Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Algoritmy pro výpočet maticové exponenciály v Sage

Vedoucí práce: Ing. Tomáš Kalvoda, Ph.D.

10. května 2015

Poděkování

Na tomto místě chci poděkovat svému vedoucímu bakalářské práce Ing. Tomáši Kalvodovi, Ph.D., za cenné rady, připomínky a za jeho ochotu a čas věnovaný pravidelným konzultacím, které mi byly vždy výbornou zpětnou vazbou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2015 .....

Odkaz na tuto práci

Tomanek, Jakub. *Algoritmy pro výpočet maticové exponenciály v Sage*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

V této práci se budeme zabývat algoritmy pro numerický výpočet maticové exponenciály. Zvýšenou pozornost budeme věnovat *open-source* programu Sage, pro který budeme implementovat méně vídanou Krylovovu metodu. Ta se od ostatních metod podstatně liší přístupem k řešení problému. Postupně si popíšeme všechny klíčové části metody, uvedeme si přístupy k její implementaci a nakonec otestujeme její přesnost a výkonnost vůči existujícím metodám v programu Sage. V samotném závěru si uvedeme praktickou ukázku využitelnosti naší implementace Krylovovy metody na reálném fyzikálním problému vedení tepla.

Klíčová slova maticová exponenciála, Krylovova metoda, Padé aproximace, Sage, NumPy, SciPy

Abstract

In this thesis we will deal with algorithms for numerical computation of matrix exponential. We will focus more closely on open-source program Sage in which we will implement infrequent Krylov's method for. This method differs from the others with the main approach for solving the problem. By little steps we will introduce all parts of this method, we will describe its implementation and at the end we will measure its precision and efficiency compared to the methods available in Sage. At the end of this thesis we will demonstrate practical example of the usage of our implementation of Krylov's method on real physical problem of heat conduction.

Keywords matrix exponential, Krylov subspace method, Padé approximation, Sage, NumPy, SciPy

Obsah

Úvod **1**

1 Teorie a metody výpočtu maticové exponenciály **3**
1.1 Motivace . . . 3
1.2 Definice maticové exponenciály . . . 4
1.3 Možnosti výpočtu maticové exponenciály . . . 5

2 Rešerše stávajících nástrojů pro výpočet maticové exponenciály **7**
2.1 Maxima . . . 7
2.2 Knihovny NumPy, SciPy . . . 7
2.3 EXPOKIT . . . 8
2.4 Mathematica . . . 8
2.5 MATLAB . . . 8
2.6 SageMath . . . 9

3 Analýza vnitřních algoritmů Krylovovy metody **11**
3.1 Arnoldiho algoritmus . . . 11
3.2 Padého aproximace . . . 14

4 Implementace Krylovovy metody **19**
4.1 Obecný popis implementace Krylovovy metody . . . 21
4.2 Typy matic . . . 21
4.3 Analýza implementace vnitřních algoritmů . . . 23
4.4 Knihovna NumPy . . . 25
4.5 Modifikace dle J. Highama . . . 26

5 Testy a měření výkonnosti implementace **29**
5.1 Doctesting v Sage . . . 29
5.2 Profilování kódu . . . 31

5.3 Měření kvality 33

6 Demonstrace využití maticové exponenciály **39**
6.1 Rovnice vedení tepla 39

Závěr **43**

Literatura **45**

A Seznam použitých zkratek **47**

B Obsah přiloženého CD **49**

Seznam obrázků

3.1 Funkce ${}_1F_1[-m; -2m](-x)$ na okolí (-2, 2) pro různá m 17

5.1 Vliv parametru Arnoldiho algoritmu na přesnost výpočtu 34
5.2 *Boxplot* naměřených chyb pro jednotlivé rozměry matic při výpočtech jejich exponenciál pomocí naší implementace Krylovovy metody s Arnoldiho parametrem rovným 15. Každá „krabicová“ část v grafu je vymezena hranicemi 1. a 3. kvartilu a „vousy“ představují nejnižší údaj 1,5 IQR spodního kvartilu a nejvyšší údaj 1,5 IQR horního kvartilu. Křížky reprezentují *outliary*. 35
5.3 Vliv řádu Padého aproximace uvnitř Krylovovy metody na přesnost výpočtu 36
5.4 Graf znázorňující dobu výpočtu maticové exponenciály v závislosti na typech a velikosti vstupní matice 36
5.5 Graf znázorňující dobu výpočtu úlohy e^A v závislosti na zvolené metodě pro její výpočet a velikosti matice. V tomto srovnání jsme použili matice typu RDF 37

6.1 Vývoj tepelného vedení na dvourozměrné mřížce 41

Seznam tabulek

4.1 Okruhy matic v Sage 22
4.2 Formáty řídkých matic definovaných knihovnou SciPy 22
4.3 Hlavní rozdíly v syntaxi programu Sage a knihovny NumPy 25
4.4 Konstanty pro algoritmus Highama 26

5.1 Přepis výstupu `line_profiler`u pro funkci `expKrylov()` 32
5.2 Přepis výstupu `line_profiler`u pro funkci `arnoldi()` 32

Úvod

Tato práce se zaměřuje na numerický výpočet maticové exponenciály. Značná část je věnována popisu a implementaci Krylovovy metody pro počítačový algebraický systém Sage.

Krylovova metoda pro výpočet maticové exponenciály je zajímavá svým přístupem k problému. Na rozdíl od aproximace e^X tato metoda numericky napočte

$$e^X v, \quad X \in \mathbb{C}^{n \times n}, \quad v \in \mathbb{C}^n.$$

Často nás totiž může zajímat pouze součin exponenciály s vektorem. Pro ilustraci si v kapitole 6 uvedeme příklad rovnice vedení tepla, pro kterou nás vždy zajímá exponenciála aplikovaná na počáteční podmínku. Nespornou výhodou tohoto přístupu je vysoká efektivita pro matice vysokých řádů i za cenu drobné ztráty přesnosti.

Maticovou exponenciálu lze v současné době počítat v několika komerčních i *open-source* produktech. Bohužel většina z těchto programů neimplementuje zmiňovanou Krylovovu metodu a proto jedním z výstupů mé práce je doplnění programu Sage o vlastní implementaci této metody.

V úvodní části 1 se zaměříme na teoretický základ a definici maticové exponenciály. Následovat bude kapitola 2, ve které se zaměříme na zmapování dosavadních řešeních této problematiky v komerčních i *open-source* produktech. Následující kapitoly 3 a 4 budou zahrnovat návrh a implementaci Krylovovy metody pro počítačový algebraický systém Sage. V předposlední kapitole 5 si uvedeme přístupy k testování, měření kvality a výkonnosti naší implementace pro Sage. V poslední kapitole 6 si uvedeme zmiňovanou ilustraci problému rovnice vedení tepla s využitím naší implementace.

Kapitola 1

Teorie a metody výpočtu maticové exponenciály

1.1 Motivace

Počítačovým algebraickým systémem se může označovat program sloužící k symbolickým či numerickým výpočtům. Tyto systémy jsou využívány zejména pro vědecké výzkumy, méně často pro běžné výpočty. Je proto od nich vyžadována maximální možná přesnost a spolehlivost. Algebraické systémy jsou stavěny tak, aby současně využily maximální výpočetní výkon současných strojů se zachováním maximální přesnosti výpočtů. Tyto dvě vlastnosti jdou občas proti sobě a faktem je, že i systémy, o kterých je řeč, jsou psány pouze lidmi a nelze tak vyloučit chybovost jistých částí systému. Problematiku chyb a jejich náprav hezky popisuje trojice španělských matematiků. Ve svém článku [1] uvádějí několik aktuálních příkladů chybných výpočtů komerčních systémů s demonstracemi na populárním programu Mathematica. Pointou jejich článku je fakt, že systémy nejsou samospásné a vícenásobné ověření výpočtů je na místě. Mimo tento článek jsme s vedoucím práce objevili poměrně závažnou chybu v systému Sage při výpočtu maticové exponenciály. Chybu se nám povedlo objevit díky možnosti nahlédnutí do zdrojových kódů. To u komerčních produktů možné není a objevení chyby je buď dílem náhody, nebo výsledkem poctivé detektivní práce.

Motivací pro vypracování této bakalářské práce je provedení rešerše v přístupech numerického výpočtu maticové exponenciály jak v systému Sage, tak i ostatních konkurenčních produktů. Ukazuje se, že systém Sage tuto část nemá zvládnutou dobře a proto je potřeba tuto oblast prozkoumat a popsat ji. Mimo tuto rešerši se pokusíme naprogramovat méně vídanou Krylovovu metodu pro výpočet maticové exponenciály. Ta je zajímavá tím, že zvládá spočítat uvažovaný problém s maticemi daleko výšších řádů než ostatní metody. Jedním z výstupů této práce by měl být opravný balíček pro systém Sage, který by řešil současnou situaci maticové exponenciály doplněnou o Krylovovu

metodu.

1.2 Definice maticové exponenciály

Definice 1. Nechť $X \in \mathbb{C}^{n\times n}$ je matice řádu n. Pak maticovou exponenciálu e^X definujeme jako nekonečnou řadu

$$e^X := \sum_{k=0}^{\infty} \frac{1}{k!} X^k. \tag{1.1}$$

Z definice je maticová exponenciála funkce nad čtvercovými maticemi typu $n \times n$ a jedná se o analogii exponenciální funkce zavedenou pro reálná čísla. Tato maticová funkce nachází uplatnění například při řešení soustav lineárních rovnic. Tuto aplikaci uvádíme v kapitole 6.

1.2.1 Základní vlastnosti

Nechť $\mathcal{O}$ je nulová matice řádu n. Pak přímo z definice (1.1) plyne rovnost

$$e^{\mathcal{O}} = I, \tag{1.2}$$

kde I je jednotková matice řádu n.

Nechť $D = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$ je diagonální matice. Její maticová exponenciála má následující tvar

$$e^D = \operatorname{diag}(e^{\lambda_1}, \ldots, e^{\lambda_n}).$$

Nechť čtvercové matice $A, B \in \mathbb{C}^{n\times n}$ jsou vzájemně komutativní, tj.

$$AB - BA = \mathcal{O},$$

pak

$$e^{A+B} = e^A e^B. \tag{1.3}$$

Pro $A, B, P \in \mathbb{C}^{n\times n}$ takové, že $B = P^{-1}AP$ platí

$$e^B = P^{-1} e^A P. \tag{1.4}$$

Této vlastnosti využijeme zejména při konstrukci matic s přesně známou exponenciálou. Na takovýchto maticích můžeme testovat přesnost a efektivitu našich algoritmů.

Pro každé $A \in \mathbb{C}^{n\times n}$ platí

$$(e^A)^{-1} = e^{-A}, \tag{1.5}$$

ekvivalentně k (1.5) pak také platí

$$e^A e^{-A} = I. \tag{1.6}$$

Pokud matice $A \in \mathbb{C}^{n\times n}$ je nilpotentní, tj. existuje $m \in \mathbb{N}$ splňující $A^m = \mathcal{O}$, pak

$$e^A = \sum_{k=0}^{m-1} \frac{1}{k!} X^k.$$

1.3 Možnosti výpočtu maticové exponenciály

Výpočet exponenciály z definice 1 není nejefektivnější cestou k jejímu získání. V této kapitole popíšeme nejčastější přístupy k výpočtu uvažované funkce.

1.3.1 Pomocí definice

Tato metoda ve skutečnosti kopíruje definici. Uživatel si zvolí míru přesnosti ε a dle ní vypočítá součet prvních n členů Taylorovy řady tak, aby chyba výpočtu byla menší než ε, tj.

$$\left\| e^A - \sum_{k=0}^{n} \frac{A^k}{k!} \right\| \leq \sum_{k=n+1}^{\infty} \frac{\|A\|^k}{k!} = e^{\|A\|} - \sum_{k=0}^{n} \frac{\|A\|^k}{k!} < \varepsilon.$$

Zde $\|A\|$ je předem zvolená maticová norma.

1.3.2 Jordanova metoda rozkladu matice

Tato metoda k zadané matici nejprve zjistí její spektrum a matici převede do Jordanova normálního tvaru J, existuje tedy regulární matice P splňující

$$A = PJP^{-1}.$$

Pokud je matice J diagonální, $J = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$, pak je exponenciála rovna

$$e^A = Pe^J P^{-1},$$

kde

$$e^J = \begin{pmatrix} e^{\lambda_1} & 0 & \cdots & 0 \\ 0 & e^{\lambda_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & e^{\lambda_n} \end{pmatrix} \in \mathbb{C}^{n\times n}.$$

V opačném případě exponenciálu vypočteme z jejího Jordanova tvaru pomocí aproximační metody popsané v [2].

