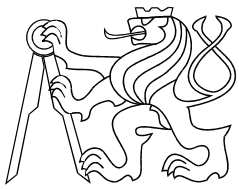
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky



Efektivní výpočty s polynomiálními maticemi v systému *MATHEMATICA* s aplikacemi v řízení

Diplomová práce

2004

Vedoucí práce: Ing. Martin Hromčík
Diplomant: Petr Kujan

Anotace

Práce se zabývá implementací efektivních algoritmů pro počítání s polynomiálními maticemi v programu *MATHEMATICA* se zaměřením na analýzu a návrh řídicích systémů. Přestože je *MATHEMATICA* velmi silný výpočetní nástroj s mnoha funkcemi pro počítání se skalárnímy polynomy nebo maticemi, kde jednotlivé prvky tvoří skalární polynomy, tak takové přímé zadání výpočetní úlohy pomocí standardních funkcí selhává - výpočetní doba pro rozměrnější matice vyšších řádů je neúnosně dlouhá. Proto jsou navrhovány speciální algoritmy. Navíc v systému chybí často používané funkce pro návrh a analýzu řídicích systémů (řešení Diofantické nebo symetrické rovnice a další). Nové funkce jsou v systému *MATHEMATICA* navrhovány s ohledem na možnost zadávat i nečíselné polynomiální matice obsahující symboly (parametry). Námi implementované funkce vycházejí z úspěšného produktu The Polynomial Toolbox for *MATLAB*.

Práce je též motivována skutečností, že autoři systému *MATHEMATICA* podporují návrh řídicích systémů ve své samostatně prodejné nástavbě *CONTROL SYSTEM PROFESSIONAL*. Některé námi vytvořené funkce by v kombinaci s tímto nástrojem mohly být dobře využitelné zejména pro urychlení výpočtů.

Annotation

This diploma thesis deals with the implementation of effective algorithms for calculations using polynomial matrices in the framework of the program *MATHEMATICA* focused on the analysis and design of control systems. Even though *MATHEMATICA* is a very powerful tool that includes many functions for calculation with scalar polynomials or matrices where individual elements form scalar polynomials, such a direct input of a mathematical problem using standard functions fails - the computing time for large matrices of higher orders is too (extremely) long. Therefore, special algorithms have been developed. In addition, the functions frequently used for design and analysis of control systems are missing in the system (solution of the Diophantine or symmetrical equations, etc.). New functions have been proposed in the system *MATHEMATICA* with respect to the possibility to use nonnumeric polynomial matrices containing symbols (parameters). The functions that we have implemented were derived from the successful product The Polynomial Toolbox for *MATLAB*.

The work has also been motivated by the fact that the authors of *MATHEMATICA* system support the design of control systems in their autonomous commercial extension package *CONTROL SYSTEM PROFESSIONAL*. Some of the functions we created could be well applicable in combination with this tool to significantly reduce the computing time.

Obsah

Obsah **3**

1 Úvod **5**
1.1 Přehled možných přístupů při návrhu a analýze řízení 5
1.2 Používané algoritmy pro polynomiální matice 6
1.2.1 Elementární polynomiální operace 6
1.2.2 Symbolické metody 7
1.2.3 Metoda Sylvesterových matic 8
1.2.4 Interpolace 8
1.2.5 Interpolace s diskrétní Fourierovou transformací 9
1.3 Systémy počítačové algebry 11
1.3.1 *MATHEMATICA*, *MAPLE* a *MATLAB* podrobněji 12
1.4 Existující software určený k výpočtům s polynomiálními maticemi . 14
1.4.1 *CONTROL SYSTEM PROFESSIONAL - MATHEMATICA* . . . 14
1.4.2 The Polynomial Toolbox for *MATLAB* 14
1.4.3 Polynomial Matrix Utilities - *MATHEMATICA* 15

2 Cíle práce **16**

3 Popis implementovaných funkcí s testy **17**
3.1 Polynomiální matice a skalární polynom 17
3.1.1 Polynomiální matice – `PM[ ]` 18
3.1.2 Koeficienty polynomiální matice – `PMC[ ]` 18
3.1.3 Matice koeficientů polynomů – `PLC[ ]` 19
3.1.4 Skalární polynom – `P[ ]` 19
3.1.5 Koeficienty skalárního polynomu – `PC[ ]` 19
3.2 Sčítání 20
3.3 Násobení 21
3.4 Maticové násobení 26
3.5 Determinant 31
3.6 Rovnice typu $A.X = B$ 34
3.7 Diofantická rovnice 39
3.8 Symetrická rovnice typu $a^*x + x^*a = b$ 47

3.9 Další užitečné funkce . . . 49
3.9.1 PMRandom[ ] . . . 49
3.9.2 PRandom[ ] . . . 50
3.9.3 StandardPolynomial[ ] . . . 51
3.9.4 Deg[ ] . . . 51
3.9.5 Variable[ ] . . . 51
3.9.6 Size[ ] . . . 51
3.9.7 Star[ ] . . . 51
3.9.8 Předefinované standardní funkce . . . 51
3.10 Globální proměnné . . . 51
3.11 Příklad použití při návrhu řízení . . . 53

4 Implementace **56**
4.1 Struktura programových souborů a instalace . . . 56
4.2 Polynomiální objekty . . . 56
4.2.1 Výhody objektového přístupu . . . 57
4.2.2 Polynomiální matice – PM[ ] . . . 60
4.2.3 Matice koeficientů polynomiální matice – PLC[ ] . . . 61

5 Implementační poznámky **63**
5.1 Seznamy typu PackedArray . . . 63
5.1.1 Reprezentace seznamů dat – List[ ] . . . 64
5.1.2 Výhody reprezentace PackedArray . . . 64
5.1.3 Funkce pracující s PackedArray . . . 65
5.1.4 Polynomiální objekty a PackedArray . . . 67
5.2 Určení typu koeficientů – IntegerQ[ ], RealQ[ ] . . . 67
5.3 Různé způsoby vytváření konstantních matic . . . 68
5.4 Konstrukce Sylvesterovy matice . . . 69
5.5 Funkce NonZeroMin[*list*] . . . 71
5.6 Funkce N[*PolObj*] . . . 73

6 Závěr **74**

7 Publikace **76**

Seznam obrázků **77**

Seznam tabulek **78**

Literatura **79**

A Tištěná dokumentace **81**

B CD s programovými soubory **82**

Kapitola 1

Úvod

Naše práce zasahuje do několika oborů. Jedná se o

- teorii řízení
- numerické metody pro polynomiální matice
- počítačové algebraické systémy - *MATHEMATICA*.

Tyto tři body jsou rozebrány v úvodní kapitole. Z teorie řízení jsou popsány možné přístupy k řešení návrhu a analýzy řízení systémů. V oddíle o numerických metodách jsou uvedeny používané algoritmy pro počítání s polynomiálními maticemi a skalárními polynomy. V kapitole o počítačových algebraických systémech je na začátku zmíněna jejich historie a vývoj. Dále se tato kapitola zaměřuje na porovnání systémů *MATHEMATICA*, *MAPLE* a *MATLAB* a vyjmenování jejich hlavních rysů. Na konci úvodní kapitoly jsou popsány existující programy, které jsou určeny k analýze a návrhu řízení a některé i k výpočtům s polynomiálními maticemi - The Polynomial Toolbox for *MATLAB*, Polynomial Matrix Utilities a *CONTROL SOLUTION*

MATHEMATICA

Dále následuje kapitola, která podrobně popi
včetně jejich algoritmů a časových nároků na výpo
porovnávány a vyhodnocovány.

Na závěr jsou vloženy ukázky programového k
jsou zmíněny některé implementační problémy a

1.1 Přehled možných přístupů při návrhu a analýze řízení

Při návrhu řízení a analýze lineárních dynamických systémů jsou používány v zásadě tři možné přístupy [1].

1. První je *metoda analýzy ve frekvenční oblasti*. Jedná se asi o nejoblíbenější metodu v řadách praktických inženýrů a to především díky její jednoduchosti

a snadnému použití v mnoha problémech řízení, zejména v průmyslu. Tato metoda vychází z analýzy frekvenční odezvy lineárních dynamických systémů. Hlavním matematickým aparátem je teorie funkcí komplexní proměnné, zvláště pak Laplaceova transformace pro časově spojité systémy a Z-transformace pro systémy diskrétní v čase. Systémy jsou popsány svojí přenosovou funkcí, která přesně vyjadřuje vztah mezi vstupem a výstupem. Mezi nevýhody této metody patří problémy s vnitřní stabilitou uzavřené smyčky a omezení na jednorozměrové systémy (SISO).

2. Nedostatky předešlého řeší další přístup, který se zabývá *stavovým popisem* systému. Vychází z vnitřního popisu stavů systému, jejichž znalost se využívá při návrhu řízení. Hlavní aparát, který se zde používá, tvoří diferenciální rovnice, vektorové prostory a teorie matic. Tento nový přístup je aplikovatelný na mnohem větší třídu systémů oproti klasickým metodám, např. mnoharozměrové (MIMO) nebo časově proměnné systémy.

3. Poslední je *polynomiální* nebo též *algebraický přístup*, který je intenzivně rozvíjen od počátku 70. let. Systém je popsán přenosovou funkcí, na kterou v tomto případě není pohlíženo jako na funkci komplexní proměnné, ale jako na algebraický objekt. Metody návrhu řízení a analýzy systémů jsou převedeny na algebraické operace s polynomiálními maticemi, nejčastěji na řešení polynomiálních maticových rovnic. Tímto způsobem je možné elegantně řešit velkou řadu problémů z teorie řízení.

Námi implementované algoritmy spadají do algebraického přístupu řešení problémů.

1.2 Používané algoritmy pro polynomiální matice

V současnosti existuje velké množství procedur i aplikací, které používají numericky stabilní a efektivní algoritmy pro výpočty s konstantními maticemi. V případě polynomiálních matic tomu tak vždy není, a proto jsou navrhovány speciální algoritmy, které jsou popsány v této kapitole.

1.2.1 Elementární polynomiální operace

Elementárními operacemi s polynomiálními maticemi se rozumí následující řádkové a sloupcové úpravy.

Elementární sloupcové úpravy

- Záměna libovolných dvou sloupců.
- Násobení libovolného sloupce nenulovým číslem.
- Násobení sloupce libovolným polynomem a přičtení výsledku k jinému sloupci.

Elementární řádkové úpravy

- Záměna libovolných dvou řádků.
- Násobení libovolného řádku nenulovým číslem.
- Násobení řádku libovolným polynomem a přičtení výsledku k jinému řádku.

Elementární sloupcové operace odpovídají levému přenásobení dané matice maticí unimodulární. Unimodulární matice je polynomiální maticí a její determinant je konstantní číslo. Podobně pro řádkové elementární úpravy se jedná o násobení unimodulární maticí zprava.

Takto popsané elementární operace jsou jednoduché a intuitivní. Hrají důležitou roli v mnoha konstruktivních důkazech z oblasti algebraické teorie a mohou být úspěšně použity v jednoduchých příkladech. Nicméně jejich numerické vlastnosti nejsou příliš uspokojivé. Algoritmy založené na elementárních polynomiálních operacích jsou numericky nestabilní. Stává se, že výpočet v nečekanou chvíli zcela selže. Navíc jsou dost náročné na výpočetní čas. Jejich nasazení na řešení složitějších praktických úloh není z těchto důvodů možné.

Na druhou stranu, elementární polynomiální operace potřebují k řešení malé množství paměti a v případě, že nedojde k selhání, jsou poměrně numericky přesné.

1.2.2 Symbolické metody

Pro symbolické metody je charakteristický způsob reprezentace polynomů, ale i jiných datových struktur. V systému *MATHEMATICA*, ale i v jiných počítačových algebraických systémech, je polynom reprezentován jako symbolický výraz, což je pro počítačové algebraické systémy typické. Výpočty se provádí nad poměrně složitou a do jisté míry i obecnou datovou strukturou. Z toho plyne největší nevýhoda tohoto přístupu a tou je obrovská časová náročnost řešení. Pro méně rozměrné objekty je rychlost výpočtu dostačující, ale pro již o něco málo větší data je výpočet nepoužitelný. V mnoha případech výpočetní čas roste exponenciálně s velikostí zadané úlohy.

Efektivita výpočtu navíc značně závisí na datových typech číselných koeficientů polynomů. Například výpočty s přesnými čísly jsou mnohem náročnější. Paměťové nároky pro přesná čísla jsou značné. Velikost obsazené paměti roste velmi rychle a takřka neomezeně v porovnání s reálnými čísly, u nichž nedochází k tak velkému růstu. Samozřejmě tento problém se týká všech algoritmů.

Na druhou stranu v některých případech může být symbolický přístup mnohem rychlejší. Zde je jednoduchý příklad násobení dvou polynomiálních matic

$$\begin{pmatrix} 1+s^{5000} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ s^2 & 0 \end{pmatrix}.$$

Jeho řešení symbolickými metodami je velmi jednoduché a rychlé, stejně jako kdybychom příklad řešili s tužkou na papíře. Pokud použijeme numerických algoritmů, které jsou popsaný v kapitole 3.4, výpočet bude pomalejší.

Symbolickými metodami tedy budeme rozumět všechny výpočty standardních funkcí nad polynomy, které jsou reprezentovány jako symbolický výraz.

Pokud tedy numerickými metodami myslíme opak symbolických metod, pak tím předpokládáme, že skalární polynomy nejsou reprezentovány ve standardním tvaru symbolického výrazu, ale třeba ve tvaru seznamů koeficientů. Tato data mohou obsahovat i symboly, a přesto s nimi můžeme provádět výpočty. To je dáno schopností některých funkcí počítat i se symbolickými daty.

1.2.3 Metoda Sylvesterových matic

Jedná se o velmi často používanou metodu při výpočtech s polynomiálními maticemi a skalárními polynomy, neboť její použití je názorné a přímé. Vychází z převodu polynomiálních matic na konstantní Sylvesterovy matice. Vstupní polynomiální matici $A(s) = A_0 + A_1 s + A_2 s^2 + ... + A_n s^n$ snadno přepíšeme na Sylvesterovu matici

$$S_A = \begin{bmatrix} A_0 & & & 0 & \\ A_1 & A_0 & & & \\ \vdots & A_1 & \ddots & & \\ A_n & \vdots & & A_0 & \dots \\ & A_n & & A_1 & \\ & & \ddots & \vdots & \\ 0 & & & A_n & \\ & \vdots & & & \end{bmatrix}.$$

Úloha dále nejčastěji vede na konstantní soustavy rovnic, které je možné v systému *MATHEMATICA* řešit i se symbolickými daty.

Nevýhodou této metody je poněkud větší náročnost na velikost použité paměti. Další problém je v odhadu výsledného stupně hledané polynomiální matice. Při jeho řešení se nejčastěji postupuje tak, že se určí horní a dolní mez stupně polynomiální matice a potom, např. binárním půlením, se hledá minimální řešení.

Metoda Sylvesterových matic je v této práci použita při skalárním a maticovém násobení a při řešení Diofantické a symetrické rovnice.

1.2.4 Interpolace

Interpolace je užitečný nástroj pro práci s polynomiálními maticemi, především pak v kombinaci s DFT. Používá se při výpočtech různých funkcí, jako je determinat, násobení, skalární mocnina nebo při řešení polynomiálních maticových rovnic i v dalších úlohách řízení.

Algoritmus 1.2.1 (Interpolace)

Nechť požadovaná operace nebo funkce je f a je aplikována na polynomiální matici (nebo skalár) $P(s)$. Výsledek je potom $X(s) = f(P(s))$. Algoritmus lze popsat ve třech krocích.

Vstup: $P(s) = P_0 + P_1 s + \ldots + P_d s^d$

Výstup: $X(s) = f(P(s)) = X_0 + X_1 s + \ldots + X_N s^N$

Krok 1. *Substituce*: *Vstupní matice je vyhodnocena v $N+1$ komplexních bodech $\{s_i \in \mathbb{C} \mid i = 0, 1, \ldots, N\}$, kde N je očekávaný stupeň výsledné matice $X(s)$. Výsledkem je seznam konstantních matic $\{Y_i \mid Y_i = P(s_i),\ i = 0, 1, \ldots, N\}$.*

Krok 2. *Výpočet konstantních matic:* *Funkce f je aplikována jednotlivě na všechny konstantní matice Y_i. Tak dostaneme konstantní matice $\{Z_i \mid Z_i = f(Y_i),\ i = 0, 1, \ldots, N\}$, které jsou partikulárním řešením hledané polynomiální matice $X(s)$.*

Krok 3. *Interpolace:* *Koeficienty výsledné polynomiální matice $X(s) = X_0 + X_1 s + \ldots + X_N s^N$ získáme z konstantních matic Z_i řešením soustavy rovnic*

$$[X_0, X_1, \ldots, X_N] \,.\, V = [Z_0, Z_1, \ldots, Z_N],$$

kde V je Vandermodeho matice. *Jestliže $X(s)$ je skalární polynom, pak*

$$V = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ s_0 & s_1 & \ldots & s_N \\ s_0^2 & s_1^2 & \ldots & s_N^2 \\ \vdots & \vdots & \vdots & \vdots \\ s_0^N & s_1^N & \ldots & s_N^N \end{bmatrix}.$$

Pro polynomiální matici je V bloková Vandermodeho matice.

V popsaném algoritmu narážíme na problém odhadu počtu a umístění interpolačních bodů. Ukazuje se, že proveditelnost výpočtu silně závisí na poloze bodů. Při nevhodné volbě polohy interpolačních bodů je matice V velmi špatně podmíněna a získané výsledky jsou nepoužitelné. Nejčastěji se interpolační body volí ve stejné vzdálenosti od reálné a imaginární osy v komplexní rovině. Lze je také generovat náhodně. Další způsob řešení spočívá ve využití Fourierových bodů, viz. metoda interpolace s DFT. Tento způsob se jeví jako nejlepší.

1.2.5 Interpolace s diskrétní Fourierovou transformací

Předchozí interpolační postup lze výrazně zlepšit použitím DFT, respektive FFT.

Definice DFT

Definice 1.2.1 (Přímá DFT) *Nechť $\mathbf{p} = [p_0, p_1, \ldots, p_N]$ je vektor komplexních čísel. Potom jeho přímá DFT je dána vektorem $\mathbf{y} = [y_0, y_1, \ldots, y_N]$, pro jehož prvky platí*

$$y_k = \sum_{i=0}^{N} p_i e^{-j\frac{2\pi k}{N+1} i}\,. \tag{1.1}$$

Vektor $\mathbf{y}$ *nazýváme obraz vektoru* $\mathbf{p}$.

Definice 1.2.2 (Inverzní DFT) *Nechť* $\mathbf{y} = [y_0, y_1, \ldots, y_N]$ *je vektor komplexních čísel. Potom jeho inverzní DFT je dána vektorem* $\mathbf{p} = [p_0, p_1, \ldots, p_N]$, *pro jehož prvky platí*

$$p_i = \frac{1}{1+N} \sum_{k=0}^{N} y_k e^{j\frac{2\pi i}{N+1}k}. \tag{1.2}$$

Jestliže $\mathbf{y}$ *je obraz* $\mathbf{p}$, *pak vzorec (1.2) vrací originální vektor* $\mathbf{p}$.

Nyní základní definice přímé a inverzní DFT rozšíříme na seznam matic.

Definice 1.2.3 (Přímá DFT seznamu matic) *Nechť* $\mathcal{P} = [P_0, P_1, \ldots, P_N]$ *je seznam komplexních matic* P_i *rozměru* $m \times n$. *Potom jeho přímá DFT je dána seznamem matic* $\mathcal{Y} = [Y_0, Y_1, \ldots, Y_N]$, *pro jehož prvky platí*

$$Y_k = \sum_{i=0}^{N} P_i e^{-j\frac{2\pi k}{N+1}i}. \tag{1.3}$$

Seznam matic $\mathcal{Y}$ *nazýváme obraz seznamu matic* $\mathcal{P}$.

Definice 1.2.4 (Inverzní DFT seznamu matic) *Nechť* $\mathcal{Y} = [Y_0, Y_1, \ldots, Y_N]$ *je seznam komplexních matic* Y_k *rozměru* $m \times n$. *Potom jeho inverzní DFT je dána seznamem matic* $\mathcal{P} = [P_0, P_1, \ldots, P_N]$, *pro jehož prvky platí*

$$P_i = \frac{1}{1+N} \sum_{k=0}^{N} Y_k e^{j\frac{2\pi i}{N+1}k}. \tag{1.4}$$

Jestliže $\mathcal{Y}$ *je obraz* $\mathcal{P}$, *pak vzorec (1.4) vrací originální vektor* $\mathcal{P}$.

DFT a Interpolace

Definice (1.1) a (1.2) přímé a inverzní DFT mají zásadní význam ve vylepšení interpolačního algoritmu 1.2.1. Nechť body s_k značí $e^{-j\frac{2\pi k}{N+1}}$ a $p(s) = p_0 + p_1\, s + \ldots + p_N\, s^N$ je polynom. Pak přímá DFT vektoru koeficientů $\mathbf{p} = [p_0, p_1, \ldots, p_N]$ je vektor hodnot $\mathbf{y} = [y_0, y_1, \ldots, y_N]$, kde $y_k = \sum_{i=0}^{N} p_i\, s_k^i$ je polynom $p(s)$ vyhodnocený v bodě s_k. Tomuto postupu odpovídá první krok interpolačního algoritmu (substituce). Na druhou stranu, pro vektor $\mathbf{z} = [z_1, z_2, \ldots, z_N]$ vztah (1.2) definuje vektor koeficientů $\mathbf{x} = [x_0, x_1, \ldots, x_N]$ polynom $x(s)$, jehož hodnoty v bodech s_i se rovnají z_i, kde $i = 1, 2, \ldots, N$. Proto třetí krok interpolačního algoritmu se speciální volbou Fourierových bodů může být nahrazen inverzní DFT.

Stejné výsledky dostaneme i pro polynomiální matici $P(s)$, stačí pouze na místo vektorů $\mathbf{p}, \mathbf{y}$ uvažovat množiny konstantních matic $\mathcal{P}, \mathcal{Y}$ a místo vztahů (1.1), (1.2) použít (1.3) a (1.4).

Následující algoritmus je modifikací algoritmu 1.2.1 s využitím DFT, resp. FFT.

