2 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.

2.1 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*.

```
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(
    "2.718281828459045235360287471352662497757247093..."
);
// 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);
```

2.2 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.

2.2.1 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:

```
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.

2.2.2 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:

```
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.

2.2.3 Arctangens

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

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

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.

2.2.4 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.

2.2.5 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.

2.3 Exponenciální funkce a přirozený logaritmus

2.3.1 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.

2.3.2 Přirozený logaritmus

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

$$log(x) = (a_{n-1} - \tfrac{e^{a_{n-1}} - n}{e^{a_{n-1}}})_{n=1}^{\infty}; \text{ kde } a_0 = x$$

Jeho výsledný algoritmus vypadá takto:

```
public static BigDecimal log(BigDecimal x) {
    BigDecimal n = x; BigDecimal term;
    // Pokud x neni v definicnim oboru
    if (x.compareTo(BigDecimal.ZERO) <= 0) {
        return null;
    }
```

```java
        // Vypocet posloupnosti
        for (int i = 1; i<=20; i++) {
            BigDecimal eToX = exp(x);
            term = eToX.subtract(n).divide(eToX, 50,
                 RoundingMode.HALF_UP);
            x = x.subtract(term);
        }
        // Vraceni vysledku
        return x.setScale(50, RoundingMode.HALF_UP);
    }
```

Výsledek má přesnost okolo $1x10^(- 50)$ a čas výpočtu je asi 500ms. Tento algoritmus není nejrychlejší, ale je velice přesný - pro větší přesnost musíte změnit $i \leq 20$ na větší číslo. Rychlost se tak sice zhorší, ale extrémně naroste přesnost. Momentální nastavení na *20* je přepal, ale pro větší vstupy je potřeba větší přesnost.

2.4 Ostatní funkce

Pro některé algoritmy jsem potřeboval funkce, které nejsou tak důležité, aby měli svou vlastní sekci, nebo jich není mnoho.

2.4.1 Faktoriál

Výpočet tohoto faktoriálu je pouze pro přirozený čísla, není naprogramována gamma funkce, která by toto zgeneralizovala. Ovšem pro užití v taylorových seriích je potřeba pouze faktoriál pro přirozená čísla. Jeho předpis je tedy jednoduchý:

$$n! = x \times (x-1) \times (x-2) \times ... \times 2 \times 1 = x \times (n-1)!$$
$$0! = 1$$

Výsledný algortimus vypadá takto:

```java
    public static BigDecimal factorial(BigDecimal n) {
        BigDecimal ans = n;
        // Pokud je vstup 0, vrati hodnotu 1
        if (n.compareTo(BigDecimal.ZERO) == 0) {
            return BigDecimal.ONE;
        }
        // Pokud je vstup zaporne cislo, vrati chybnou hodnotu
        if (n.compareTo(BigDecimal.ZERO) < 0) {
            return null;
        }
        // Jinak se provede cyklus pro vypocet factorialu
        for (BigDecimal k = new BigDecimal(1); k.compareTo(n) < 0; k
             = k.add(BigDecimal.ONE)) {
            ans = ans.multiply(n.subtract(k));
        }
```

```
        // Vrati hodnotu
        return ans;
    }
```

Tento algoritmus je velice rychlý - nemělo smysl měřit jeho rychlost. Navíc se jeho rychlost zpomaluje s větším číslem.

2.4.2 Sign funkce

Tato funkce je velice jednoduchá:

$$sgnx = \begin{cases} -1 & : x < 0 \\ 0 & : x = 0 \\ 1 & : x > 0 \end{cases}$$

Její implementace je taktéž jednoduchá, ikdyž vypadá složitě.

```
    public static BigDecimal sign(BigDecimal x) {
        // Pokud je vstup mensi jak 0
        if (x.compareTo(BigDecimal.ZERO) < 0) {
            return new BigDecimal(-1);
        // Pokud je vstup roven 0
        } else if (x.compareTo(BigDecimal.ZERO) == 0) {
            return BigDecimal.ZERO;
        // Pokud je vstup vetsi jak 0
        } else {
            return BigDecimal.ONE;
        }
    }
```

3 ComplexNumber

V tomto souboru je vyřešena implementace kompexních čísel a jejich základních funkcí do Javy. Jako základní funkce se rozumí jejich operace, absolutní hodnota, sgn funkce a trigonometrické s hyperbolickými funkcemi. Dále tento soubor obsahuje i základní vypsání funkcí, statické hodnoty a jejich konstruktory.

