Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

BAKALÁŘSKÁ PRÁCE

Plzeň, 2007

Michal Kubricht

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Knihovna pro výpočty v projektivním prostoru

Plzeň, 2007 Michal Kubricht

Abstract

The floating point arithmetic library emerged from demands on higher algorithm stability, particularly in numerical computations in computer graphics, eventually in other scientific fields. One of the major present-day problems with computer computations is the division operation. It has a significant impact on result accuracy and computational speed of an algorithm. The amount of application of this operation can be greatly limited in the so-called projective space.

Obsah

Prohlášení

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

V Plzni, dne ....

Michal Kubricht

1 Úvod

Knihovna pro výpočty v projektivním prostoru vznikla z požadavků na vyšší stabilitu algoritmů zejména v počítačové grafice, případně i v jiných numerických výpočtech. Jeden z problémů, které se snaží knihovna odstranit, je nepřesnost výsledků počítačem prováděných matematických operací. Relativně velký podíl na těchto nepřesnostech má operace dělení. Používání této operace se dá výrazně omezit využitím principů projektivního prostoru, především využitím homogenních souřadnic.

Cílem bakalářské práce je nastudování projektivního rozšíření Eukleidovského prostoru a následné využití těchto poznatků při implementaci knihovny. K dispozici byla dodána funkcionalita, která má být modifikována za použití objektově orientovaných technologií v jazyce C# na platformě Microsoft .NET Framework.

V průběhu vývoje knihovny probíhala spolupráce s V. Ondračkou. Třídy PLib a PlibException byly vytvořeny V. Ondračkou v rámci oborového projektu na katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni.

1.1 Obsah teoretické části

V teoretické části budou vysvětleny základní pojmy používané při řešení problému.

- Projektivní prostor – definice, vztah k Eukleidovskému prostoru.
- Homogenní souřadnice – definice, přepočet do souřadnic Eukleidovského prostoru, reprezentace bodů a vektorů.
- Použití projektivního prostoru.

1.2 Obsah realizační části

V realizační části je stručná specifikace použitých datových typů a podporovaných funkcí. Je provedena důkladná analýza možností reprezentace dat a zvážení postupů pro udržování

stability výpočtů. V rámci analýzy jsou provedena některá testování. Na základě získaných výsledků je učiněna volba datových struktur knihovny. Datové struktury jsou popsány včetně způsobu implementace. V téže kapitole jsou uvedeny i některé důležité metody knihovny. Dále je popsáno testování dosažených výsledků. V závěru této části je uveden návrh na optimalizaci knihovny.

2 Teoretická část

2.1 Projektivní prostor

Projektivní prostor P^n nad tělesem reálných čísel je definován vztahem

$$P^n = (E^{n+1})/\sim \tag{2.1}$$

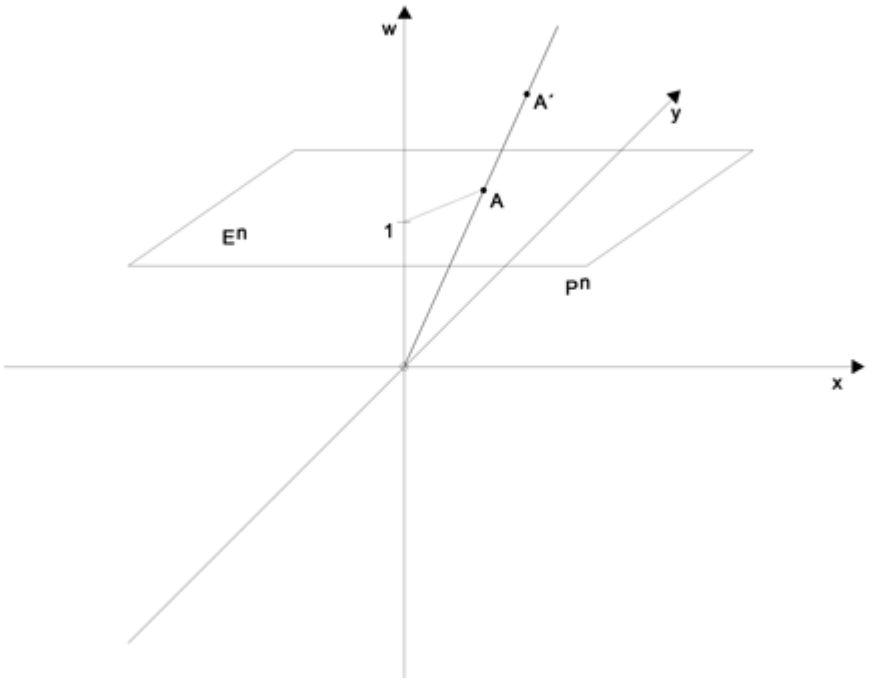
s ekvivalencí definovanou vztahem

$$(x_0, \ldots, x_n) \sim (\lambda \cdot x_0, \ldots, \lambda \cdot x_n), \tag{2.2}$$

kde $\lambda \in \mathbb{R} \setminus \{0\}$ je libovolné nenulové reálné číslo a $n \in \mathbb{N} \setminus \{0\}$ je přirozené nenulové číslo.

Na základě uvedeného vztahu (2.1) se dá hovořit o projektivním prostoru jako o rozšíření Eukleidovského prostoru E^n o jeden rozměr. Dále je ze vztahu (2.2) patrné, že pro každý bod v Eukleidovském prostoru existuje v projektivním prostoru nekonečně mnoho obrazů tohoto bodu. Z geometrického hlediska je každá takováto množina (neboli každá třída vytvořená rozkladem podle ekvivalence) přímkou v projektivním prostoru. Tato přímka prochází počátkem $(0, \ldots, 0)$ a je zřejmé, že v něm není definovaná.

Pokud $n = 2$, můžeme vztah projektivního prostoru a Eukleidovského prostoru graficky znázornit (viz obr. 2.1). Kartézské souřadnice bodu A v Eukleidovském prostoru E^2 jsou (a, b). Přímka ležící na bodech A a A' je onou zmiňovanou množina všech obrazů bodu A.



Obr. 2.1 Geometrický vztah mezi Eukleidovským a projektivním prostorem

2.2 Homogenní souřadnice

Pro potřebu výpočtů v projektivním prostoru bylo třeba zavést souřadnice. Tyto souřadnice byly nazvány homogenními souřadnicemi.

Homogenní souřadnice bodu projektivního prostoru dimenze n jsou obvykle zapisovány ve tvaru $(x_1, x_2, \ldots, x_n, x_w)$, tj. řádkovým vektorem o délce $n+1$, kde $x_i \in \mathbb{R}$ a $i \in \{1, \ldots, n, w\}$.

Homogenními souřadnicemi $(x_1, \ldots, x_n, x_w)$ konečného bodu Eukleidovského prostoru s kartézskými souřadnicemi $(x'_1, \ldots, x'_n)$ je uspořádaná $(n+1)$-tice skalárních hodnot, pro něž platí

$$\frac{x_i}{x_w} = x'_i, \quad i \in \{1, \ldots, n\}, \tag{2.3}$$

kde x_w je tzv. homogenní složka.

Navíc ze vztahu (2.2) plyne neexistence bodu o souřadnicích $(0, \ldots, 0, 0)$.

Dva body Eukleidovského prostoru vyjádřené v projektivním prostoru pomocí homogenních souřadnic $(x_1, \ldots, x_n, x_w)$ a $(x'_1, \ldots, x'_n, x'_w)$ jsou totožné, pokud platí

$$x'_i = \lambda \cdot x_i, \quad \lambda \in \mathbb{R} \setminus \{0\}, \quad x_i \in \mathbb{R}, \quad i \in \{1, \ldots, n, w\}. \tag{2.4}$$

Z této věty vyplývá jednoduchý vztah mezi souřadnými systémy obou prostorů a vzorce pro jejich vzájemný převod.

Pokud chceme převést bod z Eukleidovského prostoru do projektivního, použijeme vzhledem k definici platný vztah

$$P^n : (x_1, \ldots, x_n) \rightarrow E^n : (x_1, \ldots, x_n, 1). \tag{2.5}$$

Při opačném převodu se využívá vztahu (2.3). Tedy

$$E^n : (x_1, \ldots, x_n, x_w) \rightarrow P^n : (\frac{x_1}{x_w}, \ldots, \frac{x_n}{x_w}). \tag{2.6}$$

2.3 Použití projektivního prostoru

Nyní se zkusme zamyslet nad využitím projektivního prostoru v implementaci algebraických operací (tak, aby se redukovalo používání operace dělení). Pro jednoduchost nejdříve vezmeme situaci, kdy $n = 1$. Pokud tedy zapíšeme dvě libovolné hodnoty v homogenních souřadnicích (a, a_w) a (b, b_w), můžeme pro ně zavést základní aritmetické operace.

Součet (resp. rozdíl) dvou skalárních hodnot v homogenních souřadnicích $(c, c_w) = (a, a_w) \pm (b, b_w)$ definujeme pomocí vztahů zvlášť pro obě složky

$$c = a \cdot b_w \pm b \cdot a_w, \tag{2.7}$$

$$c_w = a_w \cdot b_w. \tag{2.8}$$

Součin dvou skalárních hodnot v homogenních souřadnicích $(c, c_w) = (a, a_w) \cdot (b, b_w)$

definujeme opět pomocí dvou vztahů

$$c = a \cdot b \,, \tag{2.9}$$

$$c_w = a_w \cdot b_w \,. \tag{2.10}$$

Stejně tak pro podíl dvou skalárních hodnot v homogenních souřadnicích $(c\,,c_w) = (a\,,a_w)\cdot(b\,,b_w)$ zadefinujeme dva vztahy pro jeho výpočet

$$c = a \cdot b_w \,, \tag{2.11}$$

$$c_w = b \cdot a_w \,. \tag{2.12}$$

Z toho je vidět, že provádíme-li podíl jednorozměrným projektivním vektorem (projektivním skalárem), lze jej plně nahradit součinem. Díky tomuto principu se nemusíme obávat situace, kdy dělíme nulou.

Všechny uvedené vztahy jsou korektní vzhledem k zavedeným homogenním souřadnicím. Lze se o tom snadno přesvědčit převedením zpět do kartézských souřadnic. Uvedeme zde pouze ověření pro podíl.

Provedeme-li převod obou operandů i výsledku operace dělení do kartézských souřadnic podle vztahu (2.3), dostáváme

$$a' = \frac{a}{a_w} \,, \tag{2.13}$$

$$b' = \frac{b}{b_w} \,, \tag{2.14}$$

$$c' = \frac{c}{c_w} \,. \tag{2.15}$$

Po dosazení (2.11) a (2.12) do (2.15) a následné úpravě získáme

$$c' = \frac{(a \cdot b_w)}{(b \cdot a_w)} = \frac{(\frac{a}{a_w})}{(\frac{b}{b_w})} = \frac{(a')}{(b')}, \tag{2.16}$$

z něhož je již evidentní korektnost operace.

Stejným způsobem lze ověřit i ostatní operace v P^1. Je tedy třeba připomenout fakt, že pro počítání v projektivním prostoru P^1 není třeba používat operaci dělení.

Nyní musíme zavést body projektivního prostoru P^n v homogenních souřadnicích, kde $n \in \mathbb{N}$ a $n \geq 2$. Mějme tři vektory v kartézských souřadnicích $(a_1, \ldots, a_n)$, $(b_1, \ldots, b_n)$, $(c_1, \ldots, c_n)$ a jeden projektivní skalár (d, d_w). Pak pro tyto vektory platí, že součet (resp. rozdíl)

$$(c_1, \ldots, c_n, c_w) = (a_1, \ldots, a_n, a_w) \pm (b_1, \ldots, b_n, c_w) \tag{2.17}$$

je definován vztahy

$$c_i = a_i \cdot b_w \pm b_i \cdot a_w, \tag{2.18}$$

$$c_w = a_w \cdot b_w, \tag{2.19}$$

skalární součin

$$(d, d_w) = (a_1, \ldots, a_n, a_w) \cdot (b_1, \ldots, b_n, b_w) \tag{2.20}$$

je definován vztahy

$$d = a_i \cdot b_i, \tag{2.21}$$

$$d_w = a_w \cdot b_w, \tag{2.22}$$

kde $i \in \{1, \ldots, n\}$.

