Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Detailní vizualizace průběhu funkce

Plzeň, 2017 Daniel Rajf

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 26.4.2017 Daniel Rajf,

Abstract

Detail function visualization

The purpose of this thesis is to visually investigate precision of various implementations of mathematic functions from `math.h`. This thesis deals with parsing of arithmetic expressions and its visualization.

The application allows visualizing even complicated arithmetic expressions. It can also compare results of the same arithmetic expressions using different implementation of `math.h` for computation.

Arithmetic expressions are parsed using the shunting-yard algorithm. In addition, the visualization uses MPIR library for high-precision computation. The application is written in C# programming language and uses .NET Framework.

Abstrakt

Tato práce se zabývá porovnáváním přesností různých implementací matematických funkcí `math.h`, zpracováním aritmetických výrazů a jejich vizualizací.

Vytvořená aplikace umožňuje vizualizovat i složitější matematické funkce. Dále umožnuje porovnávat výsledky stejných funkcí, které jsou spočítané různými algoritmy.

Aritmetické výrazy jsou zpracovány pomocí algoritmu shunting-yard. Vizualizace využívá knihovnu MPIR pro výpočty s vysokou přesností. Aplikace je napsaná v jazyce C# a využívá technologii .NET Framework.

Obsah

1 Úvod ... 3
2 Zobrazení desetinných čísel v paměti počítače ... 4
2.1 Poloviční přesnost (binary16) ... 5
2.2 Základní přesnost (binary32) ... 5
2.3 Dvojitá přesnost (binary64) ... 6
2.4 Čtyřnásobná přesnost (binary128) ... 6
2.5 Další formáty IEEE 754 ... 6
3 Implementace matematických funkcí v jazyce C ... 8
3.1 Funkce sin(x) v knihovně Sun libmcr ... 10
4 Vizualizace velkých množin dat ... 14
5 Volba technologií ... 16
5.1 Jazyk C# ... 16
5.2 Technologie .NET Framework ... 17
6 Návrh aplikace ... 18
7 Popis implementace ... 20
7.1 Aritmetické výrazy ... 20
7.1.1 Zpracování aritmetického výrazu v infixové notaci ... 20
7.1.2 Validace aritmetického výrazu v postfixové podobě ... 23
7.1.3 Vyhodnocování aritmetického výrazu ... 24
7.2 Výpočet matematických funkcí ... 25
7.2.1 Vestavěné funkce ... 25
7.2.2 Nativní knihovny ... 25
7.3 Vizualizace matematických funkcí ... 30
7.3.1 Posunutí a přiblížení ... 30
7.3.2 Režimy vykreslování ... 31
7.3.3 Vykreslování vizualizace ... 36
7.4 Vyhledávání rozdílů na intervalu ... 37
7.5 Ukládání hodnot matematických funkcí do souboru ... 39

7.6 Zobrazení seznamu funkcí v uživatelském rozhraní ... 41
8 Testování ... 45
9 Závěr ... 48
Literatura ... 50
Uživatelská příručka ... 55

1 Úvod

Cílem této práce je vizualizace běžných matematických funkcí aproximovaných pomocí různých implementací knihovních funkcí `math.h`. Práce se dále zabývá pozorováním průběhu těchto funkcí a hledáním rozdílů ve výsledcích jednotlivých algoritmů.

Algoritmy pro výpočet složitějších matematických funkcí zpravidla využívají aproximace vypočítané pomocí polynomů tak, aby dosáhly co nejpřesnějších výsledků. Touto metodou však nelze dosáhnout přesných výsledků. Další nepřesnost je dána reprezentací desetinných čísel v počítači, které pracují s omezenou konečnou přesností. Proto budeme zobrazovat průběh funkce s přesností na jednotlivé bity parametrů i výsledku a na této úrovni budeme také hodnoty porovnávat.

Práce také porovnává různé stupně přesnosti čísel s pohyblivou desetinnou čárkou a jejich další omezení.

2 Zobrazení desetinných čísel v paměti počítače

Nejčastěji používaná reprezentace desetinných čísel je definována standardem IEEE 754 [1], resp. nejnovější revizí IEEE 754-2008 [2]. Tento standard definuje formáty pro reprezentaci čísel v pohyblivé desetinné čárce a zahrnuje pravidla zaokrouhlování, speciální hodnoty, dostupné operace a zpracování výjimek. Tento standard je zabudován ve většině dnešních mikroprocesorů a grafických karet.

Konečné číslo x je vyjádřeno vztahem [3]:

$$x = (-1)^s \cdot 2^{(e-p)} \cdot m \tag{2.1}$$

kde s je znaménkový bit, e je hodnota exponentu posunutého o hodnotu p tak, aby bylo toto číslo v paměti vždy kladné, m je mantisa.

Tento standard specifikuje také tyto speciální hodnoty:

- Kladná nula $s = 0$; $e = 0$; $m = 0$
- Záporná nula $s = 1$; $e = 0$; $m = 0$
- Kladné nekonečno $s = 0$; všechny bity u e nastaveny na 1; $m = 0$
- Záporné nekonečno $s = 1$; všechny bity u e nastaveny na 1; $m = 0$
- NaN *(not a number)* $s = 0$ nebo 1; všechny bity u e nastaveny na 1; $m > 0$

Dále formát rozlišuje dva druhy čísel. Normalizované číslo je číslo, které má v mantise uloženou hodnotu z intervalu $\langle 1; 2-\varepsilon \rangle$, kde ε je nejmenší hodnota mezi dvěma čísly, tj. číslo s hexadecimální reprezentací rovné `0x1`. První bit mantisy by byl vždy rovný jedné, a proto se neukládá.

Denormalizovaná čísla jsou hodnoty z intervalu $(0; 1-\varepsilon\rangle$, kde ε je nejmenší hodnota mezi dvěma čísly. Tento druh čísel se využívá pro velmi malé hodnoty. První bit mantisy by v tomto případě byl vždy rovný nule a exponent je nastavený na nejnižší hodnotu.

V případě, že se jedná o normalizované nebo denormalizované číslo, je možné získat nejbližší další (resp. nejbližší předchozí) číslo přetypováním na celé číslo o stejné bitové délce a přičtením (resp. odečtením) čísla 1.

Standard definuje několik binárních formátů s různými stupni přesnosti.

2.1 Poloviční přesnost (binary16)

Poloviční přesnost, v některých programovacích jazycích označována jako `half`, zabírá 16 bitů paměti, konkrétně 1 bit pro znaménko, 5 bitů pro exponent a 10 bitů pro mantisu. Konečné číslo x je vyjádřeno vztahem:

$$x = (-1)^s \cdot 2^{(e-15)} \cdot \left(1 + \sum_{i=1}^{10} b_{10-i} \cdot 2^{-i}\right) \tag{2.2}$$

Nejmenší možné kladné číslo (denormalizované) je $2^{-24} \approx 5{,}96 \cdot 10^{-8}$, nejmenší kladné normalizované je $2^{-14} \approx 6{,}10 \cdot 10^{-5}$ a největší reprezentovatelné číslo je pouze $(2 - 2^{-10}) \cdot 2^{15} = 65504$. Takto nízká přesnost je využívána hlavně grafickými kartami, které pracují s větším objemem dat, pro úsporu paměti a urychlení výpočtů na úkor nižší přesnosti. Formát byl zařazen až do nové verze standardu IEEE 754-2008 [2].

2.2 Základní přesnost (binary32)

Základní přesnost, v některých programovacích jazycích označována jako `float`, zabírá 32 bitů paměti, konkrétně 1 bit pro znaménko, 8 bitů pro exponent a 23 bitů pro mantisu. Konečné číslo x je vyjádřeno vztahem:

$$x = (-1)^s \cdot 2^{(e-127)} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) \tag{2.3}$$

Nejmenší možné kladné číslo (denormalizované) je $2^{-149} \approx 1{,}4 \cdot 10^{-45}$, nejmenší kladné normalizované je $2^{-126} \approx 1{,}1755 \cdot 10^{-38}$ a největší reprezentovatelné číslo je $(2 - 2^{-23}) \cdot 2^{128} \approx 3{,}403 \cdot 10^{38}$.

2.3 Dvojitá přesnost (binary64)

Dvojitá přesnost, v některých programovacích jazycích označována jako `double`, zabírá 64 bitů paměti, konkrétně 1 bit pro znaménko, 11 bitů pro exponent a 52 bitů pro mantisu. Konečné číslo x je vyjádřeno vztahem:

$$x = (-1)^s \cdot 2^{(e-1023)} \cdot \left(1 + \sum_{i=1}^{52} b_{52-i} \cdot 2^{-i}\right) \tag{2.4}$$

Nejmenší možné kladné číslo (denormalizované) je $2^{-1074} \approx 5 \cdot 10^{-324}$, nejmenší kladné normalizované je $2^{-1022} \approx 2{,}2251 \cdot 10^{-308}$ a největší reprezentovatelné číslo $(2 - 2^{-52}) \cdot 2^{1024} \approx 1{,}7977 \cdot 10^{308}$.

2.4 Čtyřnásobná přesnost (binary128)

Čtyřnásobná přesnost zabírá 128 bitů paměti, konkrétně 1 bit pro znaménko, 15 bitů pro exponent a 112 bitů pro mantisu. Formát byl také zařazen až do nové verze standardu IEEE 754-2008 [2]. Takto vysoká přesnost není v současnosti procesory, které jsou postavené na architektuře x86, podporována. Proto je nutné pro práci s tak vysokou přesností využít softwarové řešení.

2.5 Další formáty IEEE 754

Původní standard IEEE 754-1985 [1] také definuje formáty rozšířené přesnosti. Tyto formáty jsou využity pro provádění interních výpočtů s vyšší

přesností, než je vyžadováno pro konečný výsledek. Tím dojde ke snížení chyby, která vzniká při zaokrouhlování mezivýpočtů.

Revize IEEE 754-2008 [2] nově definuje také desetinné formáty `decimal64` a `decimal128`. Tyto formáty však nejsou běžným hardwarem (procesory založené na architektuře x86 a ARM) podporovány a musí být zpracovány softwarově. Využívají se hlavně ve finančnictví, kde je potřeba správné desetinné zaokrouhlování.

Tato práce se zaměřuje hlavně na čísla s dvojitou přesností (`binary64`). Pro tuto přesnost je také primárně implementována většina matematických funkcí ve standardních knihovnách, viz kapitola 3.

3 Implementace matematických funkcí v jazyce C

Překladače jazyka C implementují základní matematické operace, tj. sčítání, odčítání, násobení, dělení a zbytek po dělení, ve formě operátorů, které se mapují přímo na instrukce procesoru a jsou zpracovány hardwarem. Většina překladačů navíc poskytuje standardní knihovnu, která implementuje mimo jiné i mnoho běžných matematických funkcí, případně může odkazovat na knihovnu, která je součástí operačního systému nebo je distribuována odděleně. Mezi tyto funkce patří mocnění, výpočet druhé odmocniny, exponenciální funkce, přirozený a desítkový logaritmus, funkce pro zaokrouhlování a goniometrické, cyklometrické a hyperbolické funkce [4] [5].

V Unixových systémech je tato knihovna považována přímo za součást operačního systému. Naproti tomu jádro operačního systému Microsoft Windows využívá vlastní knihovny, které nejsou přístupné jiným aplikacím [6]. V případě použití překladače od firmy Microsoft pro překlad vlastní aplikace je nutné do operačního systému doinstalovat příslušné distribuční balíčky Microsoft Visual C++ [7] [8]. S každou novou verzí prostředí Microsoft Visual Studio zpravidla vychází nová verze překladače a knihovny, na kterou se linkují přeložené aplikace tímto překladačem [9].

Základní matematické operace zpravidla bývají vyhodnocovány přímo procesorem. Všechny procesory založené na architektuře x86, které podporují instrukční sadu x87 pro práci s čísly s pohyblivou desetinnou čárkou, tyto operace podporují a navíc obsahují instrukce pro výpočet druhé odmocniny a goniometrických funkcí (sinus, kosinus, tangens). Ty však používají pouze 66-bitovou aproximaci čísla π pro redukci argumentu [10] [11], která je před samotným výpočtem vyžadována, protože goniometrické a cyklometrické funkce bývají implementovány pomocí polynomů o různých stupních na úzkém intervalu, například od 0 do π. Algoritmy využívají periodicity funkcí. Tím, že se hodnoty funkce opakují, je možné vytvořit algoritmus, který dokáže zpracovat pouze určitý interval vstupních hodnot. Před provedením samotného algoritmu je provedena takzvaná redukce argumentu, která sníží (nebo případně jinak upraví)

vstupní hodnotu tak, aby jí algoritmus mohl použít pro výpočet s tím, že výsledek by měl být stejný, jako by byl výpočet proveden s původní hodnotou. Proto nepřesná redukce argumentu pro vyšší hodnoty způsobuje ztrátu přesnosti ještě před tím, než dojde k výpočtu hodnoty goniometrické funkce.

Pro použití na procesorech, které neumožňují výpočet těchto funkcí, nebo pro zvýšení přesnosti, se využívají softwarová řešení. Obě varianty však využívají podobné algoritmy.

