

Wichterlovo gymnázium, Ostrava-Poruba, příspěvková organizace

# KNIHOVNA PRO PRÁCI S REÁLNÝMI ČÍSLY

Maturitní práce z Informatiky a výpočetní techniky

Jiří Škrobánek

Vedoucí práce  
Mgr. Rostislav Fojtík, Ph.D.

15. dubna 2018  
Ostrava

# PROHLÁŠENÍ

Prohlašuji, že jsem tuto práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

V Ostravě 15. dubna 2018

---

# ABSTRAKT

Tato práce se zabývá počítačovou aritmetikou. Byla vytvořena knihovna pro reprezentaci čísel ve tvaru zlomku. Využití racionálních čísel není v informatice příliš časté, nejsou ani obsaženy ve většině základních knihoven programovacích jazyků. (C/C++<sup>1</sup>, C#, Java, Javascript, atd.) Přesto existují situace, kde je vhodné racionální číselné typy použít. Zejména v případě, kdy potřebujeme přesný výsledek, který nedokážeme získat výpočtem s číslem s plovoucí řádovou čárkou.

Knihovna implementuje racionální čísla jako strukturu **Rational** a pro výpočty s libovolnou přesností jako strukturu **BigRational**. Zároveň je ponechána syntax aritmetických operací, provádějí se tedy stejně, jako by proměnné byly celá čísla. Obsaženy jsou i obecné metody s nejčastějším využitím.

# ABSTRACT

This maturita thesis concerns with computer arithmetic. A library for representation of numbers in the form of fractions was created. The use of rational numbers is not very common in informatics. Programming languages do not predominantly include them in standard libraries. (C/C++<sup>2</sup>, C#, Java, Javascript, atd.) Despite that, situations exist where using rational numeric types is viable. In particular those where we need precise results, which cannot be reached with floating point arithmetic. The library implements rational numbers as **Rational** structure and with arbitrary precision as **BigRational** structure. The syntax of arithmetic operations remains nearly the same as with integral variables. General methods with frequent use are also included.

---

<sup>1</sup>V C je možné při překladu zdrojového kódu vyhodnotit zlomek.

<sup>2</sup>In C language it is possible to evaluate fractions at compile-time.

# OBSAH

<b>Obsah</b>	<b>4</b>
<b>1 Úvod</b>	<b>5</b>
1.1 Obecný úvod . . . . .	5
1.2 Konkrétní problém . . . . .	5
<b>2 Reprezentace čísel</b>	<b>7</b>
2.1 Celočíselné datové typy . . . . .	7
2.2 Typy s pevnou řádovou čárkou . . . . .	7
2.3 Typy s plovoucí řádovou čárkou . . . . .	8
2.4 Desetinné typy . . . . .	8
2.5 Racionální datové typy . . . . .	8
<b>3 Číselné typy s plovoucí řádovou čárkou</b>	<b>9</b>
3.1 Rozsah . . . . .	9
3.2 Přesnost s ohledem na výkon . . . . .	9
3.3 Problémy . . . . .	9
<b>4 Racionální datové typy</b>	<b>12</b>
4.1 Způsob uložení . . . . .	12
4.2 Formy racionálních čísel . . . . .	12
<b>5 Popis knihovny racionální aritmetiky pro .NET Core</b>	<b>14</b>
5.1 Požadavky na knihovnu . . . . .	14
5.2 Ústřední obsah knihovny . . . . .	15
5.3 Speciální hodnoty . . . . .	16
5.4 Vybraná funkčnost a algoritmy . . . . .	16
5.5 BigRational . . . . .	21
5.6 Vyhodnocování výrazů . . . . .	21
<b>6 Základní přehled knihovny RationalTypes</b>	<b>24</b>
<b>7 Závěr</b>	<b>26</b>
<b>Seznam obrázků</b>	<b>27</b>
<b>Seznam tabulek</b>	<b>27</b>
<b>Seznam příloh</b>	<b>27</b>
<b>Literatura</b>	<b>28</b>

# Kapitola 1

## ÚVOD

### 1.1 Obecný úvod

V současnosti je situace taková, že programátoři v drtivé většině případů přistupují k necelým číslům jako k reálným číslům. To vede k tomu, že je implementují vždy jako čísla s plovoucí řádovou čárkou. Přestože jsou tyto typy univerzální, na některá využití se příliš nehodí. (To je zpravidla z důvodu nepřesnosti, nebo úzkého rozsahu hodnot.) Toto je dáno především tím, že jiné přístupy v programovacích jazycích v jejich základních knihovnách nejsou obsaženy.

Vytvořená knihovna `RationalTypes` přináší možnost racionální aritmetiky do jazyka C# a na platformu .NET Core. Ze způsobů implementace byl zvolen zlomkový tvar před jinými alternativami. Operace s racionálními čísly uzavřené na racionální čísla nejsou funkce, ale provádí se normálními operátory.

Další součástí knihovny jsou třídy `Polynomial` a `RationalPolynomial`, které slouží k práci s polynomy (s racionálními koeficienty). Knihovna dokáže najít racionální kořeny takových polynomů. Jednou možností použití tedy je hledání řešení rovnic.

Ovšem dost způsobů užití racionálních čísel je velmi specifických, jak vidíme na následujícím příkladu.

### 1.2 Konkrétní problém

Máme nějaké množství úseček spojujících mřížové body. Dále máme přímku  $p$ . Tyto úsečky potřebujeme seřadit podle (orientovaného) úhlu, který svírají s  $p$ . Jako základní řešení bychom mohli například spočítat odchylku od osy ve zvoleném souřadném systému a tyto odchylky porovnávat.

To bychom nejspíše dělali pomocí funkce nějaké cyklotrické funkce<sup>1</sup>, jenže ty jsou jak hodně pomalé, tak celkem nepřesné. Mohlo by se tedy stát, že pořadí některých úseček bude obrácené, pokud budou mít mezi sebou velmi podobný úhel. K porovnání směrnic dvou přímek můžeme dále použít vektorový součin.

Nejlepší pravděpodobně však bude v souřadném systému spočítat rozdíl v hodnotách souřadnic pro krajní body úseček. Tímto získáme směrnici úsečky:  $k = (A_y - B_y) : (A_x - B_x)$ , které bude racionální číslo. Pak jen stačí umět racionální čísla porovnávat.

Zajímavé také je, že i všechny průsečíky úseček s krajními body v mřížových bodech mají racionální souřadnice, na které také můžeme použít racionální čísla. Dokonce i když použijeme tyto průsečíky jako krajní body nových úseček, jejich průsečíky stále budou mít racionální souřadnice.

