České vysoké učení technické v Praze

Fakulta Elektrotechnická
Katedra řídicí techniky

Numerické algoritmy pro polynomiální matice v jazyce C++

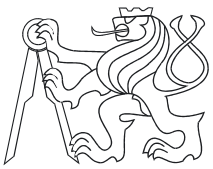
Diplomová práce

Leoš Halmo

2004

České vysoké učení technické v Praze

Fakulta Elektrotechnická
Katedra řídicí techniky



Numerické algoritmy pro polynomiální matice v jazyce C++

Diplomová práce

Autor: Leoš Halmo
halmol@fel.cvut.cz
Vedoucí diplomové práce: Ing. Zdeněk Hurák
z.hurak@c-a-k.cz
Centrum Aplikované Kybernetiky, ČVUT v Praze
Oponent: Ing. Josef Čapek, Ph.D.
capekj@control.felk.cvut.cz
Katedra řídicí techniky, ČVUT v Praze
datum odevzdání: leden, 2004

Anotace

Tato práce se zabývá implementací numerických algoritmů pro polynomiální matice v programovacím jazyce C++. Hlavním výsledkem je objektově orientovaná knihovna PolPack++, která umožňuje provádět základní numerické operace s polynomiálními maticemi, řešit lineární polynomiální rovnice, provádět trojúhelníkování polynomialních matic a počítat jejich determinant či hodnost. Pro návrh byly použity moderní pokročilé programovací techniky, které jsou v práci podrobně popsány.

Annotation

This work deals with implementation of numerical algorithms for polynomial matrices in C++ programming language. The main result of this work is a new object-oriented library PolPack++, which can perform basic numerical operation with polynomial matrices. Functions for solving linear equation with polynomial matrices, triangularization, determinant computation and rank evaluation are included. New advanced programming techniques were used for design this library. Descriptions of these techniques are also included.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne ....

podpis

Poděkování

Na tomto místě bych chtěl především poděkovat Ing. Zdeňku Hurákovi za vedení a podporu při tvorbě této diplomové práce. Dále poděkování patří Joergu Walterovi, vývojáři knihovny uBLAS, za pomoc při řešení problémů s touto knihovnou.

Obsah

Kapitola 1

Úvod

Polynomiální přístup je velmi rychle se rozvíjející oblast teorie řízení. Lineární systémy mohou být popsány polynomiálními maticovými zlomky. Popis vlastností systémů a návrh řízení lze provést algebraickou manipulací s polynomiálními maticemi. Polynomiální matice mohou být použity pro návrh moderních regulátorů (LQG, $\mathcal{H}_2$, $\mathcal{H}_\infty$, ℓ_1), široké uplatnění mají také v oblasti návrhu filtrů a zpracování signálů (Kalmanův filtr, Wienerův filter).

Pro výpočty s polynomiálními maticemi existuje velmi omezené množství softwarových produktů. Nejznámějším a nejúplnější je komerční Polynomial Toolbox pro Matlab od firmy PolyX Ltd [11]. Velmi omezenou sadu funkcí pro polynomiální matice zahrnuje nekomerční produkt Scilab vyvinutý institucí Inria [8]. Některé funkce pro polynomiální matice obsahuje poslední verze komerčního softwaru Maple od Waterloo Maple Inc [9] a program Mathematica od Wolfram Research Inc [10]. V současné době vzniká zároveň nová knihovna pro polynomiální matice v jazyce java – Polynomial Matrices in Java [12] vyvíjená Michalem Paděrou a sada funkcí pro systém Mathematica vyvíjená Jiřím Kujanem.

Cílem této práce bylo navrhnout a implementovat novou volně dostupnou knihovnu pro počítání s polynomiálními maticemi v jazyce C++.

1.1 Výsledky práce

Hlavním výsledkem této práce je objektově orientovaná knihovna pro počítání s polynomiálními maticemi PolPack++ napsaná v jazyce C++. Knihovna je volně dostupná na servru sourceforge.net [13]. Náhled domovské stránky je umístěn v příloze.

Účelem této práce nebyla implementace numerických algoritmů pro konstantní matice. Tyto algoritmy jsou však nezbytně nutné pro implementaci algoritmů pro polynomiální matice. Proto je v práci provedeno srovnání nejznámějších a nejpoužívanějších numerických knihoven pro výpočty lineární algebry.

Protože se práce zabývá objektovým programováním numerických algoritmů v C++, jsou zde ukázány možné problémy při použití tohoto jazyka pro psaní numerických algoritmů, které se podílejí na snížení výpočetní rychlosti a zároveň jsou popsány moderní programovací techniky založené na použití šablon, které řeší uvedené problémy.

Nakonec bylo provedeno srovnání výkonnosti vybraných funkcí knihovny PolPack++ a komerčního Polynomial Toolboxu pro Matlab.

1.2 Struktura dokumentu

Diplomová práce je rozvržena do osmi kapitol:

Kapitola 1 – Úvod (tato kapitola).

Kapitola 2 – Protože se práce zabývá programováním numerických algoritmů v jazyce C++, jsou v této kapitole ukázána možná úskalí při psaní numerických algoritmů objektově orientovaným přístupem.

Kapitola 3 – v této kapitole jsou popsány nejznámější a nejpoužívanější knihovny numerické lineární algebry a je zde zdůvodněn výběr knihoven, které byly použity pro implementaci numerických algoritmů pro polynomiální matice v knihovně PolPack++.

Kapitola 4 – v této kapitole je popsány moderní programovací techniky, které umožňují zvýšení výkonnosti numerického softwaru. Všechny popsané techniky využívají výhod generického programování za pomoci šablon. Tyto techniky jsou použity v knihovně uBLAS, kterou využívá PolPack++ a samozřejmě i v samotném PolPacku.

Kapitola 5 – v této kapitole je podrobně popsána implementace knihovny PolPack++.

Kapitola 6 – tato kapitola se zabývá popisem složitějších numerické algoritmy pro polynomiální matice. Je zde uveden podrobnější popis algoritmu pro výpočet determinantu, hodnosti a kořenů polynomiální matice.

Kapitola 7 – v této kapitole jsou provedeny vybrané numerické experimenty. Jsou zde výsledky srovnání rychlostí knihovny PolPack++ a Polynomial Toolboxu pro Matlab.

Kapitola 8 – závěr.

1.3 Co je to vlastně polynomiální matice?

Polynomiální matice rozměru $m \times n$ a stupně d je polynom

$$P(s) = P_0 + sP_1 + s^2P_2 + \cdots + s^dP_d, \tag{1.1}$$

kde P_i jsou konstantní matice rozměru $m \times n$. To znamená, že je to matice jejíž prvky jsou polynomy. Nechť $p_{ij}(s)$, kde $i = 1, \ldots, m$ a $j = 1, \ldots, n$ je element polynomiální matice. Označme $\deg p_{ij}(s)$ jako stupeň polynomu $p_{ij}(s)$, potom čísla

$$\rho_i = \max_j \deg p_{ij}(s), i = 1, \ldots, m \tag{1.2}$$

$$\gamma_j = \max_i \deg p_{ij}(s), j = 1, \ldots, n \tag{1.3}$$

jsou řádkový a sloupcový stupeň polynomiální matice $P(s)$.

Proměnná s charakterizuje typ polynomiální matice. V teorii řízení se používají čtyři parametry – s a p pro spojitou polynomiální matici a proměnné z a d pro diskrétní matici. Diskrétní polynomiální matice může obsahovat i polynomy se záporným mocninou proměnné z a d. Tyto matice nazýváme oboustranné polynomiální matice.

Předpokládejme tedy, že

$$\rho_i, i = 1, \ldots, m \tag{1.4}$$

$$\gamma_j, j = 1, \ldots, n \tag{1.5}$$

jsou řádkové a sloupcové stupně matice $P(s)$. Matice hlavních sloupcových koeficientů polynomiální matice M_c (column leading coefficient matrix) je konstantní matice, jejíž prvky na pozici (i, j) jsou koeficienty polynomů u γ_j-té mocniny na pozici (i, j) polynomiální matice $P(s)$.

Matice hlavních řádkových koeficientů polynomiální matice M_r (row leading coefficient matrix) je konstantní matice, jejíž prvky na pozici (i, j) jsou koeficienty polynomů u ρ_i-té mocniny na pozici (i, j) polynomiální matice $P(s)$.

O polynomiální matici $P(s)$ řekneme, že je sloupcově redukovaná, pokud matice M_c má plnou sloupcovou hodnost.

O polynomiální matici $P(s)$ řekneme, že je řádkově redukovaná, pokud matice M_r má plnou řádkovou hodnost.

Mějme spojitou polynomiální matici $P(s)$ a diskrétní matici $Q(z)$. Potom matice $P^*(s)$ a $Q^*(z)$

$$P^*(s) = P^H(-s) \tag{1.6}$$

$$Q^*(z) = Q^H(z^{-1}) \tag{1.7}$$

jsou konjugovaně transponované matice k maticím $P(s)$ a $Q(z)$.

Příklad oboustranné polynomiální matice

$$P(z) = z^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 5 \end{pmatrix} + \begin{pmatrix} 2 & 0 \\ 3 & 1 \end{pmatrix} + z \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + z^2 \begin{pmatrix} 0 & 0 \\ 3 & 0 \end{pmatrix} \tag{1.8}$$

Matici můžeme zapsat ve tvaru matice polynomů:

$$P(z) = \begin{bmatrix} z^{-1} + 2 & z \\ 3 + 3z^2 & 5z^{-1} + 1 \end{bmatrix} \tag{1.9}$$

Matice má vedoucí (leading) stupeň 2, koncový (trailing) stupeň 1.

Kapitola 2

Objektově orientované programování numerických algoritmů

V této kapitole jsou ukázány výhody a nevýhody použití objektově orientovaného programování v C++ pro návrh numerických knihoven. C++ umožňuje poměrně jednoduše navrhnout nové datové typy (např. matice, vektory) s přetíženými operátory (pro sčítání, násobení, ...), které umožňují použít standardní matematickou notaci při zápisu programu. Z hlediska čitelnosti programu a uživatelského komfortu se proto použití C++ zdá být skvělým řešením. C++ se však pro psaní numerických knihoven příliš neprosadilo. Při srovnání výkonnosti s obdobnými knihovnami psanými v jiných programovacích jazycích (především Fortran a C) se ukazuje, že výkonnost těchto knihoven je velmi slabá. Proto v minulosti programátorská komunita mnohdy používala kombinované řešení. Z důvodu zaručení přijatelné výkonnosti byly kritické části kódu psány v jazyce Fortran.

Slabá výkonnost knihoven psaných v C++ je zpravidla způsobena těmito faktory:

- Použití virtuálních funkcí (dynamický polymorfismus)
- Využívání odkládacích objektů (temporaries)

Volání virtuálních funkcí nelze optimalizovat překladačem, protože volaná funkce není známa v době překladu. Další nevýhodou podepisující se na snížení rychlosti výpočtu je potřeba vícenásobného přístupu do paměti oproti volání klasických funkcí. Z těchto důvodů by se virtuální funkce měly používat pouze v případě, že funkce je výpočetně dlouhá nebo není volána „příliš často“.

Časté a zcela nevhodné použití dynamického polymorfismu demonstruje následující příklad:

```
class Matrix {
      public: virtual double operator(int i, int j) = 0;
}

class SymetricMatrix : public matrix {
```

```
      public: virtual double operator(int i, int j);
}

class UpperTriangualMatrix : public matrix {
      public: virtual double operator(int i, int j);
}
```

Zde přetížený operátor pro přístup k jednotlivým prvkům matice způsobí značné snížení rychlosti výpočtu jakéhokoliv algoritmu. Dobře napsané numerické knihovny se tomuto problému vyhýbají za pomoci techniky zvané Barton-Nackman trik [3, 4], který je vysvětlen v sekci 4.2, nebo méně častěji pomocí tzv. enginu [3].

Dalším uvedeným problémem je vznik odkládacích objektů (temporaries). Problém nastává při výpočtu výrazu s přetíženými operátory:

```
vector<double> a(n), b(n), c(n), d(n);
a = b + c - d;
```

V tomto případě překladač generuje kód podobný tomuto:

```
vector* _t1 = new vector(n);
    for(int i=0; i < n; i++)
        _t1(i) = b(i) + c(i);

vector* _t2 = new vector(n);
    for(int i=0; i < n; i++)
        _t2(i) = _t1(i) + b(i);

for(int i=0; i < n; i++)
        a(i) = _t2(i) + _t1(i) ;

delete _t2; delete _t1;
```

To znamená, že jsou potřeba vytvořit dva odkládací objekty a výpočet proběhne ve třech iteračních cyklech. Pro malá pole se snížení rychlosti projeví především kvůli použití operátorů `new` a `delete`, které jsou časově velmi náročné. V [3] je uvedeno, že snížení výkonu je až na 1/10 oproti obdobnému problému napsaném v jazyce C. Pro středně velké pole (in cache array) se projeví použití nadbytečných iteračních cyklů a přístupů do paměti. V těchto případech je rychlost výpočtu na 30-50% oproti úloze psané v jazyce C. Z důvodu použití odkládacích objektů je také využíváno více paměti, takže při zvětšení objektů dochází dříve k zaplnění paměti cache, což vede k dalšímu snížení rychlosti výpočtu.

Vyřešení tohoto problému je popsáno v odstavci 4.3 za pomoci techniky expression templates [1, 3, 4], která umožní vyhnout se použití odkládacích objektům a dále provede výpočet libovolného výrazu v jednom iteračním cyklu. Při použití vhodného překladače můžeme díky této technice dosáhnout rychlosti výpočtu srovnatelné a v některých případech i větší ve srovnání s jazykem C nebo Fortran.

Napsat numerickou knihovnu, která bude dostatečně rychlá (alespoň ve srovnání s obdobnou v C nebo Fortranu) a využívající výhody objektově orientovaného programování je ale vzhledem ke složitosti uvedených problémů poměrně obtížné. Je třeba využít moderních programovacích technik, které jsou velmi složité. Zejména při použití expression templates techniky je kód velmi složitý a čitelný pouze pro autora.

Kapitola 3

Knihovny pro numerické výpočty

V mnoha algoritmech pro polynomiální matice je třeba provádět výpočty lineární algebry, jako je řešení soustav lineárních rovnic, výpočty vlastních a singulárních čísel a maticových rozkladů. Některé algoritmy pro polynomiální matice používají diskrétní Furierovu transformaci (DFT). Účelem této práce však není implementace těchto problémů, pro které existuje mnoho hotových a spolehlivých knihoven. Proto bylo potřeba vybrat vhodné knihovny řešící tyto úlohy. V této kapitole jsou popsány vybrané knihovny z oblasti lineární algebry a je zdůvodněn výběr knihoven, které jsou použity v knihovně PolPack++.

3.1 Přehled knihoven pro numerické výpočty lineární algebry

Knihoven pro numerické výpočty lineární algebry existuje celá řada. V tomto odstavci jsou popsány základní rysy nejznámějších a nejpoužívanějších.

3.1.1 BLAS

BLAS (Basic Linear Algebra Subprograms) je knihovna napsaná v jazyce Fortran77. Skládá se ze tří úrovní - Level 1 BLAS pro vektor-vektor operace (např. skalární součin), Level 2 BLAS pro vektor-matice operace a Level 3 BLAS matice-matice operace. Obsahuje pouze základní funkce pro práci s obecnými typy reálných i komplexních (v jednoduché a dvojité přesnosti) vektorů a matic. Nemá implementovány algoritmy vyšší lineární algebry jako je řešení lineárních rovnic a maticové rozklady. Pro výpočty s řídkými maticemi je k dispozici rozšíření knihovny – SparseBLAS. Existuje mnoho optimalizovaných verzí pro různé platformy, které dodávají výrobci hardwaru ke svým produktům. K dispozici je také verze v jazyce C, která byla vygenerována automatickým převodem z Fortranu (CBLAS) a verze v jazyce Java (Java BLAS). Knihovna BLAS je natolik rozřená, že se její interface stal standardem. Knihovna BLAS je dostupná na `http://www.netlib.org/blas/`.

3.1.2 LAPACK

LAPACK (Linear Algebra PACKage) je velice rozsáhlá knihovna napsaná v jazyce Fortran77. Poskytuje funkce pro řešení soustav lineárních rovnic, problému nejmenších čtverců, maticové rozklady (QR, SVD, LU, Choleskyho rozklad, Schurův rozklad, zobecněný Schurův rozklad). Pro svůj běh potřebuje knihovnu BLAS. LAPACK společně s knihovnou BLAS jsou pravděpodobně nejpoužívanější knihovny pro vědecké výpočty. Využívá ji mnoho komerčních i nekomerčních programových produktů (např. Matlab, Octave, Scilab). Stejně jako v případě BLASu, existuje automaticky generovaná verze v jazyce C (CLAPACK) a v jazyce Java. Knihovna LAPACK je dostupná na adrese `http://www.netlib.org/lapack/`.

3.1.3 ATLAS

Atlas (Automatically Tuned Linear Algebra Software) je knihovna obsahující C a Fortran77 interface do BLASu a podmnožiny LAPACKu (bohužel pouze funkce pro řešení soustav lineárních rovnic) a zároveň název výzkumného projektu zabývající se automatickou optimalizací softwaru. Knihovna je optimalizována pro mnoho platforem. Při překladu knihovny je zjišťován typ procesoru a instrukční sada a automaticky se vybírají rutiny optimalizované pro vybraný procesor. Knihovna ATLAS je dostupná na adrese `http://math-atlas.sourceforge.net/`.

3.1.4 Intel MKL

Intel MKL (Math Kernel Library)je jediná komerční knihovna zmíněná v této kapitole. Je to v opět verze Cblasu, Blasu a Lapacku optimalizovaná pro procesory firmy Intel. Dále obsahuje rutiny pro výpočet jedno a dvojrozměrné rychlé Furierovy transformace (FFT) Více informací o této knihovně lze nalézt na adrese `http://www.intel.com/software/products/mkl/`.