Algoritmus 1.2.2 (Interpolace s FFT)

Vstup: $P(s) = P_0 + P_1 s + \ldots + P_d s^d$

Výstup: $X(s) = f(P(s)) = X_0 + X_1 s + \ldots + X_N s^N$

Krok 1. ***Přímá DFT:*** *Přímá DFT (1.3) je aplikována na seznam matic, který tvoří koeficienty $P(s)$, $\mathcal{P} = [P_0, P_1, \ldots, P_d, O_{d+1}, \ldots O_N]$. Navíc seznam matic $\mathcal{P}$ je doplněn nulovými maticemi stejného rozměru do počtu $N + 1$. Takto obdržíme seznam matic $\{Y_i \mid Y_i = P(s_i), i = 0, 1, \ldots, N\}$, který odpovídá hodnotám $P(s)$ ve Fourierových bodech.*

Krok 2. ***Výpočet konstantních matic:*** *Funkce f je aplikována jednotlivě na všechny konstantní matice Y_i. Tím dostaneme konstantní matice $\{Z_i \mid Z_i = f(Y_i), i = 0, 1, \ldots, N\}$, které jsou partikulárním řešením hledané polynomiální matice $X(s)$.*

Krok 3. ***Zpětná DFT:*** *Výsledné koeficienty $\{X_0, X_1, \ldots, X_N\}$ hledané polynomiální matice $X(s)$ získáme ze seznamu konstantních matic Z_i užitím inversní DFT podle (1.4).*

1.3 Systémy počítačové algebry

Počátky vývoje systémů počítačové algebry jsou spjaty s nástupem programovacích jazyků FORTRAN a především LISP. Jako první program, který byl schopen provádět symbolické manipulace s matematickými výrazy byl v roce 1961 program SAIN (Symbolic Automatic INtegration). Dále ho následovaly programy FORMAC, MATHLAB, REDUCE, SCRATCHPAD, muMATH a další.

Na počátku osmdesátých let dochází s nástupem pracovních stanic založených na mikroprocesorech k velkému rozvoji počítačových algebraických systémů. Jako první [illegible] of Waterloo objevil systém *MAPLE*, který byl

1.3.1 *MATHEMATICA*, *MAPLE* a *MATLAB* podrobněji

Všechny tři programy disponují vlastnostmi, které jsou dnes charakteristické pro všechny systémy tohoto druhu[1]. Jedná se především o manipulaci s výrazy, které obsahují neznámé. Dokáží provádět redukci nebo expanzi neurčitých výrazů, jako jsou polynomy, racionální funkce nebo mocninné řady. Dále jsou schopny provádět výpočty s libovolně velkými čísly, mají přesnou racionální aritmetiku a při výpočtech v pohyblivé řádové čárce lze použít libovolné přesnosti. Mají v sobě zabudované velké množství běžných matematických funkcí a konstant. Dokáží řešit řadu problémů z matematické analýzy, jako je výpočet limity a derivace funkce, řešení určitých nebo neurčitých integrálů a řešení rovnic. Samozřejmě tyto funkce jsou doplněny důležitými numerickými algoritmy pro hledání kořenů rovnic, řešení soustav lineárních rovnic, výpočty vlastních čísel a vektorů matic nebo numerickou integraci.

Velmi důležitou vlastností je schopnost rozšiřitelnosti. Ta je dána mocným programovacím jazykem, který umožňuje doplnit systémy o nové algoritmy a funkce.

- *MATHEMATICA*
  Jedná se o jeden z nejrozšířenějších systémů svého druhu. Úzce spojuje schopnosti symbolických a numerický výpočtů, grafický výstup a programovací jazyk. Pracovní dokumenty jsou ve formě sešitů (obrázek 1.1), které si uchovávají veškeré výsledky (vstupy, výstupy a grafiku). Uživatelské prostředí je velice příjemné, propracované a dále rozšiřitelné. Lze ho snadno použít k dokumentaci nebo např. k interaktivní prezentaci výsledků [2, 4].

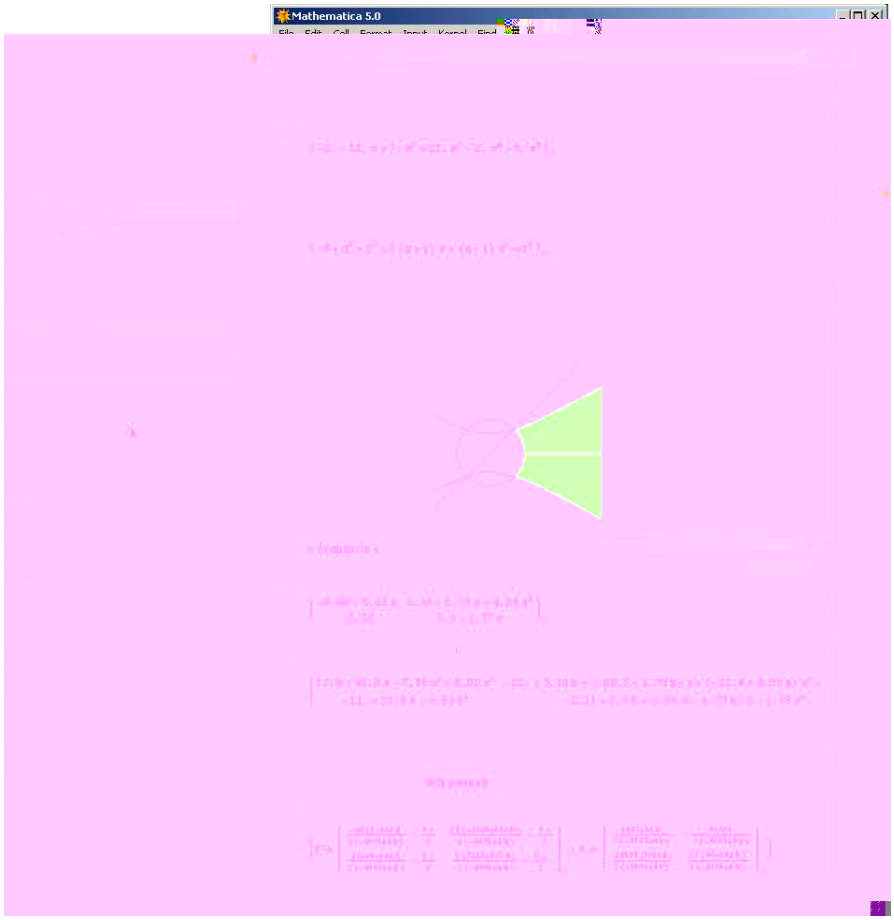
  Dále se vyznačuje schopností pracovat s jinými aplikacemi, jako Excel, Word nebo složité databázové systémy. Pomocí rozhraní MathLink je možno při výpočtech využívat vlastní naprogramované funkce nebo celé aplikace. Lze ho snadno rozšířit o oborově zaměřené nástavby, jako *CONTROL SYSTEM PROFESSIONAL* a mnoho dalších[2].

  Nyní je dostupná nejnovější verze systému *MATHEMATICA 5*, která obsahuje řadu nových numerických a symbolických algoritmů. Jedná se především o optimalizaci v oblasti numerické lineární algebry (BLAS a LAPACK technologie) [5]. Výpočty lineárních rovnic nebo maticového násobení byly oproti předchozím verzím několikanásobně urychleny. Rychlost výpočtů je srovnatelná a v některých případech i větší než v systému *MATLAB* (na stejném HW).

- *MAPLE*
  Je v současnosti výhodný zejména pro své výukové a prezentační možnosti. Podobně jako *MATHEMATICA* je schopen si uchovávat informace o výpočtech

[1] Matlab má trochu odlišné postavení. Jedná se především o program určený k numerickým výpočtům, avšak jeho nástavba Symbolic Math Toolbox ho zařazuje mezi algebraické počítačové systémy, neboť Symbolic Math Toolbox je tvořen výpočetním jádrem systému Maple.

[2] Advanced Numerical Methods, Signals and Systems, Digital Image Processing, Experimental Data Analyst, Finance Essentials, Fuzzy Logic, Neural Networks, ...
Informace o produktech lze získat na adrese `http://www.wolfram.com/`



Obrázek 1.1: *MATHEMATICA* Notebook.

ve formě grafických zápisů a ty pak kdykoliv zopakovat. Podrobnější informace lze nalézt v [6, 9].

- *MATLAB*
  Jak již bylo řečeno, je zaměřen především na numerické výpočty v pohyblivé řádové čárce s pevně danou přesností. Základním datovým prvkem je dvourozměrné pole. Od verze 5 lze navíc deklarovat vícerozměrná pole. Dále byl systém obohacen o možnost objektového programování, rychlejší a propracovanější grafický výstup, nové vizualizační funkce a rozšířen o další základní matematické funkce [10, 11].

  MATLAB byl od počátku navrhován tak, aby mohl snadno přistupovat k matematickým knihovnám vyvinutým v projektech LINPACK a EISPACK. Původně byl určen pro OS UNIX a tato okolnost je dodnes patrná i v OS Windows, kde jsou příkazy, na rozdíl od předchozích systémů, zadávány do

příkazové řádky. Na druhou stranu *MATLAB* mnohem lépe podporuje práci programátorů při vývoji nových funkcí a to svým editorem kódu s podporou ladění a krokování vytvářených funkcí.

Najdeme ho skoro na všech technicky zaměřených univerzitách, kde slouží k výuce matematiky a dalších technických předmětů. V praxi je využíván jako vysoce efektivní nástroj pro výzkum (modelování, simulace) a analýzu dat.

MATLAB je rozšiřitelný o rozsáhlé sady dalších funkcí, které řeší určité okruhy problémů v daném oboru a nazývají se Toolboxy. Velmi oblíbenou samostatnou nástavbou je Simulink. Dokáže řešit soustavy nelineárních diferenciálních rovnic v grafickém prostředí připomínající zapojení na analogovém počítači. Umožňuje graficky sledovat výstupy v libovolném místě zapojení.

1.4 Existující software určený k výpočtům s polynomiálními maticemi

V této kapitole je stručně popsán existující software, resp. rozšiřující nástavby pro systém *MATHEMATICA* a *MATLAB*, které se zabývají výpočty s polynomiálními maticemi, analýzou systémů a návrhem řízení. Námi naprogramované funkce budou často srovnávány právě s těmito systémy.

1.4.1 *Control System Professional - Mathematica*

Jedná se o soubor funkcí rozšiřující systém *MATHEMATICA*. Zabývá se řešením problémů z teorie systémů a řízení. Zahrnuje dva možné přístupy řešení spojitých a diskrétních systémů a to klasický a moderní.

Obsahuje nové objekty a řadu funkcí pracujících s nimi. Samozřejmě nechybí podrobná nápověda v elektronické i tištěné podobě [12].

1.4.2 The Polynomial Toolbox for *Matlab*

The Polynomial Toolbox (dále jen PTX) představuje ucelenou sadu programů zaměřených na výpočty s polynomy a polynomiálními maticemi s širokým spektrem aplikací. Nejvíce se uplatňuje v oblastech teorie systémů a signálů. Zde používané algoritmy výpočtů jsou založeny na pokročilých polynomiálních metodách, které jsou vyvíjeny v rámci národních i evropských výzkumných projektů [13]. V posledních letech byla řada algoritmů publikována v prestižních časopisech a na mezinárodních konferencích [15].

PTX obsahuje řadu funkcí pro řešení lineárních maticových polynomiálních rovnic, které jsou založeny na metodách Sylvesterových matic. Metody využívající FFT se uplatňují při výpočtech hodnosti matic, determinantu a dalších operací. Dále zahrnuje několik metod pro výpočty spektrální faktorizace a maticových zlomků přenosové funkce MIMO systémů. Najdeme tu také řadu funkcí pro testování a řešení problémů z oblasti robustní stability. Navíc PTX obsahuje funkce pro konverzi

objektů mezi nástavbami Control System Toolbox, Symbolic Math Toolbox a SIMULINK.

1.4.3 Polynomial Matrix Utilities - *MATHEMATICA*

Autorem je A. Pascoletti z univerzity v Udine. Jedná se o jedinou sadu funkcí v systému *MATHEMATICA*, která provádí některé pokročilé algebraické výpočty s polynomiálními maticemi [17].

Polynomiální matice jsou reprezentovány maticemi, které obsahují polynomy jako symbolické výrazy a lze je zadávat v proměnné z ale i z^{-1}. Koeficienty polynomiální matice mohou obsahovat i parametry.

Poskytované funkce počítají Smithovu, Hermitovu a McMillanovu formu polynomiálních matic. Tyto funkce vracejí kromě výsledné formy i unimodulární matici transformace. Další funkce jsou určeny pro výpočet levého a pravého největšího společného dělitele a nejmenšího společného násobku, zbytku po dělení a řešení Diofantické rovnice.

Většina algoritmů je založena na sloupcových a řádkových úpravách vstupní polynomiální matice nebo se provádějí časově náročné symbolické výpočty nad polynomiálními výrazy. V kapitolách o elementárních polynomiálních operacích 1.2.1 a symbolických metodách výpočtu 1.2.2 jsou podrobněji popsány nevýhody těchto metod a v kapitole 3.7 o řešení Diofantické rovnice jsou také experimentálními výpočty potvrzeny.

Kapitola 2

Cíle práce

Systém *MATHEMATICA* patří ve své oblasti k nejrozšířenějším a k nejvýkonnějším programům. Mezi jeho největší přednosti patří úzké spojení symbolických a rychlých numerických výpočtů. Obsahuje velké množství matematických funkcí a velmi mocný programovací jazyk. Přesto poskytované nástroje pro výpočty s polynomiálními maticemi jsou z pohledu analýzy a návrhu řízení velmi omezené. V systému chybí celá řada často používaných funkcí a v případě aplikace standardních rutin na rozměrnější polynomy nebo polynomiální matice je výpočet neefektivní a časově velmi náročný, takřka nepoužitelný.

Práce je též motivována skutečností, že autoři systému *MATHEMATICA* podporují návrh řídicích systémů ve své samostatně prodejné nástavbě *CONTROL SYSTEM PROFESSIONAL*. Některé námi vytvořené funkce by v kombinaci s tímto nástrojem mohly být dobře využitelné zejména pro urychlení některých výpočtů.

Toto jsou hlavní důvody, proč jsme se rozhodli v systému *MATHEMATICA* implementovat efektivní algoritmy pro počítání s polynomiálními maticemi. Jako hlavní vzor jsme použili The Polynomial Toolbox for *MATLAB*, který je častým používáním v inženýrské praxi velmi dobře prověřen.

Dlouhodobý zájem o polynomiální metody v řízení na Elektrotechnické fakultě ČVUT - Katedře řídící techniky je dalším důvodem naší volby. Jsou zde intenzivně studovány a rozvíjeny nové postupy analýzy a návrhu řízení polynomiálními metodami, které byly prezentovány na mnoha prestižních konferencích v zahraničí. Mezi hlavní osobnosti v daném oboru patří Prof. Ing. Vladimír Kučera, DrSc. a Doc. Ing. Michael Šebek, DrSc. Velmi dobře jsou v této problematice rozvinuty trvalé spolupráce se zahraničními univerzitami a firmami. Jmenovitě se jedná o LAAS-CNRS Toulouse (F), University of Twente (NL), Politechnica di Milano (I) a Aristotle University of Thessaloniki (GR). V loňském roce se nám podařilo navázat spolupráci s Aristotelovou univerzitou v Soluni, kde jsme společně zahájili práci na implementaci nových funkcí pro polynomiální matice s dvěma proměnnými.

Souběžně s touto prací se pracuje na implementaci polynomiálních metod v jazycích C++, JAVA a systému *MAPLE*.

Kapitola 3

Popis implementovaných funkcí s testy

Naší snahou nebylo vytvořit pouhou sadu funkcí v systému *MATHEMATICA*, ale celé prostředí, které uživateli nabízí snadnou a intuitivní práci při výpočtech s polynomiálními maticemi a skalárními polynomy. Proto byly předefinovány standardní funkce a vytvořeny nové polynomiální objekty. K dispozici je také elektronický návod a několik palet s předpřipravenými funkcemi.

Inicializace a nastavení globálních proměnných se provádí standardním způsobem pro zavádění programových balíčků.

```
In[1] :=   << Polynomial`
```

V jednotlivých kapitolách byly prováděny experimentální výpočty porovnávající rychlosti výpočtů jednotlivých metod v systému *MATHEMATICA 5* [4]. Rychlost výpočtů byla také porovnávána se systémem *MATLAB 6.5* [10] s nástavbou The Polynomial Toolbox 3.0 [13]. Funkce `DESolve` byla jako jediná porovnávána s balíčkem Polynomial Matrix Utilities [17]. Všechny testy byly prováděny na počítači PC Duron 750MHz, 256MB RAM s Windows 2000. Naměřené časy jsou v sekundách (značka ** zastupuje výpočet delší než 1000 sekund).

3.1 Polynomiální matice a skalární polynom

Polynomiální matice mohou být zadávány v různých tvarech pomocí funkcí `PM[`*`pm, var`*`]`, `PMC[`*`pmc, var`*`]` a `PLC[`*`plc, var`*`]`. Všechny tyto funkce mají stejný výstup a to právě objekt polynomiální matice, funkci `PolyMat[`*`pmc, var, deg`*`]`, která má přesně definované argumenty. Jestliže je dána polynomiální matice $A(s) = A_0 + A_1 s + \ldots + A_{d_A} s^{d_A}$, pak objekt polynomiální matice je ve tvaru `PolyMat[{`$\mathtt{A_0}, \mathtt{A_1}, \ldots, \mathtt{A_{d_A}}$`}, s, `$\mathtt{d_A}$`]`.

Objekt polynomiální matice je zobrazován v libovolném tvaru pomocí interní funkce `Format[ ]`, kterou řídí globální proměnná `$Format`. Grafický výstup ve tvaru

polynomiální matice zajistí hodnota `"Nice"`, která je nastavena jako standardní. Další možné tvary výstupního formátu pro polynomiální matice a skalární polynomy jsou ukázány v kapitole 3.10, která popisuje všechny globální proměnné.

Podobně se pracuje i se skalárními polynomy. Skalární polynom se zadává funkcemi `P[p, var]` nebo `PC[pc, var]`. Jejich výstup je převeden na objekt `Poly[pc, var, deg]`.

3.1.1 Polynomiální matice – PM[ ]

Zadání ve tvaru polynomiální matice, tj. matice (seznam, `List`), ve které jsou na jednotlivých pozicích příslušné polynomy jako symbolické výrazy (standardní formát pro zadávání polynomů). Funkce je ve tvaru `PM[pm,var]`. První argument *pm* je vstupní polynomiální matice. Argument *var* určuje proměnnou polynomiální matice *pm*.

```
In[2] :=  PM[{{k + s^2, 1 - s}, {s^ , -1 + . s}}, s]
Out[2] =
```

$$\begin{pmatrix} k+s^2 & 1-s \\ s^3 & -1+0.5\,s \end{pmatrix}_s$$

Na jednotlivých pozicích matice *pm* mohou být zadávány i objekty skalárních polynomů.

```
In[3] :=  PM[{{ P[k + s^2, s], 1 - s}, {s^ , PC[{-1, . }]}}, s]
Out[3] =
```

$$\begin{pmatrix} k+s^2 & 1-s \\ s^3 & -1+0.5\,s \end{pmatrix}_s$$

Pokud je polynomiální matice bez symbolů, nemusí se uvádět druhý parametr proměnné *var*. Proměnná se vyhledá automaticky.

```
In[4] :=  PM[{{ 1+s  2+s , - }, {s^2, 1-s^2}}]
Out[4] =
```

$$\begin{pmatrix} 2+3\,s+s^2 & -3 \\ s^2 & -1+s^2 \end{pmatrix}_s$$

3.1.2 Koeficienty polynomiální matice – PMC[ ]

Zadání ve tvaru maticových koeficientů polynomiální matice, tj. seznam matic jednotlivých mocnin polynomiální matice. Funkce je ve tvaru `PMC[pmc, var]`.

```
In[5] :=  PMC[{{{k,1},{0,-1}}, {{0,-1},{0,0. }}, {{1,0}, {0,0}},
                {{0,0}, {1,0}}}, s]
Out[5] =
```

$$\begin{pmatrix} k+s^2 & 1-s \\ s^3 & -1+0.5\,s \end{pmatrix}_s$$

Pokud není zadán druhý parametr *var*, automaticky se dosadí hodnota z globální proměnné `$Variable`. Vstupní sesnam dat *pmc* nesmí obsahovat symbol shodný s proměnnou *var*, jinak je generováno chybové hlášení.

```
In[6] :=  PMC[{{{s,1},{0,-1}}, {{0,-1},{0,0. }}, {{1,0}, {0,0}},
              {{0,0}, {1,0}}}]          $Variable === s
Out[6] =

        General::errcoe: Input coefficients contain variable s.
        General::notpolyformat: Input is not in Poly format.
```

3.1.3 Matice koeficientů polynomů – `PLC[ ]`

Zadání ve tvaru seznamu koeficientů polynomů na jednotlivých pozicích polynomiální matice. Funkce je ve tvaru `PLC[`*`plc`*`,`*`var`*`]`.

```
In[7] :=  PLC[{{{k, 0, 1}, {1, -1}}, {{0,0,1}, {-1,. }}}, s]
Out[7] =
```

$$\begin{pmatrix} k+s^2 & 1-s \\ s^3 & -1+0.5\,s \end{pmatrix}_s$$

Při zadávání dat *plc* a *var* platí stejné vlastnosti jako u funkce `PMC`.

3.1.4 Skalární polynom – `P[ ]`

Zadání ve tvaru polynomu (symbolický výraz).

```
In[8] :=  P[k + 2s + s^2, s]
Out[8] =
```

$$\left(k+2\,s+s^2\right)_s$$

3.1.5 Koeficienty skalárního polynomu – `PC[ ]`

Zadání polynomu pomocí seznamu koeficientů jednotlivých mocnin.

```
In[9] :=  PC[{k,2,1}]
Out[9] =
```

$$\left(k+2\,s+s^2\right)_s$$

Další možné tvary zadání a různá chybová hlášení jsou rozebrány v dokumentaci.

3.2 Sčítání

Funkce `A+B` sečte polynomiální objekty `A, B` a provede nulování reálných čísel podle globální proměnné `$Zeroing`.