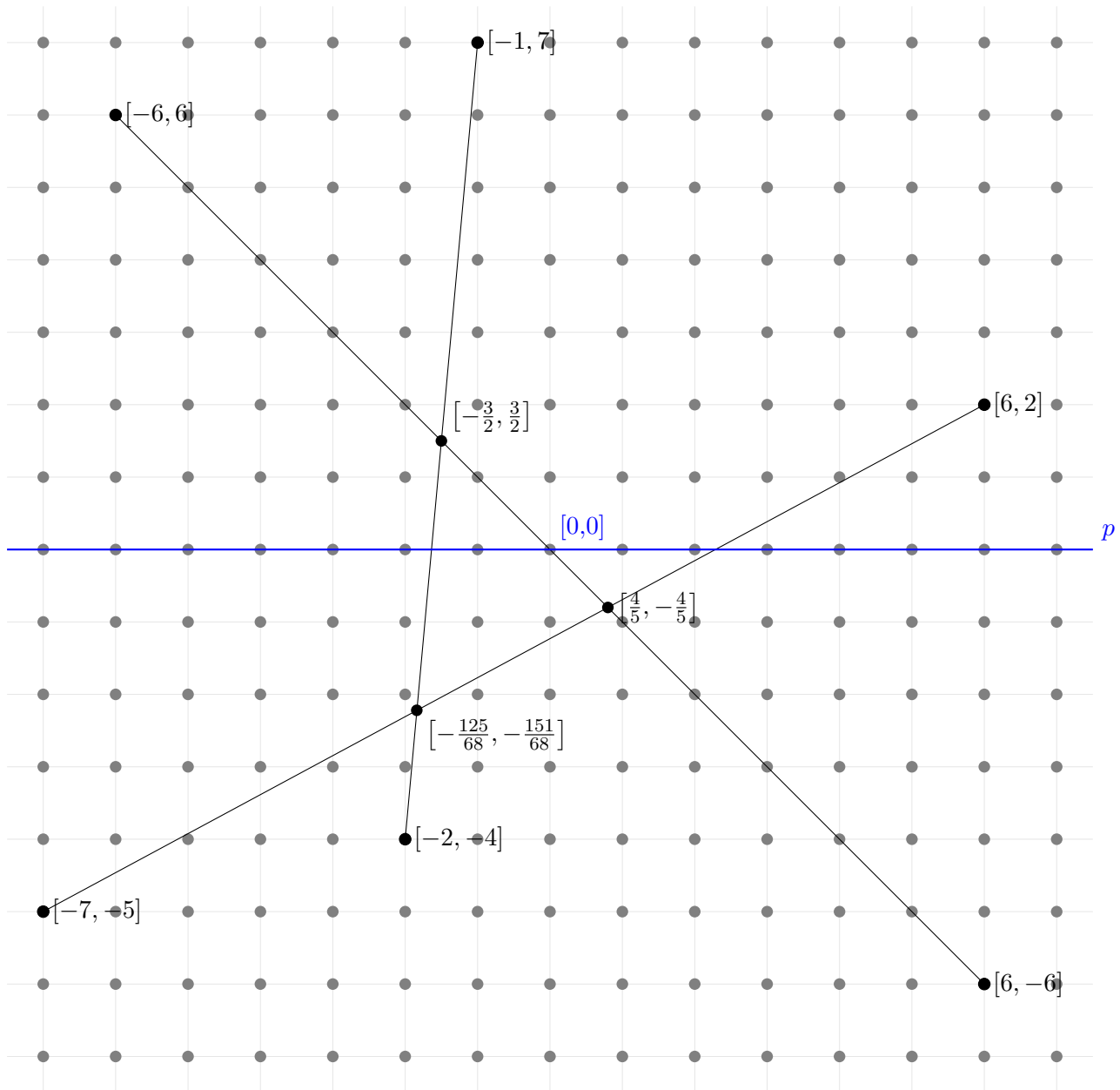
Důkaz: Pokud mají přímky stejnou ypsilonovou souřadnici pro  $x = 0$ , máme jejich průsečík. Jinak mějme obecné rovnice dvou přímek nerovnoběžných s osou  $x$ , kde koeficienty jsou racionální čísla:  $x + d_1y + c_1 = 0$  a  $x + d_2y + c_2 = 0$ , jejich průsečík, jestliže se protínají v jednom bodě:

$$[(d_1c_2 - d_2c_1) : (d_2 - d_1), (d_1 - d_2) : (c_2 - c_1)]$$

Průsečík má opět racionální souřadnice. Vynechaný případ nejméně jedné přímky rovnoběžné s osou  $x$  vyřešíme bez dalších výpočtů tak, že na chvíli řekneme, že osy  $x$  a  $y$  jsou označeny opačně. Ještě zbývá, že je jedna přímka rovnoběžná s  $x$  a druhá s  $y$ . Pak je ale zřejmé, že jejich průsečík je mřížový bod.

---

<sup>1</sup>V C `<math.h>` je to například `atan(double, double)` a `atan2(double, double)`.



Obrázek 1.1: Úsečky a mřížové body

Povšimněme si, že průsečíky mají racionální souřadnice.

# Kapitola 2

## REPREZENTACE ČÍSEL

V počítači můžeme čísla reprezentovat různými způsoby. Podle požadavků na přesnost, rozsah hodnot, operace a také podle výkonu, který máme k dispozici, zvolíme adekvátní datový typ. Následuje krátký přehled způsobů.

### 2.1 Celočíselné datové typy

Potřeba aritmetiky s celými čísly je bezpodmínečně nutná už jen pro správné adresování paměti počítače. Všechny instrukční sady proto podporují nějaký celočíselný typ. Většina operací probíhá na proměnných tohoto typu. Všechny operace na celých číslech fungují velmi rychle. Budeme je používat k implementaci dalších čísel.

Speciálním celočíselným typem jsou celá čísla s libovolným rozsahem, omezená pouze pamětí počítače.

### 2.2 Typy s pevnou řádovou čárkou

Základem pro tyto typy je nějaký celočíselný typ. Část binární reprezentace ovšem tvoří necelou část čísla. Tento typ se prakticky nevyužívá a programovací jazyky většinou ani nenabízejí v základních knihovnách práci s takovou proměnnou. V zásadě platí, že lze použít k reprezentaci celočíselný datový typ a když to operace žádá provést bitový posun.

Snadno můžeme v běžných jazycích, pokud použijeme na celou i necelou část 64bitovou proměnnou, kde využijeme pouze 32 bitů z každé. Součin  $ab$  vypočteme:

$$\lfloor ab \rfloor = (\lfloor a \rfloor \times \langle b \rangle + \lfloor b \rfloor \times \langle a \rangle + ((\lfloor a \rfloor \times \lfloor b \rfloor) \ll 32)) \gg 32$$

$$\langle ab \rangle = (((\lfloor a \rfloor \times \langle b \rangle + \lfloor b \rfloor \times \langle a \rangle) \ll 32) + \langle a \rangle \times \langle b \rangle) \gg 32$$

Kde  $\langle n \rangle$  značí necelou část  $n$ ,  $\lfloor n \rfloor$  značí dolní celou část  $n$  a  $\gg$  je operátor logického posunu doprava. Mnohem efektivnější ovšem bude implementovat takové operace přímo v příslušném assembly kódu. Například na ARMu můžeme násobení provést na pouhých 11 instrukcích.

Číslo se uloží jako souvislých 64 bitů, prvních 32 bitů bude celá část, dalších 32 bitů bude necelá část.

```
LDM r0, {r0, r1}
```

```
LDM r2, {r2, r3}
```

```
UMULL r4, r5, r1, r2
```

```
UMULL r6, r7, r0, r3
```

```
ADDS r5, r5, r7
```

```
ADC r4, r4, r6
```

```
UMULL r6, r7, r1, r3
```

```
ADDS r5, r5, r6
```

```
ADC r4, r4, #0
```

```
MLA r4, r0, r2, r4
```

```
STM r10, {r4, r5}
```

Načteme 64 bitů z adresy v registru r0.

Načteme 64 bitů z adresy v registru r2.

Vynásobíme celou a necelou část.

