1 BigDecimalMath

V tomto souboru řeším matematické funkce, které nejsou normálně dostupné pro java modul *BigDecimal*. Řešení těchto funkcí je vybráno tak, aby bylo co nejvíce nejpřesnější a jednoduše na provedení, ale zároveň aby čas vypočtení byl co nejmneší.

1.1 Konstanty a čísla

Součástí *BigDecimalMath* jsou i konstanty a jiná užitečná čísla, pro rychlejší a přehlednější programování. *BigDecimal* modul pro Javu již nějaké takovéto čísla obsahuje, například *BigDecimal.ZERO* a *BigDecimal.ONE*. Ovšem jiné variace, například pro číslo dva si musíme definovat vlastnoručně, proto jsem některá tato čísla definoval pro jednoduší použití v *BigDecimalMath*.

```java
import java.math.BigDecimal;
import java.math.RoundingMode;
// Minus jedna
public static final BigDecimal MINUSONE = new BigDecimal(-1);
// Dva
public static final BigDecimal TWO = new BigDecimal(2);
// Pi hodnota - napsana na 1000 desetinnych mist
public static final BigDecimal PI = new BigDecimal(
    "3.14159265358979323846264338327950288419716939 9..."
);
// E hodnota - napsana na 1000 desetinnych mist
public static final BigDecimal E = new BigDecimal(
    "2.71828182845904523536028747135266249775724709 3..."
);
// 2*Pi
pubic static final BigDecmal TWOPI = PI.multiply(TWO);
// Pi/2
public static final BigDecimal HALFPI = PI.divide(TWO, 1000,
     RoundingMode.HALF_UP);
// Minus Pi/2
public static final BigDecimal MINHALFPI =
     MINUSONE.multiply(HALFPI);
```

1.2 Trigonometrické a Hyperbolické funkce

Původní použití pro tento soubor bylo použití při implementaci complexních čísel a jejich funkcí do Java modulu *BigDecimal*. Použití modulu *Math* nebylo na místě, kvůli jeho omezení na *double* hodnoty. Z tohoto důvodu jsem musel napsat nové algoritmy na výpočty trigonometrických funkcí, které jsem potřeboval použít.

Tento script neobsahuje všechny trigonometrické či hyperbolické funkce, jen ty, které jsem potřeboval.

1.2.1 Sinus

Pro aproximaci sinu z čísla jsem použil taylorovu sérii, která vypadá takto:

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2x+1)!} x^{2x+1}.$$

Výsledný algoritmus vypadá tedy takto:

```java
public static BigDecimal sin(BigDecimal x) {
    BigDecimal ans = new BigDecimal(0);
    // Suma
    for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
        BigDecimal(50)) <= 0; n = n.add(BigDecimal.ONE)) {
        // Citatel
        BigDecimal numerator = MINUSONE.pow(n.intValue());
        // Jmenovatel
        BigDecimal denominator =
            factorial((TWO.multiply(n)).add(BigDecimal.ONE));
        // Vzorec uvnitr sumy
        ans = ans.add((numerator.divide(denominator, 1000,
            RoundingMode.HALF_UP)).multiply(
            x.pow(((TWO.multiply(n)).add(
            BigDecimal.ONE)).intValue())));
    }
    // Vraceni vysledku
    return ans.setScale(50, RoundingMode.HALF_UP);
}
```

Tento algoritmus je nastaven na přesnost okolo 1×10^{-50} s časem vypočítání okolo 10ms. Pro větší přesnost zvyšte *n.compareTo(new BigDecimal(50))* hodnotu na vyšší.

1.2.2 Cosinus

Pro aproximaci cosinu je opět použita taylorova série, která vypadá takto:

$$cosx = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2x}$$

Výsledný algoritmu vypadá tedy takto:

```java
public static BigDecimal cos(BigDecimal x) {
    BigDecimal ans = new BigDecimal(0);
    // Suma
    for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
        BigDecimal(50)) <= 0; n = n.add(BigDecimal.ONE)) {
        // Citatel
        BigDecimal numerator = MINUSONE.pow(n.intValue());
```

```
        // Jmenovatel
        BigDecimal denominator = factorial(TWO.multiply(n));
        // Vzorec uvnitr sumy
        ans = ans.add((numerator.divide(denominator, 1000,
            RoundingMode.HALF_UP)).multiply(
            x.pow((TWO.multiply(n)).intValue())));
    }
    // Vraceni vysledku
    return ans.setScale(50, RoundingMode.HALF_UP);
}
```

Tento algoritmus je nastaven na přesnost okolo 1×10^{-50} s časem vypočítání okolo 10ms. Pro větší přesnost zvyšte *n.compareTo(new BigDecimal(50))* hodnotu na vyšší.

1.2.3 Arctangens

Pro aproximaci arctangensu jsem opět použil taylorovy série, konkrétně jejich kombinaci, která bypadá takto:

$$arctan x = \begin{cases} \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} & : |x| \leq 1 \\ \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)x^{2n+1}} & : x \geq 1 \\ -\frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)x^{2n+1}} & : x \leq -1 \end{cases} \tag{1}$$