Ověření, zda jsou tyto operace správně zavedeny, ukážeme pouze na sčítání, ostatní lze provést analogicky. Toto ověření provedeme opět s využitím vztahu (2.3) a nejdříve převedeme

vektory do kartézských souřadnic. Máme tedy

$$(a'_1, \ldots, a'_n) = (\frac{a_1}{a_w}, \ldots, \frac{a_n}{a_w}), \tag{2.23}$$

$$(b'_1, \ldots, b'_n) = (\frac{b_1}{b_w}, \ldots, \frac{b_n}{b_w}), \tag{2.24}$$

$$(c'_1, \ldots, c'_n) = (\frac{c_1}{c_w}, \ldots, \frac{c_n}{c_w}). \tag{2.25}$$

Po součtu těchto vektorů a následné úpravě s využitím vztahů (2.19), (2.20) získáme

$$(c'_1, \ldots, c'_n) = (a'_1 + b'_1, \ldots, a'_n + b'_n) = \ldots \tag{2.26}$$

$$\ldots = (\frac{a_1}{a_w} + \frac{b_1}{b_w}, \ldots, \frac{a_n}{a_w} + \frac{b_n}{b_w}) =$$

$$\ldots = (\frac{(a_1 \cdot b_w + b_1 \cdot a_w)}{(a_w \cdot b_w)}, \ldots, \frac{(a_n \cdot b_w + b_n \cdot a_w)}{(a_w \cdot b_w)}) = (\frac{c_1}{c_w}, \ldots, \frac{c_n}{c_w}).$$

3 Realizační část

3.1 Specifikace požadavků

Na začátku bylo třeba upřesnit požadavky na knihovnu. Níže je uveden jejich stručný přehled, původní specifikace dodaná vedoucím bakalářské práce je obsažena v příloze B. Referenční příručka všech funkcí knihovny je realizována v elektronické podobě na CD.

Datové typy:

- Double1 – skalární hodnota v Eukleidovském prostoru
- Double1P – jednorozměrný vektor s homogenní složkou (dále v textu nazývaný projektivní skalár)
- DoubleN – N-rozměrný vektor v Eukleidovském prostoru E^n
- DoubleNP – N-rozměrný vektor v projektivním prostoru P^n
- DoubleNM – Matice typu $N \times N$ v Eukleidovském prostoru E^n
- Double2, Double2P, Double3, Double3P, Double4, Double4P – vektory lišící se počtem složek (Eukleidovské i projektivní), datové typy pro zpětnou kompatibilitu s předchozí verzí knihovny

Operace:

- Aritmetické – výpočty mezi skalární hodnotou a vektorem (Eukleidovským i projektivním), případně maticí.
- Vektorové – práce s vektory v Eukleidovském a projektivním prostoru, včetně skalárního součinu, vektorový součin má význam řešit pouze pro vektory maximálně se 4 složkami v případě Eukleidovského prostoru a s maximálně 3 složkami v případě projektivního prostoru.

- Maticové – operace sčítání, odečítání a násobení matic.

3.2 Analýza

3.2.1 Normalizace

Před vlastní analýzou možností datových struktur, uvedeme několik slov o normalizaci vektorů s homogenní složkou. Vezmeme si například vztahy (2.18) a (2.19) pro sčítání projektivních vektorů, vidíme z nich, že homogenní složka se vypočítá jako součin homogenních složek obou operandů. Pokud budou obě tyto složky větší jak 1 a sčítání bude probíhat například v nějakém iteračním algoritmu, tak se velmi rychle může stát, že homogenní složka výsledku způsobí překročení rozsahu typu double. K předcházení těchto problémů se využívá právě normalizace. Její princip následuje.

Využijeme k tomu znalosti o uložení desetinných čísel v počítači. Každé takové číslo se ukládá ve formátu, kdy je zvlášť uložená mantisa i exponent (znaménko nehraje v tomto případě roli a proto budeme uvažovat, že je obsažené v mantise). Mantisu čísla x označíme x_m a exponent x_e. Každé desetinné číslo x tedy můžeme přepsat ve tvaru $x = x_m \cdot 2^{x_e}$.

Předpokládáme, že máme projektivní vektor $(x_1, \ldots, x_n, x_w)$. Podle posledního vztahu dostáváme

$$(x_1, \ldots, x_n, x_w) = (x_{1m} \cdot 2^{x_{1e}}, \ldots, x_{nm} \cdot 2^{x_{ne}}, x_{wm} \cdot 2^{x_{we}}), \tag{3.1}$$

pak s využitím vztahu (2.2) můžeme provést tyto úpravy

$$(x_1, \ldots, x_n, x_w) = (n \cdot x_1, \ldots, n \cdot x_n, n \cdot x_w) = \ldots \tag{3.2}$$

$$\ldots = (x_{1m} \cdot \frac{2^{x_{1e}}}{2^{x_{we}}}, \ldots, x_{nm} \cdot \frac{2^{x_{ne}}}{2^{x_{we}}}, x_{wm} \cdot \frac{2^{x_{we}}}{2^{x_{we}}}) = \ldots$$

$$\ldots = (x_{1m} \cdot 2^{(x_{1e} - x_{we})}, \ldots, x_{nm} \cdot 2^{(x_{ne} - x_{we})}, x_{wm}).$$

Pouhou úpravou jsme získali ekvivalentní vyjádření, v němž je exponent homogenní složky roven nule. Tím je zajištěno, že při násobení homogenní složkou, které je při výpočtech často využíváno, nedochází k jeho velkým změnám. Díky tomu, že jsme zkrátili pouze část

exponentu, nedošlo ke ztrátě přesnosti desetinného čísla, o kterou jde v numerických výpočtech především.

3.2.2 Původní implementace datových typů

Vedoucím práce byla dodána základní verze knihovny, která obsahovala částečnou implementaci pomocí struktur bez využívání objektů. Složky jednotlivých vektorů byly definovány napevno. Ke všem byl umožněn veřejný přístup umožňující zápis i čtení přímo. Všechny patřičné operace byly implementovány pomocí přetížení operátorů ve všech kombinacích, u kterých byla vyžadována funkčnost.

Výhoda této implementace byla v použití struktur jazyka C#, které jsou tzv. hodnotovými typy a při přiřazování proměnných těchto typů dochází ke zkopírování celého objektu. Při práci si tedy nemusíme dávat pozor na reference dvou nebo více proměnných na stejné místo v paměti.

Alokace paměti pro struktury probíhá na zásobníku. Tato alokace je obecně považována za rychlejší, než alokace paměti na haldě. Více o porovnání struktur a objektů je zmíněno v kapitole 3.2.5.

Na druhé straně je u tohoto řešení zjevná nevýhoda v tom, že by bylo potřeba naprogramovat všechny kombinace přetížení operátorů, které by knihovna zprostředkovávala. Navíc je tímto způsobem nemožné pokrýt obecně n -rozměrné vektory.

3.2.3 Návrh úprav původní implementace

V návaznosti na předchozí řešení byla prozkoumána i možnost sjednocení operací pomocí *unsafe* kódu (speciální konstrukce jazyka C# umožňující práci s ukazateli). V tomto případě byla snaha omezit psaní většího množství kódu, který by byl náchylný na chybu při programování. Stále by bylo třeba psát všechna přetížení operátorů jako v předchozím řešení, ale tělo těchto částí kódu by bylo prakticky totožné pro každý typ a měnil by se pouze název volané metody.

Uvedeme ukázku takového kódu – metodu pro sčítání dvou projektivních vektorů.

Ukázka 3.1

```
public static unsafe double* Add2P(double* p_left, double*
        p_right, double* p_result, int size)
{
    // získání homogenních složek operandů
    double left_w = *(p_left + size);
    double right_w = *(p_right + size);
    // pro všechny prvky vektoru
    for (int i = 0; i < size; i++)
    {
        // spočítáme výslednou hodnotu každé složky
        *p_result = (*p_left) * right_w +
                    (*p_right) * left_w;
        // a posuneme ukazatele na další hodnotu
        p_left++;
        p_right++;
        p_result++;
    }
    // spočítáme výslednou hodnotu homogenní složky
    *p_result = left_w * right_w;
    // vrátíme ukazatel výsledek
    return p_result;
}
```

Metoda je veřejná statická, aby byla přístupná všem strukturám v programu bez zbytečných referencí. Jako parametry přijímá ukazatele na levý operand, pravý operand, alokované místo výsledku operace, počet složek vektoru bez složky homogenní.

Nejdříve jsou získány homogenní složky obou operandů, poté ve *for-cyklu* spočítají jednotlivé složky vektoru podle vztahu (2.18). Nakonec se uloží výsledná homogenní složka.

Ve *for-cyklu* je použita tzv. pointerová aritmetika, známá především z jazyků C a C++. Například „`(*p_left)`“ značí dereferenci ukazatele na místo v paměti, kam ukazuje, „`p_left++`“ značí posunutí ukazatele v paměti (o tolik bytů, kolik zabírá datový typ, na který ukazatel ukazuje).

Nyní uvedeme kód v přetížení operátoru pro volání patřičné metody.

Ukázka 3.2

```
public static Double3P operator +(Double3P left,
        Double3P right)
{
    // alokování místa pro výsledek operace
    Double3P result = new Double3P();
    unsafe
    {
        // získání ukazatelů na operandy a výsledek
        double* p_left = (double*)&left;
        double* p_right = (double*)&right;
        double* p_result_alocation = (double*)&result;

        // volání metody pro součet dvou vektorů
        PLib.Add2P(p_left, p_right, p_result_alocation,
            Double3P.N);
    }
    // vrácení výsledku operace
    return result;
}
```

V této ukázce se nejdříve alokuje paměť pro výsledek operace, poté se pomocí operátoru reference vytvoří ukazatele na všechny tři požadované struktury a provede se volání metody pro určení součtu vektorů.

Výhodou tohoto přístupu je, že stejně jako předchozí verze, používá struktury. Navíc takto navržené přetížení operátorů je méně náchylné na chyby při psaní samotného kódu. V případě implementace pro další datové typy, které se liší pouze počtem složek vektoru, se dá kód v ukázce (3.2) snadno automaticky upravit (pomocí prakticky každého editoru). V příslušném editoru (např. VS.NET 2005) se použije funkce nahradit a bude nahrazeno klíčové slovo **Double3P** jiným (např. **Double4P**) a výsledný kód bude funkční bez dalších změn. V případě implementace jiného operátoru pro stejný datový typ se pouze změní název volané metody pro výpočet. Tímto by bylo možné automatické generování kódu struktur s různým počtem složek vektoru (kromě projektivních skalárů, které jsou pro jejich specifické použití implementovány zvlášť).

Jedna z nevýhod původní implementace však stále platí. Nebylo by možné podporovat

vektory obecně o n složkách. Navíc přibyla menší nevýhoda v tom, že se při práci s vektory kromě metody obsluhující přetížený operátor, provádí volání metody pro výpočet požadované operace.

3.2.4 Návrh datových typů pomocí objektů

Zaměřme se na možnost implementace datových typů s použitím objektů místo struktur. Každý takový objekt by obsahoval rozměr vektoru, pole se složkami vektoru a případně i homogenní složku, pokud by se jednalo o třídu reprezentující projektivní vektor.

Díky objektům a použití polí je možné uchovávat obecně n -rozměrné vektory.

Nevýhody použití objektů jsou v zásadě dvě. První tkví v tom, že paměť pro objekty se alokuje na haldě, což je pomalejší než alokace struktur na zásobníku. Podrobnější informace následují v další kapitole. Druhou a závažnější nevýhodou je skutečnost, že proměnné takových datových typů jsou referenční (nikoliv hodnotové jako u struktur). To v praxi znamená, že při přiřazování mezi proměnnými stejného typu nedochází ke kopírování celých objektů v paměti, ale pouze ke zkopírování jejich reference. V softwarovém inženýrství se pro tento jev používá termín mělká kopie. Zde je uveden příklad.

Ukázka 3.3

```
// vytvoření projektivního vektoru dimneze 4
DoubleNP a = new DoubleNP(new double[] { 2, 3, 4, 5 });
Console.Write("a.W = " + a.W.ToString() + "\n");
DoubleNP b = a; // vytvoření mělké kopie
b.W = 2; // změna dat b, která se projeví i v a!
Console.Write("a.W = " + a.W.ToString());
```

V obou případech se vypíší jiná data. V prvním výpisu to bude "`a.W = 1`" a ve druhém "`a.W = 2`". Důvodem je to, že přiřazením proměnné `a` do `b` se vytvořila mělká kopie a obě proměnné v tuto chvíli ukazují na stejný objekt v paměti. Neboli – jakákoliv změna dat se nutně musí promítnout v obou proměnných současně. Na tuto skutečnost je nutné při práci s datovými typy knihovny dávat velký pozor.