Stejně pro druhou dvojici.

Sečteme necelé části ze součinů.

Sečteme celé části ze součinů.

Vynásobíme necelé části výsledku,

přičteme k necelé části,

přičteme příznak přenosu k celé části výsledku.

Vynásobíme a přičteme celé části k výsledku.

Výsledek uložíme na adresu v r10.

Použití tohoto typu má smysl, pokud očekáváme, že budou všechny hodnoty v určitém rozsahu (v lepším případě dále od 0). Vhodné by například mohlo být ukládat takto výměru pozemku, nečekáme totiž extrémně malé, ani velké hodnoty.

## 2.3 Typy s plovoucí řádovou čárkou

Říkejme jim dále `float`. Potřeba využít `float` nastává pokud musíme pokrýt obrovský rozsah hodnot. Přitom nesmí vadit o kolik jednotek se výpočet odchýlí, ale o kolik procent se odchýlí. Zároveň je `float` univerzální tak, že prakticky ke všem výpočtům můžeme `float` použít. Mnohdy je to však neefektivní z hlediska výkonu či výsledné přesnosti.

## 2.4 Desetinné typy

Desetinné typy nejsou příliš často používané. Vznikly kvůli tomu, že určité relativně běžné číselné hodnoty nedokážeme reprezentovat přesně v binární soustavě, i když mají v desítkové soustavě konečný desetinný rozvoj. Příkladem může být třeba 0,1.

$$0,1 = 0,0001100110011\overline{0011}_2$$

Při operacích s financemi je podstatné, aby nedocházelo k chybám, právě zde se ovšem vyskytují necelé části, které neumíme pomocí floatu reprezentovat přesně.

## 2.5 Racionální datové typy

Těmito se budeme především zabývat. Nemají žádnou hardwarovou implementaci, proto interně využívají celá čísla.



## Kapitola 3

# ČÍSELNÉ TYPY S PLOVOUCÍ ŘÁDOVOU ČÁRKOU

Zaměříme se především na nepřesnosti vznikající v souvislosti s používáním těchto typů.

### 3.1 Rozsah

Název	v C	Velikost	Znaménko	Exponent	Mantisa	$\approx$ Cifer
binary16		16b	1b	5b	10b	3
binary32	float	32b	1b	8b	23b	7
binary64	double	64b	1b	11b	52b	16
binary128	<i>long double</i>	128b	1b	15b	112b	34
binary256		256b	1b	19b	236b	71
extended	<i>long double</i>	80b	1b	15b	63b	19

Od long double je požadováno, aby byl alespoň tak přesný jako double, ale přesná implementace se liší podle hardwaru.

Tabulka 3.1: Standard IEEE 754

### 3.2 Přesnost s ohledem na výkon

V souladu s tím, co je logické očekávat, použití vyšší přesnosti znamená nižší výkon. Složitější výpočty se zpravidla provádí na grafických procesorech, neboť mají řádově vyšší výkon než CPU. Rozdíl ve výkonu je patrný taktéž u vysoce výkonných čipů. 64bitová přesnost se sebou nese nejméně šestnáctkrát nižší výkon u běžných GPU než 32bitová přesnost. Nvidia Quadro je pak speciální řada pro datová centra.

### 3.3 Problémy

Hlavními problémy jsou přetečení a podtečení. Ukážeme ale problém, který se racionální proměnnou odstraní.

Model	Vydání	Výkon GFLOPS			Cena
		half	single	double	
AMD Radeon RX Vega 64	14. 7. 2017	20431	10215	638	\$490
Nvidia TITAN Xp	6. 4. 2017	158	10790	337	\$1200
Nvidia Quadro GP100	6. 2. 2017		9519	5300	\$6500
Intel Core i7-8700K	Q4 2017	-	61	32	\$359

Pozn. Pro srovnání model procesoru pro stolní počítače.

Tabulka 3.2: Výkon grafických procesorů

### Postupná ztráta přesnosti

Hodnoty, které `double` dokáže přesně uložit, můžeme zapsat jako:

$$a \cdot 2^e, a \in \langle -2^{53}, 2^{53} \rangle, e \in \langle -1024, 1023 \rangle$$

Uvažme jednoduchou funkci

$$f(x) = \frac{kx}{l} - 1 = \frac{kx - l}{l}$$

jejími pevnými body (tzn.  $f(x_c) = x_c$ ) jsou takováto  $x_c$ :

$$x_c = \frac{l}{k - l}$$

Pro konkrétní hodnoty  $l = 2, k = 7 : x_c = 0, 4$  implementujeme v jazyce C.

```
void iteruj() {
    int iter = 0; double state = 0.4;
    while(iter++ < VAL) {
        state *= 7.0; state /= 2.0; state -= 1.0;
        printf("%d: %.8f", iter, state);
    }
}
```

### Při použití 32bitové proměnné float:

1: 0.39999998	6: 0.39997780	11: 0.38835073	16: -5.71842003
2: 0.39999986	7: 0.39992237	12: 0.35922754	17: -21.01446915
3: 0.39999950	8: 0.39972830	13: 0.25729632	18: -74.55064392
4: 0.39999819	9: 0.39904904	14: -0.09946287	19: -261.92724609
5: 0.39999366	10: 0.39667165	15: -1.34811997	20: -917.74536133

### Při použití 64bitové proměnné double:

1: 0.400000000	8: 0.400000000	15: 0.400000000	22: 0.40002698
2: 0.400000000	9: 0.400000000	16: 0.400000001	23: 0.40009444
3: 0.400000000	10: 0.400000000	17: 0.400000005	24: 0.40033055
4: 0.400000000	11: 0.400000000	18: 0.400000018	25: 0.40115691
5: 0.400000000	12: 0.400000000	19: 0.400000063	26: 0.40404919
6: 0.400000000	13: 0.400000000	20: 0.40000220	27: 0.41417216
7: 0.400000000	14: 0.400000000	21: 0.40000771	28: 0.44960255

29: 0.57360894	31: 2.52670948	33: 26.45219109	35: 319.53934082
30: 1.00763128	32: 7.84348317	34: 91.58266881	36: 1117.38769287

**Při použití konkrétní implementace long double:**

Zřejmě se jedná o 80bitovou přesnost.

1: 0.40000000	11: 0.40000000	21: 0.40000000	31: 0.39919377
2: 0.40000000	12: 0.40000000	22: 0.39999999	32: 0.39717819
3: 0.40000000	13: 0.40000000	23: 0.39999996	33: 0.39012366
4: 0.40000000	14: 0.40000000	24: 0.39999987	34: 0.36543280
5: 0.40000000	15: 0.40000000	25: 0.39999956	35: 0.27901479
6: 0.40000000	16: 0.40000000	26: 0.39999846	36: -0.02344825
7: 0.40000000	17: 0.40000000	27: 0.39999463	37: -1.08206886
8: 0.40000000	18: 0.40000000	28: 0.39998120	38: -4.78724102
9: 0.40000000	19: 0.40000000	29: 0.39993419	39: -17.75534357
10: 0.40000000	20: 0.40000000	30: 0.39976965	40: -63.14370248

Vidíme, že po nepříliš vysokém počtu iterací už proměnná nabývá zcela odlišných hodnot, než by měla a než můžeme použít dále. `float` totiž v podstatě reprezentuje určitý interval čísel. Tento interval se ale s každou provedenou operací zvyšuje, až do bodu, kdy je proměnná zcela nepoužitelná. Toto se příliš nezmění ani pokud využijeme nejlepší dostupnou přesnost v C.