3.1 Konstrukory, display funkce a převody.

V základu tu jsou definované 4 otevřené proměnné, které jsou používány ve funkcích. Převod mezi nimi je ukázán dále v tomto dokumentu.

```
public BigDecimal REAL; // Realna hodnota komplexniho cisla
public BigDecimal IMG; // Imaginarni hodnota komplexniho cisla
public BigDecimal R; // Polomer komplexniho cisla v
    goniometrickem tvaru
public BigDecimal PHI; // Uhel ke komplexnimu cislu v
    goniometrickem tvaru
```

3.1.1 Konstruktory

Je zde definovaných několik konstruktorů, které vytvoří toto komplexní číslo. Prvním z nich je tzv. prázdný konstruktor, který nastaví hodnotu $0 + 0i$.

```
public ComplexNumber() {
    REAL = BigDecimal.ZERO; IMG = BigDecimal.ZERO;
}
```

Dále jsou konstruktory, kteří přijímají jakékoliv číselné hodnoty po dvojicích. Pordporované hodnoty jsou *int, double, ,float, BigInteger a BigDecimal.* Příkladný konstruktor tedy vypadá takto:

```
public ComplexNumber(double a, double b) {
    REAL = new BigDecimal(a); IMG = new BigDecimal(b);
}
```

Vrátí tedy hodnotu $a + bi$, kde a i b jsou *double* hodnoty. Tyto konstruktory jsou nastaveny na dvojice - to znamená že musíte do konstruktoru vložit dvě hodnoty stejného typu.
Jejich zavolání vypadá následovně:

```
ComplexNumber a = new ComplexNumber(); // Prazdny konstruktor
ComplexNumber b = new ComplexNumber(1, 2); // Konstruktor o
    hodnote "1+2i"
```

3.1.2 Display funkce

Display funkce vypíše hezky formátovaně komplexní tvar čísel. Jsou nastaveny na podmínky, aby dokázali měnit znaménko mezi reálnou a komplexní hodnotou, jinak by se čísla buď na sebe nalepila či by se vypsaly dvě znaménka vedle sebe. Případně vypíší reálnou či imaginární část, pokud jsou rovny nule.

```java
public void display() {
    // Pokud je realna cast rovna 0, vypise pouze imaginarni cast
    if (REAL.compareTo(BigDecimal.ZERO) == 0) {
        System.out.println(
            IMG.toString()+"i"
        );
        return;
    // Pokud je imaginarni cas rovna 0, vypise pouze realnou cast
    } else if (IMG.compareTo(BigDecimal.ZERO) == 0) {
        System.out.println(
            REAL.toString()
        );
        return;
    }
    // Zmeni znamenko podle velikosti imaginarni casti
    if (IMG.compareTo(BigDecimal.ZERO) < 0) {
        System.out.println(
            REAL.toString()+"-"+IMG.abs().toString()+"i"
        );
        return;
    } else {
        System.out.println(
            REAL.toString()+"+"+IMG.toString()+"i"
        );
        return;
    }
}
```

Zavolání těchto funkcí je tedy následovné:

```java
ComplexNumber a = new ComplexNumber(5, 4);
a.display();
// 5+4i
```

3.1.3 Převod do goniometrického tvaru

Goniomtrický tvar se skládá z poloměru r a úhlu ϕ ke komplexnímu číslu $z = x + yi$. Výpočet poloměru je následovný:

$$r = |z| = \sqrt{x^2 + y^2}$$

A výpočet úhlu ϕ je:

$$\phi = arg(z) = \begin{cases} 2arctan(\frac{y}{\sqrt{x^2+y^2}+x}) & \text{if } y \neq 0 \\ \pi & \text{if } x < 0 \text{ and } y = 0 \\ 0 & \text{if } y = 0 \text{ and } x > 0 \\ undefined & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

Tento tvar je použit při výpočtu určitých operacích, napřiklad při exponenciální funkce.
Algoritmus na vytvoření těchto proměnných je následovný:

```java
public void polar_conversion() {
    // Polomer
    R = (REAL.pow(2).add(IMG.pow(2))).sqrt(new MathContext(50));
    // Fi hodnota
    // Treti podminka z predesleho vzorce
    if (REAL.compareTo(BigDecimal.ZERO) != 0 &&
         IMG.compareTo(BigDecimal.ZERO) == 0) {
        PHI = BigDecimal.ZERO;
    // Prvni podminka
    } else if (REAL.compareTo(BigDecimal.ZERO) > 0 ||
         IMG.compareTo(BigDecimal.ZERO) != 0) {
        BigDecimal TWO = new BigDecimal(2);
        PHI = (
            TWO.multiply(BigDecimalMath.arctan(
            IMG.divide(R.add(REAL), 50, RoundingMode.HALF_UP)))
        );
        PHI = PHI.setScale(50, RoundingMode.HALF_UP);
    // Druha podminka
    } else if (REAL.compareTo(BigDecimal.ZERO) < 0 &&
         IMG.compareTo(BigDecimal.ZERO) == 0) {
        PHI = new BigDecimal(Math.PI);
        PHI = PHI.setScale(50, RoundingMode.HALF_UP);
    // Posledni podminka
    } else {
        PHI = null;
    }
}
```