Sčítat lze pouze polynomiální matice stejných rozměrů, jinak je generováno chybové hlášení a na výstup je vrácen vstup, který se rovná funkci `Plus[ ]` s argumenty polynomiálních matic různých rozměrů. Matice, které lze sečíst jsou sečteny.

```
In[10] :=  Options[PMRandom] = {Data -  {Integer, {-9, 9}}};
           A = PMRandom[1, 2];
           B = PMRandom[1, 2,  ];
           C = PMRandom[1, 2];
           A + B + C
Out[10] =

        General::plusdim: Matrices not of the same dimensions.
        General::plusdim: Matrices not of the same dimensions.
```

$$\begin{pmatrix} 8-16\,s & -4+5\,s \\ 3-7\,s & -4-9\,s \end{pmatrix}_s + \begin{pmatrix} 8-8\,s & 3-8\,s & -6+s \\ 2-4\,s & 5-6\,s & -3+5\,s \end{pmatrix}_s$$

Tento výstup je v souladu s výstupy v systému *MATHEMATICA*. Použitá funkce `PMRandom[deg, size, opt]` generuje náhodnou polynomiální matici stupně *deg* a rozměru *size* s čísly podle *opt*. Funkce `PMRandom` je rozebrána podrobně v kapitole 3.9.1.

Dále je možné *sčítat polynomiální matice se skalárními polynomy*. Zadaný skalární polynom je přičten k jednotlivým prvkům polynomiální matice.

```
In[11] :=  PM[{{s^2,- },{1+s,-2s}}] + P[1+s]      PolyMat + Poly
Out[11] =
```

$$\begin{pmatrix} 1+s+s^2 & -2+s \\ 2+2\,s & 1-s \end{pmatrix}_s$$

K polynomiálním objektům lze také *přičíst číslo nebo symbol*, který ale nesmí být shodný s proměnnou daného polynomiálního objektu, jinak se sčítání neprovede a vrátí se vstup s chybovým hlášením.

```
In[12] :=  PM[{{s^2,- },{1+s,-2s}}] + k       PolyMat + Symbol
Out[12] =
```

$$\begin{pmatrix} k+s^2 & -3+k \\ 1+k+s & k-2\,s \end{pmatrix}_s$$

Někdy je užitečné symbol nepřičítat k danému objektu. To nastává v případech, kdy symbol zastupuje polynom nebo polynomiální matici. Pak není možné daný symbol přičíst jako konstantu. Proto je potřeba nejprve provést příkaz `IsPoly[symbol]`, kterým deklarujeme, že daný symbol není konstanta, ale polynomiální objekt. Tato konstrukce je rozebrána v obecném použití v [16].

```
In[13] :=   IsPoly[k]
            term = PM[{{1 + s, s^2}, {0, -s}}] + k
Out[13] =
```

$$k + \begin{pmatrix} s^2 & -3 \\ 1+s & -2\,s \end{pmatrix}_s$$

Nyní lze snadno použít příkaz dosazení `/.` a dopočítat daný výraz.

```
In[14] :=   term /. k -  P[1-s]
Out[14] =
```

$$\begin{pmatrix} 1-s+s^2 & -2-s \\ 2 & 1-3\,s \end{pmatrix}_s$$

Použité metody výpočtu

Provádí se pouhé sčítání konstantních matic nebo čísel u odpovídajících mocnin polynomiálních matic nebo skalárních polynomů.

Nechť $A(s) = A_0 + A_1 s + ... + A_{d_A} s^{d_A}$ a $B(s) = B_0 + B_1 s + ... + B_{d_B} s^{d_B}$ jsou vstupní polynomiální matice, pak platí

$$A(s) + B(s) =$$

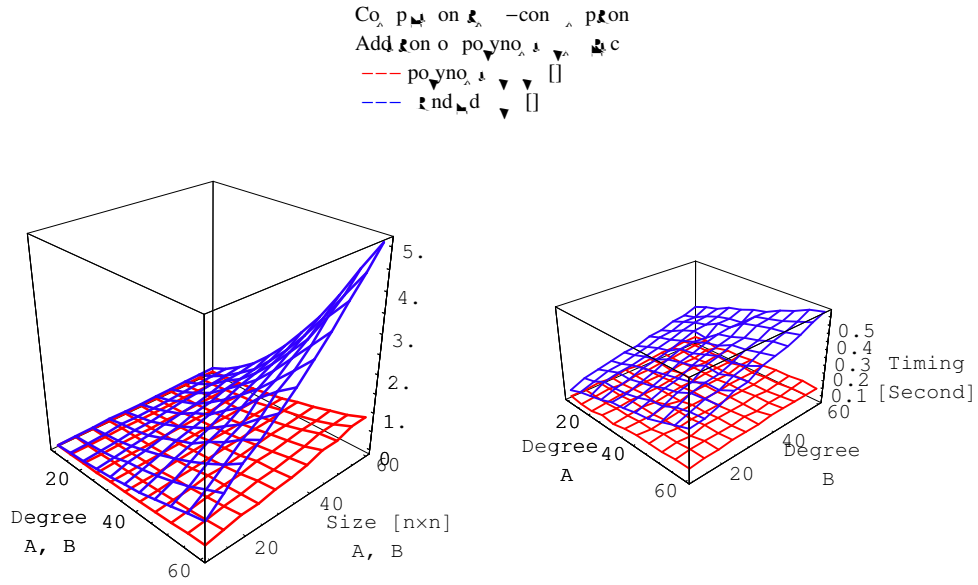
$$\begin{cases} (A_0 + B_0) + (A_1 + B_1) + \ldots + (A_{d_A} + B_{d_B}) s^{d_A} & \text{, pro } d_A = d_B \\ (A_0 + B_0) + \ldots + (A_{d_B} + B_{d_B}) s^{d_B} + A_{d_B+1} s^{d_B+1} + \ldots + A_{d_A} s^{d_A} & \text{, pro } d_A > d_B \\ (A_0 + B_0) + \ldots + (A_{d_A} + B_{d_A}) s^{d_A} + B_{d_A+1} s^{d_A+1} + \ldots + B_{d_B} s^{d_B} & \text{, pro } d_A < d_B \end{cases}$$

Porovnání časové náročnosti výpočtu

Již při jednoduché operaci sčítání rozměrných polynomiálních matic velkých stupňů je časová úspora výpočtu oproti standardním prostředkům systému *MATHEMATICA* značná. A to proto, že námi definované sčítání pracuje s vytvořenými polynomiálními objekty, u kterých jednotlivé koeficienty polynomiální matice jsou reprezentovány konstantními maticemi. Samotné sčítání polynomiálních matic se tudíž omezuje pouze na součty příslušných konstantních matic koeficientů, které není časově náročné, na rozdíl od standardní funkce pro sčítání, kde na každý prvek polynomiální matice je pohlíženo jako na obecný výraz. Součty jednotlivých prvků jsou tedy prováděny nad obecnými výrazy. Tato operace je sice naprosto bezproblémová, ale ve své obecnosti není při výpočtu tak efektivní jako součet konstantních matic. Rychlost výpočtu `A+B` pro různé rozměry a stupně reálných polynomiálních matic je graficky porovnána na obrázku 3.1.

3.3 Násobení

V systému *MATHEMATICA* je nutné rozlišovat mezi maticovým a „standardním“ násobením (po prvcích). Maticové násobení se provádí funkcí `Dot[a, b]`, resp. `a.b`.



Obrázek 3.1: Porovnání časové náročnosti výpočtu funkce `Plus[A,B]` - metoda Polynomial a Standard.

Funkci `Dot` je také nutné používat k násobení polynomiálních matic, které bude podrobně rozebráno v další kapitole. Násobení mezi výrazy (čísla, symboly, seznamy, ...) se provádí příkazem `Times[a, b]`, resp. `a*b` nebo pouze `a⊔b`. Tento příkaz se volá pro násobení mezi skalárními polynomy nebo při násobení polynomiální matice konstantou, symbolem nebo skalárním polynomem. Násobení se aplikuje na všechny prvky polynomiální matice.

Po vykonané operaci násobení se provede nulování reálných čísel podle globální proměnné `$Zeroing`. V případě nulových nejvyšších koeficientů se upravuje i stupeň výsledné polynomiální matice. Použití funkce `$Zeroing` je vysvětleno v kapitole 3.10.

```
In[15] :=   P[1+s]   P[1-s]        Poly   Poly
Out[15] =
```

$$(1 - s^2)_s$$

```
In[16] :=   P[1+2s]   PM[{{1 + s, s^2}, {0, -s}}]     Poly   PolyMat
Out[16] =
```

$$\begin{pmatrix} 1+3\,s+2\,s^2 & s^2+2\,s^3 \\ 0 & -s-2\,s^2 \end{pmatrix}_s$$

```
In[17] :=   Clear[k];
            k   PM[{{1 + s, s^2}, {0, -s}}]     Symbol   PolyMat
Out[17] =
```

$$\begin{pmatrix} k + k\,s & k\,s^2 \\ 0 & -k\,s \end{pmatrix}_s$$

V tomto případě symbol může navíc zastupovat skalární polynom nebo číslo a pak je nutné výpočet neprovádět. Postupuje se podobně jako v minulém příkladě u sčítání.

```
In[18] :=   IsPoly[k];
            term = k   PM[{{1 + s, s^2}, {0, -s}}]
Out[18] =
```

$$k \begin{pmatrix} 1 + s & s^2 \\ 0 & -s \end{pmatrix}_s$$

```
In[19] :=   term /. k -  P[1 + 2s]
Out[19] =
```

$$\begin{pmatrix} 1 + 3\,s + 2\,s^2 & s^2 + 2\,s^3 \\ 0 & -s - 2\,s^2 \end{pmatrix}_s$$

Použité metody výpočtu

Násobení polynomiální matice (nebo skalárního polynomu) číslem (nebo symbolem) je pouhé přenásobení všech koeficientů daným číslem (nebo symbolem).

Početně zajímavější je případ násobení mezi skalárními polynomy nebo skalárním polynomem a polynomiální maticí. Podrobněji rozebereme pouze případ násobení mezi skalárními polynomy, neboť násobení mezi skalárním polynomem a polynomiální maticí není nic jiného než násobení daného skalárního polynomu mezi všemi prvky (skalárními polynomy) dané polynomiální matice.

Implementovány jsou následující dvě metody výpočtu.

Sylvesterova metoda je založena na převodu prvního skalárního polynomu na Sylvesterovu matici a následném maticovém násobení s koeficienty druhého polynomu. Řešením je sloupcový vektor, který tvoří koeficienty hledaného polynomu. Obecné použití Sylvesterovy metody je popsáno v kapitole 1.2.3.

Vstup: *Skalární polynomy*

$$a(s) = a_0 + a_1 s + \ldots + a_{d_a} s^{d_a},$$

$$b(s) = b_0 + b_1 s + \ldots + b_{d_b} s^{d_b}.$$

Výstup: *Skalární polynom*

$$c(s) = a(s) * b(s) = c_0 + c_1 s + \ldots + c_{(d_a+d_b)} s^{(d_a+d_b)}.$$

Krok 1. *Konstrukce Sylvesterovy matice.*

$$S_a = \begin{bmatrix} a_0 & & & 0 \\ a_1 & a_0 & & \\ \vdots & a_1 & \ddots & \\ a_{d_a} & \vdots & & a_0 \\ & a_{d_a} & & a_1 \\ & & \ddots & \vdots \\ 0 & & & a_{d_a} \end{bmatrix}_{[1+d_a+d_b \times 1+d_b]} , \quad S_b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{d_b} \end{bmatrix}_{[1+d_b \times 1]} \tag{3.1}$$

Krok 2. *Maticové násobení $S_{ab} = S_a \,.\, S_b$. Prvky vektoru S_{ab} jsou koeficienty hledaného polynomu $c(s)$.*

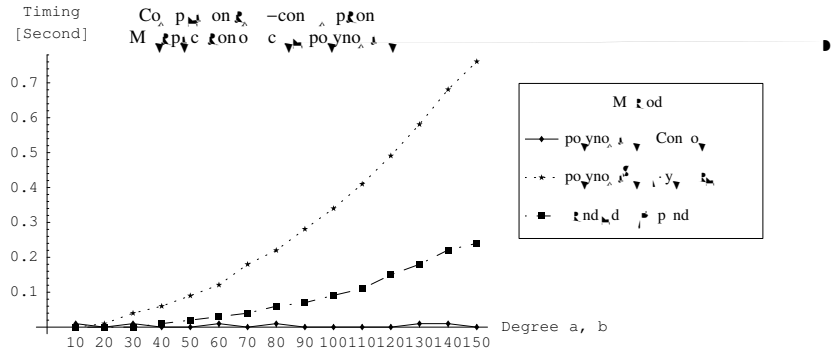
Metoda konvoluce provádí konvoluci koeficientů vstupních skalárních polynomů. Příkaz vykonává pouze jediná funkce `ListConvolve[ ]`, která pracuje velmi efektivně.

Výpočet konvoluce pro polynomy $a(s) = a_0 + a_1 s$ a $b(s) = b_0 + b_1 s + b_2 s^2$ dává okamžitě výsledek

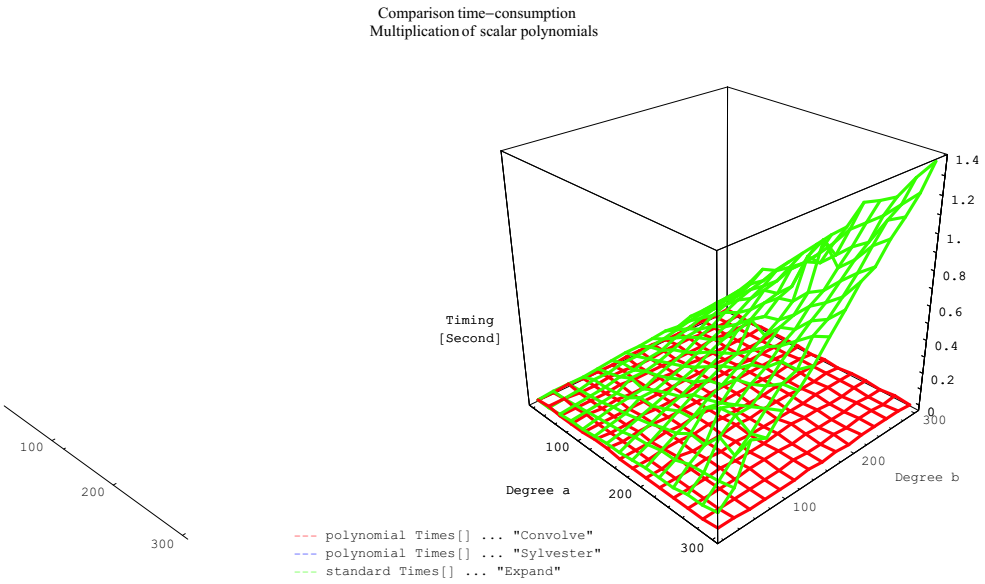
$$c(s) = a_0\, b_0 \; + \; (a_0\, b_1 + a_1\, b_0)\, s \; + \; (a_0\, b_2 + a_1\, b_1)\, s^2 \; + \; (a_1\, b_2)\, s^3 \,.$$

Porovnání časové náročnosti výpočtu

Při výpočtu násobení dvou skalárních polynomů se ukazuje jako nejrychlejší metoda konvoluce. Druhá nejrychlejší metoda je standardní symbolické násobení a následná expanze výrazu. Metoda Sylvesterových matic je v tomto případě nejpomalejší. Porovnání rychlostí výpočtů jednotlivých metod je znázorněno na obrázku 3.2.



Obrázek 3.2: Porovnání časové náročnosti výpočtu funkce `Times[a,b]` - metoda Convolve, Sylvester a Expand.


Comparison time−consumption
Multiplication of scalar polynomials
Timing
[Second]
1.4
1.2
1.
0.8
0.6
0.4
0.2
0
100
200
300
Degree a
100
200
300
Degree b
--- polynomial Times[] ... "Convolve"
--- polynomial Times[] ... "Sylvester"
--- standard Times[] ... "Expand"

Ze všech grafů je patrné, že nejrychlejší výpočetní časy má metoda konvoluce. Je to dáno tím, že se pro výpočet volá jediná interní funkce `ListConvolve[pca, pcb, {1,-1},0]`, která počítá konvoluci dvou vektorů velmi efektivně. Celkový počet násobení a sčítání dvou čísel je při výpočtu $1 + d_a + d_b + 2d_a d_b$, na rozdíl od metody Sylvesterových matic, která je sice v principu stejná, ale navíc se provádí zbytečné násobení nulou v matici S_a (3.1). Počet všech násobení a sčítání se rovná $(1 + 2d_b)(1 + d_a + d_b)$. Nároky na paměť počítače jsou také mnohem větší, neboť se násobí rozměrná matice S_a vektorem S_b. Překvapivě rychle počítá i standardní funkce s expanzí výrazu `Expand[a b]`.

V tabulce 3.1 jsou porovnány časy výpočtů funkce `Times[a,b]` a funkce `a. b` v Polynomiálním Toolboxu pro *MATLAB*.

Tabulka 3.1: Porovnání časová náročnosti výpočtu násobení skalárních polynomů $a * b$.

Stupeň polynomů `a, b`	*MATHEMATICA* `"Convolve"`	*PTX MATLAB*
500	0.	0.02
1500	0.01	0.04
3000	0.03	0.12
5000	0.05	0.60

3.4 Maticové násobení

Maticové násobení se v systému *MATHEMATICA* provádí příkazem `Dot[A, B]`, resp. `A.B`. Funkce `Dot` se používá i při násobení polynomiálních matic.

Příkaz `A.B` vrací maticově vynásobené polynomiální matice `A, B` a navíc provede nulování reálných čísel podle globální proměnné `$Zeroing`. Samozřejmě počet sloupců matice `A` musí být roven počtu řádků matice `B`.

Syntaxe zadávání příkazů je intuitivní a v souladu se systémem *MATHEMATICA*.

```
In[20] :=   A = PMRandom[1, 2];        stupeň 1, rozměr [2x2]
```

$$\texttt{A.PM[}\begin{pmatrix} 1+\texttt{s} & \texttt{s}^2 \\ -\ \texttt{s} & \texttt{k} \end{pmatrix}\texttt{,s]}$$

```
Out[20] =
```

$$\begin{pmatrix} 5 - 48\,s - 13\,s^2 & 9\,k + k\,s + 15\,s^2 - 24\,s^3 \\ 1 - 43\,s - 9\,s^2 & 7\,k + 3\,s^2 - 27\,s^3 \end{pmatrix}_s$$

Pokud nesouhlasí rozměry matic, je generováno varovné hlášení a zadaný vstup je vrácen na výstup. V případě, že se násobí několik matic najednou, provede se

násobení všech matic, u kterých je to rozměrově možné. Ostatní jsou vráceny na výstup a je generováno varovné hlášení.

```
In[21] :=   a = PMRandom[1, 2,  ];      stupeň 1, rozměr [2x ]
            b = PMRandom[1,  , 2];
            c = PMRandom[1,  , 2];
Out[21] =

        PolDot::errsize: Matrices of inconsistent dimensions.
```

$$\begin{pmatrix} -67-123\,s-21\,s^2 & -96-59\,s+42\,s^2 \\ -24-45\,s-29\,s^2 & -21-28\,s-s^2 \end{pmatrix}_s . \begin{pmatrix} 2+2\,s & -9+4\,s \\ -5-3\,s & 5-3\,s \\ -9+s & -1+7\,s \end{pmatrix}_s$$

Také je možné použít při násobení symbol, který zastupuje polynomiální matici. V tomto případě není nutné symbol předem deklarovat funkcí `IsPoly[]`, neboť zde nemůže dojít k záměně s parametrem nebo skalárním polynomem, viz. předešlá kapitola.

```
In[22] :=   c = PMRandom[1, 2, 2];
            d = PMRandom[0, 2, 2];
            term = a.x.c.d
Out[22] =
```

$$\begin{pmatrix} 5-9\,s & 9 & 1+2\,s \\ 7 & -8+9\,s & 1+6\,s \end{pmatrix}_s .x. \begin{pmatrix} 40+44\,s & 8-4\,s \\ -9-95\,s-90\,s^2 & -5+13\,s+30\,s^2 \end{pmatrix}_s$$

Nyní lze použít příkaz dosazení a dopočítat celý výraz.

```
In[23] :=   term /. x -  b
Out[23] =
```

$$\begin{pmatrix} 3206+7273\,s-3338\,s^2-3917\,s^3 & 814+965\,s-50\,s^2-489\,s^3 \\ -3439+1038\,s+14665\,s^2-6072\,s^3 & -915+1702\,s-267\,s^2+936\,s^3 \end{pmatrix}_s$$

Použité metody výpočtu

Implementovány jsou následující čtyři metody výpočtu.

Sylvesterova metoda je shodná s postupem jako u násobení skalárních polynomů, ale místo s čísly se zde pracuje s konstantními maticemi. Metoda je založena na převodu polynomiálních matic na konstantní Syvesterovy matice a jejich následném vynásobení. Obecné použití Sylvesterovy metody bylo popsáno v kapitole 1.2.3.

Vstup: *Polynomiální matice*

$$A(s)_{[p\times r]} = A_0 + A_1 s + \ldots + A_{d_A} s^{d_A},$$

$$B(s)_{[r\times q]} = B_0 + B_1 s + \ldots + B_{d_B} s^{d_B}.$$

Výstup: *Skalární polynom*

$$C(s)_{[p\times q]} = A(s)\,.\,B(s) = C_0 + C_1 s + \ldots + C_{(d_A+d_B)} s^{(d_A+d_B)}$$

Krok 1. *Konstrukce Sylvesterových matic.*

$$S_A = \begin{bmatrix} A_0 & & & 0 \\ A_1 & A_0 & & \\ \vdots & A_1 & \ddots & \\ A_{d_A} & \vdots & & A_0 \\ & A_{d_A} & & A_1 \\ & & \ddots & \vdots \\ 0 & & & A_{d_A} \end{bmatrix}_{[(1+d_A+d_B)p\times\ (1+d_B)r]} , S_B = \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_{d_B} \end{bmatrix}_{[(1+d_B)r\times\ q]} \tag{3.2}$$

Krok 2. *Maticové násobení* $S_{AB} = S_A\,.\,S_B = [C_0, C_1, \ldots, C_{d_A+d_B}]$. *Submatice* C_i *rozměru* $[p \times q]$ *matice* S_{AB} *jsou maticové koeficienty hledané polynomiální matice* $C(s)$.

