Taylorreihen und ihre Implementierung mit JAVA

Ac 2016 - 2022

Ist f eine n-mal differenzierbare Funktion, dann heißt das Polynom

$$T_n(x;x_0)=\sum_{k=0}^{n}\frac{(x-x_0)^k}{k!}\cdot f^{(k)}(x_0)$$

Taylorpolynom n-ten Grades von f zur Entwicklungsstelle x_0 .

Ersetzt man n durch ∞ , so spricht man von einer **Taylorreihe** $T_\infty(x;x_0)$, die i.A. nicht konvergent ist.

Das Ziel ist die Erzeugung einer **konvergenten** Taylorreihe, die identisch mit f ist !

Für das entsprechende Taylorpolynom bedeutet das:
Je höher der Grad n des Polynoms, desto besser wird die Approximation in der Umgebung von x_0 .

Das **Restglied R_n** gibt den Unterschied zwischen f(x) und dem Taylorpolynom T_n an. Es gilt:

$$R_n(x;x_0)=\frac{(x-x_0)^{n+1}}{(n+1)!}\cdot f^{(n+1)}(\xi)\quad \text{für } \xi=x_0+\vartheta\cdot(x-x_0)\ \text{ mit }\ \vartheta\in]0;1[$$

$$\text{Für } x_0=0 \text{ gilt: } R_n(x)=\frac{x^{n+1}}{(n+1)!}\cdot f^{(n+1)}(\vartheta\cdot x)\quad \text{mit }\ \vartheta\in]0;1[$$

Das Prinzip für die Bildung des Taylorpolynoms T ist, **dass jede Ableitung von T(x) an der Stelle x_0 mit der entsprechenden Ableitung von f(x) übereinstimmen muss .**

Es gilt also für k von 0 bis n : $T^{(k)}(x_0)=f^{(k)}(x_0)$

Hierfür ein Beispiel:

f(x) = sin(x); $x_0 = 0$. (genauere Analyse weiter unten)

f ' (x) = cos(x) f " (x) = -sin(x) f ''' (x) = -cos(x) $f^{(4)}(x) = \sin(x) = f(x)$

Also ist f ' (0) = 1 f " (0) = 0 f ''' (0) = -1 $f^{(4)}(0) = 0$ usw.

Jedes 2. Glied fällt weg und für die restlichen Glieder alternieren die Vorzeichen.

Das Taylorpolynom für sin(x) um die Entwicklungsstelle 0 lautet dann:

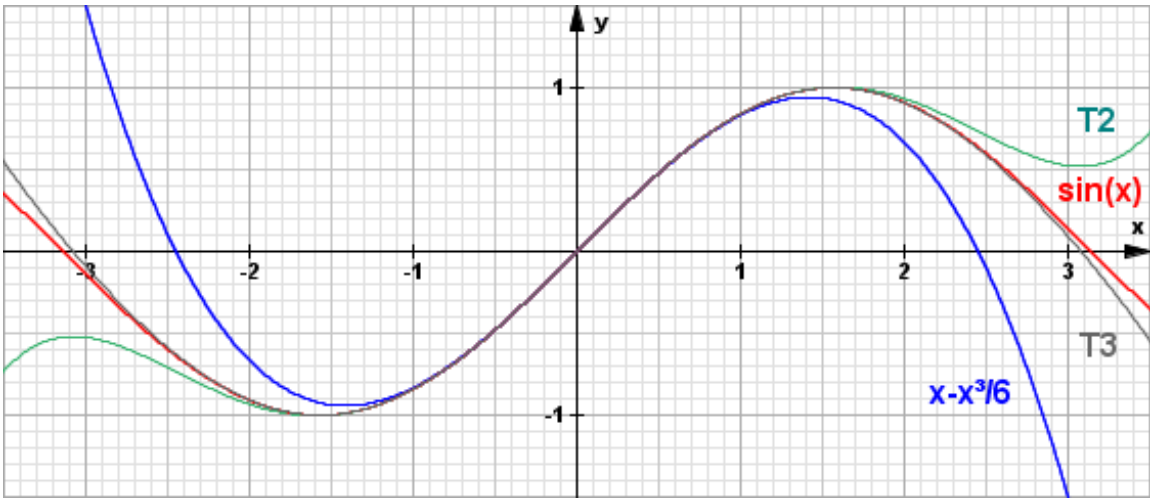
$$\sin(x)\approx T(x)=\sum_{k=0}^{n}\frac{(-1)^k}{(2k+1)!}\cdot x^{2k+1}=x-\frac{x^3}{6}+\frac{x^5}{120}-\frac{x^7}{5040}\pm\ldots$$

Restglied: $$R_n=(-1)^n\cdot\frac{x^{2n+1}}{(2n+1)!}\cdot\cos(\vartheta\cdot x)\ ;\ 0<\vartheta<1$$

Ergänzung: Für konvergente alternierende Reihen kann man auch mit dem Summanden hinter dem zuletzt verwendeten Summanden abschätzen (hier: Index 2n+3).

Es gilt dann : $|R_n| \le \frac{x^{2n+3}}{2n+3}$; gilt für konvergente alternierende Reihen

Grafische Veranschaulichung:



Anmerkungen:

T2 ist das Polynom für k = 0 bis 2, also bis zum Term $x^5 / 120$.
T3 ist das Polynom für k = 0 bis 3, also bis zum Term $x^7 / 5040$.

Fazit: Je höher der Grad n gewählt wird, um so besser ist die Approximation !
Gut ist die Approximation jedoch nur in einem bestimmten Bereich, etwa [0;1].

Weitere Beispiele für Taylorreihen :

$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k \cdot \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \pm \ldots \quad ; x \in \mathbb{R}$$

$$\arctan(x) = \sum_{k=0}^{\infty} (-1)^k \cdot \frac{x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \pm \ldots \quad ; \quad |x| \le 1$$

$$\ln(1+x) = \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{x^k}{k} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} \pm \ldots \quad ; -1 < x \le 1$$

Achtung: Hier beginnt der Summenindex mit k = 1 .

Die Reihen für arctan(x) und ln(1+x) **konvergieren sehr langsam**.

Weiter unten werden **schnellere Alternativen** erläutert.

Computerprogrammierung:

Zur Berechnung von Approximationswerten mit dem Computer muss ein Programm die Potenzsummen der Taylorreihe bilden können, was aber mit einer for-Schleife recht einfach erledigt werden kann.

Da mit größer werdendem Exponenten die (Potenz-) Summanden in der Regel immer kleiner werden, ist es aus numerischen Gründen ratsam, die Summanden "**von hinten nach vorne**" abzuarbeiten.