3.1.5 uBLAS

uBLAS je stále se rozvíjející knihovna psaná v jazyce C++. Obsahuje šablony tříd mnoha typů matic a vektorů. Velkou výhodou této knihovny je rychlost jejího vývoje a rychlost, s jakou jsou opravovány nalezené chyby. Naopak její slabinou je nedostatečná dokumentace. Paralelně s touto knihovnou vzniká balík , který provádí napojení této knihovny na Atlas a LAPACK (bindigs library). Více informací o této knihovně je uvedeno v odstavci 3.3.1. Knihovna je volně dostupná na adrese `http://www.boost.org`.

3.1.6 BLITZ++

Blitz++ je knihovna napsaná v jazyce C++ zaměřená na práci s poli. Obsahuje šablony třídy pro řádkově (C–style array) nebo sloupcově (Fortran–style array) orientované pole

dimenze 1-12 s volitelným rozsahem indexů. Pro výpočet výrazů používá techniku expression templates, díky které dosahuje vysoké výpočetní rychlosti (přibližně na úrovni jazyku Fortran). Knihovna nemá implementovány algoritmy lineární algebry. Knihovna BLITZ++ je dostupná na adrese `http://www.oonumerics.org/blitz/`.

3.1.7 LAPACK++

LAPACK++ je objektová knihovna napsaná v C++. Pro výpočty úloh lineární algebry ale používá napojení na LAPACK nebo CLAPACK. Obsahuje proto třídy matic a vektorů svojí strukturou odpovídající typům, které se používají v LAPACKu. Vývoj této knihovny již byl ukončen a její tvůrce tvrdí, že všechny její vlastnosti by měla využít nová knihovna – TNT. Knihovna LAPACK++ je dostupná na adrese `http://math.nist.gov/lapack++/`.

3.1.8 TNT

TNT (Template Numerical Toolkit) obsahuje pouze šablony pro jedno, dvou a třírozměrné slopcově a řádkově orientované pole na kterých lze provádět pouze základní numerické operace (sčítání, odčítání a násobení). Nevyužívá moderní techniky jako je expression templates proto její výpočetní rychlost je velmi slabá. Tato knihovna by měla podle autora nahradit knihovny LAPACK++, SparseLib (knihovna počítající s řídkými maticemi) a knihovnu IML++ (knihovna pro řešení soustav lineárních rovnic využívající iterační algoritmy). Bohužel v současné době obsahuje velmi málo ze zmíněných knihoven a její vývoj je velmi pomalý (pokud se úplně nezastavil). Knihovna TNT je dostupná na adrese `http://math.nist.gov/tnt/`.

3.1.9 MTL

MTL (Matrix Template Library) je knihovna napsaná v jazyce C++. Obsahuje šablony pro různé typy matic (sloupcově a řádkově orientované plné, trojúhelníkové, diagonální, symetrické a řídké) využívající pro uložení dat různé paměťové modely. Knihovna nepoužívá přetížené operátory. Obsahuje funkce přibližně na úrovni knihovny BLAS. Pro složitější výpočty lineární algebry lze využít napojení na LAPACK. Knihovna MTL je dostupná na adrese `http://www.osl.iu.edu/research/mtl/`.

3.2 Srovnání knihoven pro numerické výpočty lineární algebry

V tomto odstavci jsou srovnány základní vlastnosti knihoven popsaných v předcházejícím odstavci formou přehledových tabulek. V tabulce 3.2 je uveden programovací jazyk, licence a číslo verze s datem vydání nebo datem poslední aktualizace. Většina z nich je šířena pod GPL licencí (GNU General Public License) [15]. Tabulka 3.2 ukazuje typy matic, s kterými

Knihovna	Jazyk	Licence	Verze/aktualizace
BLAS	Fortran77	free	?
LAPACK	Fortran77	free	3.0/květen 2003
Atlas	C/Fortran77	GPL	3.6.0/prosinec 2003
Intel MKL	C/Fortran77	komerční	6.1/?
uBLAS	C++	GPL	ublas_2003_12_21/listopad 2003
Blitz++	C++	GPL	0.6/říjen 2002
LAPACK++	C++	GPL	1.1a/únor 2000
TNT	C++	GPL	1.2/červen 2002
MTL	C++	GPL	2.1.2/září 2002

Tabulka 3.1: Základní údaje

knihovna	obecná	trojúhelníková	symetrická	diagonílní	třídiag.	řídká
BLAS	√					
LAPACK	√	√	√		√	
Atlas	√		√			
Intel MKL	√	√	√	√	√	√
uBLAS	√	√	√	√	√	√
Blitz++	√					
LAPACK++	√	√	√		√	
TNT	√					
MTL	√	√	√	√	√	√

Tabulka 3.2: Typy matic

knihovny pracují a v tabulce 3.2 jsou ukázány, které algoritmy lineární algebry knihovny zahrnují.

3.3 Výběr knihoven pro implementaci polynomiálních algoritmů

Jeden z možných způsobů uložení polynomiálních matic je za pomoci pole matic, které reprezentují maticové koeficienty u jednotlivých mocnin polynomiální matice. Tento způsob uložení je právě použit v knihovně PolPack++. Proto při návrhu knihovny bylo třeba nejprve vybrat nejlépe objektovou knihovnu, která obsahuje vhodné typy matic pro reprezentaci maticových koeficientů. Při výběru byl velký důraz kladen na to, zda se knihovna stále rozvíjí, jaký je na ní ohlas a zda je poskytována technická podpora. Samozřejmě další velmi důležité hledisko výběru byla výpočetní rychlost. Z knihoven uvedených v předcházejících odstavcích této kapitoly těmto podmínkám nejlépe vyhovují knihovny uBLAS a Blitz++. Blitz++ je však knihovna, která je zaměřena na práci s vícerozměrnými poli a tenzory. Umožňuje použít jeden typ matice (obecná matice)

Knihovna	LU	QR	SVD	Cholesky	QZ	Schur
BLAS						
LAPACK	√	√	√	√	√	√
Atlas	√			√		
Intel MKL	√	√	√	√	√	√
uBLAS						
Blitz++						
LAPACK++	√	√	√	√	√	√
TNT						
MTL						

Tabulka 3.3: Algoritmy

reprezentovanou polem. Naproti tomu uBLAS je zaměřen právě na výpočty lineární algebry, má implementováno mnoho různých typů matic a vektorů. Ani jedna z těchto knihoven však není schopna řešit soustavy lineárních rovnic, provádět výpočty vlastních čísel a maticové rozklady. Protože uBLAS umožňuje použít formát uložení dat matice, které je kompatibilní s formátem použitým v knihovně LAPACK, bylo rozhodnuto použít hybridní řešení – pro uložení maticových koeficientů a základní výpočty jako je sčítání, odčítání a násobení knihovna uBLAS a pro výpočty lineární algebry knihovna LAPACK, která v podstatě nemá ve své oblasti rovnocennou konkurenci. Podrobný popis je uveden v [14]. Knihovna uBLAS je podrobněji popsána v následující sekci.

Na začátku této kapitoly bylo uvedeno, že některé algoritmy pro polynomiální matice používají diskrétní Furierovu transformaci (DFT). Knihoven pro výpočet DFT opět existuje velké množství. Z nich byla vybrána knihovna FFTW hlavně z důvodu vysoké výpočetní rychlosti této knihovny. Její hlavní vlastnosti jsou popsány v odstavci 3.3.2.

3.3.1 Knihovna uBLAS

uBLAS je knihovna šablonových tříd psaná v jazyce C++. Je součástí velice rozsáhlého souboru knihoven – Boost (`http://www.boost.org`). Svojí funkčností odpovídá přibližně knihovně BLAS. Používá přetížené operátory a pro vyhodnocování výrazů využívá výhod techniky expression templates (podrobnosti o této technice jsou uvedeny v odstavci 4.3). Pro přístup k jednotlivým prvkům matic lze přistupovat za pomoci přetížených indexních operátorů nebo pomocí iterátorů, které jsou kompatibilní s iterátory z STL knihovny. Díky tomu se dá na objekty knihovny aplikovat velké množství standardních algoritmů (např. třídění).

uBLAS obsahuje šablony tříd pro následující typy vektorů:

- obecný vektor (general vector)
- jednotkový vektor (unit vector)

- nulový vektor (general vector)
- řídký vektor (sparse vektor)
- komprimovaný vektor (compressed vector)

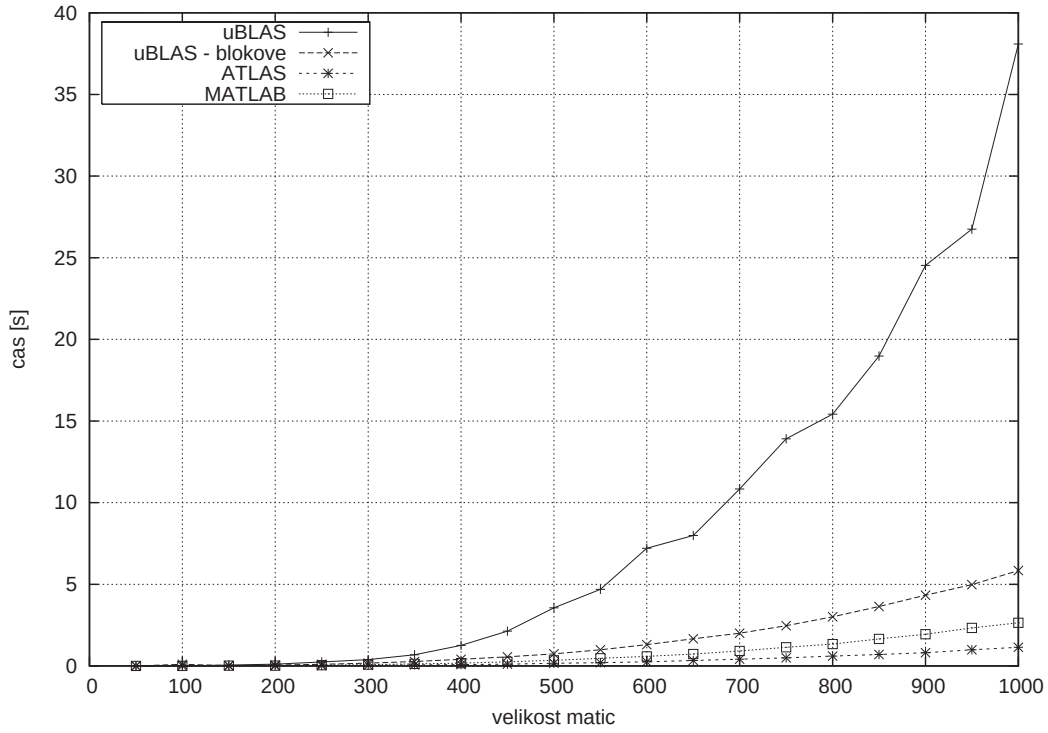
a matic:

- obecná matice (general matrix)
- jednotková matice (identity matrix)
- nulová matice (zero matrix)
- trojúhelníková matice (triangular matrix)
- symetrická matice (symmetric matrix)
- hermitovsky symetrická matice (hermitian matrix)
- pásová matice (banded matrix)
- řídká matice (sparse matrix)
- komprimovaná matice (compressed matrix)

S výjimkou jednotkové a nulové mohou všechny matice být uloženy v řádkové (row major) nebo sloupcové formě (column major). Forma matice je specifikována šablonovým typem. Dalším šablonovým typem společným pro všechny matice je typ kontejneru, ve kterém jsou uložena vnitřní data. Data matice jsou mapována do jednorozměrného pole. uBLAS má implementovány dva kontejnery pro uložení dat. Jeden obsahuje dynamicky alokované pole (unbounded array) a druhý statické pole (bounded array), který má svoji velikost jako šablonový typ. Použít se dají i jiné kompatibilní kontejnery, např. `vector<T>` ze standardní knihovny.

Dále obsahuje objekty pro tvorbu řezů matic (range, sliced sub matrix) a vektorů a objekty pro indexování řádků a slopců matic.

Rychlost výpočtu je na vysoké úrovni díky použité technice expression templates. Grafy na obrázcích 3.1 a 3.2 ukazují srovnání rychlosti násobení dvou čtvercových matic v uBLAS a Atlasu (v grafu 3.2 je ještě ukázána rychlost násobení v Matlabu verze 6.5, který používá vlastí verzi Atlasu). Pro násobení v uBLAS jsou implementovány dva algoritmy. Mimo klasického násobení lze použít blokový algoritmus, který lépe využívá paměť cache a tím dosahuje vyšší rychlosti pro násobení větších matice. Klasické násobení je pro velmi velké matice prakticky nepoužitelné. Pro matice o velikosti 2000x2000 doba výpočtu na testovacím počítači přesáhla 17 minut. Z druhého grafu je zřejmé,že pro malé matice (přibližně do velikosti 120x120) je blokový algoritmus ve srovnání s klasickým pomalejší. Měření rychlosti proběhlo na počítači vybaveném procesorem AMD Duron s taktovací frekvencí 1,2GHz, L1 a L2 cache o velikosti 64kB.



Obrázek 3.1: Srovnání rychlosti násobení v Ublasu a Atlasu

Jak již bylo uvedeno, tato knihovna obsahuje funkce, které přibližně odpovídají knihovně BLAS. To znamená, že nemá implementovány nezbytné algoritmy pro řešení soustav lineárních rovnic, maticové rozklady a výpočty vlastních čísel. Pro řešení těchto problémů vznikla knihovna provádějící napojení uBLASu na LAPACK a Atlas (bindings library). Bohužel v současnosti není tato knihovna součástí knihovny Boost.

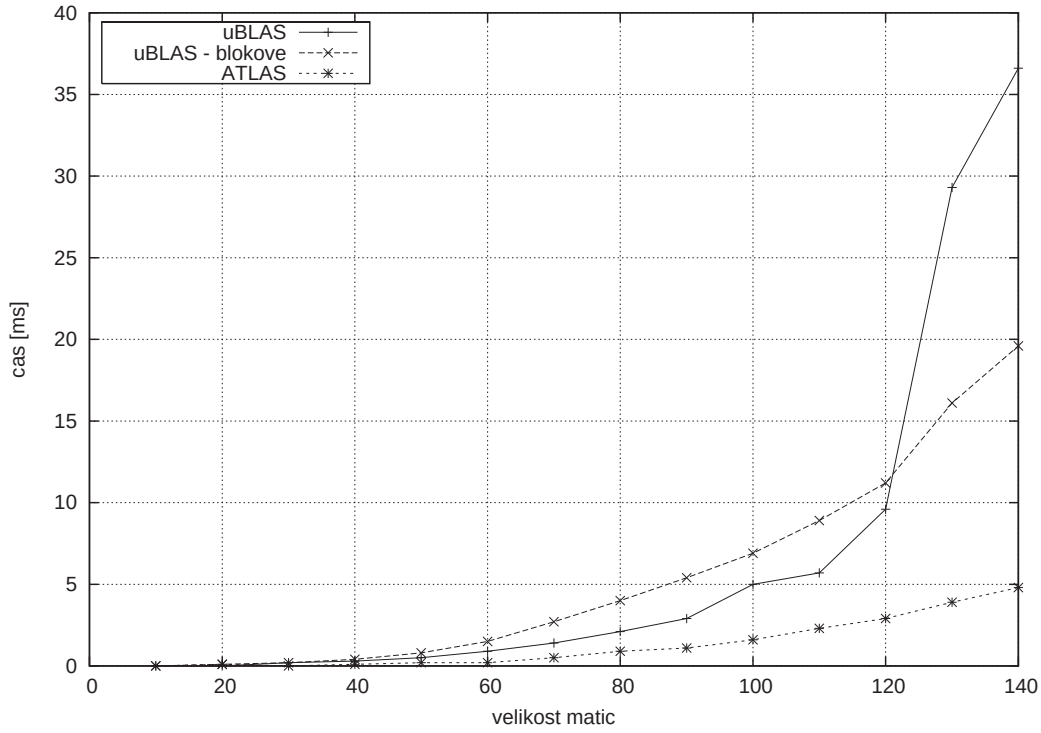
3.3.2 Knihovna FFTW

FFTW (the Fastest Furier Transform in the West) je soubor velmi rychlých rutin napsaných v jazyce C pro výpočet diskrétní Furierovy transformace (DFT) a jejích speciálních případů.

FFTW je schopna provádět výpočet DFT pro reálná a komplexní data v libovolné dimenzi a libovolné délky. Pro libovolnou délku dat (včetně délky rovnající se prvočíslu) je použit algoritmus, jehož asymptotická složitost je $n \log n$. FFTW počítá v základní verzi nejrychleji transformaci, jejichž délku lze rozložit na součin prvočísel 2, 3, 5 a 7. Pokud je potřeba efektivnější výpočet i pro jiný rozklad prvočísel, je možné použít generátor kódu, který je schopen vygenerovat velice efektivní rutinu pro výpočet transformace libovolné délky.

FFTW podporuje nové skupiny instrukcí SIMD (Single Instruction Multiple Data) – SSE1, SSE2 a 3DNow!

Další zajímavou vlastností je, že umožňuje provést najednou výpočet mnoha trans-



Obrázek 3.2: Srovnání rychlosti násobení v Ublasu a Atlasu

formací pro data, která jsou uspořádána v jednorozměrném poli, přičemž data pro jednu transformaci nemusí být souvislá.

