VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYBRANÁ ROZŠÍŘENÍ ALGEBRAICKÉHO SYSTÉMU OCTAVE

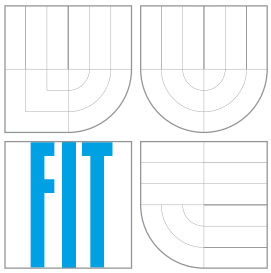
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK SALAČ

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYBRANÁ ROZŠÍŘENÍ ALGEBRAICKÉHO SYSTÉMU OCTAVE

SELECTED EXTENSIONS OF THE ALBEGRAIC SYSTEM OCTAVE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK SALAČ

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2009

ZDE PATRI ZADANI

Abstrakt

Práce se zabývá problematikou řešení soustavy lineárních rovnic v prostředí číslicového počítače. Popisuje základní používané algoritmy s důrazem na jejich silné a slabé stránky. Věnuje se obecným problémům jako je časová složitost a paměťová náročnost daných algoritmů. V závěru popisujeme průběh implementace vybraných procedur do algebraického systému Octave.

Abstract

This work deals with issues linked to solving system of linear equations in the environment of numerical computer. It describes the fundamental algorithms emphasizing their positive as well as negative sides. The work is devoted to general issues such as time complexity and memory demandingness of given algorithms. In the last part, the process of implementation of selected procedures into the algebraic system Octave is described.

Klíčová slova

Octave, matice, řešení lineárních rovnic, předpodmínění, Conjugate gradient squared method, BiConjugate squared method, BiConjugate squared stabilized method

Keywords

Octave, matrix, solution of linear equations , preconditioning, Conjugate gradient squared method, BiConjugate squared method, BiConjugate squared stabilized method

Citace

Radek Salač: Vybraná rozšíření algebraického systému Octave, bakalářská práce, Brno, FIT VUT v Brně, 2009

Vybraná rozšíření algebraického systému Octave

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Tomáše Vojnara, Ph.D.

.....
Radek Salač
18. května 2009

Poděkování

Tímto bych chtěl poděkovat panu Doc. Ing. Tomáši Vojnarovi, Ph.D. za rady, trpělivost a věcné připomínky a paní Mgr. Ivaně Vařekové, zástupkyni společnosti Red Hat za pomoc a rady při vypracování této práce.

Obsah

1 Úvod **3**

2 Octave **5**
2.1 Octave jako program . . . 5
2.2 Základy jazyka Octave . . . 5
2.2.1 Komentáře . . . 6
2.2.2 První příkazy . . . 6
2.2.3 Proměnné . . . 6
2.2.4 Struktury . . . 8
2.2.5 Řízení toku programu . . . 9
2.2.6 Cykly . . . 9
2.2.7 Funkce . . . 10
2.2.8 Hledání chyb . . . 12
2.2.9 Organizace modulů . . . 12

3 Řešení soustav lineárních rovnic **13**
3.1 Úvod do problematiky . . . 13
3.2 Časová složitost . . . 13
3.3 Paměťová náročnost . . . 13
3.4 Základní pojmy . . . 14
3.4.1 Soustava rovnic . . . 14
3.4.2 Transponovaná matice . . . 14
3.4.3 Determinant . . . 14
3.4.4 Lineární kombinace a závislost vektorů . . . 15
3.4.5 Hodnost matice . . . 15
3.4.6 Podmíněnost matice . . . 16
3.4.7 Pozitivně definitní matice . . . 16
3.5 Řešení soustav lineárních rovnic v prostředí číslicového počítače . . . 16
3.5.1 Řídké matice . . . 17
3.6 Algoritmy pro řešení soustav lineárních rovnic . . . 18
3.6.1 Stacionární algoritmy . . . 18
3.6.2 Iterační algoritmy . . . 20
3.6.3 Popis vybraných iteračních algoritmů . . . 21

4 Předpodmínění **23**
4.1 Teorie . . . 23
4.2 Používané algoritmy . . . 24
4.2.1 Jacobiho preconditioner . . . 25

4.2.2 Procedury z rodiny ILU . . . 25
4.3 Implementace v Octave . . . 25

5 Implementace vybraných iteračních algoritmů 27
5.1 Způsob implementace . . . 27
5.2 Postup práce . . . 28
5.3 Popis rozhraní implementovaných procedur . . . 29

6 Testy 31
6.1 Optimalizace kódu . . . 31
6.2 Testovací metodika . . . 32
6.3 Výsledky měření . . . 33

7 Další implementované funkce 35
7.1 Analýza problému . . . 35
7.1.1 Graphviz . . . 35
7.2 Návrh řešení . . . 36
7.3 Vlastní implementace . . . 36
7.4 Porovnání a ukázky grafů . . . 36
7.5 Podpůrné funkce . . . 38

8 Závěr 40

Literatura 41

Seznam příloh 42

A Konfigurace studentského serveru Merlin 43

B Obsah CD 44

Kapitola 1

Úvod

S rozvojem našich znalostí ve všech vědních oborech jde ruku v ruce potřeba být schopen provádět stále náročnější výpočty. Ne vždy máme k dispozici hardwarové vybavení schopné provádět miliardy operací za vteřinu. Často by ani takový výkon nestačil. Proto hledáme cesty, jak provádět všechny typy operací co možná nejefektivněji.

Mezi problémy, se kterými se v praxi potkáváme, patří i řešení soustav lineárních rovnic. V konkrétních situacích již není výjimkou řešení soustav o 10 000 a více neznámých. Takové soustavy je s ohledem na časovou složitost a paměťovou náročnost obtížné řešit klasickými postupy. Navíc lze předpokládat, že velikost soustav, které budeme nuceni řešit, bude stále narůstat.

Jako typický příklad takovéto rozsáhlé úlohy můžeme uvést aplikaci metody konečných prvků[1]. Metoda konečných prvků je numerická metoda, jejíž princip je založen na převodu spojitého modelu do modelu tvořeného konečným počtem prvků. Se vzrůstající hustotou těchto prvků pak vzrůstá i přesnost následných výpočtů. Tato metoda se využívá například k simulaci deformací, proudění tepla nebo namáhání konstrukcí.

Problému řešení soustav lineárních rovnic se budeme věnovat i obecněji. Popíšeme si nejznámější postupy a provedeme diskuzi jejich výhod a nevýhod. Jak si totiž ukážeme pro konkrétní případy může být jeden algoritmus výhodnější než algoritmus druhý. U vybraných algoritmů si popíšeme i jejich časovou složitost a paměťovou náročnost.

Poměrně zajímavou technikou, která může za jistých okolností zjednodušit řešení soustavy lineárních rovnic, je takzvané předpodmínění, anglicky preconditioning. Při použití této techniky dosahují zpravidla iterační algoritmy rychlejší konvergence a dosažené řešení je méně náchylné na chyby vzniklé v důsledku zaokrouhlování.

Pro usnadnění numerických výpočtů existuje celá řada nástrojů, mezi jinými například Octave, což je open-source program sloužící ke složitým numerickým výpočtům. Je podobný Matlabu a ve velké míře je s ním kompatibilní. Je tedy pravděpodobné, že program napsaný pro Matlab půjde spustit v Octave a naopak.

