

# Algorithmensammlung

Walter Fendt

24. März 2024

(zuletzt geändert 20. April 2024)

## Inhaltsverzeichnis

<b>0</b>	<b>Vorbemerkung</b>	<b>3</b>
<b>1</b>	<b>Ganze Zahlen</b>	<b>4</b>
1.1	Grundlagen . . . . .	4
1.2	Primzahltest . . . . .	5
1.3	Primfaktorzerlegung . . . . .	6
1.4	Größter gemeinsamer Teiler . . . . .	7
1.5	Kleinstes gemeinsames Vielfaches . . . . .	8
1.6	Fakultät . . . . .	9
1.7	Binomialkoeffizient . . . . .	10
1.8	Eulersche Phi-Funktion . . . . .	11
1.9	Jacobi-Symbol . . . . .	13
<b>2</b>	<b>Rationale Zahlen</b>	<b>15</b>
2.1	Grundlagen . . . . .	15
2.2	Addition von rationalen Zahlen . . . . .	17
2.3	Subtraktion von rationalen Zahlen . . . . .	19
2.4	Multiplikation von rationalen Zahlen . . . . .	20
2.5	Division von rationalen Zahlen . . . . .	22
2.6	Umwandlung in einen Dezimalbruch . . . . .	24
<b>3</b>	<b>Lineare Algebra</b>	<b>27</b>
3.1	Vektoren, Grundlagen . . . . .	27
3.2	Addition von Vektoren . . . . .	29
3.3	Subtraktion von Vektoren . . . . .	30
3.4	Skalare Multiplikation von Vektoren . . . . .	31
3.5	Skalarprodukt von Vektoren . . . . .	32
3.6	Kreuzprodukt von Vektoren . . . . .	33
3.7	Matrizen, Grundlagen . . . . .	34

3.8	Rang einer Matrix . . . . .	36
3.9	Addition von Matrizen . . . . .	38
3.10	Subtraktion von Matrizen . . . . .	39
3.11	Multiplikation einer Matrix mit einem Vektor . . . . .	40
3.12	Multiplikation von Matrizen . . . . .	41
3.13	Gauß-Jordan-Algorithmus . . . . .	42
3.14	Inverse Matrix . . . . .	44
3.15	Determinante . . . . .	46
3.16	Charakteristisches Polynom . . . . .	48
<b>4</b>	<b>Univariate Polynome</b>	<b>49</b>
4.1	Grundlagen . . . . .	49
4.2	Horner-Schema . . . . .	51
4.3	Addition von univariaten Polynomen . . . . .	52
4.4	Subtraktion von univariaten Polynomen . . . . .	53
4.5	Multiplikation von univariaten Polynomen . . . . .	54
4.6	Division von univariaten Polynomen (mit Rest) . . . . .	56
<b>5</b>	<b>Multivariate Polynome</b>	<b>58</b>
5.1	Monome, Grundlagen . . . . .	58
5.2	Addition von Monomen . . . . .	60
5.3	Multiplikation von Monomen . . . . .	61
5.4	Division von Monomen . . . . .	62
5.5	Multivariate Polynome, Grundlagen . . . . .	63
5.6	Addition von multivariaten Polynomen . . . . .	65
5.7	Subtraktion von multivariaten Polynomen . . . . .	66
5.8	Multiplikation von multivariaten Polynomen . . . . .	67
5.9	Leitmonom eines multivariaten Polynoms . . . . .	69
5.10	Division von multivariaten Polynomen . . . . .	70
<b>6</b>	<b>Integration</b>	<b>71</b>
6.1	Trapezregel . . . . .	71
6.2	Simpson-Regel . . . . .	71
6.3	Romberg-Verfahren . . . . .	72
<b>7</b>	<b>Sonstiges</b>	<b>73</b>
7.1	Gaußsche Osterformel, Version von 1816 . . . . .	73

## 0 Vorbemerkung

Diese Sammlung von Algorithmen entstand aus mehreren Programmierprojekten und meiner Mitarbeit am Wikibook „Algorithmensammlung“. Im Wesentlichen handelt es sich um grundlegende Algorithmen aus mehreren Gebieten der Mathematik. Die Algorithmen sind nicht hinsichtlich der Rechenzeit optimiert, da mir die Verständlichkeit wichtiger erschien. Sie sind als Java-Methoden formuliert (bevorzugt als Klassenmethoden). Die zugehörigen Java-Klassen sind `NumberQ` (rationale Zahl), `VectorQ` (Vektor mit rationalen Koordinaten), `MatrixQ` (Matrix mit rationalen Elementen), `PolynomialQ` (univariates Polynom mit rationalen Koeffizienten) und `MultivariateQ` (multivariates Polynom mit rationalen Koeffizienten). In etlichen Fällen sind Pseudocode-Beschreibungen der Algorithmen hinzugefügt. Eine Erweiterung der Sammlung ist geplant.

# 1 Ganze Zahlen

## 1.1 Grundlagen

Folgende Rechenoperationen und Funktionen werden vorausgesetzt:

Addition (+), Subtraktion (−), Multiplikation ( $\cdot$ ), ganzzahlige Division (div oder /),  
Divisionsrest (mod); selbstverständlich keine Division durch 0

Maximum zweier Zahlen (max), Minimum zweier Zahlen (min)

## 1.2 Primzahltest

Für eine natürliche Zahl  $n$  soll überprüft werden, ob es sich um eine Primzahl handelt.

**Pseudocode-Funktion isPrime:**

```
function isPrime( $n$ )
  // Vorausgesetzt: Natürliche Zahl  $n$ 
  if  $n \leq 1$  then return false
   $d \leftarrow 2$ 
  while  $d \cdot d \leq n$  do
    if  $\text{mod}(n, d) = 0$  then return false
     $d \leftarrow d + 1$ 
  return true
```

**Java-Methode isPrime:**

Verwendete Klasse: `java.math.BigInteger`

```
// Überprüfung der Primzahleigenschaft
// n ... Gegebene Zahl (ganz)

public static boolean isPrime (BigInteger n) {
    BigInteger one = BigInteger.ONE;
    if (n.compareTo(one) <= 0) return false;
    BigInteger d = BigInteger.valueOf(2);
    while (d.multiply(d).compareTo(n) <= 0) {
        if (n.mod(d).signum() == 0) return false;
        d = d.add(one);
    }
    return true;
}
```

### 1.3 Primfaktorzerlegung

Eine natürliche Zahl  $n$  soll in ihre Primfaktoren zerlegt werden.

#### Pseudocode-Funktion primeFactorization:

```
function primeFactorization( $n$ )
  // Vorausgesetzt: Natürliche Zahl  $n$ 
   $f \leftarrow$  Leere Liste
  if  $n = 1$  then return  $f$ 
   $d \leftarrow 2$ 
  while  $d \cdot d \leq n$  do
    if  $\text{mod}(n, d) = 0$  then
      [
        Füge  $d$  zur Liste  $f$  hinzu
         $n \leftarrow n/d$ 
      ]
    else  $d \leftarrow d + 1$ 
  Füge  $n$  zur Liste  $f$  hinzu
  return  $f$ 
```

#### Java-Methode primeFactorization:

Verwendete Klassen: java.math.BigInteger, java.util.Vector

```
// Zerlegung in Primfaktoren (Probedivisionen):
// n ... Gegebene Zahl (natürlich)
// Rückgabewert: Liste der Primfaktoren (eventuell mehrfach vorkommend,
// aufsteigend sortiert)

public static Vector<BigInteger> primeFactorization (BigInteger n) {
  Vector<BigInteger> f = new Vector<BigInteger>();
  BigInteger one = BigInteger.ONE;
  if (n.compareTo(one) == 0) return f;
  BigInteger d = BigInteger.valueOf(2);
  while (d.multiply(d).compareTo(n) <= 0) {
    if (n.mod(d).signum() == 0) {
      f.add(d);
      n = n.divide(d);
    }
    else d = d.add(one);
  }
  f.add(n);
  return f;
}
```

## 1.4 Größter gemeinsamer Teiler

Zu zwei natürlichen Zahlen  $m$  und  $n$  soll der größte gemeinsame Teiler berechnet werden (Funktion `gcd`, engl. *greatest common divisor*). Dazu verwendet man sinnvollerweise den euklidischen Algorithmus.

### Pseudocode-Funktion `gcd`:

```
function gcd( $m, n$ )  
  // Vorausgesetzt: Natürliche Zahlen  $m$  und  $n$   
   $a \leftarrow m$   
   $b \leftarrow n$   
  while  $b \neq 0$  do  
    [  
       $c \leftarrow \text{mod}(a, b)$   
       $a \leftarrow b$   
       $b \leftarrow c$   
    ]  
  return  $a$ 
```

### Java-Methode `gcd`:

Verwendete Klasse: `java.math.BigInteger`

Diese Klasse besitzt bereits eine Methode `gcd` für den größten gemeinsamen Teiler.

```
// Größter gemeinsamer Teiler:  
// m, n ... Gegebene Zahlen (natürliche Zahlen)  
  
public static BigInteger gcd (BigInteger m, BigInteger n) {  
    return m.gcd(n);  
}
```

## 1.5 Kleinstes gemeinsames Vielfaches

Die Berechnung des kleinsten gemeinsamen Vielfachen (Funktion lcm, engl. *least common multiple*) kann auf die Bestimmung des größten gemeinsamen Teilers (Funktion gcd) zurückgeführt werden.

### Pseudocode-Funktion lcm:

```
function lcm( $m, n$ )  
  // Vorausgesetzt: Natürliche Zahlen  $m$  und  $n$   
   $d \leftarrow \text{gcd}(m, n)$   
  return  $(m/d) \cdot n$ 
```

### Java-Methode lcm:

Verwendete Klasse: java.math.BigInteger

```
// Kleinstes gemeinsames Vielfaches:  
// m, n ... Gegebene Zahlen (natürliche Zahlen)  
  
public static BigInteger lcm (BigInteger m, BigInteger n) {  
    BigInteger d = m.gcd(n);  
    return m.divide(d).multiply(n);  
}
```



## 1.6 Fakultät

Für eine nicht-negative ganze Zahl  $n$  soll die Fakultät  $n!$  berechnet werden.

### Pseudocode-Funktion factorial:

```
function factorial( $n$ )  
  // Vorausgesetzt: Nicht-negative ganze Zahl  $n$   
   $f \leftarrow 1$   
  for  $i \leftarrow 1$  to  $n$  do  
     $f \leftarrow f \cdot i$   
  return  $f$ 
```

### Java-Methode factorial:

Verwendete Klasse: `java.math.BigInteger`

```
// Fakultät:  
// n ... Argument (natürliche Zahl oder 0)  
  
public static BigInteger factorial (int n) {  
    BigInteger f = BigInteger.ONE;  
    for (int i=2; i<=n; i++)  
        f = f.multiply(BigInteger.valueOf(i));  
    return f;  
}
```

## 1.7 Binomialkoeffizient

Zu zwei nicht-negativen Zahlen  $n$  und  $k$  soll der Binomialkoeffizient  $\binom{n}{k}$  berechnet werden.

**Pseudocode-Funktion binomialCoefficient:**

```
function binomialCoefficient( $n, k$ )
  // Vorausgesetzt: Nicht-negative ganze Zahlen  $n$  und  $k$ 
  if  $k > n$  then return 0
   $k' \leftarrow \min(k, n - k)$ 
   $b \leftarrow 1$ 
  for  $i \leftarrow 1$  to  $k'$  do
     $b \leftarrow b \cdot (n + 1 - i) / i$ 
  return  $b$ 
```

**Java-Methode binomialCoefficient:**

Verwendete Klasse: java.math.BigInteger

```
// Binomialkoeffizient:
// n ... obere Zahl (natürlich oder 0)
// k ... untere Zahl (natürlich oder 0)

public static BigInteger binomialCoefficient (int n, int k) {
  if (k > n) return BigInteger.ZERO;
  k = Math.min(k,n-k);
  BigInteger b = BigInteger.ONE;
  for (int i=1; i<=k; i++) {
    b = b.multiply(BigInteger.valueOf(n+1-i));
    b = b.divide(BigInteger.valueOf(i));
  }
  return b;
}
```

## 1.8 Eulersche Phi-Funktion

Die eulersche Phi-Funktion ( $\varphi$ -Funktion) gibt für eine natürliche Zahl  $n$  an, wie viele der Zahlen  $1, \dots, n$  zu  $n$  teilerfremd sind. Zur Berechnung von  $\varphi(n)$  kann man folgende Eigenschaften der Phi-Funktion verwenden:

- Für eine Primzahlpotenz  $p^k$  gilt  $\varphi(p^k) = p^k - p^{k-1}$ .
- Für teilerfremde Zahlen  $m, n$  gilt  $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ .

### Java-Methode `firstPrime`:

Zunächst wird eine Hilfsmethode `firstPrime` formuliert, die für eine natürliche Zahl  $n > 1$  den kleinsten Primfaktor liefert.

```
// Hilfsmethode: Kleinster Primfaktor
// n ... Gegebene Zahl (natürlich, größer als 1)

private static BigInteger firstPrime (BigInteger n) throws Exception {
    if (n.compareTo(BigInteger.ONE) <= 0)
        throw new Exception("Argument unzulässig!");
    BigInteger d = BigInteger.valueOf(2);
    while (d.multiply(d).compareTo(n) <= 0) {
        if (n.mod(d).signum() == 0) return d;
        d = d.add(BigInteger.ONE);
    }
    return n;
}
```

### Java-Methode eulerPhi:

```
// Eulersche Phi-Funktion:  
// n ... Gegebene Zahl (natürlich)  
  
public static BigInteger eulerPhi (BigInteger n) throws Exception {  
    if (n.signum() <= 0)  
        throw new Exception("Nur natürliche Zahlen zulässig!");  
    BigInteger phi = BigInteger.ONE;  
    while (n.compareTo(BigInteger.ONE) > 0) {  
        BigInteger p = firstPrime(n);  
        BigInteger p1 = BigInteger.ONE;  
        while (n.mod(p).signum() == 0) {  
            p1 = p1.multiply(p);  
            n = n.divide(p);  
        }  
        BigInteger f = p1.subtract(p1.divide(p));  
        phi = phi.multiply(f);  
    }  
    return phi;  
}
```

## 1.9 Jacobi-Symbol

Das Legendre-Symbol  $\left(\frac{a}{b}\right)$  und das ebenso geschriebene Jacobi-Symbol spielen in der Zahlentheorie eine wichtige Rolle. Beim Legendre-Symbol wird vorausgesetzt, dass  $b$  eine ungerade Primzahl ist.

$$\left(\frac{a}{b}\right) = \begin{cases} 1 & \text{wenn } a \text{ quadratischer Rest modulo } b \\ -1 & \text{wenn } a \text{ quadratischer Nichtrest modulo } b \\ 0 & \text{wenn } b \text{ Teiler von } a \end{cases}$$

Das allgemeinere Jacobi-Symbol (für ungerades natürliches  $b$ ) lässt sich mithilfe der Primfaktorzerlegung auf das Legendre-Symbol zurückführen. Für  $b = p_1^{e_1} \cdot \dots \cdot p_k^{e_k}$  gilt, wenn  $p_1, \dots, p_k$  verschiedene (ungerade) Primzahlen und  $e_1, \dots, e_k \in \mathbb{N}$  die zugehörigen Exponenten sind:

$$\left(\frac{a}{b}\right) = \left(\frac{a}{p_1}\right)^{e_1} \cdot \dots \cdot \left(\frac{a}{p_k}\right)^{e_k}$$

Allerdings ist diese Art der Berechnung bei extrem großen Zahlen sehr aufwendig. Der folgende Algorithmus beruht auf dem quadratischen Reziprozitätsgesetz (entdeckt von Leonhard Euler, bewiesen von Carl Friedrich Gauß) und dem zugehörigen 2. Ergänzungssatz.

**Pseudocode-Funktion jacobi:**

```
function jacobi( $a, b$ )
  // Vorausgesetzt: Ganze Zahl  $a$ , ungerade natürliche Zahl  $b$ 
   $x \leftarrow a$ 
   $y \leftarrow b$ 
   $j \leftarrow 1$ 
  if  $y = 1$  then return 1
  while  $x \neq 1$  do
    Berechne  $q \in \mathbb{Z}, r \in \mathbb{N}_0$ , wobei  $x = q \cdot y + r$  und  $0 \leq r < y$ 
    if  $r = 0$  then return 0
    Berechne  $k, x' \in \mathbb{N}_0$ , wobei  $r = 2^k \cdot x'$  und  $x'$  ungerade
    if  $k \equiv 1 \pmod{2}$  and  $y \equiv \pm 3 \pmod{8}$  then  $j \leftarrow -j$ 
    if  $x' = 1$  then return  $j$ 
    if  $\frac{x'-1}{2} \cdot \frac{y-1}{2} \equiv 1 \pmod{2}$  then  $j \leftarrow -j$ 
     $x \leftarrow y$ 
     $y \leftarrow x'$ 
  return  $j$ 
```

Quelle: [de.wikipedia.org/wiki/Jacobi-Symbol](https://de.wikipedia.org/wiki/Jacobi-Symbol)

## Java-Methode jacobi:

```
public static final BigInteger ONE = BigInteger.ONE;
public static final BigInteger TWO = BigInteger.valueOf(2);
public static final BigInteger FOUR = BigInteger.valueOf(4);
public static final BigInteger EIGHT = BigInteger.valueOf(8);

// Jacobi-Symbol:
// a ... Obere Zahl (ganz)
// b ... Untere Zahl (natürlich, ungerade)

public static int jacobi (BigInteger a, BigInteger b) {
    BigInteger x = a, y = b;
    if (y.compareTo(ONE) == 0) return 1;
    int j = 1;
    while (x.compareTo(ONE) != 0) {
        BigInteger r = x.mod(y);
        if (r.signum() == 0) return 0;
        int k = 0;
        while (r.mod(TWO).signum() == 0) {
            r = r.divide(TWO);
            k++;
        }
        int r8 = y.mod(EIGHT).intValue();
        if (k%2 != 0 && (r8 == 3 || r8 == 5)) j = -j;
        if (r.compareTo(ONE) == 0) return j;
        BigInteger e1 = r.subtract(ONE);
        BigInteger e2 = y.subtract(ONE);
        BigInteger e = e1.multiply(e2).divide(FOUR);
        if (e.mod(TWO).signum() != 0) j = -j;
        x = y; y = r;
    } // Ende while
    return j;
}
```

## 2 Rationale Zahlen

### 2.1 Grundlagen

Darstellung als Verbund von zwei ganzen Zahlen, *num* (Zähler, engl. *numerator*) und *denom* (Nenner, engl. *denominator*).

Normalisierung: Zähler und Nenner teilerfremd (vollständig kürzen!), Nenner positiv

**Pseudocode-Funktion numberQ:**

```
function numberQ(n, d)
  // Vorausgesetzt: Ganze Zahlen n (Zähler) und d (Nenner,  $d \neq 0$ )
  if  $d = 0$  then return undefined
   $g \leftarrow \text{gcd}(n, d)$  // größter gemeinsamer Teiler
  if  $d < 0$  then  $g \leftarrow -g$ 
   $n \leftarrow n/g$ 
   $d \leftarrow d/g$ 
  return {num : n, denom : d}
```

### Java-Klasse NumberQ

Die Klasse `java.math.BigInteger` wird vorausgesetzt. Für rationale Zahlen wird eine neue Klasse `NumberQ` entwickelt. Die folgenden Zeilen zeigen die Attribute der Klasse (`num` und `denom`), zwei einfache Konstruktoren und die grundlegende Methode `normal` zur Normalisierung (vollständig gekürzter Bruch, Nenner positiv).

```
public class NumberQ {

    BigInteger num;                // Zähler
    BigInteger denom;             // Nenner

    // Konstruktor:

    public NumberQ (BigInteger n, BigInteger d) throws Exception {
        if (d.signum() == 0) throw new Exception("Nenner gleich 0");
        num = n; denom = d;
        normal();
    }

    // Spezieller Konstruktor für die Zahl 0:

    public NumberQ () {
        num = BigInteger.ZERO; denom = BigInteger.ONE;
    }
}
```

```

// Normalisierung:

public static NumberQ normal (BigInteger n, BigInteger d) {
    if (d.signum() == 0) return null;
    BigInteger g = n.gcd(d);
    if (d.signum() < 0) g = g.negate();
    n = n.divide(g); d = d.divide(g);
    NumberQ q = new NumberQ();
    q.num = n; q.denom = d;
    return q;
}

// Kleinstes gemeinsames Vielfaches:

public static BigInteger lcm (BigInteger n1, BigInteger n2) {
    if (n1.signum() == 0 && n2.signum() == 0)
        return BigInteger.ZERO;
    n1 = n1.abs(); n2 = n2.abs();
    BigInteger g = n1.gcd(n2);
    return n1.divide(g).multiply(n2);
}

// Normalisierung:

private void normal () {
    NumberQ q = normal(num,denom);
    num = q.num; denom = q.denom;
}

} // Ende der Klasse NumberQ

```



## 2.2 Addition von rationalen Zahlen

Es werden drei leicht unterschiedliche Versionen angegeben, zunächst ein einfacher Algorithmus, dann zwei optimierte Algorithmen, bei denen kleine Zahlen angestrebt werden.

In Version 1 verwendet man das Produkt der gegebenen Nenner als gemeinsamen Nenner.

### Version 1, Pseudocode-Funktion addQ:

```
function addQ(x, y)
  // Vorausgesetzt: Rationale Zahlen x und y
  d ← x.denom · y.denom
  n ← x.num · y.denom + y.num · x.denom
  return numberQ(n, d)
```

### Version 1, Java-Methode NumberQ.add:

```
// Summe von zwei rationalen Zahlen, Version 1 (nicht optimiert):

public static NumberQ add (NumberQ x, NumberQ y) {
    BigInteger n1 = x.num, d1 = x.denom;
    BigInteger n2 = y.num, d2 = y.denom;
    BigInteger d = d1.multiply(d2);
    BigInteger n = n1.multiply(d2).add(n2.multiply(d1));
    return normal(n,d);
}
```

Version 2 entspricht der aus dem Schulunterricht vertrauten Vorgehensweise.

### Version 2, Pseudocode-Funktion addQ2:

```
function addQ2(x, y)
  d ← lcm(x.denom, y.denom) // kleinstes gemeinsames Vielfaches
  f1 ← d/x.denom
  f2 ← d/y.denom
  n ← f1 · x.num + f2 · y.num
  return numberQ(n, d)
```

### Version 2, Java-Methode NumberQ.add2:

```
// Summe von zwei rationalen Zahlen, Version 2 (Schulmethode):

public static NumberQ add2 (NumberQ x, NumberQ y) {
    BigInteger n1 = x.num, d1 = x.denom;
    BigInteger n2 = y.num, d2 = y.denom;
    BigInteger d = lcm(d1,d2);
    BigInteger f1 = d.divide(d1);
    BigInteger f2 = d.divide(d2);
    BigInteger n = f1.multiply(n1).add(f2.multiply(n2));
    return normal(n,d);
}
```

Version 3 nach Michael Kaplan, Computeralgebra (vorteilhaft für sehr große Zahlen):

### Version 3, Pseudocode-Funktion addQ3:

```
function addQ3(x, y)
     $g_1 \leftarrow \text{gcd}(x.\text{denom}, y.\text{denom})$  // größter gemeinsamer Teiler
     $f_1 \leftarrow x.\text{denom}/g_1$ 
     $f_2 \leftarrow y.\text{denom}/g_1$ 
     $n \leftarrow f_2 \cdot x.\text{num} + f_1 \cdot y.\text{num}$ 
     $d \leftarrow f_1 \cdot y.\text{denom}$ 
     $g_2 \leftarrow \text{gcd}(n, g_1)$  // größter gemeinsamer Teiler
     $n \leftarrow n/g_2$ 
     $d \leftarrow d/g_2$ 
    return numberQ(n, d)
```

### Version 3, Java-Methode NumberQ.add3:

```
// Summe von zwei rationalen Zahlen, Version 3
// Nach Michael Kaplan, Computeralgebra

public static NumberQ add3 (NumberQ x, NumberQ y) {
    BigInteger n1 = x.num, d1 = x.denom;
    BigInteger n2 = y.num, d2 = y.denom;
    BigInteger g1 = d1.gcd(d2);
    BigInteger f1 = d1.divide(g1);
    BigInteger f2 = d2.divide(g1);
    BigInteger n = f2.multiply(n1).add(f1.multiply(n2));
    BigInteger d = f1.multiply(d2);
    BigInteger g2 = n.gcd(d);
    n = n.divide(g2); d = d.divide(g2);
    return normal(n,d);
}
```

### 2.3 Subtraktion von rationalen Zahlen

Die Subtraktion wird entsprechend implementiert wie die Addition. Das Pluszeichen ist jeweils durch ein Minuszeichen zu ersetzen. Im Java-Quelltext ist statt der Methode `add` von `BigInteger` die Methode `subtract` zu verwenden.