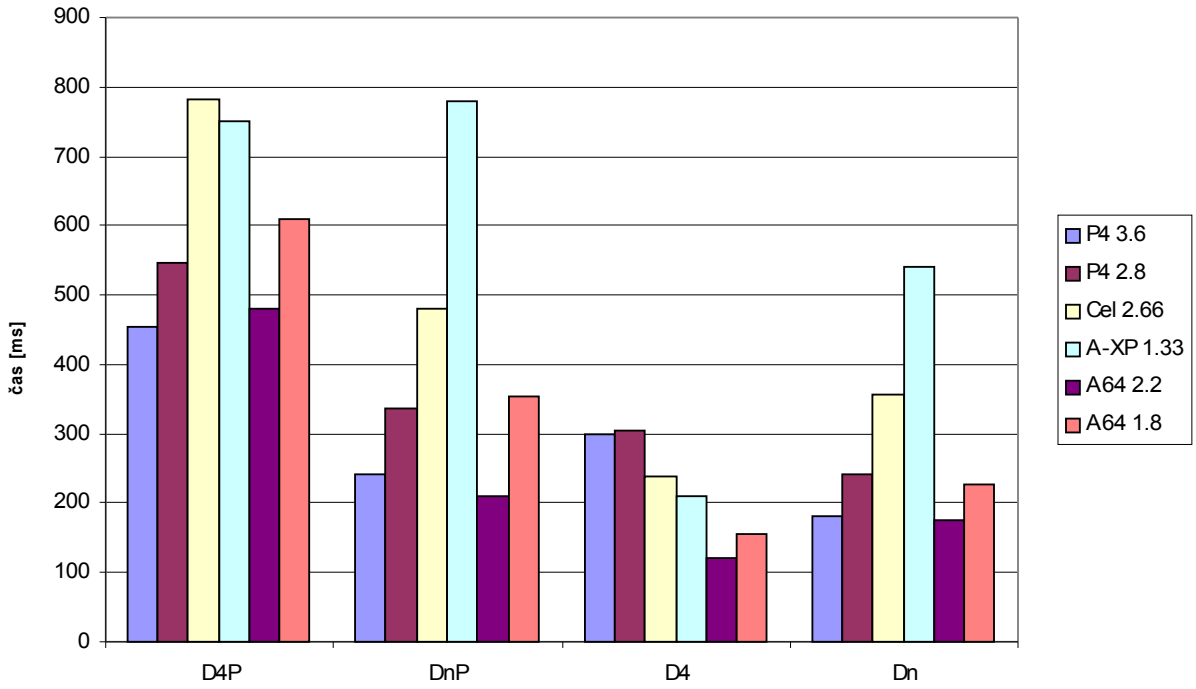
3.2.5 Srovnání struktur a objektů

Následující přehled základních rozdílů se týká struktur a objektů v jazyce C# na platformě .NET Framework.

Struktury patří mezi hodnotové typy a jejich alokace probíhá na zásobníku. To, že patří mezi hodnotové typy, znamená, že přiřazení mezi proměnnými těchto typů způsobí zkopírování celé struktury, což přináší jistý výkonnostní pokles při častém kopírování. Kopírování probíhá i v případě, že jsou hodnotové typy použity jako parametry metod. U parametrů metod se tomu však dá zabránit použitím klíčového slova *ref*. Tento způsob chování však poskytuje výhodu v tom, že není třeba věnovat zvýšenou pozornost při manipulaci s jejich daty. Ta totiž nezpůsobuje změnu instance přiřazenou více proměnným.

Objekty patří mezi referenční typy a jejich instance se vytváří na haldě. Referenční typy kopírují při přiřazení pouze reference, takže pak proměnné ukazují na stejné místo. Jakákoliv změna se tak promítne do všech proměnných, které na tento měněný objekt odkazují.

V průběhu vývoje knihovny jsme obě varianty naprogramovali a porovnali. Byly provedeny rychlostní testy na šesti různých typech procesorů. Zvlášť byly provedeny testy pro vektory v Eukleidovském i projektivním prostoru. Výsledky jsou zobrazeny v následujícím grafu 3.1.



Graf 3.1 Doba výpočtu 10^6 operací sčítání čtyřrozměrných vektorů

Testovací procesory: Intel Pentium 4, 3.6GHz; Intel Pentium 4, 2.8GHz; Intel Pentium Celeron 2.66GHz, AMD Athlon XP 1500+, 1.33GHz; AMD Athlon 64 3400+, 2.2GHz; AMD Athlon 64 2800+, 1.8GHz.

Legenda: Popisky zobrazené na horizontální ose popisují datový typ, pro který byly provedeny testy, nad ním jsou dosažené časy. D4P *značí výpočet v projektivním prostoru s využitím struktur pro reprezentaci dat.* DnP *značí výpočet v projektivním prostoru s využitím objektů.* D4 *značí výpočet Eukleidovském prostoru s využitím struktur.* Dn *značí výpočet v Eukleidovském prostoru s využitím objektů.*

Výsledek testu se výrazným způsobem liší pro každý typ procesoru. Záleží nejenom na taktovací frekvenci, ale samozřejmě i na výrobci a dalších parametrech. Nemělo by tedy smysl uvádět nějaké konkrétní porovnání pro každý procesor zvlášť.

Pro novější typy procesorů je v Eukleidovském prostoru rychlost práce s použitím struktur a s použitím objektů srovnatelná. V projektivním prostoru jsou objekty přibližně dvakrát

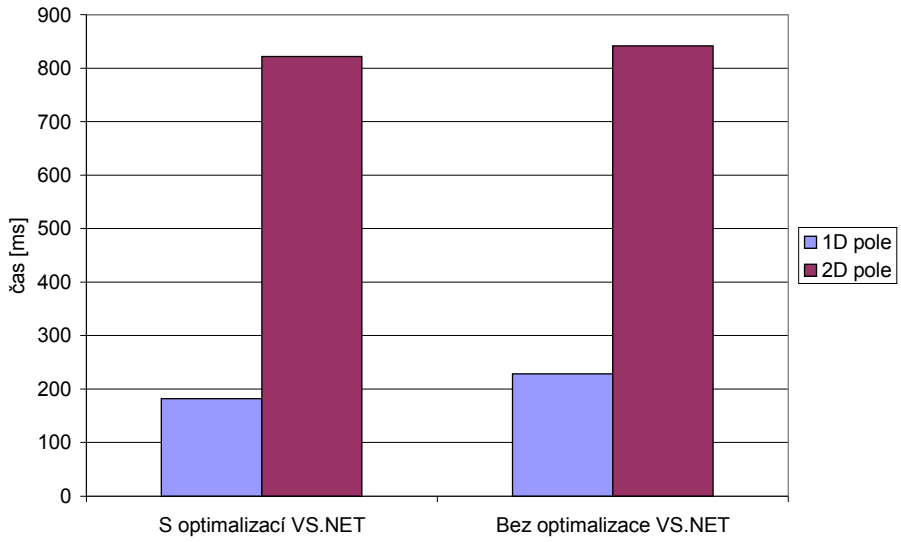
rychlejší. To pro nás bylo překvapivým zjištěním, neboť na počátku jsme odhadovali, že tomu bude naopak a použití struktur bude rychlejší. Pro procesor Athlon XP 1500+ se starší architekturou byly výsledky diametrálně odlišné a spíše tak potvrzovaly původní domněnku. Z analýzy výsledků i ze samotného grafu je však patrné, že novější typy procesorů jsou více optimalizovány na práci s objekty. Je ovšem třeba podotknout, že v případě měření šlo spíše o hrubou představu výkonnostních rozdílu, neboť ani na jedné z implementací nebyly provedeny rozsáhlejší optimalizace.

Pokusíme se o částečné zdůvodnění naměřených výsledků. Zvažme, co se děje při alokaci paměti na zásobníku. Alokační rutina najde prázdné místo požadované velikosti a vrací na něj ukazatel. Tento ukazatel může ukazovat prakticky na náhodnou pozici. Naproti tomu garbage collector, který je zodpovědný za alokaci místa na haldě, nemusí hledat volnou paměť, protože zná pozici, na které se nalézá. Ukazatel na paměť, kterou garbage collector vrací, přímo sousedí s poslední vyžádanou pamětí. Díky tomu je mnohem více pravděpodobné, že požadovaná paměť už je načtená v cache paměti procesoru a proto může být i přístup k datům rychlejší.

3.2.6 Porovnání výkonnostních rozdílů 1D a 2D polí

Vzhledem k tomu, že knihovna podporuje i práci s maticemi, byl otestován ještě jeden aspekt reprezentace dat a to rychlost implementace pomocí jednorozměrných a dvourozměrných polí. Pro testování bylo provedeno 10^6 operací odečítání dvou čtvercových matic řádu 4. Výsledky jsou zobrazeny na grafu 3.2. Z grafu je patrné, že implementace pomocí jednorozměrného pole je zhruba čtyřikrát rychlejší než implementace pomocí dvourozměrných polí.

Testy byly prováděny na procesorech *Intel Pentium 4, 3.6GHz* a *AMD Athlon 64 2800+, 1800GHz* se zapnutou optimalizací kódu ve VS.NET i bez ní. V obou případech i na obou procesorech byly výsledky prakticky totožné.



Graf 3.2 Doba výpočtu 10^6 operací odečítání čtvercových matic řádu 4

3.3 Popis implementace

3.3.1 Volba datových struktur a popis tříd

Na základě analýzy a výsledků provedených testů bylo rozhodnuto, že implementace datových struktur bude provedena pomocí objektů. Umožňuje to jednoduchým způsobem naprogramovat podporu knihovny pro n -rozměrné vektory. Vzhledem k původní verzi knihovny byly vytvořeny také třídy zaručující zpětnou kompatibilitu se staršími verzemi.

Následuje seznam a popis jednotlivých tříd programu. Kompletní referenční příručka knihovny je uvedena v příloze B.

Double1P

Třída reprezentující projektivní skalár (jednorozměrný vektor v projektivním prostoru). Třída dědí třídu **DoubleNP**. Obsahuje property X pro čtení i zápis první složky vektoru. Přístup k homogenní složce je zajištěn v rodičovské třídě. Obsahuje přetížení operátorů pro implicitní konverzi z datového typu *double* i na něj. Dále obsahuje přetížené operátory porovnání mezi **Double1P** a double a porovnání mezi dvěma argumenty typu **Double1P** a přetížení operátorů pro výpočty s projektivním skalárem.

DoubleN

Třída reprezentující vektor v Eukleidovském prostoru. Složky vektoru uchovává v poli double hodnot. Velikost pole je závislá na atributu N. Tento atribut je zjištěn a nastaven v konstruktoru a v průběhu existence každé instance se nemění (je pro něj použito klíčové slovo *readonly*). Pro přístup k poli je vytvořena property pro čtení, aby se předešlo nesprávnému zacházení, které by mohlo způsobit chybu programu. Obsahuje přetížení operátorů pro implicitní konverzi na datový typ *double[]* i opačným směrem a explicitní konverzi z datového typu **DoubleNP**. Dále obsahuje přetížené operátory porovnání na rovnost či nerovnost vektorů mezi datovými typy **DoubleN** i **DoubleNP** a přetížení operátorů pro výpočty s vektory.

DoubleNP

Třída reprezentující vektor v projektivním prostoru. Složky vektoru uchovává v poli double hodnot. Velikost pole je závislá na atributu N. Tento atribut je zjištěn a nastaven v konstruktoru a v průběhu existence každé instance se nemění (je pro něj použito klíčové slovo *readonly*). Pro přístup k poli je vytvořena property pro čtení, aby se předešlo nesprávnému zacházení, které by mohlo způsobit chybu programu. Obsahuje přetížení operátorů pro implicitní konverzi z datového typu *double[]* a explicitní konverzi na datový typ *double[]*. Dále obsahuje přetížení operátory porovnání na rovnost či nerovnost vektorů mezi datovými typy **DoubleNP** a přetížení operátorů pro výpočty s vektory v projektivním prostoru.

DoubleNM

Třída reprezentující čtvercové matice skalárních hodnot Eukleidovského prostoru obecně řádu n. Vzhledem k testům uvedeným v kapitole 3.1.6 bylo pro reprezentaci dat zvoleno jednorozměrné pole. Přináší to výhody z hlediska rychlosti přístupu, na druhou stranu je třeba se na prvky matice odkazovat pomocí jednoho indexu. Přepočet je zřejmý – prvek na pozici $[i, j]$ se v jednorozměrném poli vyskytuje na indexu $[i \cdot N + j]$, přičemž je třeba dbát na to, že prvky v poli jsou indexovány od 0. Stejně jako u předchozích tří typů i v této třídě se vyskytuje přetížení operátorů pro výpočty s datovým typem **DoubleNM**.