FFTW používá pro výpočet mnoho různých rutin, proto je výpočet rozdělen do dvou fází. V první fázi se hledá vhodný algoritmus pro daná data – vytváří se tzv. plán. Plán může být vytvořen dvěma způsoby – buď je použita nějaká heuristika a vhodný plán je odhadnut (což je rychlé, ale nemusí se zvolit nejoptimálnější algoritmus výpočtu transformace), nebo je provedena hlubší časově náročnější analýza dat, která zvolí pro výpočet optimální rutinu. Druhá možnost je vhodná pro případy, kdy se provádí výpočet stále stejně dlouhé transformace, protože plán může být vytvořen jednou a uchován pro další použití. Ve druhé fázi teprve probíhá samotný výpočet transformace s využitím vytvořeného plánu.

Kapitola 4

Programovací techniky použité v knihovně uBLAS

V této kapitole je popsáno několik moderních programovacích technik které jsou použity v knihovně uBLAS a které budou využity v knihovně PolPack++. Všechny byly vyvinuty v polovině 90. let minulého století a všechny využívají výhod generického programování pomocí šablon (templates) pro zvýšení výkonnosti.

4.1 Traits

Technika traits byla poprvé publikována v roce 1995 Nathanem Myersem v časopise C++ Report [2]. Traits jsou šablony tříd, které umožňují mapovat „jednu věc na jinou“. Mapovat lze z

- typu
- hodnoty proměnné známé v době překladu (např. konstanta)
- kombinace předcházejících

na libovolný jiný typ, konstantu nebo funkci. Používá se např. pro určení výsledného typu při numerických operacích s objekty různých šablonových typů (viz. příklad níže), pro určení hodnoty nějaké konstanty v závislosti na vstupním typu šablony (viz odstavec 5.2), atd.

Příklad - mapování typu

Předpokládejme následující operátor pro sčítání vektorů:

```
template<class T1, class T2> vector<???> operator+ (const
vector<T1>&, const vector<T2>&);
```

V tomto příkladě nastává problém s určením návratového typu, který by měl být např:

```
vector<int> + vector<char> = vector<int>
vector<double> + vector<int> = vector<double>
vector<complex<double> > + vector<double> = vector<complex<double> >
//atd.
```

To znamená, že potřebujeme mapovat z nějaké kombinace dvou typů na jeden typ. To nám provede právě následující třída:

```
template<class T1, class T2> struct promote_trait { };

#define declare_promote(T1, T2, T3) \ template<> \ struct
promote_trait<T1, T2> { \
   typedef T3 T_promote; \
};

declare_promote(int, char, int); declare_promote(double, int,
double); declare_promote(complex<double>, double,
complex<double>); ...
```

Návratový typ nám poskytne třída `promote_trait`. Nyní můžeme operátor pro sčítání upravit do této podoby:

```
template<class T1, class T2> vector<typename
promote_trait<T1,T2>::T_promote> operator+ (const vector<T1>&,
const vector<T2>&);
```

Z příkladu je vidět, že pro překlad je nutné použít překladač umožňující použití parciální specializace šablon. To není, zejména u starších překladačů, zcela samozřejmé. Příkladem je např. komerční překladač VC6.0 od firmy Microsoft.

4.2 Barton-Nackman trik

Tato technika se někdy také nazývá „curiously defined recursive templates“. Jak již bylo uvedeno v kapitole 2, umožňuje vyhnout se použití virtuálních funkcí (dynamického polymorfismu).

Definujme bázovou třídu následujícím způsobem:

```
template class<T_leaf> class Matrix{ public:
  T_leaf& assign_leaf(){
     return static_cast<T_leaf>(*this);
  }

  double operator () (int i, int j){
    return assign_leaf()(i,j);
}
```

a odvozené třídy takto:

```
class SymmetricMatrix : public Matrix<SymmetricMatrix> { ... }

class UpperTriangularMatrix : public Matrix<UpperTriangularMatrix>
{ ... }
```

Trik je v tom, že bázová třída má šablonový typ, který je typem třídy od ní odvozené. To zajistí, že kompletní informace o objektu jsou známy v době překladu. Metody mohou být specializovány v odvozených třídách (standardně jsou definovány v bázové třídě, volitelně mohou být předefinovány v třídě odvozené).

4.3 Expression templates

V kapitole 2 je ukázáno, že při klasickém použití přetížených operátorů pro výpočet hodnot výrazů je pro každý operátor generován jeden odkládací objekt, který představuje výsledek vypočteného podvýrazu. Technika expression templates představuje zcela rozdílný přístup výpočtu výrazů. Vyhýbá se použití odkládacích objektů a celý výraz spočítá v jednom iteračním cyklu.

K porozumění této technice je třeba nejprve porozumět rekurzivním šablonám. Mějme následující šablonu třídy s dvěma šablonovými parametry:

```
template<class Left, class Right> class X { };
```

Třída může mít sebe sama jako šablonový typ. To umožní vytvořit struktury, jako je strom typů:

```
X<X<A,B>,X<C,D> > x;
```

Výraz `A+B+C` může být reprezentován typem

```
X<Vector, plus, X<Vector, plus, Vector> > x;
```

K vytvoření takového stromu reprezentující výraz se použije právě přetížený operátor. V tomto případě tedy operátor nevrací vyčíslený podvýraz (při klasickém použití reprezentovaný odkládacím objektem), ale objekt, který pouze reprezentuje podstrom syntaktického stromu výrazu. Tento objekt je schopen vyčíslit hodnotu podvýrazu pro prvky matic (vektorů nebo jiných kontejnerů) určených indexem. Jinými slovy, přetížený operátor nevrací celou vypočtenou matici (nebo vektor apod.), ale objekt, který je schopen vypočítat hodnotu jednoho prvku matice dané indexem. Samotný výpočet celého výrazu se provede až v přiřazovacím operátoru, který provede iteraci přes všechny indexy matice.

Pro reprezentaci výrazu zavedeme třídu `Expr<T,E>`, která reprezentuje celý výraz. Přesněji, tato třída zapouzdřuje výraz, který je reprezentován šablonovým parametrem `E`.

```
template<class T, class E> class Expr {
    E e;
  public:
    Expr( const E& e ) : e(e) {}
    T operator[] ( int n ) const { return e[n]; }
};
```

Nyní nadefinujeme šablonu třídy `Array<T,E>`, která bude zapouzdřovat samotná data. V této třídě přetížíme přiřazovací operátor (`operator=`), který provede výpočet celého výrazu.

```
template<class T>
class Array {
  public:
    template<class E>
    Array& operator=( const Expr<T,E>& x )
    {
        for( int i=0; i < size(); i++ )
          data[i] = x[i];
        return *this;
    }
// other public members, data, etc.
  private:
    T* data;
};
```

Následující třída `ConstRef<class T, class Container>` obsahuje konstantní referenci na kontejner. V našem případě se bude jednat o referenci na třídu `Array<T>`. Důvod jejího použití je zřejmý z globálního přetíženého operátoru. Jejím použitím zabráníme vytváření kopií objektu typu `Array<T>`.

```
template<class T, class Container> class ConstRef {
    const Container& c;
  public:
    ConstRef( const Container& c_) : c(c_) {}
    T operator[] ( int n ) const { return c[n]; }
};
```

Třída `BinaryOperator<T, A, B, Op>` představuje jeden uzel syntaktického stromu výrazu. Parametry A, B představují větve stromu výrazu a jsou to buď podvýrazy (reprezentované třídou `Expr<T, E>`) nebo reference na kontejner (třída `Array<T>`), typ `Op` reprezentuje operátor aplikovaný na typy `A` a `B`.

```
template<class T, class A, class B, class Op> class BinaryOperator
{
```

```
    A a;
    B b;
  public:
    BinaryOperator( const A& a_, const B& b_ )
        : a(a_), b(b_) {}

    T operator[] ( int n ) const {
        return Op::apply( a[n], b[n] );
    }
};

 template<class T> class OpAdd {
  public:
    static inline T apply( const T& a, const T& b )
    {
        return a + b;
    }
};
```

Přetížený operátor má jako vstupní parametry třídu `Array` a vrací objekt `Expr<T, E>`, kde šablonovým typem `E` je objekt typu `BinaryOperator<T, E>`.

```
template<class T> Expr< T, BinaryOperator< T, ConstRef< T,
Array<T> >,
     ConstRef< T, Array<T> >, OpAdd<T> > >
operator+( const Array<T>& a, const Array<T>& b ) {

    typedef BinaryOperator< T, ConstRef< T, Array<T> >,
                            ConstRef< T, Array<T> >,
                            OpAdd<T> > ExprT;
    return Expr< T, ExprT > (
        ExprT ( ConstRef<T, Array<T> >(a),
                        ConstRef<T, Array<T> >(b) ) );
}
```

Tento operátor definuje sčítací operaci pouze pro dva objekty typu `Array<T>` (tzn. výraz `A+B`). Pro výpočet složitějšího výrazu (např. `A+B+C`) však musíme definovat další globální přetížený operátor pro součet objektu typu `Array` a výrazu reprezentovaného třídou `Expr`:

```
template<class T, class E> Expr< T, BinaryOperator< T, ConstRef<
T, Array<T> >,
  Expr< T, E>, OpAdd<T> > >
operator+( const Array<T>& a, const Expr<T,E>& b )
```

Zde je právě vidět rekurentnost šablony třídy `Expr<T, E>`. Šablonový typ `E` je typu `BinaryOperator<T,A,B,Op>`, jehož typ B je opět třída `Expr`.

Pro součet libovolně dlouhého výrazu musí být definovány čtyři binární sčítací operátory – `Array + Array`, `Array + Expr`, `Expr + Array` a `Expr + Expr`. To znamená, že pro každou binární operaci musíme nadefinovat čtyři globální přetížené operátory, pro každou unární operaci jsou potřeba dva operátory (unární operátor aplikovaný na `Array` a na `Expr`).

Zde jsou vidět nevýhody této techniky. Značným způsobem narůstá délka zdrojového kódu, který je navíc naprosto nečitelný. Velké množství šablon tříd a šablon globálních funkcí způsobí prodloužení doby překladu (v důsledku generování tříd ze šablon).

Mějme následující příklad:

```
Array<T> A, B, C;
...
A = B + C;
```

Co se vlastně stane, přeložíme-li tento kód? Voláním sčítacího operátoru je vytvořen objekt typu `Expr<T,E>`, kde `E` je:

```
BinaryOperator< T, ConstRef< T, Array<T> >, ConstRef< T, Array<T>
>, OpAdd<T> >
```

Vytvořený objekt typu `Expr` má soukromou objektovou proměnnou `e` právě typu `E`. Voláním operátoru přiřazení (z objektu `A`) je zavolán indexní operátor třídy `Expr` (viz definice třídy `Array` výše). Zavoláním indexního operátoru pro třídu typu `Expr` dojde k volání indexního operátoru soukromého objektu `e`, který je typu `BinaryOperator` (tedy `E`). Objekt `e` má v sobě uloženy odkazy na objekty `A`, `B` typu `Array<T>` prostřednictvím objektu typu `ConstRef`. Dojde-li tedy k volání indexního operátoru objektu `e`, dojde k přečtení hodnot z objektu `A` a `B` u příslušného indexu (viz definice třídy `BinaryOperator`) a voláním statické funkce `apply` třídy `OpAdd` je proveden součet těchto hodnot, který je vrácen indexním operátorem. To tedy znamená, že iterací přes všechny indexy je proveden součet vektorů `B` a `C` a výsledek přiřazen do objektu `A` (bez použití odkládacího objektu). Je tedy vygenerován kód podobný tomuto:

```
for(int i = 0; i < size; i++)
    A[i] = B[i] + C[i];
```

Kapitola 5

Popis implementace knihovny PolPack++

Knihovna PolPack++ je šablonová knihovna obsahující třídy a funkce pro počítání s polynomiálními maticemi. Pro svůj běh potřebuje tři knihovny:

- uBLAS - knihovna použitá pro uložení maticových koeficientů polynomiálních matice
- LAPACK - knihovna pro numerické výpočty lineární algebry
- FFTW - Knihovna pro výpočet diskrétní Furierovy transformace.

Základní třídou knihovny je šablona třídy `PolMatrix<T>` (podrobně popsána v odstavci 5.4), která reprezentuje polynomiální matici. Tato třída obsahuje základní metody pro práci s polynomiální maticí – např. funkce pro zjištění velikosti, stupně, typu proměnné, přetížené indexní operátory pro přístup k jednotlivým maticovám koeficientům, dále složitější numerické algoritmy pro výpočet determinantu, hodnosti a kořenů.

Třída `PolMatrix<T>` je odvozená od bázové třídy `PolBase<>` s využitím Bartonova-Nackmanova triku popsaném v odstavci 4.2. Pro tuto třídu jsou potom definovány základní numerické operace, pro jejichž výpočet je použita technika expression templates (viz 4.3). Jsou to tyto operace – sčítání, odčítání, násobení, násobení skalárem, transpozice a konjugovaná transpozice. To, že jsou tyto operace definovány pro bázovou abstraktní třídu se dá využít v budoucnu pro rozšíření knihovny o nové typy polynomiálních matic (např. trojúhelníková, řídká polynomiální matice, ...). Tím že tyto třídy odvodíme od třídy `BasePol`, tak pro ně zároveň definujeme uvedené matematické operace.

Dále knihovna umožňuje provádět trojúhelníkování polynomiální matice (triangularization) a řešit čtyři typy lineárních rovnic s polynomiálními maticemi - $A(s)X(s) = B(s)$, $X(s)A(s) = B(s)$, $A(s)X(s) + B(s)Y(s) = C(s)$ a $X(s)A(s) + Y(s)B(s) = C(s)$, kde $A(s), B(s)$ a $C(s)$ jsou známé polynomiální matice, $X(s)$ a $Y(s)$ jsou neznámé hledané matice. Pro triangularizaci a pro každý typ rovnice je implementována jedna samostatná třída, která řeší daný problém.

5.1 Struktura souborů knihovny PolPack++

Protože je celé knihovna psána s využitím šablon, jsou všechny implementované třídy a funkce umístěny v hlavičkových souborech a to v prostoru jmen (namespace) `polpack`. Knihovna se skládá z následujících hlavičkových souborů:

`polmatrix.h` - v souboru je definována šablona třídy `PolMatrix<T>`, která je podrobně popsána odstavci 5.4. Dále je zde implementována funkce `operator<<` umožňující použít pro tisk polynomální matice standardní datový proud.

`polexpr.h` - obsahuje šablonu bázové třídy `PolBase<>`, šablony tříd a globálních funkcí pro implementaci techniky expression templates.

`axb.h` - obsahuje šablonu třídy `AXBsolver<T>` pro řešení lineární rovnice $A(s)X(s) = B(s)$. Třída je podrobně popsána v odstavci 5.7.

`xab.h` - obsahuje šablonu třídy `XABsolver<T>` pro řešení lineární rovnice $X(s)A(s) = B(s)$. Třída je podrobně popsána v odstavci 5.6.

`axbyc.h` - obsahuje šablonu třídy `AXBYCsolver<T>` pro řešení lineární rovnice $A(s)X(s) + B(s)Y(s) = B(s)$. Třída je podrobně popsána v odstavci 5.8.

`xayby.h` - obsahuje šablonu třídy `XAYBCsolver<T>` pro řešení lineární rovnice $X(s)A(s) + Y(s)B(s) = B(s)$.
Třída je podrobně popsána v odstavci 5.9.

`linalg.h` – v tomto souboru jsou šablony numerických algoritmů lineární algebry, které nejsou implementovány v LAPACKu – např. determinant konstantní matice počítaný s využitím LU rozkladu, hodnost konstantní matice (za pomoci SVD rozkladu), maticové normy, atd.

`global.h` – obsahuje šablony třídy `global<T>`. Tato třída je podrobně popsána v odstavci 5.2.

`traits.h` – obsahuje třídu `trait1<>`. Tato třída je podrobně popsána v odstavci 5.3.

`fft.h` – tento soubor obsahuje C++ interface do knihovny FFTW. Jsou zde přetíženy funkce pro výpočet reálných a komplexních diskrétních Furierových transformací.

`rand.h` – tento soubor obsahuje funkce pro náhodnou inicializaci polynomiální matice.

`bind.h` – v knihovně bidings, která provádí napojení knihovny uBLAS na LAPACK, chybí základní funkce pro QR faktorizaci a výpočet vlastních čísel. V tomto souboru je tedy C++ interface pro uvedené funkce LAPACKu. Interface je implementován stejným způsobem, jako v knihovně bindigs.

5.2 Třída global

Šablona třídy `global<T>` obsahuje globální statické proměnné typu T – `global::zeroing` a `global::eps`. Typ T může být pouze `double` nebo `float`.

Proměnná `zeroin` se používá jako prahovací mez např. při nulování maticových koeficientů (viz metoda `zeroing()` třídy `PolMatrix()`). Proměnná `eps` je zpravidla použita při výpočtu hodnosti matic, kde se počítá počet singulárních čísel větších než `eps`.

Protože se jedná o statické proměnné, mohou být kdykoliv změněny, např:

```
global<double>::zeroing = 10e-6;
```

nebo navráceny do implicitního stavu za pomoci statické metody `setDefault()`:

```
global<double>::setDefault();
```

5.3 Třída trait1

Šablová třídy `trait1<T>` je třída provádějící mapování typu `T` na typ `typename trait1<T>::t1_type`. Technika, která je pro to použita, je podrobně popsána v odstavci 4.1. Třída mapuje následující typy:

```
double --> double
float  --> float
complex<double> --> double
complex<float>  --> float
T  -->  T
```

Příkladem použití je např. výpočet normy matice. Norma se může samozřejmě počítat pro komplexní matici, ale výsledná norma je vždy reálná. Tato třída nám tedy umožní určit návratový typ funkcí které mají komplexní vstup ale reálný výstup.