Beispiel zur Verdeutlichung des numerischen Problems :

$$\sum_{k=0}^{17} \frac{x^k}{k!} = 1{,}0100501670841682 \quad , \qquad \text{aber} \qquad \sum_{k=17}^{0} \frac{x^k}{k!} = 1{,}0100501670841680$$

Das auf 17 Dezimalen genaue Ergebnis ist übrigens: 1,01005016708416805

Hinweis: Alle Programme wurden mit JAVA 17 erstellt !

JAVA-Methode:

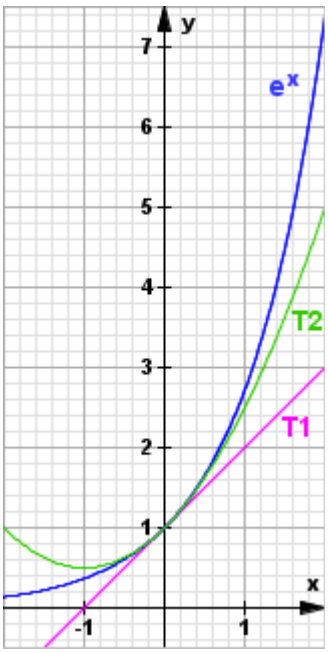
Mögliche Implementierung mit Funktionsterm f .

Aus Geschwindigkeitsgründen können Terme auch anders aufgestellt werden (Beispiele s. unten !).

```
static double f(double x, int k) {
      // Beispielfunktion für Taylorreihen;  hier:   ln(x)
      return 2.0 / (2 * k + 1) * Math.pow((x - 1) / (x + 1), 2 * k + 1);
}

static double taylorReihe(double x, int n) {
      // berechnet sum(f(x,k),k,0,n)
      double sum = 0.0;
      for (int k = n; k >= 0; k--)
            sum = sum + f(x, k);
      return sum;
}
```

1) Die Taylorreihe für e^x :



$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \ldots + \frac{x^n}{n!} + R_n \;;\; x > 0$$

Für das sog. "Restglied" gilt: $R_n = \frac{x^{n+1}}{(n+1)!} \cdot e^{\vartheta \cdot x}$ mit $0 < \vartheta < 1$

Für **x < 0** verwende man $e^{-|x|} = 1 / e^{|x|}$.

Anmerkung: Ersetzt man x durch **x·ln(a)** , so kann man mit obiger Formel auch $e^{x \cdot \ln(a)} = a^x$ berechnen.

Die e^x - Formel lässt sich umformen (HORNER-Schema), so dass man ohne Fakultäten und Potenzen auskommt:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + \frac{x}{1} \cdot (1 + \frac{x}{2} \cdot (1 + \frac{x}{3} \cdot (1 + \frac{x}{4} \cdot (1 + \ldots + \frac{x}{n-1} \cdot (1 + \frac{x}{n} \cdot 1)))) + R_n \;;\; x > 0$$

Nun kann man den Summenterm von “innen” nach “außen” abarbeiten, d.h. man beginnt mit 1 .

Ein geeignetes Java-Programm ist das folgende (ohne Eingabe-Check) :

```
static double expReihe(double x, int n) {
   // berechnet sum(x^i/i!,i,0,n) trickreich (HORNER-SCHEMA) !
   double sum = 1.0;
   for (int i = n; i > 0; i--)
      sum = sum / i * x + 1.0;
   return sum;
}
```

Die Reihe für e^x konvergiert am besten in der Nähe von x = 0 , daher betrachten wir zuerst x = 0,1 :

Für den Fall $e^{0,1}$ ist der maximale Fehler laut Restglied $\frac{0,1^{n+1}}{(n+1)!} \cdot e^{\vartheta \cdot 0,1} < \frac{0,1^{n+1}}{(n+1)!} \cdot e^{0,1} \approx \frac{1,105}{10^{n+1} \cdot (n+1)!}$

Wir berechnen $e^{0,1}$ = 1,105170918075647**6248117** ... auf 15 Dezimalen genau .

Ansatz: $\frac{1,105}{10^{n+1} \cdot (n+1)!} < 0,5 \cdot 10^{-15} \Rightarrow 10^{n+1} \cdot (n+1)! > 2,21 \cdot 10^{15} \Rightarrow n \geq 9$

Mit n = 9 ergibt sich $e^{0,1}$ = 1.1051709180756477 (In der Tat: 15 Dezimalen genau !) .

Probiert man allerdings **größere x-Werte wie x = 10** aus, so wird das Ergebnis sehr ungenau !

e^{10} = 10086.573192239859 (n=9) Ergebnis ganz falsch ! (richtig: 22026,4657948067165169...)

Allerdings muss laut Restgliedformel das n deutlich größer gewählt werden, was wir nun machen wollen:

Neuer Ansatz: $\frac{10^{n+1}\cdot e^{10}}{(n+1)!} < 5\cdot 10^{-16} \Rightarrow \frac{10^{n+1}\cdot 22026,47}{(n+1)!} < 5\cdot 10^{-16} \Rightarrow \cdot\frac{(n+1)!}{10^{n+1}} > 4,4\cdot 10^{19} \Rightarrow n \geq 57$

Mit n = 57 ergibt sich e^{10} = 22026.465794806714 (In der Tat: 15 Dezimalen genau !).
Allerdings ein Riesenrechenaufwand !

Zur Vermeidung dieses Rechenaufwands sollte man den Exponenten x z.B. durch die folgende Umformung "reduzieren" , falls er "zu weit von 0 entfernt" liegt:

$e^x = \left(e^{\frac{x}{m}}\right)^m$. Zum Beispiel kann man **m = (int) (10x)** für x > 0,1 wählen . Dann gilt: x/m ≈ 0,1 .

Man berechnet also erst $e^{\frac{x}{m}} \approx e^{0,1}$ mit der Summenformel und potenziert das Ergebnis mit m .