1.3.3 *Scaling and squaring*

Scaling and squaring je dle [2] jedna z nejpopulárnějších metod pro aproximaci maticové exponenciály. Tato metoda je založena na aproximaci

$$e^A \approx (r_m(2^{-s}A))^{2^s},$$

kde $r_m(x)$ je $[m/m]$-tý rozvoj Padého aproximace a $s \in \mathbb{N}$. V této metodě jsou m a s vstupními parametry a jejich volbou lze ovlivňovat rychlost a přesnost výpočtu. Potenciální problém je ukryt ve volbě přespříliš velkého parametru s, jež má za následek nepřesnosti v operacích v pohyblivé desetinné čárce (tzv. *over-scaling*, [2]).

1.3.4 Krylovova metoda

U této metody je důležité zmínit hlavní rozdíl oproti výše uvedeným metodám. Přístup Krylovovy metody spočívá v aproximaci exponenciály matice na konkrétním vektoru (tj. $\exp(A)v$) na rozdíl od výpočtu exponenciály matice ($\exp(A)$)[1]. Krylovova metoda počítá exponenciálu z projekce původní velké matice A na Krylovův podprostor generovaný lineárním obalem

$$K_m = \text{span}\,\{v, Av, \ldots, A^{m-1}v\},$$

kde m bývá typicky značně menší než řád matice A. Výpočet exponenciály tak provádíme na matici daleko menších rozměrů, než byl rozměr původní matice, viz [3]. Pro výpočet báze K_m a uvažovanou projekci matice A na K_m existuje několik algoritmů. Jeden z nejznámějších je Arnoldiho algoritmus, jehož výstupem je ortonormální báze $V_m = \{\frac{v}{\|v\|_2}, v_2, \ldots, v_m\}$ podprostoru K_m a Hessenbergova matice H_m, která reprezentuje projekci A na K_m [4]. Dle [4] má výsledná aproximace následující tvar

$$e^A v \approx \beta V_m e^{H_m} e_1, \qquad \beta = \|v\|_2,$$

kde $\|v\|_2$ je Euklidovská norma vektoru v.

Pokud bychom chtěli vypočítat e^A pomocí Krylovovy metody, stačilo by popisovanou metodu pustit na bazické vektory $\{e_1, \ldots, e_n\}$[2], a z výsledných sloupců zrekonstruovat e^A. Tento proces by se dal navíc jednoduše paralelizovat.

[1] $\exp(A)$ je jiné označení pro e^A

[2] Připomeňme, že n chápeme jako rozměr matice A.

Kapitola 2

Rešerše stávajících nástrojů pro výpočet maticové exponenciály

Pro vyčíslitelnost maticové exponenciály existuje v současné době několik nástrojů. Jsou zde zástupci jak komerčního software, tak i samotné funkce v rámci svobodných matematických knihoven či *open-source* programů. V přehledu si uvedeme stručné charakteristiky těchto nástrojů spolu s popisem jejich funkcí, které se dotýkají našeho problému.

2.1 Maxima

Jedná se o svobodný počítačový algebraický systém. Umožňuje nám vypočítat exponenciálu matice se symbolickými hodnotami a výsledky nám vrací jako symbolické výrazy. Funkce pro tento výpočet se jmenuje `matrixexp(m[,x])`, jejíž argumenty jsou matice (`m`) a volitelný parametr multiplikativní konstanty matice `x` (s výchozí hodnotou 1) Tato funkce je součástí knihovny `linearalgebra`. Bohužel selhává na maticích s čísly s pohyblivou řadovou čárkou, a to i velmi malých rozměrů. Z chybové hlášky zjistíme, že Maxima pro takové matice není schopna získat spektrum matice.

2.2 Knihovny NumPy, SciPy

Knihovna NumPy představuje pro jazyk Python základní balíček matematických funkcí. Zavádí mimo jiné objekt n-dimenzionálních polí a implementuje řadu funkcí z lineární algebry. Knihovna SciPy staví na datových typech NumPy a rozšiřuje paletu matematických funkcí a datových typů. Pro nás bude zajímavých 7 datových typů řídkých matic. K výpočtu maticové exponenciály nám SciPy nabízí následující tři funkce:

- `expm(A[,q])`
  Jejími argumenty jsou umocňovaná matice a volitelně řád Padého apro-

ximace. Z komentářů zdrojového kódu vyplývá, že argument `q` se již delší dobu nevyužívá a je ignorován. Ve skutečnosti je výpočet této funkce zprostředkován stejnojmennou funkcí `expm(A)` z knihovny `scipy.sparse.linalg`, kde je využita metoda Padého aproximace.

- `expm2(A)`
  Jediným argumentem je umocňovaná matice. Tato funkce pro výpočet maticové exponenciály využívá metodu Jordanova rozkladu matice.

- `expm3(A, q=20)`
  V této funkci se využívá aproximace Taylorovým polynomem. Prvním parametrem je umocňovaná matice a parametr `q` určuje řád Taylorova polynomu.

Obě knihovny jsou dostupné pro použití s licencí BSD.

2.3 EXPOKIT

Autorem knihovny EXPOKIT je Roger Sidje. Jeho knihovna se zaměřuje čistě na problematiku výpočtu maticové exponenciály. Je napsána v jazyce Fortran 77 a tato knihovna jako jedna z mála zjištěných nástrojů implementuje Krylovovu metodu. Kromě této metody taky obsahuje implementaci Padého aproximace a Chebyshevovy metody. Spolu se zdrojovými kódy je volně dostupná pro nekomerční použití [5].

2.4 Mathematica

Mathematica je komerční software vyvíjený společností Wolfram Research. Uživateli poskytuje pro výpočet exponenciály funkci `MatrixExp[...]` ve dvou variantách. První z nich na základě parametru matice vrací maticovou exponenciálu. Druhá varianta si bere kromě matice ještě vektor. Vrací pak exponenciálu matice aplikovanou na vektor. Obecně se ve zdroji [2] uvádí, že funkce `MatrixExp` využívá k výpočtu metodu *Scaling and squaring*. Na oficiálním webu společnosti Wolfram Research však existuje nepatrná poznámka [6], že se popisovaná funkce používá mimo zmiňovanou metodu taky metodu aproximací v Krylovových podprostorech.

2.5 MATLAB

MATLAB je výrobcem, společností MathWorks, charakterizován jako jazyk v interaktivním prostředí pro vývoj algoritmů, numerické výpočty a vizualizaci dat. V MATLABu je k výpočtu maticové exponenciály určena funkce `expm`. MATLAB narozdíl od programu Mathematica využívá k výpočtu maticové

exponenciály pouze metodu *Scaling and squaring*. Tento fakt je uveřejněn v oficiální dokumentaci u popisované metody [2].

2.6 SageMath

Tento program je zkráceně nazýván Sage a jedná se o *open-source* program pro matematické výpočty. Iniciátorem a lídrem tohoto projektu je matematik William Stein a do tohoto projektu přispívá mnoho dalších uživatelů. Posláním tohoto programu je stát se relevantní alternativou ke komerčním produktům jako jsou Maple, Magma, Mathematica nebo MATLAB. Je volně šiřitelný pod licencí GPL. Sage pro své výpočty využívá mnoho existujících *open-source* knihoven a programů a sestavuje tyto kousky do jednoho celku. Na oficiálním webu se můžeme dočíst, že počet těchto menších knihoven se blíží k číslovce 100. Sage je multiplatformní a je možné si jej stáhnout a používat v rámci OS, i když ve Windows pouze pomocí virtualizace. Mimo to se v posledních letech rozvinul vývoj Sage na online verzi, dnes dostupnou na adrese `cloud.sagemath.org`. Výhodou této online verze je možnost vytváření sdílených projektů mezi více uživateli. Tento online nástroj jsem osobně využil při vytváření této práce.

Tento nástroj používá pro výpočet maticové exponenciály dvě funkce. Hlavní funkce `exp()` se opírá o knihovnu Maxima. Tato metoda je využívána jako výchozí metoda pro objekt třídy `sage.matrix.matrix_generic_dense.Matrix_genericdense`.

Druhou funkcí je stejnojmenná `exp(A, algorithm, order)`. Hlavní rozdíl oproti předchozí funkci tkví v parametrech. Funkce si bere jako parametr samotnou matici a volitelně i další parametry. Důležité je, že pro využití této metody je potřeba matice v odlišném datovém typu než pro výše uvedenou funkci `exp()` (`sage.matrix.matrix_dense.matrix_double_dense`). Tento typ je před výpočtem převeden do typu matice knihovny `Scipy` a následné výpočty jsou taktéž prováděny touto knihovnou. Výsledek výpočtu je následně převeden zpět do původního typu.

Parametr `algorithm` označuje metodu pro výpočet exponenciály. Možnosti pro tento parametr jsou:

- `taylor` metoda výpočtu založená na aproximaci Taylorovým polynomem. Po bližším prozkoumání je tato metoda s výchozím parametrem `order` krajně nepřesná. Už pro relativně malé matice rozměrů 2×2 se výsledky výpočtu oproti jiným metodám liší řádově 10^4. V takovém případě by měla být výchozí hodnota parametru nastavena na vyšší hodnotu, nebo v lepším případě by si funkce měla vynutit vyplnění tohoto parametru.

- `eig` metoda výpočtu založená na Jordanově normálním rozkladu.

- `pade` metoda výpočtu založená na Padé aproximaci. Je to také výchozí hodnota.

Parametr `order` má význam pouze pro algoritmus `taylor` a vyznačuje s kolikátým rozvojem Taylorova polynomu se bude počítat.

Kapitola 3

Analýza vnitřních algoritmů Krylovovy metody

3.1 Arnoldiho algoritmus

Předpokládejme, že máme matici $A \in \mathbb{C}^{n\times n}$ a vektor $v \in \mathbb{C}^n$, pro který chceme aproximovat hodnotu $e^A v$. Pro Arnoldiho algoritmus musíme také znát dimenzi Krylovova podprostoru (ozn. m), která by měla být značně menší než řád matice A. Tato trojice tvoří vstup Arnoldiho algoritmu.

Algorithm 1 Arnoldiho algoritmus – pseudokód

Require: $A \in \mathbb{C}^{n\times n}, v \in \mathbb{C}^n, m \geq 1$
1: $v_1 \leftarrow v/\|v\|_2$
2: **for** $j = 1$ **to** m **do**
3: $w \leftarrow Av_j$
4: **for** $i = 1$ **to** j **do**
5: $h_{i,j} \leftarrow \langle w, v_i \rangle$
6: $w \leftarrow w - h_{i,j}v_i$
7: **end for**
8: $h_{j+1,j} \leftarrow \|w\|_2$
9: $v_{j+1} \leftarrow w/h_{j+1,j}$
10: **end for**

Výstupem tohoto algoritmu je dvojice matic $V_m = (v_1, \ldots, v_m)$ a $H_m = (h_{ij})_{i,j=1}^m$. Bližší informace o konstrukci těchto matic jsou rozebrány níže v této kapitole. Pokud jde o implementaci, tak ta je pak s podporou knihovny NumPy či prostředí Sage pouhým kopírováním kroků uvedeného pseudokódu.

3.1.1 Analýza pseudokódu

Pro pochopení výstupu tohoto algoritmu je dobré poukázat na několik kroků ve výpočtu. První důležitou částí je normalizace vstupního vektoru v a vnitřní *for* cyklus na řádcích 4-7, kde je prováděn upravený Gram-Schmidtův algoritmus. Výsledkem je proto ortonormální báze

$$V_m = (v_1, v_2, \ldots, v_m), \tag{3.1}$$

resp. soubor vektorů, který generuje Krylovův podprostor

$$K_m = \text{span}\,\{v, Av, A^2v, \ldots, A^{m-1}v\} \tag{3.2}$$

Na tomto místě se hodí zmínit, že se vždy nemusí jednat o bázi. V případě, že vektor v zvolíme například jako vlastní vektor matice A nebo matice A je nilpotentní pro jistou mocninu i, kde $i < m - 1$, pak výstupem Arnoldiho algoritmu zcela jistě není lineárně nezávislý soubor vektorů. Za Krylovův podprostor pak bereme lineární obal výstupní množiny vektorů a dimenze uvažovaného podprostoru je ostře menší než zvolené m. Zdroje rozebírající Arnoldiho algoritmus ale tuto nepřesnost nijak nezmiňují a množinu (3.2) označují za bázi.