PLib

Třída poskytující základní metody používané v knihovně. Metody jsou veřejné statické,

takže se dají používat i mimo knihovnu. Jedná se například o metody *NormalizeExp*, *Check* nebo *DeSign*, které slouží v udržování stability a správnosti výpočtů. Dále o metody *Cross* nebo *Length*, které slouží přímo uživatelům knihovny pro matematické výpočty. Popis zajímavých algoritmů některých metod je uveden v následující kapitole 3.2.2.

PlibException

Tato třída představuje vlastní výjimku napsanou pro knihovnu, která je vyhazována především ve dvou případech. Jednak při chybném zacházení s jejími daty, což znamená například pokus o sečtení dvou vektorů o různém počtu složek, a také v případě, že došlo k přetečení či podtečení dat. Více podrobností u popisu implementace metody *Check* třídy **PLib**.

Nyní uvedeme pro úplnost i seznam ostatních tříd knihovny, které nepřináší knihovně další funkčnost, ale jsou v ní pro zachování zpětné kompatibility.

Double1

Třída reprezentující skalární hodnotu v Eukleidovském prostoru. Obsahuje přetížení operátorů pro implicitní konverzi z datového typu double i na něj. Dále obsahuje property X pro čtení i zápis skalární hodnoty. Je potomkem třídy **DoubleN**.

Double2

Třída reprezentující vektor se dvěma složkami v Eukleidovském prostoru. Obsahuje property X pro čtení i zápis první složky vektoru a property Y pro přístup ke druhé složce vektoru. Je potomkem třídy **DoubleN**.

Double3

Třída reprezentující vektor se třemi složkami v Eukleidovském prostoru. Obsahuje property X, Y a Z pro čtení i zápis složek vektoru. Je potomkem třídy **DoubleN**.

Double4

Třída reprezentující vektor se čtyřmi složkami v Eukleidovském prostoru. Obsahuje property X, Y, Z a S pro čtení i zápis složek vektoru. Je potomkem třídy **DoubleN**.

Double2P

Třída reprezentující vektor se dvěma složkami v projektivním prostoru. Obsahuje property X a Y pro čtení i zápis složek vektoru. Je potomkem třídy **DoubleNP**. Přístup k homogenní složce je umožněn pomocí atributu W rodičovské třídy.

Double3P

Třída reprezentující vektor se třemi složkami v projektivním prostoru. Obsahuje property X, Y a Z pro čtení i zápis složek vektoru. Je potomkem třídy **DoubleNP**. Přístup k homogenní složce je umožněn pomocí atributu W rodičovské třídy.

Double4P

Třída reprezentující vektor se čtyřmi složkami v projektivním prostoru. Obsahuje property X, Y, Z a S pro čtení i zápis složek vektoru. Je potomkem třídy **DoubleNP**. Přístup k homogenní složce je umožněn pomocí atributu W rodičovské třídy.

3.3.2 Popis implementace některých algoritmů

Korektní implementace přetížených operátorů

Pro představu konkrétní implementace jednotlivých binárních operací zde uvedeme dva příklady, které zachycují základní myšlenky. Ostatní algoritmy jsou analogické.

Následující část kódu představuje dělení dvou projektivních skalárů.

Ukázka 3.4

```
public static Double1P operator /(Double1P a, Double1P b)
{
    // otestování znaménka homogenních složek obou vektorů
    // pokud je záporné, je provedeno prohození znamének
    // všech složek vektoru
    if (a.W < 0) PLib.DeSign(a);
    if (b.W < 0) Plib.DeSign(b);

    // provedení normalizace vzhledem k homogenní složce
    PLib.NormalizeExp(a);
    Plib.NormalizeExp(b);
```

```
    // alokování a inicializace proměnné pro výsledek
    Double1P r = new Double1P();

    // vypočtení složek výsledného vektoru
    r.D[0] = a.D[0] * b.W;
    r.W = b.D[0] * a.W;

    // kontrola platnosti výsledku
    if (!Plib.Check(r))
        // vyhození výjimky při detekované chybě
        throw new PlibException(String.Format
            (CultureInfo.CurrentCulture, "\nargument 1:{0}\n
            argument 2: {1}\nresult: {2}", a, b, r));

    // vrácení výsledku operace
    return r;
}
```

Nejdříve se provádí testování, zda jsou homogenní složky obou vektorů nezáporné. Pokud to neplatí, je provedeno prohození znamének u každé složky vektoru. Ve všech přetíženích operátorů je dbáno na udržování nezáporné homogenní složky vektorů. Poté je provedena normalizace obou operandů (více o normalizaci dále v této části kapitoly nebo v části 3.1.1). Následně je provedeno dělení operandů podle vztahů 2.11 a 2.12. Nakonec je zavolána metoda *Check*, která zjišťuje zda jsou složky výsledku platné hodnoty *double (v*íce o metodě *Check* také dále v této části kapitoly). Pokud kontrola vrací *false*, výsledek je chybný a je vyvolána výjimka s výpisem obou operandů a výsledku.

Další ukázkou je kód, který vykonává operaci porovnání dvou vektorů v projektivním prostoru.

Ukázka 3.5

```
public static bool operator ==(DoubleNP a, DoubleNP b)
{
    // uložení počtu složek vektoru pro podmínku for-cyklu
    int n = a.N;
    // porovnání počtu složek obou operandů
    if (n != b.N)
```

```
            // vyhození výjimky při nerovnosti
            throw new PLibException(String.Format
                (CultureInfo.CurrentCulture, "Cannot apply
                operator \"==\" to operands:\na: {0}\nb: {1}\n
                Dimension inequality.", a, b));

        // otestování znaménka homogenních složek obou vektorů
        // pokud je záporné, je provedeno prohození znamének
        // všech složek vektoru
        if (a.W < 0) PLib.DeSign(a);
        if (b.W < 0) Plib.DeSign(b);

        // provedení normalizace vzhledem k homogenní složce
        PLib.NormalizeExp(a);
        Plib.NormalizeExp(b);

        // deklarace proměnné pro výsledek
        bool result = true;

        // for-cyklus pro zjišťování rovnosti složek vektoru
        for (int i = 0; i < n; i++)
            // operace bitového součinu pro urychlení výpočtu
            result &= (a.D[i] * b.W == a.W * b.D[i]);

        // vrácení výsledku porovnání
        return result;
    }
```

V případě binárních operací a obecně n-rozměrných vektorů je nutné nejdříve otestovat, zda má vůbec taková operace smysl. Pokud jsou porovnávány dva vektory o různém počtu složek, je vyhozena výjimka PLibException s výpisem, že požadovaný operátor nelze aplikovat na porovnávané operandy. V opačném případě se pokračuje stejně jako u předchozí metody – je provedeno testování homogenní složky a normalizace obou operandů. Následně se porovnávají postupně všechny složky vektoru násobené homogenní složkou opačného operandu, což jsme mohli provést na základě znalostí vztahů pro homogenní souřadnice. Pro urychlení je využito bitového násobení místo dodatečné podmínky pro vyskočení z cyklu v případě nerovnosti některých odpovídajících složek vektorů.

Normalizace

Nyní prozkoumejme algoritmus použitý pro normalizaci vektorů s homogenní složkou.

Ukázka 3.6

```
public static void NormalizeExp(DoubleNP a)
{
    // získáme sekvenci bitů homogenní složky
    long W = System.BitConverter.DoubleToInt64Bits(a.W);

    // vymaskujeme bity exponentu a posuneme ho do nuly
    // máme tak bity hodnotu exponentu
    long wExp = (W & 0x7FF0000000000000) - 0x3FF0000000000000;

    // pokud je exponent nenulový, má smysl pokračovat dál
    if (wExp != 0)
    {
        // pro všechny složky vektoru
        for (int i = 0; i < a.N; i++)
        {
            // získáme sekvenci bitů
            long Q = System.BitConverter.DoubleToInt64Bits
                (a.D[i]);

            // pokud není číslo nula a má tedy smysl pokračovat
            // v úpravě jeho exponentu
            if (Q != 0)
            {
                // získáme novou hodnotu jeho exponentu jako
                // rozdíl vymaskované části a hodnoty exponentu
                // homgenní složky
                long exp = (Q & 0x7FF0000000000000) - wExp;

                // pokud došlo k přetečení exponentu
                // je vyhozena výjimka PLibException
                if (((exp >> 1) & 0x4007FFFFFFFFFFF) != 0)
                    throw new PlibException(String.Format
                        (CultureInfo.CurrentCulture, "\n
                        Exponent over/underflow during
                        normalization of vector {0},
                        component {1}", a, i));

                // uložení nové hodnoty složky jako bitový
```

```
                    // součet původní mantisy, nového exponentu
                    // a znaménka
                    a.D[i] = System.BitConverter.Int64BitsToDouble
                        ((Q & 0x000FFFFFFFFFFFFF) | exp |
                        (((Q >> 1) & 0x4000000000000000) << 1));
                }
            }
            // nakonec je uložena i nová hodnota homogenní složky
            // jako součet původní mantisy, nulového exponentu
            // a znaménka
            a.W = System.BitConverter.Int64BitsToDouble
                ((W & 0x000FFFFFFFFFFFFF) | 0x3FF0000000000000 |
                (((W >> 1) & 0x4000000000000000) << 1));
        }
    }
```

Popis normalizace je uveden v kapitole 3.1.1. Na tomto místě shrneme pouze hlavní myšlenku. Vektory v projektivním prostoru jsou normalizovány vzhledem k jejich homogenní složce. Provádí se na úrovni bitové reprezentace hodnot *double* v paměti. Hodnoty *double* jsou uloženy na 64 bitech, přičemž 1. bit připadá na znaménko, dalších 11 bitů obsahuje hodnotu exponentu a posledních 52 bitů tvoří mantisu. Každou *double* hodnotu x můžeme tedy symbolicky zapsat jako

$$x = (-1)^{znamenko} \cdot 2^{exponent - bias} \cdot 1.\text{mantisa}\,, \tag{3.3}$$

kde *bias* (někdy též označován v anglosaské literatuře jako *exponent bias*) značí hodnotu, díky které je uložený exponent vždy kladný. Jedná se o tzv. kód s posunutou nulou.

Při normalizaci se neztrácí přesnost díky tomu, že se mění pouze hodnota exponentu, nikoliv mantisy. Na začátku metody se převede *double* hodnota homogenní složky na sekvenci bitů, kterou uložíme do proměnné typu *long*. Do další proměnné typu *long* si uložíme hodnotu exponentu. To provedeme tak, že ze sekvence bitů vymaskujeme 2. až 12. bit a posuneme ho do počátku odečtením hodnoty *bias*. Pokud je v této chvíli exponent rovný nule, nemá smysl pokračovat v úpravě exponentů jednotlivých složek vektoru. V opačném případě algoritmus vstoupí do for-cyklu, který projde přes všechny složky vektoru (samozřejmě kromě homogenní). Pro každou složku se určí její bitová sekvence a z ní se vymaskuje exponent. Nová hodnota

exponentu je poté spočítána jako odečtení exponentu homogenní složky od původní hodnoty v souladu se vztahem 3.2 v kapitole 3.2.1. V extrémních případech by ovšem mohlo dojít k přetečení či podtečení exponentu. Proto je tato možnost testována pomocí bitového součinu a musí platit, že se nevyskytuje žádná jednička jinde než na bitech exponentu. Vzhledem k omezenému rozsahu proměnných *long* jsou obě hodnoty při testu bitově posunuty o jeden bit doprava. Nyní stačí už jenom uložit novou hodnotu složky jako bitový součet mantisy, exponentu a znaménka. Znaménko je ukládáno zvlášť, opět kvůli omezenému rozsahu *long*. Po skončení for-cyklu se totéž provede i s hodnotou homogenní složky, pouze s tím rozdílem, že exponent je napevno nastaven na nulu.

Check

Další hojně používanou metodou v knihovně je metoda **Check**, která slouží k ověřování platnosti vypočtených hodnot. Automaticky je volána při každém výpočtu provedeném pomocí metod přetížených operátorů, ale může být volána i explicitně pro ověření dat datových typů **DoubleN**, **DoubleNP** i **DoubleNM**. Vzhledem k podobnosti všech tří přetížení si zde uvedeme pouze jednu variantu.