Výsledný algoritmus vypadá tedy takto:

```
public static BigDecimal arctan(BigDecimal x) {
    // Pokud je |x| <= 1
    if (x.abs().compareTo(BigDecimal.ONE) <= 0) {
        BigDecimal ans = new BigDecimal(0);
        // Suma
        for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
             BigDecimal(50)) <= 0; n = n.add(BigDecimal.ONE)) {
            // Citatel
            BigDecimal numerator = MINUSONE.pow(n.intValue());
            // Jmenovatel
            BigDecimal denominator =
                 (TWO.multiply(n)).add(BigDecimal.ONE);
            // Vzorec uvnitr sumy
            ans = ans.add((numerator.divide(denominator, 1000,
                RoundingMode.HALF_UP)).multiply(
                x.pow(((TWO.multiply(n)).add(
                BigDecimal.ONE)).intValue())));
```

```
            // Vraceni vysledku
        }
        return ans.setScale(50, RoundingMode.HALF_UP);
    // Jinak
    } else {
        BigDecimal ans = new BigDecimal(0);
        // Suma
        for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
             BigDecimal(50)) <= 0; n = n.add(BigDecimal.ONE)) {
            // Citatel
            BigDecimal numerator = MINUSONE.pow(n.intValue());
            // Jmenovatel
            BigDecimal denominator =
                ((TWO.multiply(n)).add(BigDecimal.ONE)).multiply(
                x.pow(((TWO.multiply(n)).add(
                BigDecimal.ONE)).intValue()));
            // Vzorec uvnitr sumy
            ans = ans.add((numerator.divide(denominator, 1000,
                 RoundingMode.HALF_UP)));
        }
        // Pokud je x > 1
        if (x.compareTo(BigDecimal.ONE) > 0) {
            return HALFPI.subtract(ans).setScale(50,
                 RoundingMode.HALF_UP);
        // Pokud je x < 1
        } else if (x.compareTo(MINUSONE) < 0) {
            return MINHALFPI.subtract(ans).setScale(50,
                 RoundingMode.HALF_UP);
        // Jinak vrati chybny vysledek
        } else {
            return ans.setScale(50, RoundingMode.HALF_UP);
        }
    }
}
```

Tento algoritmus je nastaven na přesnost okolo 1×10^{-50} s časem vypočítání okolo 10ms. Pro větší přesnost zvyšte *n.compareTo(new BigDecimal(50))* hodnotu na vyšší.

1.2.4 Hyperbolický sinus

Pro aproximaci hyperbolického sinusu jsem použil opět taylorovu sérii.

$$sinhx = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$$

Výsledný algoritmus vypadá tedy takto:

```
public static BigDecimal sinh(BigDecimal x) {
    BigDecimal ans = new BigDecimal(0);
    // Suma
    for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
         BigDecimal(50)) <= 0; n = n.add(BigDecimal.ONE)) {
        // Citatel
        BigDecimal numerator = x.pow((2*(n.intValue())+1));
        // Jmenovatel
        BigDecimal denominator =
            factorial((TWO.multiply(n)).add(BigDecimal.ONE));
        // Vzorec uvnitr sumy
        ans = ans.add(numerator.divide(denominator, 1000,
            RoundingMode.HALF_UP));
    }
    // Vraceni vysledku
    return ans.setScale(50, RoundingMode.HALF_UP);
}
```

Tento algoritmus je nastaven na přesnost okolo 1×10^{-50} s časem vypočítání okolo 10ms. Pro větší přesnost zvyšte *n.compareTo(new BigDecimal(50))* hodnotu na vyšší.

1.2.5 Hyperbolický cosinus

Pro aproximaci hyperbolického cosinu je opět užita taylorova série.

$$coshx = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$$

Výsledný algoritmus vypadá tedy takto:

```
public static BigDecimal cosh(BigDecimal x) {
    BigDecimal ans = new BigDecimal(0);
    // Suma
    for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
         BigDecimal(50)) <= 0; n = n.add(BigDecimal.ONE)) {
        // Citatel
        BigDecimal numerator = x.pow(2*(n.intValue()));
        // Jmenovatel
        BigDecimal denominator = factorial(TWO.multiply(n));
        // Vzorec uvnitr sumy
        ans = ans.add(numerator.divide(denominator, 1000,
            RoundingMode.HALF_UP));
    }
    // Vraceni vysledku
    return ans.setScale(50, RoundingMode.HALF_UP);
}
```

Tento algoritmus je nastaven na přesnost okolo 1×10^{-50} s časem vypočítání okolo 10ms. Pro větší přesnost zvyšte *n.compareTo(new BigDecimal(50))* hodnotu na vyšší.

1.3 Exponenciální funkce a přirozený logaritmus

1.3.1 Exponenciální funkce

Exponenciální funkce ve tvaru $exp(x) = e^x$ je vypočítaná taylorovou sérií, která vypadá takto:

$$exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Algoritmus pro vypočítaní vypadá tedy takto:

```
public static BigDecimal exp(BigDecimal z) {
    BigDecimal ans = new BigDecimal(0);
    // Suma
    for (BigDecimal n = BigDecimal.ZERO; n.compareTo(new
        BigDecimal(150)) <= 0; n = n.add(BigDecimal.ONE)) {
        ans = ans.add((z.pow(n.intValue())).divide(factorial(n),
            50, RoundingMode.HALF_UP));
    }
    // Navraceni vysledku
    return ans;
}
```

Tento algoritmus je nastaven na přesnost okolo 1×10^{-50} s časem vypočítání okolo 15ms. Pro větší přesnost zvyšte *n.compareTo(new BigDecimal(50))* hodnotu na vyšší.

1.3.2 Přirozený logaritmus

Algorimizace přirozeného logarimu je složitá. Integrální vzorec je časově velice neefektivní, tudíž jsem se rozhodl použít lehce upravený Newtonův algoritmus. Vypadá takto:

$$log(x) = \sum^{\infty} x - \frac{e^x - x}{e^x}$$

1.4 Zdroje

Zdroj pro aproximace trigonometrických funkcí:
    Wikipedia: Taylor series
    Wikiproof: Power series expansion for real arctangnet function
Zdroj pro aproximace hyperbolických funkcí:
    Wikipedia: Taylor series
Zdroj pro exponenciální funkci:

Wikipedia: Taylor series