Vytvořená knihovna tuto nepřesnost zcela odstraňuje, po libovolném počtu iterací stále výsledek bude 0,4. Hodnoty totiž reprezentují pouze bod na číselné ose, nikoliv interval.

Hardwarová implementace ovšem způsobuje, že přesný výpočet je mnohem pomalejší. Experimentálně jsme zjistili, že pokud provedeme řádově  $10^6$  operací s `floaty`, zabere to podle výkonu počítače okolo 0,01 s. Stejně operace s racionální proměnnou může trvat až 1 s.

## Kapitola 4

# RACIONÁLNÍ DATOVÉ TYPY

V porovnání s `floatem` by racionální datový typ v iterování smyčkou v předchozí kapitole neztrácel přesnost a stále udržoval přesnou hodnotu  $0,4$ , potažmo  $2/5$ .

### 4.1 Způsob uložení

Největší smysl dává využít dvě celočíselné proměnné pro uložení čitatele a jmenovatele. Přitom je vhodné určit, že záporné znaménko může mít pouze čítel.

Alternativou je uložení předperiody a periody a jejich pozice vzhledem k desetinné čárce. Tedy z tohoto zápisu racionálního čísla bychom uložili  $p, q, e$ .

$$p \cdot b^e + \sum_1^{\infty} q \cdot b^{e-if}; p, f \in \mathbb{Z}, q, b \in \mathbb{N}, f = \lceil \log_b(q) \rceil$$

Přitom proměnná  $p$  by odpovídala předperiodě, konstanta  $b$  by byl základ číselné soustavy, i.e. 2. O řádu předperiody by rozhodovalo  $e$ . Bezprostředně za předperiodou by se v zápisu v soustavě základu  $b$  opakovala perioda  $q$ . Vzhledem k tomu, že nevíme kolik bitů perioda ve skutečnosti obsahuje, museli bychom rozhodnout, že perioda musí začínat jedničkou a zarovnat ji odzadu.

Jednu třetinu  $(0, \overline{01}_2)$  bychom uložili jako:  $p = 0, q = 10_2, e = -10_2$ .

Tento tvar sice dobře vypovídá o tom, jak je číslo velké, ale nevíme nic o tom, jaký by byl zlomkový tvar, podstatný pro násobení a hledání dělitelů.

Ještě by bylo možné ukládat celou část čísla zvlášť od necelé. Tím by vlastně vzniklo smíšené číslo. V takovém případě dokážeme uložit i číslo, které má jak velký jmenovatel, tak velkou hodnotu. Avšak toto většinou není potřeba, snižuje se tím výkon a je možné tuto implementaci vystavět na zlomkové implementaci.

Další možnost je ukládat koeficienty řetězového zlomku, ani základní aritmetické operace však není možné na běžných procesorech rychle provádět, proto toto není vhodné řešení.

### 4.2 Formy racionálních čísel

Formou čísla označme všechny výrazy, které se číslu rovnají. Všechny formy jednoho čísla tedy tvoří třídu ekvivalence vygenerovanou binární relací „rovná se“ na množině zlomků s čitateli a jmenovateli celými čísly, respektive takové množině, že jsou čítel a jmenovatel z nějaké konečné podmnožiny celých čísel.

Nabízí se otázka, kolik čísel umíme reprezentovat, tedy kolik tříd ekvivalence bude pro daný rozsah čitatele a jmenovatele existovat. Jinak řečeno, jaká je šance, že bude čítel nesoudělný se jmenovatelem.

Pravděpodobnost tohoto je pro 2 celá kladná čísla rovna  $\zeta^{-1}(2) = 6\pi^{-2} \approx 0,607$ . Kde  $\zeta$  značí Riemannovu zeta funkci. Toto zde nebudeme dokazovat. Z toho plyne, že pro dostatečně velký rozsah čitatele a jmenovatele bude okolo 60 % hodnot zlomků v základním tvaru.

Další věcí je, že pokud je jmenovatel záporný, změnou znaménka u čitatele a jmenovatele vznikne zlomek, který má stejnou hodnotu. Proto polovinu všech forem nebudeme využívat.

Při použití reprezentace čísel v podílovém tvaru je podíl reprezentovatelných čísel ku všem kombinacím bitů okolo tří desetin. Tedy nepotřebujeme až tak moc paměti zbytečně navíc oproti například ukládání předperiody a periody nebo plovoucí řádové čárce.

## Kapitola 5

# POPIS KNIHOVNY RACIONÁLNÍ ARITMETIKY PRO .NET CORE

Pro vytvoření knihovny byl zvolen jazyk C# jako součást CLI (Common Language Infrastructure) ve frameworku .NET Core, který je dostupný na Linux, Windows i Mac. Navíc je licencován pod MIT licencí. CLI umožňuje, aby byla knihovna využita i v jiném jazyku, tím může být nejčastěji C++/CLI<sup>1</sup>. Knihovna racionální číslo implementuje jako strukturu `Rational`.

### 5.1 Požadavky na knihovnu

1. Přesné výpočty sčítání, odčítání, násobení, dělení,
2. převod na řetězec ve zlomkovém tvaru a řetězec tvaru desetinného čísla,
3. vytvoření z řetězce jako zlomku a desetinného čísla,
4. převod do celočíselné proměnné se zaokrouhlením,
5. převod na číslo s plovoucí řádovou čárkou a zpět,
6. možnost pracovat s aproximovanými hodnotami nerepresentovatelných čísel.

Výpočet  $5 \cdot \frac{7}{4} + 6$  by v Javě mohl vypadat takto:

```
static void main(String [] args)
{
    Rational R = new Rational(5,1);
    Rational Q = new Rational(7,4);
    System.out.println(R.times(Q).plus(6).toString());
}
```

Vidíme, že by zřejmě bylo vhodné, kdyby operace s racionálními čísly vypadaly podobně, jako bychom je zapsali mimo programovací jazyk. V jazyku C# se tomuto můžeme přiblížit mnohem více, neboť můžeme předefinovat operátory. Navíc je definováno implicitní/explicitní přetypování.

```
static void Main(string [] args)
{
    var R = (Rational)5;
    var Q = (Rational)7/4;
    Console.WriteLine(R * Q + 6);
}
```

Příklad kódu ve VB.NET:

---

<sup>1</sup>K často používaným jazykům patří i Visual Basic .NET, zejména pro malé projekty pro svou jednoduchost.

```

Module Program
  Sub Main( args As String() )
    Dim R As RationalTypes.Rational
    Dim Exponent As Object
    R = New Rational(-5, 4)
    Exponent = 13
    Console.WriteLine(R.Pow(Exponent))
    Exponent = -2
    Console.WriteLine(R.Pow(Exponent))
  End Sub
End Module

```

Dosažené výsledky bychom chtěli dále použít v člověku srozumitelném tvaru. Proto je třeba převádět čísla na text.

## 5.2 Ústřední obsah knihovny

Základem knihovny je interface `IRational`<sup>2</sup>

Tento interface implementují 2 struktury. Struktura se hodí více než třída, protože chceme s racionálními čísly pracovat podobně, jako by to byla čísla typu `int` nebo `float`, což jsou struktury. Při většině operací navíc bude vznikat nový objekt.