## 2.4 Multiplikation von rationalen Zahlen

Es werden zwei leicht unterschiedliche Versionen angegeben, zunächst ein einfacher Algorithmus, dann ein optimierter Algorithmus, bei dem kleine Zahlen angestrebt werden.

In Version 1 wird sofort multipliziert (Zähler mal Zähler, Nenner mal Nenner) und erst nachträglich gekürzt.

### Version 1, Pseudocode-Funktion mulQ:

```
function mulQ(x, y)  
  // Vorausgesetzt: Rationale Zahlen x und y  
  n ← x.num · y.num  
  d ← x.denom · y.denom  
  return numberQ(n, d)
```

### Version 1, Java-Methode NumberQ.mul:

```
// Produkt von zwei rationalen Zahlen, Version 1 (nicht optimiert):
```

```
public static NumberQ mul (NumberQ x, NumberQ y) {  
    BigInteger n1 = x.num, d1 = x.denom;  
    BigInteger n2 = y.num, d2 = y.denom;  
    BigInteger n = n1.multiply(n2);  
    BigInteger d = d1.multiply(d2);  
    return normal(n,d);  
}
```

In Version 2 werden gemeinsame Faktoren schon vor der eigentlichen Multiplikation herausgekürzt.

### Version 2, Pseudocode-Funktion mulQ2:

```
function mulQ2(x, y)
   $f_1 \leftarrow \text{gcd}(x.\text{num}, y.\text{denom})$ 
   $f_2 \leftarrow \text{gcd}(x.\text{denom}, y.\text{num})$ 
   $n_1 \leftarrow x.\text{num} / f_1$ 
   $d_1 \leftarrow x.\text{denom} / f_2$ 
   $n_2 \leftarrow y.\text{num} / f_2$ 
   $d_2 \leftarrow y.\text{denom} / f_1$ 
   $n \leftarrow n_1 \cdot n_2$ 
   $d \leftarrow d_1 \cdot d_2$ 
  return numberQ(n, d)
```

### Version 2, Java-Methode NumberQ.mul2:

// Produkt von zwei rationalen Zahlen, Version 2 (mit vorherigem Kürzen):

```
public static NumberQ mul2 (NumberQ x, NumberQ y) {
    BigInteger n1 = x.num, d1 = x.denom;
    BigInteger n2 = y.num, d2 = y.denom;
    BigInteger f1 = n1.gcd(d2);
    BigInteger f2 = d1.gcd(n2);
    n1 = n1.divide(f1);
    d1 = d1.divide(f2);
    n2 = n2.divide(f2);
    d2 = d2.divide(f1);
    BigInteger n = n1.multiply(n2);
    BigInteger d = d1.multiply(d2);
    return normal(n,d);
}
```

## 2.5 Division von rationalen Zahlen

Version 1 der Division ist nicht optimiert. Zu beachten ist, dass der Divisor nicht gleich 0 sein darf.

### Version1, Pseudocode-Funktion divQ:

```
function divQ(x, y)
  // Vorausgesetzt: Rationale Zahlen x und y
  if y = 0 then return undefined
  n ← x.num · y.denom
  d ← x.denom · y.num
  return numberQ(n, d)
```

### Version 1, Java-Methode NumberQ.div:

```
// Quotient von zwei rationalen Zahlen, Version 1 (nicht optimiert):

public static NumberQ div (NumberQ x, NumberQ y) throws Exception {
    if (y.isZero()) throw new Exception("Division durch 0 verboten!");
    BigInteger n1 = x.num, d1 = x.denom;
    BigInteger n2 = y.num, d2 = y.denom;
    BigInteger n = n1.multiply(d2);
    BigInteger d = d1.multiply(n2);
    return normal(n,d);
}
```

Wie bei der Multiplikation ist es sinnvoll, gemeinsame Faktoren vor der eigentlichen Division herauszukürzen.

#### Version2, Pseudocode-Funktion divQ2:

```
function divQ2(x, y)
  if y = 0 then return undefined
   $f_1 \leftarrow \text{gcd}(x.\text{num}, y.\text{num})$ 
   $f_2 \leftarrow \text{gcd}(x.\text{denom}, y.\text{denom})$ 
   $n_1 \leftarrow x.\text{num}/f_1$ 
   $d_1 \leftarrow x.\text{denom}/f_2$ 
   $n_2 \leftarrow y.\text{num}/f_1$ 
   $d_2 \leftarrow y.\text{denom}/f_2$ 
   $n \leftarrow n_1 \cdot d_2$ 
   $d \leftarrow d_1 \cdot n_2$ 
```

#### Version 2, Java-Methode NumberQ.div2:

// Quotient von zwei rationalen Zahlen, Version 2 (vorheriges Kürzen):

```
public static NumberQ div2 (NumberQ x, NumberQ y) throws Exception {
    if (y.isZero()) throw new Exception("Division durch 0 verboten!");
    BigInteger n1 = x.num, d1 = x.denom;
    BigInteger n2 = y.num, d2 = y.denom;
    BigInteger f1 = n1.gcd(n2);
    BigInteger f2 = d1.gcd(d2);
    n1 = n1.divide(f1);
    d1 = d1.divide(f2);
    n2 = n2.divide(f1);
    d2 = d2.divide(f2);
    BigInteger n = n1.multiply(d2);
    BigInteger d = d1.multiply(n2);
    return normal(n,d);
}
```

## 2.6 Umwandlung in einen Dezimalbruch

Pseudocode-Funktion toDecimalQ:

```
function toDecimalQ(x)
  // Vorausgesetzt: Nicht-negative rationale Zahl x (vollständig gekürzt)
  // mit Zähler num und Nenner denom
  n ← x.num
  d ← x.denom
  s1 ← leere Zeichenkette // Vorkommastellen
  s2 ← leere Zeichenkette // Normale Nachkommastellen
  s3 ← leere Zeichenkette // Periode
  s1 ← append(s, div(n, d))
  n ← mod(n, d)
  if n = 0 then return [s1, s2, s3]
  e2 ← 0 // Exponent für Primfaktor 2
  e5 ← 0 // Exponent für Primfaktor 5
  h ← d
  while mod(h, 2) = 0 do
    [e2 ← e2 + 1
     h ← div(h, 2)]
  while mod(h, 5) = 0 do
    [e5 ← e5 + 1
     h ← div(h, 5)]
  e ← max(e2, e5)
  for i ← 1 to e do
    [n ← n · 10
     s2 ← append(s2, div(n, d))
     n ← mod(n, d)]
  if n = 0 then return [s1, s2, s3]
  lp ← 0 // Periodenlänge
  n0 ← n
  repeat
    [lp ← lp + 1
     n ← n · 10
     s3 ← append(s3, div(n, d))
     n ← div(n, d)]
  until n = n0
  return [s1, s1, s2]
```

**Erläuterung:** Der Rückgabewert der Funktion ist ein Array, das aus drei Zeichenketten besteht: Die erste Zeichenkette entspricht den Vorkommastellen, die zweite den „normalen“ Nachkommastellen und die dritte der Periode. Die zweite und die dritte Zeichenkette können auch leer sein.



## Java-Methode NumberQ.stringsDec

Die folgende Methode ist etwas allgemeiner als die vorhergehende Pseudocode-Funktion, weil die gegebene Zahl auch negativ sein kann.

```
// Array von Zeichenketten für Schreibweise als Dezimalbruch:
// Ein normalisierter Bruch wird vorausgesetzt.

public String[] stringsDec () {
    String[] a = new String[4];
    a[2] = a[3] = "";
    a[0] = (num.signum() < 0 ? "-" : "");           // Vorzeichen
    BigInteger n = num.abs(), d = denom;
    a[1] = "" + n.divide(d);
    n = n.mod(d);
    if (n.signum() == 0) return a;                 // Ganze Zahl
    int e2 = 0, e5 = 0;
    BigInteger h = d;
    BigInteger two = BigInteger.valueOf(2);
    BigInteger five = BigInteger.valueOf(5);
    while (h.mod(two).signum() == 0) {
        e2++; h = h.divide(two);
    }
    while (h.mod(five).signum() == 0) {
        e5++; h = h.divide(five);
    }
    int e = Math.max(e2, e5);
    for (int i=0; i<e; i++) {
        n = n.multiply(BigInteger.TEN);
        a[2] += n.divide(d);
        n = n.mod(d);
    }
    if (n.signum() == 0) return a;                 // Endlicher Dezimalbruch
}
```

```
int lp = 0;
BigInteger n0 = n;
do {
    lp++;
    if (lp > 1000) {a[3] += "..."; break;}
    n = n.multiply(BigInteger.TEN);
    a[3] += n.divide(d);
    n = n.mod(d);
}
while (n.compareTo(n0) != 0);
return a;                                     // Unendlicher Dezimalbruch
}
```

## 3 Lineare Algebra

### 3.1 Vektoren, Grundlagen

Zum Rechnen mit Vektoren und Matrizen werden zwei weitere Java-Klassen `VectorQ` (Vektor mit rationalen Koordinaten) und `MatrixQ` (Matrix mit rationalen Elementen) eingeführt, die auf der Klasse `NumberQ` aufbauen.

Ein Vektor eines  $n$ -dimensionalen Vektorraums ( $n > 0$ ) wird beschrieben durch ein Array der Länge  $n$ . Die Arrayelemente gehören der Klasse `NumberQ` an.