Pomineme-li zmíněné speciální případy, tak Arnoldiho algoritmus produkuje dle uvedeného pseudokódu soubor $m+1$ vzájemně ortogonálních vektorů. My budeme ale pro následující výpočty využívat pouze prvních m vektorů, jakožto ortogonální bázi Krylovova podprostoru. Matice V_m vypadá následovně:

$$V_m = \left(\frac{v_1}{\|v_1\|_2}, \frac{Av_1 - h_{1,1}v_1}{\|Av_1 - h_{1,1}v_1\|_2}, \frac{A^2v_2 - \sum_{i=1}^{2} h_{i,2}v_i}{\|A^2v_2 - \sum_{i=1}^{2} h_{i,2}v_i\|_2}, \cdots, \frac{A^{m-1}v_{m-1} - \sum_{i=1}^{m-1} h_{i,m-1}v_i}{\|A^{m-1}v_{m-1} - \sum_{i=1}^{m-1} h_{i,m-1}v_i\|_2} \right)$$

Matice (h_{ij}), $i = 1, \ldots, m+1$, $j = 1, \ldots, m$ na výstupu Arnoldiho algoritmu má ve skutečnosti rozměry $(m+1) \times m$. Tato matice je ve sloupcích vyplněná dílčími kroky upraveného Gram–Schmidtova ortogonalizačního algoritmu a pro úplnost si uvedeme její analytické vyjádření

$$(h_{ij}) = \begin{pmatrix} \langle Av_1, v_1 \rangle & \langle A^2v_2, v_1 \rangle & \cdots & \langle A^m v_m, v_1 \rangle \\ \|Av_1 - h_{1,1}v_1\|_2 & \langle A^2v_2 - \sum_{i=1}^{1} h_{i,2}v_i, v_2 \rangle & \cdots & \langle A^m v_m - \sum_{i=1}^{1} h_{i,m}v_i, v_2 \rangle \\ 0 & \|A^2v_2 - \sum_{i=1}^{2} h_{i,2}v_i\|_2 & \cdots & \langle A^m v_m - \sum_{i=1}^{2} h_{i,m}v_i, v_3 \rangle \\ \vdots & 0 & \ddots & \vdots \\ \vdots & 0 & \ddots & \langle A^m v_m - \sum_{i=1}^{m} h_{i,m}v_i, v_m \rangle \\ 0 & \cdots & 0 & \|A^m v_m - \sum_{i=1}^{m} h_{i,m}v_i\|_2 \end{pmatrix}$$

Označme si Hessembergovu matici H_m o rozměrech $m \times m$ tvořenou prvky matice $(h_{ij})_{i,j=1}^{m}$, pak dle [4] platí vztah:

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^T \tag{3.3}$$

Člen e_m chápejme jako sloupcový jednotkový vektor s jedničkou na m-té pozici. Výraz $h_{m+1,m}v_{m+1}e_m^T$ je pak matice typu $n \times m$, která má v m-tém sloupci přebývající vektor v_{m+1} z Arnoldiho algoritmu a ten je pronásoben prvkem $h_{m+1,m}$. Dále pak víme, že V_m je ortogonální, takže platí $V_m^{-1} = V_m^T$. Pronásobením (3.3) V_m^T zleva se druhý člen pravé strany vynuluje a ve výsledku dostaneme vztah

$$H_m = V_m^T A V_m. \tag{3.4}$$

Díky této rovnosti můžeme matici H_m označit jako projekci lineární transformace matice A na Krylovův podprostor K_m generovaný V_m. Výsledně nám tato rovnost dovoluje položit následující aproximaci

$$e^A v \approx V_m V_m^T e^A v = \beta V_m V_m^T e^A v_1 = \beta V_m V_m^T e^A V_m e_1 = \beta V_m e^{H_m} e_1, \tag{3.5}$$

kde $\beta = \|v\|_2$, a $V_m^T v_1 = e_1$.

3.1.2 Odhad složitosti

Odhad složitosti Arnoldiho algoritmu vypočteme z počtu operací násobení. Pro jednotlivé členy odhadu budeme odkazovat na jednotlivé řádky pseudokódu.

Uvnitř vnějšího *for*-cyklu na řádce 3 se n-krát násobí řádkový vektor matice A s vektorem v_j. Reálný počet násobení čísel je tedy n^2 a jelikož tento počet nijak nezávisí na iteračním parametru j, tak na 3. řádkek pseudokódu připadá mn^2 násobení. Ve vnitřním *for*-cyklu se pak na 5. a 6. řádku provádí po n násobeních. Jednou je to v rámci skalárního součinu vektorů délky n a v druhém případě se jedná násobení vektoru stejné délky číslem. Přesný počet násobení připadajících těmto dvou řádkám můžeme vyjádřit následujícím výrazem

$$\sum_{j=1}^{m} \sum_{i=1}^{j} 2n = 2 \sum_{j=1}^{m} jn = m^2 n + mn.$$

Celková složitost Arnoldiho algoritmu v počtu násobení čísel je

$$mn^2 + m^2 n + mn = \mathcal{O}(mn^2 + m^2 n).$$

3.1.3 Volba optimální hodnoty parametru m

Pojednání o optimálním parametru m pro Krylovovu metodu maticové exponenciály jsem se nikde nedočetl. Provedl jsem ale zajímavá pozorování, díky kterým se mi povedlo částečně zjistit zmiňované optimum. Postupně jsem si generoval husté matice, u kterých jsem znal jejich přesnou maticovou exponenciálu, a porovnával je s výstupem mé implementace na základě měnícího se parametru m. Je šokující, že toto optimum v řadě případů nikterak nezávisí na velikosti vstupní matice. Postupy a závěry pozorování blíže rozeberu v kapitole 5.

3.2 Padého aproximace

Padého aproximaci využívám k implementaci vlastní *Scaling and squaring* metody pro výpočet exponenciály z Hessenbergovy matice získanou Arnoldiho algoritmem. Tyto funkce lze v mé implementaci najít pod názvy `_pade_hyper geometric(...)` nebo `expPade(...)`. Pro podrobnější vhled do těchto funkcí viz kapitola Implementace 4.

Definice 2. Mějme zadanou funkci $f(x)$ a čísla $m \geq 0$ a $n > 0$. Padého aproximací pak nazýváme racionální lomenou funkci označovanou typem $[m/n]$

$$f(x) \approx R_{m,n}(x) = \frac{\sum_{i=0}^{m} a_i x}{1 + \sum_{j=1}^{n} b_j x} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m}{1 + b_1 x + b_2 x^2 + \cdots + b_n x^n} \tag{3.6}$$

splňující

$$\forall i \in \{0, 1, \ldots, m+n\} : f^{(i)}(0) = R^{(i)}(0). \tag{3.7}$$

Metody výpočtu koeficientů pro exponenciálu

V metodě *Scaling and squaring* pro maticovou exponenciálu se využívá $[m/m]$–tá Padého aproximace. Z definice exponenciální funkce víme, že má tvar

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

Úlohou je vypočíst $\{a_0, \ldots, a_m, b_1, \ldots, b_m\}$ tak, aby byla splněna podmínka v (3.7),

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \approx \sum_{i=0}^{2m} \frac{x^i}{i!} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m}{1 + b_1 x + b_2 x^2 + \cdots + b_m x^m}. \tag{3.8}$$

Soustava rovnic

Pro jednou z možností dopočtu neznámých koeficientů budeme potřebovat poslední rovnost aproximace.

$$\sum_{i=0}^{2m} \frac{x^i}{i!} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m}{1 + b_1 x + b_2 x^2 + \cdots + b_m x^m}. \tag{3.9}$$

Pro zjednodušení zápisu si zadefinujme $c_i := \frac{1}{i!}$. Vynásobíme-li (3.9) jmenovatelem pravé strany, dostaneme

$$(1 + b_1 x + \cdots + b_m x^m)(c_0 + c_1 x + \cdots + c_{2m} x^{2m}) = (a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m).$$

Je zřejmé, že po vynásobení levé strany rovnosti se musí koeficienty u x^{m+1}, x^{m+2}, $\cdots$, x^{2m} rovnat nule. To vede na soustavu m rovnic o m neznámých:

$$\begin{aligned} b_m c_1 + b_{m-1} c_2 + \cdots + b_1 c_m + c_{m+1} &= 0 \\ b_m c_2 + b_{m-1} c_3 + \cdots + b_1 c_{m+1} + c_{m+2} &= 0 \\ &\vdots \\ b_m c_m + b_{m-1} c_{m+1} + \cdots + b_1 c_{2m-1} + c_{2m} &= 0 \end{aligned}$$

Maticově můžeme tuto soustavu přepsat do podoby

$$\begin{pmatrix} c_1 & c_2 & \cdots & c_m \\ c_2 & c_3 & \cdots & c_{m+1} \\ \vdots & \vdots & & \vdots \\ c_m & c_{m+1} & \cdots & c_{2m-1} \end{pmatrix} \begin{pmatrix} b_m \\ b_{m-1} \\ \vdots \\ b_1 \end{pmatrix} = - \begin{pmatrix} c_{m+1} \\ c_{m+2} \\ \vdots \\ c_{2m} \end{pmatrix}$$

Vyřešením této soustavy pak zbývá dopočítat koeficienty $a_0, \cdots, a_m$, pro něž platí následující vztahy

$$\begin{aligned} a_0 &= c_0, \\ a_1 &= c_1 + b_1 c_0, \\ a_2 &= c_2 + b_1 c_1 + b_2 c_0, \\ &\vdots \\ a_m &= c_m - \sum_{i=1}^{m} b_i c_{m-1}. \end{aligned}$$

Podrobněji viz [7].

Hypergeometrická funkce

Koeficienty Padého aproximace lze vyjádřit pomocí hypergeometrických funkcí.

Definice 3. Hypergeometrická řada je mocninná řada tvaru $\sum_{n=0}^{\infty} c_n x^n$, přičemž $c_0 = 1$, a existují polynomy P, Q s koeficienty u nejvyšších mocnin rovných jedné, $st(P) = p \geq 0$, $st(Q) = q > 0$, takové, že platí

$$\frac{c_{n+1}}{c_n} = \frac{P(n)}{Q(n)} \cdot \frac{1}{(n+1)}, \quad n \in \mathbb{N}_0 \tag{3.10}$$

Pokud rozložíme P, Q na (komplexní) kořenové činitele, dostaneme

$$\frac{c_{n+1}}{c_n} = \frac{(a_1 + n)(a_2 + n) \ldots (a_p + n)}{(b_1 + n)(b_2 + n) \ldots (b_q + n)} \cdot \frac{1}{(n+1)}, \quad n \in \mathbb{N}_0 \tag{3.11}$$

Tuto situaci zachycujeme zápisem

$$\sum_{n=0}^{\infty} c_n x^n = {}_pF_q[a_1, \ldots, a_p; b_1, \ldots, b_q](x) \tag{3.12}$$

Vyjděme z definice a upravme si (3.11) do následující podoby

$$c_{n+1} = c_n \cdot \frac{(a_1+n)(a_2+n)\dots(a_p+n)}{(b_1+n)(b_2+n)\dots(b_q+n)} \cdot \frac{1}{(n+1)}, \quad n \in \mathbb{N}_0 \tag{3.13}$$

Z rekurentního vyjádření jednoduše odvodíme vztah pro c_n

$$c_n = \frac{\prod_{j=1}^{p}(a_j)_n}{\prod_{j=1}^{q}(b_j)_n} \cdot \frac{1}{n!}, \tag{3.14}$$

kde symbol $(a)_n$ nazýváme Pochhammerovým symbolem a je definován vztahy

$$(a)_n := a(a+1)\dots(a+n-1), \ n \in \mathbb{N}, \quad (a)_0 = 1. \tag{3.15}$$

Hypergeometrickou řadu pak můžeme z rovností (3.14) a (3.12) přepsat do tvaru

$${}_pF_q[a_1,\dots,a_p;b_1,\dots,b_q](x) = \sum_{n=0}^{\infty} \frac{\prod_{j=1}^{p}(a_j)_n}{\prod_{j=1}^{q}(b_j)_n} \cdot \frac{x^n}{n!} \tag{3.16}$$

Zajímavým faktem je přímý vztah mezi koeficienty Padého aproximace pro exponenciální funkci a hypergeometrickou funkcí [8]. Namísto řešení soustavy rovnic a dopočítávání koeficientů můžeme tyto koeficienty vypočítat explicitně dle následujícího vztahu

$$R_{m,n}(x) = \frac{{}_1F_1[-m;-m-n](x)}{{}_1F_1[-n;-n-m](-x)} \tag{3.17}$$

Právě díky tomuto vztahu se metoda *scaling and squaring* s využitím $[m/m]$-tého rozvoje Padého aproximace velice jednoduše implementuje. Pokud totiž v rovnici (3.17) položíme $m = n$, pak dostaneme rovnost

$$R_{m,m}(x) = \frac{{}_1F_1[-m;-2m](x)}{{}_1F_1[-m;-2m](-x)} \tag{3.18}$$

a v této formě se po rozvinutí ${}_1F_1[-m;-2m](z)$ čitatel liší od jmenovatele pouze znaménkem u lichých mocnin hodnoty x. Rozviňme si tedy pravou stranu rovnosti pomocí (3.16) a (3.15). Dalším pozorováním zjistíme, že využívaný Pochhammerův symbol $(-m)_k$ bude pro $k > m+1$ roven nule[3]. Číselné řady se nám tímto zredukují na konečné součty,