Metoda přímého výpočtu koeficientů počítá v cyklu podle následujícího vztahu

$$C(s) = A(s)\,.\,B(s) = \sum_{k=0}^{d_A+d_B} \Big(\sum_{i=max(0,\,k-d_B)}^{min(k,\,d_A)} A_i.B_{k-i} \Big) * s^k.$$

Tento vztah odpovídá násobení submatic v maticích S_A a S_B ze Sylvesterovy metody. Hlavní výhoda této metody je, že se zde neprovádí zbytečné násobení nulovou maticí. Implementace této metody se ukazuje jako výpočetně nejrychlejší.

Metoda konvoluce provádí konvoluci koeficientů matic $A(s), B(s)$. Metoda je implementována pomocí jediné funkce

`ListConvolve[{` $A_0, \ldots, A_{d_A}$ `}, {` $B_0, \ldots, B_{d_B}$`}, {1, -1},`
`ZeroMatrix[Size[A], Dot, Plus, 1] ].`

Tato metoda se při násobení skalárních polynomů osvědčila a byla vyhodnocena jako nejrychlejší. V této maticové modifikaci již nepracuje tak efektivně a je nejhorší.

Metoda FFT je efektivní numerická metoda výpočtu. Matice $A(s), B(s)$ nemohou obsahovat parametry (omezení funkce `Fourier`). V kapitole 1.2.5 je algoritmus odvozen. Postup řešení lze popsat ve třech krocích.

Krok 1. *Výpočet diskrétní Fourierovy transformace seznamu matic*

$$\{A_0, A_1, \ldots, A_{d_A}, O_{d_A+1}, \ldots, O_{d_A+d_B}\},$$

$$\{B_0, B_1, \ldots, B_{d_B}, O_{d_B+1}, \ldots, O_{d_A+d_B}\},$$

které odpovídají koeficientům polynomiálních matic $A(s)$ a $B(s)$ doplněným o příslušný počet nulových matic. Tím obdržíme nové seznamy matic $\{X_k, \mid k = 0, 1, \ldots, d_A + d_B\}$ *a* $\{Y_{k}maticT\ m\ obdr\ \ me\ nov$

```
Out[25] =
```

$$\{2.57\ \texttt{Second}, \begin{pmatrix} 0 & 1+s^{5000} \\ s^{200} & 0 \end{pmatrix}_s, \}$$

Nyní převedeme polynomiální objekty na symbolické výrazy a provedeme standardní násobení `Dot[ ]` s následnou expanzí.

```
In[25] :=   syma = StandardPolynomial[a]
            symb = StandardPolynomial[b]
            Map[Expand, syma.symb, {2}] // Timing
```

```
Out[25] =
```

$$\{\{1+s^{5000}, 0\}, \{0, 1\}\}$$

```
Out[26] =
```

$$\{\{0, 1\}, \{s^{200}, 0\}\}$$

```
Out[27] =
```

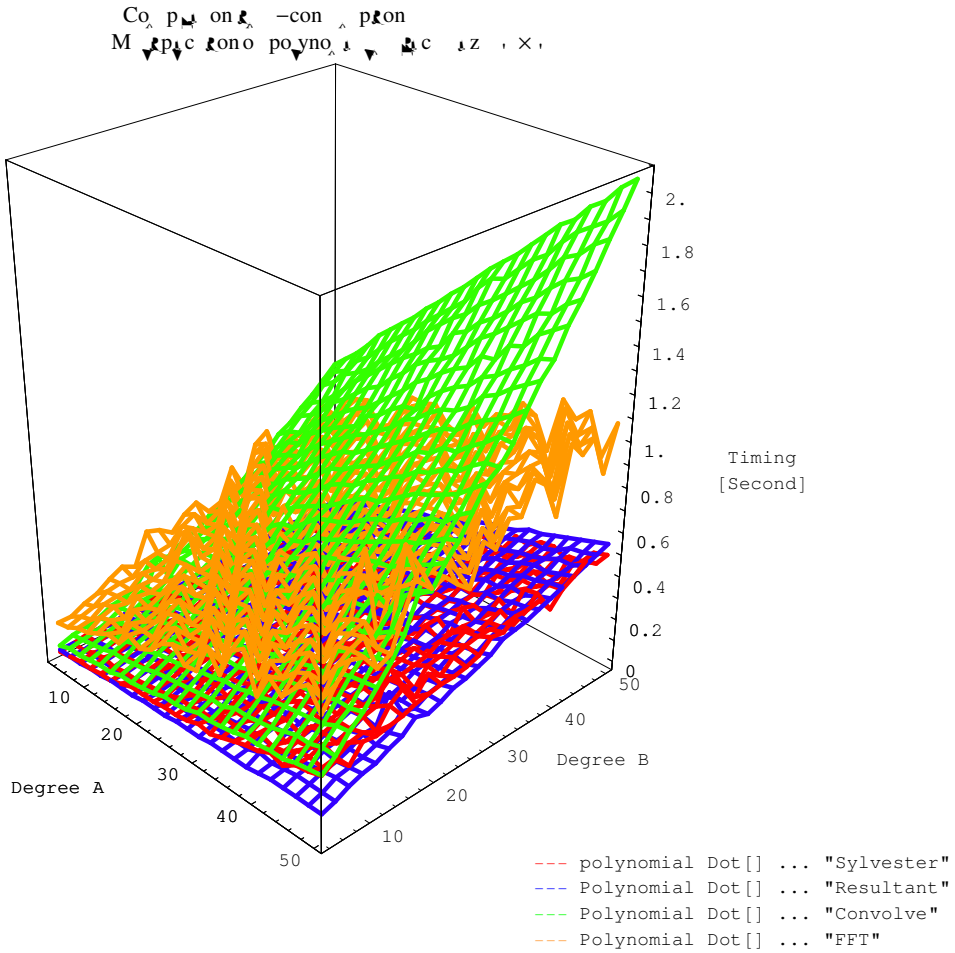
$$\{0.\ \texttt{Second}, \{\{0, 1+s^{5000}\}, \{s^{200}, 0\}\}\}$$

Rychlost výpočtu je nulová.

Na obrázku 3.6 jsou znázorněny výpočetní časy všech čtyř metod již bez standardní funkce, která je oproti ostatním metodám nesrovnatelně pomalejší.

Na obrázku 3.7 jsou detailně zobrazeny výpočetní časy a porovnány jednotlivé metody výpočtů. V prvním grafu je porovnána Sylvesterova a přímá metoda výpočtu koeficientů. Jak je vidět, tak metoda přímého výpočtu je o málo pomalejší pouze v případech, kdy stupeň $d_A > d_B$. Jestliže $d_A \ll d_B$, pak při výpočtu Sylvesterovou metodou se provádí mnohokrát zbytečné násobení nulou (Sylvesterova matice S_A obsahuje nenulové prvky pouze v úzkém pruhu kolem diagonály) na rozdíl od metody přímého výpočtu, kde se s nulovými maticemi nepočítá. V druhém grafu jsou srovnány časy výpočtů metodou konvoluce a FFT. Zpočátku vychází rychleji metoda konvoluce, ale pro větší stupně polynomiálních matic je rychlejší metoda založená na FFT. Z grafu je patrné, že u těchto metod tolik nezáleží na rozdílu d_A a d_B. Dále si můžeme všimnout jak rychlost výpočtu metodou FFT závisí na mocninách čísla 2 součtu $d_A + d_B$.

V tabulce 3.2 jsou porovnány časy výpočtů v systému *MATHEMATICA* a *PTX* pro *MATLAB*.



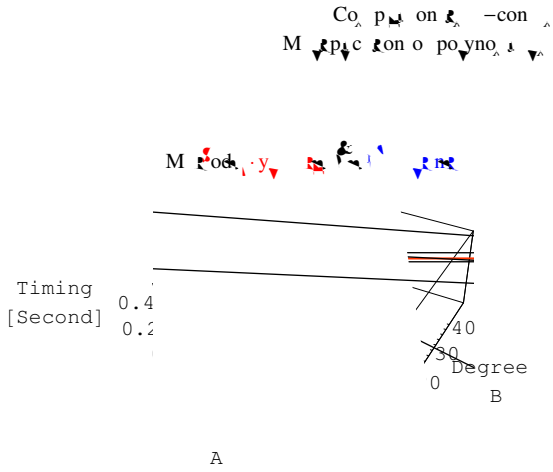
Obrázek 3.6: Porovnání časové náročnosti výpočtu funkce `Dot[A,B]` - metoda Sylvester, Resultant, Convolve a FFT.

3.5 Determinant

Funkce `Det[A]` vypočte determinant čtvercové polynomiální matice a provede nulování reálných čísel podle globální proměnné `$Zeroing`.

Námi definovaná funkce počítá pouze s číselnými polynomiálními maticemi. Jestliže vstupní matice obsahuje parametry, pak se výpočet provádí standardní funkcí pro výpočet determinantu čtvercové matice. V tomto případě je čas výpočtu pro rozměrnější matice větších stupňu značně dlouhý. Omezení na výpočty s číselnými polynomiálními maticemi je dáno metodou, která používá DFT resp. funkci `Fourier[ ]`, která pracuje pouze se seznamy čísel.

Počítání s nečíselnými polynomiálními maticemi lze uskutečnit např. Vandermondeho solverem, který v naší práci není dosud implementovaný.


Timing
[Second]
0.4
0.2
40
30
Degree
0
B
A

```
Det[A]
Out[29] =
```

$$\begin{pmatrix} 0.53846153846153846153846153846154 & -1.8000000000000000000000000000000\,s \\ 0.33333333343333333333333333333333\,s & 1.5714285714285714285714285714286\,s \end{pmatrix}_s$$

$$(\,0.84615384615384615384615384615384\,s + 0.60000000018000000000000000000000\,s^2\,)_s$$

Použité metody výpočtu

Dosud je implementována pouze jediná metoda, která je založená na DFT. V kapitole 1.2.5 je algoritmus odvozen.

Metoda FFT. Postup řešení lze popsat ve třech krocích

Vstup: *Čtvercová polynomiální matice* $P(s)_{[n\times n]} = P_0 + P_1 s + \ldots + P_{d_P} s^{d_P}$.

Výstup: *Skalární polynom* $p(s) = p_0 + p_1 s + \ldots + p_{d_p} s^{d_p}$.

Krok 1. *Výpočet horní hranice stupně hledaného polynomu* $p(s)$ *podle vztahu*

$$d_p = \min\{\sum_{i=1}^{n} deg_{c_i}(P(s)), \sum_{i=1}^{n} deg_{r_i}(P(s))\},$$

kde $deg_{c_i}(P(s))$ *je i-tý sloupcový stupeň a* $deg_{r_i}(P(s))$ *je i-tý řádkový stupeň.*

Krok 2. *Výpočet diskrétní Fourierovy transformace seznamu matic*

$$\{P_0, P_1, \ldots, P_{d_P}, O_{d_P+1}, \ldots, O_{d_p}\},$$

který odpovídá koeficientům polynomiální matice $P(s)$ *doplněným o příslušný počet nulových matic. Obdržíme tím nový seznam matic* $\{X_k, \,|\, k = 0, 1, \ldots, d_P + d_p\}$.

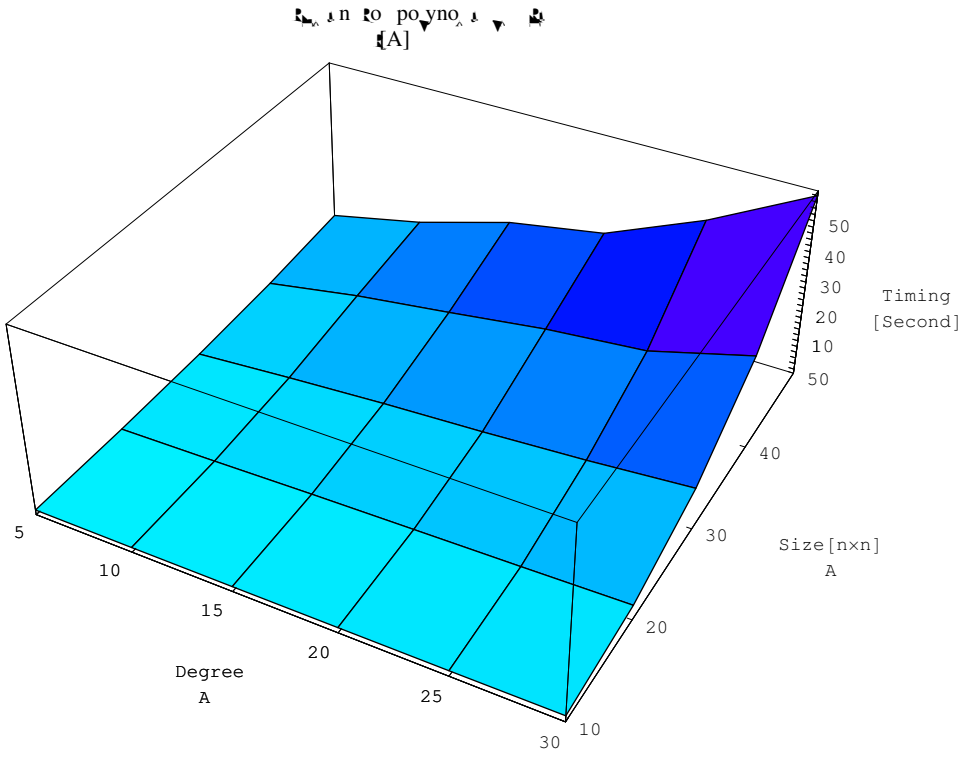
Krok 3. *Výpočet seznamu čísel* $y = \{y_0, y_1, \ldots, y_{d_P+d_p}\}$, *kde* $\{y_k = Det(X_k) \,|\, k = 0, 1, \ldots, d_P + d_p\}$.

Krok 4. *Užití inverzní diskrétní Fourierovy transformace na seznam čísel* $\{y_k \,|\, k = 0, 1, \ldots, d_P + d_p\}$. *Získáme tím seznam čísel* $\{p_k \,|\, k = 0, 1, \ldots, d_P + d_p\}$, *jehož prvky odpovídají hledaným koeficientům polynomu* $p(s)$.

Porovnání časové náročnosti výpočtu

Na obrázku 3.8 je graf znázorňující časovou náročnost výpočtu funkce `Det[A]` v závislosti na rozměru vstupní čtvercové polynomiální matice a jejím stupni.

V tabulce 3.3 jsou znázorněny výpočetní časy funkce `Det[A]` a stejné funkce `det` , v Polynomiálním Toolboxu pro *MATLAB*.



Obrázek 3.8: Časová náročnost výpočtu funkce `Det[A]`.

Tabulka 3.3: Porovnání časové náročnosti výpočtu funkce det A.

Vstupní matice d_A	n	*MATHEMATICA* "FFT"	"standard"	*PTX MATLAB*
5	12	0.08	955	0.05
10	20	0.82	**	0.19
20	20	1.73	**	0.47
50	20	3.59	**	0.72
20	10	0.25	**	0.03
20	50	18.29	**	4.85

3.6 Rovnice typu $A.X = B$

Příkaz `AXBSolve[A,B]` nalezne partikulární řešení polynomiální rovnice $A.X = B$, kde A a B jsou polynomiální matice nebo skalární polynomy. Pokud řešení neexistuje, funkce vrací hodnotu `Null` s varovným hlášením, že nebylo nalezeno žádné

řešení.

Funkce `AXBSolve[A,B,degree,{varX}]` vypočte řešení X stupně *degree*. Pokud není třetí parametr *degree* zadán, vyhledá se řešení X nejmenšího možného stupně pomocí iteračního výpočtu. Jestliže *degree* je záporné číslo, vypočte se řešení X největšího stupně, který tvoří horní hranici pro nalezení řešení při iterativním výpočtu.

Čtvrtý parametr $\{varX\}$ je nepovinný. Pokud je uveden, pak výsledek řešení je vrácen ve tvaru „pravidla“ `Rule`, $\{$`varX` $\rightarrow$ `X`$\}$. Funkce `Rule[]` se často používá v kombinaci s příkazem dosazení `Replace[]`, zkráceně `/.` [2]. Jinak je vrácena samotná hledaná polynomiální matice X.

Funkce `XABSolve[A,B,degree,{varX}]` řeší rovnici typu $X.A = B$. Argumenty funkce mají stejný význam jako u funkce `AXBSolve[]`.

```
In[30] :=
a = PM[{{8.2 - 6.2 s - 3.5 s^2, 8.7, 0, -5.6}, {2.4 + 2.4 s, -1.3 + 2.8 s, 2.6, -7.}}, s]
b = PM[{{-4.8 - 8.2 s - 5.9 s^2, 4.6 - 6.1 s + 8.4 s^2}, {-4.9 + 5.3 s, 0}}, s]
x1 = AXBSolve[a, b, {x}]        najde řešení X minimálního stupně
x2 = AXBSolve[a, b, 2, {x}]        najde řešení X stupně 2
a.x == b /. {x1, x2}        ověření správnosti
```

```
Out[30] =
{x -> {{1.69, -2.4}, {-1.19, 2.43}, {-0.08 + 0.69 s, 1.9 + 9.69 s}, {1.47 - 0.4 s, -0.57 + 3.75 s}}_s}
```

```
Out[31] =
{x -> {{1.33, -1.03}, {-0.97 - 0.3 s, 1.12 - 1.24 s}, {-0.12 + 0.43 s + 0.92 s^2, -0.06 - 0.06 s - 0.97 s^2}, {1.29 - 0.47 s + 0.22 s^2, -0.58 + 0.3 s - 0.86 s^2}}_s}
```

```
Out[32] =
{True, True}
```

V následujícím případě nebylo nalezeno žádné řešení. Je generováno varovné hlášení a je vrácena hodnota `Null`.

In[33] :=

$$\texttt{a} = \texttt{PM}[\begin{pmatrix} 2.7 & -3.4 + 1.5\,s + 4.7\,s^2 \\ -1.4 & 6.1 + 8.9\,s \end{pmatrix}, \texttt{s}]$$

$$\texttt{b} = \texttt{PM}[\begin{pmatrix} 3.\,s & -4. - 0.8\,s \\ 4.1 + 9.4\,s & -9. \end{pmatrix}, \texttt{s}]$$

$$\texttt{AXBSolve}[\texttt{a}, \texttt{b}]$$

Out[33] =

AXBSolve::nosol: No polynomial solution was found.

Snadno se počítá i s polynomiálními maticemi, které navíc obsahují parametry.

In[34] :=

$$\texttt{a} = \texttt{PM}[\begin{pmatrix} k & 1+s & 4 \\ -3\,s^2 & 2\,s & -6+s^2 \end{pmatrix}, \texttt{s}];$$

$$\texttt{b} = \texttt{PM}[\begin{pmatrix} -2+5\,s \\ s \end{pmatrix}];$$

$$\texttt{sol} = \texttt{AXBSolve}[\texttt{a}, \texttt{b}, \{\texttt{x}\}]$$

$$\texttt{Simplify}[\texttt{a}.\texttt{x} == \texttt{b}/.\texttt{sol}]$$ ověření správnosti

Out[34] =

$$\left\{x \to \begin{pmatrix} \frac{-6\,(441+42\,k+k^2)}{-324+855\,k+84\,k^2+2\,k^3} + \frac{9\,(12+k)\,s}{-324+855\,k+84\,k^2+2\,k^3} + \frac{9\,(21+k)\,s^2}{-324+855\,k+84\,k^2+2\,k^3} \\ 1 + \frac{81\,(12+k)}{-324+855\,k+84\,k^2+2\,k^3} - \frac{9\,(252+33\,k+k^2)\,s}{-324+855\,k+84\,k^2+2\,k^3} \\ \frac{27\,(12+k)\,s}{-324+855\,k+84\,k^2+2\,k^3} + \frac{27\,(21+k)\,s^2}{-324+855\,k+84\,k^2+2\,k^3} \end{pmatrix}_s\right\}$$

Out[35] =

True

Použité metody výpočtu

Metoda Sylvesterových matic je založena na převodu polynomiálních matic na konstantní Sylvesterovy matice a následném řešení konstantní lineární soustavy rovnic. Zásadní problém je v odhadnutí stupně d_X hledané matice X. Vyhledávání minimálního stupně d_X se provádí váženým binárním vyhledáváním mezi vypočtenou dolní a horní hranicí odhadnutého stupně. Obecné použití Sylvesterovy metody je popsáno v kapitole 1.2.3.

Vstup: *Polynomiální matice*

$$A(s)_{[p\times r]} = A_0 + A_1 s + \ldots + A_{d_A} s^{d_A},$$

$$B(s)_{[p\times q]} = B_0 + B_1 s + \ldots + B_{d_B} s^{d_B}.$$

Výstup: *Polynomiální matice* $X(s)_{[r\times q]}$, *která vyhovuje rovnici*

$$A(s)\,.\,X(s) \;=\; B(s)$$

a má minimální stupeň

Krok 1. *Konstrukce Sylvesterových matic.*

$$S_A = \begin{bmatrix} A_0 & & & 0 \\ A_1 & A_0 & & \\ \vdots & A_1 & \ddots & \\ A_{d_A} & \vdots & & A_0 \\ & A_{d_A} & & A_1 \\ & & \ddots & \vdots \\ 0 & & & A_{d_A} \end{bmatrix}_{[(1+d_A+d_B)p\times\ (1+d_X)r]} , S_B = \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_{d_B} \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{[(1+d_A+d_B)p\times\ q]} \tag{3.3}$$

Krok 2. *Řešení konstantní lineární maticové soustavy rovnic* $S_A.S_X = S_B$.

Krok 3. *Kroky 1 a 2 se opakují pro různé stupně* d_X *dokud není vybráno řešení s nejmenším možným stupněm. Stupeň* d_X *se určuje iterací pomocí váženého binárního vyhledávání mezi minimálním a maximálním možným stupněm, přičemž hranice se upravují podle existence předešlého řešení.*

Krok 4. *Řešením rovnice v kroku 2 získáme matici* S_X, *kterou lze zapsat ve tvaru*

$$S_X \;=\; [X_0, X_1, \ldots, X_{d_X}]^T_{[q\times(1+d_x)r]}.$$

Prvky $\{X_0, X_1, \ldots, X_{d_X}\}$ *matice* S_X *tvoří koeficienty výsledné matice řešení.*

Rovnice $X\,.\,A \;=\; B$ se řeší převodem na rovnici typu $A\,.\,X \;=\; B$. Neboť po transpozici celé rovnice dostaneme rovnici $A^T\,.\,X^T \;=\; B^T$, která odpovídá rovnici $\mathbb{A}\,.\,\mathbb{X} = \mathbb{B}$. Pro výslednou matici potom platí $X = \mathbb{X}^T$.