Pro prvotní seznámení se s Octave je vhodné navštívit domovskou stránku projektu[2]. Zde je uceleně na jednom místě k dispozici dokumentace k Octave, odkaz na e-mailovou konferenci, přes kterou je pak možné komunikovat s vývojáři, sekce poskytující informace dobrovolníkům přispívajícím do Octave, a v neposlední řadě jsou zde i zdrojové kódy. Z nich můžeme kompilací získat spustitelnou aplikaci. Druhou variantou je stažení již zkompilované verze Octave pro náš operační systém. Většina linuxových distribucí má Octave ve svém

[1] http://en.wikipedia.org/wiki/Finite_element_method
[2] http://www.gnu.org/software/octave/

repositáři softwaru. Verzi pro další operační systémy, například Windows nebo Mac Os X, je možné získat ze stránek Octave Forge[3], kde je i mnoho dalších rozšíření pro Octave.

Jak jsem již zmínil výše, Octave je open-source, což znamená, že k dispozici jsou i jeho zdrojové kódy. To přináší celou řadu výhod. Zdrojové kódy je možné studovat, vylepšovat a po překladu spouštět i na jiných platformách. Toto vše je u aplikace, která je šířena pouze ve své binární podobě, velice obtížné a v některých případech dokonce nemožné.

Cílem této práce je ve spolupráci s firmou Red Hat implementovat některé chybějící funkce do nástroje Octave. Jedná se převážně o funkce spojené právě s řešením soustav lineárních rovnic. Konkrétně pak:

- BiConjugate Squared Method
- Conjugate Gradient Squared Method
- BiConjugate Squared Stabilized Method

Práce je navíc rozšířena o několik oprav v již stávajícím kódu a implementaci funkce pro vykreslení orientovaného grafu. Jak se totiž ukázalo, v Octave nejsou nástroje, které by uživateli daly jednoduchou možnost takový graf komfortně vykreslit. Procedura gplot se ukázala jako zcela nevhodná, proto jsme byli nuceni najít jiný nástroj, který by uživateli dovolil co možná nejjednodušeji vykreslit orientovaný graf a tím snadno vizualizovat některé přechodové matice. Pro tuto vizualizaci je navíc možné použít další implementované funkce, které vygenerují některé z požadovaných rozložení vrcholů.

[3] http://octave.sourceforge.net/

Kapitola 2

Octave

V této kapitole se seznámíme s programem Octave, jeho původem a historií. Naučíme se, jak s programem pracovat, ovládat ho a jakým způsobem psát jednoduché funkce. V neposlední řadě si řekneme i něco málo o nástrojích pro hledaní chyb, které jsou v Octave dostupné, a které nám mohou v mnoha případech usnadnit práci. Na závěr si povíme o způsobu, jakým jsou funkce v Octave organizovány.

2.1 Octave jako program

GNU Octave je program sloužící ke složitým numerickým výpočtům. Je plně ovladatelný z příkazové řádky, avšak existují i nadstavby, které přidávají grafické rozhraní. Je tvořen množstvím modulů, jež reprezentují jednotlivé funkce, které Octave nabízí svým uživatelům. Tyto moduly mohou být napsány v celé škále různých jazyků jako jsou C, C++ nebo Fortran.

Vznik programu Octave se datuje k roku 1988. Byl vyvinut na dvou amerických univerzitách, konkrétně University of Wisconsin-Madison a University of Texas. Cílem bylo stvořit výukový nástroj pro snazší výpočty spojené s chemickými reakcemi. Hlavním cílem projektu bylo dát studentům k dispozici prostředí, které by je oprostilo od problémů spojených s programováním na nižší úrovni – jako jsou práce s pamětí a problémy spojené s překladem aplikací. Tím měl zároveň umožnit soustředit se pouze na část návrhu algoritmu a provedení vlastního výpočtu.

Další podrobnosti o historii je možné získat přímo na stránkách projektu [1], odkud byla převzata i tato podkapitola.

2.2 Základy jazyka Octave

Jazyk Octave je podobný Pascalu a do velké míry shodný s jazykem Matlabu, s kterým se snaží být co možná nejvíce kompatibilní. Má vestavěný garbage collector, což je nástroj, který hledá paměť, na kterou neodkazuje žádná uživateli přístupná reference. Takovou paměť pak označuje jako volnou a používá ji pro uložení nových dat.

Standardně je Octave spouštěno v interaktivním režimu. To znamená, že ihned po spuštění programu můžeme zadávat příkazy na standardní vstup a ze standardního výstupu číst výsledky.

[1] http://www.gnu.org/software/octave/about.html

V této kapitole budeme dodržovat konvenci, kdy námi zadaný vstup v Octave bude psán strojopisem a předcházet mu bude znak „>“. Řádky, které tímto znakem nezačínají a jsou psány strojopisem, jsou výpisy programu Octave.

2.2.1 Komentáře

Komentáře jsou v Octave uvedeny znakem `#` nebo `%` a pokračují až do konce řádku. V interaktivním režimu nemají velký význam, používají se hlavně při psaní funkce. To je obzvláště důležité pro správný chod funkce *help*, která je součástí Octave. Tato funkce je schopna číst komentáře ve zdrojových souborech a pokud jsou v tomto souboru nalezeny komentáře daného typu, zobrazí uživateli základní informace pro správné použití dané funkce. Mezi jinými například popis vstupních a výstupních parametrů.

2.2.2 První příkazy

Mezi nejjednodušší operace, které můžeme v Octave provést, patří vyhodnocení matematického výrazu.

```
> 1 + 1
ans = 2
```

Octave výraz vyhodnotí a vypíše jeho výsledek. Pokud za výraz napíšeme středník, výpis je potlačen a neprovede se. V tomto příkladu se navíc stalo to, že výsledek byl uložen do proměnné *ans*. O tom si ale více povíme v další sekci.

2.2.3 Proměnné

V předchozí sekci jsme si ukázali, jakým způsobem můžeme spočítat hodnotu výrazu. V praxi však často potřebujeme tuto hodnotu někde uchovávat pro další použití. K tomu slouží proměnné. Proměnná je pojmenované místo v paměti, na které byl uložen náš výsledek nebo jiná hodnota. K tomuto výsledku se pak můžeme vrátit a znovu ho použít. Proměnnou zavedeme prostým přiřazením výrazu, například takto:

```
> a = 1
a = 1
```

Tím jsme vytvořili proměnou a s hodnotou 1. Vzhledem k tomu, že výsledkem operace přiřazení je hodnota výrazu na pravé straně, oznámí nám Octave hodnotu výpisem, jako by se jednalo o výsledek jakékoli jiné operace. Při definování proměnné nemusíme uvádět její typ. Octave je jazyk dynamicky typovaný a typ proměnné se tedy detekuje sám za běhu programu. Mezi v Octave podporované datové typy patří:

reálné číslo (například *3.14*),

komplexní číslo (například *3.14 + 1i*),

matice (tomuto se budeme věnovat později),

řetězec (například ’řetězec’),

struktura (tomuto typu bude věnována samostatná sekce),

pole buněk (tento datový typ v naší práci nenašel uplatnění, zde tedy odkážeme zájemce na dokumentaci[2]).

Pokud nyní budeme chtít s proměnou a pracovat, můžeme napsat příkaz:

```
> b = a + 1
b = 2
```

čímž jsme provedli nové přiřazení výsledku do proměnné b.

S proměnnými jsme se již setkali v sekci 2.2.2, kde jsme si uvedli příklad, kdy byl výsledek uložen do proměnné *ans*. To je speciální proměnná, kam Octave ukládá vždy poslední výsledek, který nebyl uložen jinam.