Ukázka 3.7

```
public static bool Check(DoubleNM a)
{
    // deklarace a inicializace proměnné pro výsledek
    bool r = true;

    // počet všech prvků
    int n =  a.N * a.N;

    // pro všechny prvky matice a pokud r není již false
    for (int i = 0; i < n && r; i++)
    {
        // se otestuje, zda nejsou shodne s jednou
        // ze speciálních (nežádaných) hodnot
        r &= (!Double.IsInfinity(a.D[i]));
        r &= (!Double.IsNaN(a.D[i]));
    }

    // vrácení výsledku testu
```

```
        return r;
    }
```

Uvedený kód provádí kontrolu všech složek matice, jestli nejsou rovny některé ze speciální hodnot datového typu *double*. Jedná se o hodnoty kladné a záporné nekonečno a o tzv. NaN hodnotu (plné znění této zkratky je Not A Number). Pokud budeme dělit *double* hodnotu hodnotou blízkou nule, dojde k přetečení rozsahu *double* a tomu je samozřejmě nutné se při výpočtech vyhnout. V praxi se neosvědčilo zapnutí kontroly přetečení a podtečení ve Visual Studiu 2005. Nebyly detekovány všechny případy, kdy k překročení rozsahu docházelo.

3.4 Testování knihovny

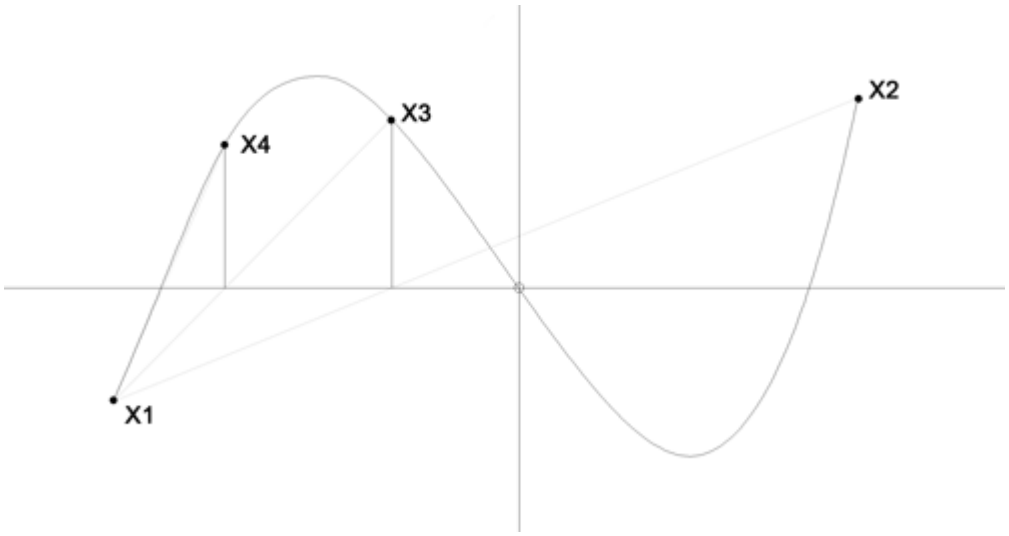
Pro testování vlastností knihovny byly implementovány dva základní algoritmy používané v numerické matematice. Nejdříve uvedeme tabulku naměřených časů pro vykonání základních aritmetických operací, viz tab 3.1, které potom dále použijeme pro odhad dob běhu jednotlivých algoritmů. Měření v rámci této kapitoly jsme provedli na procesoru AMD Athlon 64 2800+, 1.8GHz v jazyce C# na platformě .NET 2.0 se zapnutou optimalizací kódu.

Typ operace	+/-	*	/	normalizace
Doba běhu [ms]	$2{,}641 \cdot 10^{-6}$	$2{,}743 \cdot 10^{-6}$	$1{,}329 \cdot 10^{-5}$	$4{,}575 \cdot 10^{-5}$

Tab. 3.1 Doba běhu jednotlivých operací v jazyce C# na platformě .NET

3.4.1 Metoda regula falsi

První algoritmus, který zde rozebereme, je regula falsi pro výpočet $F(x) = 0$ (v anglosaské literatuře někdy uváděna jako *False Position Method*), viz obr. 3.1. Jedná se o startovací metodu, neboli takovou, která hledaný průsečík najde, pokud nějaký v zadaném intervalu existuje. Uvedeme zde základní myšlenku této metody.



Obr. 3.1 Grafické znázornění průběhu metody regula falsi

Při spuštění algoritmu je zadána funkce a krajní body intervalu, na kterém se hledá řešení, a přesnost výsledku. Počáteční podmínkou algoritmu je správně zadaný interval, který obsahuje alespoň jedno řešení rovnice $F(x) = 0$ a zároveň funkční hodnoty krajních bodů mají opačná znaménka. Výpočet probíhá iteračně. V každé iteraci se provádí následující postup. Určí se průsečík osy *x* a přímky ležící na posledních dvou známých krajních bodech. Stanovení průsečíku p s krajními body a a b se provádí podle vztahu

$$p = a + \frac{(b-a)\cdot F(a)}{F(a)-F(b)} \tag{3.4}$$

Dále se určí funkční hodnota v bodě průsečíku. Jsou určeny nové dva krajní body tak, aby platilo, že funkční hodnoty v těchto bodech mají opačné znaménko. Iterační výpočet končí, pokud rozdíl po sobě jdoucích bodů je menší než požadovaná přesnost (případně pokud je dosaženo maximálního počtu iterací).

Algoritmus jsme částečně optimalizovali tak, aby se neprováděly některé zbytečné operace. Výpis zdrojového kódu, kterým se testovalo v Eukleidovském prostoru pomocí proměnných *double*, je uveden v ukázce 3.8. Pro kratší zápis se předpokládá splnění počátečních podmínek metody.

Ukázka 3.8

```
public static Double1 FooEukl(double a)
{
    // výpočet funkční hodnoty funkce
    return (0.04 * a * a + 2 * a – 100);
}

public static double SolveRegulaFalsiEukl(double a, double b)
{
    double fa, fb, fc, swap;
    double c = 0;
    int count = 0;
    double del = 2 * EPS;

    // výpočet funkčních hodnot krajních bodů intervalu
    fa = FooEukl(a);
    fb = FooEukl(b);

    // zajištění toho, aby bod [a, fa] ležel pod osou x a
    // ušetřilo se tak násobení při rozhodování o novém
    // krajním bodu intervalu
    if (fa > 0)
    {
        // prohození bodu i jejich funkčních hodnot
        swap = a; a = b; b = swap;
        swap = fa; fa = fb; fb = swap;
    }

    // dokud iterace nedosáhla požadované přesnosti řešení
    // případně nedosáhla maximálniho počtu iterací, což je
    // pojistka, aby výpočet vždy doběhl v konečném čase
    while (count < MAX_POCET && Math.Abs(del) > EPS)
    {
        // výpočet prùsečíku osy x a přímky procházející
        // krajními body intervalu
        c = a + (b - a) * fa / (fa - fb);
        // a výpočet jeho funkční hodnoty
        fc = FooEukl(c);

        // pokud je tato funkční hodnota záporná, je
        // nahrazen bod a, pro který vždy platí, že jeho
        // funkční hodnota je také záporná
        if (fc < 0)
        {
```

```
            // určení rozdílu posledních dvou po sobě
            // jdoucích bodů intervalu
            del = a - c;
            a = c;
            fa = fc;
        }
        // v opačném případě je nahrazen bod b
        else
        {
            // určení rozdílu posledních dvou po sobě
            // jdoucích bodů intervalu
            del = b - c;
            b = c;
            fb = fc;
        }
        // zvýšení počtu provedených iterací
        count++;
    }
    // vrácení výsledku metody
    return c;
}
```

V této verzi algoritmu, i ve všech ostatních zde uvedených, je prováděn test ukončení výpočtu s požadovanou přesností. Pokud je rozdíl po sobě jdoucích bodů na jednom z konců intervalu menší, než zadaná přesnost, je výpočet ukončen. V rámci metody je možné také zvolit test porovnávající funkční hodnotu nově nalezené iterace s přesností.

Druhá realizovaná verze téhož algoritmu, tentokrát však v projektivním prostoru s využitím knihovny PLib a datového typu **Double1P**, je v ukázce 3.9.

Ukázka 3.9

```
public static Double1P FooPLib(Double1P a)
{
    // výpočet funkční hodnoty v projektivním prostoru
    return (0.04 * a + 2) * a - 100;
}

public static Double1P SolveRegulaFalsiPLib(
        Double1P a, Double1P b)
{
    Double1P fa, fb, fc, swap;
```

```
Double1P c = new Double1P(0);
int count = 0;
Double1P del = new Double1P(2 * EPS);

// výpočet funkčních hodnot krajních bodů intervalu
fa = FooPLib(a);
fb = FooPLib(b);

// zajištění toho, aby bod [a, fa] ležel pod osou x a
// ušetřilo se tak násobení při rozhodování o novém
// krajním bodu intervalu
if (fa > 0)
{
    swap = a; a = b; b = swap;
    swap = fa; fa = fb; fb = swap;
}

try
{
// dokud iterace nedosáhla požadované přesnosti řešení
// případně nedosáhla maximálního počtu iterací, což je
// pojistka, aby výpočet vždy doběhl v konečném čase
while (count < MAX_POCET && del > EPS)
{
    // vypočtení průsečíku osy x a přímky procházející
    // krajními body intervalu
    c = (a + (b - a) * fa / (fa - fb));
    // výpočet funkční hodnoty
    fc = FooPLib(c);
    // pokud je tato funkční hodnota záporná, je
    // nahrazen bod a, pro který vždy platí, že jeho
    // funkční hodnota je také záporná
    // stačí zjistit znaménko složky x, protože
    // homogenní složka w je vždy kladná
    if (fc.X < 0)
    {
        // určení rozdílu posledních dvou po sobě
        // jdoucích bodů intervalu
        del = a - c;
        // vzhledem k charakteru algoritmu není potřeba
        // novou instanci a stačí předat referenci
        a = c;
        fa = fc;
    }
    else
```

```
            {
                // určení rozdílu posledních dvou po sobě
                // jdoucích bodů intervalu
                del = b - c;
                // vzhledem k charakteru algoritmu není potřeba
                // novou instanci a stačí předat referenci
                b = c;
                fb = fc;
            }
            // je třeba zajistit absolultní hodnotu proměnné del
            // aby bylo možné provádět test na přesnost
            del.X = Math.Abs(del.X);
            // zvýšení počtu iterací
            count++;
        }
        }
        // odchytávání výjimky, kterou vyhatuje knihovna Plib
        // v případě detekování přetečení či podtečení přesnosti
        catch (PLibException)
        {
            return c;
        }
        // vrácení výsledku
        return c;
    }
```

Třetí verze téže metody používá stejně jako druhá verze projektivní prostor pro odstranění používání operace dělení. Nevyužívá přitom datových typů poskytovaných knihovnou PLib, ale zahrnuje vlastní implementaci veškerých operací v projektivním prostoru, viz ukázka 3.10.

Ukázka 3.10

```
public static void FooProj(double a, double aw,
        out double c, out double cw)
{
    c = 0.04 * a * a + (2 * a - 100 * aw) * aw;
    cw = aw * aw;
}

public static double SolveRegulaFalsiProj(double a, double b)
{
    // deklarování všech potřebných proměnných včetně těch,
    // co představují homogenní složky
    double fa, fb, swap;
```

```
double faw = 1; double fbw = 1;
double fc = 0; double fcw = 1;
double aw = 1; double bw = 1;
double c = 0; double cw = 1;
int count = 0;
double del = 2 * EPS; double delw = 1;
double semires;

// výpočet funkční hodnoty bodu a
FooProj(a, 1, out fa, out faw);
// a jeho normalizace
NormalizeExp(fa, faw, out fa, out faw);
// totéž pro druhý krajní bod intervalu
FooProj(b, 1, out fb, out fbw);
NormalizeExp(fb, fbw, out fb, out fbw);

// zajištění toho, aby bod [a, fa] ležel pod osou x a
// ušetřilo se tak násobení při rozhodování o novém
// krajním bodu intervalu
if (fa > 0)
{
    swap = a; a = b; b = swap;
    swap = fa; fa = fb; fb = swap;
}

// dokud iterace nedosáhla požadované přesnosti řešení
// případně nedosáhla maximálniho počtu iterací, což je
// pojistka, aby výpočet vždy doběhl v konečném čase
while (count < MAX_POCET && Math.Abs(del) > EPS)
{
    // výpočet obou složek průsečíku osy x a přímky
    // procházející krajními body intervalu
    // je použita pomocná proměnná, aby se nepočítalo
    // totéž zbytečně dvakrát
    semires = aw * bw * faw * (fa * fbw - fb * faw);
    c = a * semires + fa * (b * aw - a * bw) * faw * fbw;
    cw = aw * semires;
    // je provedena normalizace
    NormalizeExp(c, cw, out c, out cw);
    // výpočet funkční hodnoty
    FooProj(c, cw, out fc, out fcw);
    // a normalizace této hodnoty
    NormalizeExp(fc, fcw, out fc, out fcw);

    // pokud je tato funkční hodnota záporná, je
```

```
            // nahrazen bod a, pro který vždy platí, že jeho
            // funkční hodnota je také záporná
            // stačí zjistit znaménko proměnné fc, protože
            // homogenní složka fcw je vždy kladná
            if (fc < 0)
            {
                // určení rozdílu posledních dvou po sobě
                // jdoucích bodů intervalu
                del = a * cw - c * aw;
                delw = aw * cw;
                a = c; aw = cw;
                fa = fc; faw = fcw;
            }
            else
            {
                // určení rozdílu posledních dvou po sobě
                // jdoucích bodů intervalu
                del = b * cw - c * bw;
                delw = bw * cw;
                b = c; bw = cw;
                fb = fc; fbw = fcw;
            }
            // zvyýšení počtu iterací
            count++;
        }
        // vrácení výsledku v Eukleidovském prostoru
        return c / cw;
    }
```

Z uvedených výpisů zdrojových kódů si můžeme udělat přehled o tom, že implementace této metody s použitím různých přístupů je velmi podobná. Následující tabulka shrnuje počet jednotlivých elementárních operací mezi proměnnými typu *double* v jedné iteraci pro každý z uvedených algoritmů. Uvedené počty zahrnují i výpočty funkčních hodnot.