3.1 Funkce sin(x) v knihovně Sun libmcr

Jako ilustrační příklad matematické knihovny jsem zvolil knihovnu Sun libmcr [12]. Jedná se o knihovnu s otevřeným zdrojovým kódem, která umožnuje počítat mocninu, logaritmus a goniometrické funkce (sinus, kosinus, tangens).

```
double __libmcr_sin(double x)
{
    int hx, ix, rnd = 0, nw, lx, ncrd;
    double z, er;

    float fw, ft;
    ix = hx = HIGH_WORD(x);
    ix = hx & 0x7fffffff;
    lx = LOW_WORD(x);
    if ((ix | lx) == 0)
        return (x);  /* sin(x) = x when x=0 */
    z = rndc;
    ft = (float)z;
    /* filter out exception argument */
    if (ix >= 0x7ff00000)
        return (x - x);     /* inf and nan */
    fw = ft * ft;
    /* if |x|<cbrt(3.0)*2**-26 */
    if (ix < 0x3e571374 || (ix == 0x3e571374 && lx <= 0x49123ef6)) {
        z = twom1022;
        if (hx < 0)
            z = -z;
        if (ix >= 0x03800000)
            return (x - z);/* if |x|>=2**-967 */
        else {             /* if |x|<=2**-1022 */
            if ((ix < 0x00100000) || ((ix - 0x00100000) | lx) == 0)
                return (x * ohu);
            /* otherwise: 2**-1022 < |x| < 2**-967 */
            else
                return ((x * two300 - z) * twom300);
        }
    }
    rnd = (*(int *)&fw) & 3;

    z = __libmcr_mx_sin(x, &er);
    ncrd = __libmcr_mx_check(&z, rnd, er);
    nw = 8;
    while (ncrd == 1) {
        z = __libmcr_mi_sin(x, nw, &ncrd, rnd);
        nw += 2;
    }
    return (z);
}
```

Výpis 1: Ukázka zdrojového kódu funkce `__libmcr_sin`, převzato z [13].

Tato kapitola se zabývá konkrétně implementací funkce sinus. Implementace je rozdělena do několika funkcí, hlavní funkce je uložena v souboru `__libmcr_sin.c` [13], viz výpis 1.

Hodnota funkce $\sin(x)$ je počítána různými vzorci na základě vstupní hodnoty x. Výpočet probíhá následovně:

1. $\sin(0) = 0$
2. $\sin(\pm\infty \,/\, \text{NaN}) = x - x$
3. pro $|x| \leq 2^{-1022}$, $\sin(x) = x \cdot \left(1 - \frac{ulp^{1}}{2}\right)$
4. pro $|x| \leq 2^{-967}$, $\sin(x) = 2^{-300} \cdot (x \cdot 2^{300} - 2^{-1022})$
5. pro $|x| < \sqrt[3]{3} \cdot 2^{-26}$, $\sin(x) = x - 2^{-1022}$
6. pro ostatní hodnoty zavolá funkci `__libmcr_mx_sin()`, která vrací téměř správně zaokrouhlené výsledky
7. pokud je výsledek vyhodnocen jako nepřesný, volá funkci `__libmcr_mi_sin()` pro výpočet se zvýšenou přesností pomocí Taylorova polynomu. Vyhodnocení probíhá tak, že algoritmus zkontroluje, zda korekce chyby je blízko hodnotě $\frac{1}{2}ulp$.

Funkce `__libmcr_mx_sin()` pracuje tímto způsobem [14]:

- S a C označuje funkci sin, resp. cos na intervalu $\langle\frac{-\pi}{4}; \frac{+\pi}{4}\rangle$.
- Předpokládejme, že argument x je zredukovaný do $y_1 + y_2 = x - k \cdot \frac{\pi}{2}$ na intervalu $\langle\frac{-\pi}{2}; \frac{+\pi}{2}\rangle$ a $n = (k \bmod 4)$.
- Nechť $S = S(y_1 + y_2)$ a $C = C(y_1 + y_2)$. V závislosti na n dostáváme převodní tabulku, viz tabulka 1.

[1] ulp - Unit in the Last Place – rozdíl dvou po sobě jdoucích desetinných čísel

n	$\sin(x)$	$\cos(x)$	$\tan(x)$
0	S	C	S/C
1	C	$-S$	$-C/S$
2	$-S$	$-C$	S/C
3	$-C$	S	$-C/S$

Tabulka 1: Převodní tabulka funkcí sinus a kosinus.

Pro výpočet S a C se využívají funkce `__libmcr_k_mx_sin` a `__libmcr_k_mx_cos`. Funkce `__libmcr_k_mx_sin` počítá hodnotu $\sin(x+t)$ Remezovou aproximací [15] s přesností až 2^{-79} pro $|x+t| < \frac{\pi}{4}$, pomocí vzorce [16]:

$$z = x \cdot x,\ s = x + x \cdot z \cdot \left(a_1 + z \cdot (a_2 + \cdots + z \cdot a_8)\right) \tag{3.1}$$

Od a_5 můžeme použít aritmetiku s dvojitou přesností:

$$t = z \cdot \left(a_5 + z \cdot \left(a_6 + z \cdot (a_7 + z \cdot a_8)\right)\right) \tag{3.2}$$

Poté můžeme simulovat vyšší přesnost:

$$s = x \cdot (1{,}0 + z \cdot \left(a_1 + z \cdot \left(a_2 + z \cdot \left(a_3 + z \cdot (a_4 + t)\right)\right)\right) \tag{3.3}$$

Podobně funguje i výpočet v `__libmcr_k_mx_cos`, kdy spočítáme $\cos(x+t)$ Remezovou aproximací [15] pomocí vzorce:

$$z = x \cdot x,\ s = 1 - \frac{z}{2} + (z \cdot z) \cdot \left(a_1 + z \cdot (a_2 + \cdots + z \cdot a_8)\right) \tag{3.4}$$

Od a_4 můžeme použít aritmetiku s dvojitou přesností:

$$t = z \cdot \left(a_4 + z \cdot \left(a_5 + z \cdot \left(a_6 + z \cdot (a_7 + z \cdot a_8)\right)\right)\right) \tag{3.5}$$

Poté můžeme simulovat vyšší přesnost:

$$s = 1 - \frac{z}{2} + z^2 \cdot \left(a_1 + z \cdot \left(a_2 + z \cdot (a_3 + t)\right)\right) \tag{3.6}$$

Pokud výsledek z `libmcr_mx_sin.c` není vyhodnocen knihovnou jako dostatečně přesný, použije se funkce ze souboru `libmcr_mi_sin.c`. Stejně jako v předchozím případě obsahuje funkci `__libmcr_mi_sin`, která volá funkce `__libmcr_k_mm_sin` a `__libmcr_k_mm_cos`. Výběr funkcí je proveden stejným způsobem jako v předchozím případě. Výpočet je prováděn pomocí Taylorova polynomu místo Remezovou aproximací následujícím způsobem [17]:

- Nechť mx je vstupní hodnota, $mc = mx$, $mr = -mx \cdot mx$, $mt = mx$.

$$\text{Pro } j \in \{2, 4, 6, \dots\} \quad mt = \frac{mt \cdot mr}{j \cdot (j+1)} \tag{3.7}$$

  Pokud $mc - mt \geq nw$ (kde nw je předem stanovená přesnost), tak zastavit výpočet, jinak $mc = mc + mt$.

- Tím spočítáme tento Taylorův polynom:

$$mc = mx - \frac{mx^3}{3!} + \frac{mx^5}{5!} - \frac{mx^7}{7!} + \cdots \tag{3.8}$$

4 Vizualizace velkých množin dat

Množinu dat označíme za velkou, pokud jí nemůžeme zobrazit na obrazovce celou. Vizualizaci limituje počet pixelů obrazovky. Při maximálním využití plochy můžeme zobrazit přesně jednu položku na jeden pixel. Pokud je množství dat větší, musí dojít k určitému zkreslení. Položky můžeme barevně odlišit a tím danou vizualizaci zpřehlednit. Současné počítače využívají barevný model RGB (red, green, blue) a 8 bitů pro každý barevný kanál. Na monitoru lze tedy zobrazit až 256^3 $(= 16\,777\,216)$ barev. Pokud bychom ale k vizualizaci využili všechny dostupné odstíny barev, bylo by prakticky nemožné okem rozlišit jednotlivé barvy.

Pro zajištění vyššího výkonu je vhodnější využít k vykreslení technologii podporující hardwarovou akceleraci grafickou kartou, která dokáže vykreslit velké množství dat několikanásobně rychleji než procesor softwarovými algoritmy.

V případě, že se pokusíme vykreslit všechna data, může být výsledná vizualizace příliš zahuštěná nebo může dojít k překrývání. Jedna z možností je využít vzorkování a následnou rekonstrukci dat [18]. Operace rekonstrukce spočívá v tom, že pomocí interpolace určujeme hodnoty mezi navzorkovanými body. Případně je možné vizualizovat s přiblížením pouze určitou podmnožinu dat, dochází zde ale vždy ke ztrátě informací. Pokud se položky překrývají, může být vhodnější položku vykreslit poloprůhlednou barvou, případně vypočítat reprezentaci překrývajících se položek nebo vytvořit shluky a ty vizualizovat [19]. Průhlednost ale stále není dostatečná k rozeznání přesného počtu překrývajících se položek.

Někdy naopak některá data chybí nebo jsou chybná. Pak je vhodné chybný záznam z vizualizace odstranit, případně dosadit vlastní data, která jsou mimo zadaný obor hodnot, aby bylo možné rozlišit chybná data od správných [20].

Pokud potřebujeme vizualizovat odchylku dvou funkcí, můžeme lišící se interval zvýraznit. To můžeme provést například různou výškou (resp. tloušťkou) zvýraznění v závislosti na velikosti rozdílu. Případně můžeme využít různé barvy. Pro malé odchylky obarvit úsek světlými (případně poloprůhlednými) barvami a pro velké odchylky obarvit úsek intenzivními barvami.

V případě, že se jedná o vizualizaci číselných hodnot, je třeba dávat pozor na to, aby se operacemi na těchto číselných hodnotách nesnížila přesnost vizualizace nebo nedošlo ke zkreslení. Problém může nastat například při sčítání nebo násobení dvou desetinných čísel, pokud je jedno číslo řádově vyšší nebo nižší než druhé. To obecně souvisí s implementací čísel s desetinnou pohyblivou čárkou a jejich omezenou přesností. Proto je nutné zkontrolovat, že vizualizace odpovídá původním datům.

Pokud je vizualizace změněna, je lepší implementovat plynulou animaci přechodu mezi snímky, aby bylo snazší sledovat změnu pozice jednotlivých položek [21]. Nejjednodušší technika pro animování je lineární interpolace.

5 Volba technologií

Jako programovací jazyk jsem zvolil jazyk C# [22] a technologii .NET Framework [23]. Celá tato práce je vyvíjena pomocí vývojového prostředí Visual Studio 2015 od firmy Microsoft.

5.1 Jazyk C#

Jazyk C# je vysokoúrovňový objektově orientovaný programovací jazyk vyvinutý firmou Microsoft v roce 2000 a stále se aktivně rozvíjí. Tento jazyk byl přímo ovlivněn jazyky C++ a Java [24], a proto obsahuje podobné prvky a syntaxi.

Oproti jazyku C++ neumožňuje vícenásobnou dědičnost. Navíc je oproti C++ typově bezpečnější, protože implicitní přetypování lze provést, pouze pokud je považováno za bezpečné [25]. Dále neumožňuje používat globální proměnné, všechny proměnné a metody musí být definovány uvnitř tříd. Pro zpřehlednění zdrojových kódů nepotřebuje ani nepodporuje dopřednou deklaraci, na pořadí metod ve zdrojovém kódu nezáleží.

Mezi výhody patří možnost pracovat s čísly (a obecně s pamětí) na nízké úrovni [26]. Jazyk C# a technologie .NET Framework dále umožňuje jednoduše volat funkce z libovolných nativních a systémových knihoven. Tato funkcionalita je nutná pro analyzování nativních knihoven, které jsou přeloženy různými překladači pro porovnání výsledků.

Jazyk C# je zpravidla překládán do jazyka Common Intermediate Language (CIL) [27]. Jedná se o jazyk nižší úrovně zásobníkového typu, který je zpracováván virtuálním strojem. Ten ho pomocí just-in-time (JIT) kompilace překládá do kódu podporovaného přímo procesorem. Tento převod probíhá přímo za běhu programu.

5.2 Technologie .NET Framework

Rozhraní .NET Framework je prostředí potřebné pro běh aplikace psané v jazyce C#, které dále obsahuje sadu knihoven, kterou mohou aplikace využívat. Tyto knihovny obsahují sadu základních funkcí pro interakci s uživatelem, matematické funkce, kolekce, funkce pro vytváření procesů a vláken, funkce pro práci se soubory a proudy a nástroje pro tvorbu grafického uživatelského rozhraní [28].

Jedná se o volitelnou součást operačního systému Microsoft Windows a zpravidla je automaticky nainstalováno při instalaci operačního systému [29]. Poslední verze rozhraní .NET Framework je 4.6.2 [30], která je součástí operačního systému Microsoft Windows 10, dále je podporována všemi staršími verzemi systému až k verzi Windows 7. Pro vývoj aplikací na operační systém Windows XP je nutné použít verzi rozhraní .NET Framework 4.0 [31].