V našem projektu budeme často pracovat s maticemi. Matici můžeme vytvořit výpisem jednotlivých prvků, kdy prvky ve sloupcích jsou odděleny čárkou nebo mezerou a jednotlivé řádky jsou odděleny středníkem:

```
> A = [1,2; 3 4]
A=
1  2
3  4
```

Tím jsme vytvořili matici o rozměrech 2×2. Pokud chceme vytvořit specifickou matici, například nulovou matici, můžeme si pomoci některými funkcemi jako například *zeros()* nebo *ones()*.

K jednotlivým prvkům přistupujeme voláním:

```
>A(m,n);
```

kde m je číslo řádku a n číslo sloupce.

Velice jednoduše také zjistíme rozměry matice voláním funkcí *rows()* a *columns()*. S maticemi můžeme dále vykonávat operace jako je násobení, dělení, sčítání a odčítaní. Například násobení by mohlo vypadalo takto:

```
> A * [2,3;4,5]
ans =
10  13
22  29
```

Octave umí mimo výše popsaných základních operací i operace pokročilejší. Z nejpoužívanějších třeba:

- výpočet inverzní matice (funkce *inv*())
- výpočet transponované matice (funkce *trans*())
- výpočet hodnosti matice (funkce *rank*())
- výpočet determinantu matice (funkce *det*())

Výpočet determinantu by v praxi mohl vypadat například takto:

```
> det([2,3;4,5])
ans = -2
```

[2] http://www.gnu.org/software/octave/doc/interpreter/Cell-Arrays.html

Pokud chceme, můžeme některé maticové operace provádět i po prvcích matice, čehož docílíme tečkou před daným operátorem. Tím se rozumí, že operace bude provedena nad každým prvkem první matice a korespondujícím prvkem, tedy prvkem na stejných souřadnicích v matici druhé. Zde je příklad, jak by taková operace vypadala:

```
> [1 1;3 4] .* [2,3;4,5]
ans =
 2  3
12 20
```

Tato vlastnost je velice důležitá, jelikož je dobře optimalizovaná. O tom se ale více zmíníme v některé z dalších kapitol.

2.2.4 Struktury

V některých případech potřebujeme data více strukturovat. Často bychom chtěli data, která spolu úzce souvisí, uchovávat pospolu v jedné datové struktuře. Matice je k tomu nevhodná, jelikož může obsahovat hodnoty pouze jednoho typu. Namísto matice lze v takovém případě užít struktury, kterou vytvoříme takto:

```
> S.a = 1
S =
{
    a =  1
}
```

Zde vidíme, že stačí uvést název struktury, následuje tečka a index, do něhož chceme uložit hodnotu. Tvar struktury není nutné nikde explicitně deklarovat, jelikož je tvořen dynamicky za běhu programu. Podívejme se dále na následující příkaz provedený nad strukturou vytvořenou výše:

```
> S.b.b = 'popisek'
S =
{
    a =  1
    b =
    {
        b = popisek
    }
}
```

Vidíme, že Octave nás neomezuje pouze na uchovávání numerických dat a že struktury lze libovolně vnořovat.

V Octave je struktur využito například u funkce *mktime*(), která ze struktury obsahující informace o datu jakými jsou například rok, den a měsíc vrátí unixovou hodnotu času[3]. Více informací o této funkci můžeme získat po vypsání příkazu:

```
> help mktime
```

[3] http://en.wikipedia.org/wiki/Unix_time

2.2.5 Řízení toku programu

V Octave máme klasickou podmínku:

if EXPRESSION1
  `STATEMENT1`
elseif EXPRESSION2
  `STATEMENT2`
else
  `STATEMENT3`
endif

Zde se postupně vyhodnocují výrazy `EXPRESSION1` až `EXPRESSION_N` a vykoná se právě ta větev programu, která následuje okamžitě za prvním výrazem `EXPRESSION`, který vrací jinou než nepravdivou hodnotu (*false*). Jako *false* jsou vyhodnoceny hodnoty:

- 0 – nula,
- " – prázdný řetězec,
- {} – prázdná struktura,
- `[]` – prázdný vektor nebo vektor jiných hodnot, které se vyhodnocují jako *false*.

Všechny ostatní hodnoty lze pak chápat jako pravdivé (*true*).

V případě porovnání dvou hodnot je při správnosti porovnání hodnota tohoto výrazu 1 (tedy *true*), v opačném případě pak 0 (tedy *false*).

Pokud ani jeden z výrazů `EXPRESSION1` ... `EXPRESSION_N` v podmínce nenabývá hodnoty *true*, hodnoty provede se část kódu v bloku `else`. Tento blok je však nepovinný, stejně jako bloky `elseif`.

2.2.6 Cykly

V Octave máme dva druhy cyklů – počítaný a nepočítaný.

Počítaný cyklus vypadá takto:

```
a = 1
for iter = 1:10
   a = a*iter
end
```

Výraz `1:10` se vyhodnotí jako vektor `[1,2,3,4,5,6,7,8,9,10]` a proměnná *iter* pak postupně nabývá všech hodnot tohoto vektoru. Pro všechny tyto hodnoty se následně provede tělo cyklu.

Nepočítaný cyklus má dvě varianty. První varianta s podmínkou na začátku má tvar:

while EXPRESSION
    `STATEMENT`
end

Tělo cyklu se provádí, dokud má výraz `EXPRESSION` pravdivou hodnotu.

Druhá varianta s podmínkou na konci má tvar:

do
    `STATEMENT`
until EXPRESSION

Tělo cyklu se provádí, dokud má výraz `EXPRESSION` nepravdivou hodnotu. V tomto případě se tělo provede minimálně jednou.

2.2.7 Funkce

V Octave můžeme využívat celou řadu vestavěných funkcí. Například výpočet determinantu matice by mohl vypadat takto:

```
> det([ 1,2 ; 3,4])
```

Některé operace pro práci s maticemi jsme si již popsali v sekci 2.2.3. K dispozici máme například i funkce pro převod z desítkové soustavy do binární (*dec2bin*()) a z binární do desítkové (*bin2dec*()). Pro kompletní seznam funkcí doporučuji nahlédnout do manuálu, který je k dostupný i v elektronické podobě[4].

Za pozornost stojí, že v Octave se všechny parametry funkcím předávají hodnotou. Nevýhodou tohoto přístupu je, že často dochází ke zbytečnému kopírování obsahu paměti, což funkce zpomaluje. Na druhou stranu pro skupinu uživatelů, pro které je Octave určeno, je toto pravděpodobně přirozenější chování, jelikož všechny funkce jsou prosty vedlejších účinků. Navíc interpret je schopen toto kopírování do určité míry omezit.

Pokud námi požadovaná funkcionalita v Octave ještě není obsažena, například z důvodu příliš specifických požadavků, můžeme si potřebnou funkci napsat sami. Funkce definujeme pomocí klíčových slov `function` a `endfunction`.

```
>function [Out1,OuT2,...OutN] = jmenoFunkce(In1,In2,...InN)
>...
>endfunction
```

Počet vstupních i výstupních parametrů není pevně dán a je možné předat jen část parametrů, jejich přesný počet je pak uvnitř funkce přístupný v proměnných *nargin* pro vstupní parametry a *nargout* pro parametry výstupní.