Typ operace	+/-	*	/	normalizace
Eukleidovský prostor	6	3	1	0
Projektivní prostor s PLib	6	23	0	18
Projektivní prostor bez PLib	6	22	0	2

Tab. 3.2 Počty elementárních operací v jednotlivých přístupech

Nyní vyčíslíme předpokládanou výpočetní náročnost jednotlivých přístupů. Znázorníme je

v následující tabulce 3.3. Hned za touto tabulkou uvedeme další, ve které uvedeme skutečně naměřené časy. Časy jsou uvedeny v milisekundách.

Verze algoritmu	Eukleidovský prostor	Projektivní prostor s PLib	Projektivní prostor bez PLib
Předpokládaná doba výpočtu [ms]	$1868{,}25 \cdot 10^{-6}$	$45121{,}75 \cdot 10^{-6}$	$8384{,}6 \cdot 10^{-6}$

Tab. 3.3 Odhadovaná doba výpočtu při 50 iteracích

Verze algoritmu	Eukleidovský prostor	Projektivní prostor s PLib	Projektivní prostor bez PLib
Předpokládaná doba výpočtu [ms]	$5570{,}3 \cdot 10^{-6}$	$93431{,}7 \cdot 10^{-6}$	$10184{,}6 \cdot 10^{-6}$

Tab 3.4 Skutečná doba výpočtu při 50 iteracích

Porovnáme tedy získané výsledky. V předpokládané časové náročnosti byly poměry následující – Eukleidovský prostor vycházel oproti projektivnímu s použitím PLib 24,15-krát rychlejší, bez použití PLib 4,49-krát rychlejší. PLib, především díky časté normalizaci při každé prováděné operaci, měla vyjít oproti verzi v homogenních souřadnicích bez jejího použití 5,38-krát pomalejší. Ve skutečných časech se poměry změnily. Eukleidovský prostor poskytoval 16,77-krát rychlejší výpočet oproti PLib a pouze 1,83-krát rychlejší výpočet než u přímo implementovaného projektivního prostoru. V rámci projektivního prostoru došlo k navýšení poměru rychlostí ku prospěchu verze bez PLib na 9,17.

Porovnání obou verzí v projektivním prostoru tedy vyznívá pro přímou implementaci. To však lze přičíst tomu, že v implementaci není prováděna při každé vypočtené operaci normalizace. Ta je prováděna pouze v kritických částech algoritmu po výpočtu průsečíku a výpočtu funkční hodnoty. Provádění normalizace po každé operaci by sice bylo možné, ale došlo by velkému prodloužení a znepřehlednění kódu, které je při programování nežádoucí. Zatímco pomocí PLib zůstává algoritmus poměrně přehledný. Z tohoto hlediska knihovna plní jeden ze svých cílů – usnadňovat implementaci algoritmů v projektivním prostoru.

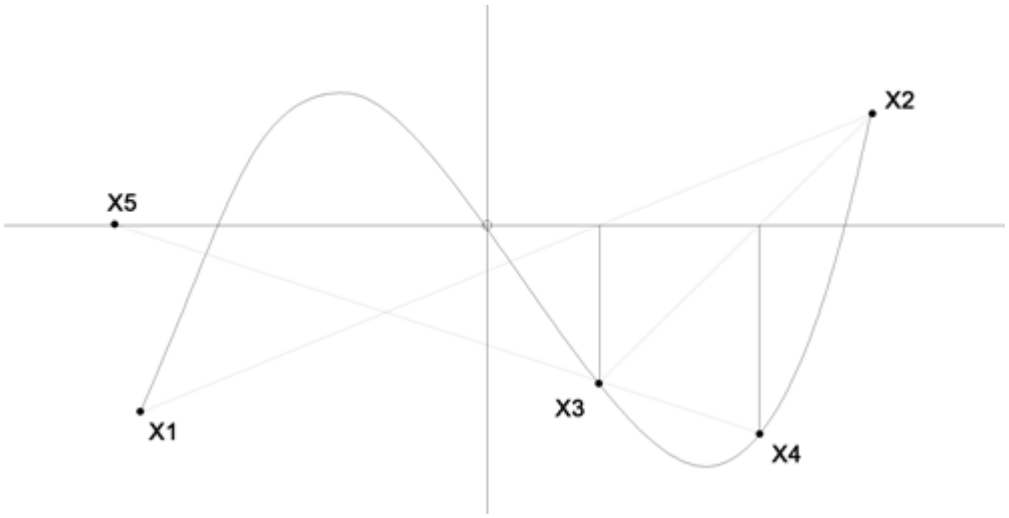
Uvedené časy byly dosaženy při hledání kořene funkce $F(x) = (0{,}04 \cdot a)^2 + 2 \cdot a$ v intervalu $\langle -10, 3000 \rangle$, viz tab. 3.4.. Tato funkce byla zvolena proto, že je relativně plochá a

snadno se tedy projeví nepřesnosti při výpočtech. Při jiném zadání se samozřejmě všechny tři metody chovají různě, ale proporcionálně naměřené rychlosti přibližně odpovídají zde uvedeným. Pro lepší porovnání časů měření byl nastaven pro všechny metody shodný maximální počet iterací na hodnotu 50. Tato hodnota byla zvolena tak, aby se naměřené časy daly považovat za dostatečně přesné a nedocházelo ke zkreslení z důvodu počáteční alokace paměti proměnných.

Poté bylo omezení počtu iterací nastaveno na hodnotu 1000, která metody neukončovala předčasně, pokud konvergovaly, ale zároveň působila jako pojistka při divergenci metody. Vzhledem k charakteru algoritmu vykazovaly všechny implementované metody relativně velkou stabilitu. Lišily se pouze počty provedených iterací a přesnost dosaženého výsledku. V těchto ohledech velmi záleželo na zadaných počátečních podmínkách a testované funkci. Zpravidla však vycházely hodnoty pro Eukleidovský prostor a projektivní prostor s využitím PLib velice podobně, takže nelze obecně říci, který z těchto dvou přístupů je na tom lépe. Naopak verze implementace projektivního prostor bez PLib dávala zpravidla méně přesné výsledky, pokud byly zadány více náročné počáteční podmínky.

3.4.2 Metoda sečen

Dalším testovaným algoritmem numerické matematiky je metoda sečen. Jedná se o metodu patřící do skupiny zpřesňujících metod hledání $F(x) = 0$. Její myšlenka je založena na velmi podobném principu – hledání průsečíku přímky s osou x. Vztah pro výpočet průsečíku je stejný jako u metody regula falsi, viz (3.4). Od předchozí metody se liší pouze v tom, že jako body přímky, která určuje další iteraci, bere poslední dvě provedené iterace. Tedy nikoli jako u metody regula falsi body, pro něž platí, že součin jejich funkčních hodnot musí být záporný. Algoritmem metody sečen není jisté, zda je průsečík nalezen. Metoda vykazuje relativně malou stabilitu pro obecně zadané funkce, její výhoda však spočívá v tom, že jako vstupní parametry nevyžaduje krajní body intervalu, ve kterém existuje průsečík. Grafické znázornění metody je na obrázku 3.2.



Obr. 3.1 Grafické znázornění průběhu metody sečen

V ukázce 3.11 uvedeme zdrojový kód, kterým byla testována metoda sečen v Eukleidovském prostoru. Po ní bude následovat i výpis metody s použitím homogenních souřadnic. A to jak s použitím PLib v ukázce 3.12, tak i s použitím přímé implementace v ukázce 3.13.

Ukázka 3.11

```
public static double SolveSecantEukl(double a, double b)
{
    // deklarace a inicializace všech potřebných proměnných
    double fa, fb;
    double fc = 1;
    double c = 0;
    count = 0;
    double del = 2 * EPS;

    // výpočet funkčních hodnot v obou zadaných bodech
    fa = FooEukl(a);
    fb = FooEukl(b);

    // dokud iterace nedosáhla požadované přesnosti řešení
    // případně nedosáhla maximálniho počtu iterací, což je
    // pojistka, aby výpočet vždy doběhl v konečném čase
    while (count < MAX_POCET && Math.Abs(del) > EPS)
    {
        // je vypočten průsečík osy x a přímky procházející
```

```
            // posledními dvěma body iterace
            c = b - ((fb * (b - a)) / (fb – fa));
            // vypočtení funkční hodnoty posledního
            // nalezeného bodu
            fc = FooEukl(c);

            // přiřazení posledních dvou hodnot iterace
            // včetně jejich funkčních hodnot
            a = b;
            fa = fb;
            b = c;
            fb = fc;

            // určení rozdílu dvou po sobě jdoucích bodů iterace
            del = b - a;

            // zvýšení počtu provedených iterací
            count++;
        }
        // vrácení výsledků
        return c;
    }
```

Jedná se klasickou implementaci metody sečen s použitím proměnných typu double a s testem přesnosti na rozdíl po sobě jdoucích iterací. Nalezení průsečíku se provádí podle stejného vztahu 3.4 jako u metody regula falsi.

Ukázka 3.12

```
public static Double1P SolveSecantPLib(Double1P a,
        Double1P b)
{
    // deklarace a inicializace všech potřebných proměnných
    Double1P fa, fb;
    Double1P fc = new Double1P(1);
    Double1P c = new Double1P(0);
    count = 0;
    Double1P del = new Double1P(2 * EPS);

    // výpočet funkčních hodnot v obou zadaných bodech
    fa = FooPLib(a);
    fb = FooPLib(b);
```

```
    try
    {
    // dokud iterace nedosáhla požadované přesnosti řešení
    // případně nedosáhla maximálniho počtu iterací, což je
    // pojistka, aby výpočet vždy doběhl v konečném čase
    while(count < MAX_POCET && del > EPS)
    {
        // vypočtení průsečíku osy x a přímky procházející
        // posledními dvěma body iterace
        c = b - ((fb * (b - a)) / (fb – fa));
        // vypočtení jeho funkční hodnoty
        fc = FooPLib(c);

        // přiřazení posledních dvou hodnot iterace
        // včetně jejich funkčních hodnot
        a = b;
        fa = fb;
        b = c;
        fb = fc;

        // určení rozdílu dvou po sobě jdoucích bodů iterace
        del = b – a;
        // určení absolutní hodnoty proměnné del
        // stačí provést pro složku, protože homogenní
        // složka w je udržovaná nezáporná knihovnou
        del.X = Math.Abs(del.X);

        // zvýšení počtu provedných iterací
        count++;
    }
    }
    // odchytávání výjimky vyhazované knihovnou
    // při detekování přetečení či podtečení přesnosti
    catch (PLibException)
    {
        // vrácení posledního známého platného výsledku
        return c;
    }
    // vrácení výsledku
    return c;
}
```

Tato verze pracuje v projektivním prostoru s využitím knihovny PLib. Následuje též verze v projektivním prostoru, avšak bez použití PLib.