5.4 Třída PolMatrix

Šablona třídy `PolMatrix<T>` reprezentuje polynomiální matici. Parametr `T` určuje typ dat polynomiální matice. Přestože se jedná o šablonu třídy, typ `T` nemůže být libovolný. Protože pro numerické výpočty je použita knihovna LAPACK, musí být typ `T` kompatibilní s typy, které jsou podporovány touto knihovnou. Jsou to následující čtyři typy – reálné `float` a `double` a komplexní typy `std::complex<float>` a `std::complex<double>`.

Třídy reprezentuje následující typy polynomiálních matic:

- spojitá jednostranná matice
- diskrétní jednostranná matice
- diskrétní oboustranná matice

Všechny typy mohou být reálné nebo komplexní. Zda se jedná o spojitou nebo diskrétní matici je dáno hodnotou soukromé proměnné `var_`, která je výčtového typu:

```
enum var_type { s_var = 1, p_var = 2, z_var = 3, d_var = 4 }
```

kde hodnoty `s_var` a `p_var` jsou pro spojitou polynomiální matici v proměnné s a p a hodnoty `z_var` a `d_var` pro diskrétní matici v proměnné z a d.

Data polynomiální matice jsou uložena ve vektoru matic

```
std::vector<ublas::matrix<T, ublas::column_major> >
```

Typ matice je obecná matice z knihovny uBLAS. Volba této knihovny byla zdůvodněna v kapitole 3. Matice musí být slopcově orientovaná z důvodu použití LAPACKu. LAPACK je psán v jazyce Fortran, ve kterém jsou dvourozměrná pole sloupcově orientovaná (tzn. uložena po sloupcích bezprostředně za sebou). Vektor byl zvolen ze standardní knihovny. Důvodem jeho použití je možnost změny velikosti se zachováním obsahu. Formát uložení je stejný pro jednostrannou i oboustrannou matici.

5.4.1 Typy definované uvnitř třídy

Uvnitř matice jsou definovány následující typy s veřejnými přístupovými právy (public):

`typedef T value_type` – typ dat polynomiální matice.

`typedef std::size_t size_type` – celočíselný kladný typ (obvykle `unsigned`) používaný pro proměnné popisující velikost a stupeň matice.

`typedef ublas::matrix<T,ublas::column_major> storage_matrix` – typ matice použité pro uložení maticových koeficientů.

`typedef typename trait1<T>::t1_type precision` – reálný typ (float nebo double). Tento typ udává, s jakou přesností jsou uloženy data polynomiální matice (samozřejmě i komplexní).

5.4.2 Soukromé parametry třídy

`std::vector<storage_matrix> data_` – vektor matic obsahující maticové koeficienty polynomiální matice.

`size_type ldeg_` – vedoucí stupeň (leading degree) polynomiální matice.

`size_type tdeg_` – koncový stupeň (trailing degree) polynomiální matice.

`size_type rows_` – počet řádků polynomiální matice.

`size_type cols_` – počet sloupců polynomiální matice.

`var_type var_` – proměnná polynomiální matice určující typ matice (spojitá nebo diskrétní). Typ `var_type` je výčtový typ definovaný v hlavičkovém souboru `polexpr.h`.

`precision scale_` – scale polynomiální matice. Význam této proměnné je vysvětlen u metody `scaling()`.

5.4.3 Konstruktory třídy

Pro vytvoření polynomiální matice je definováno pět konstruktorů. Všechny jsou stejné pro spojitou i diskrétní verzi polynomiální matice.

`PolMatrix()` - implicitní konstruktor. Konstruktoru vytváří prázdnou matici – nealokuje žádnou dynamickou paměť. Všechny soukromé proměnné s výjimkou proměnné `scale` inicializuje na nulu (`scale` je roven jedné). To znamená, že není určen typ matice (spojitá nebo diskrétní).

`PolMatrix(size_type DEG, size_type ROWS, size_type COLS, var_type VAR = s_var)` - konstruktor vytvářející jednostrannou polynomiální matici stupně `DEG` a velikosti `ROWS` × `COLS`. Typ matice je určen parametrem `VAR`, který má implicitní hodnotu `s_var`. Pokud tedy není specifikován, je vytvořena spojitá verze matice v proměnné s.

`PolMatrix(size_type TDEG, size_type LDEG, size_type ROWS, size_type COLS, var_type VAR = z_var)` - konstruktor vytvářející oboustrannou polynomiální matici hlavního stupně `LDEG`, vedlejšího stupně `TDEG` a velikosti `ROWS` × `COLS`. Implicitní typ matice je diskrétní v proměnné z. V principu umožňuje vytvořit i spojitou oboustrannou matici, která ale nemá v teorii řízení význam.

`PolMatrix(const PolMatrix& PM)` – kopírovací konstruktor. Vytváří hlubokou kopii objektu `PM`.

`template <class X>`
`PolMatrix(const Expr<storage_matrix, X>& EXP)` – konstruktor vytvářející objekt polynomiální matice z matematického výrazu. Tento konstruktor musel být nadefinován v důsledku použití techniky expression templates, protože matematický výraz s polynomiálními maticemi je speciální objekt. Proto není možné pro vytvoření objektu z výrazu použít kopírovací konstruktor.

5.4.4 Metody třídy

`void resize(size_type NEWtdeg, size_type NEWldeg, size_type NEWrows, size_type NEWcols)` – provádí změnu velikosti a stupně polynomální matice. Původní obsah polynomiální matice není zachován. Nová matice má velikost `NEWrows`× `NEWcols`, hlavní stupeň `NEWldeg` a vedlejší stupeň `NEWtdeg`.

`PolMatrix& operator = (const PolMatrix& PM)` – přiřazovací operátor vytvářející hlubokou kopii polynomiální matice `PM`.

`PolMatrix& operator = (const storage_matrix M)` – operátor přiřazuje polynomiální matici konstantní matici. To znamená, že je vytvořena polynomiální matice stupně nula.

`template<class X>`
`PolMatrix& operator=(const Expr<storage_matrix, X>& x)` – operátor pro vyhodnocení matematického výrazu s polynomiálními maticemi.

`void swap(PolMatrix& M)` – provádí prohození dat dvou polynomiálních matic.

`storage_matrix& operator[] (int d)` – operátor pro přístup k maticovému koeficientu polynomiální matice u d-té mocniny. Operátor vrací referenci, proto ho lze použít na levé i pravé straně přiřazovacího operátoru.

`storage_matrix operator[] (int d) const` – konstantní verze předcházejícího operátoru.

`value_type& operator() (int d)` – tento operátor má obdobnou funkci jako předcházející. Lze ho však použít pouze pro polynomiální matici velikosti 1×1. Opět přistupuje k maticovým koeficientům, ale provádí konverzi matice na typ `value_type`.

`value_type& operator() (int d, size_type m, size_type n)` – operátor pro přístup k jednotlivým prvkům maticových koeficientů polynomiální matice. Proměnná `d` určuje maticový koeficient (u `d`-té mocniny proměnné polynomiální matice), `m` a `n` určují prvek koeficientu.

`PolMatrix operator() (size_type m, size_type n)` – operátor vrací polynom v m-tém řádku a n-tém sloupci polynomiální matice.

`PolMatrix operator() (size_type m1, size_type m2, size_type n1, size_type n2)` – operátor vrací řez z polynomiální matice, který začíná v řádku `m1` a sloupci `n1` a končí v řádku `m2` a sloupci `n2` polynomiální matice.

`const storage_matrix& mat(int n) const` – tato funkce vrací referenci na maticový koeficient u n-té mocniny polynomiální matici. Téměř stejnou funkci má výše popsaný indexní operátor. Tato funkce však nemá kontrolu rozsahu a v případě, že `n` leží mimo interval $\langle -tdeg, ldeg \rangle$, tak vrací prázdnou matici. Tato metoda má speciální použití ve spojitost s třídou `PolBase` (viz 5.5).

`void setVar(var_type VAR)` – nastavuje typ proměnné polynomiální matice.

`void print()` – provádí výpis maticových koeficientů polynomiální matice.

`void rand(precision max = 5.0, char type = 'i')` – naplní matici náhodnými koeficienty v rozsahu 0-max. Parametr `type` musí být `'i'` pro celočíselné koeficienty nebo `'r'` pro reálné koeficienty.

`storage_matrix valueAt(const value_type& value)` – vypočítá hodnotu polynomiální matice v bodě `value`. Je třeba si uvědomit, že `value` je typu `value_type`. To znamená, že tato metoda není schopna vypočítat hodnotu reálné polynomiální matice v komplexním bodě. To je způsobeno tím, že maticové koeficienty (matice typu `ublas::matrix<T,E>`) nelze násobit jiným typem, než typem dat těchto matic. Proto byla implementována ještě jiná metoda pro výpočet hodnoty polynomiální matice – `valueAtComp`. Pro výpočet je použita maticová verze Hornerova algoritmu.

`ublas::matrix<std::complex<precision>, ublas::column_major> valueAtComp(const std::complex<precision>& value)` – tato metoda provede výpočet hodnoty polynomiální matice (komplexní i reálné) v komplexním bodě `value`. Pro komplexní matici je algoritmus stejný jako u metody `valueAt`, pro reálnou matici se počítá zvlášť reálná a komplexní část, z nichž se sestaví výsledná komplexní matice.

`void values(int n, std::complex<precision>* out)` – počítá `n` hodnot (konstantních matic) polynomiální matice v bodech ležících na jednotkové kružnici komplexní roviny. Pro výpočet je použita diskrétní Furierova transformace. Hodnoty (konstantní matice) jsou postupně uloženy do jednorozměrného pole. Algoritmus výpočtu je podrobně popsán v [7].

`void shift(int n = 1)` – tato metoda provádí násobení polynomiální matice pro-

měnným parametrem umocněným na `n`.

`void sylvMatrix(storage_matrix& M, size_type nb, char type = 'c')` – metoda sestavuje Sylvestrovu matici. Proměnná `type` musí být `'c'` pro sloupcovou nebo `'r'` pro řádkovou formu Sylvestrovy matice. Počet nenulových bloků sloupci resp. řádku je dán hodnotou `nb`.

`void compainMatrix(storage_matrix& M)` – metoda sestavuje blokovou companion matici.

`void hurwitzMatrix(storage_matrix& M, size_type n = 0)` – metoda sestavuje blokovou Hurwitzovu matici.

`void scaling(precision k)` – metoda provádí změnu měřítka polynomiální matice (scale). Je-li s proměnná polynomiální matice, potom je nahrazena novou proměnnou $\hat{s} = ks$. To znamená, že maticový koeficient u i-té mocniny s je vydělen hodnotou k^i. Privátní proměnné `scale_` je přiřazena hodnota `k`.

`void zeroing(precision tol = global<precision>::zeroing)` – metoda nuluje prvky maticových koeficientů, které jsou menší než `tol`.

`void clearing()` – metoda maže nulové maticové koeficienty. Prochází postupně všechny maticové koeficienty od nejvyšší mocniny po nultou mocninu až do doby, než nalezne nenulový koeficient. Při průchodu maže nulové koeficienty (uvolňuje paměť). To znamená, že sníží stupeň polynomiální matice. Je-li polynomiální matice oboustranná, tak se operace opakuje i pro záporné stupně matice.

`ublas::vector<size_type> rowDeg()` – metoda vrací řádkový stupeň polynomiální matice.

`ublas::vector<size_type> colDeg()` - metoda vrací sloupcový stupeň polynomiální matice.

`ublas::vector<size_type> rowLdeg()` – metoda vrací hlavní řádkový stupeň polynomiální matice.

`ublas::vector<size_type> colLdeg()` – metoda vrací hlavní řádkový stupeň polynomiální matice.

`ublas::vector<size_type> rowTdeg()` – metoda vrací vedlejší řádkový stupeň polynomiální matice.

`ublas::vector<size_type> colTdeg()` – metoda vrací vedlejší sloupcový stupeň

polynomiální matice.

`storage_matrix rowLcoef()` – metoda vrací matici hlavních řádkových koeficientů polynomiální matice (row leading coefficient matrix)

`storage_matrix colLcoef()` – metoda vrací matici hlavních slopcových koeficientů polynomiální matice (row leading coefficient matrix)

`storage_matrix rowTcoef()` – metoda vrací matici vedlejších řádkových koeficientů polynomiální matice (row trailing coefficient matrix)

`storage_matrix colTcoef()` – metoda vrací matici vedlejších řádkových koeficientů polynomiální matice (row trailing coefficient matrix)

`bool isRowRed()` – metoda provádí test řádkové redukovanosti polynomiální matice.

`bool isColRed()` – metoda provádí test sloupcové redukovanosti polynomiální matice

`double norm(char type = 'b', char norm = '2')` – metoda provádí výpočet norem polynomiální matice. Proměnná `type` určuje, z čeho se bude norma počítat. Pokud je `type` rovno `'b'`, potom se z maticových koeficientů sestavuje složená matice

$$A_c = [\ A_{-tdeg} \quad A_{-tdeg+1} \quad \dots \quad A_{ldeg-1} \quad A_{ldeg-1}\ ] \tag{5.1}$$

a norma se počítá z matice A_c. Pokud je `type` rovno `'l'`, potom se počítá norma maticového koeficientu u nejvyšší mocniny. Poslední hodnotou, které může `type` nabývat je `'m'` a potom se počítají normy všech maticových koeficientů a vrací se maximální. Parametr `norm` určuje typ počítané normy a může nabývat hodnot `'1'` pro řádkovou normu, `'2'` pro kvadratickou normu (největší singulární číslo), `'i'` pro sloupcovou normu a `'f'` pro Frobeniovu normu.

`PolMatrix det(precision tol = global<precision>::zeroing)` – metoda počítá determinant polynomiální matice. Algoritmus výpočtu je podrobně popsán v kapitole 6.

`size_type rank(precision tol = global<precision>::eps)` – metoda počítá hodnost polynomiální matice. Algoritmus výpočtu je podrobně popsán v kapitole 6.

`void roots(ublas::vector<std::complex<precision>& r, char method = 'd', precision tol = global<precision>::zeroing)` – metoda provádí výpočet kořenů polynomiální matice. Proměnná `d` určuje použitý algoritmus a může nabývat hodnoty `'d'`, pak je použit výpočet pomocí determinantu, nebo `'e'` a pak se provede výpočet pomocí vlastních čísel. Algoritmus výpočtu je podrobně popsán v kapitole 6.

`size_type deg() const` – metoda vrací hlavní stupeň (leading degree) polynomiální matice.

`size_type ldeg() const` – metoda vrací hlavní stupeň (leading degree) polynomiální matice.

`size_type tdeg() const` – metoda vrací vedlejší stupeň (trailing degree) polynomiální matice

`size_type rows() const` – metoda vrací počet řádků polynomiální matice.

`size_type cols() const` – metoda vrací počet sloupců polynomiální matice.

`var_type var() const` – metoda vrací typ proměnné polynomiálmí matice.

`double scale() const` – metoda vrací velikost soukromé proměnné `scale_`.

5.5 Základní numerické operace a třída PolBase

V tomto odstavci je popsána definice základních numerických operací pro polynomiální matice. Operace jsou definovány pro třídu `PolBase<M,I>` za pomoci techniky expression templates, jejíž implementace pro jednoduchý kontejner je podrobněji popsána v kapitole 4. Numerické operace pro třídu PolBase jsou implementovány obdobným způsobem. V tomto odstavci jsou pouze uvedeny rozdíly proti implementaci v kapitole 4.

Od třídy `PolBase` je za pomoci Bartonova-Nackmanova triku (viz kapitola 4) odvozena třída `PolMatrix<T>`:

```
template <class M, class I>
class PolBase {
...
}

template<class T>
class PolMatrix : public
    PolBase<ublas::matrix<T, ublas::column_major>, PolMatrix<T> > {
...
}
```

Šablonový typ `M` je tedy typ maticových koeficientů a typ `I` je typ odvozené třídy. Třída `PolBase` má všechny metody definovány s přístupovými právy `public` následujícím způsobem:

```
template <class M, class I>
class PolBase {
public:

    const M& operator[]( int n ) const {
        return static_cast<const I*>(this)->mat(n);
    }

    size_type ldeg() const {
        return ( static_cast<const I*>(this)->ldeg() );
    }

    ...
}
```

Přetížený indexní operátor volá metodu `mat(int n)` zděděného objektu, která zpřístupňuje maticové koeficienty polynomiální matice, které jsou typu `M`. Stejným způsobem je definována metoda ldeg() vracející hlavní stupeň polynomiální matice a dále metody pro určení vedlejšího stupně, velikosti matic, typu proměnné a velikost parametru scale.

Dále bylo potřeba upravit třídu `ConstRef` (její význam je uveden v odstavci 4.3):

```
template <class M, class C>
class ConstRef {
    const C& c_;
public:
    ConstRef( const C& c ) : c_(c) {}

    M operator[](int n) const { return c_[n]; }
    size_type ldeg() const { return c_.ldeg(); }
    ...
};
```

Šablonový typ `C` je typ třídy, na který třída odkazuje (to bude vždy `PolBase`) Samotná reference na třídu je soukromým parametrem. Dále má definován indexní operátor a metody, které zpřístupní všechny metody třídy typu `C` (tedy `PolBase`). Typ `M` je opět typ maticového koeficientu.

Třída `Expr` reprezentující matematický výraz, byla upravena do následující podoby:

```
template <class M, class E>
class Expr {
public:
    size_type ldeg() const { return ldeg_; }
```

```
    ...

    M operator[](int n) const { return e_[n]; }
private:
    E e_;
    size_type ldeg_, tdeg_;
    ...
};
```

Samotný výraz je reprezentován privátním objektem `e_`. Ostatní soukromé parametry třídy udávají informace o výrazu, tzn. výsledný hlavní stupeň výrazu (proměnná `ldeg_`), vedlejší stupeň, velikosti matice, typ proměnné výrazu a scale. Tyto proměnné jsou inicializovány konstruktorem třídy.