V některých programovacích jazycích, například v jazyce *c*, se můžeme setkat s pojmem ukazatel na funkci. Takový ukazatel je proměnná obsahující adresu paměti, na které se nachází daná funkce. Přes tento ukazatel pak můžeme funkci spouštět bez toho, aniž bychom znali její název. Zároveň je možné hodnotu tohoto ukazatele měnit a tím měnit i funkci, která se bude vykonávat. Toho jsme schopni docílit i v Octave takto:

```
> fce = @det;
```

Před název funkce umístíme znak @ a výsledek následně uložíme do proměnné.

Pokud se nám v programu vyskytuje jednoduchý, často se opakující výraz, je možné ho zapsat pomocí anonymní funkce (též se používá výraz lambda funkce). Anonymní funkce se definuje bez klíčového slova `function` a místo jména použijeme opět znak @. Výraz pak uvedeme za touto deklarací. Návratová hodnota funkce je hodnota tohoto výrazu:

```
> @(In1,In2,...InN) EXPRESION;
```

Je vidět, že možnosti, které tato vlastnost poskytuje, nejsou příliš široké a jsou spíše na úrovni makra. Přesto si své místo najde, například při zadávání jednoduchých podmínek, které využijeme při psaní filtrů. Velmi užitečné jsou tyto funkce také v případě, že

[4] http://www.gnu.org/software/octave/doc/interpreter/

píšeme nějakou funkci, uvnitř které jsme nuceni použít již hotovou vestavěnou funkci, která jako jeden ze svých argumentů přebírá právě funkci. Například pokud potřebujeme provést numerickou integraci za použití funkce *quad()*, nemusíme vytvářet integrovanou funkci v globálním jmenném prostoru, ale můžeme si pomoci právě anonymní funkcí:

```
>quad (@(x) sin (x) + sin(x^2), 0, 3));
```

Popis hlavičky

Jak jsme se zmínili v sekci o komentářích, je dobré u námi psaných funkcí dodržovat jisté konvence v psaní komentářů. Před každou námi napsanou funkcí by měl být speciálně formátovaný komentář, pro který platí:

1. Komentář se nachází před samotnou funkcí a každý řádek začíná znaky '`## `'.
2. První řádek má tvar:

   ```
   ## -*- texinfo -*-
   ```

3. Na druhém řádku uvedeme podobu volání funkce s minimálním možným počtem vstupních parametrů:

   ```
   ## @deftypefn {Function File} {} jmenoFunkce (@var{param1})
   ```

4. Na třetím řádku můžeme uvést, jak vypadá volání funkce se všemi nepovinnými parametry:

   ```
   ## @deftypefnx {Function File} {} jmenoFunkce (@var{param1},...@var{paramN})
   ```

5. Následuje popis funkce:

   ```
   ## popis
   ```

6. Dále pak popis jednotlivých argumentů:

   ```
   ## popis @var{param1} parametru 1
   ## popis @var{paramN} parametru N
   ```

7. Poté následuje volný řádek: '`## `'.
8. Dále je zde popis návratových hodnot:

   ```
   ## popis výstupního parametru @var{out1}
   ## popis výstupního parametru @var{out2}
   ```

9. A celý výpis ukončíme řádkem:

   ```
   @end deftypefn
   ```

2.2.8 Hledání chyb

Při psaní vlastních funkcí je často potřeba provádět optimalizaci a hledání chyb. K tomu se běžně používá aplikace nazývaná debugger. U kompilovaných jazyků je většinou nutné, aby aplikace byly zkompilovány speciálním způsobem, který usnadňuje hledání chyb a umožňuje přístup k dodatečným informacím. U skriptovacích jazyků a jazyků fungujících v prostředí virtuálního stroje je situace jednodušší. Většinou jsme schopni tyto informace získávat bez větších potíží. Octave lze také přepnout do ladícího režimu a tím si ulehčit vývoj. Slouží k tomu trojice příkazů:

- *debug_on_warning(1)* : Přepne Octave do ladícího režimu při výskytu chyby úrovně warning, což je chyba, se kterou se Octave dokáže vypořádat bez naší pomoci, ale pravděpodobně značí chybu v algoritmu.
- *debug_on_error(1)* : Přepne Octave do ladícího režimu při výskytu chyby úrovně error. Toto jsou chyby, ze kterých se Octave zotavit nedokáže a algoritmus se zastaví.
- *dbstop('jmenoFunkce',1)*: Přepne Octave do ladícího režimu uvnitř funkce *jmenoFunkce* na řádku 1. Nutno dodat, že číslo řádku, na kterém dojde k přepnutí do ladícího režimu, je pouze orientační údaj. Octave provádí různé optimalizace při překladu a tato hodnota tedy málokdy odpovídá skutečnému číslu řádku v zdrojovém kódu.

V ladícím režimu jsme schopni nechat si vypsat obsah libovolné proměnné prostým vypsáním jejího názvu. Taktéž můžeme krokovat algoritmus voláním funkce *dbnext*. Přestože se zdá na první pohled tento postup nepříliš pohodlný, poskytuje nám mnoho užitečných informací a je plně postačující.

2.2.9 Organizace modulů

Jako jednu z výhod skriptovacího jazyka Octave jsme si uvedli, že není nutné nové funkce překládat. Tyto funkce jsou načteny automaticky a interpretovány za běhu programu. Aby však bylo možné tyto funkce načíst a interpretovat, musí být uloženy v souborech stejného jména s příponou *m*. Například funkce *bcg* tedy bude uložena v souboru *bcg.m*. Tento soubor pak musí být uložen v jednom z předem definovaných adresářů. Seznam těchto adresářů lze získat příkazem *path()*. Příkazem *addpath(adresář)* jsme schopni do tohoto seznamu adresář přidat, příkazem *rmpath(adresář)* pak můžeme adresář ze seznamu odstranit.

Kapitola 3

Řešení soustav lineárních rovnic

V této kapitole si popíšeme problém řešení soustav lineárních rovnic a představíme si některé používané postupy. Provedeme srovnání těchto algoritmů a popíšeme jejich silné a slabé stránky.

3.1 Úvod do problematiky

Řešení soustav lineárních rovnic je problém, se kterým se setkáváme často nejenom v matematice, ale i v mnoha jiných vědních odvětvích. Přestože se může zdát, že v tomto směru není možné a snad ani nutné vyhledávat nové postupy, opak je pravdou. S postupujícím pokrokem ve všech vědních disciplínách vyvstává potřeba řešení rozsáhlejších problémů s rostoucí přesností.

3.2 Časová složitost

Dnes již nejsou výjimkou soustavy o 10 000 proměnných. U nich se ve velké míře projeví časová složitost použitého algoritmu. Například algoritmus s časovou složitostí $O\left(n^3\right)$ bude muset vykonat 1 000 000 000 000 operací. Počet operací prudce narůstá a můžeme se dostat do stavu, kdy se doba pro řešení problému pro nás stane neakceptovatelnou, obzvláště provádíme-li daný výpočet opakovaně.

Řešením může být získání výkonnějšího hardwaru, ale tím vyřešíme jenom následky, ne samotný problém. Navíc výkonnější hardware je často drahý a ne vždy dostupný. Mnohem výhodnější je tedy zvolit takový postup, který by snížil časovou náročnost. Pokud bychom pro výše zmíněný příklad našli postup s časovou složitostí $O\left(n^2\right)$, celkový počet operací by byl 10 000krát menší.