Porovnání časové náročnosti výpočtu

Pro srovnání jsou nyní uvedeny dva možné postupy výpočtu pomocí standardních metod a funkcí systému *MATHEMATICA*.

Konstrukce Sylvesterových matic ve skutečnosti není nic jiného než přímá maticová varianta metody neurčitých koeficientů. Samotnou metodu neurčitých koeficientů lze snadno implementovat a řešit pomocí interní funkce `SolveAlways[]`, např. takto.

Výpočet 1.

```
1     řešení rovnice A.X == B
2
3     konstrukce neznámé matice X s koeficienty x[i,j,d]
4       v proměnné s a stupně deg
5   X =
6       Sum[
7           Table[
8               x[i, j, d] s^d
9               , {i, r = Dimensions[A][[2]]}
10              , {j, c = Dimensions[B][[2]]}
11          ]
12          , {d, 0, deg}
13      ];
14
15    porovnání koeficientů polynomů na jednotlivých
16        pozicích matic A.X a B
17    řešení soustavy rovnic
18  X /. SolveAlways[
19          Flatten[
20              MapThread[Equal, {A.X, B}, 2]
21          ]
22          , s
23        ][[1]]
```

Tato funkce dává správné výsledky, ale časová náročnost výpočtu je nesrovnatelně větší oproti funkci `AXBSolve[ ]`. Důvodů je několik. Nejpomalejší místo výpočtu je ve funkci `SolveAlways[ ]`, která pracně hledá všechna řešení. Další velmi pomalou operací je maticové násobení polynomiálních matic $A \,.\, X$, kde prvky matice X tvoří nečíselné polynomy $x[i,j,0] + x[i,j,1]s + \ldots x[i,j,deg]s^{deg}$. Prvky $\{x[i,j,d] \mid i = 1 \ldots r,\, j = 1 \ldots c,\, d = 0 \ldots deg\}$ jsou neznámé parametry, s kterými se velmi pracně počítá.

Značného urychlení výpočtu lze dosáhnout náhradou funkce `SolveAlways[ ]` rychlejší interní funkcí `LinearSolve[ ]` a převodem polynomů na seznamy jejich koeficientů, z nichž již lze sestrojit příslušné rovnice a následně matice, které odpovídají upraveným Sylvesterovým maticím.

Výpočet 2.

```
24    výpočet pomocí rychlejší funkce LinearSolve[]
25      převod polynomů na seznamy koeficientů
26      konstrukce odpovídajících rovnic
27      transformace do odpovídajícího maticového tvaru
28  lssol =
```

```
29      LinearSolve
30          LinearEquationsToMatrices[
31              Flatten[
32                  MapThread[
33                    Equal,
34                    {Map[CoefficientList[#, s]&, A.X, {2}],
35                     Map[CoefficientList[#, s]&,
36                             B + 0.s^ Max[Exponent[A, s]]+deg , {2}]
37                    }
38                    ,
39                  ]
40              ],
41              Flatten[
42                  Array[x[#2, # , #1] &, {deg + 1, r, c}, {0, 1, 1}]
43              ]
44          ];
45
46       konstrukce v´ysledn´eho polynomi´aln´ matice
47  Sum[
48      Partition[Partition[lssol, c], r][[i + 1]] s^i
49      , {i, 0, maxd}
50  ]
```

V tabulce 3.4 jsou porovnány výpočetní časy funkce `AXBSolve[ ]` a stejné funkce `axb` v Polynomiálním Toolboxu pro *MATLAB* (váhová funkce binárního vyhledávání je stejná). Také jsou uvedeny časy variant výpočtů 1 a 2.

Tabulka 3.4: Porovnání časové náročnosti výpočtu polynomiální rovnice $A.X = B$ (stupeň matice A a B je d a rozměry jsou $[n \times n+5]$ a $[n \times n]$).

d	n	`AXBSolve[]`	V 1	V 2	*PTX MATLAB*
2	5	0.08	284.6	1.2	0.12
2	20	0.28	**	**	0.29
5	10	0.62	**	**	1.1
10	10	69.0	**	**	61.5
10	15	200.0	**	**	158.2

3.7 Diofantická rovnice

Funkce `DESolve[eq]` řeší dva typy Diofantických rovnic

$$A.X + B.Y + C.Z + \ldots \quad = \quad D \tag{3.4}$$

$$X.A + Y.B + Z.C + \ldots \quad = \quad D. \tag{3.5}$$

Symboly A, B, C, D jsou polynomiální matice nebo skalární polynomy, které mohou obsahovat parametry. Rovnici lze zadávat i ve složitějším tvaru, který po úpravě vede na předepsaný typ rovnice. Funkce vrací pouze partikulární řešení minimálního stupně, pokud existuje.

```
In[36] :=
```

$$\texttt{a} = \texttt{PM}[\begin{pmatrix} 1+s \\ k \end{pmatrix}, \texttt{s}]$$

$$\texttt{b} = \texttt{PM}[\begin{pmatrix} -s & 3+2\,s \\ 2\,s^2 & 5 \end{pmatrix}, \texttt{s}]$$

$$\texttt{c} = \texttt{PM}[(\,2\,s \quad 1-s\,), \texttt{s}]$$

$$\texttt{d} = \texttt{PM}[\begin{pmatrix} 2 & -3+s^2 \\ 2+s & 13 \end{pmatrix}, \texttt{s}]$$

$$\texttt{sol} = \texttt{DESolve}[\texttt{eq} = (- \ * \texttt{a}).\texttt{x} + \texttt{b.y} + \texttt{Transpose}[\texttt{c}].\texttt{c.z} == \texttt{d} + \texttt{P}[1 - \texttt{s}^2]]$$

```
Out[369] =
```

$$\{x \to \left(\frac{5\,k}{-3-41\,k+24\,k^2} + \frac{(1-k)\,s}{-3-41\,k+24\,k^2} \quad \frac{-57-406\,k}{3\,(-3-41\,k+24\,k^2)} - \frac{2\,(-43+5\,k)\,s}{3\,(-3-41\,k+24\,k^2)} \right)_s,$$

$$z \to \begin{pmatrix} \frac{9+76\,k-45\,k^2}{2\,(-3-41\,k+24\,k^2)} & \frac{43-48\,k+5\,k^2}{-3-41\,k+24\,k^2} \\ \frac{-3\,(-2-19\,k+11\,k^2)}{-3-41\,k+24\,k^2} & \frac{43-91\,k+10\,k^2}{-3-41\,k+24\,k^2} \end{pmatrix}_s,$$

$$y \to \begin{pmatrix} \frac{-3\,(-1-10\,k+6\,k^2)}{-3-41\,k+24\,k^2} & \frac{23+18\,k-12\,k^2}{-3-41\,k+24\,k^2} \\ \frac{3\,(-1-12\,k+8\,k^2)}{-3-41\,k+24\,k^2} & \frac{-17-108\,k-16\,k^2}{-3-41\,k+24\,k^2} \end{pmatrix}_s \}$$

Dosazením do vstupní rovnice a následnou úpravou se snadno přesvědčíme o správnosti řešení.

```
In[37] :=   eq /. sol // Simplify

Out[37] =

            True
```

Zadáním druhého parametru `DESolve[eq, degree]` funkce vrací řešení daného stupně *degree*, pokud existuje.

```
In[38] :=   SetOptions[PMRandom, MaxDeg -   ];
            a = PMRandom[{2, 2}]; b = PMRandom[{ , 2}];
            c = PMRandom[{2, 2}];
            sol = DESolve[eq = x.a + y.b == c,  ]
            eq /. sol        ověřená správnosti

Out[38] =
```

$$\{x \to \begin{pmatrix} 0.41 + 1.17\,s + 1.38\,s^2 + 0.85\,s^3 & -0.79 - 0.88\,s - 1.21\,s^2 - 0.57\,s^3 \\ 0.22 + 0.15\,s - 0.76\,s^2 + 0.082\,s^3 & 0.23 + 0.63\,s - 0.052\,s^2 + 0.065\,s^3 \end{pmatrix},$$
$$y \to \begin{pmatrix} 0. & 0.74 + 1.91\,s & -0.30 - 1.08\,s - 1.42\,s^2 - 0.71\,s^3 \\ 0. & -1.015 + 0.038\,s & -1.41 + 0.54\,s + 0.52\,s^2 - 0.069\,s^3 \end{pmatrix}\}$$

```
Out[39] =
```

True

Výpočet rovnice $a\,x + b\,y = c$

Pokud se jedná o rovnici ve tvaru $a\,x + b\,y == c$, kde a, b a c jsou skalární polynomy, je výhodnější použít funkci `AXBYCSolve[a,b,c]`, která používá upravenou metodu výpočtu. Na hledaná řešení je navíc možné klást speciální požadavky na minimálnost stupně x nebo y. Pokud není uveden žádný požadavek, pak řešení skalárního polynomu x má minimální stupeň. Pokud je uvedena volba `MinimalDegree- "Y"`, je výsledné řešení y minimálního stupně.

```
In[40] :=    a = P[ .   + 6.6 s];
             b = P[ .6  - 7.  s + 8.6 s^2];
             c = P[2.0  +  .1 s + 9.  s^2 - 7.  s^  + 1.  s^ ];
             sol1 = AXBYCSolve[a, b, c, {x, y}]
             sol2 = AXBYCSolve[a, b, c, {x, y}, MinimumDegree -  "Y"]
             sol  = DESolve[eq = a x + b y == c]
             eq /. {sol1, sol2, sol }
```

```
Out[40] =
```

$$\{x \to (0.87 + 0.66\,s)_s\,,\ y \to \left(-0.15 - 0.71\,s + 0.17\,s^2\right)_s\}$$

```
Out[41] =
```

$$\{\{x \to \left(0.19 + 1.71\,s - 1.23\,s^2 + 0.23\,s^3\right)_s\,,\ y \to (0.24)_s\}$$

```
Out[42] =
```

$$\{\{x \to \left(0.74 + 0.84\,s - 0.20\,s^2\right)_s\,,\ y \to \left(-0.077 - 0.55\,s + 0.17\,s^2\right)_s\}$$

```
Out[43] =
```

{True, True, True}

Použité metody výpočtu

Funkce `DESolve[ ]`

V případě polynomiálních matic, ale i skalárních polynomů u funkce `DESolve[eq]`, se vychází z možnosti přepisu zadané Diofantické rovnice na rovnici typu

$$\mathbb{A}\,.\,\mathbb{X} = \mathbb{B}, \tag{3.6}$$

jejíž řešení bylo rozebráno v kapitole 3.6.

Transformace na rovnici (3.6) se provádí následujícím způsobem.

- Rovnici $A.X + B.Y + C.Z + \ldots = D$ snadno přepíšeme do požadovaného tvaru $\mathbb{A}.\mathbb{X} = \mathbb{B}$ takto
$$[\,A \quad B \quad C \quad \ldots\,]\,.\begin{bmatrix} X \\ Y \\ Z \\ \vdots \end{bmatrix} = D$$
- Rovnici $X.A + Y.B + Z.C + \ldots = D$ podobně jako v předchozím případě přepíšeme do požadovaného tvaru $\mathbb{A}.\mathbb{X} = \mathbb{B}$ tímto způsobem
$$[\,A^T \quad B^T \quad C^T \quad \ldots\,]\,.\begin{bmatrix} X^T \\ Y^T \\ Z^T \\ \vdots \end{bmatrix} = D^T$$

Hledané matice X, Y, Z jsou zrekonstruovány z vypočtené složené matice $\mathbb{X}$.

Funkce `AXBYCSolve[ ]`

Funkce `AXBYCSolve[a,b,c]`, která počítá se skalárními polynomy, nevyužívá možnosti přepisu na rovnici (3.6) jako funkce `DESolve[eq]`.

Pro skalární polynomy lze z analýzy počtu neznámých koeficientů a počtu rovnic přímo zjistit stupně hledaných polynomů x a y a přímo zkonstruovat odpovídající Sylvesterovy matice. Jestliže polynomy a a b jsou nesoudělné, pak řešení rovnice

$$a\,x \;+\; b\,y \;=\; c$$

s minimálním stupněm x dostaneme volbou těchto stupňů

$$\begin{aligned} \deg x &= \deg b - 1 \\ \deg y &= \begin{cases} \deg a - 1 & \text{pro} \quad \deg a + \deg b > \deg c \\ \deg c - \deg b & \text{pro} \quad \deg a + \deg b \leq \deg c \end{cases} \end{aligned} \tag{3.7}$$

Zkonstruovaná soustava rovnic má řešení právě tehdy, když největší společný dělitel polynomů a, b dělí polynom c a má jediné řešení právě když $(a, b) = 1$.

V případě polynomiálních matic nelze takto snadno spočítat stupně neznámých polynomiálních matic a proto se používá iterační postup výpočtu v určitých mezích možných stupňů, který byl již popsán.

Všechny dosud popsané metody mají jednu nevýhodu a to, že hledají pouze jedno partikulární řešení namísto kompletního obecného řešení.

Problém obecného řešení lze vyřešit pomocí funkce, která počítá nulový prostor složené matice $\mathbb{A}$ z rovnice (3.6), tj. nenulové řešení rovnice $\mathbb{A}.\mathbb{X} = 0$, které tvoří prostor parametrů. Funkce pro výpočet nulového prostoru matice $\mathbb{A}$ nebyla zatím implementována.

Potom obecné řešení Diofantické rovnice

$$AX + BY = C \tag{3.8}$$

lze zapsat ve tvaru

$$\begin{aligned} X &= X_0 + T\,X_1 \\ Y &= Y_0 + T\,Y_1, \end{aligned} \tag{3.9}$$

kde polynomiální matice X_0, Y_0 odpovídají partikulárnímu řešení rovnice (3.8) a X_1, Y_1 jsou řešením příslušné homegenní rovnice. T je libovolná polynomiální matice odpovídajících rozměrů.

Pokud známe obecné řešení Diofantické rovnice, pak lze klást další podmínky na hledaná řešení podobně jako u funkce `AXBYCSolve`. Nejčastějším požadavkem je minimalizovat stupeň řešení matice X nebo Y. K tomu je zapotřebí další funkce pro dělení polynomiálních matic, která vrací podíl a zbytek po dělení. Tato funkce také není doposud implementována.

Jestliže platí, že $\deg X_1 \leq \deg X_0$, pak k polynomiálním maticím X_0 a X_1 lze vždy nalézt polynomiální matice U a V takové, že

$$X_0 = U.X_1 + V, \qquad \text{kde} \qquad \deg U < \deg V. \tag{3.10}$$

Polynomiální matice U je podíl a V je je zbytek při dělení polynomiální matice X_0 polynomiální maticí X_1.

Nyní lze obecné řešení (3.9) zapsat ve tvaru

$$X = U.X_1 + V + T\,X_1 = (U + T)X_1 + V.$$

Jestliže zvolíme $T = -U$ dostaneme řešení s minimálním stupně polynomiální matice X ve tvaru

$$\begin{aligned} X &= V \\ Y &= Y_0 - U\,Y_1. \end{aligned}$$

Obecná teorie o řešitelnosti skalární Diofantické rovnice, řešení pomocí největšího společného dělitele

Nechť je dána rovnice

$$ax + by = c, \tag{3.11}$$

kde a,b a c jsou zadané skalární polynomy a x a y jsou hledané skalární polynomy.

Nejprve určíme podmínku řešitelnosti. Rovnice (3.11) má řešení právě tehdy, když největší společný dělitel g polynomů a a b dělí polynom c. To znamená, že pro polynom c platí následující rovnost $c = c^0 g$. Zkrácený zápis této podmínky vypadá následovně

$$(a, b)|c. \tag{3.12}$$

Podmínku řešitelnosti ověříme následujícím způsobem. Jestliže g je největší společný dělitel a a b, pak platí rovnosti $a = ga^0$ a $b = gb^0$, kde a^0 a b^0 jsou nesoudělné polynomy. Rovnici (3.11) lze přepsat na tvar

$$g(a^0 x + b^0 y) = c$$

a po vynásobení $1/g$

$$a^0 x + b^0 y = \frac{c}{g} = c^0. \tag{3.13}$$

Aby x a y z poslední rovnice (3.13) byly polynomy, musí být podíl c/g polynom, tj. $g|c$.

Řešení rovnice (3.11) určíme z vlastnosti největšího společného dělitele (a, b). Platí následující tvrzení. Je-li $g = (a, b)$, pak lze vždy nalézt polynomy p, q, r a s takové, že platí rovnosti

$$ap + bq = g \tag{3.14}$$

$$ar + bs = 0. \tag{3.15}$$

Navíc $l = -ar = bs$ je nejmenší společný násobek a a b. Přenásobením rovnice (3.14) polynomem c^0 dostaneme následující rovnici

$$apc^0 + bqc^0 = gc^0 = c. \tag{3.16}$$

Polynomy

$$x = pc^0 \qquad \text{a} \qquad y = qc^0 \tag{3.17}$$

z rovnice (3.16) jsou právě řešením naší rovnice (3.11).

Toto nalezené řešení není jediným řešením, které vyhovuje Diofantické rovnici (3.11). Všechna řešení lze zapsat ve tvaru

$$x = x_0 + x_1 t \tag{3.18}$$

$$y = y_0 + y_1 t, \tag{3.19}$$

kde t značí libovolný polynom. Polynomy x_0 a y_0 jsou partikulárním řešením rovnice (3.11), kterým vyhoví například řešení (3.17). Polynomy x_1 a y_1 jsou řešením homogenní rovnice

$$ax + by = 0. \tag{3.20}$$

Důkaz provedeme přímo dosazením do rovnice (3.11)

$$a(x_0 + x_1 t) + b(y_0 + y_1 t) = c$$

a následnou úpravou se podle předpokladů přesvědčíme o rovnosti

$$\underbrace{ax_0 + by_0}_{c} + t\,\underbrace{ax_1 + by_1}_{0} = c.$$

Řešení homogenní rovnice (3.20) určíme snadno. Rovnici přepíšeme na tvar

$$g(a^0 x + b^0 y) = 0.$$

Potom určitě platí rovnice $g(a^0 b^0 + b^0(-a^0)) = 0$, neboli polynomy b_0 a $-a_0$ jsou řešením (3.20). Protože polynomy b_0 a $-a_0$ jsou nesoudělné, tvoří nejjednodušší řešení, které musí být obsaženo v každém jiném řešení. Tedy obecné řešení je ve tvaru

$$x = b^0 t \qquad \text{a} \qquad y = -a^0 t, \tag{3.21}$$

kde t je libovolný polynom.

Nakonec lze obecné řešení napsat ve tvaru

$$\begin{aligned} x &= p\frac{c}{g} + b^0 t \\ y &= q\frac{c}{g} - a^0 t \end{aligned}$$

a vezmeme-li v úvahu rovnici (3.15) a (3.20) dostaneme výsledek

$$\begin{aligned} x &= p\frac{c}{g} + rt \\ y &= q\frac{c}{g} + st. \end{aligned}$$

Takto popsané řešení Diofantické rovnice spočívá tedy v nalezení polynomů p, q, r a s při výpočtu největšího společného dělitele g polynomů a a b.

Popsaný postup výpočtu lze velmi snadno implemetovat pomocí interních funkcí systému *MATHEMATICA*. Zápis programového kódu může vypadat například takto.

Výpočet 1.

```
51    (* výpočet rce. ax+by=c pomocí funkce GCD
52       zavedením potřebného balíčku
53 << Algebra`PolynomialExtendedGCD`
```

```
54
55      (* výpočet nejvyššího spol. dělitele a pol. p, q *)
56  {g, {p, q}} = PolynomialExtendedGCD[a, b]
57
58      (* dělení polynomů *)
59  c0 = PolynomialQuotient[c, g, s]
60  a0 = PolynomialQuotient[a, g, s]
61  b0 = PolynomialQuotient[b, g, s]
62
63      (* výsledný tvar řešení *)
64  If[PolynomialRemainder[c, g, s] === 0,
65    {x -  Expand[p c0] + b0 t,
66     y -  Expand[q c0] - a0 t}
67    , Null
68  ]
```

Velká nevýhoda této implementace je ve velmi malé rychlosti výpočtu funkce `PolynomialExtendedGCD[ ]`[1] a nutnost zadávat koeficienty polynomů jako přesná čísla. To znamená, že v případě reálných čísel je nutné volat standardní funkci `Rationalize[`*polynom*`,0]`, která převede koeficienty polynomu na racionální čísla, která ale v případě reálných čísel s přesností 16 míst mají značně velký čitatel i jmenovatel. Rychlost výpočtu s těmito racionálními čísly je již velmi nízká.

Porovnání časové náročnosti výpočtu

V tabulce 3.5 jsou porovnány rychlosti výpočtů funkce `DESolve[A.X+B.Y==C]` a funkce `DiophantineSolve[A,B,C]` z balíčku Polynomial Matrix Utilities. Protože funkce `DiophantineSolve[ ]` provádí výpočty pouze s polynomiálními maticemi, které mají koeficienty jako přesná čísla, je nutné generovat náhodné matice s celočíselnými koeficienty, jinak v případě reálných čísel dochází k zacyklení výpočtu.

Funkce `DiophantineEquation[ ]` nejprve provede test existence řešení. Testuje se shoda hodností složených matic A, B a A, B, C a rovnost Hermitových forem složených matic $A, B, 0$ a A, B, C. Jestliže je výsledek testu kladný, pak se dále v cyklu dosazují polynomiální matice s neurčitými koeficienty, které jsou řešeny pomocí funkce `SolveAlways[ ]`, dokud není nalezeno řešení. Stupeň řešení se v cyklu zvětšuje o jedničku.

Dále jsou v tabulce 3.5 uvedeny rychlosti výpočtů funkcí `DESolve[ ]` a stejné funkce `axbyc` v Polynomiálním Toolboxu pro *MATLAB* s polynomiálními maticemi, které mají reálné koeficienty. Z tabulky je patrný rozdíl v rychlosti výpočtu funkce `DESolve[ ]` s reálnými a celočíselnými (přesnými) koeficienty.