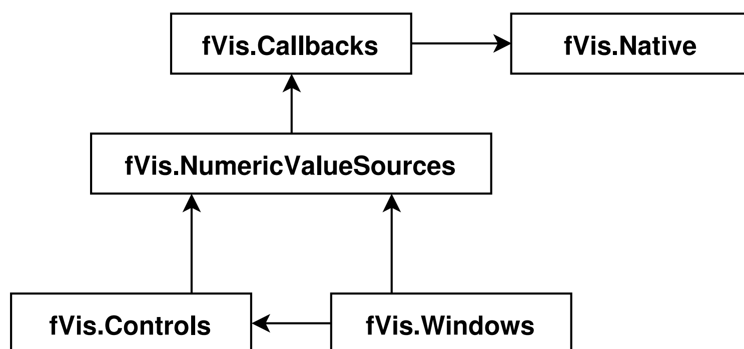
Mezi hlavní nevýhody patří zejména složitější spouštění těchto aplikací na jiných operačních systémech než Microsoft Windows. Na dalších operačních systémech, například Linux a Mac OS X, lze pro spouštění a překlad aplikací využít open-source řešení Mono [32], které však není firmou Microsoft přímo podporováno a neobsahuje všechny funkce dostupné v rozhraní .NET Framework. Stále se však aktivně vyvíjí a snaží se dále zvyšovat kompatibilitu s rozhraním .NET Framework [33].

Aplikace, které využívají jen základní funkce .NET Framework, lze pomocí řešení Mono bez úprav spustit. Aplikace využívající grafické uživatelské rozhraní je vhodnější přepracovat, aby využívalo grafickou knihovnu pro daný operační systém, protože každý operační systém podporuje odlišné grafické prvky.

6 Návrh aplikace

Aplikace by se měla skládat z jednotlivých menších součástí. Pro rozdělení aplikace na větší celky by bylo vhodné využít jmenné prostory, které jsou součástí jazyka C#. Ty by obsahovaly jednotlivé třídy.

Diagram závislostí jmenných prostorů je na obrázku 1. Dále budou popsány jednotlivé části diagramu. Všechny jmenné prostory budou součástí hlavního jmenného prostoru `fVis`.



Obrázek 1: Diagram jmenných prostorů.

Ve jmenném prostoru `fVis.NumericValuesSources` budou třídy, které budou poskytovat číselné hodnoty, které se poté budou vizualizovat. V tomto ohledu by měly dané třídy pracovat podobně jako standardní matematické funkce.

Abychom mohli ve výpočtech používat různé implementace matematických operátorů a funkcí, je vhodné vytvořit třídu, která by obsahovala ukazatele na jednotlivé matematické funkce a operátory. Jednotlivé implementace by od této třídy dědily a pozměnily by jednotlivé ukazatele tak, aby směřovaly na jejich vlastní implementace. Tato funkcionalita by byla součástí jmenného prostoru `fVis.Callbacks`.

Dále je nutné implementovat mechanismus pro načítání nativních knihoven a volání jejich funkcí. To by bylo využito pro načítání knihoven, které by obsahovaly různé implementace matematických funkcí. Tyto knihovny jsou většinou napsané v jazyce C nebo v jiném jazyce, který se překládá přímo

na strojový kód. Tato funkcionalita by byla součástí jmenného prostoru `fVis.Native`.

Jmenný prostor `fVis.Controls` by obsahoval všechny vytvořené prvky uživatelského rozhraní. Mezi ně bude patřit prvek, který bude vykreslovat samotnou vizualizaci, a prvek, který bude umožňovat manipulaci se seznamem matematických funkcí, které se budou vizualizovat a navzájem porovnávat.

Jmenný prostor `fVis.Windows` by obsahoval všechna okna a dialogy. Aplikace bude využívat pro uživatelské rozhraní sadu tříd Windows Forms, která je přímo součástí rozhraní .NET Framework.

Extenzní metody, kterou budou využívané různými částmi aplikace, budou součástí jmenného prostoru `fVis.Extensions`. Ty by měly hlavně rozšiřovat funkcionalitu datových typů, které jsou součástí technologie .NET Framework.

7 Popis implementace

7.1 Aritmetické výrazy

7.1.1 Zpracování aritmetického výrazu v infixové notaci

Pro tuto úlohu bylo nutné implementovat funkci pro zpracování aritmetických výrazů. Rozhodl jsem se využít algoritmus shunting-yard [34], který umožňuje převádět matematické výrazy z infixové notace (viz obrázek 2) do postfixové notace (viz obrázek 3) nebo do syntaktického stromu.

$$(1+2)\ /\ (3\cdot 4)$$

Obrázek 2: Ukázka výrazu v infixové notaci.

$$1\ 2\ +\ 3\ 4\ \cdot\ /$$

Obrázek 3: Ukázka výrazu v postfixové notaci.

Základ algoritmu funguje jako zásobníkový automat, který využívá dva zásobníky, proměnnou, do které ukládá druh poslední zpracované položky, a proměnnou, do které ukládá počet neukončených závorek. První zásobník obsahuje jednotlivé položky postfixové notace, označme ho jako `postfix`. Druhý zásobník dočasně obsahuje nalezené operátory, označme ho jako `operators`. Automat postupně prochází vstupní řetězec obsahující matematický výraz v infixové notaci.

Pokud narazí na číslici, tečku nebo +/− (v případě, že poslední položka byla operátor nebo závorka), pokusí se zpracovat položku jako číslo. Záznam o tomto číslu přidá do zásobníku `postfix`. Číslo je nutné specifikovat bez oddělovače tisíců, a pokud se jedná o desetinné číslo, je nutné uvést desetinnou tečku. Algoritmus dále umožňuje před číslem specifikovat + (resp. −) pro kladné (resp. záporné číslo).

Pokud narazí na písmeno, projde řetězec dál, dokud nenarazí na jiný druh znaku než písmeno. Z této posloupnosti znaku vytvoří nový řetězec, který porovná

se známými identifikátory funkcí. Pokud najde shodu, přidá záznam o funkci do zásobníku `operators`. Pokud se nejedná o známou funkci (viz tabulka 2), zkontroluje, zda se jedná alespoň o známou konstantu. Pokud se nejedná ani o známou konstantu (viz tabulka 3), označí řetězec jako proměnnou a přidá o ní záznam do zásobníku `postfix`. Toto umožňuje používat ve výrazu různé označení proměnné, ale pro vyhodnocení výrazu je nutné, aby všechny proměnné měly stejné označení. V opačném případě výraz vyhodnotí jako chybný. Zároveň zkontroluje, zda poslední položka byla funkce nebo ukončení závorky, v tomto případě výraz také vyhodnotí jako chybný. Algoritmus dále automaticky vkládá operaci násobení, pokud předchozí položka byla číslo. To urychluje psaní složitějších aritmetických výrazů.

Identifikátor	Popis
abs	Absolutní hodnota
sqrt	Druhá odmocnina
exp	Exponenciální funkce se základem **e**
ln	Přirozený logaritmus
log	Desítkový logaritmus
sin	Sinus
cos	Kosinus
tan	Tangens
asin	Arkus sinus
acos	Arkus kosinus
atan	Arkus tangens
sinh	Hyperbolický sinus
cosh	Hyperbolický kosinus
tanh	Hyperbolický tangens
round	Zaokrouhlení
floor	Zaokrouhlení směrem dolů
ceil	Zaokrouhlení směrem nahoru

Tabulka 2: Tabulka vybraných matematických funkcí z `math.h`.

Označení	Popis
pi	Ludolfovo číslo (π)
e	Eulerovo číslo

Tabulka 3: Tabulka známých konstant.

Pokud narazí na znak `(`, přidá záznam o závorce do zásobníku `operators` a navýší počet neukončených závorek. Zároveň zkontroluje, zda poslední položka byla číslo nebo ukončení závorky, v tomto případě výraz vyhodnotí jako chybný.

Pokud narazí na znak `)`, vybírá položky ze zásobníku `operators` a zároveň je vkládá do zásobníku `postfix` tak dlouho, dokud nenarazí na záznam o závorce. Zároveň sníží počet neukončených závorek. Také zkontroluje, zda poslední položka byla operátor, funkce nebo závorka, v tomto případě výraz vyhodnotí jako chybný.

Pokud narazí na znak `#`, předčasně ukončí zpracování řetězce. To umožňuje přidat na konec výrazu komentář nebo jinou zprávu.

Pokud se nejedná ani o jednu z těchto možností, pokusí se vyhledat znak v tabulce matematických operátorů (viz tabulka 4). Při nalezení shody, je nutné správně určit pořadí operátorů při převodu na postfixovou notaci. Algoritmus postupně vybírá položky ze zásobníku `operators`, dokud nenarazí na konec zásobníku nebo záznam o závorce. Pokud narazí na záznam o funkci nebo operátoru, který má nižší prioritu než právě zpracovaný operátor, zastaví vybírání položek ze zásobníku, v opačném případě položku přesune ze zásobníku `operators` do zásobníku `postfix`. Nakonec do zásobníku `postfix` přidá záznam o právě zpracovaném operátoru. Také zkontroluje, zda poslední položka byla jiný operátor, funkce nebo závorka, v tomto případě výraz vyhodnotí jako chybný. Pokud se znak nenachází v tabulce matematických operátorů, vyhodnotí výraz také jako chybný.

Označení	Popis
+	Sčítání
-	Odčítání
*	Násobení
/	Dělení
^	Mocnění
%	Zbytek po dělení

Tabulka 4: Tabulka matematických operátorů.

Po zpracování celého řetězce algoritmus zkontroluje, že počet neukončených závorek je nula a poslední položka není operátor, funkce nebo závorka. V opačném případě výraz vyhodnotí jako chybný. Všechny položky ze zásobníku `operators` postupně vybere a vloží je do zásobníku `postfix`. Zásobník `postfix` v tento okamžik obsahuje jednotlivé položky v postfixovém pořadí.

7.1.2 Validace aritmetického výrazu v postfixové podobě

Po zpracování aritmetického výrazu dojde k rychlé validaci. Ta by však neměla být nutná, protože výraz by byl vyhodnocen jako chybný už při zpracování.

Validace probíhá tak, že se položky v postfixovém pořadí překopírují do pole a postupně se prochází. V případě, že algoritmus narazí na položku obsahující operátor, pokusí se před ní najít dvě položky, které obsahují číselnou hodnotu nebo proměnnou. Tyto dvě položky vymaže a položku s operátorem přepíše na položku s číselnou hodnotou, ale nevyhodnocuje samotný výsledek. Pokud před operátorem dvě požadované položky nenalezne, označí výraz jako chybný.

V případě, že narazí na funkci, pokusí se před ní najít pouze jednu položku obsahující číselnou hodnotu nebo proměnnou. Jinak dále pracuje stejně. Tímto způsobem projde celý postfixový výraz.

Na závěr algoritmus ověří, že v poli zůstala pouze jedna číselná položka. Ta by měla při následném výpočtu aritmetického výrazu obsahovat výsledek. Pokud v poli

nezůstala žádná položka nebo v něm zůstala více jak jedna položka, výraz se označí jako chybný. Stejně tak se výraz označí jako chybný v případě, že položka neobsahovala číslo ale nějaký jiný typ, například operátor nebo funkci.

V případě, že ani zde není výraz označen jako chybný, je tento výraz bezchybný a může být dále vyhodnocován. V opačném případě je vyhozena výjimka `fVis.NumericValueSources.SyntaxException`. Výjimka obsahuje původní řetězec s aritmetickým výrazem, index znaku, který způsobil chybu, a druh chyby.

7.1.3 Vyhodnocování aritmetického výrazu

Vyhodnocování lze provést na zpracovaném aritmetickém výrazu opakovaně, bez nutnosti znovu výraz zpracovat. Všechny případné chyby aritmetického výrazu byly odhaleny už v předchozích dvou fázích, proto není nutné aritmetický výraz dále kontrolovat. Vyhodnocení probíhá podobně jako validace. Postfixový výraz opět překopíruje do pole, se kterým poté manipuluje. Položky se postupně prochází.

V případě, že algoritmus narazí na položku obsahující operátor, najde před ní dvě položky obsahující konstantu nebo proměnnou. V tomto okamžiku za proměnnou dosadí zvolené číslo. Obě tyto položky bude považovat za operandy a vypočítá výsledek této operace, který uloží do pole místo původní položky obsahující tento operátor. Položky, které obsahují dvě použitá čísla, se z pole odstraní.

V případě, že algoritmus narazí na položku obsahující funkci, najde před ní jednu položku obsahující konstantu nebo proměnnou. V tomto okamžiku za proměnnou dosadí zvolené číslo. Tuto položku bude považovat za vstupní hodnotu funkce a vypočítá výstupní hodnotu, kterou uloží do pole místo původní položky obsahující tuto funkci. Položku, která obsahovala vstupní hodnotu, z pole odstraní.

Po vyhodnocení všech operátorů a funkcí zbyde v poli pouze jediná položka s výsledkem. Tento výsledek je poté vrácen.

7.2 Výpočet matematických funkcí

Aby bylo možné využívat různé knihovny pro výpočet matematických funkcí, aplikace využívá systém ukazatelů na funkce, konkrétně v jazyce C# je toto realizováno pomocí delegátů.