Struktura `Rational` umožňuje operace na racionálních číslech s 64bitovou přesností čitatele a jmenovatele.

Struktura `BigRational` umožňuje operace na racionálních číslech s teoreticky neomezenou přesností. V implementaci je použit typ `BigInteger`. Prakticky je však přesnost omezena množstvím dostupné paměti a dlouhé zápisy čísel budou výrazně limitovat výkon.

Budeme chtít předefinovat tyto binární<sup>3</sup> operace:

operátor	operace	zápis	výsledek
+	sčítání	<b>a+b</b>	$a + b$
-	odčítání	<b>a-b</b>	$a - b$
*	násobení	<b>a*b</b>	$ab$
/	dělení	<b>a/b</b>	$\frac{a}{b}$
%	modulo	<b>a%b</b>	zbytek $a$ po dělení $b$
==	rovná se	<b>a==b</b>	$a = b$
!=	nerovná se	<b>a!=b</b>	$a \neq b$
<	menší než	<b>a&lt;b</b>	$a < b$
>	větší než	<b>a&gt;b</b>	$a > b$
<=	menší nebo rovno	<b>a&lt;=b</b>	$a \leq b$
>=	větší nebo rovno	<b>a&gt;=b</b>	$a \geq b$

Tabulka 5.1: Přehled aritmetických a logických operátorů

<sup>2</sup>Takové jsou konvence jazyka C#: Interface se nazývá charakterem jevu s předponou I-. Poznamenejme ještě, že název by se zkušeným angličtinářům neměl plést s iracionálním číslem. To jednak proto, že je R velké, ale taky v *irrational* se r objevuje zdvojené.

<sup>3</sup>Binární jsou z hlediska arity operace.

Zřejmě bude potřeba provádět i takové operace, kdy jeden z operandů bude ve formátu celého čísla. Proto tyto operátory definujeme pro dvojice racionální & racionální, racionální & 64bitové celé číslo. Kratší celá čísla už nemusíme řešit, protože se implicitně převedou na 64bitová.

Taktéž je podstatné, že u operací, kde nezáleží na pořadí operandů, tedy komutativních operací, je potřeba v případě rozdílnosti typů předdefinovat operátory pro obě pořadí operandů.

Samozřejmě naše množina reprezentovatelných čísel pochopitelně není uzavřena vůči těmto operacím. To však platí i o operacích na celých číslech, kde dochází k přetečení. V některých případech potřebujeme přetečení odhalit.

### 5.3 Speciální hodnoty

Po inicializaci se `Rational` převede na zlomek v základním tvaru. To stejné platí pro výsledky všech výpočtů. Toto sice má vliv na rychlost operací, ale na druhou stranu výrazně omezuje přetékání. Proto ale nebudou některé hodnoty jako 36/81 nikdy uloženy. Složitost všech operací se takto zvýší o složitost krácení zlomku. Nicméně, kdyby nějakým způsobem takové hodnoty vstoupily do operace, nezpůsobí to chybný výsledek.<sup>4</sup>

K těmto hodnotám ovšem existují další, které je třeba uživateli neumožnit vytvořit a výpočty nesmí vzniknout. Jsou to všechny hodnoty s nulovým jmenovatelem.

### 5.4 Vybraná funkčnost a algoritmy

#### Největší společný dělitel a nejmenší společný násobek

Pro velké množství operací (především krácení) budeme potřebovat určit největší společný dělitel (GCD), k tomu použijeme binární algoritmus pro hledání největšího společného dělitele. Budeme potřebovat počítat GCD i pro více než 2 čísla, v takovém případě použijeme GCD předchozích čísel jako vstup do algoritmu spolu s následujícím číslem.

Na hledání největšího společného dělitele existuje několik rychlých algoritmů. Tím hlavním je Euklidův algoritmus, ale ten nepoužijeme. Místo něj využijeme binární GCD algoritmus (Steinův algoritmus), který má podobnou myšlenku a stejnou asymptotickou složitost, ale v praxi je o desítky procent rychlejší.

Nejmenší společný násobek (LCM) budeme počítat podle známého vztahu:  $ab = lcm(a, b) \cdot gcd(a, b)$  Pro více čísel použijeme stejnou metodu jako v případě dělitele.

#### Násobení

Podstatné je, že pokud může být výsledek uložen, stále není zaručeno, že výpočet nepřeteče. To vidíme například na:

$$\frac{a}{b} \cdot \frac{b}{a} = 1$$

Pokud ovšem provedeme výpočet v tomto pořadí:

$$(a \cdot b) : (b \cdot a)$$

Výpočet zbytečně může přetéct.<sup>5</sup> Proto násobení implementujeme takto:

```
public static Rational operator *(Rational p, Rational q)
{
    long l = GCDE(p.Numerator, q.Denominator);
    long m = GCDE(p.Denominator, q.Numerator);
    checked
    {
```

<sup>4</sup>Krácení proběhne i pro čísla rovnající se 0, ta budou vždy převedena na 0/1.

<sup>5</sup>Tento konkrétní výpočet sice bude mít i tak správný výsledek, ale to většinou nebude platit.



```

        return new Rational((p.Numerator / l) * (q.Numerator / m),
        (p.Denominator / m) * (q.Denominator / l));
    }
}

```

Obecně je vhodné před násobením provést dělení, zde krácení společným dělitelem.

Kdo bude knihovnu používat určitě bude chtít vědět, zda operace proběhla v pořádku. K tomu slouží blok kódu `checked`. Pokud dojde k přetečení v tomto prostředí, program ohlásí chybu `System.OverflowException`.

V opačném případě by uživatel nedokázal poznat, jestli je výsledek správný či nikoliv. Ve všech metodách, kde se očekává, že přetékaní bude probíhat a že je tam nežádoucí, je použito prostředí `checked`.

## Zaokrouhlování

Knihovna umožňuje pět módů zaokrouhlování.

Druh zaokrouhlení	Název funkce
Dolní celá část	Floor
Horní celá část	Ceiling
K nule	Truncate
K nekonečnu	Stretch*
Aritmetické zaokrouhlení	Round

Tabulka 5.2: Módy zaokrouhlování

\* Metoda se běžně neužívá, proto nemá běžně zavedený název.

Zaokrouhlování je implementováno dvěma způsoby. Jednak je možno použít metodu, která upraví čísel a jmenovatel asociované proměnné. Nebo se dá využít vlastnost, která vrací přímo celočíselný výstup.

Dolní celá část vrací nejbližší nižší nebo rovné celé číslo.

Horní celá část vrací nejbližší vyšší nebo rovné celé číslo.

Zaokrouhlování k nule vrací nejbližší celé číslo, které je blíže k 0 na číselné ose. K tomuto se dá použít celočíselné dělení.

Zaokrouhlování k nule vrací nejbližší celé číslo, které je dále od 0 na číselné ose.

Aritmetické zaokrouhlování se skládá z rozboru případů.

```

public long GetRound
{
    get
    {
        var cmp = (_numerator * Sgn) % _denominator;
        if (cmp > _denominator / 2) { return GetStretch; }
        else if (cmp * 2 == _denominator) { return GetCeiling; }
        else { return GetTruncate; }
    }
}

```

Vypočteme hodnotu `cmp`, i.e. zbytek absolutní hodnoty z čitatele po dělení jmenovatelem. Pokud je `cmp` vyšší než dolní celá část poloviny jmenovatele, číslo se zaokrouhlí k nekonečnu. Pokud se dvojnásobek `cmp` rovná jmenovateli, pak hodnota tohoto čísla leží přesně mezi dvěma celými čísly, a proto se má zaokrouhlit nahoru. Zbývá pouze případ, že `cmp` je menší než polovina jmenovatele, pak se má číslo zaokrouhlit k nule.