JAVA-Methode für e^x (mit Reduktion des Arguments) :

```
static double expReihe_redu_pow(double x, int n) {
   // berechnet sum(x^i/i!,i,0,n)
   // Reduktion durch e^x = (e^(x/j))^j ;  j = [10x] ,  falls x > 0,1
   if (x == 0)
      return 1;

   if (x < 0)
      return 1.0 / expReihe_redu_pow(-x, n);

   boolean reduziert = (x > 0.1);
   int m = 1;
   if (reduziert) { // falls x > 0.1:  e^x = (e^(x/m))^m ;  m = [10x]
      m = (int) (10 *x);
      x = x / m;
   }

   // Beginn der Iteration
   double sum = 1.0;
   for (int i = n; i > 0; i--)
      sum = sum / i * x + 1.0;

   if (reduziert)
      sum = Math.pow(sum, m);

   return sum;
}
```

Für e^x mit x > 1 reicht n = 9 nicht mehr aus, wie man an den folgenden Beispielen erkennen kann:

e^2 = 7,389056098930662 13 Dezimalen korrekt (richtig: 7,3890560989306502 23...)
e^{10} = 22026,465794806893 9 Dezimalen korrekt (richtig: 22026,4657948067165169...)
e^{20} = 4.85165195409797E8 13 Dezimalen korrekt (richtig: 4,8516519540979027796...E8)
e^{100} = 2.6881171418163485E43 12 Dezimalen korrekt (richtig: 2,688117141816135448...E43)

Bei diesen Exponenten x pflanzt sich der Fehler um ein Vielfaches fort, da ja das Ergebnis von $e^{0,1}$ noch mit m potenziert werden muss (Potenzierung vergrößert den Fehler um ca. den Faktor m !).

Eine Erhöhung des n-Wertes auf n = 10 bringt keine Verbesserung bei e^2 = 7.389056098930662 (13 richtige Dezimalen). Auch durch weitere Erhöhung von n wird es nicht genauer.

Dies kann nur mit der beschränkten Genauigkeit bei double-Rechnungen zu tun haben !

Wie sieht es bei deutlich größeren x-Werten aus ?
Kann man hier die Genauigkeit durch mehr Schleifenumläufe erhöhen ?

Tests ergeben, dass sich auch hier keine Verbesserung erzielen lässt !
Die Genauigkeit der double-Rechnungen hat ihre Grenzen !

Eine Alternative ist die Verwendung des Datentyps BigDecimal !!

Hiermit kann man beliebig viele Stellen berechnen lassen und ist nicht von Beschränkungen wie bei "double" abhängig !

Alternative :

Eine weitere Möglichkeit der Verkleinerung des Exponenten x im Falle großer x-Werte ist die folgende:

Falls x > ln(2): Setze **x = k · ln(2) + r** , wobei r ∈ [0 ; ln(2) [der Rest von x nach der Reduktion ist.

Dann gilt: $e^x = e^{k\cdot\ln(2)+r} = e^{k\cdot\ln(2)}\cdot e^r = (e^{\ln(2)})^k\cdot e^r = 2^k\cdot e^r$

Bestimmung von k und r :

Berechne: k = int (x / ln(2)) und r = frac (x / ln(2)) = x % ln(2) [in Java]

Der Wert für n zur Berechnung von e^r auf z.B. 15 Dezimalen muss vorher bekannt sein !
Er ergibt sich aus dem Restglied, wobei r < ln(2) zu berücksichtigen ist :

$$R_n = \frac{r^{n+1}}{(n+1)!}\cdot e^{\vartheta\cdot r} < \frac{r^{n+1}}{(n+1)!}\cdot e^{r} < \frac{\ln(2)^{n+1}}{(n+1)!}\cdot e^{\ln(2)} = \frac{\ln(2)^{n+1}}{(n+1)!}\cdot 2 = 2\cdot\frac{\ln(2)^{n+1}}{(n+1)!}$$

$$R_n < 5\cdot 10^{-16} \Rightarrow 2\cdot\frac{\ln(2)^{n+1}}{(n+1)!} < 5\cdot 10^{-16} \Rightarrow \frac{(n+1)!}{\ln(2)^{n+1}} > 4\cdot 10^{15} \Rightarrow n \geq 15$$

Je nach Größe von r kann auch ein kleineres n genügen !

Die JAVA-Methoden für e^x sehen dann so aus :

```
static double ln2 = Math.log(2);

static double expReihe_redu_mult(double x, int n) {
   // berechnet sum(x^i/i!,i,0,n)
   // Reduktion durch  e^x = 2^k * e^r; k = int(x/ln(2)); r = frac(x/ln(2)), falls x > ln2
   if (x == 0)
      return 1;

   if (x < 0)
      return 1.0 / expReihe_redu_mult(-x, n);

   boolean reduziert = (x > ln2);
   long k = 1;
   if (reduziert) {
      // falls x > ln2:  e^x = 2^k * e^r
      k = (long) (x / ln2);
      x = x % ln2;
      System.out.println("k = " + k + "     x = " + x);
   }
   // Beginn der Iteration
   double sum = 1.0;
   for (int i = n; i > 0; i--)
      sum = sum / i * x + 1.0;

   System.out.println("1 << k = " + (1 << k));
   if (reduziert) {
      if (k < 63)
         sum = (1 << k) * sum; // 2^k * sum
      else
         sum = yMal2hochK(sum, k); // da  1 << k (für k > 62) den Long-Bereich übertrifft
   }

   return sum;
}

static double yMal2hochK(double y, long k) {
   // Hilfsmetode für expReihe_redu_mult
   for (long i = 0; i < k; i++)
      y = y*2;
    return y;
}
```

Einige Ergebnisse mit n = 15 Iterationen :

e^{1} = 2.7182818284590455 15 Dezimalen korrekt (richtig: 2,7182818284590452...)

$e^{\ln(2)}$ = 1.9999999999999998 15 Dezimalen korrekt(Rundung) richtig: 2,0

e^{3} = 20.085536923187668 15 Dezimalen korrekt (richtig: 20,0855369231876677 ...)

e^{10} = 22026.465794806725 14 Dezimalen korrekt (richtig: 22026.46579480671651 ...)

e^{20} = 4.851651954097906 4E8 14 Dezimalen korrekt (richtig: 4,851651954097902779...E8)

e^{100} = 2.688117141816 1445E43 13 Dezimalen korrekt (richtig: 2,688117141816135448...E43)

Fazit: Je größer x, desto deutlicher macht sich die beschränkte Genauigkeit bei double-Rechnungen bemerkbar (s. oben) !

Ergänzung (Eulersche Zahl e)

Setzt man in der e^x - Formel **x = 1**, so erhält man die **Eulersche Zahl e ≈ 2,718281828459045235**.

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1+\frac{1}{1}\cdot(1+\frac{1}{2}\cdot(1+\frac{1}{3}\cdot(1+\frac{1}{4}\cdot(1+\ldots+\frac{1}{n-1}\cdot(1+\frac{1}{n}))))) + R_n$$

Algorithmus: `sum = 1.0;  für i von n ab bis 1:  sum = sum / i + 1.0`

Das Restglied lässt sich speziell für e (e^x mit x = 1) geschickt abschätzen:

Für x = 1 gilt: $R_n = \frac{1}{(n+1)!}\cdot e^{\vartheta} \;\; mit \;\; 0<\vartheta<1\,; \;\; \Rightarrow \; R_n < \frac{e}{(n+1)!}$

Wir wollen e auf 15 Dezimalen genau berechnen, d.h. $R_n = \frac{e}{(n+1)!} < \varepsilon = 5\cdot 10^{-16}$.