3.3 Paměťová náročnost

Problém s časem zmíněný v předešlé sekci však není jediným problémem, se kterým se musíme vypořádat. Druhým, neméně významným problémem je paměťová náročnost. Veškerá data potřebná pro chod algoritmu je nutné uchovávat v paměti. V případě nedostatku paměti dochází ke swapování, což dále zpomaluje běh programu. Budeme předpokládat, že v nejjednodušším případě jediné, co budeme uchovávat, je vlastní soustava rovnic. Taková soustava bez dodatečných optimalizací spotřebuje $O\left(n * (m+1)\right)$ paměti, kde n je počet

rovnic a m počet neznámých. Proto i zde budeme hledat způsob, jak tuto soustavu uložit efektivněji a paměťovou náročnost tak snížit.

3.4 Základní pojmy

Nejprve si vysvětlíme základní pojmy, které se budou vyskytovat v následujícím textu, abychom s nimi mohli dále pracovat.

3.4.1 Soustava rovnic

Mějme jako příklad soustavu rovnic:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ & & & & \vdots & & & & \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array} \tag{3.1}$$

Toto je obecná soustava m rovnic o n neznámých a vektorem pravých stran $\vec{b}$. Cílem řešení je najít takové hodnoty neznámých $x_1, \dots, x_n$, kdy budou splněny všechny rovnosti.

3.4.2 Transponovaná matice

Pojem transponovaná matice, který si nyní zavedeme, nachází využití například ve funkci *bicg*, popsané v sekci 3.6.3.

Nechť $A = (a_{ij})$ je matice typu $m \times n$. Pak matice $A^T = (a_{ij}^T)$ je typu $n \times m$, kde

$$a_{ij}^T = a_{ji}$$

pro $i \in \{1, 2, \dots m\}$ a $j \in \{1, 2, \dots n\}$, se nazývá transponovaná matice k matici A. Tedy transponovanou matici získáme z původní matice záměnou řádků za sloupce. Převzato z[4].

O matici, pro kterou platí:

$$A = A^T$$

hovoříme jako o *symetrické*.

3.4.3 Determinant

Vzhledem k tomu, že některé další funkce využívají hodnotu determinantu matice, zavedeme i tento pojem. Následující odstavec byl převzat z[1].

Determinantem čtvercové matice řádu n nazýváme součet všech součinů n prvků této matice takových, že v žádném z uvedených součinů se nevyskytují dva prvky z téhož řádku ani z téhož sloupce. Každý součin přitom násobíme čísly r a s, kde r představuje číslo -1 umocněno na hodnotu příslušnou pořadí prvních indexů a s číslo -1 umocněno na hodnotu příslušnou pořadí druhých indexů.

Existují jednoduché vzorce pro výpočet determinantů matic o rozměrech 1×1, 2×2 a 3×3. Konkrétně platí:

$$det\,(a) = a. \tag{3.2}$$

[1] http://cs.wikipedia.org/wiki/Determinant

Pro matici o rozměru 2 platí:

$$det \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} = a_1 b_2 - a_2 b_1. \tag{3.3}$$

Pro rozměr 3 platí:

$$det \begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} = a_1 b_2 c_3 + c_1 a_2 b_3 + b_1 c_2 a_3 - a_3 b_2 c_1 - a_1 b_3 c_2 - a_2 b_1 c_3. \tag{3.4}$$

Často však potřebujeme znát i determinanty matic vyšších řádů. Pro ně existuje několik vzorců, které je možné použít[2]. Pro představu si uvedeme pouze jediný, který využívá Laplaceův rozvoj:

$$det\,(A) = (-1)^{k+1}\, a_{k1} M_{k1} + (-1)^{k+2}\, a_{k2} M_{k2} + \ldots + (-1)^{k+n}\, a_{kn} M_{kn} \tag{3.5}$$

zde A je původní matice, k pořadové číslo prvku rozvoje, a_{kn} je prvek matice A na pozici k, n a konečně M_{kn} je subdeterminant (neboli *minor*) matice A vzniklý odstraněním k-tého řádku a n-tého sloupce z matice A.

Za povšimnutí stojí rekurze plynoucí z tohoto zápisu, která způsobuje poměrně značnou náročnost algoritmu.

3.4.4 Lineární kombinace a závislost vektorů

Lineární kombinací se rozumí operace:

$$a_1 \vec{v}_1 + a_2 \vec{v}_2 + \ldots + a_n \vec{v}_n \tag{3.6}$$

kde $a_1, a_2, \ldots a_n$ jsou skaláry a $\vec{v}_1, \vec{v}_2, \ldots \vec{v}_n$ jsou vektory.

Vektory $\vec{v}_1, \ldots \vec{v}_n$ označíme jako lineárně závislé, pokud existují takové nenulové koeficienty $a_1, a_2 ... a_n$, pro které platí:

$$0 = a_1 \vec{v}_1 + a_2 \vec{v}_2 + \ldots + a_n \vec{v}_n \tag{3.7}$$

Pokud tato podmínka neplatí, označíme dané vektory jako lineárně nezávislé. Pro lineárně závislé vektory platí, že jsme schopni minimálně jeden z vektorů vyjádřit jako lineární kombinaci vektorů zbývajících.

3.4.5 Hodnost matice

Hodnost matice je dalším pojmem velice úzce souvisejícím s řešitelností soustavy rovnic. Jedná se o číslo, které vyjadřuje počet lineárně nezávislých řádků matice a značí se jako

$$h(M),$$

kde M je daná matice.

[2] http://www.aristoteles.cz/matematika/linearni_algebra/determinanty/determinanty-a-matice-sarrusovo-pravidlo.php

Dle Frobeniovy věty pak můžeme určit řešitelnost dané soustavy. Tato věta zní[3]:

> „Nehomogenní soustava lineárních algebraických rovnic má řešení pouze v případě, že hodnost matice soustavy $h(A)$ je rovna hodnosti rozšířené matice soustavy $h(A|\vec{b})$. Pokud je $h(A)$ rovno počtu neznámých, má soustava jedno řešení; pokud je $h(A)$ menší než počet neznámých, je řešení nekonečně mnoho (je-li větší než počet neznámých, nemůže být splněna předchozí podmínka a soustava tedy nemá řešení).“

V této větě padl pojem rozšířená matice soustavy, který bude vysvětlen v sekci 3.5.

3.4.6 Podmíněnost matice

Hodnotu podmíněnosti matice je možné spočítat jako

$$\kappa\left(A\right) = \|A\| \cdot \|A^t\| \tag{3.8}$$

$\|A\|$ značí *normu matice* A.

Platí, že pro dobře podmíněné matice, je hodnota κ malá. Soustavy popsané takovouto maticí jsou pak dobře řešitelné. Více se tomuto budeme věnovat v kapitole 4.

3.4.7 Pozitivně definitní matice

Matice A se nazývá pozitivně definitní, pokud platí:

$$x^T Ax > 0$$

pro každý nenulový vektor x.