Ukázka 3.13

```
public static double SolveSecantProj(double a, double b)
{
    // deklarace a inicializace všech potřebných proměnných
    double fa, faw, fb, fbw;
    double aw = 1;
    double bw = 1;
    double fc = 1;
    double fcw = 1;
    double c = 0;
    double cw = 1;
    count = 0;
    double del = 2 * EPS;
    double delw = 1;
    double semires;

    // výpočet funkčních hodnot
    FooProj(a, aw, out fa, out faw);
    FooProj(b, bw, out fb, out fbw);

    // dokud iterace nedosáhla požadované přesnosti řešení
    // případně nedosáhla maximálního počtu iterací, což je
    // pojistka, aby výpočet vždy doběhl v konečném čase
    while (count < MAX_POCET && Math.Abs(del) > EPS * delw)
    {
        // výpočet obou složek průsečíku osy x a přímky
        // procházející krajními body intervalu
        // je použita pomocná proměnná, aby se nepočítalo
        // totéž zbytečně dvakrát
        semires = aw * bw * faw * (fa * fbw - fb * faw);
        c = a * semires + fa * (b * aw - a * bw) * faw * fbw;
        cw = aw * semires;
        // je provedena normalizace
        NormalizeExp(c, cw, out c, out cw);
        // výpočet funkční hodnoty bodu průsečíku
        FooProj(c, cw, out fc, out fcw);
        // opět je provedena normalizace
        NormalizeExp(fc, fcw, out fc, out fcw);

        // přiřazení posledních dvou hodnot iterace
        // včetně jejich funkčních hodnot
        a = b; aw = bw;
        fa = fb; faw = fbw;
        b = c; bw = cw;
```

```
            fb = fc; fbw = fcw;

            // výpočet rozdílu posledních dvou bodů iterace
            del = b * aw - a * bw;
            delw = aw * bw;

            // zvýšení počtu provedených iterací
            count++;
        }

        // vrácení výsledku
        return c / cw;
    }
```

V následující tabulce 3.5 opět shrneme počty prováděných jednotlivých elementárních operací mezi proměnnými typu double.

Typ operace	+/-	*	/	normalizace
Eukleidovský prostor	6	3	1	0
Projektivní prostor s PLib	6	21	0	18
Projektivní prostor bez PLib	6	22	0	2

Tab. 3.5 Počty elementárních operací v jednotlivých verzích algoritmu

Nyní opět vyčíslíme předpokládanou výpočetní náročnost jednotlivých přístupů. Znázorníme je v následující tabulce 3.6. Dále v tabulce 3.7 uvedeme rovnou i naměřené počty milisekund, které jednotlivé metody skutečně běžely na procesoru AMD Athlon 64 2800+, 1,8GHz.

Verze algoritmu	Eukleidovský prostor	Projektivní prostor s PLib	Projektivní prostor bez PLib
Předpokládaná doba výpočtu [ms]	$373{,}65 \cdot 10^{-6}$	$9024{,}35 \cdot 10^{-6}$	$1676{,}92 \cdot 10^{-6}$

Tab. 3.6 Odhadovaná doba výpočtu při 10 iteracích

Verze algoritmu	Eukleidovský prostor	Projektivní prostor s PLib	Projektivní prostor bez PLib
Skutečná doba výpočtu [ms]	$1207{,}0 \cdot 10^{-6}$	$21930{,}7 \cdot 10^{-6}$	$1981{,}2 \cdot 10^{-6}$

Tab 3.7 Skutečná doba výpočtu při 10 iteracích

Ze získaných výsledků vidíme, že řešení v Eukleidovském prostoru mělo být podle předpokladu zhruba 24,15-krát rychlejší, ve skutečnosti to bylo téměř 18,17-krát rychlejší. Tytéž porovnání mezi Eukleidovským prostorem a projektivním bez knihovny PLib vyznívají opět příznivěji pro Eukleidovský prostoru – v předpokládaném čase měl algoritmus běžet 4,48-krát rychleji, ve skutečnosti pouze 1,64-krát rychleji. Srovnání časů v projektivním prostoru vycházelo podle všech očekávání jako rychlejší bez použití PLib – předpoklad byl 5,38-krát rychlejší průběh, skutečně naměřené časy ukazují 11,07-krát kratší čas pro přímou implementaci homogenních souřadnic. Tyto časy byly naměřeny při nalézání kořene funkce $F(x) = (0{,}04 \cdot a)^2 + 2 \cdot a - 100$ s počátečními body -10, 200.

Opět platí, že veškeré testování zvlášť tohoto algoritmu silně závisí na volbě počátečních podmínek a zadané funkci. I při správném zadání počátečních podmínek, které vede ke konvergenci, je metoda citlivá na přesnost vypočítaných iterací. Díky tomu jsme mohli snáze porovnávat stabilitu jednotlivých přístupů. Ve většině testovaných funkcí i zadaných počátečních bodů vykazovala největší dosaženou přesnost verze využívající homogenních souřadnic a PLib. Tato skutečnost se těžko dá dokázat exaktně, ale takové bylo provedeno pozorování. Mezi testované funkce patřily polynomy n-tého stupně jejichž funkční hodnoty jsme schopni vyjádřit pomocí základních aritmetických operací. Druhá verze v projektivním prostoru naopak vykazovala nejnižší stabilitu výpočtů. Její výsledky byly dosahovány ve větším počtu iterací avšak i přesto často více nepřesné, než u ostatních dvou implementací. Tato skutečnost je připisována tomu, že v průběhu výpočtů nebyla prováděna normalizace po každé aritmetické operaci tak jako v případě PLib. Implementace takového přístupu, kde by se prováděla normalizace při každé operaci by kód velice prodloužila a znepřehlednila. Kód by tím byl velmi náchylný na chyby programátora. Z tohoto pohledu knihovna poskytuje pohodlnou práci a jak již bylo řečeno plní jednu z jejích hlavních úloh usnadnění vývoje algoritmů v projektivním prostoru.

3.5 Návrh na optimalizaci knihovny

V současné verzi knihovny jsou modifikátory přístupu ke všem složkám vektoru *public*. Z tohoto důvodu se provádí při volání metod přetížených operátorů nejdříve testování homogenní

složky. Kontroluje se, zda je nezáporná a můžou tedy být korektně provedeny výpočty. V opačném případě se provede otočení znaménka u všech složek vektoru. Dalším krokem je provedení normalizace pro všechny operandy v homogenních souřadnicích, jejichž homogenní složka má exponent jiný než nula. Vlivem těchto kontrol dochází k výraznému zpomalení provádění algoritmů (viz srovnání časů v kapitolách 3.3.2 a 3.3.1).

Navrhované optimalizace by prováděly pouze jednu normalizaci výsledku operace. Pro ošetření situací, kdy jsou složky vektorů nastavovány přímo uživatelem, by se zakázal přímý přístup k jednotlivým složkám vektorů a použily by se *property metody*, což jsou speciální konstrukce jazyka *C#* pro přístup privátním parametrům tříd. V těchto metodách by se při každé změně homogenní složky vektoru mohla volat normalizace. Stejně tak by se tento postup použil v konstruktorech tříd. Tímto by uživatel neměl, díky provádění kontrol okamžitě, možnost získat při provádění výpočtů jiný než normalizovaný vektor. Tutéž myšlenku lze aplikovat i na kontrolu nezápornosti homogenní složky. Tu by bylo zapotřebí volat pouze na výsledky operace dělení, kde může dojít ke změně znaménka homogenní složky, a samozřejmě při volání v *property* metodách a konstruktorech.

4 Závěr

Závěrem práce shrňme základní vlastnosti implementované knihovny. Knihovna je implementována v jazyce C# pod platformou Microsoft .NET Framework. Usnadňuje testování algoritmů pracujících v projektivním prostoru.

Algoritmy v projektivním prostoru může programátor implementovat sám bez použití knihovny, ale knihovna mu ulehčí práci při testování funkčnosti navrhovaných algoritmů. Proto je výhodné ji využít, pokud není kladen důraz na rychlost výpočtů.

Byla vytvořena podpora pro n-rozměrné vektory, matice i pro vektory s homogenní složkou. Pro práci s vektory a maticemi byly implementovány všechny základní matematické operace – obousměrný převod mezi projektivním a Eukleidovským prostorem, sčítání, odečítání, násobení, dělení, porovnávání vektorů, skalární i vektorový součin. Vektorový součin je možný pouze pro vektory omezené velikosti. Zápis algoritmů pomocí vytvořených tříd umožňuje používání operátorů namísto volání metod. Více o poskytovaných funkcí v příloze B – Referenční příručka knihovny PLib.

Knihovna byla navržena tak, aby podporovala větší stabilitu algoritmů, zejména co se týká používání operace dělení ve výpočtech. Nahrazení operace dělení násobením a používání normalizace částečně řeší problém přesnosti desetinných čísel.

Práce mě utvrdila v tom, že má rozhodně cenu se ve výpočetních systémech zabývat optimalizací algoritmů nejenom z hlediska výpočetní složitosti, ale také z hlediska stability.

Práce byla realizována v rámci projektu „Virtuální vědecko-pedagogické centrum počítačové grafiky a vizualizace dat, projekt MŠMT ČR 2C 06002“.

Knihovna byla použita a ověřena v rámci předmětu KIV/APG. Dále byla v rámci projektu Virtual využita Ing. J. Skálou pro výpočty při konstrukci Delaunayovy triangulace a Voronoiových diagramů v homogenních souřadnicích.

5 Přehled zkratek

E^n – Eukleidovský prostor dimenze n

P^n – projektivní prostor dimenze n

VS.NET – Microsoft Visual Studio .NET 2005

6 Seznam použité literatury a zdrojů

[1] *Skala, V.*: "Intersection Computation in Projective Space using Homogeneous Coordinates", osobní komunikace, 2006

[2] *Ondračka, V.*: "Technická zpráva v rámci oborového projektu", *http://herakles.zcu.cz/education/apg/plib*, 2006

[3] *Skala, V.*: "Length, Area and Volume Computation in Homogeneous Coordinates", *International Journal of Image and Graphics*, Vol. 6, No. 4, pp 625–639, World Scientific Publishing Company, 2006

[4] *Skala, V., Kaiser, J., Ondračka, V.*: "Library for Computation in the Projective Space", 6th International Conference Proceedings of Aplimat 2007, STU Bratislava, Slovenská republika, 2007, (ISBN: 978-80-969562-8-9), 2007

[5] *Skála, J.*: "Projektivní geometrie, Delaunayova triangulace a Voronoiovy diagramy", Technická dokumentace v rámci projektu VIRTUAL, 2007

[6] *Weisstein, Eric W.*: "Projective Space", *MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/ProjectiveSpace.html*

[7] *Weisstein, Eric W.*: "Homogeneous Coordinates", *MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/HomogeneousCoordinates.html*

[8] *Weisstein, Eric W.*: "Homogeneous Coordinates", *MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/HomogeneousCoordinates.html*

[9] Wikipedia: "Homogeneous Coordinates" *http://en.wikipedia.org/wiki/Homogeneous_coordinates*

[10] Wikipedia: "Projective Space" *http://en.wikipedia.org/wiki/Projective_space*

[11] Wikipedia: "Double Precision" *http://en.wikipedia.org/wiki/Double_precision*

Příloha A : Uživatelská příručka

A.1 Úvod

PLib je knihovna pro testování algoritmů v projektivním prostoru. Je distribuována jako DLL modul. Nejdříve se seznámíme s požadavky knihovny a s tím, jak začít nový projekt s PLib. Dále si představíme její datové typy a vysvětlíme si jejich používání na konkrétních příkladech. V závěru se zaměříme na rady a varování před nejčastějšími potížemi při práci s PLib.

Některé části této příručky jsou převzaty z [2].

A.2 Technické požadavky

Pro běh samotné knihovny pod systémy Microsoft Windows je nutné mít nainstalován Microsoft .NET Framework. Funkčnost na ostatních systémech není známa a pravděpodobně nebude podporována.

Pro psaní vlastních kódů s PLib je třeba vlastnit a mít nainstalováno Microsoft Visual Studio 2005.

A.3 Vytvoření projektu s PLib

Pro správné vytvoření projektu stačí dodržet tyto 3 kroky:

1) Spustit Visual Studio 2005, vytvořit nový projekt v jazyce C# (např. konzolovou aplikaci).

2) Přidat do referencí projektu knihovnu PLib.dll (Add reference... > Browse)

3) Na začátek zdrojového kódu všech tříd, ve kterých budete chtít knihovnu používat, přidat jmenný prostor (“`using Zcu.MathUtils;`“).

A.4 Datové typy

Kompletní seznam datových typů včetně popisu i popisu jejich konstruktorů a přetížených operátorů naleznete v referenční příručce PLib (příloha B).

Vektory v Eukleidovském prostoru představují datové typy **Double1**, **Double2**, **Double3**, **Double4** a **DoubleN**. Jejich rozdíl je pouze v počtu podporovaných složek. **Double1** podporuje jednu složku (jedná se tedy o skalární hodnotu), **Double2** obsahuje dvě složky, **Double3** tři složky, **Double4** čtyři složky a konečně **DoubleN** reprezentuje vektor Eukleidovského prostoru o n složkách.