Třída `BinaryOp` operator zůstala úplně stejná. Třída `OpAdd` provádějící samotný součet však již musela být upravena:

```
template <class M>
class OpAdd {
public:
    static inline M apply( const M& a, const M& b ) {
        if (a.size1() == 0)
            return b;
        if (b.size1() == 0)
            return a;
        return a + b;
    }
};
```

Implementace uvedená v kapitole 4 umožňuje sčítat pouze stejně dlouhá pole (objekty `Array`). Při výpočtu součtu je volán indexní operátor třídy Array a v případě, že by délka polí byla rozdílná, by došlo k překročení meze u kratšího pole. To se dá jednoduše ošetřit tak, že operátor při překročení meze pole vrátí hodnotu nula. Podobně je to provedeno v případě třídy `PolBase`, jejíž indexní operátor zavolá metodu `mat(int n)` třídy `PolMatrix`, která v případě překročení mezí vrátí prázdnou matici. To nám umožní sčítat matice s různými stupni. Statická metoda `aplly` třídy `OpAdd` tedy provádí součet pouze tehdy, jsou-li obě sčítané matice neprázdné.

Globální přetížené operátory pro sčítání vypadají obdobně jako v odstavci 4.3. Byly do nich přidány funkce pro kontrolu shodnosti velikosti, proměnné `var_` a `scale_` sčítaných objektů pomocí standardního makra `assert`. Operátor vytváří a vrací objekt typu `Expr`, který reprezentuje výraz, a do nějž je uložen stupeň výsledného výrazu, velikost matic, hodnotu proměnné `var_` a velikost hodnoty scale.

Stejným způsobem, jako je provedeno sčítání je provedeno odčítání. Samotné odčítání provede třída `OpSub` (obdoba třídy `OpAdd`), která opět obsahuje statickou metodu `apply` provádějící výpočet rozdílu. Globální operátory pro odčítání jsou úplně stejné (samozřejmě až na typ operátoru)

5.5.1 Násobení

Způsobem, kterým je implementováno sčítání a odčítání, však již nelze provést násobení. Pro násobení matic proto byla implementována nová třída `ExprProd`, která je obdobou třídy `BinaryOp`. Třída `ExprProd` sama provádí výpočet násobení aniž by k tomu používala další třídu.

```
template <class M, class A, class B>
class ExprProd {
    A a_;
    B b_;
 public:
    ExprProd(const A& a, const B& b) : a_(a), b_(b) {}

    M operator[] (int n) const {
    ...
    }
};
```

šablonové typy `A` a `B` jsou typy násobených objektů (mohou být tedy typu `Expr` a `ConstRef`, která odkazuje na třídu `PolBase`). Samotný výpočet násobení je proveden v přetížené indexním operátoru. Operátor tedy vrací částečný výsledek násobení – tedy konstantní matici odpovídající indexu `n` (což je matice u `n`-té mocniny polynomiální matice, která je výsledkem výrazu). Algoritmu násobení je v principu konvoluce vektorů matic. Voláním indexního operátoru se však nepočítá kompletní konvoluce, ale pouze její výsledek na `n`-té pozici. K násobení matic při výpočtu konvoluce lze použít několik způsobů – klasické a blokové násobení matic knihovny uBLAS nebo lze za pomoci napojení uBLASu na LAPack a Atlas využít násobení za pomoci knihovny BLAS, kterou tyto knihovny obsahují.

Dále je samozřejmě třeba přetížit operátor násobení formou globálních operátorových funkcí. Operátory se velmi podobají operátorům pro sčítání a odčítání. Rozdíl je v tom, že vrací objekt typu `Expr`, který má jeden šablonový typ `ExprProd`.

5.5.2 Násobení skalárem

Pro násobení skalárem bylo třeba nejprve definovat třídu reprezentující skalár:

```
template <class T>
```

```
class Scalar {
    T t_;
public:
    explicit Scalar(const T& t) : t_(t) {}

    T operator[]( int n ) const { return t_; }
};
```

Dále je třeba nadefinovat třídu provádějící samotné násobení skalárem (obdoba tříd `OpAdd`, `OpSub`):

```
template<class T1, class T2>
class OpMulScalar {
 public:
    static inline typename PromoteTraits<T1, T2>::T_promote
    apply(const T1& a, const T2& b) {
        return a * b;
    }
};
```

U statické funkce `apply` je třeba určit návratový typ. Typy `T1` a `T2` reprezentují skalár a matici (maticový koeficient) nebo naopak. Návratový typ je vždy matice. Pro rozlišení, zda je to typ `T1` nebo `T2`, slouží třída třída `PromoteTraits<T1, T2>` (využívá techniku Traits popsanou v kapitole 4). Pro násobení skalárem je ještě třeba přetížit globální operátorové funkce.

5.5.3 Unární operace

Dále má třída `PolBase` definován unární operátor minus a operátory transpozice a konjugované transpozice. Pro implementaci unárních operátorů je třeba implementovat třídu `UnaryOp`, která má obdobný význam, jako měla třída `BinaryOp`:

```
template <class M, class A, class Op>
class UnaryOp {
    A a_;
public:
    UnaryOp( const A& a ) : a_(a) {}

    M operator[]( int n ) const {
        return Op::apply( a_[n] );
    }
};
```

Pro unární minus se dále musí implementovat třída `OpUnaryMinus` aplikující samotný operátor:

```
template <class M>
class OpUnaryMinus {
 public:
    static inline M apply( const M& a ) {
        return -a;
    }
};
```

a nakonec je samozřejmě třeba přetížit operátor minus pro třídu `PolBase` a třídu `Expr`. Obdobným způsobem, jako je implementován operátor unární minus, je implementována transpozice a konjugovaná transpozice polynomiální matice. Pro tyto operace však není definován žádný operátor, ale funkce – `transpose` pro transpozici matice a `conjTransposeC`, `conjTransposeD` pro transpozici spojité a diskrétní konjugované transpozice. Pro konjugovanou transpozici jsou tedy definovány funkce zvlášť pro spojitý a diskrétní případ, což je způsobeno odlišnou definicí (a tedy i výpočtem) konjugované transpozice pro oba případy. V hlavičkové souboru `polmatrix.h` je potom implementována inline funkce `conjTranspose`, která sjednocuje oba případy do volání jedné funkce.

Nakonec jednoduchý příklad toho, co jsme schopni s knihovnou PolPack++ počítat bez použití odkládacích objektů a v jednom iteračním cyklu:

```
PolMatrix<double> A(2,2,2), B(1,2,2), C(3,2,2), D(1,2,2), E(2,2,2), X
...
X = 2.0 * A + traspose(B) * (C + conjTranpose(D)) - E;
```

5.6 Třída XABsolver

Třída `XABsolver<T>` řeší lineární polynomyální matici $X(s)A(s) = B(s)$. Definovaná je v hlavičkové souboru `xab.h`. Šablonový typ `T` je typ dat polynomiálních matic, pro než se řeší rovnice (nebo-li jinak `PolMatrix<T>::value_type`).

Algoritmus řešení rovnice $X(s)A(s) = B(s)$

Nechť

$$A = A_0 + A_1 s + \cdots + A_{d_a} s^{d_a} \tag{5.2}$$
$$B = B_0 + B_1 s + \cdots + B_{d_b} s^{d_b} \tag{5.3}$$
$$X = X_0 + X_1 s + \cdots + X_d s^d \tag{5.4}$$

Potom můžeme rovnici $A(s)X(s) = B(s)$ přepsat do soustavy rovnic

$$\left[\begin{array}{cccc} X_0 & X_1 & \dots & X_d \end{array}\right] A_s = \left[\begin{array}{ccccc} B_0 & B_1 & \dots & B_{d_b} & 0 \quad \dots \end{array}\right], \tag{5.5}$$

kde matice A_s je bloková sylvestrova matice polynomiální matice $A(s)$ a má tvar

$$A_s = \begin{pmatrix} A_0 & A_1 & \cdots & A_{d_a} & 0 & 0 & \dots & 0 \\ 0 & A_0 & A_1 & \cdots & A_{d_a} & 0 & \dots & 0 \\ 0 & 0 & A_0 & A_1 & \cdots & A_{d_a} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \dots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & A_0 & A_1 & \dots & A_{d_a} \end{pmatrix}. \tag{5.6}$$

Matice A_s má $k+1$ blokových řádků. Nyní je ještě třeba určit stupeň k matice $X(s)$. Jeho minimum je dáno vztahem

$$k = \max\left(\max_i(\deg_{c_i} B(s) - \deg_{c_i} A(s)), 0\right), \tag{5.7}$$

kde $\deg_{c_i} A(s)$ a $\deg_{c_i} B(s)$ jsou i-té sloupcové stupně matice A a B. Horní odhad stupně určíme s využitím lemma 4.1 v [5]. Rovnici $XA = B$ přepíšeme do tvaru

$$\begin{bmatrix} A^T & -B^T \end{bmatrix} \begin{bmatrix} X^T \\ I \end{bmatrix} = 0. \tag{5.8}$$

Označme

$$\widehat{X} = \begin{bmatrix} X^T \\ I \end{bmatrix} = 0. \tag{5.9}$$

$\widehat{X}$ je nulový prostor (null space) složené matice $[\ A^T \quad -B^T\ ]$. Jeho maximální stupeň je dle uvedeného lemma

$$d = \sum_i \deg_{c_i}[\ A^T \quad -B^T\ ] - \min_i \deg_{c_i}[\ A^T \quad -B^T\ ]. \tag{5.10}$$

Z toho potom plyne, že maximální stupeň je také roven d. Algoritmus hledající řešení s minimálním stupněm bude následující:

Vstup: Polynomiální matice $A(s)$ rozměru $m \times n$ a $B(s)$ rozměru $k \times n$.

Výstup: matice X rozměru $k \times m$.

Krok 1 Test možnosti existence řešení – nechť $r_1 = \operatorname{rank} A(s)$ a nechť

$$r_2 = \operatorname{rank} \begin{bmatrix} A(s) \\ B(s) \end{bmatrix}. \tag{5.11}$$

Jestliže $r1 \neq r2$, řešení nemůže existovat – ukonči algoritmus.

Krok 2 Nechť *mindeg* je minimální stupeň matice $X(s)$ vypočtený dle 5.7 a nechť *maxdeg* je maximální stupeň matice $X(s)$ vypočtený dle 5.10.

Krok 3 Nalezni minimální stupeň d matice $X(s)$ na intervalu $\langle mindeg, maxdeg\rangle$ pro který je řešitélná soustava 5.5.

Krok 4 Řeš soustavu 5.5 pro nalezené d.

Poznámka 1. Minimální stupeň d na se nejčastěji hledá binárním půlením intervalu. Nalezneme-li nějaké řešení stupně $\hat{d} \in \langle mindeg, maxdeg \rangle$, potom určitě existují řešení stupně $\hat{d}, \ldots, maxdeg$ a dále zkoušíme hledat řešení na intervalu $\langle mindeg, \hat{d} \rangle$. Vybereme stupeň uprostřed tohoto intervalu a pokud existuje řešení tohoto stupně, pokračujeme hledáním v dolní části intervalu, pokud neexistuje pokračujeme v hledání řešení v horní části.

5.6.1 Privátní proměnné třídy XABsolver

`PolMatrix<T>* A_, B_` – proměnné `A_` a `B_` jsou ukazatele na objekt typy `PolMatrix<T>`. Tyto ukazatele odkazují na polynomiální matice definující řešenou rovnici. Ukazatele jsou použity, aby nebyly vytvářeny kopie objektů.

5.6.2 Konstruktory třídy XABsolver

`XABsolver()` – implicitní konstruktor

`XABsolver(PolMatrix<T>& A, PolMatrix<T>& B)` – konstruktor ukládá do privátních parametrů `A_` a `B_` adresy polynomiálních matic `A` a `B` (matic, pro něž se řeší rovnice).

5.6.3 Veřejné metody třídy XABsolver

`void setA(PolMatrix<T>& A)` – metoda ukládá do privátního parametru `A_` adresu polynomiální matice `A`.

`void setB(PolMatrix<T>& B)` – metoda ukládá do privátního parametru `B_` adresu polynomiální matice `B`.

`bool solutionTest()` – metoda testuje možnost existence řešení polynomiální rovnice. Provádí výpočet hodnosti matice `A` a pokud matice `A` nemá plnou řádkovou hodnost i matice složené z `A` a `B`. Řešení může existovat pouze při rovnosti těchto hodností.

`size_type lowerBound()` – metoda počítá dolní odhad stupně řešení polynomiální matice.

`size_type upperBound()` – metoda počítá horní odhad stupně řešení polynomiální matice.

`bool solve(PolMatrix<T>& X, int deg = -1)` – metoda pokoušející se řešit polynomiální rovnici. Vrací `true` v případě nalezení řešení. Řešení je uloženo v proměnné `X`.

Proměnná deg udává stupeň hledaného řešení. Je-li menší než 0 (implicitní případ), hledá se řešení s minimálním stupněm.

5.7 Třída AXBsolver

Třída AXBsolver<T> řeší lineární polynomiální matici $A(s)X(s) = B(s)$. Definovaná je v hlavičkové souboru axb.h. Šablonový typ T je typ dat polynomiálních matic, pro něž se řeší rovnice.

Algoritmus řešení rovnice $A(s)X(s) = B(s)$

Rovnici lze transpozicí převést na tvar

$$X(s)^T A(s)^T = B(s)^T. \tag{5.12}$$

Tuto úlohu však řeší třída XABsolver<T> popsaná v předcházejícím odstavci (5.6) a proto je tato třída pro řešení rovnice použita.

5.7.1 Privátní proměnné třídy AXBsolver

PolMatrix<T> A_, B_ – proměnné obsahují transponované kopie matic, pro něž se řeší rovnice.

XABsolver<T> solver_ – objekt, který provádí řešení rovnice pro A_ a B_.

5.7.2 Konstruktory třídy AXBsolver

AXBsolver() – implicitní konstruktor

AXBsolver(const PolMatrix<T>& A, const PolMatrix<T>& B) – konstruktor provede přiřazení transponovaných matic A a B do privátních proměnných A_ a B_. Dále provede inicializaci privátního objektu solver_.

5.7.3 Veřejné metody třídy AXBsolver

void setA(const PolMatrix<T>& A) – metoda přiřadí privátní proměnné A_ transpozici polynomiální matici A a zároveň je provedena inicializace objektu solver_.

void setB(const PolMatrix<T>& B) – metoda přiřadí privátní proměnné B_ transpozici polynomiální matici B a zároveň je provedena inicializace objektu solver_.

bool solutionTest() – metoda provádí test možnosti řešení (volá funkci solver_.solutionTest())

`size_type lowerBound()` – metoda počítá dolní odhad stupně řešení (volá funkci `solver_.lowerBound()`).

`size_type upperBound()` – metoda počítá horní odhad stupně řešení (volá funkci `solver_.upperBound()`).

`bool solve(PolMatrix<T>& X, int deg = -1)` – metoda pokoušející se řešit polynomiální rovnici. Vrací `true` v případě nalezení řešení. Volá metodu `solve(PolMatrix<T>&X, int deg)` objektu `solver_`.

5.7.4 Příklad výpočtu

Hledáme řešení polynomiální rovnice

$$\begin{bmatrix} -8-7s & 3-2s \\ 3+4s & 4+6s \end{bmatrix} X = \begin{bmatrix} -44-62s-9s^2 & 33-58s-50s^2 \\ 37+59s+10s^2 & 3+18s+14s^2 \end{bmatrix} \tag{5.13}$$

Řešení x nalezne následující program:

```
#include "axb.h"
using namespace polpack;

int main(int argc, char *argv[]) {

    PolMatrix<double> A(2,2,2), B(3,2,2), X;
    PolMatrix<double>::storage_matrix a0(2,2), a1(2,2),
                                      b0(2,2), b1(2,2), b2(2,2);
    \\inicializace matic
    a0(0,0) = -8.0; a0(0,1) = 3.0; a0(1,0) = 3.0; a0(1,1) = 4.0;
    a1(0,0) = -7.0; a1(0,1) = -2.0; a1(1,0) = 4.0; a1(1,1) = 6.0;
    b0(0,0) = -44.0; b0(0,1) = 33.0; b0(1,0) = 37.0; b0(1,1) = 3.0;
    b1(0,0) = -62.0; b1(0,1) = -58.0; b1(1,0) = 59.0; b1(1,1) = 18.0;
    b2(0,0) = -9.0; b2(0,1) = -50.0; b2(1,0) = 10.0; b2(1,1) = 14.0;
    A[0] = a0; A[1] = a1;
    B[0] = b0; B[1] = b1; B[2] = b2;

    cout << A << endl << B << endl;

    AXBsolver<double> solver;
    solver.setA(A); solver.setB(B); //inicializace solveru

    if (solver.solve(X)) //pokus o hledání reseni s min. stupnem
        cout << "solution is:" << endl << X;
```

```
    else
        cout << "no solution found";

    return 0;
}
```

Výstup programu je následující:

```
Polynomial matrix - size: 2x2, degree - 2
-8 - 7s    3 - 2s
3 + 4s     4 + 6s

Polynomial matrix - size: 2x2, degree - 3
-44 - 62s - 9s^2    33 - 58s - 50s^2
37 + 59s + 10s^2    3 + 18s + 14s^2

solution is:
Polynomial matrix - size: 2x2, degree - 1
7 + 1s    -3 + 8s
4 + 1s    3 - 3s
```

5.8 Třída AXBYCsolver

Třída `AXBYCsolver<T>` řeší lineární rovnici $A(s)X(s) + B(s)Y(s) = C(s)$, kde $A(s)$, $B(s)$ a $C(s)$ jsou známé polynomiální matice a $X(s)$ a $Y(s)$ jsou hledané polynomiální matice. Třída je definovaná v hlavičkovém souboru `axbyc.h`. Šablonový typ `T` je typ dat polynomiálních matic, pro něž se řeší rovnice.