3.5 Řešení soustav lineárních rovnic v prostředí číslicového počítače

První, co nás bude zajímat, je způsob reprezentace těchto rovnic v počítači, a to z hlediska paměťové náročnosti a snadnosti přístupu k jednotlivým prvkům. Praxe ukazuje, že nejvhodnější je maticový zápis, a to z důvodu uniformity, kdy uchováváme pouze koeficienty jednotlivých neznámých a pravou stranu rovnice, kterou budeme nazývat vektorem pravých stran. Pravou stranu pak můžeme uchovávat jako součást této matice. Takové matici pak říkáme rozšířená matice soustavy:

$$\left(\begin{array}{ccc|c} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & b_2 \\ a_{m1} & \cdots & a_{mn} & b_3 \end{array}\right)$$

Jinou možností je nechat pravou stranu osamostatněnou:

$$\left(\begin{array}{ccc} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{array}\right) \left(\begin{array}{c} b_1 \\ b_2 \\ b_3 \end{array}\right)$$

[3] http://cs.wikipedia.org/wiki/Soustava_line%C3%A1rn%C3%ADch_rovnic

Z hlediska rychlosti se jako nepatrně lepší jeví první řešení[4]. Důvodem je fakt, že proces alokování paměti je pomalý a našim přáním je počet pomalých operací co možná nejvíce omezit. Se vzrůstající velikostí řešené soustavy se však tento vliv stává zanedbatelný ve srovnání s náročností samotného výpočtu. Proto je v praxi pouze na nás, které řešení zvolíme.

3.5.1 Řídké matice

Pro některé skupiny problémů je častým jevem, že obsahují prvky se stejným, často pak nulovým koeficientem. Logickou otázkou je, zda je nutné takovým prvkům vyhradit místo v paměti. Vezměme jako extrémní příklad jednotkovou matici, tj. matici, která má na diagonále hodnoty 1, mimo ni má prvky nulové. Taková matice o rozměrech 100×100 prvků by zabrala v paměti 10 000 paměťových jednotek v závislosti na velikosti uchovávaného číselného typu. Bylo by tedy vhodné použít strukturu, která nemusí explicitně uchovávat tyto nulové prvky v paměti. Nenulových prvků by pak bylo pouze 100 a paměťová náročnost by signifikantně klesla.

Za tímto učelem se většinou využívá řídké matice. U této struktury se předpokládá, že není-li řečeno jinak, jsou prvky nulové. Obecně můžou mít i jinou hodnotu, avšak v prostředí Octave tomu tak není. Hodnotu těchto prvků pak není nutné uchovávat v paměti a je možné tak snížit paměťovou náročnost. Tento postup si samozřejmě vyžádá i režii navíc. Musíme řešit problém s uchováním informace, na které pozici se nachází prvek nenulové hodnoty. Takový záznam může vypadat například takto: $[x, y]\, value$, kde x a y jsou souřadnice daného prvku a $value$ je hodnota tohoto prvku. Vidíme, že pro zápis tohoto jednoho prvku potřebujeme v paměti vyhradit 3 paměťové jednotky. Jednotková matice z výše uvedeného příkladu by pak zabrala 300 paměťových jednotek, což je stále nesrovnatelně méně než 10 000 v případě klasické matice.

Dalším podstatnějším problémem, který musíme řešit, je přístup k prvkům této matice. V prvním případě, pokud jsme chtěli přistoupit k prvku na souřadnicích $[x, y]$, mohli jsme místo, kde se nachází daná paměťová buňka, přesně najít jednoduchým výpočtem $\mathit{offset} + x \times n + y$. Zde n je počet sloupců na řádku. *Offset* pak značí adresu paměti, od které je matice uložena. Časová složitost tohoto postupu je $O(1)$, tedy konstantní. Pokud však použijeme matici řídkou, tento postup zvolit nemůžeme. Je-li například řídká matice uložena jako seznam prvků nenulové hodnoty, musíme tento seznam projít a porovnáváním zjistit, jestli se zde námi hledaný prvek nachází nebo ne. I v tomto případě sice můžeme použít techniky jako jsou hashované tabulky, ale rychlosti přístupu na úrovní klasické matice v principu dosáhnout nelze.

Pokud tedy nejsme omezení velikostí operační paměti, je výhodné použít matici klasickou. V situacích, kdy jsme nuceni pracovat s velkými soustavami, které mají velký počet prvků stejné hodnoty je pro nás však výhodnější využít matici řídkou. Paměťově nároky jsou v takovém případě mnohem menší a čas nutný pro přístup k jednotlivým prvkům v nejhorším případě roste lineárně s počtem prvků matice. Časová složitost je tedy v nejhorším $O(n)$. V případě potřeby můžeme uchovávat data například ve stromové struktuře. Pak by časová složitost byla $O(log_2 n)$.

[4] V prostředí vestavěných systému je situace odlišná, zde se o přístup k paměti většinou stará programátor sám a je na něm, jaký způsob zvolí.

3.6 Algoritmy pro řešení soustav lineárních rovnic

Nyní, když jsme si zavedli všechny potřebné pojmy, můžeme soustavu rovnic zapsat jako:

$$A\vec{x} = \vec{b},$$

kde A je matice koeficientu, $\vec{b}$ je vektor pravých stran a $\vec{x}$ je vektor řešení soustavy, který se snažíme nalézt. Algoritmy hledající toto řešení soustavy rovnic můžeme rozdělit do dvou základních skupin:

- Stacionární algoritmy, které nám dají přesné řešení (pomineme-li chyby vzniklé v souvislosti s reprezentací čísel v číslicových počítačích).
- Iterační algoritmy, které se k přesnému řešení snaží přiblížit.

3.6.1 Stacionární algoritmy

V této části nás budou zajímat tyto stacionární algoritmy:

- Gaussova eliminační metoda,
- Gaussova-Jordanova redukce,
- Kramerovo pravidlo.

Tyto algoritmy upravují tvar soustavy takovým způsobem, aby postupně vyjádřily všechny neznámé. K tomu můžeme použít pouze operace, které nemění hodnost matice, což je pojem zavedený v podsekci 3.4.5. Jedná se o:

- násobení či dělení řádků číslem různým od nuly,
- přičítání nebo odčítání násobků řádků k jinému řádku,
- transpozice libovolných řádků soustavy.

Gaussova eliminační metoda

Gaussova eliminační metoda převede matici A na horní trojúhelníkový tvar. Následně je možné posoudit řešitelnost celé soustavy. Platí, že pokud je v horní trojúhelníkové matici více nulových řádků, než je počet prvků ve vektoru pravé strany, soustava nemá řešení. Pokud je počet řádků stejný, soustava má právě jedno řešení. V jiném případě má řešení více. V případě, že je soustava řešitelná, postupným vyjadřováním a dosazováním získáme hodnotu všech neznámých.

Celý postup si předvedeme na příkladu. Mějme soustavu:

$$\left(\begin{array}{ccc|c} x & y & z & \\ 3 & 7 & 8 & 6 \\ 3 & 9 & 9 & 11 \\ 3 & 9 & 13 & 19 \end{array}\right)$$

Tuto soustavu převedeme na rozšířenou matici soustavy popsanou v sekci 3.5 a dále na horní trojúhelníkovou matici za použití výše popsaných operací.

$$\left(\begin{array}{ccc|c} x & y & z & \\ 3 & 7 & 8 & 6 \\ 0 & 2 & 1 & 5 \\ 0 & 0 & 4 & 8 \end{array}\right)$$

Nyní můžeme zjistit přesnou hodnotu neznámé z a dosadit do druhého řádku.

$$\left(\begin{array}{ccc|c} x & y & z & \\ 3 & 7 & 8 & 6 \\ 0 & 2 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

Nyní vyjádříme neznámou y a dosazením do prvního řádku získáme výsledný tvar:

$$\left(\begin{array}{ccc|c} x & y & z & \\ 1 & 0 & 0 & -6.8\overline{3} \\ 0 & 1 & 0 & 1.5 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

Časová náročnost tohoto algoritmu je $O(n^3)$ [2].