Die Umformung liefert $(n+1)! > 2e\cdot 10^{15} \approx 5{,}437\cdot 10^{15} \Rightarrow n \geq 17$

In der Tat ist $e \approx \sum_{k=17}^{0} \frac{1}{k!} = 2{,}718281828459045$ auf 15 Dezimalen genau!

Literaturwert: e = 2,71828182845904523536 ...

Übrigens: Eine Rechnung mit Fakultäten (k!) statt mit dem ausgeklammerten Term für e liefert das Ergebnis 2,7182818284590455

Zum Vergleich: Math.exp(1) sowie Math.E liefern auch das Ergebnis 2,718281828459045 .

2) Die Taylorreihe für **ln(x)** bzw. **ln(1 + x)** :

$$\ln(1+x) = \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{x^k}{k} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} \pm \ldots \; ; \quad -1 < x \le 1$$

Da diese Reihe **sehr langsam** konvergiert, konstruiert man mittels ln(1+x) - ln(1-x) = ln((1+x) / (1-x))

eine neue, deutlich besser konvergierende Reihe.

$$\ln(1+x) - \ln(1-x) = \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{x^k}{k} - \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{(-x)^k}{k}$$

$$= x - \frac{x^2}{2} + \frac{x^3}{3} \mp \ldots - (-x - \frac{(-x)^2}{2} + \frac{(-x)^3}{3} \mp \ldots)$$

Man erkennt, dass sich Potenzen mit geraden Exponenten wegsubtrahieren. Daher bleibt folgendes:

$$\ln(\frac{1+x}{1-x}) = \ln(1+x) - \ln(1-x) = 2 \cdot (x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \ldots) = 2 \cdot \sum_{k=0}^{\infty} \frac{x^{2k+1}}{2k+1}$$

Die Substitution $y = \frac{1+x}{1-x} \Leftrightarrow x = \frac{y-1}{y+1}$ liefert $\ln(y) = 2 \cdot \sum_{k=0}^{\infty} \frac{1}{2k+1} \cdot \left(\frac{y-1}{y+1}\right)^{2k+1}$

Durch Umbenennung (x für y) erhält man endlich die ln(x) - Reihe:

$$\ln(x) = 2 \cdot \sum_{k=0}^{\infty} \frac{1}{2k+1} \cdot \left(\frac{x-1}{x+1}\right)^{2k+1} + R_n$$

$$= 2 \cdot (\frac{x-1}{x+1} + \frac{1}{3} \cdot \left(\frac{x-1}{x+1}\right)^3 + \ldots + \frac{1}{2n+1} \cdot \left(\frac{x-1}{x+1}\right)^{2n+1}) + R_n \; ; \quad x > 0$$

Für das Restglied gilt:

$$|R_n| = \frac{(x-1)^2}{2 \cdot |x|} \cdot \left(\frac{x-1}{x+1}\right)^{2n}$$

Diese Reihe lässt sich mittels der Substitution

z = (x-1) / (x+1)

einfacher notieren und bringt Rechenvorteile, wenn man geschickt ausklammert:

$$\ln(x) = 2\cdot\sum_{k=0}^{\infty}\frac{z^{2k+1}}{2k+1} = 2\cdot(z+\frac{z^3}{3}+\frac{z^5}{5}+\ ...\ +\frac{z^{2n+1}}{2n+1}+\ ...) \qquad \text{mit} \qquad z=\frac{x-1}{x+1}$$

Ausklammern: $\ln(x) = 2\cdot(\frac{z}{1}+z^2\cdot(\frac{z}{3}+z^2\cdot(\frac{z}{5}+z^2\cdot(\frac{z}{7}+\ ...\ +z^2\cdot(\frac{z}{2n+1}+\ ...)...)$

Diese Reihe für ln(x) konvergiert für alle x > 0, aber am besten in der Nähe von x = 1 !

Falls **0 < x < 1**, dann sind sowohl z als auch ln(x) negativ !

In diesen Fällen verwende man ln(1/x) = ln(1) – ln(x) = -ln(x) bzw **ln(x) = - ln(1/x)**

Ein einfaches **Java-Programm** für diese Ausklammer-Methode (HORNER-Schema):

```
static double lnReiheTaylorAusklammern(double x, int n) {
   // ln(x) = 2*sum(z^(2k+1)/(2k+1),k,0,n)   mit  z=(x-1)/(x+1)  ;  für x > 0 !!

   double z = (x - 1) / (x + 1);
   double quadr= z*z;
   double sum = 0.0;
   for (int k = 2*n+1; k > 0; k -= 2)
      sum = sum * quadr + z / k;

   return 2 * sum;
}
```

Beispielrechnung für **ln(2)** mit Restgliedabschätzung (ohne Reduzierung):

Wir wollen ln(2) auf 15 Dezimalen genau berechnen, d.h. $|R_n| = \frac{1}{2\cdot 2}\cdot\left(\frac{1}{3}\right)^{2n}$

Aus dem Ansatz $R_n < 5\cdot10^{-16}$ folgt:

$$\frac{1}{4}\cdot\left(\frac{1}{3}\right)^{2n} < 5\cdot10^{-16} \Leftrightarrow 3^{2n} > \frac{1}{2}\cdot10^{15} \Leftrightarrow 2n\cdot\lg(3) > \lg(\frac{1}{2})+15 \Leftrightarrow n > \frac{15-\lg(2)}{2\cdot\lg(3)} \approx 15{,}4$$

Daher sollte n = 16 für die vorgegebene Genauigkeit genügen. Wir berechnen also

$\ln(2) \approx \sum_{k=0}^{16}\frac{2}{2k+1}\cdot\left(\frac{1}{3}\right)^{2k+1} = 0.6931471805599455$. Dies ist auf 15 Dezimalen genau !

Beispielrechnung für **ln(100)** mit Restgliedabschätzung:

$$\frac{99^2}{2\cdot100}\cdot\left(\frac{99}{101}\right)^{2n} < 5\cdot10^{-16} \Leftrightarrow \left(\frac{101}{99}\right)^{2n} > 9801\cdot10^{13} \Leftrightarrow 2n\cdot\lg(\frac{101}{99}) > \lg(9801)+13 \Leftrightarrow$$

$$n > \frac{13+\lg(9801)}{2\cdot\lg(\frac{101}{99})} \approx 978{,}1$$

Immerhin müssen wir hier schon bis n = 979 rechnen, um die vorgegebene Genauigkeit zu erreichen.