#### Java-Klasse `VectorQ`:

```
public class VectorQ {  
  
    private NumberQ[] coord;  
  
    // Konstruktor:  
    // c ... Array der Koeffizienten  
  
    public VectorQ (NumberQ[] c) {  
        coord = c;  
    }  
  
}
```

#### Java-Methoden `VectorQ.dimensionV` und `VectorQ.dimensionVV`:

Diese Hilfsmethoden dienen dazu, Ausnahmen auszulösen, wenn es Probleme mit der Dimension gibt.

```
// Dimensionsprüfung für einen Vektor:  
// c ... Array der Vektorkoordinaten  
// Rückgabewert: Dimension  
  
public static int dimensionV (NumberQ[] c) throws Exception {  
    int n = c.length;  
    if (n == 0) throw new Exception("Dimension 0 nicht zulässig!");  
    return n;  
}
```

```
// Dimensionsprüfung für zwei Vektoren:  
  
public static int dimensionVV (NumberQ[] c1, NumberQ[] c2)  
throws Exception {  
    int n1 = dimensionV(c1), n2 = dimensionV(c2);  
    if (n1 != n2) throw new Exception("Dimensionsfehler!");  
    return n1;  
}
```

## 3.2 Addition von Vektoren

Java-Methode VectorQ.add:

```
// Summe von zwei Vektoren:  
// c1, c2 ... Arrays der Vektorkoordinaten  
// Rückgabewert: Array der Vektorkoordinaten für die Summe  
  
public static NumberQ[] add (NumberQ[] c1, NumberQ[] c2)  
throws Exception {  
    int n = dimensionVV(c1,c2);                // Ausnahme möglich  
    NumberQ[] su = new NumberQ[n];  
    for (int i=0; i<n; i++) su[i] = c1[i].add(c2[i]);  
    return su;  
}  
  
// Summe des gegebenen Vektors und eines weiteren Vektors:  
// v ... 2. Summand  
  
public VectorQ add (VectorQ v) throws Exception {  
    return new VectorQ(add(coord,v.coord));  
}
```

### 3.3 Subtraktion von Vektoren

Java-Methode VectorQ.sub:

```
// Differenz von zwei Vektoren:  
// c1, c2 ... Arrays der Vektorkoordinaten  
// Rückgabewert: Array der Vektorkoordinaten für die Differenz  
  
public static NumberQ[] sub (NumberQ[] c1, NumberQ[] c2)  
throws Exception {  
    int n = dimensionVV(c1,c2);                // Ausnahme möglich  
    NumberQ[] di = new NumberQ[n];  
    for (int i=0; i<n; i++) di[i] = c1[i].sub(c2[i]);  
    return di;  
}  
  
// Differenz des gegebenen Vektors und eines weiteren Vektors:  
// v ... Subtrahend  
  
public VectorQ sub (VectorQ v) throws Exception {  
    return new VectorQ(sub(coord,v.coord));  
}
```

### 3.4 Skalare Multiplikation von Vektoren

Java-Methode VectorQ.mul:

```
// Skalare Multiplikation (Skalar mal Vektor):
// f ... Skalar
// c ... Array der Koordinaten des gegebenen Vektors
// Rückgabewert: Array der Vektorkoordinaten für das Produkt

public static NumberQ[] mul (NumberQ f, NumberQ[] c) throws Exception {
    int n = dimensionV(c);                // Ausnahme möglich
    NumberQ[] pr = new NumberQ[n];
    for (int i=0; i<n; i++) pr[i] = f.mul(c[i]);
    return pr;
}

// Produkt eines Skalars und des gegebenen Vektors:
// f ... Skalar als Faktor

public VectorQ mul (NumberQ f) throws Exception {
    return new VectorQ(mul(f,coord));
}
```

### 3.5 Skalarprodukt von Vektoren

Java-Methode `VectorQ.innerProduct`:

```
// Skalarprodukt zweier Vektoren:  
// c1, c2 ... Arrays der Vektorkoordinaten  
// Rückgabewert: Skalarprodukt  
  
public static NumberQ innerProduct (NumberQ[] c1, NumberQ[] c2)  
throws Exception {  
    int n = dimensionVV(c1,c2);           // Ausnahme möglich  
    NumberQ su = new NumberQ();  
    for (int i=0; i<n; i++) su = su.add(c1[i].mul(c2[i]));  
    return su;  
}  
  
// Skalarprodukt des gegebenen Vektors und eines weiteren Vektors:  
// v ... 2. Faktor  
  
public NumberQ innerProduct (VectorQ v) throws Exception {  
    return innerProduct(coord,v.coord);  
}
```



### 3.6 Kreuzprodukt von Vektoren

Java-Methode `VectorQ.crossProduct`:

```
// Hilfsroutine:

private static NumberQ diff (NumberQ[] c1, NumberQ[] c2, int i) {
    int i1 = (i+1)%3, i2 = (i+2)%3;
    NumberQ pr1 = c1[i1].mul(c2[i2]);
    NumberQ pr2 = c1[i2].mul(c2[i1]);
    return pr1.sub(pr2);
}

// Kreuzprodukt zweier Vektoren:
// c1, c2 ... Arrays der Vektorkoordinaten
// Rückgabewert: Array der Vektorkoordinaten für das Produkt

public static NumberQ[] crossProduct (NumberQ[] c1, NumberQ[] c2)
throws Exception {
    int n = dimensionVV(c1,c2); // Ausnahme möglich
    if (n != 3) throw new Exception("Nur Dimension 3 zulässig!");
    NumberQ[] pr = new NumberQ[3];
    for (int i=0; i<3; i++) pr[i] = diff(c1,c2,i);
    return pr;
}

// Kreuzprodukt des gegebenen Vektors und eines weiteren Vektors:
// v ... 2. Faktor

public VectorQ crossProduct (VectorQ v) throws Exception {
    return new VectorQ(crossProduct(coord,v.coord));
}
```

### 3.7 Matrizen, Grundlagen

Die Java-Klasse `MatrixQ` ist für Matrizen mit rationalen Elementen zuständig. Sie verwendet ein zweifach indiziertes Array von `NumberQ`-Objekten.

Die beiden folgenden Methoden dienen der Dimensionsprüfung und können Ausnahmen auslösen.

#### Java-Methoden `MatrixQ.dimensionsM` und `MatrixQ.dimensionsMM`:

```
// Dimensionsprüfung für eine Matrix:
// c ... Array der Matrixelemente (zweifach indiziert)
// Rückgabewert: Array, bestehend aus Zeilenzahl und Spaltenzahl

public static int[] dimensionsM (NumberQ[][] c) throws Exception {
    int m = c.length;
    if (m == 0) throw new Exception("Matrix mit 0 Zeilen!");
    int n = c[0].length;
    if (n == 0) throw new Exception("Matrix mit 0 Spalten!");
    for (int i=1; i<m; i++) {
        if (c[i].length != n)
            throw new Exception("Zeilenlänge unterschiedlich!");
    }
    int[] d = new int[2];
    d[0] = m; d[1] = n;
    return d;
}

// Dimensionsprüfung für zwei Matrizen:
// c1, c2 ... Arrays der Matrixelemente (zweifach indiziert)
// Rückgabewert: Array, bestehend aus Zeilenzahl und Spaltenzahl

public static int[] dimensionsMM (NumberQ[][] c1, NumberQ[][] c2)
throws Exception {
    int[] dim1 = dimensionsM(c1);
    int[] dim2 = dimensionsM(c2);
    if (dim1[0] != dim2[0] || dim1[1] != dim2[1])
        throw new Exception("Dimensionsfehler!");
    return dim1;
}
```

### Java-Methode MatrixQ.dimensionQuadratic:

```
// Dimensionsprüfung für quadratische Matrix:
// c ... Array der Matrixelemente (zweifach indiziert)
// Rückgabewert: Zeilen- bzw. Spaltenzahl

public static int dimensionQuadratic (NumberQ[] [] c) throws Exception {
    int[] dim = dimensionsM(c);           // Ausnahme möglich
    if (dim[0] != dim[1]) throw new Exception("Matrix nicht quadratisch!");
    return dim[0];
}
```

Häufig benötigt wird die Vertauschung von Matrixzeilen oder -spalten.

### Java-Methoden MatrixQ.swapLines und MatrixQ.swapColumns:

```
// Vertauschung zweier Matrixzeilen:
// a ... Array der Matrixelemente (zweifach indiziert)
// i1, i2 ... Zeilenindizes

private static void swapLines (NumberQ[] [] a, int i1, int i2) {
    NumberQ[] h = a[i1];
    a[i1] = a[i2];
    a[i2] = h;
}

// Vertauschung zweier Matrixspalten:
// a ... Array der Matrixelemente (zweifach indiziert)
// i1, i2 ... Spaltenindizes

private static void swapColumns (NumberQ[] [] a, int j1, int j2) {
    for (int i=0; i<a.length; i++) {
        NumberQ h = a[i][j1];
        a[i][j1] = a[i][j2];
        a[i][j2] = h;
    }
}
```

### 3.8 Rang einer Matrix

Pseudocode-Funktion rankMQ (rekursiv):

```
function rankMQ(c)
  // Vorausgesetzt: Matrix c (zweifach indiziertes Array, Indizes ab 1)
  m ← Zeilenzahl von c
  n ← Spaltenzahl von c
  a ← Kopie von c
  i0 ← -1
  j0 ← -1
  i ← 1
  while i0 < 0 and i ≤ m do
    [j ← 1
     while j0 < 0 and j ≤ n do
       [if a[i][j] ≠ 0 then
         [i0 ← i
          [j0 ← j
           [j ← j + 1
            [i ← i + 1
             if i0 < 0 then return 0
             if i0 ≠ 1 then Zeilen 1 und i0 von a vertauschen
             if j0 ≠ 1 then Spalten 1 und j0 von a vertauschen
             b ← Neues Array mit m - 1 Zeilen und n - 1 Spalten
             for i ← 2 to m do
               [f ← divQ(a[i][1], a[1][1])
                for j ← 2 to n do
                  [p ← mulQ(f, a[1][j])
                   [b[i - 1][j - 1] ← subQ(a[i][j], p)
             return 1 + rankMQ(b) // Rekursion
```

## Java-Methode MatrixQ.rank:

```
// Rang einer Matrix (rekursive Methode):
// c ... Array der Matrixelemente (zweifach indiziert)

public static int rank (NumberQ[] [] c) throws Exception {
    int[] dim = dimensionsM(c);                // Ausnahme möglich
    int m = dim[0], n = dim[1];
    NumberQ[] [] a = arrayCopy(c);
    int i0 = -1, j0 = -1, i = 0;
    while (i0 < 0 && i < m) {
        int j = 0;
        while (j0 < 0 && j < n) {
            if (!a[i][j].isZero()) {i0 = i; j0 = j;}
            j++;
        }
        i++;
    }
    if (i0 < 0) return 0;
    if (m == 1 || n == 1) return 1;
    if (i0 != 0) swapLines(a,0,i0);
    if (j0 != 0) swapColumns(a,0,j0);
    NumberQ[] [] b = new NumberQ[m-1][n-1];
    for (i=1; i<m; i++) {
        NumberQ f = a[i][0].div(a[0][0]);
        for (int j=1; j<n; j++)
            b[i-1][j-1] = a[i][j].sub(f.mul(a[0][j]));
    }
    return 1+rank(b);
}
```

### 3.9 Addition von Matrizen

Java-Methode MatrixQ.add:

```
// Summe zweier Matrizen:  
// c1, c2 ... Arrays der Matrixelemente  
// Rückgabewert: Array der Matrixelemente für die Summe  
  
public static NumberQ[][] add (NumberQ[][] c1, NumberQ[][] c2)  
throws Exception {  
    int[] dim = dimensionsMM(c1,c2);           // Ausnahme möglich  
    int m = dim[0], n = dim[1];  
    NumberQ[][] su = new NumberQ[m][n];  
    for (int i=0; i<m; i++)  
        for (int j=0; j<n; j++)  
            su[i][j] = c1[i][j].add(c2[i][j]);  
    return su;  
}  
  
// Summe der gegebenen Matrix und einer weiteren Matrix:  
// m ... 2. Summand  
  
public MatrixQ add (MatrixQ m) throws Exception {  
    return new MatrixQ(add(coord,m.coord));  
}
```

### 3.10 Subtraktion von Matrizen

Java-Methode MatrixQ.sub:

```
// Differenz zweier Matrizen:
// c1, c2 ... Arrays der Matrixelemente
// Rückgabewert: Array der Matrixelemente für die Differenz

public static NumberQ[][] sub (NumberQ[][] c1, NumberQ[][] c2)
throws Exception {
    int[] dim = dimensionsMM(c1,c2);           // Ausnahme möglich
    int m = dim[0], n = dim[1];
    NumberQ[][] di = new NumberQ[m][n];
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
            di[i][j] = c1[i][j].sub(c2[i][j]);
    return di;
}

// Differenz der gegebenen Matrix und einer weiteren Matrix:
// m ... Subtrahend

public MatrixQ sub (MatrixQ m) throws Exception {
    return new MatrixQ(sub(coord,m.coord));
}
```