Gaussova-Jordanova eliminační metoda

Gaussova-Jordanova eliminační metoda je variantou Gaussovy eliminační metody, která upravuje matici rovnou do tvaru, kdy jsou všechny prvky vyjma diagonálních rovny nule. Toho docílíme tím, že v každém kroku postupné eliminace volíme jeden prvek matice pivotem a povolenými operacemi upravíme matici do tvaru, kdy všechny prvky nad i pod pivotem jsou rovny nule. Tato metoda je pomalejší zhruba o 50 procent než prostá Gaussova eliminační metoda [2]. Jejím přínosem však je schopnost řešit více soustav se stejnou levou stranou najednou. Opět uvedeme příklad. Mějme tři soustavy rovnic:

$$\left(\begin{array}{cc|c} 1 & 1 & 3 \\ 2 & 1 & 15 \end{array}\right) \left(\begin{array}{cc|c} 1 & 1 & 6 \\ 2 & 1 & 9 \end{array}\right) \left(\begin{array}{cc|c} 1 & 1 & 8 \\ 2 & 1 & 10 \end{array}\right)$$

Všimněme si stejných levých stran. V tomto případě můžeme všechny rovnice zapsat takto:

$$\left(\begin{array}{cc|ccc} 1 & 1 & 3 & 6 & 8 \\ 2 & 1 & 15 & 9 & 10 \end{array}\right).$$

Výše popsanými operacemi upravujeme. Nejprve odečteme první řádek od druhého:

$$\left(\begin{array}{cc|ccc} 1 & 1 & 3 & 6 & 8 \\ 1 & 0 & 12 & 3 & 2 \end{array}\right).$$

Následně pak odečteme druhý řádek od prvního:

$$\left(\begin{array}{cc|ccc} 0 & 1 & -9 & 3 & 6 \\ 1 & 0 & 12 & 3 & 2 \end{array}\right).$$

Nyní již vidíme řešení všech tří rovnic. Oproti řešení samostatné rovnice jsme museli vykonat operace navíc nad pravou stranou, která obsahovala nyní i výsledky obou předcházejících rovnic, což algoritmus zpomalilo. Pokud bychom však všechny tři rovnice řešili samostatně, prováděli bychom pokaždé stejné operace nad levou stranou, čehož jsme nyní byli ušetřeni a časová náročnost tak poklesla téměř o 2/3.

Kramerovo pravidlo

Kramerovo pravidlo využívá faktu, že pokud vyměníme v původní matici n-tý sloupec za vektor řešení, pak n-tý prvek tohoto vektoru můžeme vypočítat jako

$$x_n = \frac{\det A_n}{\det A}, \tag{3.9}$$

kde A_n je upravená matice. Je vidět, že pro výpočet všech neznámých touto metodou musíme spočítat $n+1$ determinantů n-tého stupně. Pro vyšší řády, jak jsme zmínili v sekci 3.4.3, se tento determinant počítá ze subdeterminantů, což vede na rekurzi a velké množství operací. Konkrétně platí, že časová složitost je $O(n!)$. Pro velké soustavy se tedy tato metoda příliš nehodí. Je však výhodná pro velké počty velmi malých soustav, kde nejsme nuceni používat pro vyjádření determinantu rekurzívní rozvoj, popřípadě speciální soustavy, kde je možné vypočítat determinant jiným, rychlejším, způsobem.

Zajímavou vlastností tohoto algoritmu je také možnost paralerizace úlohy. Výpočty jednotlivých determinantů je možné rozdělit a distribuovat. Také nám umožňuje vypočítat hodnotu pouze jednoho konkrétního prvku, aniž bychom museli řešit celou soustavu.

3.6.2 Iterační algoritmy

V této kapitole se seznámíme s nejpoužívanějšími iteračními algoritmy. Popíšeme si obecné iterační algoritmy a principy, kterými se řídí.

Principy iteračních algoritmů

Iterační algoritmy jsou ve značné míře odlišné od algoritmů stacionárních. Většina iteračních algoritmů je založena na postupu, kdy za pomocí méně přesného výsledku získáme výsledek nový, přesnější. Tuto činnost můžeme provádět až do chvíle, kdy naplňuje v některém ohledu naše očekávání. Většinou se jedná o dosažení požadované přesnosti. Výhodou těchto algoritmů je menší náročnost na výpočetní zdroje. Nevýhodou pak, že řešení nemusí nalézt a to i přesto, že existuje. Občas je proto nutné vyzkoušet více různých metod.

Iterační metody existují pro řešení různých skupin problémů. Například pro určení druhé odmocniny z a můžeme využít rovnice:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right), \tag{3.10}$$

kde x_0 je počáteční odhad, například můžeme zvolit $x_0 = A$.

Lze pak dokázat, že platí:

$$\sqrt{A} = \lim_{n\to\infty} x_n \tag{3.11}$$

Abychom mohli říci, že daný výsledek je pro nás dostatečně přesný, musíme být schopni určit, jak moc se liší od skutečného výsledku. Velikost této chyby se nazývá reziduum. S hodnotou rezidua však v našem případě přímo nepracujeme, jelikož nezohledňuje velikost soustavy, ve které se pohybujeme. Mnohem více nás zajímá hodnota relativního rezidua, kterou budeme značit ε. Vzhledem k vlastnosti iteračních algoritmů, kdy většinou platí, že výsledek následující iterace je přesnější než výsledek iterace předchozí, můžeme také využít rozdílu těchto dvou hodnot k odhadu nepřesnosti. Tato vlastnost, postupně se přibližovat přesnému řešení, se nazývá konvergence. Pokud se od přesného řešení naopak vzdalujeme, nazýváme tento jev divergence.

Abychom zabránili zacyklení algoritmu, stanovíme si podmínky při kterých výpočet ukončíme:

- aktuální hodnota relativního rezidua je menší než požadovaná přesnost,
- metoda diverguje,
- překročili jsme maximální počet iterací.

Zde první podmínka značí úspěšné ukončení výpočtu, kdy jsme našli vyhovující výsledek. Druhá podmínka značí nevhodnost dané metody pro konkrétní úlohu. V určitých případech metoda konverguje ale pouze velmi pomalu, z toho důvodu je vhodné kontrolovat i celkový počet iterací a při dosažení předem daného počtu těchto iterací výpočet ukončit.

3.6.3 Popis vybraných iteračních algoritmů

V této sekci si ve zkratce popíšeme tři základní iterační metody pro řešení soustavy rovnic, které byly v rámci naší práce implementovány. Matematický základ, na kterém jsou postaveny, je poměrně rozsáhlý a přesahuje rámec této kapitoly, proto zde budou uvedeny jen základní principy. Veškeré důkazy, odvození a podrobnosti k matematickému základu lze najít v publikacích [5], [6] a [7].

Conjugate Gradient Squared Method

První iterační metoda, kterou se budeme zabývat, je Conjugate Gradient Squared Method neboli *cgs*. Přestože patří k nejstarším, poskytuje poměrně dobré výsledky. Jejím problémem však je divergence, ke které dochází pokud je soustava špatně podmíněná. Problematiku podmíněnosti soustavy si více popíšeme v samostatné sekci. Zde je prozatím postačující vědět, že i malá změna vstupních parametrů špatně podmíněné soustavy vyvolá velkou změnu výsledku této soustavy. Což je jev pro iterační algoritmy nevhodný.