Vektory v projektivním prostoru představují datové typy **Double1P**, **Double2P**, **Double3P**, **Double4P** a **DoubleNP**. Sémantika pojmenování je shodná s pojmenováním vektorů v Eukleidovském prostoru. Vektory v projektivním prostoru (někdy nazývané také jako projektivní vektory) obsahují navíc homogenní složku.

V knihovně jsou také podporovány matice v Eukleidovském prostoru. Ty jsou přístupné pomocí datového typu **DoubleNM**.

A.5 Práce s knihovnou

Práce s přetíženými operátory

Nejdříve si ukážeme příklad práce se základními operacemi, které lze provádět v rámci poskytovaných datových typů. Práci s přetíženými operátory předvedeme na násobení dvou matic, matice s projektivním vektorem a dvou projektivních vektorů. Uvedeme komentovaný výpis kódu, který takové operace provádí.

Ukázka A.1

```
// deklarace matice M1 a její inicializace hodnot
// z jednorozměrného pole double
DoubleNM M1 = new DoubleNM(new double[]
                  { 2, 3, 1, 4, 6, 2, 3, 1, 2 });

// deklarace matice M2 a její inicializace hodnot
// z dvourozměrného pole double
```

```
DoubleNM M2 = new DoubleNM(new double[,]
                  { { 2, 3, 4 }, { 5, 6, 7 }, { 8, 9, 1} });

// deklarace proměnné pro výsledek operace
DoubleNM M;

// deklarace a inicializace projektivního vektoru pomocí
// obecného datového typu DoubleNP celkem se čtyřmi složkami
// včetně homogenní
DoubleNP v1 = new DoubleNP(new double[] { 1, 2, 3 }, 3);

// deklarace a inicializace projektivního vektoru pomocí
// datového typu Double3P, homogenní složka nebyla zadána
// a bude defaultně nastavena na 1
Double3P v2 = new Double3P(4, 5, -1);

// deklarace proměnné pro výsledek operace
DoubleNP v;

// intuitivní zápis násobení dvou matic
M = M1 * M2;
// a výpis výsledku
Console.WrileLine("Výsledek násobení matic: " + M.ToString());

// násobení matice a vektoru
// výsledek je typu DoubleNP celkem se čtyřmi složkami
v = M1 * v1;

// skalární součin dvou vektorů
// výsledek součinu je typu Double1P, ale díky implicintínu
// přetypování je celkový výsledek typu DoubleNP celkem
// se dvěma složkami
v = v1 * v2;
// výpis výsledku
Console.WrileLine("Výsledek násobení vektorů: " +
v.ToString());
```

V úvodu provedeme deklarace použitých proměnných. Provede se deklarace matice M1 a její inicializace hodnot z ednorozměrného pole double, deklarace matice M1 a její inicializace hodnot z dvourozměrného pole double a deklarace proměnné pro výsledek násobení matic. Totéž, co s proměnnými pro matice, se provede i s proměnnými pro vektory – deklarace a následná inicializace některým z možných konstruktorů. Po vytvoření příslušných proměnných s nimi už

můžeme přímo pracovat – provést násobení a výpis výsledků.

Práce s metodami knihovny

Druhým příkladem je výpočet průsečíku dvou přímek v rovině. Opět uvedeme zdrojový kód prokládaný komentáři. Tato část je převzata z [2] a lehce modifikována.

Ukázka A.2

```
// vytvoříme bod v projektivním prostoru
Double2P X0 = new Double2P(new double[] { 0, 0 });
// a směrový vektor usečky v projektivním prostoru (w = 0)
Double2P S = new Double2P(2, 1, 0);

// pro určení i druhé přímky deklarujeme další dva body
// pomocí tohoto konstruktoru se automaticky nastaví w = 1
Double2P X2 = new Double2P(1, 2);
// vytvoření bodu s nulovým vektorem složek a w = 1
Double2P X3 = new Double2P();
// přímý přístup ke složce bodu X3
X3.X = 3;

// vektorový součin pro určení první přímky
Double2P p1 = PLib.Cross(X0, new Double2P(X0 + S));
// vektorový součin pro určení druhé přímky
Double2P p2 = PLib.Cross(X2, X3);

// získání průsečíku taktéž s pomocí vektorového součinu
DoubleNP X = PLib.Cross(p1, p2);
```

Nejdříve deklarujeme proměnné s homogenní složkou, která nám umožní výpočet průsečíku použitím vektorového součinu. Záměrně byly použity různé typy konstruktorů, aby byly demonstrovány různé možnosti v zacházení s datovými typy. Proměnná X0 je vytvořena pomocí konstruktoru z rodičovské třídy **DoubleNP**, proměnná S byla vytvořena ze všech tří specifikovaných hodnot (homogenní složka nastavená na 0 značí, že se jedná o vektor, nikoliv bod v projektivním prostoru), X2 byla vytvořena ze zadaných hodnot, přičemž homogenní složka byla nastavena defaultně na 1, a proměnná X3 je vytvořena pomocí prázdného konstruktoru, který nastaví všechny složky vektoru na 0 a homogenní složku na 1. Všechny datové typy umožňují přímý přístup ke všem složkám. Tímto způsobem je provedena změna první složky

proměnné X3. Nyní, když máme zadány body a vektory, můžeme určit přímky pomocí vektorového součinu. V případě určování přímky p1 vidíme, že jsme použili trik, kterým lze obejít nemožnost přetypování z **DoubleNP** na **Double2P**. V posledním kroku taktéž pomocí vektorového součinu určíme výsledný průsečík dvou přímek.

A.6 Rady a varování pro práci s PLib

Přiřazování proměnných

Při práci s datovými typy knihovny je třeba dávat velký pozor na to, že se jedná o třídy a platí pro ně tedy totéž, co jakékoliv jiné proměnné referenčního typu. Každá taková proměnná neobsahuje přímo data, ale pouze referenci na místo v paměti, kde leží. Pokud dojde k přiřazení těchto proměnných, nezkopírují se celá data do jiné části paměti, ale ke zkopírování oné reference a obě proměnné zároveň ukazují na stejná data, což může být potenciálně velmi nebezpečné. Uveďme si příklad.

Ukázka A.3

```
Double1P a = new Double1P(7, 2); // vytvoření nové proměnné
Double1P b = a;                  // přiřazení reference
b.W = 3;                         // změna dat obou proměnných a i
b
Console.Write(a.W);              // vypíše 3
```

V knihovně se takový problém dá řešit snadno pomocí vytvoření nové instance objektu z objektu původního.

Ukázka A.4

```
Double1P a = new Double1P(7, 2); // vytvoření nové proměnné
Double1P b = new Double1P(a); // vytvoření nové instance z a
b.W = 3;                         // změna dat pouze proměnné b
Console.Write(a.W);              // vypíše původní hodnotu 2
```

Problém malých čísel

Tato část je citována z [2].

Při výpočtech se můžeme dostat do situace, kdy pracujeme s velmi malými čísly (řádově $<10^{-100}$). Poté se lze setkat s problémem, kdy knihovna zdánlivě nefunguje tak, jak bychom očekávali. Není to však problém knihovny, ale obecně počítačové aritmetiky, resp. omezených možností reprezentace čísel. Příklad z praxe:

Testujeme, zda součin dvou "projektivních skalárů" (tj. $x = [X, w]^T$) je menší než 0:

```
if (a * b < 0) DelejNeco();
```

Pokud bude jedna z hodnot kladná a druhá záporná, očekáváme, že se zavolá metoda DelejNeco(). Pokud se postupně s hodnotami *a* i *b* blížíme k nule, nastane situace, kdy se metoda přestane volat. To je způsobeno prostým numerickým podtečením. Pokud je např. $a = k \cdot 10^{-200}$ a $b = j \cdot 10^{-150}$, je jejich součin $c = j \cdot k \cdot 10^{(-200+150)}$, což je pod rozsahem *double* (cca $\pm 5 \cdot 10^{-324}$ až $\pm 1{,}7 \cdot 10^{308}$). Výsledkem operace je tedy nula, při porovnání s nulou získáme *false* a metoda se nezavolá. V praxi nepomohlo ani zapnutí kontroly přetečení/podtečení, tato situace nebyla detekována. Nezbývá, než při přibližování k nule zvolit hranici, při které výpočet ukončíme s přibližným výsledkem. Volba této hranice závisí na použitých operacích. V tomto případě platí, že při násobení se exponenty sčítají, je tedy vhodné zvolit za mez polovinu dolního rozsahu typu *double*, tedy cca 10^{-160}.

Tento problém nelze z rychlostních důvodů bez hardwarové podpory uspokojivě řešit v rámci knihovny. V kritických případech, kdy z nějakého důvodu nepoužijeme ohraničení přesnosti výpočtu, může být řešením manuální kontrola podtečení exponentu, kdy operaci s čísly provedeme "ručně". Např. při násobení zvlášť vynásobíme mantisy a znaménkové bity a zvlášť sečteme exponenty, přičemž jejich součet maskujeme a porovnáme s nulou.

Přímý přístup k datům

Další věcí, na kterou je třeba dávat pozor, je přímý přístup ke složkám vektorů. Tím je myšlen přístup k prvkům pole, ve kterém jsou složky uloženy. Při pokusu o přístup na neexistující index v poli bude vyhozena standardní výjimka *IndexOutOfRangeException*. Knihovna nemá šanci tuto možnost uživateli zamezit. Nejedná se o tak nebezpečnou chybu,

protože výpočet je přerušen, ale přesto je třeba mít stále na paměti, že prvky v poli jsou indexovány od 0.

Paralelní výpočty

Vzhledem k určení knihovny pro testování algoritmů v projektivním prostoru, nejsou podporovány paralelní výpočty. Tzn, že není možné spustit výpočet běžící na více vláknech, aniž by velice pravděpodobně došlo k chybnému výpočtu. Je to způsobeno použitím sdílené proměnné při procházení složkami vektoru.

Příloha B: Specifikace zadání

Specifikace knihovny byla dodaná vedoucím bakalářské práce.

Datové typy

Vektory – Eukleidovské

DoubleN (**x**)

Vektory – Projektivní

DoubleNP (**x**, w)

Matice – Eukleidovské

DoubleNM (**x**)

Operace

Aritmetické

±	C.x := A * B.w ± B.x C.w := B.w
*	C.x := A * B.x C.w := B.w
/	C.x := A * B.w pouze pro jednorozměrný DoubleNP C.w := B.x

Porovnání pouze pro jednorozměrný DoubleNP

≠	A * B.w ≠ B.x
=	A * B.w = B.x
≤	A * B.w ≤ B.x (analogicky pro ostatní porovnání)

DoublexP C = DoublexP B ***op*** double A

/	C.x := B.x C.w := A * B.w

Operace

DoubleNP C = DoubleNP A ***op*** DoubleNP B

Aritmetické

±	C.x := A.x * B.w ± A.w * B.x C.w := A.w * B.w
*	C.x := A.x * B.x C.w := A.w * B.w
/	C.x := A.x * B.w jen pro Double1P C.w := A.w * B.x

Porovnání pouze s Double1P

≠	A.x * B.w ≠ B.x * A.w
=	A.x * B.w = B.x * A.w
≤	A.x * B.w ≤ B.x * A.w (analogicky pro ostatní porovnání)

Analogicky

DoubleNP C = DoubleNP B ***op*** DoubleNP A

/	C.x := B.x * A.w pouze pro Double1P C.w := A.x * B.w

±	C.x := A*B.w ± B.x C.w := B.w
*	C.x := A*B.x C.w := B.w
/	C.x := A*B.w pouze pro Double1P C.w := B.x

Vektorové operace

Euclidean

±	c := a ± b pro DoubleN
*	c := a * b skalární součin

Matice

±	C := A ± B pro DoubleNM
*	C := A * B pro DoubleNM

Matice-vektor operace

*	C := a * B Double1, a DoubleNM
*	c := A * b DoubleNM, vektor DoubleN

Projektivní

±	c := a ± b for DoubleNP
*	c := a * b skalární součin

Euclidean - projective

*	C := A * b DoubleNM, DoubleNP násobení Eukleidovské matice projektivním vektorem výsledek - vektor DoubleNP

Konverze

```
double[N] := DoubleNP.D
DoubleNP := double[N]
```

Gettery

```
double x = A.D[i];
```

Settery

```
A.D[0] := 376.78;
A.D[1] := 1.22;
A.D[i] := 1.55;
```