### 3.11 Multiplikation einer Matrix mit einem Vektor

Java-Methode MatrixQ.mul:

```
// Produkt einer Matrix und eines Vektors:
// c1 ... Array der Matrixelemente
// c2 ... Array der Vektorkoordinaten
// Rückgabewert: Array der Vektorkoordinaten für das Produkt

public static NumberQ[] mul (NumberQ[][] c1, NumberQ[] c2)
throws Exception {
    int n = dimensionProdMV(c1,c2);           // Ausnahme möglich
    NumberQ[] pr = new NumberQ[n];
    for (int i=0; i<n; i++)
        pr[i] = VectorQ.innerProduct(c1[i],c2);
    return pr;
}

// Produkt der gegebenen Matrix und eines Vektors:
// v ... Vektor als 2. Faktor
// Rückgabewert: Array der Vektorkoordinaten für das Produkt

public VectorQ mul (VectorQ v) throws Exception {
    return new VectorQ(mul(coord,v.getCoord()));
}
```



### 3.12 Multiplikation von Matrizen

#### Java-Methode MatrixQ.dimensionsProdMM:

```
// Dimensionsprüfung für das Produkt zweier Matrizen:
// c1, c2 ... Array der Matrixelemente
// Rückgabewert: Array bestehend aus Zeilenzahl und Spaltenzahl
// der Produktmatrix

public static int[] dimensionsProdMM (NumberQ[][] c1, NumberQ[][] c2)
throws Exception {
    int[] dim1 = dimensionsM(c1);           // Ausnahme möglich
    int[] dim2 = dimensionsM(c2);           // Ausnahme möglich
    if (dim1[1] != dim2[0]) throw new Exception("Dimensionsfehler!");
    int[] d = new int[2];
    d[0] = dim1[0]; d[1] = dim2[1];
    return d;
}
```

#### Java-Methode MatrixQ.mul:

```
// Produkt zweier Matrizen:
// c1, c2 ... Arrays der Matrixelemente
// Rückgabewert: Array der Matrixelemente für das Produkt

public static NumberQ[][] mul (NumberQ[][] c1, NumberQ[][] c2)
throws Exception {
    int[] dim = dimensionsProdMM(c1,c2);           // Ausnahme möglich
    int m = dim[0], n = dim[1];
    NumberQ[][] pr = new NumberQ[m][n];
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++) {
            NumberQ su = new NumberQ();
            for (int k=0; k<c2.length; k++)
                su = su.add(c1[i][k].mul(c2[k][j]));
            pr[i][j] = su;
        }
    return pr;
}
```

### 3.13 Gauß-Jordan-Algorithmus

Ein lineares Gleichungssystem, gegeben durch eine invertierbare (daher quadratische) Matrix  $m$  (zweifach indiziertes Array, Indizes ab 1) und einen Vektor  $v$  (einfach indiziertes Array, Index ab 1), soll gelöst werden. Die Lösung wird in Form eines einfach indizierten Arrays ausgegeben.

#### Pseudocode-Funktion `gaussJordanQ`:

```
function gaussJordanQ( $m, v$ )
  // Vorausgesetzt: Invertierbare Matrix  $m$ , Vektor  $v$ 
   $n \leftarrow$  Zeilenzahl von  $m$ 
  if  $n \neq$  (Spaltenzahl von  $m$ ) then return undefined
  if  $n \neq$  (Länge von  $v$ ) then return undefined
   $a \leftarrow$  Neue Matrix mit  $n$  Zeilen und  $n + 1$  Spalten
  for  $i \leftarrow 1$  to  $n$  do
    [ for  $j \leftarrow 1$  to  $n$  do  $a[i][j] \leftarrow m[i][j]$ 
       $a[i][n + 1] \leftarrow v[i]$ 
    ]
  for  $j \leftarrow 1$  to  $n$  do
    [  $p \leftarrow j$ 
      while  $p \leq n$  and  $a[p][j] = 0$  do  $p \leftarrow p + 1$ 
      if  $p = n + 1$  then return undefined
      if  $p \neq j$  then Zeilen  $j$  und  $p$  von Matrix  $a$  vertauschen
       $f \leftarrow a[j][j]$ 
      Zeile  $j$  von Matrix  $a$  durch  $f$  dividieren
      for  $i \leftarrow 1$  to  $n$  do
        [ if  $i \neq j$  then
          [  $f \leftarrow a[i][j]$ 
            In Matrix  $a$  von Zeile  $i$  das  $f$ -fache von Zeile  $j$  subtrahieren
          ]
        ]
    ]
   $x \leftarrow$  Neues Array der Länge  $n$ 
  for  $i \leftarrow 1$  to  $n$  do  $x[i] \leftarrow a[i][n + 1]$ 
  return  $x$ 
```

Quelle: [de.wikibooks.org/wiki/Algorithmensammlung:\\_Numerik:\\_Gauß-Jordan-Algorithmus](https://de.wikibooks.org/wiki/Algorithmensammlung:_Numerik:_Gau%C3%9F-Jordan-Algorithmus)

## Java-Methode MatrixQ.gaussJordan:

```
// Lösung eines linearen Gleichungssystems (Gauß-Jordan-Algorithmus):
// c1 ... Matrix der Koeffizienten (quadratisch, invertierbar)
// c2 ... Vektor für den inhomogenen Teil des Gleichungssystems

public static NumberQ[] gaussJordan (NumberQ[] [] c1, NumberQ[] c2)
throws Exception {
    int n = dimensionQuadratic(c1);           // Ausnahme möglich
    if (c2.length != n) throw new Exception("Dimensionsfehler!");
    NumberQ[] [] a = new NumberQ[n][n+1];
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) a[i][j] = c1[i][j];
        a[i][n] = c2[i];
    }
    for (int j=0; j<n; j++) {
        int p = j;
        while (p < n && a[p][j].isZero()) p++;
        if (p == n) throw new Exception("Matrix nicht invertierbar!");
        if (p != j) swapLines(a,j,p);
        NumberQ f = a[j][j];
        for (int k=0; k<=n; k++) a[j][k] = a[j][k].div(f);
        for (int i=0; i<n; i++) {
            if (i == j) continue;
            f = a[i][j];
            for (int k=0; k<=n; k++)
                a[i][k] = a[i][k].sub(f.mul(a[j][k]));
        } // Ende for (i)
    } // Ende for (j)
    NumberQ[] x = new NumberQ[n];
    for (int i=0; i<n; i++) x[i] = a[i][n];
    return x;
}
```

### 3.14 Inverse Matrix

Zu einer invertierbaren (und daher quadratischen) Matrix  $m$  soll die inverse Matrix ermittelt werden.

**Pseudocode-Funktion inverseMQ:**

```
function inverseMQ( $m$ )
  // Vorausgesetzt: Matrix  $m$ 
   $n \leftarrow$  Zeilenzahl von  $m$ 
  if  $n \neq$  (Spaltenzahl von  $m$ ) then return undefined
   $a \leftarrow$  Kopie von  $m$ 
   $b \leftarrow$  ( $n, n$ )-Einheitsmatrix
  for  $j \leftarrow 1$  to  $n$  do
     $p \leftarrow j$ 
    while  $p \leq n$  and  $a[p][j] = 0$  do  $p \leftarrow p + 1$ 
    if  $p = n + 1$  then return undefined
    if  $p \neq j$  then
      [ Zeilen  $j$  und  $p$  von Matrix  $a$  vertauschen
      [ Zeilen  $j$  und  $p$  von Matrix  $b$  vertauschen
     $f \leftarrow a[j][j]$ 
    Zeile  $j$  von Matrix  $a$  durch  $f$  dividieren
    Zeile  $j$  von Matrix  $b$  durch  $f$  dividieren
    for  $i \leftarrow 1$  to  $n$  do
      if  $i \neq j$  then
        [  $f \leftarrow a[i][j]$ 
        [ In Matrix  $a$  von Zeile  $i$  das  $f$ -fache der Zeile  $j$  subtrahieren
        [ In Matrix  $b$  von Zeile  $i$  das  $f$ -fache der Zeile  $j$  subtrahieren
  return  $b$ 
```

Quelle: [de.wikibooks.org/wiki/Algorithmensammlung:\\_Numerik:\\_Inverse\\_Matrix](http://de.wikibooks.org/wiki/Algorithmensammlung:_Numerik:_Inverse_Matrix)

## Java-Methode MatrixQ.inverse:

```
// Inverse Matrix:
// c ... Array der Matrixelemente
// Rückgabewert: Array der Matrixelemente für die inverse Matrix

public static NumberQ[][] inverse (NumberQ[][] c) throws Exception {
    int n = dimensionQuadratic(c);           // Ausnahme möglich
    NumberQ[][] a = arrayCopy(c);
    NumberQ[][] b = arrayIdentity(n);
    for (int j=0; j<n; j++) {
        int p = j;
        while (p < n && a[p][j].isZero()) p++;
        if (p == n) throw new Exception("Matrix nicht invertierbar!");
        if (p != j) {
            swapLines(a,j,p);
            swapLines(b,j,p);
        }
        NumberQ f = a[j][j];
        for (int k=0; k<n; k++) {
            a[j][k] = a[j][k].div(f);
            b[j][k] = b[j][k].div(f);
        }
        for (int i=0; i<n; i++) {
            if (i == j) continue;
            f = a[i][j];
            for (int k=0; k<n; k++) {
                NumberQ pr = f.mul(a[j][k]);
                a[i][k] = a[i][k].sub(pr);
                pr = f.mul(b[j][k]);
                b[i][k] = b[i][k].sub(pr);
            } // Ende for (k)
        } // Ende for (i)
    } // Ende for (j)
    return b;
}
```

### 3.15 Determinante

Der folgende Algorithmus zur Berechnung einer Determinante verwendet – analog zum Gauß’schen Eliminationsverfahren – zwei Arten von Zeilenumformungen:

- Bei der Vertauschung zweier Matrixzeilen bleibt die Determinante betragsmäßig gleich, das Vorzeichen kehrt sich um.
- Subtrahiert man von einer Matrixzeile ein beliebiges Vielfaches einer anderen Zeile, so wird die Determinante dadurch nicht verändert.

Mithilfe solcher Zeilenumformungen wird die gegebene Matrix in Stufenform gebracht. Es entsteht eine obere Dreiecksmatrix, deren Determinante sich ganz einfach durch Multiplikation aller Elemente der Hauptdiagonale ergibt.

#### Pseudocode-Funktion `determinantMQ`:

```
function determinantMQ(m)
  // Vorausgesetzt: Quadratische Matrix m (zweifach indiziertes Array)
  n ← Länge bzw. Zeilenzahl von m
  for i ← 1 to n do
    if n ≠ (Länge bzw. Spaltenzahl von m[i]) then return undefined
  a ← Kopie von m
  d ← 1
  for j ← 1 to n do
    [p ← j
    while p ≤ n and a[p][j] = 0 do p ← p + 1
    if p > n then return 0
    if p ≠ j then
      [Zeilen j und p von Matrix a vertauschen
      d ← −d
    d ← d · a[j][j]
    for i ← j + 1 to n do
      [f ← a[i][j]/a[j][j]
      for k ← 1 to n do
        [a[i][k] ← a[i][k] − a[j][k] · f
    return d
```

Quelle: [de.wikibooks.org/wiki/Algorithmensammlung:\\_Numerik:\\_Determinante](https://de.wikibooks.org/wiki/Algorithmensammlung:_Numerik:_Determinante)

## Java-Methode MatrixQ.determinant:

```
// Determinante:
// c ... Array der Matricelemente

public static NumberQ determinant (NumberQ[] [] c) throws Exception {
    int n = dimensionQuadratic(c);           // Ausnahme möglich
    NumberQ[] [] a = arrayCopy(c);
    NumberQ d = new NumberQ(1);
    for (int j=0; j<n; j++) {
        int p = j;
        while (p < n && a[p][j].signum() == 0) p++;
        if (p == n) return new NumberQ();
        if (p != j) {
            swapLines(a,j,p);
            d = d.negate();
        }
        d = d.mul(a[j][j]);
        for (int i=j+1; i<n; i++) {
            NumberQ f = a[i][j].div(a[j][j]);
            for (int k=0; k<n; k++) {
                NumberQ h = a[j][k].mul(f);
                a[i][k] = a[i][k].sub(h);
            }
        }
    }
    return d;
}
```

### 3.16 Charakteristisches Polynom

Java-Methode `MatrixQ.characteristicPolynomial`:

```
// Charakteristisches Polynom:
// c ... Array der Matrixelemente
// Rückgabewert: Koeffizienten-Array des charakteristischen Polynoms

public static NumberQ[] characteristicPolynomial (NumberQ[] [] c)
throws Exception {
    int n = dimensionQuadratic(c);                // Ausnahme möglich
    NumberQ[] [] b = arrayCopy(c);
    NumberQ[] p = new NumberQ[n+1];
    p[n] = new NumberQ(1);
    NumberQ[] t = new NumberQ[n];
    for (int i=1; i<=n; i++) {
        if (i > 1) b = mul(b,c);
        NumberQ su = new NumberQ();
        for (int j=0; j<n; j++) su = su.add(b[j][j]);
        t[i-1] = su;
        su = new NumberQ();
        for (int k=0; k<i; k++)
            su = su.add(t[k].mul(p[n-i+k+1]));
        p[n-i] = su.div(new NumberQ(i)).negate();
    }
    return p;
}
```

Quelle: [de.wikipedia.org/wiki/Algorithmus\\_von\\_Faddejew-Leverrier](http://de.wikipedia.org/wiki/Algorithmus_von_Faddejew-Leverrier)



## 4 Univariate Polynome

### 4.1 Grundlagen

Univariate Polynome sind Polynome mit nur einer Variablen. Ein univariates Polynom kann durch ein Array von Koeffizienten beschrieben werden. Für ein Polynom des Grades  $n$  benötigt man ein Array der Länge  $n + 1$ . Das Nullpolynom wird hier – entgegen der üblichen Definition – als Polynom 0. Grades aufgefasst.