Ein ziemlich großer Rechenaufwand !

Wir berechnen also $\ln(100) \approx \sum_{k=0}^{979} \frac{2}{2k+1} \cdot \left(\frac{99}{101}\right)^{2k+1} = 4.60517018598809$.

14 Dezimalen sind genau, die 15. wird nicht angezeigt (müsste aber definitiv 1 sein !)

Allgemeine Restgliedabschätzung für Genauigkeit auf d Dezimalen:

$$|R_n| = \frac{(x-1)^2}{2\cdot|x|} \cdot \left(\frac{x-1}{x+1}\right)^{2n} < 0{,}5 \cdot 10^{-d} \quad \Rightarrow \quad \lg\left(\frac{(x-1)^2}{|x|}\right) + \lg\left(\left(\frac{|x-1|}{x+1}\right)^{2n}\right) > -d \quad \Rightarrow$$

$$2\cdot\lg(|x-1|) - \lg(|x|) + 2n\cdot\lg\left(\frac{|x-1|}{x+1}\right) > -d \quad \Rightarrow \quad 2n\cdot\lg\left(\frac{|x-1|}{x+1}\right) > \lg(|x|) - 2\cdot\lg(|x-1|) - d \Rightarrow$$

$$\boxed{n > \frac{\lg(x) - 2\cdot\lg(|x-1|) - d}{2\cdot\lg\left(\frac{|x-1|}{x+1}\right)} \quad \text{für} \quad x>0}$$

Probe, ob die Formel für x = 0,5 funktioniert :

Man errechnet n > 15,4 , also sollte n = 16 reichen, um 15 Dezimalen zu bestimmen.

Ergebnis: `ln(0.5) = -0.6931471805599453` Alle 16 Dezimalen sind korrekt !

Vorsicht:

Falls x ganz nahe bei 1 liegt (etwa 0,99 oder 1,001) , dann bewegt sich lg(|x-1|) gegen -Unendlich, und das ist ungünstig für obige Restgliedabschätzung !

In diesem Falle sollte man besser z.B. ln(x) = ln(0,8x * 1,25) = ln(0,8x) + ln(1,25) verwenden !

Zurück zu den großen x-Werten :

Wegen des hohen Rechenaufwands bei **großen Argumenten x** sollte man x in Faktoren zerlegen, die "in der Nähe von 1" liegen (aber nicht sehr nahe an 1, wie die vorherige Betrachtung gezeigt hat), falls **x > √2** ≈ 1,414 (Erläuterung s. unten)

Vorgehensweise:

Man zerlegt x gemäß $x = 2^m \cdot 2^{-m} \cdot x$ mit geeignetem m .
So erhält man wegen ln(x) = ln(ab) = ln(a) + ln(b) folgendes :

ln(x) = $\ln(2^m) + \ln(x/2^m) = \ln(2^{2m/2}) + \ln(x/2^m) = 2m\cdot\ln(2^{1/2}) + \ln(x/2^m) =$ **$2m \cdot \ln(\sqrt{2}) + \ln(x / 2^m)$** .
Dabei sollte $x/2^m$ zwischen $1/\sqrt{2}$ und $\sqrt{2}$ liegen . Wie erreicht man das ?
Am besten dividiert man x so lange durch 2, bis das Ergebnis unter √2 liegt.

Beispiel: x = 100 (vgl. Ergebnis oben)

100/2 = 50 50/2 = 25 25/2 = 12,5 12,5/2 = 6,25 …. Ergebnis: $100/2^7 = 0{,}78125 < \sqrt{2}$
Mit m = 7 erhalten wir: $\ln(100/2^7) = \mathbf{14 \cdot \ln(\sqrt{2}) + \ln(100 / 2^7)} = ?$

Wir benötigen 2 Logarithmus-Approximationen mit x = 0,78125 und $x = \sqrt{2}$.

Wie viele Schleifenumläufe braucht man, um 15 Dezimalen richtig zu berechnen ?

x = 0,78125 : n > 7,6 , also bis n = 8 Approximation ln(0,78125) = `-0.24686007793152578`
Verwendet man aber 1 / x = 1,28 , so ergibt sich genau das gleiche Ergebnis !

$x = \sqrt{2}$: n > 9,2 , also bis n = 10 Approximation $\ln(\sqrt{2})$ = `0.34657359027997275...`

Approximation für $\mathbf{\ln(100) = 14 \cdot \ln(\sqrt{2}) + \ln(100 / 2^7)}$ =
`4.852030263919619 - 0.24686007793152578 = 4,60517018598809322`
Hier sind 14 Stellen richtig !
Genauer wird es auch durch Erhöhung der n-Werte nicht !

Die Transformation (Reduzierung) $\mathbf{\ln(x) = 2m \cdot \ln(\sqrt{2}) + \ln(x / 2^m)}$
ist aus Geschwindigkeitsgründen auf jeden Fall der direkten Methode vorzuziehen.

Alternative Reduktionsmöglichkeit:

Man kann auch $\mathbf{\ln(x) = 2 \cdot \ln(\sqrt{x})}$ verwenden (evtl. mehrmals), um x zu verkleinern !

Beispiel: x = 2500

$2500^{1/23} = 1{,}4051... < \sqrt{2}$
Dann ist $\ln(2500) = 23 \cdot \ln(2500^{1/23})$ = `7.824046010856293 (14 Dezimalen)`

Weitere Beispiele:

`ln(10)` = `2.3025850929940463` die letzten Dezimalen müssten 56 sein
`ln(10^6)` = `13.815510557964277` die letzte Dezimale müsste 4 sein