## Odmocňování

V určitých situacích je potřeba zjistit, zda je číslo nějakou mocninou racionálního čísla. Za tímto účelem je implementována metoda `HighestPower`.

```
public int HighestPower()
{
    int res = 1; long _n = Math.Abs(Numerator), _d = Denominator;
    if (_n == 0 || _n == 1) { return int.MaxValue; }
    for (;;)
    {
        Next: foreach (int p in SmallPrimes)
        {
            bool a = Utility.Pow(binarySearchPower(_n, p), p) == _n;
            bool b = Utility.Pow(binarySearchPower(_d, p), p) == _d;
            if (a && b)
            {
                _n = binarySearchPower(_n, p);
                _d = binarySearchPower(_d, p);
                res *= p; goto Next;
            }
        }
        break;
    }
    if (Numerator < 0)
    {
        // Negative number cannot be even power.
        while ((res & 1) == 0)
        { res = res >> 1; }
    }
    return res;

    long binarySearchPower(long n, long p)
    {
        long l = 0, r = n, i = (n + 1) / 2;
        while (l < r)
        {
            try
            {
                if (Utility.Pow(i, p) > n) { r = i - 1; }
                else if (Utility.Pow(i, p) < n) { l = i; }
                else { return i; }
            }
            catch (OverflowException)
            { r = i - 1; }
            finally
            { i = (l + r + 1) / 2; }
        }
        return i;
    }
}
```

Tato metoda nám vrátí nejvyšší celočíselný exponent  $e$  takový, že  $b^e$  se rovná hodnotě instance  $a$  a  $b$  je racionální číslo. Efektivní způsob, jak toto určit, je kombinace umocňování na prvočíselné exponenty a použití binárního hledání. Pokud totiž pro racionální  $a_i, i \in \{nic, 1, 2\}$  a celočíselné  $e_i, i \in \{1, 2\}$  platí

$a^1 = a_1^{e_1} = a_2^{e_2} = a_{12}^{lcm(e_1, e_2)}$ , pak je i  $a_{12}$  racionální. Budeme proto zkoušet malá prvočísla  $p$  (do 64), jestli nějaká  $p$ -tá odmocnina není racionální. Pokud je, dále stačí zkoumat pouze  $p$ -tou odmocninu pro dělitelnost. Nakonec se nalezená prvočísla, vynásobí.

Odmocninu budeme hledat binárním hledáním.<sup>6</sup> Ověřování rozdělíme na čitatele a jmenovatele, obě hodnoty musí být po odmocnění celočíselné. Spočítáme mocninu  $m$  pro číslo uprostřed intervalu 1 až  $n$ , podle velikosti  $m$  vzhledem k  $n$  upravíme meze intervalu. Takto postupujeme, dokud není velikost intervalu 1.

Na závěr umocníme nalezené číslo pro čitatele i jmenovatel na  $p$ . Pokud se v obou případech výsledky rovnají čitateli, respektive jmenovateli, je  $p$ -tá odmocnina racionální.

Výpočet takto popsany provedeme pro číslo kladné, pokud je ovšem číslo záporné, budeme pracovat s číslem opačným. Pokud by byl výsledek v takovém případě sudý, budeme ho dělit dvěma, dokud nebude lichý.

Důležité je uvědomit si, že 0 a 1 jsou vzhledem k umocňování pevné body, zvolený výsledek pro tuto funkci je proto nejvyšší možný – `int.MaxValue`.

Podle výsledku `HighestPower(x)` se dá rychle určit, jestli bude  $\sqrt[n]{x}$  racionální číslo. To nastane když  $n|x$ . (Pro celé kladné  $n$ ).

Skutečnou hodnotu odmocniny potom vypočteme touto metodou:

```
public Rational Pow(Rational Exponent)
{
    if (Exponent <= 0 && this == 0)
    { throw new DivideByZeroException(); }
    else if (Exponent == 0)
    {
        if (this == 0) { throw new DivideByZeroException(); }
        return (Rational)1;
    }
    else if (this == 0) { return (Rational)0; }
    else if (this < 0 && (Exponent.Denominator & 1) == 0)
    { throw new ComplexResultException(); }
    long l = 0, r = Abs(Numerator); long i = (Abs(Numerator) + 1) / 2;
    long _n = 0, _d = 0;
    while (l < r)
    {
        try
        {
            if (Utility.Pow(i, Exponent.Denominator) > Abs(Numerator))
            { r = i - 1; }
            else if (Utility.Pow(i, Exponent.Denominator) < Abs(Numerator))
            { l = i; }
            else { _n = i; break; }
        }
        catch { r = i - 1; }
        finally { i = (l + r + 1) / 2; }
    }
    if (_n == 0) { throw new IrrationalResultException(); }
    l = 0; r = Denominator; i = (Denominator + 1) / 2;
    while (l < r)
    {
        try
        {
            if (Utility.Pow(i, Exponent.Denominator) > Denominator)
            { r = i - 1; }

```

<sup>6</sup>Najít odmocninu z čísla bychom mohli teoreticky i funkcí `Math.Pow(double, double)`, ale tu nepoužijeme.

```

        else if (Utility.Pow(i, Exponent.Denominator) < Denominator)
        { l = i; }
        else { _d = i; break; }
    }
    catch { r = i - 1; }
    finally { i = (l + r + 1) / 2; }
}
if (_d == 0) { throw new IrrationalResultException(); }
var res = new Rational(Utility.Pow(Sgn * _n, Abs(Exponent.Numerator)),
    Utility.Pow(_d, Abs(Exponent.Numerator)));
if (Exponent > 0) { return res; }
else { return res.Inverse; }
}

```

Nejdříve rozebereme výjimečné případy. 0 neumíme umocnit na nekladný exponent, proto oznámíme `DivideByZeroException`. Může se stát, že bude výsledek mít čitatele nebo jmenovatele příliš velký, v takovém případě ohlásíme `OverflowException`.

Kromě systémových výjimek `DivideByZeroException` a `OverflowException` mohou nastat ještě další případy:

- `IrrationalResultException` je třeba ohlásit, když výsledek není racionální a nemůžeme ho proto vrátit.
- `ComplexResultException` je třeba ohlásit, když existuje pouze komplexní mocnina.

Tyto výjimky jsou definovány přímo v knihovně.

Nejdříve provedeme odmocnění, poté umocnění. Budeme počítat s nezáporným exponentem, pokud má být exponent záporný, nakonec vrátíme převrácenou hodnotu výsledku. Nejdříve vyřešíme jmenovatel exponentu. Jako v předchozí metodě najdeme binárním hledáním řešení pro čitatele a jmenovatel zvlášť. Poté provedeme umocnění. Díky automatickým úpravám ve struktuře `Rational` bude exponent v nejjednodušším možném tvaru.

## Hledání kořenů rovnic

Jedno z možných použití racionálních proměnných je hledání racionálních kořenů polynomů. Knihovna implementuje třídu `Polynomial`, tato třída obsahuje pole celočíselných koeficientů polynomu. Jak je známo, všechny racionální kořeny polynomu s celočíselnými koeficienty musí dělit konstantní člen a být násobkem vedoucího koeficientu. Hornerovým algoritmem proto dokážeme tyto kořeny nalézt a to v přesném tvaru.