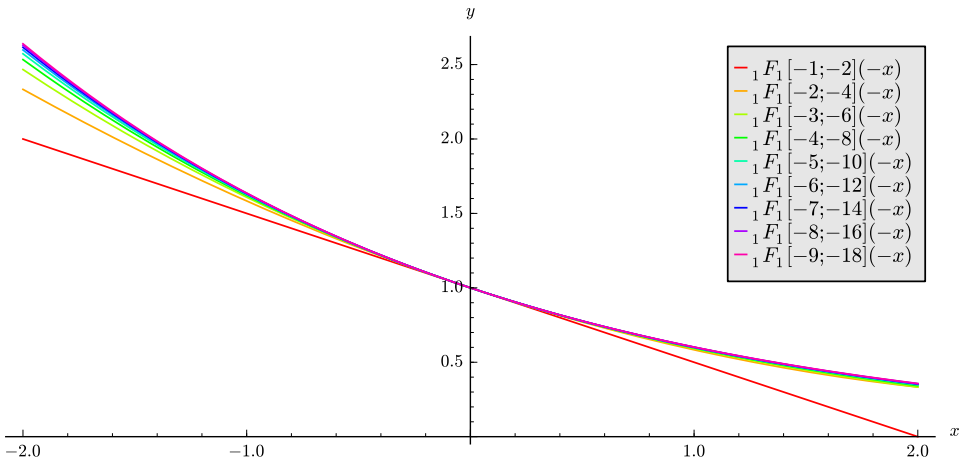
$$\frac{{}_1F_1[-m;-2m](x)}{{}_1F_1[-m;-2m](-x)} = \frac{\sum_{k=0}^{\infty} \frac{(-m)_k}{(-2m)_k} \cdot \frac{x^k}{k!}}{\sum_{k=0}^{\infty} \frac{(-m)_k}{(-2m)_k} \cdot \frac{(-x)^k}{k!}} = \tag{3.19}$$

$$= \frac{\sum_{k=0}^{\infty} \frac{(-m)(-m+1)\cdots(-m+k-1)}{(-2m)(-2m+1)\cdots(-2m+k-1)} \cdot \frac{x^k}{k!}}{\sum_{k=0}^{\infty} \frac{(-m)(-m+1)\cdots(-m+k-1)}{(-2m)(-2m+1)\cdots(-2m+k-1)} \cdot \frac{(-x)^k}{k!}} = \tag{3.20}$$

$$= \frac{\sum_{k=0}^{m} \frac{(-m)(-m+1)\cdots(-m+k-1)}{(-2m)(-2m+1)\cdots(-2m+k-1)} \cdot \frac{x^k}{k!}}{\sum_{k=0}^{m} \frac{(-m)(-m+1)\cdots(-m+k-1)}{(-2m)(-2m+1)\cdots(-2m+k-1)} \cdot \frac{(-x)^k}{k!}} \tag{3.21}$$

[3]Všimněme si, že k vynulování členu $(-2m)_k$ nikdy nedojde

Pro úplnost ještě zbývá ukázat existenci inverzní matice, jež stanoví celý jmenovatel výrazu (3.21). Protože tuto funkci budeme využívat při implementaci metody *Scaling and squaring*, uveďme si zde důležitost parametru škálování pro tuto metodu. Tento parametr je volen tak, aby matice, z níž se bude počítat Padého aproximace, měla Euklidovskou normu menší než jedna. Faktem je, že na okolí bodu 0, např. $(-1, 1)$ jsou funkční hodnoty ${}_1F_1[-m; -2m](-x)$ blízko 1. Tuto skutečnost ilustruje následující graf 3.1. Odtud plyne, že matice, jejíž inverzi počítáme, je blízko jednotkové matici a je tedy regulární.



Obrázek 3.1: Funkce ${}_1F_1[-m; -2m](-x)$ na okolí (-2, 2) pro různá m

I když teď víme, že uvažovaná inverze matice existuje, nemusí být úplně zřejmé kterým způsobem správně dopočítat zlomek (3.21). Zda čitatel násobit inverzí jmenovatele zleva, či zprava. Následujícím lemmatem si dokážeme, že obě varianty vedou na stejný výsledek.

Lemma 1. Nechť $C, B \in \mathbb{C}^{n\times n}$ jsou navzájem komutativní matice a nechť existuje inverze k matici B, potom je matice C komutativní s B^{-1}.

Důkaz.

$$\begin{aligned} C &= C \\ C &= CBB^{-1} \quad / \text{ Komutativita } C \text{ a } B \\ C &= BCB^{-1} \quad / \cdot B^{-1} \text{ zleva} \\ B^{-1}C &= CB^{-1} \end{aligned}$$

□

Pro využití výše uvedeného lemmatu stačí ukázat, že čitatel a jmenovatel (3.21) splňují všechny jeho předpoklady. Pokud se budeme dívat na čitatele a jmenovatele uvažovaného zlomku, jako na různé polynomy, do kterých dosazujeme stejnou matici, pak nám stačí si uvědomit, že součiny těchto polynomů

jsou komutativní. Díky tomu, že polynomy obsahují mocniny pouze jedné matice, můžeme se opřít o komutativitu matice samé se sebou, která pro matice našeho problému obecně platí. Pro komplexní koeficienty polynomu je pak komutativita s jinými koeficienty dána z axiomů množiny komplexních čísel. Díky tomu víme, že maticový součet čitatele komutuje se maticovým součtem jmenovatele. Taky jsme si v předešlých odstavcích ukázali, že pro jmenovatel našeho zlomku existuje jeho inverze. S využitím lemmatu 1 máme jistotu, že je jedno, zda násobíme čitatele vypočtenou inverzí zleva nebo zprava.

Na vyčíslitelnost (3.21) se můžeme také dívat jako na soustavu

$$N = DX \quad N, D, X \in \mathbb{C}^{n \times n}, \tag{3.22}$$

kde N chápeme jako matici čitatele a D jako matici jmenovatel odkazovaného zlomku. Náročnost vyřešení této soustavy je srovnatelná s náročností výpočtu inverzní matice. Oproti přístupu s inverzní maticí si ale ušetříme jedno násobení matice s maticí. [9]

Kapitola 4

Implementace Krylovovy metody

V této kapitole se budeme blíže věnovat samotné implementaci Krylovovy metody. Rozebereme zde klíčové části této metody a posléze okomentujeme implementaci jako jeden celek.

Většina částí Sage je psána v jazyce Cython, který by se dal charakterizovat jako kompilovaný Python. Kód psaný v Cythonu je při svém překladu nejprve přeložen do syntaxe jazyka C a následně zkompilován do binární podoby. Takto přeložený kód má oproti běžnému Pythonu řádově lepší výkonnost a blíží se efektivitě jazyka C. Implementaci však budeme pro jednoduchost uvádět v jazyce Python. Později v kapitole 5 zjistíme, že naše algoritmy stráví většinu času vykonáváním metod optimalizovaných knihoven a překompilováním našeho kódu si nikterak nepomůžeme. Jak jsme si v kapitole 2 krátce uvedli, Sage je projekt, který integruje mnoho existujících *open-source* programů a knihoven pro matematické výpočty. Všechny tyto menší projekty pak sám využívá v rámci definic svých metod. Fungují buďto jako podpůrné prvky v metodách nebo jsou tyto knihovny samy redefinovány pro datové typy samotného Sage. Zároveň nám Sage dovoluje v rámci notebooku importovat jemu známé knihovny a využívat jejich metody přímo v jejich originální podobě. V naší implementaci se jedná zejména o využití několika metod pro manipulaci s maticemi, zejména pak metody pro jejich násobení. Naprogramovali jsme tedy dvě verze pro porovnání rychlosti Sage. V první implementaci pro manipulaci s maticemi využíváme pouze metody programu Sage. V ní zároveň předpokládáme na vstupu Sageovský typ matice a vektoru. Ve druhé implementaci se pak spoléháme pouze na originální metody a datové typy knihovny NumPy[4]. Předpoklad je takový, že verze Numpy by měla být daleko rychlejší, neboť využívá jednoduché datové typy a optimalizované knihovní funkce.

[4]Tyto dvě implementace budeme často zkráceně rozlišovat jako *Sage*, nebo *NumPy* verze Krylovovy metody

Zmiňované dvě realizace Krylovovy metody mají jednu nevýhodu, a to množství parametrů, které do ní vstupují. Na jednu stranu by se tento fakt dal brát jako výhoda, protože znalý uživatel má tímto větší možnosti pro ovlivnění přesnosti výpočtu. Na druhou stranu, pro běžné použití je třeba zajistit dobrou přesnost s výchozími parametry. Speciální implementace Krylovovy metody řeší problém s parametry pro algoritmus *Scaling and squaring*. V této verzi je použit algoritmus J. Highama pro odhad parametru pro škálování a řád Padého aproximace. Algoritmus J. Highama jsme implementovali jak pro verzi Sage, tak i verzi s NumPy maticemi.

Naše hlavní funkce, reprezentující Krylovovu metodu, se bude jmenovat `expKrylov(...)` a v následujícím odstavci si popíšeme všechny její parametry.

```
1 #Predpis funkce maticove exponencialy metodou Krylovova
2 def expKrylov(A,v,arnoldi_m=25,pade_m=13,pade_s=0,method='pade')
```

Pomineme-li matici `A` a vektor `v`, jakožto nutné parametry, zbývají nám čtyři vstupní parametry. Prvním z nich je parametr velikosti Krylovova podprostoru, vstupujícího do Arnoldiho algoritmu. Následující dva parametry vstupují do algoritmu *Scaling and squaring*, využívající Padého aproximaci. Jeden z nich je parametr škálovatelnosti a druhý ovlivňuje rozvoj Padého aproximace. Posledním parametrem `method` dovolujeme uživateli volbu pro způsob výpočtu maticové exponenciály. Uživatel má na výběr mezi výchozí metodou `'pade'` a metodou `'higham'`. Nutno zde podotknout, že výběrem metody `higham` zůstanou parametry `pade_m` a `pade_s` nevyužity[5]. I když by se tyto dvě zmiňované metody daly rozdělit do dvou funkcí, rozhodl jsem se pro toto řešení z důvodu plánované integrace naší implementace do systému Sage. Inspiroval jsem se pro toto řešení z již existující implementace metody `exp(...)`, která taktéž dovoluje volbu metody pro výpočet maticové exponenciály.

Problematice optimálního parametru pro volbu dimenze Krylovova podprostoru se budeme blíže věnovat v kapitole 5, protože v žádném zjištěném zdroji o Krylovově metodě není jasně zdokumentováno, jakým způsobem je vhodné tento parametr volit.

Narozdíl od parametru pro Arnoldiho algoritmus jsou volby optimálních parametrů pro metodu *Scaling and squaring* popisovány hned v několika článcích. Běžným přístupem je volba škálovacího parametru s jako nejmenší mocninu dvou, pro kterou platí $\|A\|/s \leq 1$. Díky vztahu $(e^{A/s})^s$ lze pak velmi efektivně dopočítat výsledek [3]. Tento parametr škálovatelnosti může někdy působit problémy v nepřesnostech výpočtů. Tento jev je nazýván efekt přeškálování a je blíže popsán v [2]. Existuje však algoritmus, jež tento problém řeší na základě charakteristiky vstupní matice. Jeho autorem je J. Higham a tímto algoritmem se budeme blíže zabývat v sekci 4.5.

[5] Highamův algoritmus si tyto parametry odhaduje sám. Tato metoda je blíže popsána v sekci 4.5

4.1 Obecný popis implementace Krylovovy metody

V těle funkce `expKrylov` ve skutečnosti nic zásadního nepočítáme. Kontrolujeme zde vstupní parametry a voláme dvě podrutiny, které odvedou celou práci. U kontroly ověřujeme, zda parametr matice reprezentuje nějaký datový typ matice, zda se jedná o čtvercovou matici. Obdobnou proceduru provádíme i pro parametr vektoru. Poté, co vstupní parametry projdou kontrolou, voláme funkci počítající Arnoldiho algoritmus. Jejím výstupem je matice obsahující bázi Krylovova podprostoru (V_m) a projekce původní matice `A` na ní (H_m). Po Arnoldiho algoritmu následuje samotný výpočet exponenciály nad zprojektovanou maticí A v Krylovově podprostoru. Tento výpočet probíhá předpisem metody *Scaling and squaring*. Pro tuto metodu potřebujeme parametr `s` pro škálování a `m`, jakožto [`m`/`m`]-tý řád rozvoje Padého aproximace. Oba tyto parametry můžeme dostat od uživatele. V případě, že se tak nestane, si je budeme muset dopočíst, či odhadnout sami. Po provedení těchto dvou rutin máme všechny stavební kameny definice Krylovovy aproximace maticové exponenciály. Stačí nám tedy pronásobit normu vstupního vektoru v s maticí V_m a výstupem Padého aproximace, jakožto maticovou exponenciálu matice H_m. Konečnou aproximací $e^A v$ je pak první sloupec popsaného součinu.

