsi verarbeitet stetige Funktionen.

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

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

Das Programm bestimmt Nullstellennäherungen von differenzierbaren Funktionen unter Verwendung der in der Vorlesung implementierten Funktionsklassen wahlweise mittels der beiden behandelten Verfahren.

NullstellenIterator	
+ dialog():	void
- funktionsAuswahl():	Funktion
- berechneNullstelle(Funktion):	void
- newton(DifferenzierbareFunktion):	double
- regulaFalsi(Funktion):	double
+ main(String[]):	static void

NullstellenIterator.java

```
// NullstellenIteration.java
import Tools.IO.*;

/**
 * Programm zur Nullstellenberechnung von Funktionen.
 */
public class NullstellenIterator
{
    /**
     * ----- */
    // Attribut

    /**
     * Speichert Differenzierbarkeit.
     */
    private boolean differenzierbar = false;

    /**
     * ----- */
    // Dialog

    /**
     * Nullstellenberechner:
     * Funktionseingabe, Nullstellenberechnung.
     */
    public void dialog()
    {
        // Ueberschrift
        System.out.println();
        System.out.println( "Nullstellenberechner" );

        // Dialog
        char weiter = 'j';
        do
        {
            // Neue Funktion
            Funktion fkt = funktionsAuswahl();
            if( fkt == null ) break;
            fkt.konsolenEingabe();

            // Funktionsausgabe
            System.out.println( "\nNullstelle von: " + fkt );

            do
            {
```

```
// Nullstellenberechnung
    System.out.println();
    berechneNullstelle( fkt);

    System.out.println();

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

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

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

/* ----- */
// Funktionseingabe
/**
 * Funktionsauswahl.
 * @return Funktion
 */
private Funktion funktionsAuswahl()
{
// Menue aller Funktionen
    System.out.println( "Funktionsauswahl");
    System.out.println
    ( " Gerade          ... 1");
    System.out.println
    ( " Polynom         ... 2");
    System.out.print
    ( " gebrochenrationale Funktion ... 3");

    int eingabe = IOTools.readInteger(": ");

// Festlegen der Funktion
    Funktion fkt = null;
    switch( eingabe)
    {
        case 1: fkt = new Gerade();
                differenzierbar = true;
                break;
        case 2: fkt = new Polynom( );
                differenzierbar = true;
                break;
        case 3: fkt = new RationaleFunktion();
                differenzierbar = false;
                break;
        default: System.out.println();
    }
}
```

```
        System.out.println( "Fehlerhafte Eingabe!");
    }
    return fkt;
}

/* ----- */
// Iteration

/**
 * Verfahrensauswahl.
 * @param fkt differenzierbare Funktion
 */
private void berechneNullstelle( Funktion fkt)
{
// Berechnen der Nullstelle durch Newton
    if( differenzierbar)
    {
        DifferenzierbareFunktion dfkt
        = (DifferenzierbareFunktion)fkt;

        System.out.println
        ( " Newton:          " + newton( dfkt));

        System.out.println();
    }

// Berechnen des Integrals durch Simpsonregel
    System.out.println
    ( " Regula falsi: " + regulaFalsi( fkt));
}

/**
 * Berechnet Nullstelle nach Newton.
 * @param fkt DifferenzierbareFunktion
 * @return Nullstellennaeherung
 */
private double newton( DifferenzierbareFunktion fkt)
{
    System.out.println( "Newton:");
    System.out.println( "Eingabe der Iterationswerte");

    double x0 = IOTools.readDouble
        ( " Startwert    x0 = ");
    double eps = IOTools.readDouble
        ( " Genauigkeit eps = ");

    int max;
    do
    {
        max = IOTools.readInteger
            ( " maximale Durchlaeufe > 0, max = ");
    } while( max < 1);

    NewtonIteration newton = new NewtonIteration();
}
```

```
        newton.setNewtonIteration( fkt, x0, eps, max);

        return newton.iterieren();
    }

/**
 * Berechnet Nullstelle nach Regula Falsi.
 * @param fkt Funktion
 * @return Nullstellennaeherung
 */
private double regulaFalsi( Funktion fkt)
{
    System.out.println( "Regula falsi:");
    System.out.println( "Eingabe der Iterationswerte");

    double x0 = 0, x1 = 0;
    do
    {
        x0 = IOTools.readDouble
            ( " 1. Naehierungswert, f(x0)>0,  x0 = ");
        x1 = IOTools.readDouble
            ( " 2. Naehierungswert, f(x1)<0,  x1 = ");
    } while( fkt.wert( x0) <= 0 || fkt.wert( x1) >= 0);

    double eps = IOTools.readDouble
        ( " Genauigkeit,                eps = ");
    int max;
    do
    {
        max = IOTools.readInteger
            ( " maximale Durchlaeufe > 0,    max = ");
    } while( max < 1);

    RegulaFalsiIteration rf = new RegulaFalsiIteration();
    rf.setRegulaFalsiIteration( fkt, x0, x1, eps, max);

    return rf.iterieren();
}

/* ----- */
// Programm

/**
 * Hauptprogramm, startet Nullstellenberechnung.
 */
public static void main( String[] args)
{
    // Berechner erzeugen
    NullstellenIterator berechner
        = new NullstellenIterator();

    // Berechner starten
    berechner.dialog();
}
```

```
}
}

/* ----- */
/*                                     // Testwerte
/*
Nullstellenberechner
Nullstelle von:
24.0 + -2.0 x^1 + -5.0 x^2 + 1.0 x^3

Mathematische Loesung: 4, 3, -2

Newton:
Eingabe der Iterationswerte
  Startwert   x0 = -3
  Genauigkeit eps = 0.0001
  maximale Durchlaeufer > 0, max = 10
0: -3.0
1: -2.2363636363636363
2: -2.01812267760654
3: -2.0001192331463127

Gefundene Nullstelle: -2.0000000052123896           // -2

Eingabe der Iterationswerte
  Startwert   x0 = 0
  Genauigkeit eps = 0.0001
  maximale Durchlaeufer > 0, max = 10
0: 0.0
1: 12.0
2: 8.748387096774193
3: 6.654571080541162
4: 5.348891161080276
5: 4.581524151017727
6: 4.182140953086981
7: 4.028245849809502
8: 4.000879964323257

Gefundene Nullstelle: 4.000000901768636           // 4

Eingabe der Iterationswerte
  Startwert   x0 = 1
  Genauigkeit eps = 0.0001
  maximale Durchlaeufer > 0, max = 10
0: 1.0

Gefundene Nullstelle: 3.0                         // 3
*/
```