JAVA-Methoden (incl. Reduzierung) :

```
static int nMinLnFkt(double x, int digits) {
   // berechnet das minimale n für die Taylorformel von ln(x)
   //
   // Zur Bestimmung eines genügend großen n-Wertes für d Dezimalen:
   // suche n so, dass gilt:
   //  (x-1)² / (2*|x|) * ((x-1)/(x+1))^(2n) < 0,5 * 10^(-d)   ==>
   //  (x-1)² / |x| * ((x-1)/(x+1))^(2n) < 10^(-d)   ==>
   //  2*lg(|x-1|) - lg(x) + 2n * lg(|x-1|/(x+1)) > -d   ==>
   // ========================================================================
   //    n  >  [ lg(x)-2*lg(|x-1|)-d ] / [ 2*lg(|x-1|/(x+1)) ]    für  x > 0
   // ========================================================================
   double zaehler = Math.log10(x) - 2.0*Math.log10(Math.abs(x-1.0))-digits;
   double nenner = 2.0*Math.log10(Math.abs(x-1.0)/(x+1.0));
   double bruch = zaehler / nenner;

   if (bruch % 1 == 0.0)
      return (int) bruch;

   return (int) (1.0 + bruch);
}


static double wurz2 = Math.sqrt(2);

static double lnReiheTaylor(double x, int digits) {
   // ln(x) = 2*sum(z^(2k+1)/(2k+1),k,0,n)   mit  z=(x-1)/(x+1)
   // incl. Reduzierung , falls  x > √(2)
   if (x <= 0)
      throw new ArithmeticException("x muss positiv sein !");

   if (x == 1)
      return 0;

   if (x < 1)
      return  -lnReiheTaylor(1/x, digits);

   int m = 0;
   boolean reduziert = (x > wurz2);
   if (reduziert) {
      // ln(x) = 2m·ln(√(2)) + ln(x/2^m)
      do {
         x = x / 2;
         m++;
      } while (x > wurz2);
   }

   int n = nMinLnFkt(x, digits);
   double sum = lnReiheTaylorAusklammern(x, n); // von oben

   if (reduziert) { // berechne ln(√(2))
      x = wurz2;
      n = nMinLnFkt(x, digits);
      double sum2 = lnReiheTaylorAusklammern(x, n);  // von oben
      sum = sum + 2*m*sum2;
   }

   return sum;
}
```

3) Die Taylorreihe für **sin(x)** / vgl. Seite 2 ! :

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \cdot \frac{x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} \pm \ldots + (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!} + R_n \; ; \; x \in \mathbb{R}$$

Umformen des Summenterms durch geschicktes Ausklammern (HORNER-Schema):

$$\sin(x) = x \cdot (1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \frac{x^8}{9!} \pm \ldots + (-1)^n \cdot \frac{x^{2n}}{(2n+1)!}) + R_n$$

$$= x \cdot (1 - \frac{x^2}{2 \cdot 3} \cdot (1 - \frac{x^2}{4 \cdot 5} \cdot (1 - \frac{x^2}{6 \cdot 7} \cdot (1 - \frac{x^2}{8 \cdot 9} \cdot (1 - \ldots - \frac{x^2}{2n \cdot (2n+1)}) \ldots)) + R_n$$

Für das sog. "**Restglied**" R_n gilt: $R_n = (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!} \cdot \cos(\vartheta \cdot x) \; mit \; 0 < \vartheta < 1$

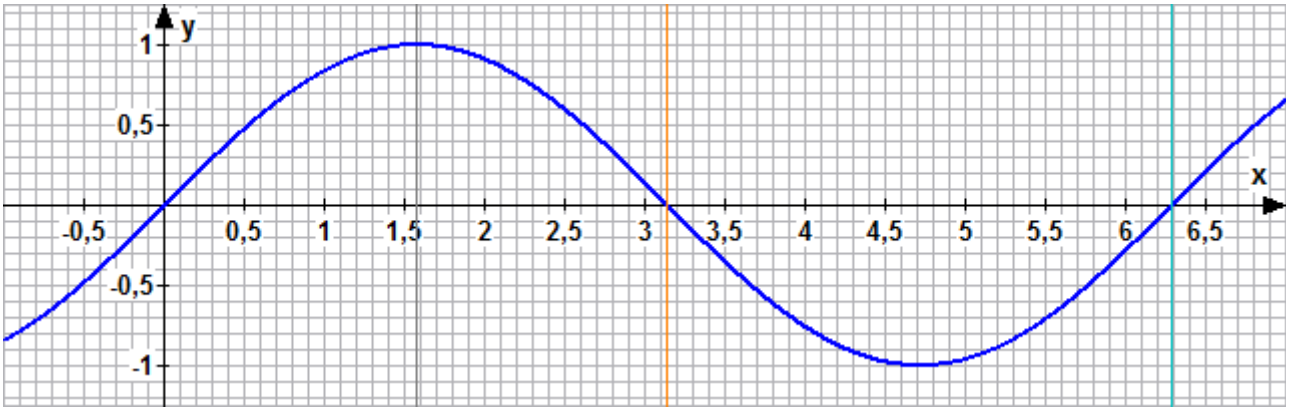
Da es sich um eine alternierende Reihe handelt, kann man auch so abschätzen: $R_n < \frac{x^{2n+3}}{(2n+3)!}$

Wählt man **$0 < x < \pi / 2$** , dann ist der Fehler sogar kleiner als $\frac{\left(\frac{\pi}{2}\right)^{2n+3}}{(2n+3)!}$

Für 15 richtige Dezimalen: $\frac{\left(\frac{\pi}{2}\right)^{2n+3}}{(2n+3)!} < 0{,}5 \cdot 10^{-15} \Rightarrow (2n+3)! \cdot \left(\frac{2}{\pi}\right)^{2n+3} > 2 \cdot 10^{15} \Rightarrow n \geq 9$

Wie transformiert man nun eine Zahl x so, dass sie in den Bereich [0 ; π /2] fällt ??

Dazu verwendet man die Periodizität des Sinus sowie andere Eigenschaften (s. Grafik) :



Für x = 0 gilt sin(0) = 0 und für x = π /2 gilt sin(π /2) = 1

Für x < 0 verwendet man: sin(x) = -sin(-x) .

Für x ≥ 2π verwendet man: sin(x) = sin(x-n·2π) ; man ersetzt x = x-2π , bis x < 2π .

Für π ≤ x < 2π verwendet man: sin(x) = -sin(x-π) .

Für π /2 < x < π verwendet man: sin(x) = sin(π -x) .

Beispielrechnung für sin(2) mit Restgliedabschätzung:

x = 2 muss zuerst transformiert werden mittels sin(2) = sin(π -2) ; also **x < π / 2**

Wir wollen sin(2) = sin(π -2) auf 15 Dezimalen genau berechnen, d.h. wir verwende n = 9 (s. oben) .

Daher ist zu berechnen:

$$\sin(2) \approx \sum_{k=0}^{9} (-1)^k \frac{(\pi-2)^{2k+1}}{(2k+1)!} = 0{,}9092974268256816$$ Sogar alle 16 Stellen sind korrekt !

Beispielrechnung für sin(-25) :

x = -25 muss zuerst transformiert werden:
sin(-25) = -sin(25) = -sin(25 - 3·2 π) = sin(25 - 3·2 π - π) = sin(π - (25 - 3·2 π - π)) .

Vereinfacht ist das sin(8 π - 25) .

Da 8 π -25 = 0,13 sehr klein ist, reicht hier sogar eine Rechnung mit n = 4 :

$$\sin(-25) \approx \sum_{k=0}^{4} (-1)^k \frac{(8\pi-25)^{2k+1}}{(2k+1)!} = 0{,}13235175009777206$$ 14 Stellen sind korrekt !