Výše uvedený popis je pro Sage i NumPy verzi naší implementace jednotný. Všechny odlišnosti se odehrávají uvnitř popisovaných podrutin, které si v následujících odstavcích popíšeme. Kontroly správnosti parametrů adekvátně zakomponujeme i do funkcí Arnoldiho algoritmu a padého aproximace, protože tyto funkce mohou být využity i samostatně. Při využití hlavních funkcí jsou kontroly zbytečné, protože sebemenší chyby se odchytí v hlavní funkci `expKrylov`.

4.2 Typy matic

Ještě než si popíšeme jednotlivé implementace algoritmů, přehledně si shrneme typy matic v Sage a v Pythonovských knihovnách NumPy a SciPy.

4.2.1 Matice v Sage

Sage třídí matice v závislosti na tom, do kterého okruhu spadají prvky dané matice. Okruh můžeme matici předat parametrem do konstruktoru matice, nebo si jej Sage vydedukuje z přímo předávaných prvků matice. Okruh matice jde změnit pomocí třídní metody `change_ring(new_ring)`. Operace s maticemi s prvky různých okruhů je samozřejmě možná a výsledek je automaticky převeden do správného typu. Tabulka 4.1 však neobsahuje všechny okruhy, které Sage pro matice nabízí. Příkladem může být okruh celých čísel $\mathbb{Z}$ mod p a několik dalších. Pro matice nad těmito speciálními okruhy však nemá smysl

Značení	Okruh	Typ
ZZ	celých čísel	`matrix_integer_dense`
QQ	racionálních čísel	`matrix_rational_dense`
RR	reálných čísel	`matrix_generic_dense`
CC	komplexních čísel	`matrix_generic_dense`
SR	symbolických výrazů	`matrix_symbolic_dense`
RDF	RR s omezenou přesností	`matrix_real_double_dense`
CDF	CC s omezenou přesností	`matrix_complex_double_dense`

Tabulka 4.1: Okruhy matic v Sage

počítat maticovou exponenciálu. Poznámku si ještě zaslouží matice typu RDF a CDF. Ty jsou Sagem navrženy pro rychlejší maticové operace a ze zdrojových kódů můžeme narazit na jisté napojení na knihovnu NumPy. Uvedené typy ve výše uvedené tabulce reprezentují husté matice. Všechny uvedené typy matic mohou být převedeny do podoby řídkých matic, avšak pouze ZZ a QQ mají svou přímou reprezentaci. Ostatní okruhy jsou u řídkých matic převedeny do obecného okruhu.

4.2.2 Matice v NumPy a SciPy

Knihovna NumPy zavádí pouze definici hustých matic a mají taky jistý způsob volby datového typu dle jejich prvků. Pracuje se zde ale s primitivními datovými typy a ne s okruhy. Mezi nejběžnější datové typy patří `int64`, `float64` a `complex128`. Matici v NumPy můžeme mít jako instanci třídy `numpy.matrixlib.defmatrix.matrix` nebo ji mít uloženou ve dvourozměrném poli `numpy.ndarray`. První varianta nabízí o něco širší paletu metod pro maticové operace. Sageovské metody však *defaultně* převádí své matice do NumPy typů právě do `numpy.ndarray`. Knihovna SciPy pak podporuje ve svých funkcích matice zadefinované knihovnou NumPy a dále rozšiřuje množinu datových typů matic o implementace 7 druhů řídkých matic.

Typ	Popis
`bsr_matrix`	*Block Sparse Row matrix*
`coo_matrix`	*COOrdinate matrix format*
`csc_matrix`	*Compressed Sparse Column matrix*
`csr_matrix`	*Compressed Sparse Row matrix*
`dia_matrix`	*Sparse matrix with DIAgonal storage*
`dok_matrix`	*Dictionary Of Keys based sparse matrix*
`lil_matrix`	*Row-based linked list sparse matrix*

Tabulka 4.2: Formáty řídkých matic definovaných knihovnou SciPy

4.3 Analýza implementace vnitřních algoritmů

Hlavní odlišnosti všech implementací spočívají ve využití různých knihoven pro manipulaci s maticemi. Protože Sageovská verze je až na implementační detaily shodná s verzí NumPy, popíšeme si jednotlivé implementace vnitřních algoritmů pouze na verzi pro Sageovské typy matic. Dále si pak pro úplnost uvedeme tabulku syntaktických odlišností.

Arnoldiho algoritmus

Po kontrole vstupních parametrů, které v ukázce vynecháme, máme na začátku funkce `arnoldi(...)` v proměnné `R` společný okruh vstupní matice a vektoru. Na druhé a třetí řádce tak inicializujeme nulové matice správného typu a rozměrů. Následujícími dvěma řádky inicializujeme první vektor a matici V. Tyto dvě řádky odpovídají bodu 1 z pseudokódu Arnoldiho algoritmu 3.1. Následující forcyklus na řádcích 8–14 je pouhým přeformulováním řádků 2–9 odkazovaného pseudokódu. Jedinou odlišností je přizpůsobení indexace ve vnitřním forcyklu. Funkce `srange(m)` vrací list celých čísel začínající nulou a končící číslem (`m-1`). Proto má vnitřní forcyklus indexy zvýšené o jedničku. Patnáctý a šestnáctý řádek pak představují jistou nápravu navíc vypočítaných dat. Konec pseudokódu totiž předpočítává vektor do následující iterace, stejně tak úvodní hodnotu do nového řádku matice H.

```
1  def arnoldi(A, v, m = 25):
2      V = matrix(R, A.ncols(), m+1)
3      H = matrix(R, m+1, m)
4      v = v / v.norm()
5      V.set_column(0, v)
6      for j in srange(m):
7          w = A * V.column(j)
8          for i in srange(j+1):
9              H[i, j] = w * V.column(i)
10             w = w - (H[i, j] * V.column(i))
11         H[j+1, j] = w.norm()
12         V.set_column(j+1, w / H[j+1, j])
13     V = V.delete_columns([m])
14     H = H.delete_rows([m])
15     return V, H
```

Padého aproximace

U této krátké funkce po kontrole vstupních parametrů nastavíme správný škálovací faktor a jím pronásobíme vstupní matici `A`. Následně pak voláme funkci `_pade_hypergeometric(A, m)`, která efektivně vypočítá koeficienty Padého

aproximace. Na výstupu pak z této funkce dostane aproximaci exponenciály, pro kterou ještě musíme provést škálovací korekci.

```
1  def expPade(A, m=13, s=0):
2      if s == 0:
3          scale = _compute_optimal_scale(A)
4      else:
5          scale = s
6      A = (1.0/(scale)) * A
7      return (_pade_hypergeometric(A, m))^scale
```

Hypergeometrická funkce

Funkce `_pade_hypergeometric(A, m)`[6] je implementací pravé strany rovnosti (3.21). Na jejím vstupu očekáváme matici `A` a parametr `m`, který nám určuje [`m`/`m`]-tý řád rozvoje Padého aproximace pro exponenciální funkci. S pomocí hypergeometrické funkce dokážeme explicitně vyjádřit koeficienty u jednotlivých mocnin matice `A`. Dokonce mezi koeficienty platí následující rekurentní vztah

$$c_{n+1} = \frac{k-m}{(k-2m)(k+1)} c_n \qquad \text{pro } {}_1F_1[-m;-2m](x)$$

Tento vztah máme v kódu na řádce 10. Připomeňme si taky fakt, že (3.21) je zlomek, kde čitatel se od jmenovatele liší pouze znaménkem u lichých mocnin argumentu. Takto lze efektivně napočítávat čitatel i jmenovatel současně. V našem kódu si tedy udržujeme k-tou mocninu vstupní matice `A` v proměnné `matrixPow`. Tu pak vhodně přenásobeným koeficientem přičítáme k čitateli (`nom`) a jmenovateli (`denom`).

V závěru pak musíme vypočítat „podíl“ těchto dvou matic, jak jsme si blíže popsali v kapitole 3. Pro Sageovskou verzi jsem pro tento výpočet volil metodu `inverse()`[7], protože metoda pomocí řešení soustav rovnic

$$DX = N \quad N, D, X \in \mathbb{C}^{m \times m} \tag{4.1}$$

není podporována pro všechny Sageovské datové typy matic[8]. Jako alternativu jsem mohl zvolit převedení Sageovské matice do NumPy matice, pak provést výpočet soustavy (4.1) pomocí knihovny NumPy a konečně pak převést řešení uvažované soustavy zpět do původního datového typu matice, který mi uživatel předal parametrem. Jak jsme si ale popsali v kapitole 3, postup s řešením soustavy rovnic je rychlejší jen o jedno násobení matic. Pro tuto volbu bychom

[6] Úvodní podtržítko v názvu funkce je v Sage dle konvencí chápáno jako označení privátní metody/funkce.

[7] Tato funkce počítá z volaného datového typu matice matici k ní inverzní

[8] Konkrétně funkce `solve_right(...)` nepodporuje matice typu RDF a CDF

museli investovat navíc čas a paměť pro alokaci kopie matice. Bylo by tedy na místě otestovat pro jak velké matice se ještě vyplatí převody datových typů matic. Tento problém jsem však netestoval a pro čistotu kódu jsem zvolil metodu pomocí inverze. Tuto inverzní matici, reprezentující jmenovatele Padého aproximace, pak stačí vynásobit s jeho čitatelem (`nom`).

```
1  def _pade_hypergeometric(A, m):
2      nom = A.new_matrix()
3      denom = A.new_matrix()
4      matrixPow = identity_matrix(A.ncols())
5      denom_sign = 1
6      coef = 1
7      for k in srange(0, m+1):
8          nom += coef * matrixPow
9          denom += denom_sign * coef * matrixPow
10         coef *= (-m + k) / ((-2*m + k)*(k + 1))
11         matrixPow = matrixPow * A
12         denom_sign *= -1
13     denom_inv = denom.inverse()
14     return denom_inv * nom
```

4.4 Knihovna NumPy

Knihovna NumPy má oproti prostředí Sage hlavní výhodu v tom, že neřeší nepřeberné množství datových typů matic. Husté matice jsou reprezentovány datovým typem `ndarray` a jedná se o primitivní dvourozměrné pole 64-bitových floatů[9]. Pro reprezentaci řídkých matic má NumPy samozřejmě samostatné datové typy. Všechny tyto typy podporují námi používané operace a mají stejnou syntaxi, jako husté matice. Následující tabulka shrnuje ty nejzákladnější odlišnosti v syntaxi prostředí Sage a knihovny NumPy.

Význam	Sage	NumPy
Násobení matic	`A * B`	`A.dot(B)`
Výběr i-tého sloupce matice	`A.column(i)`	`A[:, i]`
Výběr i-té řádky matice	`A.row(i)`	`A[i, :]`
Nastavení i-tého sloupce	`A.set_column(i,v)`	`A[i,:]=v`
Výpočet normy matice	`A.norm()`[10]	`linalg.norm(A)`
Výpočet normy vektoru	`v.norm()`	`linalg.norm(v)`
Výpočet i-té mocniny matice	`A^i`	`matrix_power(...)`[11]

Tabulka 4.3: Hlavní rozdíly v syntaxi programu Sage a knihovny NumPy

[9] 64 bit - 11 bitů pro exponent, 52 bitů pro mantisu

4.5 Modifikace dle J. Highama

Jak jsem již výše zmiňoval, algoritmus J. Highama obecně řeší dva parametry metody *Scaling and squaring* pro výpočet maticové exponenciály. Pro první z parametrů, parametr `s`, má tento algoritmus následující tabulku

m	θ_m
3	1.495585217958292 e-2
5	2.539398330063230 e-1
7	9.504178996162932 e-1
9	2.097847961257068
13	5.371920351148152

Tabulka 4.4: Konstanty pro algoritmus Highama

K této tabulce konstant je nutné poznamenat, že její hodnoty byly zvoleny speciálně pro výpočty prováděné v 64-bitových doublech [2].