Mimo to ještě knihovna obsahuje třídu `RationalPolynomial`, kde koeficienty mohou nabývat racionálních hodnot. Nedojde ke změně kořenů, pokud se koeficienty vynásobí nejmenším společným násobkem jejich jmenovatelů. Takto se zároveň dá získat instance třídy `Polynomial` a kořeny hledat tomuto objektu.

## Rozklad na součin

Jestliže umíme najít kořeny polynomů, umíme také polynom rozložit na součet závorek typu: (proměnná – nulový bod). Stačí najít algoritmem všechny nulové body. Může se ovšem stát, že některý kořen bude vícenásobný, v takovém případě je potřeba polynom vydělit a opětovně nulový bod vyzkoušet. Na konci může zůstat z polynomu několik členů, které se nepodařilo rozložit. To může být proto, že nejsou kořeny racionální, nebo tím, že jsou přímo komplexní. Uvědomme si, že zbytek kořenů mít sice musí podle základní věty algebry, ale takové kořeny neumíme obecně hledat.

## Převod na text

Pro uživatele je textový výstup velmi důležitý. Pro výpis do konzole jako prostý text se čítel a jmenovatel jednoduše oddělí lomítkem. Počítáme s tím, že uživatel bude s největší pravděpodobností text formátovat podle pravidel matematického zápisu v programu  $\text{\TeX}$ . Toto formátování se využívá rovněž na Wikipedii a de facto i v MS Office. Dokonce existují volné knihovny v Java Scriptu, které naformátují podle těchto pravidel i HTML dokument<sup>7</sup>.

Kromě toho se v určitých situacích hodí zaokrouhlené desetinné číslo. Toto umíme dokonce v různých základech soustavy.

## 5.5 BigRational

Tato struktura je podstatná především kvůli výpočtům s mnoha kroky. Krácením totiž nemusí vůbec dojít k redukci a velikost čítele a jmenovatele může rychle narůstat. Vidíme například:

$$2^{64} = 18446744073709551616 < 22!$$

Vynásobením prvních 22 přirozených čísel (nebo jejich převrácených hodnot) dojde k překročení velikosti proměnné `long`. Při použití proměnné typu `Rational` by operace skončila ohlášením chyby `OverflowException`.

Pro mnohá využití knihovny je ovšem potřeba počítat s takto velkými čísly.

Pro čítel a jmenovatel je použit `BigInteger`, což je celé číslo omezené pouze pamětí počítače. Prakticky se tedy nemůže stát, že by proměnná přetekla. `BigRational` se ve většině případů chová stejně jako `Rational`. Přitom `Rational` se dá konvertovat na `BigRational`. Operace lze provádět, i když je jeden z operandů `Rational` a druhý `BigRational`. Výsledkem je v takovém případě `BigRational`, nejedná-li se o logické operace.

## 5.6 Vyhodnocování výrazů

Pro základní výpočty (+, -, \*, /) můžeme využít na počítači řadu programů. Například vestavěný kalkulátor, nebo mnohem lepší programy typu Mathlab, Scilab, Octave. Tyto programy ovšem nedokáží uživateli poskytnout výsledek ve tvaru zlomku. Vidíme to na obrázku 5.1, pro porovnání je stejný výpočet proveden ve vytvořené aplikaci (obrázek 5.2).

Knihovna proto obsahuje třídu na vyhodnocování těchto výrazů. Algoritmus, kterým funguje je popsán níže:

Použijme pro ilustraci následující výraz:

$$63 + 2(25 + 7/8) - 17 * 22/6$$

Výraz se nejdříve rozdělí na členy a operátory, u toho jsou mezery mezi členy a operátory ignorovány. Pokud žádný operátor není přítomen, implicitní je násobení.

$$\{63, +, 2, (25 + 7/8), -, 17, 22, /, 6\}$$

Členy ohraničené závorkami se vyhodnotí rekurzivně jako plnohodnotné výrazy.

$$\left\{ 63, +, 2, \frac{207}{8}, -, 17, 22, /, 6 \right\}$$

Další fází je, že se od začátku výrazu začnou vyhodnocovat členy. Zavedou se dvě proměnné inicializované na 0: Průběžná hodnota a výsledná hodnota. Ve chvíli, kdy další člen je + nebo -, průběžná hodnota se přičte k výsledné hodnotě. Pak se průběžná hodnota nastaví na 1.

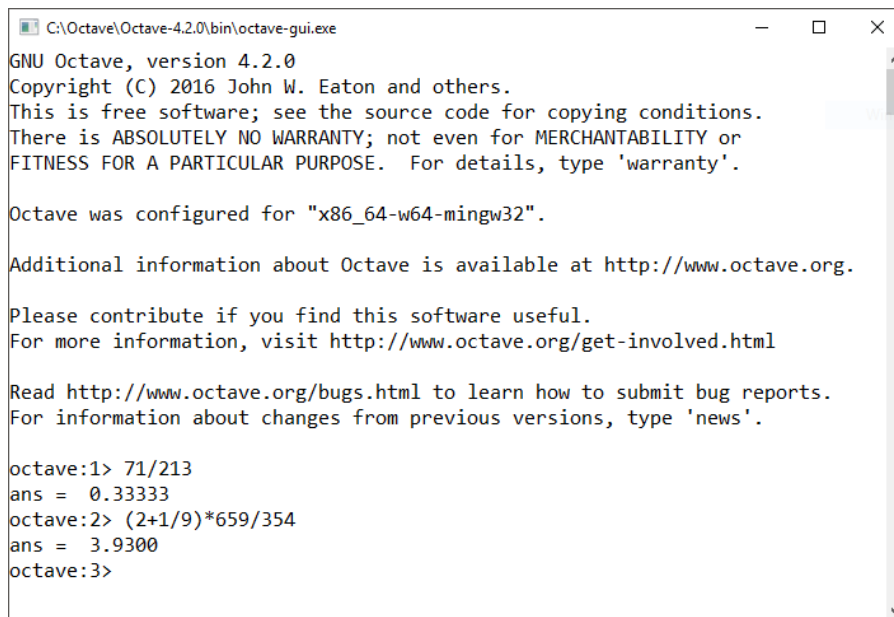
Operace \* a / způsobí, že se průběžná hodnota vynásobí, respektive vydělí následujícím členem.

<sup>7</sup>Například MathJax: <https://www.mathjax.org/>

Při vyhodnocování prvního členu se do průběžné hodnoty uloží 63, další člen tuto hodnotu přičte k výsledné hodnotě. Třetí člen nastaví průběžnou hodnotu na 2, následující člen tuto hodnotu vynásobí. Mínus způsobí, že se průběžná hodnota přičte k výsledné. Zároveň mínus nastaví do proměnné, že se má při další příležitosti průběžná hodnota odečíst, ne přičíst.