Algoritmus řešení rovnice $A(s)X(s) + B(s)Y(s) = C(s)$

Řešenou rovnici můžeme přepsat do maticového tvaru

$$\begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = C, \tag{5.14}$$

jehož transpozicí získáme tvar

$$\begin{bmatrix} X^T & Y^T \end{bmatrix} \begin{bmatrix} A^T \\ B^T \end{bmatrix} = C^T. \tag{5.15}$$

Označme

$$A_s = \begin{bmatrix} A^T \\ B^T \end{bmatrix}, B_s = C^T, X_s = \begin{bmatrix} X^T & Y^T \end{bmatrix}. \tag{5.16}$$

Nyní stačí vyřešit rovnice $X_s(s)A_s(s) = B_s(s)$ a výsledné řešení $X(s)$ a $Y(s)$ získáme z matice $X_s(s)$. K řešení této rovnice lze tedy použít třídu `XABsolver<T>`.

5.8.1 Privátní proměnné třídy AXBYCsolver

`XABsolver<T> solver_` – objekt použitý k řešení rovnice.

`bool issolver` – proměnná říkající, zda je inicializován objekt `solver_`. Inicializace se provádí privátní metodou `make_solver()`.

`PolMatrix<T> *A_, *B_, *C_` – odkazy na objekty polynomiálních matic, pro něž se řeší rovnice.

`PolMatrix<T> As_, Bs_, Xs_` – tyto objekty se předávají objektu `solver_`. Odpovídají maticím v 5.16.

5.8.2 Privátní metody třídy AXBYCsolver

`void make_solver()` – sestaví polynomiální matice `As` a `Bs` a provede inicializaci objektu `solver_` (volá metody `solver_.setA(As)` a `solver_.setA(Bs)`). Hodnota proměnné `issolver` je nastavena na hodnotu `true`.

5.8.3 Konstruktory třídy AXBYCsolver

`AXBYCsolver()` – implicitní konstruktor.

`AXBYCsolver(const PolMatrix<T>& A, const PolMatrix<T>& B, const PolMatrix<T>& C)` – konstuktor přiřadí soukromým parametrům `A_`, `B_` a `C_` adresy objektů `A`, `B` a `C` a volá metodu `make_solver()`

5.8.4 Veřejné metody třídy AXBYCsolver

`void setA(const PolMatrix<T>& A)` – metoda ukládá do privátního parametru `A_` adresu polynomiální matice `A`. Hodnota proměnné `issolver` je nastavena na `false`.

`void setB(const PolMatrix<T>& B)` – metoda ukládá do privátního parametru `B_` adresu polynomiální matice `B`. Hodnota proměnné `issolver` je nastavena na `false`.

`void setC(const PolMatrix<T>& C)` – metoda ukládá do privátního parametru `C_` adresu polynomiální matice `C`. Hodnota proměnné `issolver` je nastavena na `false`.

`bool solutionTest()` – v případě, že není inicializovaný objekt `solver_`, je provedena jeho inicializace a následně test možnosti existence řešení (provede volání `solver_.solutionTest()`.

`size_type lowerBound()` – vrací horní odhad stupně řešení.

`size_type upperBound()` – vrací dolní odhad stupně řešení.

`bool solve(PolMatrix<T>& X, PolMatrix<T>& Y, int deg = -1)` – pokouší se řešit rovnici. Hledá řešení se stupněm `deg`. V případě že `deg` je menší než nula, hledá řešení s minimálním stupněm. Pokud nalezne řešení, vrací `true` a výsledek ukládá do `X` a `Y`.

5.9 Třída XAYBCsolver

Třída `AXBYCsolver<T>` řeší lineární rovnici $X(s)A(s) + Y(s)B(s) = C(s)$, kde $A(s)$, $B(s)$ a $C(s)$ jsou známé polynomiální matice a $X(s)$ a $Y(s)$ jsou hledané polynomiální matice. Třída je definovaná hlavičkové souboru `xaybc.h`. Šablonový typ `T` je typ dat polynomiálních matic, pro něž se řeší rovnice.

Algoritmus řešení rovnice $X(s)A(s) + Y(s)B(s) = C(s)$

Rovnici můžeme upravit do maticového tvaru

$$\begin{bmatrix} X & Y \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = C \tag{5.17}$$

Označme

$$A_s = \begin{bmatrix} A \\ B \end{bmatrix}, B_s = C, X_s = \begin{bmatrix} X & Y \end{bmatrix}. \tag{5.18}$$

Nyní stačí vyřešit rovnice $X_s(s)A_s(s) = B_s(s)$ a výsledné řešení $X(s)$ a $Y(s)$ získáme z matice $X_s(s)$. K řešení této rovnice lze tedy použít třídu `XABsolver<T>`.

5.9.1 Privátní proměnné třídy XAYBCsolver

`XABsolver<T> solver_` – objekt použitý k řešení rovnice.

`bool issolver` – proměnná říkající, zda je inicializován objekt `solver_`. Inicializace se provádí privátní metodou `make_solver()`.

`PolMatrix<T> *A_, *B_, *C_` – odkazy na objekty polynomiálních matic, pro něž se řeší rovnice.

`PolMatrix<T> As_, Bs_, Xs_` – tyto objekty se předávají objektu `solver_`. Odpovídají maticím v 5.18.

5.9.2 Privátní metody třídy AXBYCsolver

`void make_solver()` – provádí inicializaci privátního objektu `solver_`. Sestavuje složené polynomiální matice a proměnnou `issolver` nastavuje na hodnotu `true`.

5.9.3 Konstruktory třídy XAYBCsolver

`AXBYCsolver()` – implicitní konstruktor.

`AXBYCsolver(const PolMatrix<T>& A, const PolMatrix<T>& B, const PolMatrix<T>& C)` – konstuktor přiřadí soukromým parametrům `A_`, `B_` a `C_` adresy objektů `A`, `B` a `C` a volá metodu `make_solver()`.

5.9.4 Veřejné metody třídy XAYBCsolver

`void setA(const PolMatrix<T>& A)` – metoda ukládá do privátního parametru `A_` adresu polynomiální matice `A`. Hodnota proměnné `issolver` je nastavena na `false`.

`void setB(const PolMatrix<T>& B)` – metoda ukládá do privátního parametru `B_` adresu polynomiální matice `B`. Hodnota proměnné `issolver` je nastavena na `false`.

`void setC(const PolMatrix<T>& C)` – metoda ukládá do privátního parametru `C_` adresu polynomiální matice `C`. Hodnota proměnné `issolver` je nastavena na `false`.

`bool solutionTest()` – v případě, že není inicializovaný objekt `solver_`, je provedena jeho inicializace a následně test možnosti existence řešení (provede volání `solver_.solutionTest()`.

`size_type lowerBound()` – vrací horní odhad stupně řešení.

`size_type upperBound()` – vrací dolní odhad stupně řešení.

`bool solve(PolMatrix<T>& X, PolMatrix<T>& Y, int deg = -1)` – pokouší se řešit rovnici. Hledá řešení se stupněm `deg`. V případě že `deg` je menší než nula, hledá řešení s minimálním stupněm. Pokud nalezne řešení, vrací `true` a výsledek ukládá do `X` a `Y`.

5.9.5 Příklad výpočtu

Příklad ukáže nalezení řešení rovnice

$$X(s)(-1+s)+Y(s)(-4+s)=-1-2s^2+10s^3 \tag{5.19}$$

Rovnici řeší následující program:

```
#include "xaybc.h"
using namespace polpack;

int main(int argc, char *argv[]) {
    PolMatrix<double> A(1,1,1), B(1,1,1), C(3,1,1), X, Y;

    A(0) = -1.0; A(1) = 1.0; //inicializace polynomu
    B(0) = -4.0; B(1) = 1.0;
    C(0) = -1.0; C(1) = 0.0; C(2) = -2.0; C(3) = -10.0;

    cout << A << endl << B << endl << C << endl;

    XAYBCsolver<double> solver;
    solver.setA(A); solver.setB(B); solver.setC(C);

    if (solver.solve(X,Y)) //hledani reseni
        cout << "solution is:" << endl << "X:" << endl
        << X << endl << "Y:" << endl << Y;
    else
        cout << "no solution found";

    return 0;
}
```

Výstup programu je následující:

```
Polynomial matrix - size: 1x1, degree - 1
-1 + s

Polynomial matrix - size: 1x1, degree - 1
-4 + s

Polynomial matrix - size: 1x1, degree - 3
-1 - 2s^2 - 10s^3

solution is:
X:
Polynomial matrix - size: 1x1, degree - 2
-1.02778 - 3.79861s - 13.0069s^2

Y:
Polynomial matrix - size: 1x1, degree - 2
0.506944 + 0.819444s + 3.00694s^2
```

5.10 Třída TRIsolver

Třída TRIsolver<T> provádí převod polynomiální matice na horní nebo dolní trojúhelníkovou polynomiální matici. Na trojúhelníkový tvar můžeme převést libovolnou polynomiální matici. To znamená, že pro každou polynomiální matici $A(s)$ existují unimodulární matice $U_c(s)$ a $U_r(s)$, které redukují matici $A(s)$ na dolní ($T_c(s)$) resp. horní ($T_r(s)$) trojúhelníkový tvar:

$$A(s)U_c(s) = T_c(s) \tag{5.20}$$
$$U_r(s)A(s) = T_r(s) \tag{5.21}$$

Algoritmus

Algoritmus využívá sylvestrovský přístup a je podrobně popsán v [5].

5.10.1 Privátní proměnné třídy TRIsolver

`bool Umatrix` – proměnná říkající, zda se bude počítat redukční matice $U(s)$

`PolMatrix<T> A_` – objekt, pro který se hledá trojúhelníkový tvar.

5.10.2 Konstruktory třídy TRIsolver

`TRIsolver()` – implicitní konstruktor.

`TRIsolver(PolMatrix<T>& A)` – konstruktor ukládá kopii objektu `A` do privátní proměnné `A_`.

5.10.3 Veřejné metody třídy TRIsolver

`size_type maxDeg()` – počítá maximální stupeň unimodulární redukční matice $U(s)$.

`bool solve(PolMatrix<T>& T, char type = 'c', typename trait1<T>::t1_type tol = global<typename trait1<T>::t1_type>::zeroing)` – metoda provádí převod polynomiální matice `A_` na trojúhelníkový tvar. Počítá se takový tvar, který vznikne redukcí matice $U(s)$ s minimálním stupněm. Redukční matice $U(s)$ není počítána. Výsledný trojúhelníkový tvar je uložen v proměnné `T`. Proměnná `type` může nabývat hodnot `'c'` a `'r'` pro výpočet horní a dolní rojúhelníkovou matici.

`bool solve(PolMatrix<T>& T, int deg, char type = 'c', typename trait1<T>::t1_type tol = global<typename trait1<T>::t1_type>::zeroing)` – tato metoda má úplně stejnou funkci jako předcházející, pouze zde přibyla proměnná `deg`, která definuje stupeň redukční matice $U(s)$.

`bool solve(PolMatrix<T>& T, PolMatrix<T>& U, char type = 'c', typename trait1<T>::t1_type = global<typename trait1<T>::t1_type>::zeroing)` – tato metoda počítá trojúhelníkový tvar matice `A_` a redukční matici s minimálním stupněm. Výsledný trojúhelníkový tvar je uložen v `T`, redukční matice v `U`.

`bool solve(PolMatrix<T>& T, PolMatrix<T>& U, int deg, char type = 'c', typename trait1<T>::t1_type = global<typename trait1<T>::t1_type>::zeroing)` – tato metoda má stejný význam jako předcházející. Rozdíl je v tom, že hledá redukční matici stupně `deg`. Samotný výpočet probíhá pouze v této metodě. Ostatní pouze metody nastavují hodnotu proměnné `Umatrix` a volají tuto metodu.

5.10.4 Příklad výpočtu

Pro polynomiální matici

$$\begin{bmatrix} 7+3s & -2-3s \\ 1-2s & -3 \end{bmatrix} \tag{5.22}$$

nalezneme horní a dolní trojúhelníkový tvar matice. Výpočet provede následující program:

```
#include "tri.h"

using namespace polpack;

int main(int argc, char *argv[]) {
    PolMatrix<double> A(1,2,2), T, U; //vytvoreni pol. matic
    PolMatrix<double>::storage_matrix a0(2,2), a1(2,2);

    a0(0,0) = 7.0; a0(0,1) = -2.0; a0(1,0) = 1.0; a0(1,1) = -3.0;
    a1(0,0) = 3.0; a1(0,1) = -3.0; a1(1,0) = -2.0; a1(1,1) = 0.0;

    A[0] = a0; A[1] = a1; //naplneni matice daty
    cout << A << endl;

    TRIsolver<double> solver;
    solver.setA(A);

    solver.solve(T,U); //nalezeni trojuhelnikove a matice
    cout << "T:" << endl << T<< endl;
    cout << "U:" << endl << U << endl;

    solver.solve(T,'r');
    cout << "T:" << endl << T << endl;
    return 0;
```

```
}
```

A zde je výstup programu:

```
Polynomial matrix - size: 2x2, degree - 1
7 + 3s    -2 - 3s
1 - 2s    -3

T:
Polynomial matrix - size: 2x2, degree - 2
-5.39429                              0
-0.44066 + 0.790149s - 0.82054s^2   -2.25489 - 1.18678s - 0.712069s^2

U:
Polynomial matrix - size: 2x2, degree - 1
-0.805345 + 0.41027s   0.237356 + 0.356034s
-0.121561 + 0.41027s   0.830747 + 0.356034s

T:
Polynomial matrix - size: 2x2, degree - 2
-6.30502                 2.69626 + 1.10676s - 0.671123s^2
0                        -2.39377 - 1.25988s - 0.755929s^2
```

Kapitola 6

Numerické algoritmy pro polynomiální matice

V této kapitole jsou popsány vybrané složitější numerické algoritmy pro polynomiální matice. Konkrétně se jedná o výpočet determinantu, hodnosti a kořenů polynomiální matice. Všechny algoritmy byly použity v knihovně PolPack++.

6.1 Determinant polynomiální matice

6.1.1 Definice determinantu

Nechť $\pi = (j_1, j_2, \ldots, j_n)$ je pořadí. Uspořádaná dvojice (j_i, j_k) je inverze v pořadí π, jestliže $i < k$ a $j_k < j_i$. Jeli číslo p počet všech inverzí v pořadí π, potom

$$\operatorname{sgn} \pi = (-1)^p. \tag{6.1}$$

Potom determinantem čtvercové matice $A(s) = [a_{ik}(s)]$ velikosti n je polynom, který je součtem všech součinů tvaru

$$\operatorname{sgn} \pi a_{1j_1} a_{2j_2} \ldots a_{nj_n}, \tag{6.2}$$

kde sčítáme přes všechna možná pořadí $\pi = (j_1, j_2, \ldots, j_n)$ přirozených čísel $1, 2, \ldots, n$.

6.1.2 Popis algoritmu

Pro výpočet byl použit interpolační algoritmus, který je podrobně popsán v [7]:

Vstup: Čtvercová polynomiální matice $A(s)$ stupně d a velikost n.

Výstup: Polynom $p(s)$ – determinant matice $A(s)$.

Krok 1 Vypočti horní odhad stupně determinantu N:

$$N = \min \left(\sum_{i=1}^{n} \deg_{c_i} P(s), \sum_{i=1}^{n} \deg_{r_i} P(s) \right), \tag{6.3}$$

kde $\deg_{c_i} P(s)$ a $\deg_{r_i} P(s)$ jsou i-té sloupcové a řádkové stupně matice $A(s)$.

Krok 2 S využitím FFT algoritmu proveď výpočet hodnot polynomiální matice v minimálně $N+1$ bodech. Výpočtem získáme množinu konstantních matic $\{Y_k | k = 0, 1, \ldots, N\}$

Krok 3 Vypočti vektor $z = [z_0, z_1, \ldots, z_N]$, kde $z_i = \det(Y_i), i = 0, 1, \ldots, N$.

Krok 4 Proveď výpočet inverzní diskrétní Furierovy transformace na vektoru z. Získáme vektor koeficientů $p = [p_0, p_1, \ldots, p_N]$ determinantu $p(s) = p_0 + p_1 s + \cdots + p_N s^N$.

6.1.3 Implementace

Výpočet determinantu provádí metoda `det` třídy `PolMatrix<T>`. Hlavička metody je:

```
PolMatrix PolMatrix::det(precision tol =
global<precision>::zeroing)
```

Výpočet hodnot polynomiální matice provádí metoda

```
void PolMatrix::values(int n, std::complex<precision>* out
```

Pro výpočet determinantu se volá funkce `T det(T* A, int n)` definovaná v hlavičkovém souboru `linalg.h`. Tato funkce používá pro výpočet determinantu LU rozklad matice. Na konci výpočtu se provádí ještě provádí nulování koeficientů menších než `eps` (metody `zeroing(precision tol)` a `clearing()`).

6.2 Hodnost polynomiální matice

6.2.1 Definice hodnosti

Existuje několik definic hodnosti polynomiální matice, které jsou ekvivalentní:

1. počet lineárně nezávislých sloupců (resp. řádků) polynomiální matice
2. počet nenulových sloupců (resp. řádků) sloupcové (resp. řádkové)Hermitovské formy polynomiální matice.
3. počet invariantních polynomů Smithovy formy polynomiální matice
4. rozdíl mezi počtem sloupců (resp. řádků) a dimenzí pravého (resp. levého) nulového prostoru polynomiální matice
5. číslo

$$\max_{\lambda \in \mathbb{C}} \operatorname{rank} A(\lambda) \tag{6.4}$$

6.2.2 Popis algoritmu

K výpočtu hodnosti je použit interpolační algoritmus, který je podrobně popsán v [5] nebo [6]:

Vstup: Polynomiální matice $A(s)$.