Hlavním přínosem této metody spočívá v autonomním výběru parametrů m a s pro Padého aproximaci. Autor v této metodě také popsal efektivní výpočet maticových mocnin pro minimalizaci počtu násobení matic. Tato metoda ale taky počítala koeficienty Padého aproximace metodou řešení soustavy rovnic. Tuto část jsem ve své implementaci nahradil Hypergeometrickou funkcí. Pro tuto metodu je nutné na začátku vypočítat jednotkovou normu vstupní matice. Poté se pro hodnoty 3, 5, 7, 9 parametru m porovnává vypočtená norma s přidruženou hodnotou θ_m výše uvedené tabulky. Porovnání se provádí vzestupně od nejmenšího m po největší a pro konečný stupeň Padého aproximace se vezme první m, pro které je θ_m větší nebo rovno předpočítané jednotkové normě vstupní matice A. Pro kterékoli takto vybrané m je škálovací parametr s roven jedné. Tuto skutečnost v rámci kódu můžeme sledovat uvnitř forcyklu na řádcích 3–5. Pokud je naopak jednotková norma matice A příliš velká, aby vyhovovala výše popsanému kritériu, bere se jako parametr m hodnota 13 a škálovací parametr s je vypočten dle následující formule

$$s = \lceil \log_2 \|A\|_1 / \theta_{13} \rceil$$

[10] Pro řídké datové typy matic tato metoda není správně implementována

[11] Plné volání funkce je tvaru `numpy.linalg.matrix_power(A, int(i))`

```
def expHigham(A):
    m_norm = linalg.norm(A, 1)
    for m in srange(4):
        if m_norm < HIGHAMS_CONST[m][1]:
            return _pade_hypergeometric(A, HIGHAMS_CONST[m][0])
    scale = ceil(log(m_norm / HIGHAMS_CONST[4][1], 2))
    M = (1/(2^scale)) * A
    return (_pade_hypergeometric(M, 13))^(2^scale)
```

Výstup funkce `_pade_hypergeometric(...)` pak výsledně umocníme škálovacím parametrem s. K umocňování pomocí operátoru „`^`“ se hodí poznamenat, že Sage má pro tento maticový operátor efektivní implementaci metody *Square and multiply*.

Kapitola 5

Testy a měření výkonnosti implementace

Pro testování a měření kvality kódu existuje mnoho přístupů. V této kapitole si představíme koncept psaní automatizovaných testů pro Sage, ukážeme si nástroj pro profilování kódu a nakonec se budeme věnovat měření přesnosti a efektivity naší implementace Krylovovy metody v závislosti na vstupních parametrech.

5.1 Doctesting v Sage

Nedílnou součástí implementací rozsáhlých systémů je jistá hierarchie testů, které ověřují správnost jak nejmenších funkcionalit, tak i širší funkčnost a propojenost celého systému. Sage je postaven na jazyce Python a ten v tomto směru vyšel vývojářům vstříc a zavádí zajímavý způsob psaní testů v rámci *docstringů*[12]. Ty jsou pro všechny funkce a metody v Sage povinné a mají svou předepsanou syntaxi. Tyto *docstringy* se dělí do pomyslných bloků, z nichž některé jsou povinné a jiné volitelné. Povinně musí mít každá funkce svůj krátký popis, blok s popisem vstupů (`INPUT:`), blok s popisem výstupů (`OUTPUT:`) a blok, který by měl svými ukázkami demonstrovat funkcionalitu dané metody (`EXAMPLES::`). Poslední povinná položka `EXAMPLES::` však neslouží pouze k ilustraci použití dané funkce, ale je využita v automatizovaných testech před každým *release* nové verze Sage. Mimo tyto povinné bloky existuje ještě řada volitelných a na tomto místě vypíchněme blok `TESTS::`, který má podobnou funkcionalitu jako `EXAMPLES::` s tou výjimkou, že obsahy těchto bloků by už

[12] *Docstring* je speciální druh komentáře přidružený k dané funkci, který slouží k její dokumentaci

neměly být pro uživatele relevantní.

```
def expPade(A, m=13, s=0):
    r"""
    Computes [m/m]'th Padé approximant of exponential function

    Its coefficients are computed by _pade_hypergeometric(A, m)
    which returns divided nominator and denominator of m'th
    degree of Padé approximation.

    INPUT:

    * "A" - input matrix

    * "m" - Integer, degre of Padé approximation

    * "s" - Integer, scaling parameter for
            'scaling and squaring' algorithm

    OUTPUT:

    * [m/m]'th Padé approximation of matrix "A"

    EXAMPLES:

    ::

        sage: A = matrix(RDF, [[2,-2],[1,1]])
        sage: expPade(A) # tol 1e-13
        [  2.7418832886396   -6.568509046621072]
        [ 3.2842545233105387 -0.5423712346712395]

    TESTS:

    Exponential of a diagonal matrix::

        sage: A = matrix(RDF, [[2,0],[0,-3]])
        sage: expA = matrix(RDF, [[exp(2),0],[0,exp(-3)]])
        sage: (expA - expPade(A)).norm() < 1e-10
        True

    """
    ... implementation
```

Na výše uvedené ukázce můžeme vidět *docstring* k naší funkci `expPade`

dle konvencí platných pro Sage. V následujícím odstavci si ještě představíme pravidla pro psaní testů. Podrobnější informace o *coding style* se pak můžeme dočíst na jeho oficiálním webu [10].

5.1.1 Testy a jejich konvence

Pro psaní testů existuje v Sage několik pravidel. Blok pro spouštění testů, či ukázek definujeme dvěmi dvojtečkami. Všechny testy nebo *examples* musí být od sebe odděleny prázdnou řádkou. K označení nastavovací části testu se na začátek řádky píše řetězec „`sage:`“. Výsledek vyhodnocení poslední takto označené řádky se pak porovnává na shodu s uvedeným předpokládaným výstupem. Ten už není nijak prefixován. Občas při psaní testů nastane situace, že přesnost výsledku testované funkcionality může záviset na verzi jádra systému či podobných okolnostech. Pro tyto situace nabízí Sage speciální modifikátory, kterými nastavujeme toleranci ke srovnávání s reálným výsledkem. V naší ukázce se jedná o modifikátor *`# tol <řádová přesnost>`*. Těchto modifikátorů je hned několik. Jejich seznam s instrukcemi použití můžeme nalézt na webu projektu Sage [10]. Automatický test souboru s naší implementací pak spouštíme příkazem

```
$ sage -t nazev_testovaneho_souboru
```

Výstup pak vypadá následovně

```
Doctesting 1 file.
sage -t expKrylov.sage
   [26 tests, 6.37 s]
----------------------------------
All tests passed!
----------------------------------
Total time for all tests: 6.8 seconds
   cpu time: 6.2 seconds
   cumulative wall time: 6.4 seconds
```

5.2 Profilování kódu

Dalším způsobem, jak se můžeme dívat na kvalitu našeho kódu a případně hledat jeho slabiny, je použití *profileru*. Profilováním kódu zjistíme počet volání jednotlivých funkcí naší implementace a jejich dobu vykonávání pro námi zadaný vstup. Na profilaci kódu je v současné době v Sage dostupný modul `prun`. Ten bohužel do svého výpisu zahrnuje i různá interní volání a celkový výstup je složitě čitelný. Existuje však nástroj `line_profiler`, který se dá do

Sage přidat. Ten dokáže ve velmi přehledné podobě přiřadit statistiky v kombinaci s profilovaným zdrojovým kódem. Pro ukázku si zde uvedeme výstup profilace funkce `expKrylov(...)` a `arnoldi(...)`.

Timer unit: 1e-06 s
Total time: 1.3135 s

Hit	Time	% Time	Line contents
			`def expKrylov(A, v, arnoldi_m=25, ...`
1	1 201 075	87.9	`  V, H = arnoldi(A, v, arnoldi_m)`
1	2	0.0	`  if method == 'pade':`
1	90 519	6.6	`    H_exp = expPade(H, m=pade_m, s=...)`
			`  elif method == 'higham':`
			`    H_exp = expHigham(H)`
			`  else`
			`    raise NotImplementedError('Metho..`
1	74 018	5.4	`  return v.norm()*V*H_exp.column(0)`

Tabulka 5.1: Přepis výstupu `line_profileru` pro funkci `expKrylov()`

Timer unit: 1e-06 s
Total time: 1.17342 s

Hit	Time	% Time	Line contents
			`def arnoldi(A, v, m=25):`
1	1 034	0.1	`  V = matrix(R, size, m+1)`
1	638	0.1	`  H = matrix(R, m+1, m)`
1	464	0.0	`  v = v / v.norm()`
1	4 955	0.4	`  V.set_column(0, v)`
26	981	0.1	`  for j in srange(m):`
25	122 495	10.8	`    w = A * V.column(j)`
350	6 301	0.6	`    for i in srange(j+1):`
325	456 119	40.2	`      H[i, j] = w * V.column(i)`
325	448 225	39.5	`      w = w - (H[i, j] * V.column(i))`
25	5 301	0.5	`    H[j+1, j] = w.norm()`
25	54 790	4.8	`    V.set_column(j+1, w / H[j+1, j])`
1	33 083	2.9	`  V = V.delete_columns([m])`
1	1 052	0.1	`  H = H.delete_rows([m])`
1	1	0.0	`  return V, H`

Tabulka 5.2: Přepis výstupu `line_profileru` pro funkci `arnoldi()`

Uvedené výpisy z `line_profileru` byly pořízeny pro náhodnou matici `A` typu RDF o rozměrech 1000×1000, náhodném vektoru správného rozměru

a defaultních parametrech pro Arnoldiho algoritmus. Jak si můžeme všimnout, nejvíce času stráví Krylovova metoda v Arnoldiho algoritmu, neboť jedině tam se provádí operace s maticí původního velkého rozměru. Přepočteno na koeficient `Time/Hit` je nejkritičtějším místem násobení matice `A` s vektorem. To se děje tolikrát, kolikrát je zvolena dimenze Krylovova podprostoru. Také díky tomuto profileru se mi podařilo zjistit, že rozměrnější typy Sageovských matic `ZZ`, `QQ`, `RR` a `CC` mají operaci násobení s vektorem oproti typům RDF, CDF, či numpy maticím slabě optimalizovanou. Pro ilustraci matice typu `RR` stejných rozměrů 1000×1000 stráví v rámci Arnoldiho algoritmu 97.7% času při operaci násobení matice s vektorem a celková doba provádění tohoto algoritmu je průměrně 51.7 s. Oproti tomu doba, kterou pořebuje Arnoldiho algoritmus pro svoje vykonání s maticí v NumPy formátu se stejnými prvky, činí průměrně 0.08 s.[13]

5.3 Měření kvality

K ověřování kvality implementace Krylovovy metody jsem přistoupil několika způsoby. Za prvé jsem se pokoušel zjistit, zda existuje vztah mezi optimální hodnotou dimenze Krylovova podprostoru a velikostí exponenciované matice. Ve zdrojích, ze kterých jsem čerpal, se o této veličině píše jen jako o „vhodně“ zvolené hodnotě, která by měla být značně menší než řád původní matice. Za druhé jsem testoval přesnost mé metody a jako poslední test jsem prováděl měření doby výpočtu jednotlivých metod.

5.3.1 Měření přesnosti výpočtu

Pro měření přesnosti jsem se opíral o znalost přesné maticové exponenciály. Tu jsem si vyrobil ze znalosti exponenciály pro diagonální matici. Pro přehlednost uvedu tyto závislosti v matematickém zápisu. Označme si diagonální matici $D \in \mathbb{C}^{n\times n}$ a regulární matici $P \in \mathbb{C}^{n\times n}$. Matice A, pro kterou budeme hledat maticovou exponenciálu pomocí Krylovovy metody, má pak následující tvar

$$A := P^{-1}DP.$$

Následně jsem využil vlastnost (1.4) a sestavil si výslednou exponenciálu matice.

$$e^A := P^{-1}e^D P.$$

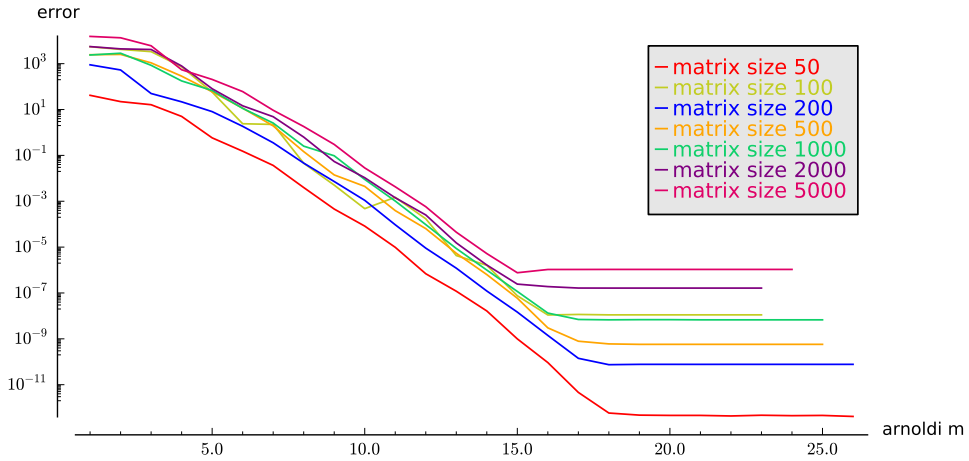
Pak jsem si vygeneroval náhodný vektor $v \in \mathbb{C}^n$ a vypočítal exponenciálu skrze mou implementaci. Za výslednou chybu jsem bral normu rozdílů dvou popisovaných způsobů výpočtu.