Das Array der Koeffizienten sollte normalisiert sein, also nicht unnötig groß.

Im Folgenden wird eine Java-Klasse `PolynomialQ` (univariates Polynom mit rationalen Koeffizienten) entwickelt. Zur Vermeidung von Rundungsfehlern wird die schon bekannte Java-Klasse `NumberQ` verwendet. Das Attribut `coeff` steht für das Array der Koeffizienten.

**Java-Klasse `PolynomialQ`:**

```
public class PolynomialQ {  
  
    private NumberQ[] coeff;                // Koeffizienten-Array  
  
    // Konstruktor:  
    // c ... Koeffizienten-Array  
  
    public PolynomialQ (NumberQ[] c) {  
        coeff = c;  
    }  
  
}
```

Die folgende Methode liefert zu einem gegebenen Array des Typs `NumberQ[]` ein möglichst kleines, gleichwertiges Koeffizienten-Array, das mindestens die Länge 0 besitzt.

**Java-Methode `PolynomialQ.normal`:**

```
// Normalisierung:  
// c ... Gegebenes Array  
  
public static NumberQ[] normal (NumberQ[] c) {  
    int i = c.length-1;  
    if (i < 0) return arrayZero(1);  
    while (c[i].isZero() && i > 0) i--;  
    return Arrays.copyOf(c,i+1);  
}
```

Der Leitkoeffizient eines Polynoms ist der Koeffizient, der zum größten Exponenten gehört.

**Java-Methode `PolynomialQ.leadCoeff`:**

```
// Leitkoeffizient:  
// c ... Koeffizienten-Array des gegebenen Polynoms (normalisiert)  
  
public static NumberQ leadCoeff (NumberQ[] c) {  
    return c[c.length-1];  
}
```

## 4.2 Horner-Schema

Für eine Polynomfunktion  $p$  (gegeben durch ein Array  $c$  von Koeffizienten) und ein Argument  $x$  soll der Funktionswert  $p(x)$  berechnet werden.

**Pseudocode-Funktion horner:**

```
function horner( $c, x$ )
  // Vorausgesetzt: Koeffizienten-Array  $c$ , Argument  $x$ 
   $y \leftarrow 0$ 
   $i \leftarrow (\text{Länge von } c) - 1$ 
  while  $i \geq 0$  do
     $y \leftarrow y \cdot x + c[i]$ 
     $i \leftarrow i - 1$ 
  return  $y$ 
```

**Java-Methode PolynomialQ.horner:**

```
// Funktionswert, berechnet mit Horner-Verfahren:
// c... Array der Koeffizienten (normalisiert, Exponent als Index)
// x ... Wert der unabhängigen Variablen
```

```
public static NumberQ horner (NumberQ[] c, NumberQ x) {
    NumberQ y = new NumberQ();
    for (int i=c.length-1; i>=0; i--) y = y.mul(x).add(c[i]);
    return y;
}
```

```
// Funktionswert, berechnet mit Horner-Verfahren:
// x ... Argument
```

```
public NumberQ horner (NumberQ x) {
    return horner(coeff,x);
}
```

### 4.3 Addition von univariaten Polynomen

Java-Methode PolynomialQ.add:

```
// Summe zweier Polynome:
// c1, c2 ... Koeffizienten-Arrays der gegebenen Polynome (normalisiert)
// Rückgabewert: Koeffizienten-Array des Summenpolynoms (normalisiert)

public static NumberQ[] add (NumberQ[] c1, NumberQ[] c2) {
    int n1 = c1.length-1, n2 = c2.length-1;
    int n = Math.max(n1,n2);
    NumberQ[] a = new NumberQ[n+1];
    for (int i=0; i<=n; i++) {
        if (i <= n1 && i <= n2) a[i] = c1[i].add(c2[i]);
        else if (i <= n1) a[i] = c1[i];
        else a[i] = c2[i];
    }
    return normal(a);
}

// Summe des gegebenen Polynoms und eines weiteren Polynoms:
// p ... Weiteres Polynom (2. Summand)

public PolynomialQ add (PolynomialQ p) {
    NumberQ[] c = add(coeff,p.coeff);
    return new PolynomialQ(c);
}
```

## 4.4 Subtraktion von univariaten Polynomen

Java-Methode PolynomialQ.sub:

```
// Differenz zweier Polynome:
// c1, c2 ... Koeffizienten-Arrays der gegebenen Polynome (normalisiert)
// Rückgabewert: Koeffizienten-Array des Differenzpolynoms (normalisiert)

public static NumberQ[] sub (NumberQ[] c1, NumberQ[] c2) {
    int n1 = c1.length-1, n2 = c2.length-1;
    int n = Math.max(n1,n2);
    NumberQ[] a = new NumberQ[n+1];
    for (int i=0; i<=n; i++) {
        if (i <= n1 && i <= n2) a[i] = c1[i].sub(c2[i]);
        else if (i <= n1) a[i] = c1[i];
        else a[i] = c2[i].negate();
    }
    return normal(a);
}

// Differenz des gegebenen Polynoms und eines weiteren Polynoms:
// p ... Weiteres Polynom (Subtrahend)

public PolynomialQ sub (PolynomialQ p) {
    NumberQ[] c = sub(coeff,p.coeff);
    return new PolynomialQ(c);
}
```

## 4.5 Multiplikation von univariaten Polynomen

### Java-Methode PolynomialQ.mulMon:

```
// Produkt eines Polynoms und eines Monoms:  
// c ... Koeffizienten-Array des gegebenen Polynoms (normalisiert)  
// f ... Koeffizient des Monoms  
// e ... Exponent des Monoms  
// Rückgabewert: Koeffizienten-Array des Produktpolynoms  
// (gegebenes Polynom mal f mal Variable hoch e, normalisiert)  
  
public static NumberQ[] mulMon (NumberQ[] c, NumberQ f, int e) {  
    int n = c.length-1;  
    NumberQ[] a = arrayZero(n+1+e);  
    for (int i=0; i<=n; i++) a[i+e] = f.mul(c[i]);  
    return normal(a);  
}
```

### Java-Methode PolynomialQ.mul:

```
// Produkt zweier Polynome:  
// c1, c2 ... Koeffizienten-Arrays der gegebenen Polynome (normalisiert)  
// Rückgabewert: Koeffizienten-Array des Produktpolynoms (normalisiert)  
  
public static NumberQ[] mul (NumberQ[] c1, NumberQ[] c2) {  
    int n1 = c1.length-1, n2 = c2.length-1;  
    int n = n1+n2;  
    NumberQ[] a = new NumberQ[n+1];  
    for (int i=0; i<=n; i++) {  
        NumberQ su = new NumberQ();  
        int kMin = Math.max(i-n2,0), kMax = Math.min(n1,i);  
        for (int k=kMin; k<=kMax; k++)  
            su = su.add(c1[k].mul(c2[i-k]));  
        a[i] = su;  
    }  
    return normal(a);  
}
```

```
// Produkt des gegebenen Polynoms und eines weiteren Polynoms:  
// p ... Weiteres Polynom (2. Faktor)  
  
public PolynomialQ mul (PolynomialQ p) {  
    NumberQ[] c = mul(coeff,p.coeff);  
    return new PolynomialQ(c);  
}
```

## 4.6 Division von univariaten Polynomen (mit Rest)

Pseudocode-Funktion `divmodPQ`:

```
function divmodPQ( $c_1, c_2$ )
  // Vorausgesetzt: Koeffizienten-Arrays  $c_1, c_2$  (normalisiert)
   $n_1 \leftarrow$  (Länge von  $c_1$ ) - 1
   $n_2 \leftarrow$  (Länge von  $c_2$ ) - 1
   $n_q \leftarrow \max(n_1 - n_2, 0)$ 
   $q \leftarrow$  Neues Array der Länge  $n_q + 1$  mit lauter Nullen
   $r \leftarrow$  Kopie von  $c_1$ 
  while (Länge von  $r$ ) - 1  $\geq n_2$  and  $r \neq$  Nullarray do
     $f \leftarrow$  divQ(Leitkoeffizient von  $r$ , Leitkoeffizient von  $c_2$ )
     $q[(\text{Länge von } r) - 1 - n_2] \leftarrow f$ 
     $pr \leftarrow$  mulMonPQ( $c_2, f, (\text{Länge von } r) - (\text{Länge von } c_2)$ )
     $r \leftarrow$  subPQ( $r, pr$ )
  return [ $q, r$ ]
```

Java-Methode `PolynomialQ.divmod`:

```
// Division mit Rest:
// c1, c2 ... Koeffizienten-Arrays der gegebenen Polynome (normalisiert)
// Rückgabewert: Zweifach indiziertes Array, bestehend aus dem
// Koeffizienten-Array des Quotientenpolynoms und dem Koeffizienten-Array
// des Restpolynoms (beide normalisiert)

public static NumberQ[][] divmod (NumberQ[] c1, NumberQ[] c2)
throws Exception {
  if (isZero(c2)) throw new Exception("Division durch 0 verboten!");
  int n1 = c1.length-1, n2 = c2.length-1;
  int nq = Math.max(n1-n2,0);
  NumberQ[] q = arrayZero(nq+1);
  NumberQ[] r = copyArray(c1);
  NumberQ lc2 = leadCoeff(c2);
  while (r.length-1 >= n2 && !isZero(r)) {
    NumberQ f = leadCoeff(r).div(lc2);
    q[r.length-1-n2] = f;
    NumberQ[] pr = mulMon(c2,f,r.length-c2.length);
    r = sub(r,pr);
  }
  NumberQ[][] a = new NumberQ[2][];
  a[0] = q; a[1] = r;
  return a;
}
```



```
// Division des gegebenen Polynoms durch ein weiteres Polynom (mit Rest):  
// p ... Weiteres Polynom (Divisor)  
  
public PolynomialQ[] divmod (PolynomialQ p) throws Exception {  
    NumberQ[] [] dm = divmod(coeff,p.coeff);  
    PolynomialQ[] a = new PolynomialQ[2];  
    a[0] = new PolynomialQ(dm[0]);  
    a[1] = new PolynomialQ(dm[1]);  
    return a;  
}
```

## 5 Multivariate Polynome

Multivariate Polynome enthalten im Allgemeinen mehrere Variable. Hier wird angenommen, dass die 26 Buchstaben des lateinischen Alphabets Bezeichner von Variablen sein können. Die Bestandteile (Summanden) von multivariaten Polynomen bezeichnet man als Monome.  $-\frac{2}{3}a^2bc^7$  ist ein typisches Beispiel für ein Monom. Es liegt nahe, multivariate Polynome durch verkettete Listen von Monomen zu beschreiben.