Po zpracování aritmetického výrazu se znovu postupně projdou jednotlivé položky postfixové notace a ke každému operátoru nebo funkci se přiřadí funkce, která se při vyhodnocování výrazu zavolá. Pro snazší manipulaci s těmito funkcemi se využívá vlastní abstraktní třída `OperatorCallbacks`, která poskytuje matematické konstanty a ukazatele na všechny dostupné funkce pro výpočet. Také specifikuje, jaké vstupní a výstupní parametry tyto funkce mají.

7.2.1 Vestavěné funkce

V základu aplikace obsahuje třídu `DotNetOperatorCallbacks`, která dědí od abstraktní třídy `OperatorCallbacks` a která přímo volá matematické funkce dostupné prostředím .NET Framework, konkrétně statické metody ve třídě `System.Math`. Dále jsem pro porovnání přesnosti datových typů `double` a `float` implementoval třídu `DotNetFloatOperatorCallbacks`, která využívá stejné metody, ale před samotným výpočtem převádí vstupní hodnoty na datový typ `float`, stejnou operaci provádí i s výsledkem. To simuluje nižší přesnost tohoto datového typu.

7.2.2 Nativní knihovny

Jazyk C# také umožňuje volat funkce z nativních knihoven. Toho využívá poslední třída `NativeOperatorCallbacks`, která načítá nativní knihovny a volá funkce v nich obsažené. Protože je aplikace tvořena výhradně pro operační systém Microsoft Windows, využívá systémové funkce tohoto operačního systému, a proto dokáže pracovat pouze na tomto operačním systému. Pomocí atributu `DllImport` [35] aplikace volá systémovou funkci `LoadLibraryEx` [36], která načte libovolnou nativní knihovnu podle zadané cesty do adresního prostoru aplikace. Jako poslední

parametr lze specifikovat způsob načtení knihovny. V tomto případě je nejvhodnější využít hodnotu `LOAD_WITH_ALTERED_SEARCH_PATH`, kdy je cesta ke knihovně specifikována pomocí absolutní cesty. Pak se nemůže stát, že by operační systém omylem načetl jinou knihovnu, než bylo požadováno. Poté pomocí stejného atributu aplikace volá systémovou funkci `GetProcAddress` [37], která vrátí adresu exportované funkce. V případě, že se funkce podle zadaného jména v knihovně nenachází, zmíněná funkce vrátí hodnotu `NULL`. Pokud daná knihovna už nebude potřebná, je možné zavolat systémovou funkci `FreeLibrary` [38], která knihovnu z adresního prostoru odstraní. To však není bezpodmínečně nutné, protože operační systém Microsoft Windows tuto operaci provádí automaticky při ukončování aplikace.

Aby bylo možné exportovanou funkci (nebo jinou funkci nespravovanou prostředím .NET Framework) zavolat, musí se ukazatel na funkci nejprve převést na delegáta určitého typu pomocí statické metody `System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer` [39]. Tato metoda vyžaduje jako parametry adresu funkce a typ delegáta, který musí mít stejné parametry a typ návratové hodnoty jako původní nespravovaná funkce. Navíc je nutné sjednotit volací konvence nespravované funkce a delegáta v prostředí .NET Framework. V jazyce C# se u delegáta specifikuje volací konvence pomocí atributu `UnmanagedFunctionPointer` [40]. Pro zvýšení kompatibility je vhodné zvolit volací konvenci `cdecl` [41], tato volací konvence je často využívána překladači jazyka C. Volba volací konvence platí pouze pro platformu x86, neboť ostatní dostupné platformy (x86-64 a ARM) využívají vlastní volací konvence, které se nedají měnit, a prostředí .NET Framework zmíněný atribut na těchto platformách ignoruje. Při překladu nativní knihovny je potřeba dbát na to, aby překladač vygeneroval u exportované funkce opět volací konvenci `cdecl`, případně nastavit překladač tak, aby vytvořil nativní knihovnu pro platformu x86-64.

Dále je nutné dávat pozor na to, aby aplikace a nativní knihovna byla přeložena pro stejnou platformu, konkrétně 64-bitový operační systém Microsoft Windows umožňuje spouštět 32-bitové i 64-bitové aplikace, ale 64-bitové aplikace

neumožňují načíst 32-bitové knihovny, to platí i obráceně, tzn. 32-bitové aplikace nenačtou 64-bitové knihovny. Operační systém Microsoft Windows tento problém řeší tím, že 64-bitová verze operačních systémů obsahuje 32-bitové i 64-bitové verze systémových knihoven, a proto už v základu umožňuje spouštět 32-bitové i 64-bitové verze aplikací.

Aby aplikace dokázala rozpoznat, že se opravdu jedná o knihovnu obsahující matematické funkce připravené pro tuto aplikaci, musí každá tato knihovna obsahovat exportovanou funkci `get_extension_info`, která vrací strukturu `extension_info` specifikující název a verzi knihovny, viz výpis 2. Tento popis knihovny (případně použité sady matematických algoritmů nebo překladače) je pouze informativní, aby mohl uživatel snadněji rozlišit jednotlivé implementace. Dále může obsahovat libovolný výčet funkcí z tabulky 2, podle toho, které funkce daná implementace obsahuje nebo které chceme testovat. To samé oplatí i o operátorech (viz tabulka 4). Deklarace jednotlivých funkcí jsou vypsány v tabulce 5. Knihovna musí pro správný běh aplikace přesně dodržet názvy funkcí, počet a typy argumentů i typ návratové hodnoty. Pokud knihovna některou z těchto funkcí nebo operátorů neobsahuje, aplikace za ní automaticky dosadí ekvivalent funkce nebo operátoru, který se nachází v prostředí .NET Framework. Proto lze v aritmetických výrazech využívat i funkce, které nejsou ve vybrané matematické knihovně dostupné. Zároveň jsou chybějící funkce a operátory u každé knihovny aplikací evidovány, aby mohl být tento nedostatek případně reflektován v grafickém uživatelském rozhraní.

```
typedef struct {
    char* library_name;

    unsigned short version_major;
    unsigned short version_minor;
    unsigned short version_build;

    compiler_flags flags;
} extension_info;
```

Výpis 2: Struktura specifikující název a verzi matematické knihovny.

Prostředí .NET Framework umožňuje vytvořit spustitelnou aplikaci pro 32-bitový i 64-bitový režim z jednoho zdrojového kódu. Proto je také aplikace distribuována v obou těchto verzích, aby bylo možné načítat knihovny, které je možné sestavit pouze v jednom ze zmíněných režimů. Potom je nutné knihovnu nahrát do jedné ze složek, podle toho zda je sestavena pro 32-bitový režim (x86) nebo pro 64-bitový režim (x64). To ale neřeší problém, kdy je například nutné porovnat 64-bitovou knihovnu s 32-bitovou knihovnou. Princip jednoho z možných řešení popisuje Franz Wimmer [42], kdy ze 64-bitové aplikace volá 32-bitovou knihovnu. 64-bitový proces vytvoří vedlejší 32-bitový proces a vytvoří mezi nimi spojení pomocí dvou rour (anglicky pipe). Nově vytvořený proces čeká na připojení rodičovského procesu. Informace o funkci, kterou chce aplikace zavolat, a vstupní parametry jsou (pomocí binární serializace) přeneseny přes rouru do nově vytvořeného procesu. Ten data přečte, potřebnou funkci zavolá a výsledek pošle rourou zpět rodičovskému procesu.

Aby bylo možné toto řešení použít, bylo provedeno několik úprav. Nejprve bylo nutné přidat kritickou sekci v části, kde se přistupuje k rourám. V případě, že se knihovnu pokoušelo volat několik vláken zároveň, docházelo k různým chybám synchronizace. Dále musela být přidána funkcionalita, která by dokázala ověřit, zda funkce podle zadaného jména v cílové knihovně existuje bez toho, aby se funkce vykonala. To je využito při ověřování, které matematické funkce knihovna obsahuje.

Poslední chybějící funkce byla možnost přečíst textový řetězec, který knihovna vrátila jako ukazatel. Oba zmíněné procesy (64-bitový i 32-bitový) mají oddělené adresní prostory, a proto vrácený ukazatel v rodičovském procesu neplatí. Ukazatel je nutné přenést pomocí roury do 32-bitového procesu, který řetězec přečte a přepošle ho rourou zpět do rodičovského procesu.

Toto řešení je však několikanásobně pomalejší než v případě přímého přístupu. Navíc, vzhledem k tomu, že využívá pouze dvě roury pro komunikaci, neumožňuje plnohodnotný paralelní přístup (viz část o kritické sekci). Pro porovnání jednotlivých matematických knihoven je ale dostačující.

Označení	Deklarace funkce
pi	`double constant_pi();`
e	`double constant_e();`
+	`double operator_add(double x, double y);`
-	`double operator_subtract(double x, double y);`
*	`double operator_multiply(double x, double y);`
/	`double operator_divide(double x, double y);`
^	`double operator_pow(double x, double y);`
%	`double operator_remainder(double x, double y);`
abs	`double operator_abs(double x);`
sqrt	`double operator_sqrt(double x);`
exp	`double operator_exp(double x);`
ln	`double operator_ln(double x);`
log	`double operator_log(double x);`
sin	`double operator_sin(double x);`
cos	`double operator_cos(double x);`
tan	`double operator_tan(double x);`
asin	`double operator_asin(double x);`
acos	`double operator_acos(double x);`
atan	`double operator_atan(double x);`
sinh	`double operator_sinh(double x);`
cosh	`double operator_cosh(double x);`
tanh	`double operator_tanh(double x);`
round	`double operator_round(double x);`
floor	`double operator_floor(double x);`
ceil	`double operator_ceil(double x);`

Tabulka 5: Tabulka mapování položek na knihovní funkce.

7.3 Vizualizace matematických funkcí

Implementace vykreslování matematických funkcí se nachází ve třídě `Graph`. Třída umožňuje vykreslit jednu nebo více funkcí. Každá z těchto funkcí je barevně odlišená.

7.3.1 Posunutí a přiblížení

Třída využívá dvě proměnné typu `long`, ve kterých je uložené posunutí osy x a y. Jedná se o celočíselné proměnné, které udávají velikost posunutí v pixelech. Tato volba má své výhody i nevýhody. Mezi výhody patří fakt, že celá čísla na rozdíl od čísel s pohyblivou desetinnou čárkou neztrácí při vyšších hodnotách přesnost. To by způsobovalo trhání při posouvání zobrazení daleko od osy x (resp. y). Mezi nevýhody patří fakt, že se velikost posunutí musí přepočítávat při každé změně přiblížení. To je provedeno tak, že se posunutí nejprve vydělí původní hodnotou přiblížení a poté vynásobí novou hodnotou přiblížení. Tento přepočet může způsobit mírnou nepřesnost, protože je hodnota posunutí nejprve převedena na číslo s pohyblivou desetinnou čárkou (`double`), poté jsou provedeny zmíněné operace a výsledná hodnota je opět převedena na celé číslo (`long`). Odchylka je ale tak malá, že při běžném používání nedochází ke skokům nebo jiným vadám. Maximální velikost posunutí od osy je navíc limitována maximální hodnotou datového typu `long` – to je přibližně $9 \cdot 10^{18}$ pixelů.

Aktuální přiblížení vizualizace je uloženo v proměnné typu `double`, aby bylo možné vizualizaci libovolně přibližovat i oddalovat. Snížená přesnost tohoto datového typu při vyšších hodnotách nijak nenarušuje fungování aplikace, protože uživatel většinou využívá přibližování pomocí násobků nebo případně přiblížení na určitou oblast. Obě zmíněné operace jsou dostatečně přesné i při vyšších hodnotách přiblížení. Hodnota proměnné udává přiblížení zároveň pro osu x i pro osu y. Přiblížení rovno jedné zobrazí funkci v původním měřítku. Přiblížení rovno deseti zvětší všechny hodnoty na desetinásobek.

Změna měřítka pomocí kolečka myši je prováděna násobením (resp. dělením) hodnotou $1{,}4$. Zároveň jsou upraveny hodnoty posunutí, aby přibližování probíhalo ve směru ke kursoru myši. Při přiblížení na určitou oblast (v případě, že uživatel drží pravé tlačítko myši) je nová hodnota přiblížení vypočtena následovně:

$$poměrX = \frac{šířka\ okna}{šířka\ vybrané\ oblasti}$$

$$poměrY = \frac{výška\ okna}{výška\ vybrané\ oblasti}$$

$$přiblížení = přiblížení \cdot \min(poměrX, poměrY)$$

Proměnná *přiblížení* poté obsahuje výslednou hodnotu přiblížení na zvolenou oblast. Minimum dvou vypočtených hodnot je zde proto, aby byla zobrazená celá oblast výběru i v případě, že je zvolená oblast v jiném poměru stran než velikost okna. Hodnoty posunutí jsou upraveny tak, aby byla přiblížená oblast ve středu obrazovky.