$$err := \|e^A v - expKrylov(A, v)\|_2$$

[13] Výpočty byly prováděny na lokální instalaci Sage (verze 6.5, Release Date: 2015-02-17) na procesoru Intel® Core™ i5 CPU M 560

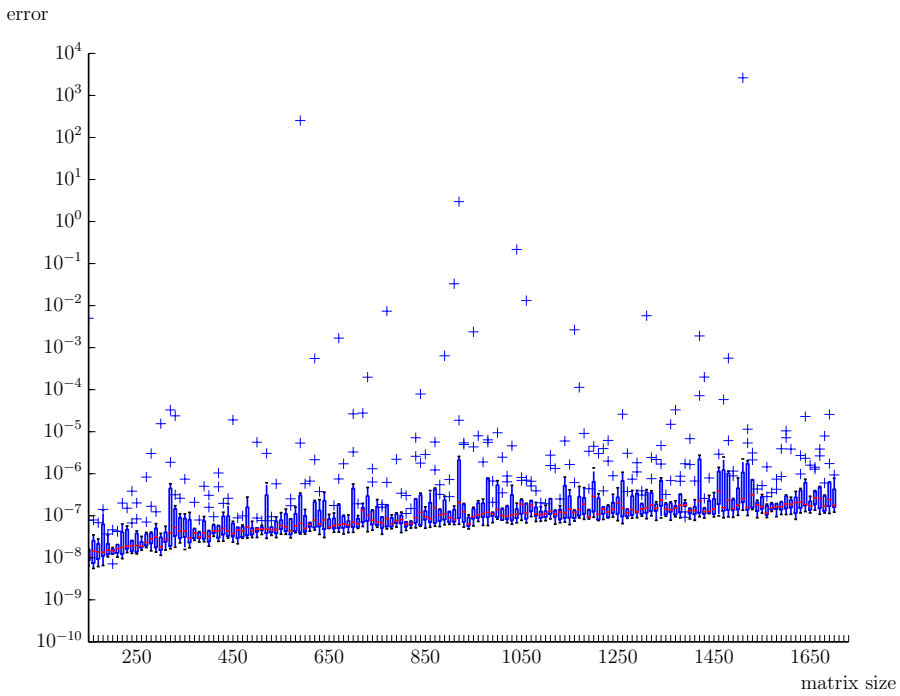
Dimenze Krylovova podprostoru

Jeden ze stěžejních parametrů ovlivňující výkonnost naší implementace je parametr `m` pro Arnoldiho algoritmus. Pro měření optimální hodnoty tohoto parametru jsem využil výše popsanou konstrukci matice s předem známou exponenciálou. Vektor a regulární matici P jsem pak náhodně generoval. Pomocí naší implementace Krylovovy metody jsem počítal maticovou exponenciálu na vektoru. Za chybu jsem bral normu z rozdílu předem známé hodnoty a napočítaného výsledku. S tímto nastavením měření jsem iteroval přes parametr Arnoldiho algoritmu až do chvíle, kdy se napočítaná chyba výsledku řádově rovnala posledních k iterací. Pro níže vyobrazený graf jsem volil $k = 8$. Z tohoto grafu si můžeme všimnout, že chyba klesá s parametrem Arnoldiho algoritmu pro matice různých velikostí prakticky stejně. Zajímavé je, že se řádová chyba pro všechny matice od hodnoty parametru 20 nemění.



Obrázek 5.1: Vliv parametru Arnoldiho algoritmu na přesnost výpočtu

Pro ověření naměřené hodnoty m pro Arnoldiho algoritmus jsem postupně generoval náhodné matice, jako v předchozím případě. Pro ty samé matice jsem vypočítal maticovou exponenciálu pomocí naší implementace Krylovovy metody s parametrem Arnoldiho algoritmu rovným 15. Chybu výpočtu jsem rovněž bral jako normu z rozdílu přesné exponenciály a té námi napočtené. Pro každou velikost matice jsem generoval 10 reprezentantů. Naměřené chyby každé desetice jsem v grafu 5.2 znázornil krabicovým grafem.

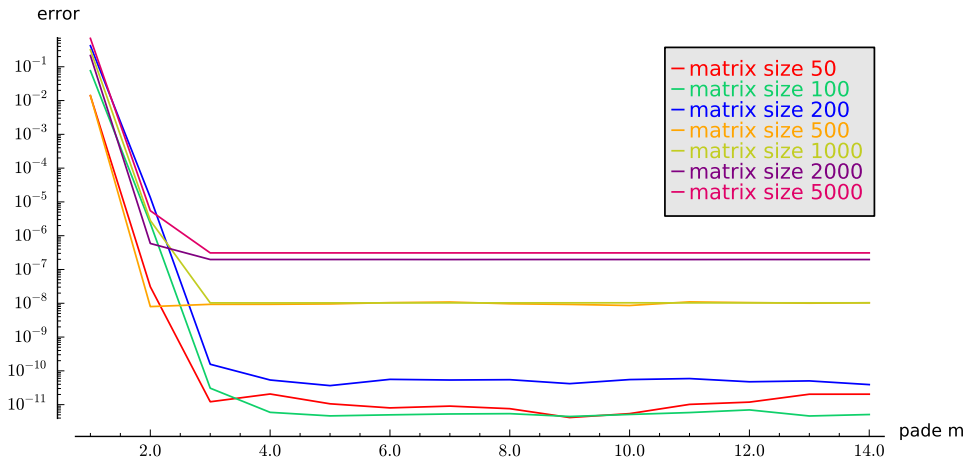


Obrázek 5.2: *Boxplot* naměřených chyb pro jednotlivé rozměry matic při výpočtech jejich exponenciál pomocí naší implementace Krylovovy metody s Arnoldiho parametrem rovným 15. Každá „krabicová“ část v grafu je vymezena hranicemi 1. a 3. kvartilu a „vousy“ představují nejnižší údaj 1,5 IQR spodního kvartilu a nejvyšší údaj 1,5 IQR horního kvartilu. Křížky reprezentují *outliary*.

Z uvedeného krabicového grafu si můžeme všimnout opticky velkého počtu *outliarů*. Tito reprezentanti tvoří nicméně pouze zanedbatelné procento ze všech testovaných matic. Jak se ukázalo tyto matice vykazují velké chyby i pro největší parametry Krylovovy metody. Matice pro graf 5.2 byly voleny v rozměrech od 50 do 1700 s krokem 10. Celkový počet testovaných matic byl tedy 1650. Když pomineme zmiňované *outliary*, pak se medián chyb u jednotlivých matic drží v rozmezí 10^{-8} až 10^{-6}.

Vliv řádu Padého aproximace na přesnost exponenciály

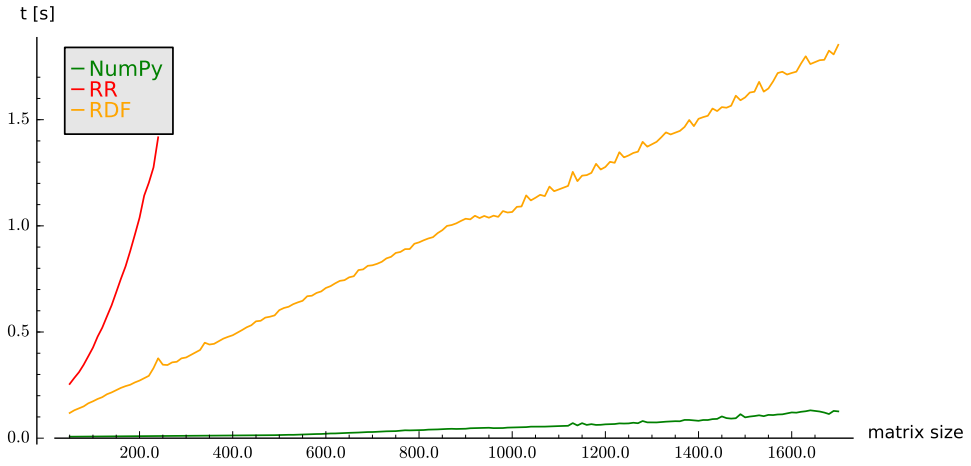
Se stejným principem jako u měření vlivu parametru pro Arnoldiho algoritmus jsem otestoval vliv řádu Padého aproximace, použité v naší implementaci, na přesnost výpočtu. Parametr vstupující do Arnoldiho algoritmu jsem pro tento test nastavil na hodnotu 25. Z níže uvedeného grafu můžeme usoudit, že řádová chyba konverguje už od hodnoty 4.



Obrázek 5.3: Vliv řádu Padého aproximace uvnitř Krylovovy metody na přesnost výpočtu

5.3.2 Měření výkonnosti

U těchto měření se budeme snažit porovnat efektivitu výpočtu obou našich implementací, které používají různé typy matic. V rámci tohoto testu budeme iterovat přes řády matic od velikosti 50 do 1700 s krokem 10. Pro každou zvolenou velikost matice budeme pětkrát generovat náhodnou matici a vektor, převedeme je do různých formátů a následně změříme dobu, kterou zabere výpočet funkce `expKrylov` resp. `expKrylovNumpy`. Ze součtu těcho časů jednotlivých typů matic pak vypočteme aritmetický průměr. Tyto hodnoty jsou vyneseny v grafu 5.5.



Obrázek 5.4: Graf znázorňující dobu výpočtu maticové exponenciály v závislosti na typech a velikosti vstupní matice

5.3.3 Porovnání `exp()` a `expKrylov(...)`

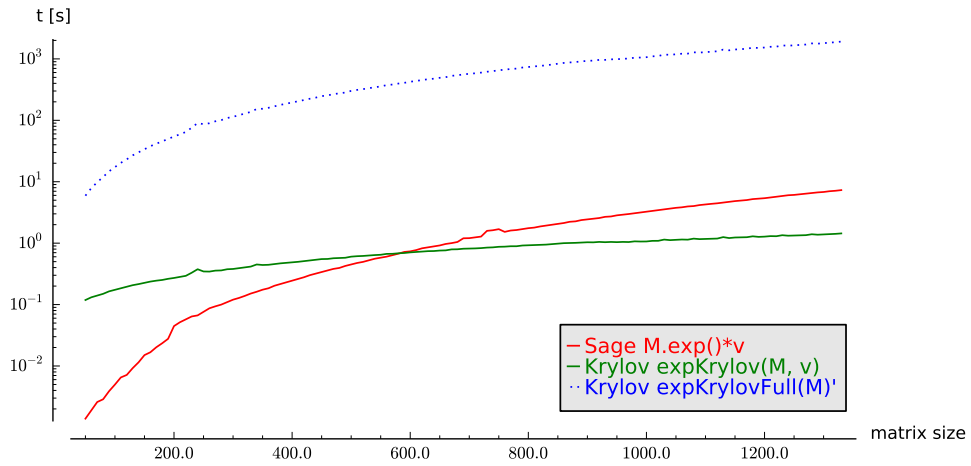
Porovnání stávající Sageovské metody `exp()` vůči naší implementaci Krylovovy metody může být krajně spekulativní, protože obě metody se hodí pro odlišný typ úlohy. Sageovská metoda je bez diskuze velice univerzální, nicméně si neporadí s maticemi vyšších řádů. Naše implementace naopak není až tak univerzální, ale velké matice ji nedělají problémy.

Přístup pro výpočet maticové exponenciály e^A pomocí Krylovovy metody jsme si popisovali v závěru první kapitoly. Vytvoříme si funkci `expKrylovFull(...)`, která bude iterovat přes všechny jednotkové vektory a s ním bude následně volat funkci `expKrylov(A, e_{i},...)`. Výsledné vektory, seřazené do matice, pak představují výsledek maticové exponenciály.

Pro úplnost zmiňme, že těchto jednotkových vektorů je přesně tolik, kolik je roven řád exponenciované matice. Kdybychom u této operace zanedbali složitost zařazování napočtených vektorů do výsledné matice, mohli bychom využít odhad

$$t(\texttt{expKrylovFull(M)}) \approx (\#\text{ sloupců }\texttt{M}) \cdot t(\texttt{expKrylov(M, v)})$$

Pro porovnání se Sageovskou metodou `exp()` spolu s naší funkcí `expKrylov(...)` si do grafu zaneseme právě tuto aproximaci (ozn. `expKrylovFull(M)'`).