### 5.1 Monome, Grundlagen

#### Java-Klasse MonomialQ:

Die Klasse `MonomialQ` verwendet zur Beschreibung eines Monoms die Attribute `coeff` (Koeffizient, hier `rational`) und `expo` (Array der Exponenten). Im Array `expo` werden die Exponenten der Variablen  $a$  (Index 0) bis  $z$  (Index 25) gespeichert.

```
public class MonomialQ {

    public static final VAR = "abcdefghijklmnopqrstuvwxyz";
    public static int nVar = VAR.length();

    private NumberQ coeff;                // Koeffizient
    private int[] expo;                   // Array der Exponenten

    // Konstruktor für Nullmonom:

    public MonomialQ () {
        coeff = new NumberQ();
        expo = new int[nVar];
    }

    // Kopier-Konstruktor:
    // m ... Gegebenes Monom

    public MonomialQ (MonomialQ m) {
        coeff = m.coeff;
        expo = new int[nVar];
        for (int i=0; i<nVar; i++) expo[i] = m.expo[i];
    }
}
```

```

// Überprüfung der Gleichartigkeit:
// m1, m2 ... Gegebene Monome

public static boolean equivalent (MonomialQ m1, MonomialQ m2) {
    if (m1.isZero() || m2.isZero()) return true;
    for (int i=0; i<nVar; i++)
        if (m1.expo[i] != m2.expo[i]) return false;
    return true;
}

// Entgegengesetztes Monom:
// m ... Gegebenes Monom

public static MonomialQ negate (MonomialQ m) {
    MonomialQ n = new MonomialQ(m);
    n.coeff = m.coeff.negate();
    return n;
}

public MonomialQ negate () {
    return negate(this);
}

// Überprüfung, ob Nullmonom:

public boolean isZero () {
    return coeff.isZero();
}

}

```

## 5.2 Addition von Monomen

Die Summe zweier Monome ist nur dann ein Monom, wenn die gegebenen Monome gleichartig sind. Ist diese Voraussetzung erfüllt, so erhält man das Ergebnismonom durch Addition der Koeffizienten und Beibehaltung der Exponenten.

### Java-Methode `MonomialQ.add`:

```
// Summe zweier gleichartiger Monome:  
// m1, m2 ... Gegebene Monome (Summanden)  
  
public static MonomialQ add (MonomialQ m1, MonomialQ m2)  
throws Exception {  
    if (!equivalent(m1,m2))  
        throw new Exception("Monome nicht gleichartig!");  
    MonomialQ m = new MonomialQ(m1);  
    m.coeff = m1.coeff.add(m2.coeff);  
    return m;  
}  
  
// Summe des gegebenen Monoms und eines weiteren Monoms:  
// m ... Weiteres Monom (2. Summand)  
  
public MonomialQ add (MonomialQ m) throws Exception {  
    return add(this,m);  
}
```

### 5.3 Multiplikation von Monomen

Bei der Multiplikation zweier Monome erhält man das Ergebnismonom durch Multiplikation der Koeffizienten und Addition entsprechender Exponenten.

**Java-Methode MonomialQ.mul:**

```
// Produkt zweier Monome:  
// m1, m2 ... Gegebene Monome (Faktoren)  
  
public static MonomialQ mul (MonomialQ m1, MonomialQ m2) {  
    MonomialQ m = new MonomialQ();  
    m.coeff = m1.coeff.mul(m2.coeff);  
    for (int i=0; i<nVar; i++) m.expo[i] = m1.expo[i]+m2.expo[i];  
    return m;  
}  
  
// Produkt des gegebenen Monoms und eines weiteren Monoms:  
// m ... Weiteres Monom (2. Faktor)  
  
public MonomialQ mul (MonomialQ m) {  
    return mul(this,m);  
}
```

## 5.4 Division von Monomen

Um zwei Monome zu dividieren, muss man ihre Koeffizienten dividieren und entsprechende Exponenten subtrahieren. Wenn sich bei einer der Subtraktionen ein negativer Exponent ergibt, ist das Ergebnis kein Monom mehr. Außerdem darf natürlich der Divisor nicht gleich dem Nullmonom sein.

### Java-Methode MonomialQ.div:

```
// Quotient zweier Monome:
// m1, m2 ... Gegebene Monome (Dividend und Divisor)

public static MonomialQ div (MonomialQ m1, MonomialQ m2)
throws Exception {
    MonomialQ m = new MonomialQ(m1);
    m.coeff = m1.coeff.div(m2.coeff);           // Ausnahme möglich
    for (int i=0; i<nVar; i++) {
        int e = m1.expo[i]-m2.expo[i];
        if (e < 0) throw new Exception("Division der Monome nicht möglich!");
        m.expo[i] = e;
    }
    return m;
}

// Quotient des gegebenen Monoms und eines weiteren Monoms
// m ... Weiteres Monom (Divisor)

public MonomialQ div (MonomialQ m) throws Exception {
    return div(this,m);
}
```

## 5.5 Multivariate Polynome, Grundlagen

Wie schon erwähnt, lässt sich ein multivariates Polynom durch eine verkettete Liste von Monomen ausdrücken. Das Nullpolynom entspricht einer leeren Liste. Die im Folgenden skizzierte Java-Klasse `MultivariateQ` verwendet die Klasse `java.util.Vector`. Wichtig ist die Methode `reduce`, mit der eine unnötig umfangreiche Liste verhindert wird. Diese Methode fasst gleichartige Monome zusammen und entfernt eventuell vorhandene Nullmonome. Eine Sortierung der Monome, beispielsweise gemäß der lexikographischen Ordnung, findet dabei nicht statt.

### Java-Klasse `MultivariateQ`:

```
public class MultivariateQ {

    private Vector<MonomialQ> list;           // Liste der Monome

    // Konstruktor für das Nullpolynom:

    public MultivariateQ () {
        list = new Vector<MonomialQ>();
    }

    // Konstruktor für ein Polynom mit nur einem Monom:

    public MultivariateQ (MonomialQ m) {
        this();
        if (!m.isZero()) list.add(m);
    }

    // Kopier-Konstruktor:
    // p ... Gegebenes multivariates Polynom

    public MultivariateQ (MultivariateQ p) {
        this();
        for (int i=0; i<p.getSize(); i++)
            list.add(new MonomialQ(p.getSummand(i)));
    }
}
```

```

// Zahl der Summanden:

public int getSize () {return list.size();}

// Einzelner Summand:
// i ... Index

public MonomialQ getSummand (int i) {return list.get(i);}

// Überprüfung, ob Nullpolynom:

public boolean isZero () {return (list.size() == 0);}

// Reduzierung der Monom-Liste:

public void reduce () {
    for (int i=0; i<getSize(); i++) {
        MonomialQ mi = getSummand(i);
        for (int j=i+1; j<getSize(); j++) {
            MonomialQ mj = getSummand(j);
            if (!mi.isEquivalent(mj)) continue;
            list.removeElementAt(j);
            MonomialQ su = new MonomialQ();
            try {su = mi.add(mj);} catch (Exception e) {}
            if (su.isZero()) list.removeElementAt(i);
            else list.set(i,su);
            i--;
        }
    }
}
}
}

```



## 5.6 Addition von multivariaten Polynomen

Java-Methode `MultivariateQ.addMonomial`:

```
// Summe eines multivariaten Polynoms und eines Monoms:

public static MultivariateQ addMonomial (MultivariateQ p, MonomialQ m) {
    MultivariateQ su = new MultivariateQ(p);
    su.list.add(new MonomialQ(m));
    su.reduce();
    return su;
}

// Summe des gegebenen multivariaten Polynoms und eines Monoms:
// m ... Monom (2. Summand)

public MultivariateQ addMonomial (MonomialQ m) {
    return addMonomial(this,m);
}
```

Java-Methode `MultivariateQ.add`:

```
// Summe zweier multivariater Polynome:
// p1, p2 ... Gegebene Polynome (Summanden)

public static MultivariateQ add (MultivariateQ p1, MultivariateQ p2) {
    MultivariateQ su = new MultivariateQ(p1);
    for (int i=0; i<p2.getSize(); i++) {
        MonomialQ m = p2.getSummand(i);
        su = addMonomial(su,m);
    }
    return su;
}

// Summe des gegebenen und eines weiteren multivariaten Polynoms:
// p ... Weiteres Polynom (2. Summand)

public MultivariateQ add (MultivariateQ p) {
    return add(this,p);
}
```

## 5.7 Subtraktion von multivariaten Polynomen

Die Subtraktion von multivariaten Polynomen lässt sich auf die Vorzeichenumkehr von Monomen und die Addition von multivariaten Polynomen zurückführen:

**Java-Methode MultivariateQ.sub:**

```
// Differenz zweier multivariater Polynome:
// p1, p2 ... Gegebene Polynome (Minuend und Subtrahend)

public static MultivariateQ sub (MultivariateQ p1, MultivariateQ p2) {
    MultivariateQ di = new MultivariateQ(p1);
    for (int i=0; i<p2.getSize(); i++) {
        MonomialQ m = p2.getSummand(i);
        di = addMonomial(di,m.negate());
    }
    return di;
}

// Differenz des gegebenen und eines weiteren multivariaten Polynoms:
// p ... Weiteres Polynom (Subtrahend)

public MultivariateQ sub (MultivariateQ p) {
    return sub(this,p);
}
```

## 5.8 Multiplikation von multivariaten Polynomen

Java-Methode `MultivariateQ.mulMonomial`:

```
// Produkt eines multivariaten Polynoms und eines Monoms
// p ... Polynom (1. Faktor)
// m ... Monom (2. Faktor)

public static MultivariateQ mulMonomial (MultivariateQ p, MonomialQ m) {
    MultivariateQ pr = new MultivariateQ();
    for (int i=0; i<p.getSize(); i++)
        pr = addMonomial(pr,p.getSummand(i).mul(m));
    return pr;
}

// Produkt des gegebenen multivariaten Polynoms und eines Monoms:
// m ... Monom (2. Faktor)

public MultivariateQ mulMonomial (MonomialQ m) {
    return mulMonomial(this,m);
}
```

## Java-Methode MultivariateQ.mul:

```
// Produkt zweier multivariater Polynome:
// p1, p2 ... Gegebene Polynome (Faktoren)

public static MultivariateQ mul (MultivariateQ p1, MultivariateQ p2) {
    MultivariateQ pr = new MultivariateQ();
    for (int i=0; i<p1.getSize(); i++) {
        MonomialQ m1 = p1.getSummand(i);
        for (int j=0; j<p2.getSize(); j++) {
            MonomialQ m2 = p2.getSummand(j);
            pr = addMonomial(pr,m1.mul(m2));
        }
    }
    return pr;
}

// Produkt des gegebenen und eines weiteren multivariaten Polynoms:
// p ... Weiteres Polynom (2. Faktor)

public MultivariateQ mul (MultivariateQ p) {
    return mul(this,p);
}
```

## 5.9 Leitmonom eines multivariaten Polynoms

Bei der Division zweier univariater Polynome benötigt man die Leitkoeffizienten, also die Koeffizienten, die zum größten Exponenten gehören. Bei multivariaten Polynomen ist es komplizierter, weil mehrere Variable und damit mehrere Exponenten im Spiel sind. Zunächst muss eine Ordnungsrelation für Monome definiert werden. Im Folgenden wird die lexikographische Ordnung verwendet. Leitmonom ist, wenn eindeutig, das Monom mit dem größten Exponenten bezüglich  $a$ . Gibt es mehrere solche Monome, so nimmt man, wenn eindeutig, das Monom mit dem größten Exponenten bezüglich  $b$ . Existieren mehrere solche Monome, so ist, wenn eindeutig, das Monom mit dem größten Exponenten bezüglich  $c$  das Leitmonom, und so weiter.

Beispiel: Das multivariate Polynom  $5x^3z^2 - 2x^3y + 4yz^2$  hat das Leitmonom  $-2x^3y$ .