Tyto hodnoty nejsou vytvořeny vždy, ale ve funkcích, kde jsou potřeba se tato funkce sama zavolá a hodnoty se vypořítají.

3.2 Konstanty a čísla

Opět, jako u *BigDecimalMath* jsou zde vytvořeny nějaké základní konstanty a čísla, pro jednodušší použití.

```java
// Nulova hodnota
public static final ComplexNumber ZERO = new ComplexNumber();
// Hodnota pro cislo jedna
public static final ComplexNumber ONE = new ComplexNumber(1, 0);
// Hodnota pro cislo minus jedna
public static final ComplexNumber MINUSONE = new
    ComplexNumber(-1, 0);
// Hodnota pro cislo I
public static final ComplexNumber I = new ComplexNumber(0, 1);
// Hodnota pro cislo 0.5I
public static final ComplexNumber HALFI = new ComplexNumber(0,
    0.5);
// Hodnota pro cislo Pi/2
public static final ComplexNumber HALFPI = new
    ComplexNumber(BigDecimalMath.PI.divide(BigDecimalMath.TWO,
    1000, RoundingMode.HALF_UP));
```

Opět některé tyto hodnoty mohou vypadat nesmyslně, proč jsou definovány, ale jsou potřebovat v některých algoritmech.

3.3 Základní operace

Definujme si 2 komplexní čísla. Číslo $c = a + bi$ a $z = x + yi$. Výsledek budu označovat za číslo α

3.3.1 Sčítání a odčítání

Tyto operace jsou velice jednoduché a intuitivní. Operace sčítání:

$$\alpha = c + z = (a + bi) + (x + yi) = (a + x) + (b + y)i$$

```java
public ComplexNumber add(ComplexNumber b) {
    ComplexNumber ans = new ComplexNumber();
    ans.REAL = REAL.add(b.REAL);
    ans.IMG = IMG.add(b.IMG);
    return ans;
}
```

Operace odčítání:

$$\alpha = c - z = (a + bi) - (x + yi) = (a - x) + (b - y)i$$

```java
public ComplexNumber subtract(ComplexNumber b) {
    ComplexNumber ans = new ComplexNumber();
    ans.REAL = REAL.subtract(b.REAL);
    ans.IMG = IMG.subtract(b.IMG);
    return ans;
}
```

3.3.2 Násobení

Tato operace je opět velice jednoduchá:

$$\alpha = c \times z = (a + bi) \times (x + yi) = (ax - by) + (ay + bx)i$$

```
public ComplexNumber multiply(ComplexNumber b) {
    ComplexNumber ans = new ComplexNumber();
    ans.REAL =
        (REAL.multiply(b.REAL)).subtract(IMG.multiply(b.IMG));
    ans.IMG = (REAL.multiply(b.IMG)).add(IMG.multiply(b.REAL));
    return ans;
}
```

3.3.3 Dělení

Tato operace je trochu složitější:

$$\frac{c}{z} = c \times \frac{1}{z} = \frac{(ax+by)+(bx-ay)i}{x^2+y^2}$$

Její algoritmus:

```
public ComplexNumber divide(ComplexNumber b) {
    ComplexNumber ans = new ComplexNumber();
    // Pokud je jmenovatel roven 0, vrati chybnou hodnotu
    if (b.REAL.compareTo(BigDecimal.ZERO) == 0 &&
        b.IMG.compareTo(BigDecimal.ZERO) == 0) {
        return null;
    }
    // Vlastni algoritmus
    ans.REAL =
        ((REAL.multiply(b.REAL)).add(IMG.multiply(b.IMG))).divide(
        (b.REAL.pow(2)).add(b.IMG.pow(2)), 50,
        RoundingMode.HALF_UP);
    ans.IMG =
        ((IMG.multiply(b.REAL)).subtract(REAL.multiply(b.IMG))
        ).divide((b.REAL.pow(2)).add(b.IMG.pow(2)), 50,
        RoundingMode.HALF_UP);
    // Vraceni vysledku
    return ans;
}
```