7.3.2 Režimy vykreslování

Třída obsahuje dva režimy vykreslování grafu. Jeden režim se využívá v situacích, kdy nelze zobrazit v jednu chvíli všechny hodnoty v zadaném intervalu, to znamená, že plocha pro vykreslování je menší než množství hodnot intervalu, který chce uživatel zobrazit. Vykreslení probíhá tak, že se pro každý pixel osy x najde původní hodnota x. To lze provést tak, že nejprve odečteme posunutí osy x a poté vydělíme aktuálním přiblížením. Tuto hodnotu dosadíme do funkce, kterou chceme vykreslit. Vypočtenou hodnotu y nejprve vynásobíme hodnotou -1, to způsobí, že kladné hodnoty budou zobrazeny směrem nahoru od osy y. Poté ji vynásobíme přiblížením a přičteme posunutí osy y. Výsledkem je hodnota y pixelu. Z původní hodnoty x pixelu a nově vypočtené hodnoty y sestrojíme dvojici. Takto vytvořené dvojice použijeme jako souřadnice, mezi kterými vykreslíme čáry.

Vypočtené hodnoty y jsou rovnou porovnávány mezi jednotlivými vybranými funkcemi. Pokud se některá z hodnot liší, vytvoří se záznam ve slovníku, který tyto

hodnoty uchovává. Klíč slovníku je původní hodnota x, uložená hodnota ve slovníku je vlastní struktura obsahující minimální a maximální vypočtenou hodnotu y a číslo záznamu, které se inkrementuje při každém přidaném záznamu. Slovník je zvolen proto, aby nebylo možné uložit dva záznamy jedné hodnoty x. Číslo záznamu je využito při odstraňování starých položek při překročení stanoveného limitu. Po překročení je odstraněno 0,5 % těchto záznamů, ty jsou nalezeny tak, že se prochází celý slovník a do vedlejšího pole se ukládají hodnoty s nejnižším číslem záznamu. Pole se při každém přidání položky seřadí od nejvyšší hodnoty po nejnižší. Pak není nutné procházet celé pole, ale je možné provést porovnání pouze s prvním prvkem pole, případně prvek nahradit záznamem s nižším číslem a pole opět seřadit.

Slovník s informacemi o rozdílech je následně využit pro vizualizaci rozdílů. Aplikace umožňuje přepínat mezi dvěma režimy zobrazení rozdílů. První režim zobrazuje čárou konstantní velikosti několika pixelů. To lze využít pro rychlé nalezení úseku, ve kterém se funkce liší.

Druhý režim upravuje délku čáry podle velikosti rozdílů v daném místě. To se provádí tak, že se nejprve projde celý slovník s rozdíly, algoritmus se pokusí namapovat hodnotu x na pixel zobrazení. Pokud se položka nachází ve výřezu, který chce uživatel právě zobrazit, spočítá se, kolik desetinným čísel se nachází mezi nejnižší a nejvyšší hodnotou rozdílu. To lze jednoduše provést tak, že se obě čísla přetypují na celočíselný datový typ o stejné velikosti (tzn. `double` na `long`) a spočítá se rozdíl těchto čísel. Všechny rozdíly, které se mapují na stejný pixel, se sečtou a uloží se. Tyto hodnoty je vhodné ukládat do pole, kde každý pixel odpovídá prvku v poli, protože předem známe šířku zobrazení. Před zobrazením se pro přehlednost hodnota v poli nejprve zlogaritmuje. Vzdálenost čísel totiž může nabývat hodnot od jednotek až po miliardy.

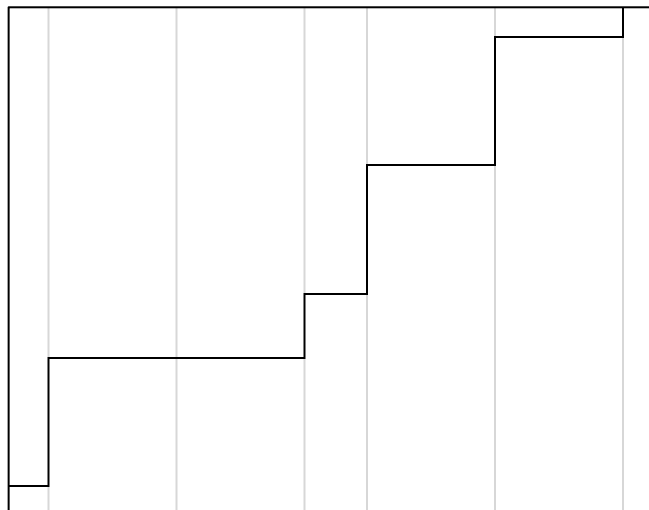
Druhý režim vykreslování grafu je určen pro vykreslování funkcí při největším přiblížení. Přepínání mezi těmito režimy provádí aplikace sama, pokud detekuje, že se alespoň 3 po sobě jdoucí pixely mapují na stejnou hodnotu x. Před zobrazením algoritmus nejprve nalezne nejnižší a nejvyšší hodnotu x, která se vejde to výřezu

zobrazení. Počátek nalezne tak, že od nuly odečte posunutí osy x a vydělí přiblížením. Podobně nalezne i konec a to tak, že šířky zobrazení odečte posunutí osy x a vydělí přiblížením. Obě hodnoty přetypuje na celočíselný datový typ o stejné velikosti. Všechna čísla mezi těmito hodnotami (včetně) projde cyklem. Iterační proměnná obsahuje celé číslo z intervalu mezi zmíněnými dvěma hodnotami, a proto je možné proměnnou přetypovat zpět na číslo s pohyblivou desetinnou čárkou. Abychom dostaly zpět hodnotu x pixelu, je nutné iterační proměnnou vynásobit přiblížením a přičíst posunutí osy x. Přetypovaná iterační proměnná je dále použita jako vstupní hodnota pro funkci, kterou potřebujeme vykreslit. Výstupní hodnotu funkce musíme opět nejprve vynásobit hodnotou -1 a přiblížením, a poté přičíst posunutí osy y. Takto opět získáme dvojice, které můžeme interpretova jako souřadnice a spojit je čárami.

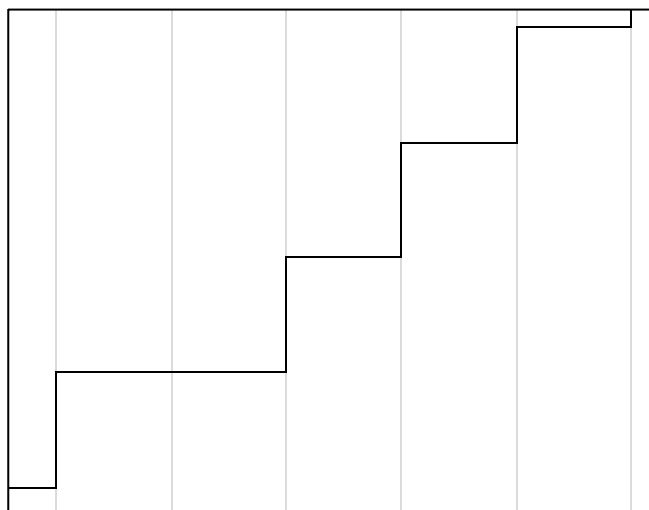
Pro zmíněné výpočty ale nelze použít standardní aritmetické operace poskytované datovým typem `double`. Hodnota přiblížení je v tomto případě natolik vysoká, že způsobuje při násobení znatelné zkreslení vizualizace. Určité zkreslení může přidat i hodnota posunutí os. Takto zkreslená vizualizace je znázorněna na obrázku 4. Svislé šedé čáry, oddělující jednotlivé hodnoty na ose x, by měly být od sebe vzdálené stejně daleko. Na tomto úseku se však znatelně projevila chyba kvůli snížené přesnosti při výpočtu. To samé platí i o výšce jednotlivých úrovní (na ose y), ty jsou od sebe navzájem vzdáleny opět o jeden bit, ale vizualizace opět neodpovídá realitě.

Abychom eliminovali výše zmíněné nepřesnosti, je nutné pro výpočet vizualizace využít knihovnu, která umožňuje počítat s vyšší přesností než je přesnost datového typu `double`. Jedna z knihoven, která toto umožňuje, se nazývá MPIR [43]. Knihovna je založena na knihovně GNU Multi Precision, stále udržována Williamem Hartem a vydána pod licencí GPL-3.0. Knihovna obsahuje funkce pro sčítání, odčítání, násobení, dělení a mocnění desetinných čísel s libovolnou přesností. Knihovna dále obsahuje funkce pro počítání se zlomky a celočíselnými hodnotami. Pro použití knihovny MPIR v jazyce C# existuje knihovna Mpir.NET [44] obsahující metody, které umožňují volat funkce zmíněné

knihovny. Tato knihovna je využívána extenzními metodami ve třídě `fVis.Extensions.DoubleExtensions`. Tyto metody umožňují počítat různé kombinace aritmetických operací, které jsou využívány vizualizací, konkrétně se jedná o odečtení a vydělení, vynásobení a přičtení. Metody vyžadují jako typy vstupních hodnot typ `double` nebo `long` v závislosti na tom, zda je daná vstupní hodnota celočíselná nebo není. Před výpočtem se vstupní hodnoty převedou do reprezentace využívané knihovnou MPIR, provedou se potřebné operace touto knihovnou a výsledek se opět převede na typ `double`, případně na celočíselný typ `int`, pokud je prováděno mapování na pixel zobrazení.



Obrázek 4: Vizualizace vypočítaná pomocí operací poskytovaných datovým typem `double`.



Obrázek 5: Vizualizace vypočtená pomocí knihovny MPIR.

Opravená vizualizace je znázorněna na obrázku 5. Po této úpravě jsou všechny svislé šedé čáry stejně daleko od sebe. Vzdálenosti mezi úrovněmi jsou také shodné.

Pro takto vysoké přiblížení je vhodné pro jednotlivé hodnoty v grafu zobrazit i jejich přesné číselné hodnoty. Metoda `ToString`, poskytovaná datovým typem `double`, nedokáže při převodu na textový řetězec zachovat původní hodnotu čísla. Vrací řetězec, který obsahuje pouze několik prvních číslic desetinné části. To je, hlavně pro velmi malá čísla, nedostačující.

Třída `DoubleConverter`, vytvořená Jonem Skeetem [45], tento problém řeší. Nejprve rozloží zadané desetinné číslo na část se znaménkem, část s exponentem a část s mantisou. Dále zkontroluje, zda se jedná o normalizované nebo denormalizované číslo. V případě denormalizovaného čísla se exponent nastaví na 1, tím se mantisa nebude dále mocnit. V případě normalizovaného čísla se nastaví nejvýznamnější bit mantisy na 1. Poté od exponentu odečteme posunutí exponentu (1023) a počet bitů, ve kterých je uložena mantisa (52). Počet bitů odečítáme, protože budeme dále s číslem pracovat, jako by bylo ve tvaru "$m,0$", místo původního "$0,m$". Na závěr se povede normalizace mantisy, aby obsahovala liché číslo. Nyní je možné na mantisu aplikovat exponent. Kladný exponent značí, kolikrát je nutné mantisu vynásobit číslem 2. Záporný exponent značí, kolikrát je nutné mantisu vydělit číslem 2. Před vrácením výsledku ve formě textového řetězce je před řetězec přidáno odpovídající znaménko.

K přesným číselným hodnotám je vhodné přidat i jejich bitovou reprezentaci. Využívá se k tomu extenzní metoda `ToBits`. Metoda převádí číselnou hodnotu na textový řetězec tak, že nejprve přetypuje číslo na celočíselný datový typ o stejné velikosti (tzn. `double` na `long`) a cyklem projde všechny bity desetinného čísla. Uvnitř cyklu, pomocí bitového posunu a bitového součinu s 1, získá hodnotu bitu, kterou přidá do výstupního řetězce. Pro přehlednost do řetězce přidá mezery oddělující znaménko od exponentu a exponent od mantisy.

7.3.3 Vykreslování vizualizace

Třída je určená pouze pro knihovnu Windows Forms a dědí od základního prvku `Control`. Tím lze snadno umístit do libovolného okna. Abychom mohli dané matematické funkce vykreslit, je potřeba překrýt metodu `OnPaint`. Ta je volána knihovnou pokaždé, když je nutné prvek překreslit. Voláním je předána instance třídy `System.Drawing.Graphics`, která umožňuje kreslit na plátno. Tím je v tomto případě oblast určená pro tento prvek uživatelského rozhraní.

Při každém překreslení je plátno nejdříve přemalováno bílou barvou. Poté jsou nakresleny osy a proběhne vykreslení samotných funkcí. Následně jsou vykresleny různé textové popisy. V případě, že uživatel drží pravé tlačítko myši, je navíc vykreslen obdélník značící oblast, ke které bude pohled zaměřen.

7.4 Vyhledávání rozdílů na intervalu

Před tím, než aplikace začne rozdíly vyhledávat, je důležité vypočítat počet hodnot, které budeme porovnávat, a odhadnout dobu běhu. Počet hodnot, které se nachází mezi dvěma čísly, získáme tak, že obě čísla přetypujeme na celočíselný datový typ o stejné velikosti (tzn. `double` na `long`) a spočítáme rozdíl těchto čísel. Aby aplikace dokázala určit odhad doby běhu, vypočítá hodnotu výrazu pouze pro menší počet hodnot, než je v požadovaném intervalu. Číslo je nutné zvolit tak, aby výpočet probíhal na dostatečně velkém vzorku dat a trval ideálně méně než sekundu, aby uživatel nemusel na odhad čekat. Po testování bylo zvoleno číslo 250 000. Na počítači s procesorem AMD FX-6300 (6× 4,2 GHz) a 8 GB RAM tento výpočet trvá zhruba 85 milisekund. Doba běhu výpočtu je zaznamenána. Následně je vydělena počtem položek ve vzorku a vynásobena skutečným počtem položek. V případě, že je vypočtená doba běhu větší než 30 sekund, je uživatel na tuto skutečnost upozorněn.