Obrázek 5.5: Graf znázorňující dobu výpočtu úlohy e^A v závislosti na zvolené metodě pro její výpočet a velikosti matice. V tomto srovnání jsme použili matice typu RDF

Z grafu můžeme vyčíst, že Krylovova metoda začíná být od matic řádu 600 řadově efektivnější oproti Sageovské funkci `exp()` násobené vektorem `v`.

Kapitola 6

Demonstrace využití maticové exponenciály

Ověření implementovaných metod budeme provádět na fyzikálním problému vedení tepla.

6.1 Rovnice vedení tepla

Pokusme se numericky vyřešit rovnici vedení tepla

$$\partial_t u = c\Delta u + q, \quad u(0) = u_0, \tag{6.1}$$

kde $u : \Omega \to \mathbb{R}$ je neznámá funkce popisující teplotní rozložení v oblasti Ω splňující předepsané hraniční podmínky na hranici oblasti Ω, $q : \Omega \to \mathbb{R}$ je předepsaná funkce popisující zdroje tepla a u_0 je předepsané počáteční rozložení teploty.

Pro demonstraci budeme uvažovat obdélníkovou oblast $\Omega = [0, L_1] \times [0, L_2]$, kterou diskretizujeme pomocí rovnoměrné pravoúhlé mřížky v x-ovém směru po $\Delta_1 = L_1/n$ a v y-novém směru po $\Delta_2 = L_2/m$. Zkonstruovaná mřížka je tedy tvořena body o souřadnicích $(x, y) = (i\Delta_1, j\Delta_2)$, kde $i = 0, 1, \ldots, n$ a $j = 0, 1, \ldots, m$. Neznámé funkční hodnoty funkce u pak sledujeme pouze v diskrétních bodech $(i\Delta_1, j\Delta_2)$. Označme si $u_{i,j} = u(i\Delta_1, j\Delta_2)$. V rovnici vedení tepla je potřeba nahradit parciální derivace jejich aproximaci pomocí konečných diferencí. Konkrétně

$$(\partial^2_{x,x} u)(i\Delta_1, j\Delta_2) \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta_1^2}, \tag{6.2}$$

$$(\partial^2_{y,y} u)(i\Delta_1, j\Delta_2) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta_2^2}, \tag{6.3}$$

pro $i = 1, \ldots, n-1$ and $j = 1, \ldots, m-1$. Důležité je si povšimnout, že hodnota $u_{i,j}$ je dána předem kdykoliv $i = 0, n$ nebo $j = 0, m$ (hraniční podmínky).

Našim cílem je tedy vyřešit následující rovnici

$$\partial_t u_{i,j}(t) = c(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta_1^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta_2^2}) + q(\Delta_1 i, \Delta_2 j) \tag{6.4}$$

Pro jednoduchost budeme předpokládat, že funkce q, popisující zdroje tepla, je nulová. V rámci konkrétního příkladu si budeme muset zavést uspořádání pro body $u_{i.j}$, abychom si mohli vyjádřit pravou stranu (6.4) jako součin matice (ozn. M), vyplněnou koeficienty, s vektorem (ozn. y_0), ve kterém budou uspořádány body $u_{i,j}$ počátečního stavu úlohy. Levá strana rovnosti (6.4) bude taky vektor a označíme si jej jako $y(t)$. Tento vztah z odkazované rovnosti si pak můžeme vyjádřit následovně

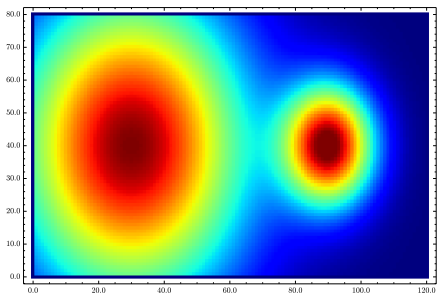
$$y(t) = My_0 \tag{6.5}$$

$$\begin{pmatrix} u'_{0,0}(t) \\ \vdots \\ u'_{i,j}(t) \\ \vdots \\ u'_{n,m}(t) \end{pmatrix} = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 \cdots & \frac{c}{\Delta_1^2} \cdots & \frac{c}{\Delta_2^2} & (-2c(\frac{1}{\Delta_1^2}+\frac{1}{\Delta_2^2})) & \frac{c}{\Delta_2^2} \cdots & \frac{c}{\Delta_1^2} \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} u_{0,0} \\ \vdots \\ u_{i-1,j} \\ \vdots \\ u_{i,j-1} \\ u_{i,j} \\ u_{i,j+1} \\ \vdots \\ u_{i+1,j} \\ \vdots \\ u_{n,m} \end{pmatrix} \tag{6.6}$$

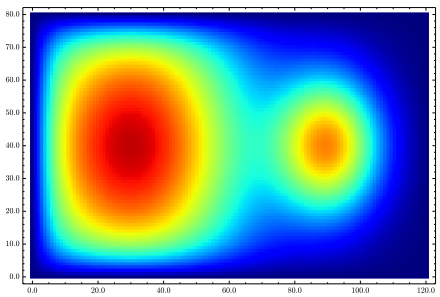
Řešení této rovnosti má pak tvar

$$y(t) = e^{tM} y_0 \tag{6.7}$$

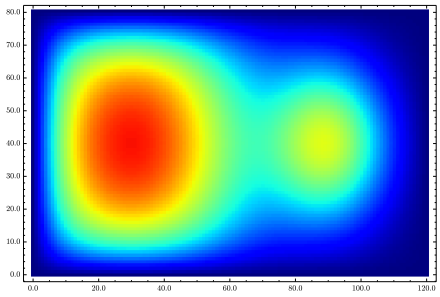
Následující ukázka demonstruje tuto metodu na konkrétním příkladě. Jako počáteční podmínky zde byly zvoleny dvě okolí s různou intenzitou tepla. Nebyl zde využit žádný zdroj tepla ani chlazení a hranice této mřížky se chovaly k šíření tepla neutrálně. Pro výpočty tohoto příkladu jsem využil naší implementaci `expKrylovNumpy(...)` a pro matici M jsem volil reprezentaci řídké matice typu CSR.



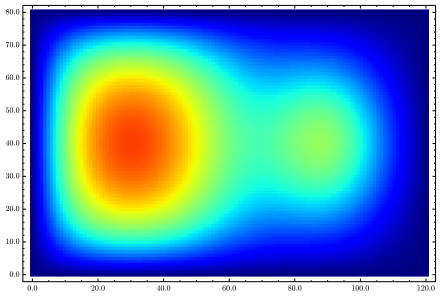
(a) $t = 0$



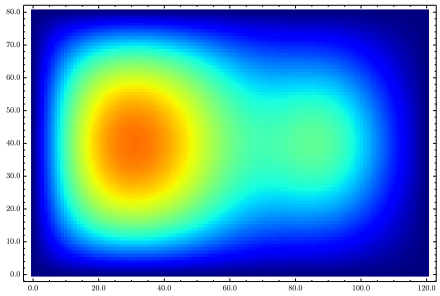
(b) $t = 0.005$



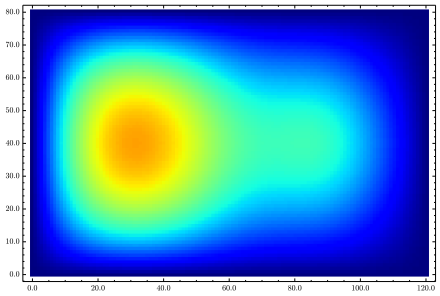
(c) $t = 0.01$



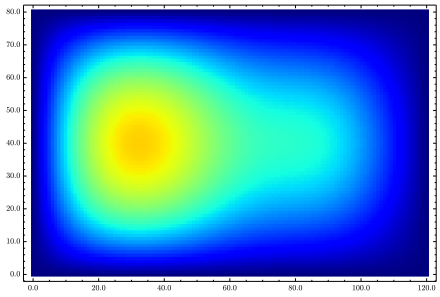
(d) $t = 0.015$



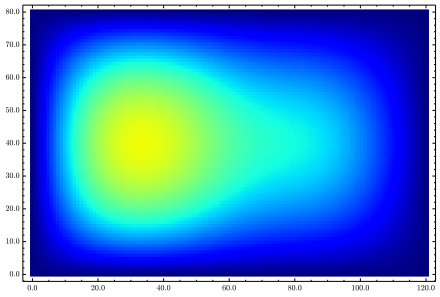
(e) $t = 0.02$



(f) $t = 0.025$



(g) $t = 0.03$



(h) $t = 0.035$

Obrázek 6.1: Vývoj tepelného vedení na dvourozměrné mřížce

Závěr

V této bakalářské práci jsme si v úvodu zadefinovali maticovou exponenciálu. Popsali jsme si nejběžnější přístupy k její numerické aproximaci a následně jsme provedli rešerši existujících nástrojů. Zvýšenou pozornost jsme přitom věnovali *open-source* programu Sage, pro nějž jsme v rámci této práce vyhotovovali implementaci Krylovovy metody.

Krylovovu metodu jsme si podrobně rozebrali v kapitole 3. Díky tomuto detailnímu rozboru jsme dokázali identifikovat její slabá másta. Popsali jsme si také *Padého aproximace*, jež jsou základem pro nejrozšířenější metodu pro výpočet maticové exponenciály. Zejména jsme si uvedli dvojí způsob výpočtu jejich koeficientů pomocí soustavy rovnic a hypergeometrické funkce.

Po rozboru Krylovovy metody jsme se přesunuli k popisu její implementace pro Sage. Na úvod jsme si uvedli dvojí možnosti přístupu k implementaci. Jednalo se o výchzí prostředí Sage a o matematické knihovny NumPy a SciPy. Ukázky a popisy zdrojových kódů základních prvků Krylovovy metody jsme si však uvedli pro prostředí Sage s tím, že jsme následně uvedli syntaktické odlišnosti těchto dvou implementací. Na závěr jsme si uvedli algoritmus Highama, jež umí autonomně volit parametry pro metodu *Scaling and squaring* dle vstupní matice.

Po popisu implementace jsme si v následující kapitole popsali nástroje pro ladění naší implementace. Popsali jsme si rovněž základní konvence pro psaní metod a funkcí pro Sage. Dále jsme provedli měření naší implementace v závislosti na jednotlivých vstupních parametrech. Zde jsme zjistili, že pro relativně dobrou přesnost si vystačíme s relativně malými parametry pro Arnoldiho algoritmus a Padého aproximace. Z provedených měření navíc vyplynulo, že tyto parametry nejsou silně provázány s velikostí vstupní matice. Dále jsme porovnali stávající implementaci maticové exponenciály, dostupnou v Sage, a naší Krylovovou metodou. Zde jsme ale zjistili, že obě metody se hodí pro odlišný typ úlohy a není možné říci, která z nich je ve všech směrech lepší.

V závěru jsme si představili reálný fyzikální problém, týkající se vedení tepla. Pro tuto ukázku jsme v Sage pro zvolené počáteční nastavení úlohy

využili implementaci Krylovovy metody.

Literatura

[1] Durán, A. J.; Pérez, M.; Varona, J. L.: The Misfortunes of a Trio of Mathematicians Using Computer Algebra Systems. Can We Trust in Them? *Notices Amer. Math. Soc.*, ročník 61, 2014: s. 1249–1252.

[2] Al-Mohy, A. H.; Higham, N. J.: A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, ročník 31, 2009: s. 970–989.

[3] Moler, C.; Loan, C. V.: Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM REVIEW*, ročník 45, 2003: s. 3–49.

[4] Saad, Y.: Analysis of some Krylov subspace approximations to the matrix exponential operator. *SIAM J. Numer. Anal.*, ročník 29, 1992: s. 209–228.

[5] Sidje, R. B.: Expokit. A Software Package for Computing Matrix Exponentials. *ACM Trans. Math. Softw.*, ročník 24, č. 1, 1998: s. 130–156.

[6] Wolfram: *Approximate Numerical Linear Algebra* [online]. Dostupné z: `https://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html`

[7] Baker, G. A.; Graves-Morris, P.: *Padé approximants, Seccond edition*, ročník 59. Cambridge University Press, 1996.

[8] A., Z.: *Explicit Multipoint Rational Interpolation Padé Table for Exponential and Power Functions*, ročník 39. American Mathematical Society, 2005.

[9] Cook, J. D.: *Don't invert that matrix* [online]. Dostupné z: `http://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/`

[10] Sage: *General Conventions* [online]. Dostupné z: `http://www.sagemath.org/doc/developer/coding_basics.html`

Příloha **A**

Seznam použitých zkratek

RDF - Real Double Field
CDF - Complex Double Field
IQR - Interquartile range

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `notebook`
  - `Heat_equation.ipynb` ..... Ukázka použití implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_Tomanek_Jakub_2015.pdf` ..... text práce ve formátu PDF
  - `ZZP_Tomanek_Jakub_2015.pdf` ..... zadání závěrečné práce