### Java-Methode `MultivariateQ.leadMonomial`

```
// Leitmonom eines multivariaten Polynoms (lexikographische Ordnung):
// p ... Gegebenes Polynom

public static MonomialQ leadMonomial (MultivariateQ p) {
    if (p.isZero()) return new MonomialQ();
    int iMax = 0;
    int[] eMax = p.getSummand(0).getExpo();
    for (int i=1; i<p.getSize(); i++) {
        int[] e = p.getSummand(i).getExpo();
        for (int j=0; j<MonomialQ.nVar; j++) {
            int d = e[j]-eMax[j];
            if (d < 0) break;
            else if (d > 0) {
                iMax = i;
                eMax = e;
                break;
            }
        }
    }
    return p.getSummand(iMax);
}
```

Quelle: Michael Kaplan, Computeralgebra

## 5.10 Division von multivariaten Polynomen

Bei multivariaten Polynomen gibt es – im Gegensatz zum univariaten Fall – keine Division mit Rest. Der folgende Algorithmus kann also nur verwendet werden, wenn der Dividend durch den Divisor teilbar ist.

### Java-Methode `MultivariateQ.div`:

```
// Quotient zweier multivariater Polynome:
// p1, p2 ... Gegebene Polynome (Dividend und Divisor)

public static MultivariateQ div (MultivariateQ p1, MultivariateQ p2)
throws Exception {
    if (p2.isZero()) throw new Exception("Division durch Null!");
    MultivariateQ r = new MultivariateQ(p1);
    MultivariateQ qu = new MultivariateQ();
    MonomialQ lm2 = p2.leadMonomial();
    while (r.getSize() > 0) {
        MonomialQ lm1 = r.leadMonomial();
        MonomialQ m = lm1.div(lm2);           // Ausnahme möglich
        qu = addMonomial(qu,m);
        r = sub(r,mul(p2,m));
    }
    return qu;
}
```

Quelle: Michael Kaplan, Computeralgebra

## 6 Integration

### 6.1 Trapezregel

Java-Methode trapezoidRule:

```
// Trapezregel (genauer Sehnentrapezregel):  
// a ... Untere Integrationsgrenze  
// b ... Obere Integrationsgrenze  
// n ... Zahl der Teilintervalle  
// Die Integrandenfunktion f muss definiert sein.  
  
public static double trapezoidRule (double a, double b, int n) {  
    double h = (b-a)/n;  
    double su = f(a)+f(b);  
    for (int i=1; i<n; i++) su += 2*f(a+i*h);  
    return su*h/2;  
}
```

### 6.2 Simpson-Regel

Java-Methode simpsonRule:

```
// Simpson-Regel:  
// a ... Untere Integrationsgrenze  
// b ... Obere Integrationsgrenze  
// n ... Zahl der Teilintervalle (muss gerade sein!)  
// Die Integrandenfunktion f muss definiert sein.  
  
public static double simpsonRule (double a, double b, int n) {  
    double h = (b-a)/n;  
    double su = f(a)+f(b);  
    for (int i=1; i<n; i+=2) su += 4*f(a+i*h);  
    for (int i=2; i<n; i+=2) su += 2*f(a+i*h);  
    return su*h/3;  
}
```

### 6.3 Romberg-Verfahren

Java-Methode rombergRule:

```
// Romberg-Näherung für ein bestimmtes Integral:
// a .... Untere Integrationsgrenze
// b .... Obere Integrationsgrenze
// it ... Zahl der Iterationen
// Die Integrandenfunktion f muss definiert sein.

public static double rombergRule (double a, double b, int it) {
    int n = 2;
    double[] t = new double[it+1];
    for (int k=0; k<=it; k++) {
        n *= 2;
        t[k] = trapezoidRule(a,b,n);
        int q = 1;
        for (int i=k-1; i>=0; i--) {
            q *= 4;
            t[i] = t[i+1]+(t[i+1]-t[i])/(q-1);
        }
    }
    return t[0];
}
```



## 7 Sonstiges

### 7.1 Gaußsche Osterformel, Version von 1816

Pseudocode:

```
function easter(y)
  // Vorausgesetzt: Jahreszahl  $y \geq 1683$ 
  if  $y \leq 1682$  then return undefined
   $a \leftarrow \text{mod}(y, 19)$ 
   $b \leftarrow \text{mod}(y, 4)$ 
   $c \leftarrow \text{mod}(y, 7)$ 
   $k \leftarrow \text{div}(y, 100)$ 
   $p \leftarrow \text{div}(8 \cdot k + 13, 25)$ 
   $q \leftarrow \text{div}(k, 4)$ 
   $M \leftarrow \text{mod}(15 + k - p - q, 30)$ 
   $d \leftarrow \text{mod}(19 \cdot a + M, 30)$ 
   $N \leftarrow \text{mod}(4 + k - q, 7)$ 
   $e \leftarrow \text{mod}(2 \cdot b + 4 \cdot c + 6 \cdot d + N, 7)$ 
   $o \leftarrow 22 + d + e$ 
  if  $d = 29$  and  $e = 6$  then  $o \leftarrow 50$  // 1. Ausnahmeregel
  if  $d = 28$  and  $e = 6$  and  $a > 10$  then  $o \leftarrow 49$  // 2. Ausnahmeregel
  if  $o \leq 31$  then return append( $o$ , ". März ", y)
  else return append( $o - 31$ , ". April ", y)
```

Quelle: [de.wikibooks.org/wiki/Algorithmensammlung:\\_Kalender:\\_Feiertage](https://de.wikibooks.org/wiki/Algorithmensammlung:_Kalender:_Feiertage)

Java-Methode:

```
public static String easter (int y) throws Exception {
  if (y <= 1682)
    throws new Exception("Formel nur für gregorianischen Kalender!");
  int a = y%19;
  int b = y%4;
  int c = y%7;
  int k = y/100;
  int p = (8*k+13)/25;
  int q = k/4;
  int M = (15+k-p-q)%30;
  int d = (19*a+M)%30;
  int N = (4+k-q)%7;
  int e = (2*b+4*c+6*d+N)%7;
  int o = 22+d+e;
```

```
if (d == 29 && e == 6) o = 50;           // 1. Ausnahmeregel
if (d == 28 && e == 6 && a > 10) o = 49; // 2. Ausnahmeregel
if (o <= 31) return ""+o+". März "+y;   // Rückgabewert für März
else return ""+(o-31)+". April "+y;     // Rückgabewert für April
}
```

## Index

- Addition
  - von Matrizen, 38
  - von Monomen, 60
  - von multivariaten Polynomen, 65
  - von rationalen Zahlen, 17
  - von univariaten Polynomen, 52
  - von Vektoren, 29
- Bestimmtes Integral, 71
- Binomialkoeffizient, 10
- charakteristisches Polynom, 48
- Determinante, 46
- Dezimalbruch, 24
- Dimensionsprüfung
  - für Matrizen, 34
  - für Vektoren, 27
- Division
  - von Monomen, 62
  - von multivariaten Polynomen, 70
  - von rationalen Zahlen, 22
  - von univariaten Polynomen, 56
- Eulersche Phi-Funktion, 11
- Fakultät, 9
- Gauß-Jordan-Algorithmus, 42
- Gaußsche Osterformel, 73
- ggT, 7
- Gleichungssystem
  - lineares, 42
- größter gemeinsamer Teiler, 7
- Horner-Schema, 51
- Integration, 71
- inverse Matrix, 44
- Jacobi-Symbol, 13
- Java-Klasse
  - MatrixQ, 34
  - MonomialQ, 58
  - MultivariateQ, 63
  - NumberQ, 15
  - PolynomialQ, 49
  - VectorQ, 27
- Java-Methode
  - binomialCoefficient, 10
  - easter, 73
  - eulerPhi, 12
  - firstPrime, 11
  - gcd, 7
  - isPrime, 5
  - jacobi, 14
  - lcm, 8
  - MatrixQ.add, 38
  - MatrixQ.characteristicPolynomial, 48
  - MatrixQ.determinant, 47
  - MatrixQ.dimensionQuadratic, 35
  - MatrixQ.dimensionsM, 34
  - MatrixQ.dimensionsProdMM, 41
  - MatrixQ.gaussJordan, 43
  - MatrixQ.inverse, 45
  - MatrixQ.mul (2), 40
  - MatrixQ.mul (3), 41
  - MatrixQ.rank, 37
  - MatrixQ.sub, 39
  - MatrixQ.swapColumns, 35
  - MatrixQ.swapLines, 35
  - MatrixQ.dimensionsMM, 34
  - MonomialQ.add, 60
  - MonomialQ.div, 62
  - MonomialQ.mul:, 61
  - MultivariateQ.add, 65
  - MultivariateQ.addMonomial, 65
  - MultivariateQ.div, 70
  - MultivariateQ.leadMonomial, 69
  - MultivariateQ.mul, 68
  - MultivariateQ.mulMonomial, 67
  - MultivariateQ.sub, 66
  - NumberQ.add, 17
  - NumberQ.add2, 18

- NumberQ.add3, 19
- NumberQ.div, 22
- NumberQ.div2, 23
- NumberQ.mul, 20
- NumberQ.mul2, 21
- NumberQ.normal, 16
- NumberQ.stringsDec, 25
- PolynomialQ.add, 52
- PolynomialQ.divmod, 56
- PolynomialQ.horner, 51
- PolynomialQ.leadCoeff, 50
- PolynomialQ.mul, 54
- PolynomialQ.mulMon, 54
- PolynomialQ.normal, 50
- PolynomialQ.sub, 53
- primeFactorization, 6
- rombergRule, 72
- simpsonRule, 71
- trapezoidRule, 71
- VectorQ.add, 29
- VectorQ.crossProduct, 33
- VectorQ.dimensionV, 27
- VectorQ.dimensionVV, 27
- VectorQ.innerProduct, 32
- VectorQ.mul, 31
- VectorQ.sub, 30

kgV, 8

kleinstes gemeinsames Vielfaches, 8

Kreuzprodukt, 33

Leitkoeffizient

- eines univariaten Polynoms, 50

Leitmonom

- eines multivariaten Polynoms, 69

lineares Gleichungssystem, 42

Matrix

- inverse, 44

Monom, 58

Multiplikation

- Matrix mal Vektor, 40
- von Matrizen, 41
- von Monomen, 61
- von multivariaten Polynomen, 67
- von rationalen Zahlen, 20
- von univariaten Polynomen, 54

Multivariates Polynom, 58

Normalisierung, 15

Osterformel

- gaußsche, 73

Phi-Funktion

- eulersche, 11

Polynom

- charakteristisches, 48
- multivariates, 58
- univariates, 49

Polynomdivision, 56

Primfaktorzerlegung, 6

Primzahltest, 5

Pseudocode-Funktion

- addQ, 17
- addQ2, 17
- addQ3, 18
- binomialCoefficient, 10
- determinantMQ, 46
- divmodPQ, 56
- divQ, 22
- divQ2, 23
- easter, 73
- gaussJordanQ, 42
- gcd, 7
- horner, 51
- inverseMQ, 44
- isPrime, 5
- jacobi, 13
- lcm, 8
- mulQ, 20
- mulQ2, 21
- numberQ, 15
- primeFactorization, 6
- rankMQ, 36
- toDecimalQ, 24

Rang

- einer Matrix, 36

rationale Zahl, 15

Romberg-Verfahren, 72

Simpson-Regel, 71

Skalare Multiplikation  
von Vektoren, 31

Skalarprodukt, 32

Subtraktion  
von Matrizen, 39  
von multivariaten Polynomen, 66  
von rationalen Zahlen, 19  
von univariaten Polynomen, 53  
von Vektoren, 30

Teiler  
größter gemeinsamer, 7

Trapezregel, 71

Umwandlung  
in einen Dezimalbruch, 24

Vertauschung  
von Matrixspalten, 35  
von Matrixzeilen, 35

Vielfaches  
kleinstes gemeinsames, 8

Zahl  
rationale, 15

Zerlegung  
in Primfaktoren, 6