V tabulce 3.6 jsou porovnány rychlosti výpočtů funkce `AXBYCSolve[]` a algoritmu využívajícího výpočtu největšího společného dělitele, jenž byl popsán jako

[1] Funkce se nalézá ve standardním balíčku `Algebra PolynomialExtendedGCD`

Tabulka 3.5: Porovnání časové náročnosti výpočtu rovnice $A.X + B.Y = C$ (stupeň čtvercových matic A, B a C rozměru n je d).

d	n	`DESolve[]` *int*	`DiophantineSolve[]` *int*	`DESolve[]` *real*	*PTX MATLAB* *real*
1	5	0.04	1.7	0.04	0.12
1	8	0.08	103.2	0.08	0.15
1	10	0.14	**	0.07	0.17
5	5	0.98	59.2	0.08	0.29
5	7	1.31	453.5	0.08	0.34
5	10	5.7	**	0.17	0.43
7	5	2.7	200.1	0.12	0.37

výpočet 1, v kterém se počítá pouze partikulární řešení. Vstupy jsou skalární polynomy s reálnými a celočíselnými koeficienty. U výpočtu podle varianty 1 jsou reálná čísla převáděna na racionální čísla pomocí funkce `Rationalize[`*polynom*`,0]`.

Tabulka 3.6: Porovnání časové náročnosti výpočtu skalární rovnice $ax + by = c$ (stupeň skalárních polynomů a, b a c je d).

d	`AXBYCSolve[]` *int*	V 1 *int*	`AXBYCSolve[]` *real*	V 1 *real* $\rightarrow$ *rational*
10	0.02	0.07	0.01	2.8
15	0.04	0.37	0.02	50.8
20	0.08	5.38	0.04	790.3
30	0.21	690.1	0.06	**
50	1.13	**	0.18	**

3.8 Symetrická rovnice typu $a^*x + x^*a = b$

Funkce `AXXABSolve[a,b]` řeší polynomiální rovnici typu $a(s)^* x(s) + x(s)^* a(s) = b(s)$ pro skalární polynomy $a(s)$, $b(s)$. Polynom $b(s)$ je symetrický polynom a platí pro něj rovnost $b(s) = b(-s)$. Pro operátor $p(s)^*$ platí $p(s)^* = p(-s)$.

```
In[44] :=   a = P[k +  s - s^2, s]
            b = P[-2 + s^2 +  s^, ]
            sol =  x -  AXXABSolve[a, b]
Out[44] =
```

$$\left(k + 3\,s - s^2 \right)_s$$

$$\left(-2 + s^2 + 5\,s^4 \right)_s$$

$$x \to \left(-\frac{1}{k} + \frac{\left(2 - 3\,k - 5\,k^2\right) s}{6\,k} - \frac{5\,s^2}{2} \right)_s$$

Nyní se dosazením snadno přesvědčíme o správnosti řešení

```
In[45] :=   IsPoly[x]
            Simplify[ Star[a] x + a Star[x] == b    /. sol ]
Out[45] =

            True
```

Použité metody výpočtu

Výpočet se provádí pomocí **metody Sylvesterových matic**.

Vstup: *Skalární polynomy*

$$\begin{aligned} a(s) &= a_0 + a_1\,s + \ldots + a_{d_a}\,s^{d_a}, \\ b(s) &= b_0 + b_2\,s^2 + + b_4\,s^4 + \ldots + b_{d_b}\,s^{d_b}. \end{aligned}$$

Výstup: *Skalární polynom* $x(s) = x_0 + x_1\,s + \ldots + x_r\,s^r$, *která vyhovuje rovnici* $a(s)^*\,x(s) + x(s)^*\,a(s) = b(s)$

Krok 1. *Konstrukce Sylvesterovy matice polynomu* a. *Bez újmy na obecnosti nechť* d_a *je liché číslo, pak*

$$S_a = \begin{bmatrix} 2\,a_0 & 0 & 0 & 0 & 0 & \\ 2\,a_2 & -2\,a_1 & 2\,a_0 & 0 & 0 & \\ \vdots & -2\,a_3 & 2\,a_2 & -2\,a_1 & 2\,a_0 & \\ 2\,a_{d_a-1} & \vdots & \vdots & -2\,a_3 & 2\,a_2 & \\ 0 & -2\,a_{d_a} & 2\,a_{d_a-1} & \vdots & \vdots & \ldots \\ 0 & 0 & 0 & -2\,a_{d_a} & 2\,a_{d_a-1} & \\ 0 & 0 & 0 & 0 & 0 & \\ & & \vdots & & & \\ 0 & 0 & 0 & 0 & 0 & \end{bmatrix}_{[r \times c]}$$

Rozměr $[r \times c]$ *matice* S_a *závisí na stupních polynomů* d_a *a* d_b

$$[r \times c] = \begin{cases} [d_b/2 + 1 \times d_b - d_a + 1], & d_a \le d_b/2 \\ [d_a \times d_a], & d_a > d_b/2 \end{cases}.$$

Krok 2. *Konstrukce Sylvesterovy matice S_b polynomu $b(s)$*

$$S_b = [\, b_0 \quad b_2 \quad \ldots \quad b_{d_b} \quad 0 \quad \ldots \quad 0\,]^T_{[1 \times r]}\,.$$

Krok 3. *Řešením soustavy*

$$S_a.[\, x_0 \quad x_1 \quad \ldots \quad x_c\,]^T = S_b$$

získáme koeficienty hledaného polynomu $x(s) = x_0 + x_1\, s + \ldots + x_r\, s^r$.

Porovnání časové náročnosti výpočtu

V tabulce 3.7 jsou znázorněny výpočetní časy funkce `AXXABSolve[]` a stejné funkce `axxab` v Polynomiálním Toolboxu pro *MATLAB*.

Tabulka 3.7: Porovnání časové náročnosti výpočtu skalární rovnice $a^*\, x + a\, x^* = b$ (stupeň polynomu $\deg a = d$ a $\deg b = 2d$).

d	`AXXABSolve[]`	*PTX MATLAB*
20	0.02	0.08
60	0.09	0.38
200	0.93	3.78
300	2.11	11.83

3.9 Další užitečné funkce

V předchozích kapitolách byly popsány doposud vytvořené základní funkce pro počítání s polynomiálními maticemi. V této kapitole jsou stručně [2] popsány jednoduší funkce, které jsou ale velmi užitečné a často používané při různých výpočtech s polynomiálními maticemi.

3.9.1 `PMRandom[ ]`

Vytvoří polynomiální matici s náhodnými koeficienty, které mohou být reálné, reálné s danou přesností, komplexní nebo celočíselné. Typ čísel se nastavuje pomocí volby `Data-` {*typ*,{*min*,*max*},*precision*}. Vynecháme-li *precision*, pak reálná čísla jsou s přesností `MachinePrecision`, která odpovídá přibližně číslu 16. Vynecháním intervalu {*min*,*max*} se berou čísla z intervalu $\langle 0, 1\rangle$.

[2] Více funkcí s podrobnějším popisem a příklady je popsáno v elektronické dokumentaci, která je součástí programového balíčku.

Příkaz `PMRandom[]` generuje náhodnou polynomiální matici. Rozměr matice je náhodný stejně jako stupně polynomů na jednotlivých pozicích matice. Maximální rozměry matice a stupně polynomů se nastavují pomocí voleb `MaxRow-` *num*, `MaxCol-` *num* a `MaxDeg-` *num*.

Příkaz `PMRandom[`*deg*`,`*row*`,`*col*`,`*var*`,`*opts*`]` generuje náhodnou polynomiální matici rozměru $[row \times col]$ a stupně *deg* v proměnné *var*. Argumenty *opts* jsou možné volby, které řídí výstup funkce. Argumenty *var* i *opts* jsou nepovinné. Jestliže argument *var* není uveden, pak jako proměnná výsledné polynomiální matice se bere hodnota z globální proměnné `$Variable`.

Trvalé nastavení voleb *opts* se provádí příkazem `SetPrecision[PMRandom,` *opts*`]`.

Pokud argumenty příkazu začínají symbolem, pak místo číselných koeficientů je dosazen symbol s indexem pozice a stupně.

V tabulce 3.8 jsou popsány funkce jednotlivých argumentů funkce `PMRandom[`*args*`]` a jejich kombinací.

Tabulka 3.8: Popis argumentů funkce `PMRandom[`*args*`]`

`PMRandom[`*args*`]`	význam
`[ ]`	náhodný rozměr matice, náhodné stupně polynomů matice
`[`*deg*`]`	náhodný rozměr matice, stupeň matice je *deg*
`[`*deg*`, `*n*`]`	matice rozměru $[n \times n]$, stupeň matice je *deg*
`[`*deg*`, `*r*`, `*c*`]`	matice rozměru $[r \times c]$, stupeň matice je *deg*
`[{`*n*`}]`	rozměr matice je $[n \times n]$, náhodné stupně polynomů matice
`[{`*r*`, `*c*`}]`	rozměr matice je $[r \times c]$, náhodné stupně polynomů matice
`[`*sym*`,...]`	platí předešlé, místo čísel symbol *sym* s indexy pozice a stupně
`[...,`*var*`]`	platí předešlé, polynomiální matice v proměnné *var*

3.9.2 `PRandom[ ]`

Vytvoří skalární polynom s náhodnými koeficienty, které mohou být reálné, reálné s danou přesností, komplexní nebo celočíselné. Typ čísel se nastavuje pomocí volby `Data-` {*typ*,{*min*,*max*},*precision*}.

Příkaz `PRandom[`*deg*`,`*var*`]` generuje náhodný skalární polynom stupně *deg* v proměnné *var*. Argument *var* je nepoviný. Pokud není uveden, pak jako proměnná se bere hodnota globální proměnné `$Variable`. Vynecháme-li argument *deg*, pak je vrácen skalární polynom náhodného stupně. Maximální stupeň polynomu je dán volbou `MaxDeg-` *num*.

Příkaz `PMRandom[`*sym*`,`*deg*`]` vrací skalární polynom stupně *deg* jehož koeficienty tvoří symbol *sym* s indexem mocniny proměnné.

3.9.3 StandardPolynomial[ ]

Příkaz `StandardPolynomial[poly]` převede polynomiální objekt *poly* do tvaru matice, jejíž prvky tvoří standardní symbolické polynomy. Pokud je *poly* objekt skalárního polynomu, pak je vrácen standardní symbolický polynom.

Polynomiální objekty jsou funkce `Poly[]` a `PolyMat[]`, které byly vytvořeny funkcemi pro zadávání skalárních polynomů a polynomiálních matic `P[], PC[], PM[], PMC[], PLC[]` nebo funkcí `PRandom[]` nebo `PMRandom[]`.

3.9.4 Deg[ ]

Funkce `Deg[poly,opts]` vrací stupeň skalárního polynomu nebo polynomiální matice.

Pro polynomiální matice jsou zde navíc volby pro určení řádkového `TypeDegree- "row"` a sloupcového stupně `TypeDegree- "col"`. Volba `TypeDegree- "ent"` vrací matici, jejíž prvky tvoří stupně jednotlivých polynomů zadané polynomiální matice.

3.9.5 Variable[ ]

Funkce `Variable[poly]` vrací proměnnou skalárního polynomu nebo polynomiální matice *poly.*

3.9.6 Size[ ]

Funkce `Size[poly]` vrací rozměry polynomiální matice *poly.* Pokud je *poly* skalární polynom, je vrácena hodnota $\{1, 1\}$.

3.9.7 Star[ ]

Funkce `Star[poly]` vrací konjugovanou a transponovanou polynomiální matici `poly`. Navíc změní znaménka u lichých mocnin.

3.9.8 Předefinované standardní funkce

Pro nové polynomiální objekty bylo předefinováno několik standardních funkcí, jejichž význam je zachován. Předefinované funkce se nejčastěji aplikují na koeficienty skalárních polynomů nebo polynomiálních matic. Jmenovitě se jedná o tyto funkce: `Transpose`, `SetPrecision`, `Rationalize`, `Dimensions`, `RealQ`, `IntegerQ`, `NumericQ`, `SquareQ`, `PolynomialQ`.

Všechny funkce jsou popsány v nápovědě, kde jsou také uvedeny příklady použití.

3.10 Globální proměnné

Globální proměnné uchovávají důležité informace, které jsou automaticky používány v mnoha funkcích polynomiálního balíčku.
Volají se hlavně v těchto situacích:

- Jestliže vstupní polynomiální objekt je uveden bez proměnné a nelze ji automaticky určit, pak je použita hodnota z globální proměnné `$Variable`.
- Při výpočtech s reálnými (nepřesnými) čísly je vždy volána funkce nulování `Chop[polyobj]`. Hodnota nulovací tolerance je upravována globální proměnnou `$Zeroing`.
- Výstup polynomiálního objektu může být zobrazován různými způsoby. Standardně jsou zobrazovány ve tvaru matice s polynomy. Další možný výstup je ve tvaru tabulky matic koeficientů polynomiální matice. Formát výstupu je řízen globální proměnnou `$Format`.
- Globální proměnnou `$Verbose` lze řídit podrobný výpis zpráv, který je generován během výpočtů.

Všechny globální proměnné začínají znakem `$var`, stejně jako u většiny globálních proměnných v systému.

Seznam všech globálních proměnných a jejich význam je popsán v tabulce 3.9.

Tabulka 3.9: Globální proměnné

globální proměnná	**význam**	**standardní hodnota**	**další možné hodnoty**
`$Variable`	implicitně použitý symbol proměnné	`s`	libovolný symbol
`$Zeroing`	tolerance použitá pro nulování reálných čísel	`10.^-10`	libovolné kladné reálné číslo
`$Format`	způsob zobrazení polynomiálního objektu	`"Nice"`	`"CoefTable"`, `"None"`
`$Verbose`	podrobné zprávy v průběhu výpočtu	`False`	`True`

Pro jednoduší předefinování globálních proměnných bez nutnosti znalosti jejich názvu byla vytvořena funkce `PolyGlobal[val]`, která sama správně přiřadí hodnotu *val* k příslušné globální proměnné. Pokud je `PolyGlobal[]` uvedena bez argumentu vytiskne seznam všech globálních proměnných a jejich aktuálních hodnot.

```
In[46] :=   PolyGlobal[]

Out[46] =

          $Format = "Nice"
          $Variable = s
          $Zeroing = 10.^-10
          $Verbose = False
```

Globální proměnné lze měnit snadno přiřazením

```
In[47] :=   $Variable = x;
```

nebo pomocí funkce `PolyGlobal[x]`, která sama rozpozná k jaké proměnné se váže daná hodnota a změní ji.

Nyní je nastaven symbol `x` jako standardní proměnná vstupní polynomiální matice, která nemá určenou proměnnou.

```
In[48] :=
```

$$\mathtt{a} = \mathtt{PMC}[\{\begin{pmatrix} 1 & k \\ 0 & -2 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 3 & 7 \end{pmatrix}\}]$$

```
Out[48] =
```

$$\begin{pmatrix} 1 & k-x \\ 3\,x & -2+7\,x \end{pmatrix}_x$$

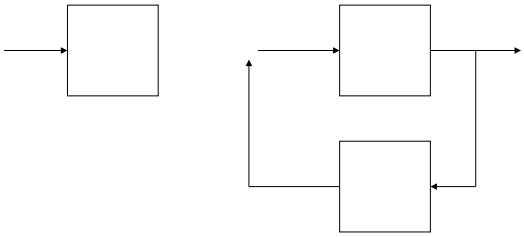
Výstupní formát pro polynomiální objekty ve tvaru `"CoefTable"` vypadá následovně.

```
In[49] :=   $Format = "CoefTable";        PolyGlobal["CoefTable"]
             a
Out[49] =

                    1    2

               1    1    k
          0
               2    0    -2


               1    0    -1
          1
               2         7
```

Poslední možný formát je `"None"`. Polynomiální objekty jsou zobrazovány bez jakékoliv grafické úpravy ve tvaru funkcí `PolyMat` a `Poly`. K původnímu grafickému formátu se vrátíme příkazem `PolyGlobal["Nice"]` nebo `$Format="Nice"`.

3.11 Příklad použití při návrhu řízení

Tento jednoduchý příklad ilustruje použití funkce `DESolve` při návrhu řízení. Zadáním úlohy je navrhnout regulátory R_1 a R_2 tak, aby přenos celého zpětnovazebního obvodu byl přesně roven zvolenému modelu M. V literatuře se tento problém také



$$q\,(\,10\,)_s + p\,(\,40 + 50\,s + 10\,s^2\,)_s == \\ (\,25\,x0 + s\,(10\,x0 + 25\,x1) + s^2\,(x0 + 10\,x1) + s^3\,x1\,)_s$$

Příkazem `DESolve` najdeme řešení Diofantické rovnice `eq`.

```
In[51] :=   sol = DESolve[eq]
Out[51] =
```

$$\{q \to (\tfrac{1}{10}\,(21\,x0 - 20\,x1) + s\,\tfrac{1}{10}\,(5\,x0 - 4\,x1))_s, \\ p \to (\tfrac{1}{10}\,(x0 + 5\,x1) + s\,\tfrac{1}{10}\,x1)_s\}$$

Volbou parametrů $x_0 = 1, x_1 = 1$ je zachována kauzalita regulátoru. Po dosazení do předešlého výsledku získáme konečné řešení.

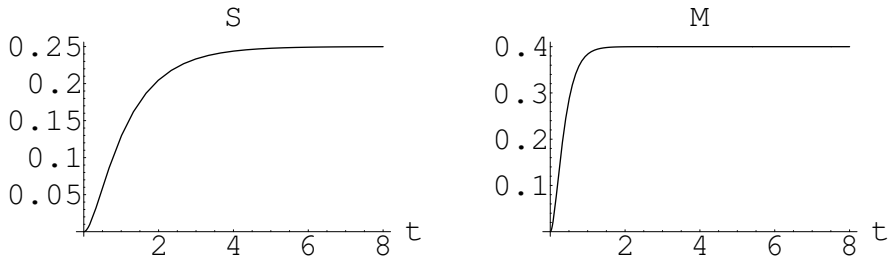
```
In[52] :=   x = {x0- 1, x1- 1};
   {sol /. x,
    r /. x}
Out[52] =
```

$$\{\{q \to (\tfrac{1}{10} + \tfrac{1}{10}\,s)_s,\ p \to (\tfrac{3}{5} + \tfrac{1}{10}\,s)_s\}, (1 + s)_s\}$$

Hledané regulátory mají přenos

$$R_1 = \frac{\frac{1}{10} + \frac{1}{10}\,s}{\frac{3}{5} + \frac{1}{10}\,s} \qquad \text{a} \qquad R_2 = \frac{1 + s}{\frac{3}{5} + \frac{1}{10}\,s}.$$

Na obrázku 3.10 jsou porovnány přechodové charakteristiky samotného systému S a systému doplněného regulátorem s požadovaným přenosem M.



Obrázek 3.10: Přechodová charakteristika soustavy.

Kapitola 4

Implementace

Na začátku kapitoly je popsána struktura vytvořených souborů jednotlivých funkcí a jejich instalace do systému. V další části jsou popisovány výhody nově vytvořených polynomiálních objektů společně s ukázkami programového kódu.

4.1 Struktura programových souborů a instalace

Všechny vytvořené programové soubory jsou umístěny v adresáři `Polynomial`, který je při instalaci uložen do libovolného adresáře, na který odkazuje globální systémová proměnná `$Path`. Standardně se jedná o adresář `\AddOns\ExtraPackages`. V nové verzi *MATHEMATICA 5* je nutné instalaci provést do adresáře, který je pevně nastaven v systémové globální proměnné `$BaseDirectory` nebo `$UserBaseDirectory`. Pokud se složka `Polynomial` nakopírujeme do jiného adresáře, pak nebude možné přistupovat k nové nápovědě v okně Help Browser.

Pro přehlednost jsou jednotlivé funkce převážně rozděleny do samostatných souborů pod názvem funkce, kterou definují.

V adresáři `Polynomial` jsou kromě programových souborů tři adresáře a to `Kernel`, `Documentation` a `FrontEnd`. První z nich, adresář `Kernel`, obsahuje jediný soubor `Init.m`, který deklaruje všechny nové funkce, jež se v případě potřeby načtou do paměti systému. Jestliže použijeme standardní příkaz pro zavedení programového balíku `<<Polynomial‘` nebo `<<Polynomial‘Master‘`, pak právě soubor `Init.m` se zavádí do systému jako první. Adresář `Documentation` obsahuje soubory s dokumentací, která se automaticky po provedení příkazu Rebuild Help Index (menu Help) doplní do nápovědy systému. Poslední adresář `FrontEnd` obsahuje předpřipravené palety pro snazší práci. Ve verzi *MATHEMATICA 5* jsou všechny nové palety z tohoto adresáře automaticky přidány do menu File, Palettes.

4.2 Polynomiální objekty

Podobně jako v nástavbě *CONTROL SYSTEM PROFESSIONAL* jsou definovány nové objekty (`TransferFunction[]` pro přenosovou funkci, `StateSpace[]` stavový popis

systému a další), tak i my jsme byli z praktických důvodů donuceni vytvořit nové objekty.

Celkem jsou definovány dva polynomiální objekty[1]. Jedná se o funkce `PolyMat[ ]` a `Poly[ ]`, které definují polynomiální matici a skalární polynom. Jejich primárním úkolem je spojení dat polynomiální matice s proměnnou a stupněm polynomiální matice, podobně jako v *CSP*.

Objekty se vytváří pomocí funkcí `PM[ ]`, `PMC[ ]` a `PLC[ ]` pro polynomíální matice a `P[ ]`, `PC[ ]` pro skalární polynomy, které byly podrobně popsány v kapitole 3.1. Výstupem těchto funkcí je právě funkce (objekt) `PolyMat[]` nebo `Poly[]`.

Argumenty funkce `PolyMat[`*`coe,var,deg`*`]` jsou přesně definovány. Jestliže je dána polynomiální matice $A(s) = A_0 + A_1 s + ... + A_{d_A} s^{d_A}$, pak argumenty objektu polynomiální matice jsou následující:

$coe = \{A_0, A_1, \ldots, A_{d_n}\}$	seznam matic jednotlivých mocnin
$var = s$	proměnná
$deg = d_A$	stupeň polynomiální matice.

Podobně pro objekt `Poly[`*`coe,var,deg`*`]` skalárního polynomu $a(s) = a_0 + a_1 s + ... + a_{d_a} s^{d_a}$ jsou argumenty následující:

$coe = \{a_0, a_1, \ldots, a_{d_a}\}$	seznam koeficientů polynomu
$var = s$	proměnná
$deg = d_a$	stupeň polynomiální matice.