Výstup: Hodnost r.

Krok 1 Vyber číslo $z_0 \in \mathbb{C}$. Nechť $i = 0$ a $r = 0$.

Krok 2 Nechť $r = \max(r, \operatorname{rank} A(z_i))$. Pokud $r = \min(m, n)$, vrať r a skonči algoritmus.

Krok 3 Nechť $i = i + 1$. Jestliže $i > k$, kde k je definováno v (6.5), vrať r a skonči algoritmus.

Krok 4 Vyber z_i rozdílné od $z_0, z_1, \ldots, z_{i-1}$. Pokračuj krokem 2.

Poznámka 1. Polynomiální matice ztrácí svoji hodnost v komplexním bodech, kterých může být maximálně

$$k = \min\left(\sum_{i=1}^{m} \deg \operatorname{row}_i A, \sum_{j=1}^{n} \deg \operatorname{col}_j A\right) \tag{6.5}$$

Pro ověření, že matice nemá plnou hodnost je proto potřeba vypočítat hodnost polynomiální matice v minimálně $k + 1$ bodech.

6.2.3 Implementace

Výpočet hodnosti provádí metoda `rank` třídy `PolMatrix<T>`. Metoda má následující hlavičku:

```
size_type PolMatrix<T>::rank(precision eps =
global<precision>::eps)
```

Metoda provede výpočet hodnot polynomiální matice v $k + 1$ bodech (body z_i) za pomoci metody `PolMatrix<T>::values(std::complex<precision> *val)`, která počítá hodnoty polynomiální matice na jednotkové kružnici komplexní roviny s využitím FFT algoritmu. Tím získáme množinu konstantních matic, u kterých je postupně počítána hodnost. Výpočet hodnosti jednotlivých matic provádí funkce `rank(T1* val,T2 eps)` definovaná v hlavičkovém souboru `linalg.h`. Funkce počítá hodnost jako počet singulárních čísel, které jsou větší než `eps`.

6.3 Kořeny polynomální matice

6.3.1 Definice kořenů

Kořeny polynomiální matice (nebo také nuly polynomiální matice, angl. roots) jsou definovány jako body komplexní roviny, ve kterých dochází ke snížení hodnosti polynomialní matice. Definice hodnosti je uvedena v odstavci 6.2.1. Maximální počet kořenů polynomiální matice je dán vztahem (6.5).

6.3.2 Algoritmus výpočtu kořenů

Pro výpočet kořenů polynomiální matice byly implementovány dva různé algoritmy. První algoritmus počítá kořeny jako kořeny determinantu, druhý algoritmus sestavuje speciální matici (companion matrix), jejíž vlastní čísla jsou právě rovna kořenům polynomiální matice.

Algoritmus 1 - výpočet kořenů pomocí determinantu

Vstup: Polynomiální matice $A(s)$ rozměru $m \times n$.

Výstup: Komplexní vektor kořenů v.

Krok 1 Vypočti $r = \mathrm{rank}(A(s))$ (hodnost polynomiální matice).

Krok 2 Pokud $r = m$ a $m = n$ (matice je čtvercová a má plnou hodnost), vypočti $d = \det(A(s))$ (determinant polynomiální matice). V případě, že $m \neq n$ (obdélníková matice) nebo $m = n$ a $r \neq m$ (čtvercová matice s sníženou hodností), pokračuj krokem 4.

Krok 3 Vypočítej komplexní vektor kořenů v polynomu d. Vrať v a ukonči algoritmus.

Krok 4 Nechť $A_1(s) = X_1 A(s) Y_1$, kde $X_1 \in \mathbb{R}^{r \times m}$ a $Y_1 \in \mathbb{R}^{n \times r}$ jsou náhodné reálné matice.

Krok 5 Nechť $d_1 = \det(A_1)$.

Krok 6 Vypočti vektor kořenů $u = (u_1, \ldots, u_n)$ polynomu d_1.

Krok 7 Nechť $A_2(s) = X_2 A(s) Y_2$, kde $X_2 \in \mathbb{R}^{r \times m}$ a $Y_2 \in \mathbb{R}^{n \times r}$.

Krok 8 Pro $i = 1, \ldots, n$ proveď: vypočti $c_i = A(v_i)$, pokud $\mathrm{abs}(c_i) < tol$, pak v_i je kořen matice $A(s)$. Parametr tol je reálná konstanta blízká nule.

Poznámka 1. Výpočet kořenů polynomu $p(s) = a_0 + a_1 s + \cdots + a_k s^k$ se provede sestavením companion matice K a výpočtem jejích vlastních čísel. Matice K má charakteristický

polynom roven polynomu $p(s)$ proto její vlastní čísla jsou rovna kořenům polynomu $p(s)$. Matice K má následující tvar:

$$K = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ -\frac{a_0}{a_k} & -\frac{a_1}{a_k} & \cdots & -\frac{a_{k-1}}{a_k} \end{pmatrix}. \tag{6.6}$$

Poznámka 2. V kroku 4 a 6 algoritmu se provádí redukce polynomiální matice na čtvercovou matici plné hodnosti. Touto redukcí se do polynomiální matice zanáší další nové kořeny. Proto se redukce provedí dvakrát a jako kořeny vybereme ty, které jsou shodné pro obě redukované matice.

Algoritmus 2 - výpočet kořenů pomocí vlastních čísel companion matice

krok 1 Sestav blokovou companion matici

$$K = \begin{pmatrix} I & 0 & \cdots & 0 \\ 0 & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & I \\ -A_0A_k^{-1} & -A_1A_k^{-1} & \cdots & -A_{k-1}A_k^{-1} \end{pmatrix}, \tag{6.7}$$

kde koeficienty A_i jsou maticové koeficienty polynomiální matice $A(z)$ u i-té mocniny.

krok 2 Vypočti vlastní čísla matice K. Ta jsou rovna kořenům polynomiální matice.

Poznámka 3. Tento algoritmus se dá použít pouze pro čtvercové polynomiální matice.
Poznámka 4. V případě, že maticový koeficient A_k je singulární, je třeba počítat zobecněná vlastní čísla.

6.3.3 Implementace

Výpočet kořenů provádí metoda `roots` třídy `PolMatrix<T>`. Metoda má následují hlavičku:

```
void PolMatrix<T>::roots(ublas::vector<std::complex<precision> &r,
    char method = 'd',
    precisin tol = global<precision>::zeroing)
```

Metoda má tři parametry – `r` je komplexní vektor knihovny Ublas, ve kterém jsou po provedení výpočtu uloženy kořeny polynomiální matice, parametr `method` určuje algoritmus výpočtu kořenů (`method` musí být `'d'` pro výpočet pomocí determinantu resp. `'e'` pro výpočet pomocí vlastních čísel). Význam parametru `tol` je zřejmý z kroku 8 algoritmu výpočtu pomocí determinantu. Výpočet vlastních čísel provádí funkce

```
void eig(ublas::matrix<T, ublas::column_major> M,
        ublas::vector<std::complex<typename trait1<T>::t1_type> >& values)
```

a výpočet zobecněných vlastních čísel funkce

```
void geig(ublas::matrix<T, ublas::column_major> A,
        ublas::matrix<T, ublas::column_major> B,
        ublas::vector<std::complex<T> >& alpha,
        ublas::vector<T>& beta )
```

Obě funkce jsou definovány v souboru `linalg.h`.

Kapitola 7

Numerické experimenty

V této kapitole jsou ukázány rychlosti výpočtu vybraných algoritmů knihovny PolPack++ a jsou srovnány s rychlostí stejných algoritmů komerčního produktu Polynomial Toolbox 3 pro Matlab od firmy Polyx Ltd. Měření rychlosti bylo provedeno na počítači s procesorem AMD Duron s taktovací frekvencí 1,2GHz a pamětí RAM o velikosti 512MB. Procesor byl vybaven pamětí L1 cache o velikosti 128KB, která je rozdělena na půl pro data a instrukce a L2 cache o velikosti 64KB. PolPack++ byl zkompilován překladačem GCC (G++) verze 3.2.0 a spuštěn v operačním systému RedHat Linux 9. Polynomial Toolbox byl spuštěn v Matlabu verze 6.5.0 pod operační systémem Windows XP.

7.1 Násobení polynomiálních matic

Výsledné rychlosti násobení ukazují grafy na obrázcích 7.1, 7.2 a 7.4. Násobeny byly pouze čtvercové jednostranné polynomiální matice.

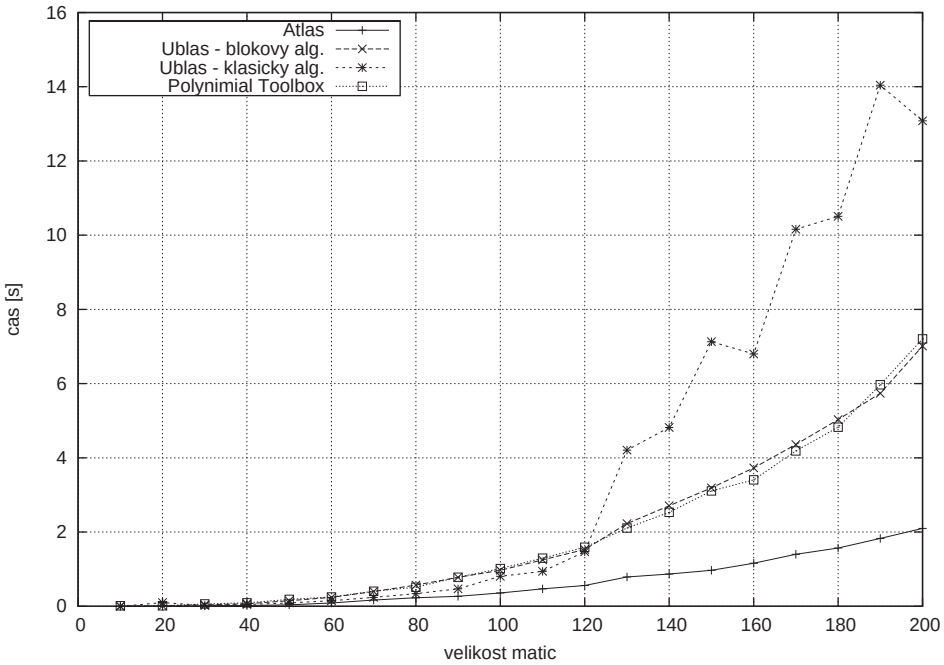
Graf na obrázku 7.1 znázorňuje násobení polynomiální matic stupně 10 a velikost se mění od 10×10 po 200×200. Nejrychlejší ve všech případech je násobí s využitím knihovny Atlas. Násobení s využitím blokového algoritmu knihovny uBLAS je přibližně stejné s rychlostí Polynomial Toolboxu. Pro matice do velikosti přibližně 120×120 je násobení blokovým algoritmem pomalejší než při použití klasického algoritmu uBLASu.

Graf na obrázku 7.2 znázorňuje násobení polynomiální matic o velikosti 10×10 a stupně v intervalu 10-300. Z obrázku je vidět, jak je blokové násobení nevhodné pro matice malých rozměrů.

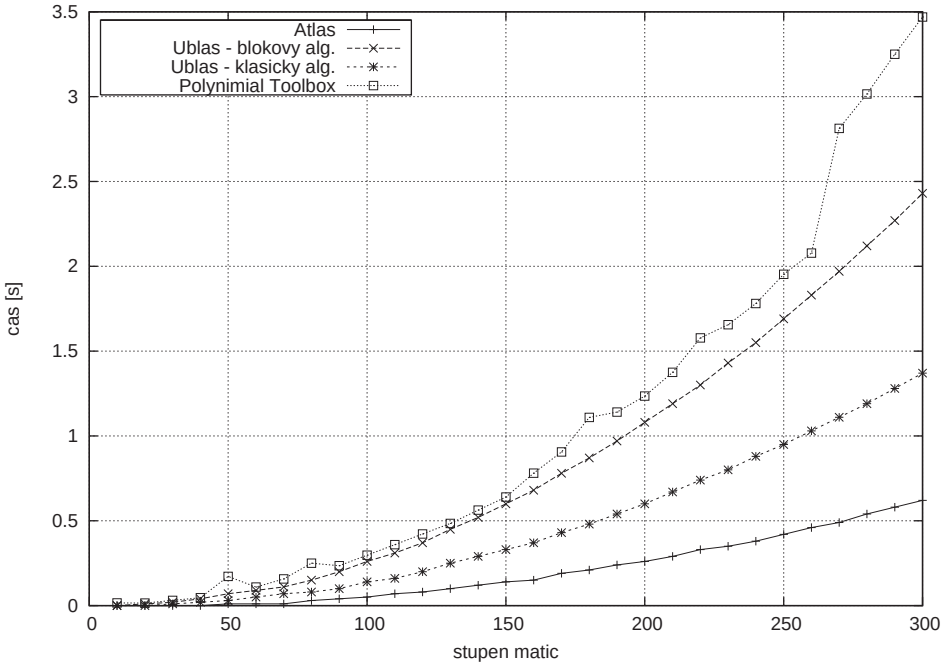
Graf na obrázku 7.2 znázorňuje násobení polynomiální matic stupně pohybujícího se v intervalu 10-100 a velikosti v rozmezí 10×10 stupně 100×100.

7.2 Výpočet determinantu polynomiální matice

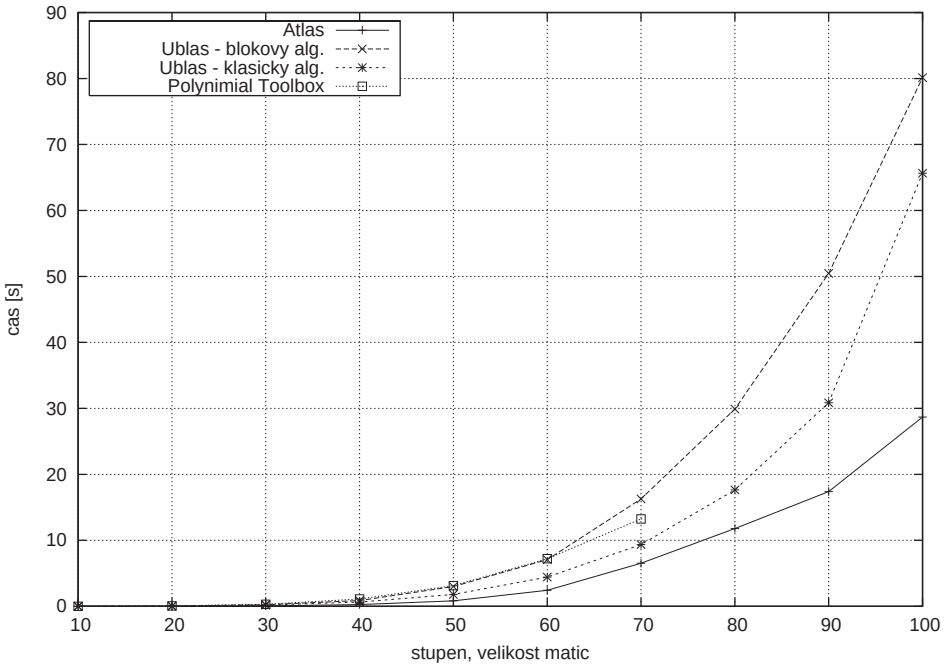
Graf na obrázku 7.2 srovnává výpočetní rychlost determinantu v PolPacku a Polynomial toolboxu. Stupeň i velikost matic se mění v rozsahu 5-50.



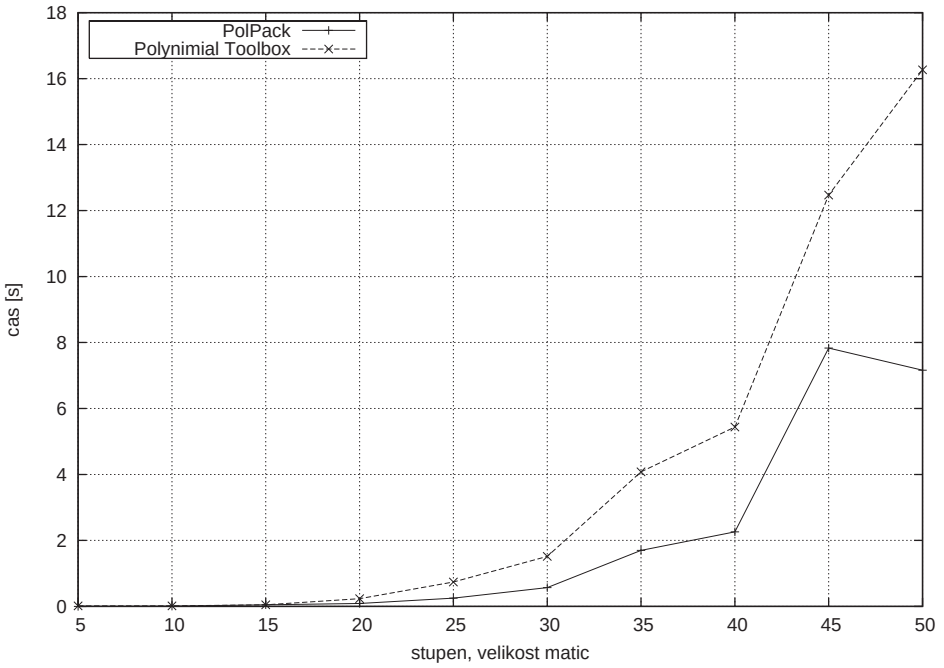
Obrázek 7.1: Srovnání rychlosti násobení polynomiálních matic stupně 10 v PolPacku a Polynomial toolboxu



Obrázek 7.2: Srovnání rychlosti násobení polynomiálních matic velikosti 10×10 v PolPacku a Polynomial toolboxu



Obrázek 7.3: Srovnání rychlosti násobení polynomiálních matic v PolPacku a Polynomial toolboxu



Obrázek 7.4: Srovnání rychlosti výpočtu determinantu polynomiálních matic v PolPacku a Polynomial toolboxu

Kapitola 8

Závěr

V této práci byl popsán vývoj nové numerické knihovny PolPack++ v jazyce C++ pro počítání s polynomiálními maticemi. Knihovna může být šířena pod licencí GPL a je volně dostupná na servru sourceforge.net.