Vyhledávání rozdílů pracuje tak, že se pro každou hodnotu x ze zadaného intervalu vypočítají hodnoty y všech funkcí, které prohledáváme. Tyto hodnoty se navzájem porovnávají. V případě, že se jedna z nich liší, algoritmus zaznamená hodnotu x a nejvyšší a nejnižší hodnotu y. Tyto informace uloží do společného slovníku.

Samotný výpočet je prováděn na vedlejším vlákně, aby neblokoval běh uživatelského rozhraní. Vedlejší vlákno navíc pro paralelizaci vyhledávání využívá třídu `System.Threading.Tasks.Parallel`, která je součástí rozhraní .NET Framework. Tato třída obsahuje metodu `For`, která umožňuje paralelizovat cyklus. Třída se pak postará o to, aby vytvořila vhodný počet vláken, a každému vláknu přidělí část intervalu, který je potřeba zpracovat. Na konci každé iterace se nachází kritická sekce, ve které se zaznamená nalezený rozdíl do slovníku, a případně se aktualizují informace v dialogu. To mírně snižuje efektivitu paralelního zpracování.

Aby bylo možné sledovat průběh vyhledávání, byl vytvořen jednoduchý dialog `fVis.Windows.ProgressDialog`. Algoritmus kontroluje, kdy naposledy odeslal

dialogu informaci o počtu zpracovaných čísel. V případě, že je tato doba delší než 4 sekundy, aktualizuje informace v dialogu a zároveň zkontroluje, že uživatel nepožádal o přerušení operace. Pokud uživatel chce vyhledávání přerušit, algoritmus ukončí všechna vlákna a oznámí dialogu, že dokončil práci.

Algoritmus navíc stále přepočítává odhad času, který zbývá do dokončení prohledávání intervalu. Ten se vypočítá tak, že se nejprve vypočítá počet zpracovaných položek od doby, kdy se naposledy počítal odhad. Tuto hodnotu vydělíme počtem sekund, které uběhly od posledního počítání odhadu. Tím získáme okamžitou rychlost zpracování v položkách za sekundu. Poté vypočítáme počet zbývajících položek tak, že od celkového počtu odečteme počet již zpracovaných položek. Získané číslo vydělíme vypočtenou okamžitou rychlostí zpracování, tím dostaneme počet sekund, které zbývají do konce prohledávání.

7.5 Ukládání hodnot matematických funkcí do souboru

Další z funkcí aplikace je možnost uložit všechny hodnoty, které spadají do určitého intervalu, zvolené matematické funkce.

Aplikace ukládá data do souboru vlastního jednoduchého binárního formátu. Všechna čísla jsou ukládána jako little-endian. Aplikace pro ukládání všech dat využívá metody třídy `BinaryWriter`, která je obsažená v prostředí .NET Framework. V prvních 5 bitech je uložen textový řetězec (kódovaný v ASCII) obsahující `"fVis\0"`. Poté následuje textový řetězec (kódovaný v UTF-8) obsahující popis uložené funkce. Následuje aktuální posunutí osy x a posunutí osy y, které je uloženo jako datový typ `long`, a poté aktuální přiblížení, které je uloženo jako datový typ `double`. Tyto informace jsou zde uloženy proto, aby bylo po načtení možné lehce nalézt oblast, ve které se nachází uložené hodnoty. Dále následuje informace o počtu hodnot, která je uložena jako datový typ long. Zbytek souboru obsahuje pole dvojic. Každá dvojice představuje hodnotu x a k ní příslušnou hodnotu y. Tyto hodnoty jsou uloženy jako datový typ `double`.

Při načítání hodnot ze souboru aplikace nejprve přečte všechny uložené informace kromě samotných hodnot. Tím zjistí počet hodnot a najde místo v souboru, kde začíná zmíněné pole dvojic. Tyto informace předá třídě `fVis.NumericValueSources.FileDataSet`, která soubor s hodnotami znovu otevře v režimu memory-mapped file a tím namapuje celý soubor do adresního prostoru aplikace. Aplikace získá ukazatel na `byte`, který ukazuje na začátek souboru v paměti, a připočte k němu začátek pole dvojic. Tento ukazatel se přetypuje na ukazatel na vlastní strukturu obsahující hodnotu x a hodnotu y. Tak může k hodnotám přistupovat přímo, podobně jako v případě pole, a operační systém, který využívá mechanismus stránkování, se sám postará o to, aby nahrál data, která jsou právě využívána, ze souboru do paměti počítače.

Aby se třída chovala stejně jako matematická funkce, je nutné implementovat metodu, která pro vstupní hodnotu x vrátí hodnotu y. Třída musí v poli dvojic vyhledat tu dvojici, která má hodnotu x shodnou se vstupní hodnotou, a vrátit k ní

příslušnou hodnotu y. Dvojice jsou seřazeny podle hodnoty x od nejnižší po nejvyšší, proto je možné implementovat vyhledávání pomocí metody půlení intervalu. To v nejhorším případě sníží počet iterací z n (v případě, že bychom procházeli pole celé sekvenčně) na $\log_2 n$.

Dřívější verze aplikace využívala třídu `MemoryDataSet`, která nejprve načetla celý soubor do paměti. Poté se již chovala podobně jako třída `FileDataSet` a také využívala metodu půlení intervalu. Toto řešení mělo několik nevýhod. Soubor musel být nejdříve celý přečten, to při větších souborech mohlo trvat i několik minut, v závislosti na rychlosti pevného disku. Dále musel počítač disponovat dost velkou pamětí na to, aby mohl celé pole do paměti načíst. Navíc rozhraní .NET Framework neumožňuje alokovat jeden objekt (v tomto případě pole dvojic), které by zabíralo více než 2 GB paměti, a to i v případě 64-bitových operačních systémů. Jedno z možných řešení bylo využít jednoduchou třídu `BigArray` [46]. Tato třída alokuje několik menších polí, aby obešla limitaci rozhraní .NET Framework. Velikost všech polí musí být shodná a zároveň musí být mocninou čísla 2. Při přístupu k tomuto poli třída nejprve požadovaný index posune o $\log_2 n$ (kde n je velikost pole) bitů doprava. Tím získá číslo pole, ve kterém se požadovaná položka nachází. Index položky v novém poli získá tak, že aplikuje na původní index bitovou masku obsahující jedničky na prvních $\log_2 n$ pozicích.

7.6 Zobrazení seznamu funkcí v uživatelském rozhraní

Pro přehledné zobrazení seznamu všech zadaných funkcí musel být implementován vlastní prvek uživatelského rozhraní. Rozhraní .NET Framework a knihovna Windows Forms obsahuje pouze velmi omezený prvek ListView, který nepodporuje žádné možnosti formátování textu a hodí se pouze pro velmi jednoduché případy.

Prvek uživatelského rozhraní popisovaný v této kapitole se nachází ve třídě `fVis.Controls.ListView` a dědí od základního prvku `Control` poskytovaného knihovnou Windows Forms. Aby mohl zaznamenávat změny v seznamu, využívá pro ukládání jednotlivých položek třídu `ObservableCollection`, která implementuje rozhraní `INotifyCollectionChanged` a obsahuje událost `CollectionChanged`. Ta se volá pokaždé, když se přidá nebo odebere položka ze seznamu. Podobně fungují i jednotlivé položky seznamu. Implementují rozhraní `INotifyPropertyChanged`, které obsahuje událost `PropertyChanged`. Ta se volá pokaždé, když se změní některá z vlastností položky. Ostatní třídy, které tyto položky zpracovávaní mohou na tyto změny reagovat.

Třída vykresluje všechny položky postupně od první do poslední. Přeskakuje ty položky, které nejsou v aktuálním výřezu zobrazení. Tento prvek je navržen pro nízký počet položek (v řádu jednotek). Pokud by byl očekávaný počet položek v řádech tisíců, bylo by vhodnější použít například algoritmus na bázi půlení intervalu, aby vyhledal část seznamu, který se bude vykreslovat, místo sekvenčního přístupu. Třída vykresluje jednotlivé části na plátno pomocí třídy `System.Drawing.Graphics`.

Každá položka se skládá ze zaškrtávacího políčka. To je určené pro rychlé vypínání a zapínání jednotlivých funkcí bez potřeby je ze seznamu odstraňovat. Hned vedle je vykreslen čtverec s výplní, která značí barvu funkce ve vizualizaci. Barvu funkce lze změnit kliknutím na tento čtverec. Využívá se k tomu třída `ColorDialog`.

Následuje předpis funkce (případně jiný textový popis) a název vybrané matematické knihovny. Tato pole podporují jednoduché formátování textu, které je implementováno ve třídě `fVis.Misc.FormattedTextBlock`. Třída pracuje tak, že nejprve projde zadaný textový řetězec a rozdělí jej na úseky textu tak, že se snaží najít nejdelší úsek, který má stejné formátování. Ke každému úseku si uloží barvu a typ písma, pozici v textu a výšku textu. Tyto informace uloží do pole, aby bylo možné pro vykreslení pouze pole projít a jednotlivé úseky vykreslit, případně změřit velikost celého bloku.

Protože metody rozhraní .NET Framework neumožňují spolehlivé měření velikosti textu a následné vykreslení bez dalších automatických odsazení, bylo potřeba vytvořit třídu, která bude volat přímo funkce operačního systému, které tuto možnost obsahují. Nejprve je nutné získat od grafického kontextu popisovač kontextu zařízení pomocí metody `GetHdc`. Ten je následně použit pro volání funkcí operačního systému. Pro měření velikosti textu je použita funkce `GetTextExtentExPoint` [47]. Ta umožňuje specifikovat maximální šířku textu, vrací skutečnou šířku textu a index posledního znaku, který se do zadané maximální šířky vešel. Samotné vykreslení se provádí pomocí funkce `TextOut` [48], která umožňuje vykreslit text přesně na zadané pozici, bez dalších odsazení a mezer. Ani jedna z funkcí nepodporuje automatické ani ruční oddělování řádků, proto je nutné rozdělit řádky ručně při tvorbě jednotlivých úseků.

Tvorba jednotlivých úseků probíhá tak, že se projde celý řetězec znak po znaku. Pokud narazí na znak `\n (0x0A)`, ukončí aktuální řádek a začne skládat další úseky textu do nového řádku. Na konci každého řádku vyrovná všechny úseky tak, aby byly na stejné úrovni. Protože třída `ListView` využívá pouze jednořádkové texty, je zpracování automaticky zastaveno na konci prvního řádku.

Pokud třída `FormattedTextBlock` narazí na `\f (0x0C)`, zkontroluje, že další znak je `[`, a najde nejbližší výskyt znaku `]`. Mezi těmito znaky lze specifikovat formátování, které se aplikuje bezprostředně po této značce. Všechny možnosti formátování jsou vypsány v tabulce 6. Pokud je specifikován obrázek, při každém vykreslení je volán delegát `ImageResourceCallback`, který musí pokaždé vrátit

obrázek o stejné velikosti, protože pozice je vypočítána pouze jednou a nedošlo by k posunutí ostatních úseků textu.

Značka	Popis
B	Zapnout/vypnout tučné písmo
I	Zapnout/vypnout kurzívu
U	Zapnout/vypnout podtržení textu
S	Zapnout/vypnout přeškrtnutí
Rc	Nastavit výchozí barvu
Rs	Nastavit výchozí styl písma
- (pomlčka)	Zarovnat zbytek řádku doprava
image:<název>	Pomocí <název> lze specifikovat obrázek, který se vykreslí
0xRRGGBB	Změnit barvu (RGB) písma specifikovanou číslem v šestnáctkové soustavě

Tabulka 6: Všechny možnosti formátování třídy FormattedTextBlock.

Pro podporu posouvání výřezu ve třídě ListView směrem nahoru nebo dolu je využita komponenta VScrollBar, která je součástí knihovny Windows Forms. Ta přímo využívá prvek posuvníku operačního systému.

Sloupec obsahující aritmetický výraz je možné upravit pomocí dvojkliku myší. Zobrazené textové pole je komponenta TextBox, která je opět součásti knihovny Windows Forms. Textové pole je umístěno přesně přes oblast, kde se nacházel původní text, a je do něj načten text z proměnné příslušné položky. Po stisku klávesy Enter, případně kliknutím mimo textové pole, je upravený text nahrán zpět do proměnné položky a textové pole je skryto. Při stisku klávesy Escape je upravený text zahozen a textové pole je opět skryto.

Prvek je navržen tak, aby se vzhledem snažil co nejvíce napodobit vzhled operačního systému Microsoft Windows. Využívá pro to třídy jmenného prostoru System.Windows.Forms a System.Windows.Forms.VisualStyles. Pro vykreslování zaškrtávacího políček využívá třídu CheckBoxRenderer, která následně volá služby operačního systému, aby políčko vykreslil, případně změřil velikost políčka. Podobně pracuje i třída VisualStyleRenderer, která však dovoluje vykreslit jakýkoliv prvek uživatelského rozhraní, který je v operačním systému definovaný. Tak dokáže navržená třída vykreslit ohraničení výběrů i zaškrtávací políčka stejně

jako aplikace, které jsou přímo obsaženy v operačním systému Microsoft Windows. V některých případech může být vykreslování prvků operačním systémem zakázáno, nejčastěji se tak stane, když uživatel vybere klasické schéma. Poté musí aplikace všechny elementy vykreslit ručně, nejčastěji pomocí jednoduchých obrazců, jako například obdélník.