4.2.1 Výhody objektového přístupu

Testy, zda je vstup polynomiální matice nebo skalární polynom v dané proměnné, se provádí pouze u vytvořujících funkcí. Test se tedy provádí pouze jedenkrát při zadání polynomiálního objektu. V těle nových funkcí se již pracuje pouze s objektem `PolyMat[]` nebo `Poly`, který se netestuje.

Všechny implementované funkce pracují pouze s novými objekty. Nyní definice nové funkce vypadá následovně.

```
1  NovaFunkce[
2     A: PolyMat[pmcA_, varA_Symbol, degA : _Integer | -Infinity ],
3     B: PolyMat[pmcB_, varB_Symbol, degB : _Integer | -Infinity ],
4     ...] := tělo funkce;
```

V těle nové funkce je proměnným `A`, `B` přiřazen pouze polynomiální objekt `PolyMat` a dále k proměnným `pmcA`, `varA`, `degA`, `pmcB`, `varB`, `degB` jsou přiřazeny konkrétní data vstupních objektů.

Funkce `PolyMat[pmcA_, varA_Symbol, degA : _Integer | -Infinity ]` reprezentuje vzor polynomiálního objektu.

[1] Slovo objekty nelze v systémy *MATHEMATCA chápat tak jako ve vyšších programovacích jazycích. Zde se jedná o jakousi abstrakci, která je tvořena pomocí různých definic funkcí a jejich argumentů.*

Často se opakující zápis vzorů objektů v těle definic argumentů nových funkcí je nahrazen kratším zápisem pomocí funkce `PolyMatObj[ ]`, která je definována následujícím způsobem.

```
5  PolyMatObj[pmcA_, varA_, degA_] =
6        PolyMat[pmcA_, varA_Symbol, degA : _Integer | -Infinity ]
```

Stejnou funkci lze též zapsat jednodušeji.

```
7  NovaFunkce[
8     A: PolyMatObj[pmcA, varA, degA],
9     B: PolyMatObj[pmcB, varB, degB], ...] := tělo funkce;
```

Další velkou výhodou v zavedení polynomiálních objektů je snadné předefinovávání standardních funkcí. Například funkce `Plus[]` byla snadno předefinována následujícím způsobem.

```
10  PolyMat /: Plus[
11     A:PolyMat[pmcA_, varA_Symbol, dega: _Integer | -Infinity ],
12     B:PolyMat[pmcB_, varB_Symbol, degb: _Integer | -Infinity ]
13             ] :=
14     Block[{sol, var, deg, tol, ...
15                   ⋮
16        Chop[
17           PolyMat[sol, var, deg]
18           , me   tol
19        ]
20     ] /; If[Size[A] === Size[B], True,
21                Message[General::plusdim]; False];
```

Zde je podstatné použití funkce `TagSet` na objekt `PolyMat` ve funkci `Plus`. Konstrukce na řádcích 10 – 13. Tím je předefinováno sčítání pro objekt `PolyMat`. Na řádku 20 je navíc použita podmínka (funkce `Condition`, zkráceně `/;`) pro sčítání. Důležité je uvést podmínku na konci definice funkce a ne za závorkou na řádku 13, kde se často umísťuje. V případě předefivávání by funkce nefungovala korektně.

Pokud by nebyl zaveden objekt a používala by se standardní struktura matice ve formě `List`, předefinování by nebylo možné, neboť funkce `List` je typu `Protect` a `Locked`. Funkci `TagSet` ani `Unprotect` nelze tedy použít.

Použití dalších nástrojů pro předefinování standardních funkcí, funkce `Unprotect[ Plus]`, `Protect[Plus]`, se v tomto případě, ale i u dalších důležitých interních funkcí (`Dot`, `Times` nebo `Det`), ukázalo nefunkční. Operace neprobíhaly zcela korektně, nedodržovalo se vyhodnocování podmínek.

U jednodušších standardních funkcí je ale konstrukce funkcí `Unprotect[]` a `Protect[]` používána. Důvodem je, aby nedocházelo k zbytečnému „přetěžování“ definice polynomiálního objektu používáním funkce `TagSet`. Předefinování pro polynomiální objekt vypadá následovně.

```
22 Unprotect[Transpose];
23    Transpose[pol:PolyMat[pmc,var,deg]] :=
24              PolyMat[Transpose/ pmc, var, deg];
25 Protect[Transpose];
```

Zavedení nových polynomiálních objektů by se mohlo zdát jako zbytečné, protože *MATHEMATICA* dobře podporuje práci s polynomy, respektive maticemi, na jejichž pozicích jsou polynomy. Pokud ale bychom polynomy nechali ve standardním tvaru výrazu[2] a prováděli s nimi výpočty, jednalo by se o symbolickou metodu výpočtu, jejíž zásadní nevýhody byly popsány v kapitole 1.2.2 a všech srovnávacích testech rychlosti výpočtů v kapitole 3. Maximálního urychlení výpočtů lze dosáhnout pouze díky rychlým operacím[3] a vhodným numerickým algoritmům nad koeficienty, které jsou tvořeny vektory nebo maticemi čísel. Vektory a matice nemusí být vždy nutně číselné, mohou obsahovat symboly, ale způsob výpočtu budeme považovat za numerický. To je dáno schopností některých funkcí provádět výpočty i se symbolickými daty.

Nyní je rozebrán problém se zbytečným testováním, pokud by nebyly vytvořeny nové objekty. Kdybychom použili standardního postupu pro polynomy ve tvaru symbolických výrazů, definice by vypadala následovně.

```
26 PolynomialMatrixQ =
27 Block[var,
28    If[MatrixQ[#],
29       var = Variables[#];
31          is polynomial,
32       And    Flatten[Map[PolynomialQ[#, var] &, #, 2]],
33          not matrix
34       False
35      ]
36 ]&;
37
38 Funkce[
39    A:_PolynomialMatrixQ,
40    B:_PolynomialMatrixQ, ...] :=
41        ...
42            PomocnaFunkce[A];
43        ...
44     ;
45
46 PomocnaFunkce[ A:_PolynomialMatrixQ ] :=   ...   ;
```

[2] Symbolický výraz.

[3] Funkce pro řešení soustavy lineárních rovnic `LinearSolve`, maticové násobení konstantních matic `Dot`, vhodné funkce pro operace se seznamy prvků.

Test `PolynomialMatrixQ` při volání pomocné funkce na řádku 39 během výpočtu se provádí zcela zbytečně, neboť byl již jednou proveden při testování argumentů. Testování by se provádělo zbytečně i s maticemi, které jsou výsledkem některé funkce, jež zaručeně vrátila polynomiální matici.

Při použití našich objektů se provede test pouze jedenkrát a to při zadávání polynomu nebo polynomiální matice. Tím v průběhu výpočtu odpadá zdlouhavé testování jednotlivých funkcí, zda jde o polynomiální matici či nikoliv.

Definice objektů se nachází v souboru `Object.m`.

4.2.2 Polynomiální matice – `PM[ ]`

Funkce `PM[`*`mat, var`*`]` vytvoří z matice *mat*, jejíž prvky tvoří skalární polynomy ve tvaru výrazu nebo objekty typu skalární polynom `Poly[]`, objekt polynomiální matice. Druhý argument *var* určuje proměnnou vstupní polynomiální matice. Pokud je vynechán, vyhledá se proměnná automaticky. V případě, že je nalezeno více symbolů a není uvedena proměnná, je generováno chybové hlášení a vrácena hodnota `Null`.

Implementace funkce vypadá následovně.

```
47      hlavní definice
48  PM[pm_ MatrixQ, var_Symbol] /; PMTest[pm,var] :=
49  Module[{plc = Map[CoefficientListZeroingLast[#, var]&,
            pm, {2}], maxdeg, padnum},
50     maxdeg = Max[Map[Length, plc, {2}]];
51     padnum = If[And    ExactNumOrSymbolInExprQ /
52                   Hold    Flatten[plc], 0, 0.];
53     If[maxdeg =!= 0,
54        PolyMat[
55           Transpose[
56                 Map[PadRight[#, maxdeg, padnum]&, plc, {2}],
57                 {2, ,1}
58              ]// ToPackedArray,
59           var, maxdeg-1
60        ],
61        PolyMat[
62           {Array[padnum&, Dimensions[pm]]}//ToPackedArray,
63           var, -Infinity
64        ]
65     ]
66  ];
67
68      další možné tvary
69  PM[pm_ MatrixQ] := PM[pm, GetVariable[pm]];
70  PM[pm_ MatrixQ, _ErrVar] := Null;
71
```

```
72      konverze PolyMat -  PolyMat, Poly[] -  PolyMat[]
73  PM[pol_PolyMat] := pol;
74  PM[pol:Poly[pc_,var_,deg_]] := PolyMat[{{#}}& /  pc, var, deg];
75
76      ostatn´ je chybn´ zad n´
77  PM[___] :=  Message[General::notpolyformat]; Null ;
```

Funkce `ToPackedArray` (řádek 62) zkouší převést seznam matic koeficientů v objektu polynomiální matice na `PackedArray`. Význam této funkce je rozebrán v kapitole o implementačních poznámkách 5.1.

Test, který určuje, zda prvky vstupní polynomiální matice jsou polynomy, se provádí funkcí `PMTest[ ]`.

```
78  PMTest[mat_,var_] :=
79      If[Global‘$Verbose, PrintVerbose["Tests if input is PM."]];
80      If[  And      PolynomialQ[#, var] || MatchQ[#,  _Poly] &
81                                      /  Hold    Flatten[mat] ,
82         True,
83         Message[General::notpoly, Short[mat]];
84         False
85      ]  ;
```

4.2.3 Matice koeficientů polynomiální matice – `PLC[ ]`

Vstupem je matice, na jejichž pozicích jsou seznamy koeficientů skalárního polynomu.

```
86      hlavn´ definice
87  PLC[plc_ PLCoefficientsQ, var:_Symbol:Global‘$Variable]
88                          /; PLCTest[plc,var] :=
89  With[{maxdeg = Max[Map[Length[DropLastZeros[#]]&, plc, {2}]],
90        padnum = If[And    ExactNumOrSymbolInExprQ
91                           /  Hold    Flatten[plc], 0, 0.]},
92     If[maxdeg =!= 0,
93        PolyMat[
94           Transpose[
95              Map[PadRight[#, maxdeg, padnum] &, plc, {2}]
96              ,{2,  , 1}
97           ]//ToPackedArray, var, maxdeg-1
98        ],
99        PolyMat[Array[p&, Dimensions[plc]], var, -Infinity]
100    ]
101 ];
102
103     konverze PolyMat -  PolyMat, Poly -  PolyMat
```

```
104  PLC[pol_PolyMat] := pol;
105  PLC[pol:Poly[pc_,var_,deg_]] := PolyMat[{{#}}&/ pc, var, deg];
106
107      ostatn´ı je chybn´e zad n´ı
108  PLC[___] :=  Message[General::notpolyformat]; Null ;
```

Test, zda vstupní matice `plc` odpovídá koeficientům polynomů, se provádí ve funkci `PLCTest[ ]`. Testuje se pouze, zda vstup neobsahuje proměnnou polynomiální matice.

```
109  PLCTest[plc_,var_] :=
110        If[Global‘$Verbose, PrintVerbose["Tests if inputis PLC"]];
111        If[Intersection[Variables[plc],{var}] === {var},
112           Message[General::errcoe,var]; False,
113           True
114        ]
115   ;
```

Implementace ostatních funkcí `PMC`, `P` a `PC` pro zadávání polynomiálních objektů je podobná jako v předešlých ukázkách pro funkce `PM` a `PLC`.

Kapitola 5

Implementační poznámky

Hlavní prioritou při programování této sady funkcí byla rychlost výpočtů. Tomu byl také podřízen styl programování, který by se na první pohled mohl zdát zbytečně složitý a komplikovaný, ale postupným rozšiřováním a testováním se ukázalo nutné provést jisté změny v programování. Potřeba předefinovávání standardních funkcí a omezení provádění zbytečných a zdlouhavých testů vedla k vytvoření nových polynomiálních objektů. V některých případech se pro urychlení výpočtu používá speciální interní funkce `Complile`.

Dále v textu následuje několik kapitol, v kterých jsou popsány některé implementační problémy a jejich řešení nebo vylepšení. Jako zdroj základních informací posloužily publikace [20, 16]. Mnoho dalších užitečných rad lze také nalézt na různých internetových stránkách [4]. Velmi cenný byl především dokument [23, 24]. V neposlední řadě se také osvědčila internetová diskuzní skupina zabývající se systémem *MATHEMATICA* [25].

Tyto poznámky jsou zde také uváděny z důvodu nedostatečného popisu a dokumentace daných problémů v elektronické nápovědě k systému *MATHEMATICA*. Jedná se o obecné problémy, které by se mohly hodit i v jiných pracech.

5.1 Seznamy typu `PackedArray`

`PackedArray` je efektivní formát interní reprezentace dat [24]. Seznam prvků (vektory, matice, vícerozměrná pole) tvořený jedním typem čísel (`Real`, `Integer` nebo `Complex`) může být interně reprezentován v tomto formátu. Pro uživatele je interní reprezentace dat seznamů typu `PackedArray` v podstatě neviditelná (běžný uživatel nerozpozná rozdíl mezi běžným seznamem a seznamem v reprezentaci `PackedArray`). Funkce pro práci s `PackeddArray` je dostupná pouze v kontextu `Developer‘`. Dobré porozumění reprezentaci seznamů může pomoci při psaní rychlých procedur s menšími paměťovými nároky.

5.1.1 Reprezentace seznamů dat – `List[ ]`

Jestliže zadáme seznam čísel,

```
In[50] :=   a = {1,2, ,,};
Out[50] =

          {1, 2, ,,}
```

pak se jeho vnitřní reprezentace liší od výstupního `OutputForm` zobrazení. Základní reprezentace seznamů je ve tvaru funkce `List[1,2, ,,]`, kterou zobrazíme pomocí funkce `FullForm[a]`. Každý prvek v seznamu je reprezentovaný jako výraz, který si s sebou nese informaci o svém typu (v tomto případě celé číslo `Integer`).

Pokud vložíme seznam odlišným způsobem,

```
In[51] :=   b = Range[1,,];
Out[51] =

          {1, 2, ,,}
```

pak je vnitřní reprezentace dat odlišná. Základní výstup je stejný s předešlým výstupem, dokonce i funkce `FullForm[b]` vrací stejný výstup. Pro všechny operace se jedná o stejná data. Samozřejmě platí test rovnosti `a === b`.

Odlišnosti v reprezentaci zjistíme pouze pomocí speciální funkce, která je dostupná pouze v kontextu `Developer‘`. Použitím funkce

```
In[52] :=   <<Developer‘                nahraje kontext Developer‘
            PackedArrayQ /  {a,b}       Developer‘PackedArrayQ
Out[52] =

          {False, True}
```

zjistíme rozdíl v reprezentaci dat.

5.1.2 Výhody reprezentace `PackedArray`

Mezi hlavní výhody reprezentace dat typu `PackedArray` patří větší rychlost výpočtů se seznamy a menší paměťové nároky. Rychlost výpočtu ilustruje následující jednoduchý příklad.

```
In[53] :=   a = Range[10^ ];
            b = a; b[[1]] = 1.;          unpack packed array

            {bc = ByteCount /  {a,b}, N[bc[[2]]/bc[[1]]]}

            {
              timpa  = Timing[a+a],
```

```
    timupa = Timing[b+b],
    timupa[[1,1]]/timpa[[1,1]]
  }
```

```
Out[53] =

        {{20000 2, , 000 6},  }

Out[54] =

        {{0.01 Second, Null}, {0.29 Second, Null}, 29.0}
```

Proměnná `b` nemůže být `PackedArray`, neboť všechny prvky seznamu nejsou stejného typu (`Integer`). Na první pozici v seznamu `b` je dosazeno reálné číslo 1.. Data v paměti v případě proměnné `a` zabírají 5krát méně místa a při operaci sčítání $x+x$ je rychlost výpočtu 29krát rychlejší.

Tak velký rozdíl v rychlosti výpočtu je způsoben použitím různých interních funkcí. Pro práci se seznamy typu `PackedArray` je volána rychlá interní funkce, která se podobá C-programu pracujícímu s dvěma poli.

V druhém případě je seznam `b` uložen jako funkce `List[...]`. Použitá funkce `Plus` jako první krok udělá spárování (`Thread`) jednotlivých prvků seznamu přibližně podle schématu

```
{b[[1]]+b[[1]], b[[2]]+b[[2]], ....}.
```

Funkce `Plus` zpracovává `10^` výrazů. Pro každou dvojici čísel sleduje informace o typu čísla, resp. výrazů (tj. testuje, zda je celé nebo reálné číslo, nebo třeba opět seznam) a podle toho určuje, jaká funkce se použije k aktuálnímu sečtení. Tento postup se všemi jeho testy je příčinou časového nárůstu výpočtu.

5.1.3 Funkce pracující s `PackedArray`

Většina funkcí s atributem `Listable` zachovává typ `PackedArray` vstupního seznamu. Další jsou funkce `Join`, `Transpose`, `Part` a `Flatten`, které upravují strukturu seznamu.

Seznam typu `PackedArray` vždy vrací funkce `Range`, neboť výstupem je vždy seznam dimenze 1 (vektor) s celočíselnými prvky. Podobně funkce `Fourier`, která vždy vrací seznam (ne nutně vektor) reálných čísel. Seznam typu `PackedArray` se nevrací pouze tehdy, když jsou na vstupu reálná čísla s přesností jinou než `MachinePrecision`, protože výstup si zachovává danou přesnost a to není pro `PackedArray` přípustné (seznam může být typu `PackedArray` pouze v tom případě, když je přesnost `MachinePrecision`).

Funkce `Map`, `Table` a `Array` generují `PackedArray` pouze od jistého čísla, které je nastaveno v systémových volbách `SystemOptions`.

```
"CompileOptions" -  {"MapCompileLength" -  100,
                     "TableCompileLength" -  2 0,
                     "ArrayCompileLength" -  2 0, ...}
```

Podle těchto hodnot platí:

```
In[55] :=   a = Table[0,{200}];
            PackedArrayQ[a]
Out[55] =

        False
```

Změníme-li volbu `TableCompileLength`, pak dostaneme výsledek.

```
In[56] :=   SetSystemOptions[
                "CompileOptions" -  "TableCompileLength" -  200];
            a = Table[0,{200}];
            PackedArrayQ[a]
Out[56] =

        True
```

Samozřejmě výstup typu $\{sym, sym, \ldots\}$ nemůže být typu `PackedArray`.

```
In[57] :=   a = Table[sym,{10}];
            PackedArrayQ[a]
Out[57] =

        False
```

Podobně si musíme dát pozor při používání funkce `Map`.

```
In[58] :=   SetSystemOptions["UnpackMessage" -  True];
            a =  #^2& /  Range[10]
            PackedArrayQ[a]

        FromPackedArray::"punpack1" : Unpacking array to level 1.

Out[58] =

        {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}

Out[59] =

        False
```

Změny, kdy je `PackedArray` převáděno do standardní reprezentace seznamu, lze sledovat po nastavení volby `SetSystemOptions["UnpackMessage" -  True]`. Od této chvíle je při každém převodu generováno varovné hlášení.

Další důležité funkce, které pracují se seznamy, jsou `ToPackedArray`, `FromPackedArray` a `PackedArrayForm`. Jejich použití je z názvů zřejmé.

```
In[60] :=   a = {1,2, ,,};
            a = ToPackedArray[a];
            PackedArrayQ[a]
Out[60] =

          True
```

Převede standardní seznam na typ `PackedArray`.

5.1.4 Polynomiální objekty a `PackedArray`

Z předcházejících ukázek je patrné výrazné zrychlení výpočtu pokud se pracuje se seznamy typu `PackedArray`. Všechna data polynomiálních objektů (koeficienty polynomiální matice) jsou proto při jejich konstrukci (funkce `PM`, `PMC`, `PLC`, `P` a `PC`) převáděny na typ `PackedArray` pomocí funkce `ToPackedArray`. Převod se uskuteční jen v případě, pokud je to možné, tj. koeficienty polynomiálních matic jsou číselné a stejného typu (`Integer`, `Real` nebo `Complex`).

Dále jsou během výpočtů používány funkce, které zachovávají seznamy koeficientů polynomiálních objektů ve tvaru `PackedArray`.

5.2 Určení typu koeficientů – `IntegerQ[ ]`, `RealQ[ ]`

Funkce `IntegerQ[`*pol*`]` vrací logickou hodnotu `True`, jestliže všechny koeficienty polynomiální matice v polynomiálním objektu *pol* jsou celá čísla (`Integer`), jinak vrací hodnotu `False`.

Funkce `RealQ` vrací hodnotu `True`, jestliže všechny koeficienty objektu *pol* jsou reálná čísla (`Real`) s přesností `MachinePrecision`, jinak vrací hodnotu `False`.

Tyto funkce se vždy volají v procedurách, kde je potřeba doplnit ke koeficientům nějaké číslo (nejčastěji nulu) a to takovým způsobem, aby po přidání čísla nedošlo ke ztrátě `PackedArray` typu seznamu. Jestliže tedy chceme rozšířit seznam `{1,2, }`, který je `PackedArray`, zprava o dvě nuly a zachovat typ seznamu `PackedArray`, pak musíme doplnit číslo 0. Pokud bychom použili číslo 0., pak by nový seznam nebyl typu `PackedArray`, neboť by v seznamu byly dva různé typy čísel (`Integer` a `Real`). Analogicky pro seznamy s reálnými čísly.

Varianta 1. `IntegerQ /`

Jako první a asi nejpřirozenější se jeví tento způsob implementace

```
In[61] :=      seznam n hodn ch real. sel, typu PackedArray
           dat = Table[Random[],{10^ }];

           And    IntegerQ /  dat   //Timing
Out[61] =
```

```
{0.2 Second, False}
```

Jak je vidět, časová náročnost výpočtu je značná.

Varianta 2. `Hold`

Vylepšení předchozí varianty by mohlo vypadat následovně.

```
In[62] :=   And   IntegerQ /  Hold   dat   //Timing
Out[62] =

{0.20 Second, False}
```

Urychlení výpočtu se nejeví jako zdařilejší. Přitom jsme použili aplikaci funkce `Hold`, která zamezí vyhodnocování prvků v seznamu po první nalezené hodnotě `False`, v našem příkladě se jednalo hned o první prvek seznamu `dat`.