JAVA-Methode:

```
static double pi = Math.PI;
static double zweiPi = 2 * Math.PI;
static double halbePi = Math.PI / 2;


static double sinReiheTaylorAusklammern(double x, int n) {
   // berechnet sum((-1)^k*x^(2k+1)/(2k+1)!,k,0,n) mit Ausklammern
   if (x == 0)
      return 0;

   if (x < 0)
      return -sinReiheTaylorAusklammern(-x, n);

   // transformiere ggfs.
   while (x >= zweiPi)
      x = x - zweiPi;
   if (x >= pi)
      return -sinReiheTaylorAusklammern(x-pi, n);
   if (x > halbePi)
      x = pi - x;

   double quad = x * x;
   double sum = 1.0;
   for (int k = n; k > 0; k--)
      sum = 1.0 - sum / (2*k) / (2*k+1) * quad ;

   return x * sum;
}
```

Der Cosinus lässt sich wegen cos(x) = sin(x+π /2) auf den Sinus zurückführen !

Alternative: $\cos(x) = \sum_{k=0}^{\infty} (-1)^k \cdot \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \pm \ldots \quad ; x \in \mathbb{R}$

Der Tangens lässt sich wegen tan(x) = sin(x) / sin(x+π /2) auch auf den Sinus zurückführen !

Alternative: $\tan(x) = \sum_{k=1}^{\infty} \frac{(-4)^k \cdot (1-4^k) \cdot B_{2k}}{(2k)!} \cdot x^{2k-1} \; ; \; |x| < \frac{\pi}{2} \; ; \; B_{2k}$ = 2k-te Bernoulli-Zahl

Genaue Definition der $\mathbf{B_k}$ (rekursiv) :

$$B_0 = 1; \; B_1 = -\frac{1}{2}; \quad B_k = \frac{-1}{k+1} \cdot \sum_{j=0}^{k-1} \binom{k+1}{j} \cdot B_j \quad \text{für } k \geq 2$$

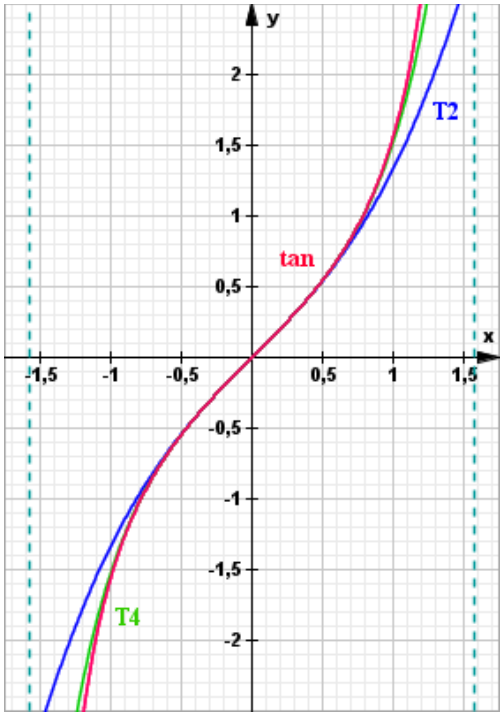
Achtung : Für **ungerade** Indizes $k \geq 3$ gilt: $\mathbf{B_k = 0}$

Einige Bernoulli-Zahlen :
$B_0 = 1$ $B_1 = -1/2$ $B_2 = 1/6$ $B_4 = -1/30$ $B_6 = 1/42$ $B_8 = -1/30$ $B_{10} = 5/66$ $B_{12} = -691/2730$
$B_{14} = 7/6$ $B_{16} = -3617/510$ $B_{18} = 43867/798$ $B_{20} = -174611/330$ $B_{22} = 854513/138$
$B_{24} = -236364091/2730$ $B_{26} = 8553103/6$ $B_{28} = -23749461029/870$ $B_{30} = 8615841276005/14322$

$$\tan(x) = x + \frac{1}{3} \cdot x^3 + \frac{2}{15} \cdot x^5 + \frac{17}{315} \cdot x^7 + \frac{62}{2835} \cdot x^9 + \frac{1382}{155925} \cdot x^{11} + \ldots \; ; \; |x| < \frac{\pi}{2}$$

Man erkennt eine gute Konvergenz für |x| < 0,5 .

Allerdings ist die Berechnung äußerst aufwendig, weil die Bernoulli-Zahlen rekursiv ermittelt werden müssen !



4) Die Taylorreihe für **arctan(x)** :

$$\arctan(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \pm \ldots + (-1)^n \cdot \frac{x^{2n+1}}{2n+1} + R_n \;;\; |x| \le 1$$

Umformen des Summenterms durch geschicktes Ausklammern:

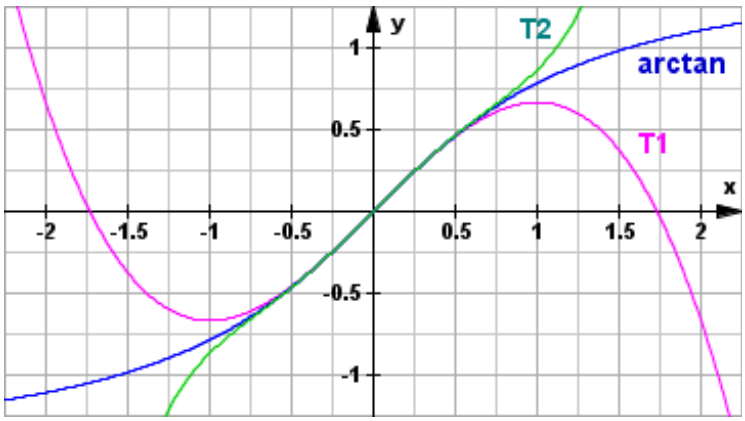
$$\arctan(x) = x \cdot (1 - \frac{x^2}{3} + \frac{x^4}{5} - \frac{x^6}{7} \pm \ldots + (-1)^k \frac{x^{2n}}{2n+1}) + R_n$$

$$\arctan(x) = x \cdot (1 - x^2 \cdot (\frac{1}{3} - x^2 \cdot (\frac{1}{5} - x^2 \cdot (\frac{1}{7} - \ldots - x^2 \cdot (\frac{1}{2n+1} \ldots)) + R_n$$

Für $x < 0$ verwendet man: $f(-x) = -f(x)$

Für das sog. "Restglied" R_n gilt:

$$R_n = (-1)^n \cdot \frac{x^{2n+1}}{2n+1} \cdot \frac{1}{1 + \vartheta \cdot x^2} \quad \text{mit } 0 < \vartheta < 1$$



Die Reihe **konvergiert ziemlich langsam** und am besten in der Nähe von 0 !
Man beachte den Konvergenzradius !