8 Testování

Během vývoje byly průběžně vytvářeny jednotkové testy pro jednotlivé třídy. Byla k tomu využita knihovna MSTest, která je integrovaná přímo do vývojového prostředí Microsoft Visual Studio. Toto vývojové prostředí též obsahuje nástroje pro analýzu pokrytí kódu. Jednotkovými testy jsou z větší části pokryty všechny třídy kromě prvků uživatelského rozhraní.

Jedna z nejobsáhlejších tříd této aplikace je třída pro zpracování a výpočet aritmetických výrazů. Tato třída je také kompletně pokryta 34 jednotkovými testy. Je zde testována většina možných platných vstupních tříd. Dále se testuje, zda algoritmus správně vyhazuje výjimky v případě zadání neplatných vstupních řetězců. To je provedeno pomocí atributu `ExpectedException`, který se připíše ke konkrétní metodě testu, který má končit výjimkou. Jeden z jeho parametrů je typ výjimky, která se očekává. Nakonec jsou zde testy, které kontrolují, že algoritmus dokáže správně dosazovat konstanty a proměnné do funkcí. Také testuje schopnost algoritmu správně vyhodnotit priority operátoru, které jsou ve výrazu použity, a porovnává výsledky vrácené algoritmem s očekávanými výsledky.

Třída `MemoryDataSet` je testována 6 jednotkovými testy. Testuje se, zda třída správně vrací očekávané hodnoty, pokud jako vstupní hodnot použijeme hodnotu mimo interval, ze začátku a z konce intervalu, nebo hodnotu uvnitř intervalu. Jednotkové testy v tomto případě pomohly odhalit chybu, kdy třída vracela nesprávné výsledky pro hodnoty na konci intervalu.

U tříd `DotNetOperatorCallbacks` a `DotNetOperatorCallbacks` jsou testovány jednotlivé matematické funkce, zda odpovídají těm původním, které se nachází ve třídě `System.Math`. Celkem se jedná o 23 jednotkových testů pro každou třídu. Třída `NativeOperatorCallbacks` je testována s pomocí knihovny Sun libmcr. Je zde testováno, že výsledky získané třídou přímo odpovídají hodnotám, které knihovna vrací. Také je testováno, že třída dokáže správně nahradit funkce, které v matematické knihovně chybí. Navíc je testováno, že instance vrací správný

název knihovny a že správně detekuje chybějící funkce. Celkem se jedná o 25 jednotkových testů.

Třída `NativeLibrary` byla pokryta 8 jednotkovými testy. Testuje se schopnost volat funkce z nativních knihoven. Také se testuje, zda třída dokáže správně detekovat neexistující funkce.

Dále jsou testovány všechny extenzní metody. To zahrnuje i metody, které převádí desetinné číslo datového typu `double` na řetězec obsahující jednotlivé bity, přesně jak je číslo uloženo v paměti počítače, a také metodu, která převádí desetinné číslo na jeho přesnou textovou reprezentaci. To je otestováno 15 jednotkovými testy. Testují nejrůznější případy, například případ, kdy je vstupní číslo kladné nebo kdy je záporné.

Uživatelské rozhraní bylo otestováno ručně tak, že bylo provedeno několik testovacích scénářů. První scénář spočíval v tom, že bylo do aplikace přidáno několik stejných aritmetických výrazů využívající stejné matematické knihovny. Poté bylo otestováno, že lze jednotlivým výrazům přiřazovat různé knihovny. Bylo vyzkoušeno vypínání a zapínání jednotlivých funkcí pomocí zaškrtávacího políčka. Poté bylo zkontrolováno, že zvýrazněné rozdíly opravdu odpovídají místům s rozdílnými hodnotami. Nakonec byl upraven jeden z výrazů a bylo zkontrolováno, že se změnily vyznačené rozdíly. Bylo otestováno posouvání plátna a možnost přibližování.

Druhý scénář testoval možnost exportovat a importovat data. Nejprve byla přidána alespoň jedna funkce a byla vybrána tak velká oblast, aby zabírala na disku okolo 4 GB místa, a bylo zahájeno exportování. Po dokončení exportování bylo zkontrolováno, že odpovídající soubor existuje, a data byla importována zpět do aplikace. Bylo otestováno, že se původní data shodují s těmi importovanými.

Další scénář testoval možnost přerušit export dat před dokončením. Nejprve byla přidána alespoň jedna funkce a byla vybrána tak velká oblast, aby zabírala na disku okolo 4 GB místa, a bylo zahájeno exportování. Po několika sekundách

bylo exportování uživatelem přerušeno. Bylo ověřeno, že se exportování do několika sekund přeruší a výsledný soubor neexistuje.

Poslední scénář ověřuje vyhledávání na intervalu. Nejprve byly přidány dvě stejné funkce. Byl vybrán tak velký interval, aby zpracování trvalo okolo 2 minut, a bylo spuštěno vyhledávání rozdílů. Po dokončení bylo ověřeno, že nebyl nalezen žádný rozdíl. Poté byla jedna z funkcí upravena, tak aby obě funkce vracely pro stejnou hodnotu x rozdílné hodnoty y, a bylo opět spuštěno vyhledávání. Po dokončení bylo zkontrolováno, že byl nalezen odpovídající počet rozdílů.

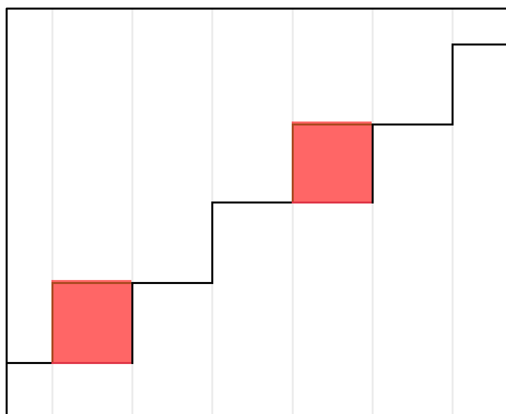
Aplikace byla testována na počítači s procesorem AMD FX-6300 (6× 4,2 GHz) a 8 GB RAM. Byla změřena doba běhu při vyhledání rozdílů dvou funkcí sinus. Výsledky jsou zaznamenány v tabulce 7.

Počet hodnot	Čas
20 000 000	0:00:29
40 000 000	0:00:56
60 000 000	0:01:28
80 000 000	0:01:54

Tabulka 7: Rychlost porovnávání dvou funkcí.

9 Závěr

Popisovaná aplikace dokáže správně zpracovat libovolný aritmetický výraz a s pomocí matematických knihoven dokáže výraz vyhodnotit. Zadanou množinu aritmetických výrazů dokáže vykreslit s libovolným přiblížením. Při nejbližším přiblížení je možné zkoumat jednotlivé bity a hledat rozdíly mezi zadanými funkcemi, případně porovnávat různé implementace matematických funkcí z různých knihoven.



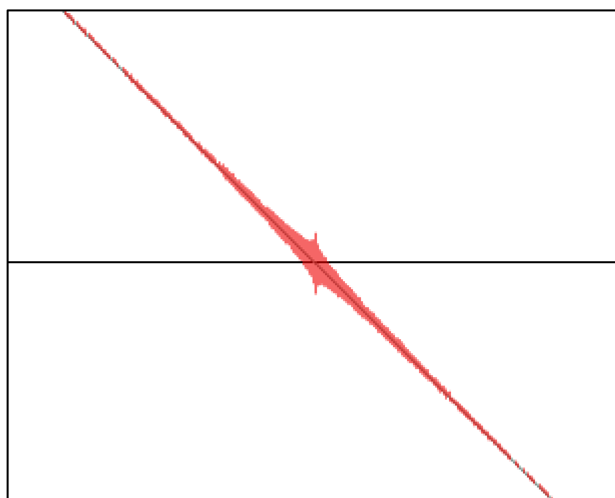
Obrázek 6: Vizualizace dvou implementací sinus se zvýrazněnými rozdíly při maximálním přiblížení.

Pomocí této aplikace bylo následně porovnáno několik knihoven. Konkrétně se jednalo o funkci sinus poskytovanou rozhraním .NET Framework, knihovnou využívající standardní funkce `math.h` přeloženou pomocí překladače Microsoft C/C++ Optimizing Compiler, a knihovnou Sun libmcr. Všechny tyto knihovny vykazovaly rozdílné chování. Nejčastěji se výsledky lišily pouze v nejméně významném bitu, viz obrázek 6. V určitých bodech byly ale rozdíly daleko zásadnější. Například hodnota $\sin(\pi)$ se však mezi rozhraním .NET Framework a knihovnou Sun libmcr lišila až o 41 bitů mantisy, viz tabulka 8. Vizualizace okolí bodu π je ukázána na obrázku 7. Tloušťka červené oblasti značí akumulovanou velikost rozdílu.

x	+3,14159265358979356008717331860680133104324308203125
sin(*x*) .NET	−0,000000000000000321628574467824890348310873378068208694 **1 01111001011 0111001011100000000000000000000000000000000000000000**
sin(*x*) libmcr	−0,000000000000000321624529935327320103979552447107383653 **1 01111001011 011100101100111011001110011001110101110100011111101**

Tabulka 8: Porovnání hodnot implementací .NET Framework a Sun libmcr.

Dále bylo otestováno chování překladače Microsoft C/C++ Optimizing Compiler. Kód, volající funkci sinus, byl přeložen pro platformu x86 a poté pro platformu x86-64. Následně byly porovnány hodnoty obou funkcí a bylo zjištěno, že se liší až ve 20 bitech mantisy. Z toho je zřejmé, že překladač využívá pro každou platformu odlišné algoritmy.



Obrázek 7: Vizualizace funkce sinus v okolí bodu π.

Literatura

[1] **IEEE Computer Society.** 754-1985 - IEEE Standard for Binary Floating-Point Arithmetic. [Online] 1985. [Citace: 20. října 2016.] http://ieeexplore.ieee.org/servlet/opac?punumber=2355.

[2] —. 754-2008 - IEEE Standard for Floating-Point. [Online] 29. srpna 2008. [Citace: 20. října 2016.] http://ieeexplore.ieee.org/servlet/opac?punumber=5976966.

[3] **Tišnovský, Pavel.** Norma IEEE 754 a příbuzní: formáty plovoucí řádové tečky. *Root.* [Online] 31. května 2006. [Citace: 7. listopadu 2016.] https://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky/.

[4] The GNU C Library: Mathematics. [Online] [Citace: 16. listopadu 2016.] https://www.gnu.org/software/libc/manual/html_node/Mathematics.html.

[5] <cmath> (math.h) - C++ Reference. *Cplusplus.com.* [Online] [Citace: 5. ledna 2017.] http://www.cplusplus.com/reference/cmath/.

[6] **Chen, Raymond.** Windows is not a Microsoft Visual C/C++ Run-Time delivery channel. [Online] 11. dubna 2014. [Citace: 17. prosince 2016.] https://blogs.msdn.microsoft.com/oldnewthing/20140411-00/?p=1273.

[7] **Microsoft.** Redistributing Visual C++ Files. *MSDN.* [Online] [Citace: 18. prosince 2016.] https://msdn.microsoft.com/en-us/library/ms235299.aspx.

[8] —. Determining Which DLLs to Redistribute. *MSDN.* [Online] [Citace: 18. prosince 2016.] https://msdn.microsoft.com/en-us/library/8kche8ah.aspx.

[9] —. CRT Library Features. *MSDN.* [Online] [Citace: 27. prosince 2016.] https://msdn.microsoft.com/en-us/library/abx4dbyh.aspx.

[10] **Intel.** Intel® 64 and IA-32 Architectures Software Developer’s Manual. [Online] září 2016. [Citace: 20. ledna 2017.] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[11] **Cornea, Marius.** FSIN Documentation Improvements in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual”. [Online] 9. října 2014.

[Citace: 20. ledna 2017.] https://software.intel.com/en-us/blogs/2014/10/09/fsin-documentation-improvements-in-the-intel-64-and-ia-32-architectures-software.

[12] **Sun Microsystems Inc.** Sun libmcr: Correctly Rounded libm. [Online] 27. dubna 2004. [Citace: 20. listopadu 2016.] https://github.com/simonbyrne/libmcr/.

[13] —. __libmcr_sin.c. *Sun libmcr.* [Online] 14. října 2014. [Citace: 5. listopadu 2016.] https://github.com/simonbyrne/libmcr/blob/master/src/__libmcr_sin.c.

[14] —. __libmcr_mx_sin.c. *Sun libmcr.* [Online] 14. října 2014. [Citace: 5. listopadu 2016.] https://github.com/simonbyrne/libmcr/blob/master/src/__libmcr_mx_sin.c.

[15] **Wolfram Research, Inc.** Remez Algorithm. *Wolfram MathWorld.* [Online] [Citace: 5. března 2017.] http://mathworld.wolfram.com/RemezAlgorithm.html.