Podstatné dále je, aby všechny chyby v zápise na vstupu byly odhaleny a program neodevzdával chybné výsledky. Chyby mohou vznikat mimo jiné následujícími způsoby:

- Nevyvážené závorky:  $5 + 3 * (1 + (10 - 11)$
- Neznámé znaky: ahoj!
- Opakovaná znaménka:  $2 + +2$
- Neplatný první člen nebo poslední člen:  $*168$
- Rozdělené číslo:  $45638950 + 45620\ 796$
- Prázdná závorka:  $5456 + () + 32216$
- Dělení 0:  $5/(7 - 21/3)$
- Příliš vysoká hodnota:  $234189094213590212806 * 2$



```
C:\Octave\Octave-4.2.0\bin\octave-gui.exe
GNU Octave, version 4.2.0
Copyright (C) 2016 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-w64-mingw32".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1> 71/213
ans = 0.33333
octave:2> (2+1/9)*659/354
ans = 3.9300
octave:3>
```

Obrázek 5.1: Výpočty v GNU Octave



```
C:\Program Files\dotnet\dotnet.exe
Výraz ke spočtení:
71/213
=
1/3
Výraz ke spočtení:
(2+1/9)*659/354
=
12521/3186
Výraz ke spočtení:
```

Obrázek 5.2: Stejné výpočty ve vytvořené aplikaci

## Kapitola 6

# ZÁKLADNÍ PŘEHLED KNIHOVNY RATIONALTYPES

### Interface IRational

#### Metody

**byte** [] Approximation(**byte**, **int**)  
**void** Ceiling()  
**void** Floor()  
**void** Round()  
**void** Stretch()  
**void** Truncate()  
**string** ToSignedString()  
**string** ToSignedTeXString()  
**string** ToTeXString()

#### Vlastnosti

**string** DecimalApproximation  
**double** DoublePrecisionValue  
**float** SinglePrecisionValue

### Struktura Rational

#### Konstruktory

Rational(**long**)  
Rational(**long**, **long**)

Implementuje interface IRational

#### Metody

Rational Abs()  
**int** HighestPower()  
Rational Parse(**string**)  
Rational Pow(**int**)  
Rational Pow(Rational)

#### Vlastnosti

Rational Inverse  
**long** Denominator  
**long** Numerator  
**sbyte** Sgn

**long** GetCeiling  
**long** GetFloor  
**long** GetRound  
**long** GetStretch  
**long** GetTruncate

#### Binární operátory pro long a Rational

!=, %, -, \*, /, +, <, >, ==, >=, <=

#### Unární operátory

+, -

### Struktura BigRational

#### Konstruktory

BigRational(**long**)  
BigRational(**long**, **long**)  
BigRational(BigInteger)  
BigRational(BigInteger, BigInteger)

Implementuje interface IRational

#### Metody

BigRational Abs()  
BigInteger Parse(**string**)  
BigRational Pow(**int**)

#### Vlastnosti

BigRational Inverse  
BigInteger Denominator  
BigInteger Numerator  
**sbyte** Sgn  
BigInteger GetCeiling  
BigInteger GetFloor  
BigInteger GetRound  
BigInteger GetStretch  
BigInteger GetTruncate

Binární operátory pro long, Rational a BigRational



!=, %, -, \*, /, +, <, >, ==, >=, <=

### Unární operátory

+, -

## Třída Polynomial

### Konstruktor

Polynomial(**long** [])

### Metody

**long** At(**long**)

**string** ToString(**char**)

### Vlastnosti

**long** [] Coefficients

**int** Degree

Polynomial Derivative

**long** this[**int**]

## Třída RationalPolynomial

### Konstruktor

RationalPolynomial(Rational [])

### Metody

**long** At(Rational)

**string** ToString(**char**)

**string** ToTeXString(**char**)

Polynomial MakePolynomial()

**void** DivideBySolution(Rational)

### Vlastnosti

Rational [] Coefficients

**int** Degree

RationalPolynomial Derivative

Rational this[**int**]

# Kapitola 7

## ZÁVĚR

Byla vytvořena vlastní implementace racionální aritmetiky v jazyce C#. Je možné ji použít prostým dynamickým odkazem (dll). A to na všech hlavních platformách (Linux, Windows, OS X).

Důraz byl kladen především na to, aby se syntakticky zápis příliš nelišil od používání `int`ů. Toho se podařilo docílit předefinováním operátorů. Většina využití tkví v začlenění do většího projektu. Ovšem sama knihovna dovede hledat racionální kořeny polynomů.<sup>1</sup>

Vyhodnocení řetězce s matematickým zápisem je způsob, jak získat výsledek ve formě zlomku. Standardní počítačové kalkulátory toto neumožňují.

Rychlost výpočtu s racionálními čísly může být ovšem výrazně nižší než u celých čísel a čísel s plovoucí řádovou čárkou. Toto není překvapivé, neboť racionální výpočty nejsou implementované hardwarově. Pokud je tedy třeba provádět řádově  $10^6$  operací za sekundu a více, je možné, že výkon nebude dostatečný.

Použití knihovny je povoleno pod licencí GNU Lesser General Public Licence verze 3. Je tedy možno nakládat prakticky libovolně.

---

<sup>1</sup>Pro toto se běžně využívají webové aplikace jako Wolfram Alpha, jenže časové prodlevy při přenosu jsou značné a interoperabilita s dalšími aplikacemi nesnadná.

# SEZNAM OBRÁZKŮ

1.1	Úsečky a mřížové body . . . . .	6
5.1	Výpočty v GNU Octave . . . . .	23
5.2	Stejné výpočty ve vytvořené aplikaci . . . . .	23

# SEZNAM TABULEK

3.1	Standard IEEE 754 . . . . .	9
3.2	Výkon grafických procesorů . . . . .	10
5.1	Přehled aritmetických a logických operátorů . . . . .	15
5.2	Módy zaokrouhlování . . . . .	17

# SEZNAM PŘÍLOH

- A. Zdrojový kód knihovny
- B. Zdrojové kódy příkladů

# LITERATURA

- [1] KNUTH, Donald Ervin. The Art of Computer Programming. Upper Saddle River, NJ: Addison-Wesley, 2011. ISBN 978-0321751041.
- [2] Cortex-M3 Devices Generic User Guide. Arm Developer [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
<https://developer.arm.com/docs/dui0552/latest/the-cortex-m3-instruction-set>
- [3] Intel Core 8700k. Ark Intel [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
[https://ark.intel.com/products/126684/intel-core-i7-8700k-processor-12m-cache-up-to-4\\_70-ghz](https://ark.intel.com/products/126684/intel-core-i7-8700k-processor-12m-cache-up-to-4_70-ghz)
- [4] Licensing:MIT. Fedora Project Wiki [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
<https://fedoraproject.org/wiki/Licensing:MIT?rd=Licensing/MIT>
- [5] AMD Radeon RX Vega 64. Tom's Hardware [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
<http://www.tomshardware.com/news/amd-radeon-rx-vega-64-specs-availability,35112.html>
- [6] Choosing Between Class and Struct. Microsoft Docs [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/-choosing-between-class-and-struct>
- [7] Řetězové zlomky a dobré aproximace [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
<http://www.karlin.mff.cuni.cz/~holub/soubory/Retez.pdf>
- [8] Polynomy a racionální lomené funkce [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
<http://math.feld.cvut.cz/mt/txtb/4/txc3ba4b.htm>
- [9] BERTHOLD K. P. HORN: Rational Arithmetic for Minicomputers [online]. [Citováno 15. dubna 2018].  
Dostupné z:  
[http://people.csail.mit.edu/bkph/papers/Rational\\_Arithmetic\\_OPT.pdf](http://people.csail.mit.edu/bkph/papers/Rational_Arithmetic_OPT.pdf)