Wegen der alternierenden Reihe kann man schließen: $R_n < \frac{x^{2n+3}}{2n+3}$

Soll z.B. **arctan(1)** auf d Dezimalen genau sein, muss n so gewählt werden, dass $\frac{1}{2n+3} < 0{,}5 \cdot 10^{-d}$.

Beispiel: $d = 15$ $\quad \frac{1}{2n+3} < 0{,}5 \cdot 10^{-15} \Leftrightarrow 2n+3 > 2 \cdot 10^{15} \Leftrightarrow n \ge 10^{15}$

Das ist entschieden zu viel (!) , denn der Integer-Bereich für n wird deutlich überschritten !

Wählt man aber **x < 0,1** mit arctan(0,1) = `0.099668652491162027378446119878020590243...` ,

dann ist der Fehler kleiner als $\frac{0{,}1^{2n+3}}{2n+3} = \frac{1}{(2n+3) \cdot 10^{2n+3}}$.

Für d Nachkommastellen gilt dann: $\frac{1}{(2n+3) \cdot 10^{2n+3}} < 0{,}5 \cdot 10^{-d} \Leftrightarrow (2n+3) \cdot 10^{2n+3} > 2 \cdot 10^{d}$

$\Leftrightarrow \quad \lg(2n+3) + 2n + 3 > \lg(2) + d \quad \Leftrightarrow \quad \lg(2n+3) + 2n > d + \lg(2) - 3$

Die folgende Tabelle zeigt, wie groß der maximale Folgenindex n mindestens sein muss, damit d Dezimalen richtig sind (Voraussetzung $x < 0{,}1$) :

d	8	9	10	11	12	13	14	15	16	17	18	19	20	50
n	3	3	4	4	5	5	6	6	7	7	8	8	9	23

Fazit: Offensichtlich muss man (sicherheitshalber) für den Fall x < 1 **n = (int) (d / 2)** wählen .

Wie aber bekommt man die x-Werte unter die Grenze von 0,1 ??

Zur Transformation von |x| > 1 auf |x| < 0,1 verwendet man $\arctan(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right)$

In der Regel ist eine **mehrfache** Anwendung der Formel erforderlich. Zum Beispiel gilt

$$\arctan(5) = 2 \cdot \arctan\left(\frac{5}{1+\sqrt{1+5^2}}\right) = 2 \cdot \arctan(0{,}81...) = 4 \cdot \arctan(0{,}35...)$$

$$= 8 \cdot \arctan(0{,}17...) = 16 \cdot \arctan(0{,}08...)$$

Durch die abschließende Multiplikation mit dem Faktor 2^m (hier: 16) wird das Ergebnis ungenauer !

JAVA-Methode:

```
static double arctanReiheTaylorAusklammern(double x, int n) {
   // berechnet sum((-1)^k*x^(2k+1)/(2k+1),k,0,n) mit Argumentreduktion und Ausklammern
   if (x == 0) return 0;

   if (x < 0) return -arctanReiheTaylorAusklammern(-x, n);

   double faktor = 1.0;
   while (x > 0.1) { // Argumentreduktion
      x = x / (1 + Math.sqrt(1 + x * x));
      faktor = faktor * 2;
   }

   double quad = x*x;
   double sum = 0.0;
   for (int k = 2*n+1; k > 0; k -= 2)
      sum = 1.0 / k - quad * sum;

   return x * sum * faktor;
}
```

Beispielrechnung für arctan(5) mit Restgliedabschätzung:

x = 5 muss zuerst transformiert werden (siehe oben). arctan(5) = 16 · arctan(0.08604899056617473) .

Wir wollen arctan(5) auf 15 Dezimalen genau berechnen, d.h.

$\frac{0{,}08604899..^{2n+1}}{2n+1} \cdot \frac{1}{1+0} < 0{,}5 \cdot 10^{-15}$ Laut obiger Tabelle reicht n = 7 . Daher ist zu berechnen:

$$\arctan(5) \approx 16 \cdot \sum_{k=0}^{7} (-1)^k \cdot \frac{0{,}0840848515...^{2k+1}}{2k+1} = 1{,}373400766945016$$ 14 Dezimalen sind richtig

Für **große x** kann man auch auf $\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$ zurückgreifen, muss dann aber

d Dezimalen von π kennen; dies ist aber besonders bei Langzahlrechnung ungünstig !

Wir betrachten nochmals den Fall kleiner x-Werte (x < 0,5), weil diese zur Bestimmung von π mittels Taylorreihen wichtig sind.

Z.B. existiert eine berühmte Reihe von John MACHIN (1706), die lautet :

$$\pi = 16\cdot\arctan(1/5) \; - \; 4\cdot\arctan(1/239)$$

Interessant ist hier die Frage nach den minimal erforderlichen n-Werten der Polynome, um eine gegebene Anzahl von d Dezimalen richtig zu berechnen.

Für 16·arctan(1/5) ist der Ansatz nach der Restglied-Formel für alternierende Reihen:

$$16\cdot\frac{(\frac{1}{5})^{2n+3}}{2n+3} < 0{,}5\cdot 10^{-d} \;\Rightarrow\; (2n+3)\cdot 5^{2n+3} > 32\cdot 10^{d} \;\Rightarrow\; \lg(2n+3)+(2n+3)\cdot\lg(5) > \lg(32)+d$$

Alternative (Schätzung): n ≥ (d + lg(16/2)) / (2·lg(5)) – 0,5 ist jeweils in Klammern angegeben.

Für d = 100 Dezimalen benötigt man mindestens n = 70 (80).
Für d = 1000 Dezimalen benötigt man mindestens n = 713 (724).
Für d = 10000 Dezimalen benötigt man mindestens n = 7150 (7162).
Für d = 10^5 Dezimalen benötigt man mindestens n = 71530.
Für d = 10^6 Dezimalen benötigt man mindestens n = 715334 (715347)
.

Für 4·arctan(1/239) ergibt sich folgendes : $\lg(2n+3)+(2n+3)\cdot\lg(239) > \lg(8)+d$

Für d = 100 Dezimalen benötigt man mindestens n = 20 (23).
Für d = 1000 Dezimalen benötigt man mindestens n = 209 (212).
Für d = 10000 Dezimalen benötigt man mindestens n = 2101 (2104).
Für d = 10^5 Dezimalen benötigt man mindestens n = 21021.
Für d = 10^6 Dezimalen benötigt man mindestens n = 210224.

Dies ist gegenüber dem anderen Summanden eine drastische Zeiteinsparung !