[16] **Sun Microsystems Inc.** __libmcr_k_mx_sin. *Sun libmcr.* [Online] 14. října 2014. [Citace: 5. listopadu 2016.] https://github.com/simonbyrne/libmcr/blob/master/src/__libmcr_k_mx_sin.c.

[17] —. __libmcr_k_mm_sin.c. *Sun libmcr.* [Online] 14. října 2014. [Citace: 5. listopadu 2016.] https://github.com/simonbyrne/libmcr/blob/master/src/__libmcr_k_mm_sin.c.

[18] **Telea, Alexandru C.** *Data Visualization: Principles nad Practice.* Wellesley : A K Peters, Ltd., 2008. ISBN-13: 978-1-56881-306-6.

[19] **Bertini, Enrico.** How do you visualize too much data? [Online] 31. ledna 2011. [Citace: 20. listopadu 2016.] http://fellinlovewithdata.com/guides/how-do-you-visualize-too-much-data.

[20] **Ward, Matthew, Grinstein, Georges a Keim, Daniel.** *Interactive Data Visualization: Foundations, Techniques, and Applications.* Natick : A K Peters, Ltd., 2010. ISBN: 978-1-56881-473-5.

[21] **Fekete, Jean-Daniel a Plaisant, Catherine.** Interactive Information Visualization of a Million Items. [Online] leden 2002. [Citace: 10. listopadu 2016.] http://hcil2.cs.umd.edu/trs/2002-01/2002-01.pdf.

[22] **Microsoft.** Introduction to the C# Language and the .NET Framework. *MSDN.* [Online] 20. července 2016. [Citace: 2. ledna 2017.] https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx.

[23] —. Overview of the .NET Framework. *MSDN.* [Online] [Citace: 2. ledna 2017.] https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx.

[24] —. C# Language Specification 5.0. [Online] 7. června 2013. [Citace: 27. prosince 2016.] https://www.microsoft.com/en-us/download/details.aspx?id=7029.

[25] —. Casting and Type Conversions (C# Programming Guide). *MSDN.* [Online] 20. července 2015. [Citace: 10. února 2017.] https://msdn.microsoft.com/en-us/library/ms173105.aspx.

[26] —. Unsafe Code and Pointers (C# Programming Guide). *MSDN.* [Online] 20. července 2015. [Citace: 19. prosince 2016.] https://msdn.microsoft.com/en-us/library/t2yzs44b.aspx.

[27] —. Chapter 1: Introducing C#. *MSDN.* [Online] srpen 2010. [Citace: 20. prosince 2016.] https://msdn.microsoft.com/en-us/library/hh145616(v=vs.88).aspx.

[28] —. .NET Framework Class Library. *MSDN.* [Online] [Citace: 4. ledna 2017.] https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx.

[29] **Stebner, Aaron.** What version of the .NET Framework is included in what version of the OS? [Online] 14. března 2007. [Citace: 25. listopadu 2016.] https://blogs.msdn.microsoft.com/astebner/2007/03/14/mailbag-what-version-of-the-net-framework-is-included-in-what-version-of-the-os/.

[30] **Microsoft.** .NET Framework Versions and Dependencies. *MSDN.* [Online] [Citace: 10. ledna 2017.] https://msdn.microsoft.com/en-us/library/bb822049(v=vs.110).aspx.

[31] —. .NET Framework 4.5 and Windows XP. [Online] [Citace: 2. ledna 2017.] https://blogs.msdn.microsoft.com/dotnet/p/dotnet45xp/.

[32] **Mono Project.** Mono - Cross platform, open source .NET framework. [Online] [Citace: 4. ledna 2017.] http://www.mono-project.com/.

[33] —. Mono Releases. [Online] 28. července 2016. [Citace: 4. ledna 2017.]

http://www.mono-project.com/docs/about-mono/releases/.

[34] **Wolf, Carol E.** Infix to postfix conversion algorithm. *Professor Emerita Carol E. Wolf, NY Computer Science Department.* [Online] [Citace: 10. března 2017.] http://csis.pace.edu/~wolf/CS122/infix-postfix.htm.

[35] **Microsoft.** DllImportAttribute Class. *MSDN.* [Online] [Citace: 2. února 2017.] https://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute(v=vs.110).aspx.

[36] —. LoadLibraryEx function. *MSDN.* [Online] [Citace: 2. února 2017.] https://msdn.microsoft.com/en-us/library/windows/desktop/ms684179(v=vs.85).aspx.

[37] —. GetProcAddress function. *MSDN.* [Online] [Citace: 2. února 2017.] https://msdn.microsoft.com/cs-cz/library/windows/desktop/ms683212(v=vs.85).aspx.

[38] —. FreeLibrary function. *MSDN.* [Online] [Citace: 10. února 2017.] https://msdn.microsoft.com/en-us/library/windows/desktop/ms683152(v=vs.85).aspx.

[39] —. Marshal.GetDelegateForFunctionPointer Method (IntPtr, Type). *MSDN.* [Online] [Citace: 2. února 2017.] https://msdn.microsoft.com/en-us/library/zdx6dyyh(v=vs.110).aspx.

[40] —. UnmanagedFunctionPointerAttribute Class. *MSDN.* [Online] [Citace: 10. února 2017.] https://msdn.microsoft.com/en-us/library/system.runtime.interopservices.unmanagedfunctionpointerattribute(v=vs.110).aspx.

[41] —. __cdecl. *MSDN.* [Online] [Citace: 10. února 2017.] https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx.

[42] **Wimmer, Franz.** Invoking unmanaged 32bit library out of 64bit process. [Online] 28. září 2015. [Citace: 13. března 2017.] https://codefoundry.de/programming/2015/09/28/legacy-wrapper-invoking-an-unmanaged-32bit-library-out-of-a-64bit-process.html.

[43] **Hart, William.** MPIR - Multiple Precision Integers and Rationals. [Online] 1. března 2017. [Citace: 7. března 2017.] https://github.com/wbhart/mpir.

[44] **Bochkanov, Sergey.** Mpir.NET. [Online] 1. listopadu 2016. [Citace: 7. března 2017.] http://wezeku.github.io/Mpir.NET/.

[45] **Skeet, Jon.** DoubleConverter. [Online] [Citace: 20. listopadu 2016.] http://www.yoda.arachsys.com/csharp/DoubleConverter.cs.

[46] **Wil, Josh.** BigArray<T>, getting around the 2GB array size limit. [Online] 10. srpna 2005. [Citace: 28. prosince 2016.] https://blogs.msdn.microsoft.com/joshwil/2005/08/10/bigarrayt-getting-around-the-2gb-array-size-limit/.

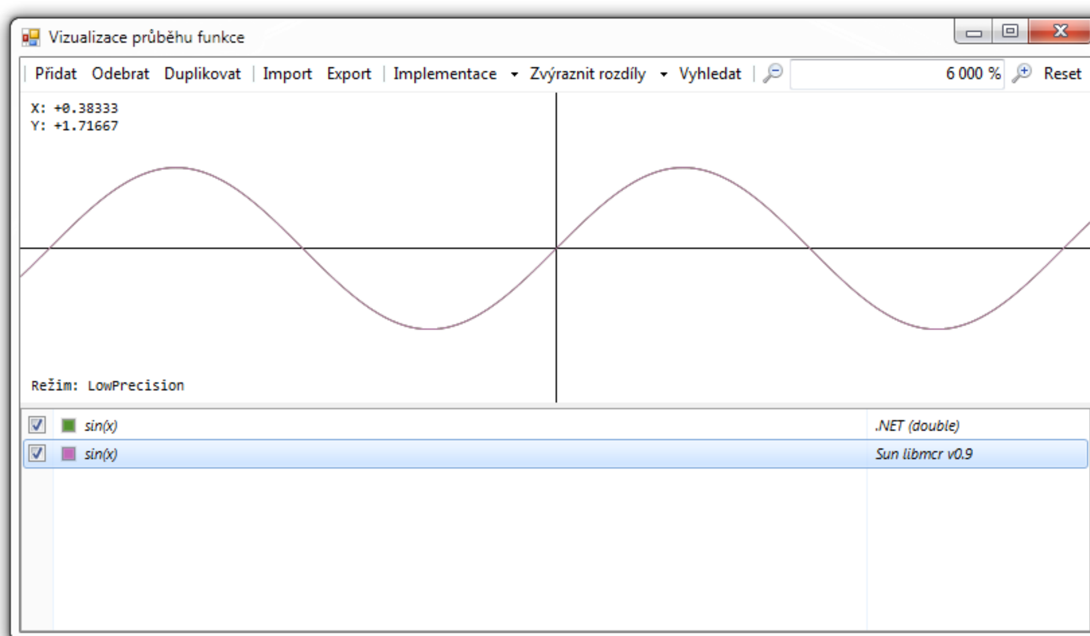
[47] **Microsoft.** GetTextExtentExPoint function. [Online] [Citace: 10. prosince 2016.] https://msdn.microsoft.com/en-us/library/windows/desktop/dd144935(v=vs.85).aspx.

[48] —. TextOut function. [Online] [Citace: 10. prosince 2016.] https://msdn.microsoft.com/en-us/library/windows/desktop/dd145133(v=vs.85).aspx.

Uživatelská příručka

Aplikaci je možné spustit na operačním systému Microsoft Windows 7 nebo novějším. Na počítači musí být nainstalováno rozhraní .NET Framework verze 4.5.2[2] nebo novější. Aplikaci lze spustit na 32-bitové i 64-bitové verzi operačního systému.

Spustí se otevřením spustitelného souboru `fVis.exe`. Na 64-bitovém operačním systému lze vynutit spuštění 32-bitové verze otevřením souboru `fVis.x86.exe`. Poté se ihned otevře hlavní okno aplikace, viz obrázek 8.



Obrázek 8: Hlavní okno aplikace.

Ve spodní části hlavního okna aplikace se nachází seznam matematických funkcí. Pomocí zaškrtávacího políčka lze jednotlivé funkce dočasně vypnout. Poté se nebudou vykreslovat, ani se nebudou hledat rozdíly mezi nimi. Následuje obdélníček, pomocí kterého je možné zvolit barvu funkce. První sloupec obsahuje název funkce. Pokud zadaný aritmetický výraz obsahuje chybu, text bude mít červenou barvu a místo, kde se chyba nachází, bude podtrženo. Na pravé straně

---

[2] Rozhraní .NET Framework lze stáhnout na adrese: https://www.microsoft.com/cs-cz/download/details.aspx?id=42642

tohoto sloupce se nachází varovná hlášení týkající se tohoto výrazu. Druhý sloupec obsahuje název matematické knihovny, která je přiřazena k tomuto aritmetickému výrazu. Velikost tohoto sloupce lze měnit přetažením svislé dělicí čáry.

V horní části je zobrazena vizualizace vybraných funkcí ze seznamu. Barva čáry funkce odpovídá barvě v seznamu. V levé horní části jsou vypsány souřadnice myši, které jsou počítané vůči osám grafu. Při větším přiblížení se vypíšou i jednotlivé hodnoty funkcí, na které ukazuje kurzor myši. Barva textu opět odpovídá barvám jednotlivých funkcí.

Pomocí hlavního panelu aplikace lze přidávat a odebírat aritmetické výrazy ze seznamu. Aritmetický výraz lze upravit dvojitým kliknutím na nápis „Zadejte aritmetický výraz" dole v seznamu.

Po zadání aritmetického výrazu lze exportovat všechny hodnoty v aktuálně zobrazeném intervalu do souboru, který lze následně do aplikace načíst.

Nabídka „Implementace" umožňuje přepínat mezi různými matematickými knihovnami. Pomocí tlačítka „Přidat" v této nabídce lze za běhu načíst libovolnou matematickou knihovnu a ihned jí začít používat. Pro automatické načtení je nutné knihovnu zkopírovat do složky `x64`, pokud se jedná o 64-bitovou knihovnu, případně do složky `x86`, pokud se jedná o 32-bitovou knihovnu.

Nabídka „Zvýraznit rozdíly" umožňuje vybrat režim zvýrazňování rozdílů mezi funkcemi. Možnost „Obyčejně" vyznačí rozdíly svislou čarou konstantní délky v místě, kde se funkce liší. Možnost „Dynamicky" vyznačí rozdíly svislou čarou s délkou v závislosti na velikosti rozdílu. Možnost „Pouze rozdíly" vypne vykreslování funkcí a vyznačí rozdíly pomocí křížků, které představují průměr nejvyšší a nejnižší hodnoty.

Tlačítko „Vyhledat" slouží k vyhledávání rozdílů na určeném intervalu. Po stisknutí tohoto tlačítka se zobrazí dialog, ve kterém je možné specifikovat interval a funkce, mezi kterými se budou rozdíly hledat. Po dokončení vyhledávání se zobrazí informace o počtu nalezených rozdílů. Nalezené rozdíly se zanesou do grafu.

Poslední část panelu slouží k přibližování a oddalování pohledu. Tlačítkem „Reset“ je možné uvést zobrazení do výchozího stavu. Tlačítko „Přiblížit k“ umožňuje maximálně přiblížit k zadané hodnotě vybrané funkce. Přiblížení je uvedeno v procentech. Hodnota přiblížení 100% značí, že hodnota 1 bude zobrazena ve vzdálenosti 1 pixelu od osy.