V PolPacku jsou implementovány šablony tříd pro tvorbu základních typů polynomiálních matic, pro základní numerické operace byly přetíženy operátory umožňující přehledný a jednoduchý zápis programu. Knihovna umožňuje řešit základní problémy teorie řízení, jako je řešení diofantických rovnic, provádět testy stability, atd.

Aby bylo dosaženo vysoké výpočetní rychlosti, byly pro návrh použity moderní programovací techniky využívající výhod generického programování pomocí šablon. Knihovna je postavena tak, aby umožnila jednoduché rozšíření o další typy polynomiálních matic.

Numerické experimenty dokazují že, se podařilo se navrhnou knihovnu, jejíž výpočetní rychlost je ve srovnání s obdobnými produkty na vyšší úrovni.

Literatura

[1] Veldhuizen, T. Expression Templates, *C++ Report*, 7(5): 2631, červen 1995.

[2] Veldhuizen, T. Traits: new and useful template technique, *C++ Report*, 7(5): 2631, červenec 1995.

[3] Veldhuizen. T *Techniques for scientific C++*. Technical Report. Indiana university, 2000

[4] Furnish. G. Disambiguated Glommable Expression Templates Reintroduced. *C++ Report*, 6(3):37-46, červen 2000.

[5] Henrion D. *Reliable algorithms for polynomial matrices*. Dizertační práce. Praha: Akademie věd České republiky, ústav teorie informace a automatizace, 1998

[6] Henrion D. and Šebek M. *Numerical Method for Polynomial Matrix Rank Evolution*. LAAS-CNRS Research report No. 97356 Proceedings of the IFAC Conference on System Structure and Control, 1998

[7] Hromcik M., Sebek M. *New algorithm for polynomial matrix determinant based on FFT*. European Control Conference. ECC '99. Karlsruhe 1999

[8] INRIA. *Scilab*. <http://www.inria.scilab.fr>

[9] Waterloo Maple Inc. *Maple*. <http://www.maplesoft.com/>

[10] Wolfram Research Inc. *Mathematica*. <http://www.wolfram.com/>

[11] PolyX Ltd. *Polynomial Toolbox for Matlab*. <http://www.polyx.com>

[12] Paděra M. *Polynomial Matrices in Java*. <http://klokan.sh.cvut.cz/ padera/polynomial/index.html>

[13] Halmo L. *PolPack++*. <http://polpackplusplus.sourceforge.net/>

[14] Anderson E. a kolektiv *LAPACK Users' Guide*. <http://www.netlib.org/lapack/lug/index.html>

[15] GPL licence. <http://www.gnu.org/copyleft/gpl.html>

Příloha A

Příklady z oblasti teorie řízení

A.1 Návrh regulátoru metodou umísťování pólů

Předpokládejme lineární systém popsaný přenosem

$$G(s) = \frac{b(s)}{a(s)} = \frac{1}{s(4+s)} \tag{A.1}$$

Pro tento systém navrhneme spojitý zpětnovazební regulátor, který zabezpečí, že póly zpětnovazebního systému budou umístěny tak, aby $\omega_n = 3$ rad/s a koeficient tlumení $\xi = 0.5$ a pozorovatel stavu obsažený v regulátoru má alespoň dvakrát vyšší úhlovou frekvenci vlastních kmitů a stejné nebo větší tlumení. Kořeny zpětnovazební smyčky potom budou tedy budou

$$s_{1,2} = -3e^{i \arccos 0{,}5} = -1.5 \pm 2.5981i \tag{A.2}$$

Systém je sice druhého řadu a tudíž i pozorovatel by měl být druhého řádu, protože ale informaci o jedné stavové veličině mám bezprostředně k dispozici z měření, můžeme navrhnout pozorovatel pouze 1.řádu

$$s_3 = -6 \tag{A.3}$$

Charakteristický polynom uzavřené smyčky potom bude

$$p(s) = a(s)x(s) + b(s)y(s) = 150 + 85s + 16s^2 + s^3, \tag{A.4}$$

kde $y(s)$ a $x(s)$ jsou čitatel a jmenovatel hledaného přenosu regulátoru. Pro nalezení $y(s)$ a $x(s)$ potřebujeme tedy řešit diofantickou rovnici A.4.

Rovnici A.4 řeší následující program:

```
#include <iostream>
#include <src/polmatrix.h>

using namespace polpack;
```

```
using namespace std;

int main(int argc, char *argv[]) {
    PolMatrix<double> a(2,1,1), b(0,1,1), c(3,1,1), x, y, z;

    a(0) = 0.0; a(1) = 4.0; a(2) = 1.0; b(0) = 1.0;
    c(0) = 54.0; c(1) = 27.0; c(2) = 9.0; c(3) = 1.0;

    cout << "a(s) =" << endl << a << endl;
    cout << "b(s) =" << endl << b << endl;
    cout << "c(s) =" << endl << c << endl;

    //vytvoreni a inicializace solveru pro reseni rovnice
    AXBYCsolver<double> solver(a,b,c);

    //reseni rovnice
    if (solver.solve(x,y)) {
        cout << "nalezeny regulator" << endl << "x(s) =" << x << endl;
        cout << "y(s) =" << endl << y << endl;
    }
    else cout << "reseni nenalezeno" << endl;
}
```

výstup programu bude:

```
a(s) =
Polynomial matrix - size: 1x1, degree - 2
4s + s^2

b(s) =
Constant matrix
1

c(s) =
Polynomial matrix - size: 1x1, degree - 3
54 + 27s + 9s^2 + s^3

nalezeny regulator x(s) =
Polynomial matrix - size: 1x1, degree - 1
5 + 1s

y(s) =
Polynomial matrix - size: 1x1, degree - 1
54 + 7s
```

A.2 Příklad - test robustní stability

Mějme lineární systém, jehož charakteristický polynom obsahuje jeden neznámý reálný parametr q:

$$p(s,q) = s^4 + (6+q)s^3 + 12s^2 + (10+q)s + 3. \tag{A.5}$$

Cílem je nalézt maximální interval $Q = (q_{min}, q_{max})$, kde je polynom $p(s,q)$, $q \in Q$ stabilní.

Polynom $p(s,q)$ přepíšeme do tvaru:

$$p(s,q) = s^4 + 6s^3 + 12s^2 + 10s + 3 + q(s^3 + s) = p_0(s) + qp_1(s). \tag{A.6}$$

Nejprve je třeba ověřit stabilitu nominálního polynomu $P_0(s)$. Pokud je stabilní, tak potom maximální interval stability Q můžeme určit pomocí testu hodnosti Hurwitzovy matice $H(p,q)$ polynomu $p(s,q)$. Hurwitzova matice $H(p,q)$ ztrácí plnou hodnost, jakmile má polynom $p(s,q)$ kořen na mezi stability. Protože matice $H(p,q)$ je polynomiální matice prvního stupně v proměnné q, určíme její kořeny, což jsou právě body, ve kterých ztrácí hodnost.

Dolní hranice intervalu stability Q je potom rovna největšímu zápornému reálnému kořenu matice $H(p,q)$, horní interval je roven minimálnímu kladnému reálnému kořenu.

Řešení pomocí PolPacku:

```
#include <complex>
#include <iostream>
#include <src/polmatrix.h>

using namespace polpack; using namespace std;

int main(int argc, char *argv[]) {
    PolMatrix<double> p0(4,1,1), p1(3,1,1), H(1,4,4);
    PolMatrix<double>::storage_matrix h0, h1;

    p0(0) = 3.0; p0(1) = 10.0; p0(2) = 12.0; p0(3) = 6.0; p0(4) = 1.0;
    p1(0) = 0.0; p1(1) = 1.0; p1(2) = 0.0; p1(3) = 1.0;

    //Hurwitzovy matice polynomu
    p0.hurwitzMatrix(h0);
    p1.hurwitzMatrix(h1,4);

    H[0] = h0; H[1] = h1;

    cout << H << endl; //polynomialni Hurwitzova matice
```

```
    ublas::vector<std::complex<double> > v;
    H.roots(v); //vypocet korenu

    cout << v << endl;

    return 0;
}
```

Výstup programu je následující:

```
Polynomial matrix - size: 4x4, degree - 1
6 + s    10 + s   0        0
1        12       3        0
0        6 + s    10 + s   0
0        1        12       3

[2]((-5.62772,0),(-11.3723,0))
```

To znamená, že matice $H(p,q)$ má reálné kořeny -5.62772 a -11.3723, proto interval stability Q je $(-5.62771, \infty)$.

A.3 Vícerozměrné systémy

Lineární vícerozměrné dynamické systémy (MIMO) můžeme reprezentovat maticovými zlomky. Příklad systému se dvěma vstupy a dvěma výstupy je např. tento maticový zlomek:

$$D^{-1}(s)N(s) = \begin{bmatrix} -1+s+5s^2 & -4s^2 \\ 1-7s+4s^2 & 8-3s+4s^2 \end{bmatrix}^{-1} \begin{bmatrix} -8-7s+3s^2 & -2+4s^2 \\ 4+6s+3s^2 & 6-6s \end{bmatrix} \quad \text{(A.7)}$$

Za pomoci PolPacku ověříme, že se jedná o ryzí maticový zlomek. To se provede srovnáním řádkových stupňů polynomiálních matic $D(s)$ a $N(s)$. To lze však provést pouze v případě, že matice $D(s)$ je řádkově redukovaná. Nakonec ověříme stabilitu systému.

Výpis programu:

```
#include <complex>
#include <iostream>
#include <src/polmatrix.h>

using namespace polpack;
using namespace std;

int main(int argc, char *argv[]) {
    PolMatrix<double> D(2,2,2), N(2,2,2);
    PolMatrix<double>::storage_matrix n0(2,2), n1(2,2), n2(2,2),
```

```
                                        d0(2,2), d1(2,2), d2(2,2);

    d0(0,0) = -1.0; d0(0,1) = 0.0; d0(1,0) = 1.0; d0(1,1) = 8.0;
    d1(0,0) = 1.0; d1(0,1) = 0.0; d1(1,0) = -7.0; d1(1,1) = -3.0;
    d2(0,0) = 5.0; d2(0,1) = -4.0; d2(1,0) = 4.0; d2(1,1) = 4.0;

    n0(0,0) = -8.0; n0(0,1) = -2.0; n0(1,0) = 4.0; n0(1,1) = 6.0;
    n1(0,0) = -7.0; n1(0,1) = 0.0; n1(1,0) = 6.0; n1(1,1) = -6.0;
    n2(0,0) = 3.0; n2(0,1) = 4.0; n2(1,0) = 3.0; n2(1,1) = 0.0;

    N[0] = n0; N[1] = n1; N[2] = n2;
    D[0] = d0; D[1] = d1; D[2] = d2;

    cout << D << endl << N << endl;

    if (D.isRowRed())
        cout << "matice D je radkove redukovana" << endl;

    ublas::vector<std::size_t> r1(2), r2(2);
    r1 = D.rowDeg(); r2 = N.rowDeg();
    cout << r1 - r2 << endl;

    ublas::vector<std::complex<double> > v;
    N.roots(v);
    cout << v << endl;

    return 0;
}
```

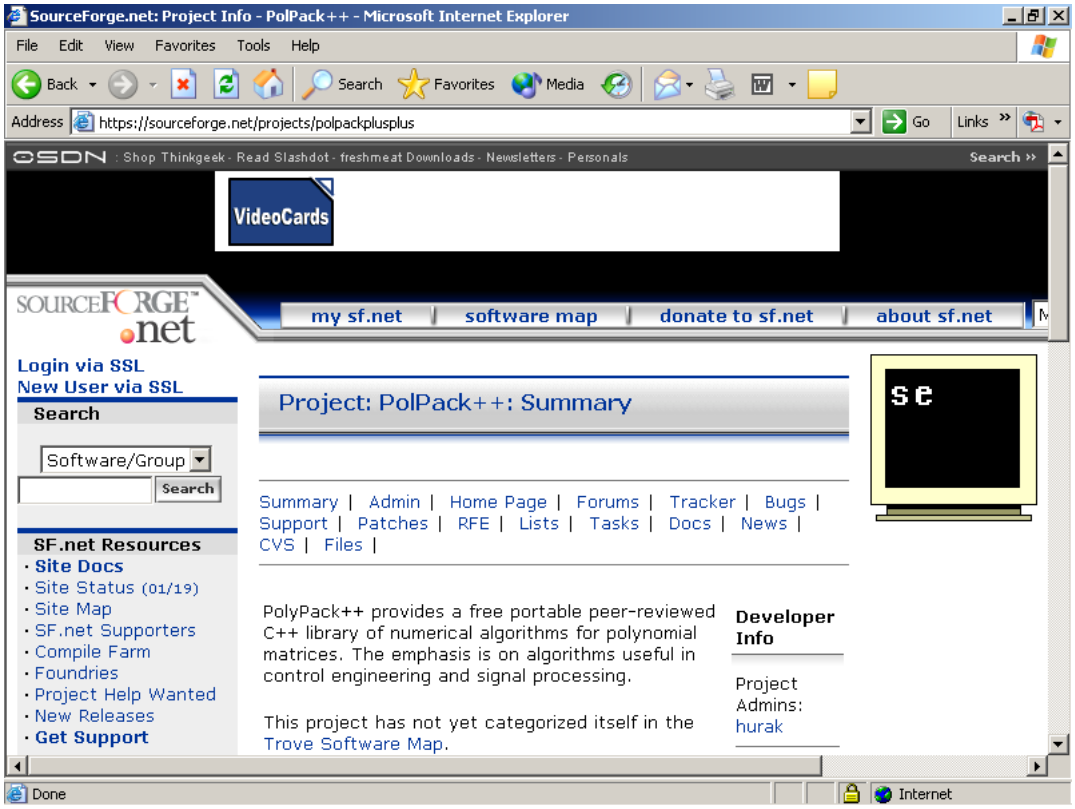
Výstup programu je:

```
Polynomial matrix - size: 2x2, degree - 2
-1 + s + 5s^2   -4s^2
1 - 7s + 4s^2   8 - 3s + 4s^2

Polynomial matrix - size: 2x2, degree - 2
-8 - 7s + 3s^2   -2 + 4s^2
4 + 6s + 3s^2    6 - 6s

matice D je radkove redukovana
[2](0,0)
[4]((-4.3397,0),(-0.847239,0),(0.843467,0.441767),(0.843467,-0.441767))
```

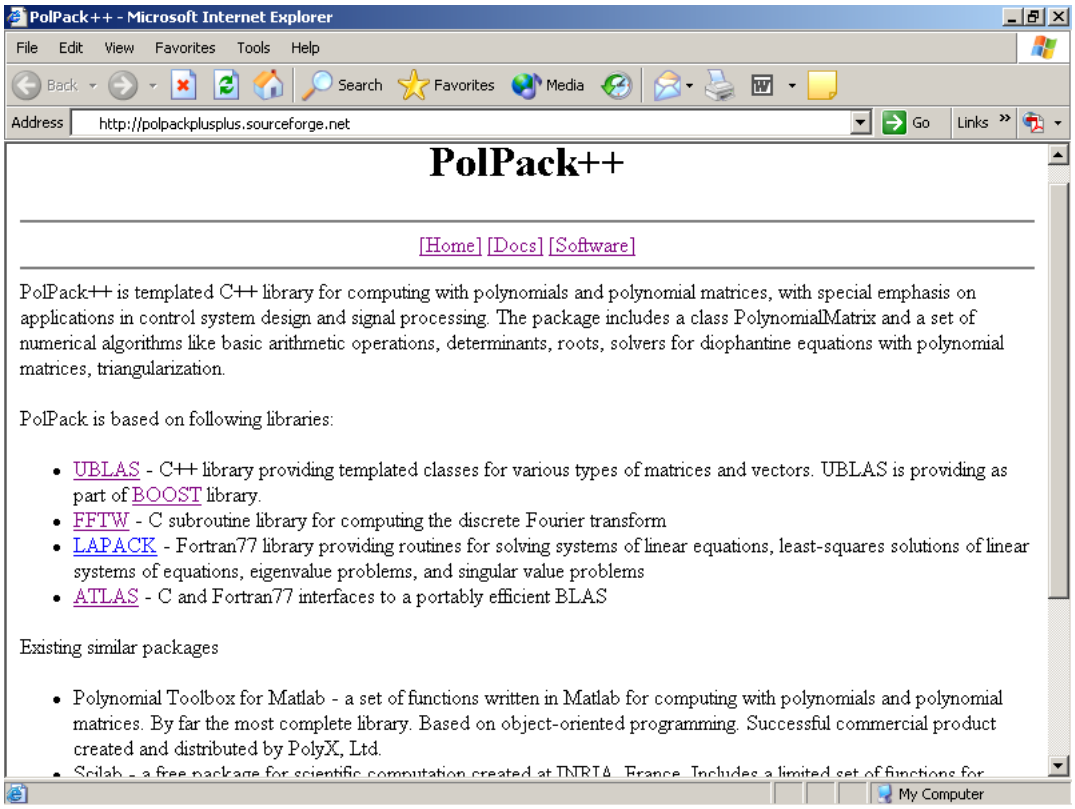
Protože rozdíl řádkových stupňů je větší roven nule, přenosová funkce systému je ryzí. Z vypočítaných kořenů je zřejmé, že systém je nestabilní.

Příloha B

Domovská stránka projektu



Obrázek B.1: Domovská stránka projektu



Obrázek B.2: Domovská stránka projektu