Tato konstrukce vychází časově lépe o proti první variantě v případě, že požadovaný test je složitější a jeho zamítnutí nastane na začátku seznamu.

Varianta 3. `PackedArrayQ`

Pokud předpokládáme, že vstupní data (seznamy) by měly být typu `PackedArray`, pak této informace lze dobře využít. V případě polynomiálních objektů jsou všechna data (seznamy koeficientů) převáděna na typ `PackedArray`.

Pokud předpokládáme, že vstup musí být typu `PackedArray`, pak k testu stačí použít jedinou funkci.

```
In[63] :=   PackedArrayQ[dat, Integer]   //Timing
Out[63] =

{0. Second, False}
```

Rychlost výpočtu je nejlepší, takřka vždy nulová.

Výsledná procedura pro objekt polynomiální matice vypadá následovně.

```
1  Unprotect[IntegerQ];
2  IntegerQ[A:PolyMatObj[pmcA,var,deg]] := PackedArrayQ[pmcA,Integer];
3  Protect[IntegerQ];
```

Funkce je podobně definována i pro objekt `Poly`. Varianta testu na reálná čísla používá funkci `PackedArrayQ[pmcA,Real]`.

5.3 Různé způsoby vytváření konstantních matic

Ve funkci `SylvestrMatrix` se používá předem připravená matice s nulovými prvky. Následující varianty popisují různé způsoby, jak lze takovou matici vytvořit.

Varianta 1. `Table`

Nejpřirozenější způsob je použít funkci `Table`.

```
In[64] :=   Timing[  mat = Table[0., {1000}, { 00}];  ]
Out[64] =

        {0.1  Second, Null}
```

Vytvořená matice je rozměru $[1000 \times 500]$ a prvky matice jsou 0.. Čas konstrukce výsledné matice `mat` je poměrně značný.

Varianta 2. `Array`

Stejný výsledek dává také funkce `Array`. První argument funkce `Array` musí být ve tvaru funkce. Jestliže tedy požadujeme nulovou matici, pak funkce má tento tvar `0.&`.

```
In[65] :=   Timing[  mat = Array[0.&, {1000, 00}];  ]
Out[65] =

        {0.16 Second, Null}
```

Čas konstrukce výsledné matice `mat` je podobný předešlé variantě.

Varianta 3. `SparseArray`

V nové verzi *MATHEMATICA 5* přibyla funkce pro řídké vícerozměrné matice (seznamy). V kombinaci s příkazem `Normal` ji využijeme pro konstrukci matice.

```
In[66] :=   Timing[ mat =
               Normal[ SparseArray[{1,1} -  0., {1000,  00}, 0.] ];
            ]
Out[66] =

        {0.01 Second, Null}
```

První argument `{1,1} - 0.` funkce `SparseArray` definuje první prvek řídké matice na pozici $\{1, 1\}$. Druhý argument určuje rozměr matice a poslední říká, jaké jsou prvky matice mimo pozici $\{1, 1\}$. Z takto vytvořené řídké matice již snadno pomocí funkce `Normal` vytvoříme normální tvar matice, dokonce ve tvaru `PackedArray`.

Čas konstrukce výsledné matice `mat` je v této variantě nejrychlejší a to 16krát.

5.4 Konstrukce Sylvesterovy matice

Při mnoha výpočtech se volá funkce, která sestrojí z dané vstupní polynomiální matice Sylvesterovu matici daných rozměrů.

Varianta 1. `RotateRight`

Z tvaru Sylvesterovy matice S_A (3.2) by nás mohla napadnout následující implementace, která vychází z rotace prvního sloupce matice S_A.

```
1  SylvestrMatrix0[A:PolyMatObj[pmc, var, deg],
2            col_?((Positive[#] && IntegerQ[#]) &)] :=
3  Module[{mat, m, n, prepnum, zeromatrix, sol},
4     {m, n} = Size[A];
5     prepnum = Which[RealQ[A], 0., IntegerQ[A], 0, True, 0];
6     zeromatrix = If[col === 1, Return[Flatten[pmc, 1]],
7                     PrepareMatrix[m*(col - 1), n, prepnum]];
8     mat = Join[Flatten[pmc, 1], zeromatrix];
9     sol = {mat};
10    Do[
11       AppendTo[sol, RotateRight[mat, i*m]]
12       , {i, col - 1}
13    ];
14    MapThread[Join, sol]
15 ];
```

Na řádku 232 se podle typu vstupní matice `A` zjišťuje, jaká nula (0 nebo 0.) bude použita k rozšíření o nulové koeficienty. Funkce `Join` na řádku 235 vytvoří první sloupec Sylvesterovy matice. Dále se v cyklu provádí rotace prvního sloupce a přidává se do seznamu `sol`. Na řádku 241 jsou pomocí funkce `MapThread` a `Join` pospojovány všechny vytvořené sloupce ze seznamu `sol` do jediné výsledné matice.

Časová náročnost je následující.

```
In[67] :=   A = PMRandom[10,20]     n hodn  matice [20x20] stup. 10
            Timing[  SylvestrMatrix0[A, , 0]; ]          , 0 sloupc
Out[67] =

        {0. 2 Second, Null}
```

Čas výpočtu není nejkratší. Jeden z důvodů je poměrně složitá práce se seznamy. Jiná, rychlejší implementace je následující.

Varianta 2. `Part`

Konstrukce probíhá ve dvou krocích. Nejprve je naráz připravena nulová matice rozměrů výsledné Sylvesterovy matice (řádek 252). Do této matice je na odpovídající pozice v cyklu dosazována (funkce `Part`) matice koeficientů vstupní polynomiální matice (řádek 249).

```
16 SylvestrMatrix[A:PolyMatObj[pmcA,var,adeg],
17           col_?((Positive[#] && IntegerQ[#]) &)] :=
18 Module[{m, n, mat, flatpmcA, prepnum,
```

```
19          deg = If[adeg === -Infinity, 0, adeg]},
20    {m, n} = Size[A];
21
22    flatpmcA = Flatten[pmcA, 1];
23    If[col === 1, Return[flatpmcA]];
24    prepnum = Which[RealQ[A], 0., IntegerQ[A], 0, True, 0];
25    mat = PrepareMatrix[m*(col + deg), col*n, prepnum];
26
27    Do[
28       mat[[
29          Range[(i - 1)*m + 1, (i + deg)*m ],
30          Range[(i - 1)*n + 1, i*n]
31       ]] = flatpmcA
32       , {i, col}
33    ];
34
35    mat
36 ];
```

Časová náročnost konstrukce Sylvesterovy matice je nyní skoro 6krát rychlejší. Tento způsob implementace, na rozdíl od předchozího, zachovává seznamy ve tvaru `PackedArray`.

```
In[68] :=   Timing[  SylvestrMatrix[A, 0]; ]          (* 0 sloupců *)
Out[68] =

          {0.06 Second, Null}
```

5.5 Funkce `NonZeroMin[`*list*`]`

V systému neexistuje žádná rychlá interní funkce, která by vracela nenulovou minimální hodnotu absolutní hodnoty daného seznamu čísel. Je pravda, že taková funkce se dá velmi snadno naprogramovat, ale rychlost výpočtu naší první (jednoduché) varianty není přijatelná. Potřebujeme, aby tato funkce byla velmi rychlá, neboť je volána na každý polynomiální objekt ve všech funkcích při nulování reálných čísel ve výsledném polynomiálním objektu.

Varianta 1. `ReplaceAll`

Velmi jednoduchá implementace.

```
37 NonZeroMin0[A : PolyMatObj[pmc, var, deg]] :=
38          Min[Abs[pmc] /. (0 | 0.) -> Infinity]
```

Rychlost řešení příliš nevyhovuje.

```
In[69] :=   A = PMRandom[20, 0];      re lnč koeficienty
            Timing[  NonZeroMin0[A] ]
Out[69] =

         {0.11 Second, 0.0000  29}
```

Porovnáním rychlosti nalezeného nenulového minima s interní funkcí `Min`, zjistíme značný rozdíl.

```
In[70] :=   Timing[  Min[Abs[A[[1]]]] ]
Out[70] =

         {0.01 Second, 0.0000  29}
```

Varianta 2. `Compile`

V definici funkce je použita funkce `Compile`. V jednoduchém cyklu hledáme minimální nenulový prvek.

```
39 NonZeroMinCompiledReal = Compile[{{dat, _Real, 3}},
40    Block[{fdat = Flatten[dat], ldat, min = 0., d},
41       ldat = Length[fdat];
42       (* min = první nenulové číslo *)
43       Do[If[Abs[fdat[[i]]] =!= 0., min = Abs[fdat[[i]]]; Break[]], {i, ldat}];
44       Do[
45          d = Abs[fdat[[i]]];
46          If[0 < d < min, min = d]
47          , {i, ldat}
48       ];
49       min
50    ]
51 ];
52
53 NonZeroMin[A:PolyMatObj[pc,var,deg]/;RealQ[A]] :=  NonZeroMinCompiledReal[pc];
```

Čas výpočtu je nyní

```
In[71] :=   Timing[  NonZeroMin[A] ]
Out[71] =

         {0.0  Second, 0.0000  29}
```

Tato varianta zrychlí výpočet asi 2krát.

5.6 Funkce N[*PolObj*]

Funkce N má v systému významné postavení. Při pokusu ji předefinovat některým ze způsobů, který byl uveden v kapitole 4.2.1, docházelo k chybám. Funkce N[*pol*] nepracovala korektně.

Jediným možným řešením bylo nastavit atribut NHoldRest k novým polynomiálním objektům. Přidělený atribut říká, že funkce N se aplikuje pouze na první argument funkce (objektu). Na další argumenty (proměnná a stupeň dané polynomiální matice) funkce není aplikována.

```
Attributes[PolyMat] = {NHoldRest};
Attributes[Poly] = {NHoldRest};
```

Nyní funkce N pracuje s polynomiálními objekty správně.

```
In[72] :=   A = PM[{{-s,   + 2s^2}, {-  + s, - }}]
            N[A,  2]      počet desetinných míst je  2
```

Out[72] =

$$\begin{pmatrix} -s & 3+2\,s^2 \\ -5+4\,s & -3 \end{pmatrix}_s$$

Out[73] =

$$\begin{pmatrix} -1.000000\ldots\,s & 3.000000\ldots+2.000000\ldots\,s^2 \\ -5.000000\ldots+4.000000\ldots\,s & -3.000000\ldots \end{pmatrix}_s$$

Kapitola 6

Závěr

Hlavním cílem naší práce bylo vyzkoušet možnosti systému *MATHEMATICA* a vytvořit sadu funkcí pro efektivní výpočty s polynomiálními maticemi a skalárními polynomy se zaměřením na analýzu a návrh řídicích systémů. Podle očekávání jsou implementované metody přímo určené pro polynomiální matice nesrovnatelně rychlejší oproti standardním symbolickým funkcím. Tyto výsledky detailně dokumentují srovnávací testy u jednotlivých funkcí.

V této fázi jsme implementovali tyto funkce:

Násobení skalárních polynomů. Nejrychlejší je metoda konvoluce koeficientů skalárního polynomu. Další metoda založená na převodu koeficientů skalárních polynomů na Sylvesterovy matice není tak efektivní, jako v případě násobení polynomiálních matic.

Maticové násobení polynomiálních matic. Pro maticové násobení vycházejí jako nejrychlejší tyto metody – metoda založená na konstrukci Sylvesterových matic a přímá metoda výpočtu koeficientů. Rychlost výpočtu u těchto metod závisí na jednotlivých stupních polynomiálních matic. Sylvesterova metoda zaostává v momentech, kdy stupeň první matice je menší než stupeň druhé matici. V tomto případě je Sylvesterova matice první polynomiální matice značně řídká a při násobení konstantních Sylvesterových matic zde dochází k zbytečnému násobení nulami. V případě přímého výpočtu je násobení nulami vynecháno.

V rychlosti výpočtu nezaostává ani metoda založená na FFT. Rychlost výpočtu se přibližuje k předešlým metodám se vzrůstajícím stupněm polynomiálních matic. Nevýhoda této metody je v nutnosti zadávat pouze číselné polynomiální matice. Poslední metoda založená na konvoluci koeficientů v maticové variantě není tak rychlá, jako v případě skalárních polynomů, kde byla nepřekonatelná. Příčina nejspíš spočívá ve špatné implementaci interní funkce pro výpočet konvoluce matic.

Determinant číselné polynomiální matice. Pro výpočet determinantu polynomiální matice byla implementována jediná metoda a to metoda založená na

FFT. Jedná se o velmi efektivní metodu výpočtu. Její jediná nevýhoda je v nutnosti zadávat pouze numerické polynomiální matice (není možné zadávat parametry). Pokud vstupní matice není číselného typu, je použita standardní symbolická metoda výpočtu. Její nasazení je ovšem velmi omezené a pro polynomiální matice větších rozměrů je z důvodů velké časové náročnosti řešení nepoužitelná.

Lineární rovnice $A.X = B$. Výpočet je založen na konstrukci Sylvesterových matic a následném řešení lineární soustavy rovnic. Stupeň výsledné matice X se hledá iteračním výpočtem (vážené binární vyhledávání).

Diofantická rovnice typu $A.X + B.Y + \ldots = C$ a $X.A + Y.B + \ldots = C$. Diofantická rovnice je přepsána na tvar $\mathbb{A}.\mathbb{X} = \mathbb{B}$ a řešena jako lineární rovnice.

Rychlost výpočtu jsme srovnávali s funkcí z programu Polynomial Matrix Utilities. V tomto programu byla funkce implementována jako ryze symbolická úloha, proto naše řešení bylo nesrovnatelně rychlejší.

V případě skalárních polynomů je výpočet jednoduší. Stupeň neznámých polynomů se nehledá iteračním způsobem, ale je na začátku přímo vypočten.

Symetrická rovnice typu $a^*x + x^*b = c$. Vstupem mohou být pouze skalární polynomy. Metoda výpočtu je založená na konstrukci upravené Sylvesterovy matice a následném řešení lineární soustavy rovnic.

Během implementace polynomiálních metod, s důrazem kladeným na rychlost výpočtu, jsme narazili na několik problémů. Zásadním krokem bylo vytvoření nových objektů pro polynomiální matice a polynomy. To by se mohlo zdát zbytečné, neboť polynomy a matice lze samozřejmě v systému *MATHEMATICA* snadno zadávat a používat. Postupné testování a rozšiřování o nové funkce však ukázalo, že bez definice objektu se nelze obejít. Pomocí nových objektů byly odstraněny problémy s neustálým testováním vstupních argumentů každé funkce, zda se jedná o matici s polynomy v dané proměnné. Další přínos nových polynomiálních objektů spočíval ve snadném předefinovávání standardních funkcí, jako např. sčítání a násobení nebo výpočet determinantu polynomiální matice.

S nově vytvořenými polynomiálními objekty je snadné přidávat další funkce a rozšiřovat tak náš balíček programů. Dále předpokládáme implementaci funkce pro řešení symetrické rovnice s polynomiálními maticemi, která se používá k řešení úlohy spektrální faktorizace polynomiálních matic.

Celkově lze říci, že systém *MATHEMATICA* se osvědčil jako velmi silný výpočetní nástroj. Mezi jeho největší přednosti patří rychlost prováděných numerických operací a možnost pracovat i s parametry (symboly) v zadání úloh.

Kapitola 7

Publikace

P[1] Kujan, P. - Hromčík, M. *New package for effective polynomial computations in MATHEMATICA.* 6th International Student Conference on Electrical Engineering, Poster'02, 23. květen, 2002, Praha.

P[2] Kujan, P. - Hromčík, M. - Šebek, M. *New package for effective polynomial computation in MATHEMATICA.* 11th Mediterranean Conference on Control and Automation. MED'03. 18.-20. červen, 2003, Rhodos, Řecko.

P[3] Kujan, P. - Hromčík, M. - Šebek, M. *New package for effective polynomial computation in MATHEMATICA.* 14th International Conference Process Control 2003, June 8-11, 2003, Štrbské Pleso, Slovakia, Proceedings ISBN 80-227-1902-1, CD-ROM

Seznam obrázků

1.1 *MATHEMATICA* Notebook. 13

3.1 Porovnání časové náročnosti výpočtu funkce `Plus[A,B]` - metoda Polynomial a Standard. 22
3.2 Porovnání časové náročnosti výpočtu funkce `Times[a,b]` - metoda Convolve, Sylvester a Expand. 24
3.3 Porovnání časové náročnosti výpočtu funkce `Times[a,b]` pro různé stupně - metoda Convolve, Sylvester a Expand. 25
3.4 Detailní graf náročnosti výpočtu funkce `Times[a,b]` - metoda konvoluce. 25
3.5 Porovnání časové náročnosti výpočtu funkce `Dot[A,B]` - metoda Standard, Sylvester, Resultant, Convolve a FFT. 29
3.6 Porovnání časové náročnosti výpočtu funkce `Dot[A,B]` - metoda Sylvester, Resultant, Convolve a FFT. 31
3.7 Porovnání časové náročnosti výpočtu funkce `Dot[A,B]` - metoda Sylvester a Resultant, Convolve a FFT. 32
3.8 Časová náročnost výpočtu funkce `Det[A]`. 34
3.9 Regulovaná soustava. 54
3.10 Přechodová charakteristika soustavy. 55

Seznam tabulek

3.1 Porovnání časová náročnosti výpočtu násobení skalárních polynomů $a * b$. 26
3.2 Porovnání časové náročnosti výpočtu maticového násobení polynomiálních matic $A.B$ (rozměr A a B je [30×30]). 32
3.3 Porovnání časové náročnosti výpočtu funkce $\det A$. 34
3.4 Porovnání časové náročnosti výpočtu polynomiální rovnice $A.X = B$ (stupeň matice A a B je d a rozměry jsou $[n \times n + 5]$ a $[n \times n]$). . 39
3.5 Porovnání časové náročnosti výpočtu rovnice $A.X + B.Y = C$ (stupeň čtvercových matic A, B a C rozměru n je d). 47
3.6 Porovnání časové náročnosti výpočtu skalární rovnice $ax + by = c$ (stupeň skalárních polynomů a, b a c je d). 47
3.7 Porovnání časové náročnosti výpočtu skalární rovnice $a^* x + a\, x^* = b$ (stupeň polynomu $\deg a = d$ a $\deg b = 2d$). 49
3.8 Popis argumentů funkce `PMRandom[args]` 50
3.9 Globální proměnné 52

Literatura

[1] Hromčík, M. *Numerical Algorithms for Polynomial Matrices.* Diplomová práce. Praha: České vysoké učení technické, elektrotechnická fakulta, katedra řídicí techniky, 1999.

[2] Wolfram, S. *The Mathematica Book.* 4. vyd., Cambridge University Press, Wolfram Media, 1999.

[3] Wolfram, S. *The Mathematica Book* [online]. 2004. ⟨`http://documents.wolfram.com/v /TheMathematicaBook/c.html`⟩.

[4] Wolfram Research, Inc. *Mathematica home page* [online]. 2004. ⟨`http://www.mathematica.com`⟩.

[5] Schoeller, P. *BLAS and LAPACK Performance in Mathematica 5.0.* In: Mathematica Developer Conference, April 10-12, 2003, Champaign, Illinois, USA. ⟨`http://library.wolfram.com/infocenter/Conferences/, 8 8`⟩.

[6] Maplesoft, a division of Waterloo Maple, Inc. *Maple Home Page* [online]. 2004. ⟨`http://www.maplesoft.com`⟩.

[7] Texas Instruments, Inc. *Derive Home Page* [online]. 2004. ⟨`http://www.derive.com`⟩.

[8] MathSoft, Inc. *MathCAD Home Page* [online]. 2004. ⟨`http://www.mathcad.com`⟩.

[9] Heck, A. *Introduction to Maple.* 2. vyd., Springer-Verlag, Berlin, 1996.

[10] MathWorks, Inc. *Matlab Home Page* [online]. 2004. ⟨`http://www.mathworks.com`⟩.

[11] Dušek, F. *Matlab a Symulink: úvod do používání.* 1. vyd., Univerzita Pardubice, 2000.

[12] Bakshee, I., *Control System Professional - Mathematica*, Wolfram research, Inc., 1996.

[13] Kwakernaak, H. - Šebek, M. *PolyX Home Page* [online]. 2004. ⟨`http://www.polyx.com`⟩.

[14] PolyX. *The Polynomial Toolbox for Matlab, Manual*, 2004.
⟨`http://www.polyx.com`⟩.

[15] Henrion, D. *Home page* [online].
⟨`http://www.laas.fr/ henrion/`⟩

[16] Bahder, T. *Mathematica for Scientists and Engineers.* Addison - Wesley, 1994.

[17] Pascoletti, A. *Polynomial Matrix Utilities - Mathematica Package* [online]. Poslední revize 2000-01-04.
⟨`http://library.wolfram.com/infocenter/MathSource/ , 1`⟩.

[18] Kučera, V. *Analysis and design of discrete linear control systems.* 1. vyd., Praha: Academia, 1991.

[19] Štecha, J. - Havlena, V. *Teorie dynamických systémů.* Praha: Ediční středisko ČVUT, 1995.

[20] Wagner, D. *Power Programming with Mathematica: the Kernel.* McGraw-Hill, 1996.

[21] Hrommčík, M. - Šebek, M., *New Algorithm for Polynomial Matrix Determinant Based on FFT.* In: Proceedings of the European Control Conference EUCA'99. Karlsruhe, Germany, September 1999.

[22] Henrion, D. *Reliable Algorithms for Polynomial Matrices.* PhD. Thesis. Praha: Ústav teorie informace a automatizace, Česká akademie věd, 1998.

[23] Ersek, T. *Mathematica Tricks Home Page* [online]. Poslední revize 2002-10-20.
⟨`http://www.verbeia.com/mathematica/tips/tip_index.html`⟩.

[24] Knapp, R. *Packed Arrays: An efficient internal storage format in Mathematica 4*, Wolfram Research, Inc. 2000.
⟨`http://library.wolfram.com/infocenter/Articles/ 1, 1`⟩.

[25] MathGroup, *Internetová diskuzní skupina.*
⟨`news://comp.soft-sys.math.mathematica`⟩.

Příloha A

Tištěná dokumentace

Příloha B

CD s programovými soubory