Podrobný matematický podklad včetně důkazu je popsán v [6]. My zde popíšeme pouze hlavní myšlenku. Lze dokázat, že pro symetrickou, pozitivně definitní matici platí, že minimum funkce

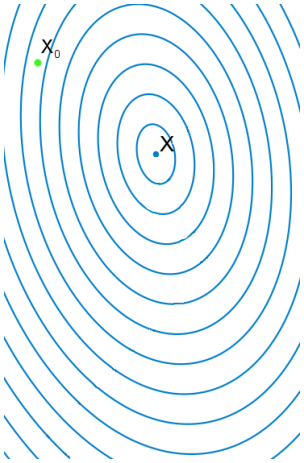
$$f(\vec{x}) = \frac{1}{2}\vec{x}^T A\vec{x} - \vec{b}^T\vec{x} + c;$$

je zároveň řešením soustavy $A\vec{x} = \vec{b}$. V případě, že A vyjadřuje soustavu rovnic o dvou neznámých, můžeme funkci f reprezentovat například obrázkem 3.1, kde elipsy jsou tvořeny množinou bodů, pro které nabývá funkce f stejné hodnoty. X_0 je takzvaný počáteční odhad výsledku. Tuto hodnotu zadá buď uživatel, nebo se zvolí jako výchozí hodnota například vektor nul. Našim přáním je zvolit takový postup, který by nás co nejrychleji zavedl do místa, kde daná funkce f nabývá svého minima, tedy do bodu X.

Problém nalezení řešení se dá dále rozdělit na dva podproblémy. První spočívá v odhadu směru, ve kterém se řešení nachází, druhý problém spočívá v odhadu vzdálenosti tohoto řešení.

Ve funkci *cgs* je celý postup řešen následovně. Směr hledání značme p, vzdálenost řešení pak r. Pro směr hledání platí:

$$p_i = \begin{cases} r_{i-1} + \beta_i p_{i-1} & i > 1 \\ b - Ax & i = 1 \end{cases},$$



Obrázek 3.1: X je minimum dané funkce.

kde i je číslo dané iterace. Pro parametr β platí:

$$\beta_i = \frac{r_{i-1}^T r_{i-1}}{r_{i-2}^T r_{i-2}}$$

Hodnotu rezidua vypočítáme v tomto případě jako:

$$r_i = \begin{cases} r_{i-1} + \alpha_i A p_{i-1} & i > 1 \\ b - Ax & i = 1 \end{cases}$$

$$r_i = r_{i-1} + \alpha_i A p_{i-1}$$

Parametr α spočteme jako:

$$\alpha_i = \frac{r_{i-1}^T r_{i-1}}{p_i^t A p_i}$$

Pro podrobný popis ostatních funkcí můžeme doporučit již výše zmíněnou publikaci [6], popřípadě [5]–níže je uveden pouze jejich nástin.

BiConjugate Squared Method

Procedura BiConjugate Squared Method neboli *bicg* vychází z *cgs*. Odstraňuje problém s občasnou divergencí svého předchůdce ovšem za cenu zpomalení celého algoritmu. Jeden z důvodů je, že pro výpočet rezidua a směru řešení využívá metoda transponovanou matici A^T, viz sekce 3.4.2. Následkem toho musíme v paměti buď uchovávat kromě matice A i její transponovanou verzi, nebo upravit metodu pro přístup k jejím prvkům, což dělá algoritmus méně čitelným.

Zajímavou teoretickou vlastností tohoto postupu je, že při určité modifikaci algoritmu jsme schopni kromě rovnice $A\vec{x} = \vec{b}$ vyřešit současně i rovnici $A^T\vec{x} = \vec{b}$.

BiConjugate Squared Stabilized Method

Metoda BiConjugate Squared Stabilized Method nebo také bicgstab, se zbavuje problémů obou předchozích metod. Většinou konverguje lépe než metoda cgs a nevyužívá transponovanou matici A. V praxi se používá asi nejčastěji.

Kapitola 4

Předpodmínění

V této kapitole si vysvětlíme některé další pojmy související se soustavami lineárních rovnic, hlavně pak pojem preconditioner (tento termín nemá doposud ustálený český ekvivalent, proto budeme používat jeho anglický tvar) a provedeme diskuzi vlivu podmíněnosti soustavy na její řešitelnost. Jako vhodný zdroj údajů, na kterém je postavena tato kapitola, posloužila publikace [1].

4.1 Teorie

Podmíněnost matice (popsáno v sekci 3.4.6) je pojem, který hraje klíčovou roli v řešitelnosti soustavy iteračními metodami. Pokud je hodnota podmíněnosti matice malá, hovoříme o soustavě zadané touto maticí jako o dobře podmíněné. U takové soustavy platí, že malá změna vstupních parametrů vyvolá jen malou změnu parametru výstupních. Pokud je hodnota podmíněnosti matice velká, tak i malá změna vstupních parametrů vyvolá velké změny na výstupu o takové soustavě hovoříme jako o špatně podmíněné.

Můžeme si uvést názorný příklad špatně podmíněné soustavy. Máme-li dvě rovnice o dvou neznámých, tak je možné zakreslit je jako dvě přímky v rovině. Řešení soustavy pak má podobu nalezení jejich průsečíku. Zde špatně podmíněnou soustavou bude případ, kdy jsou tyto přímky téměř rovnoběžné. I malá zaokrouhlovací chyba, které se jen málokdy vyhneme, pak posune průsečík o velkou vzdálenost oproti správné hodnotě.

Špatně podmíněné soustavy jsou obecně hůře řešitelné, jelikož velký vliv na výsledek mají i chyby způsobené zaokrouhlováním. K tomuto problému bude nutně docházet vlivem reprezentace desetinných míst v běžných počítačích. Reálné číslo je totiž uloženo jako mantisa a exponent a pro jejich reprezentaci máme jen omezený počet bitů. Zde nepředpokládáme specializovaný hardware ani software, kde je reálné číslo reprezentované například formátem BCD[1]. Pro iterační metody jsou tyto soustavy velice nevhodné. U většiny algoritmů se s postupně zmenšující odchylkou, zmenšuje i velikost změny nově odhadnutého výsledku od výsledku předchozího. U dobře podmíněných soustav se tímto způsobem přibližujeme správnému řešení, ale u soustav špatně podmíněných i tato nepatrná změna vyvolá velký posun, který ve výsledku může způsobit, že se od správného řešení naopak vzdálíme.

[1] Binary-coded decimal: způsob reprezentace, kdy každému číslu v desítkové soustavě odpovídá přesně daná sekvence bitů v soustavě binární.

Našim přáním by tedy bylo, aby pokud možno všechny námi řešené soustavy byly dobře podmíněné. Ke zlepšení podmíněnosti můžeme využít pravidla, které říká, že pokud máme soustavu:

$$A\vec{x} = \vec{b}, \tag{4.1}$$

můžeme obě strany rovnice vynásobit maticí P a bude platit:

$$PA\vec{x} = P\vec{b} \tag{4.2}$$

Takováto matice se nazývá preconditioner.

Matici P volíme tak, aby výsledná matice PA měla nižší podmíněnost, než původní matice A, a byla tak snadněji řešitelná. Problémem ale je najít takovou matici P. V praxi se používá několik postupů. Prvním je použití obecné matice, která vykazuje dobré průměrné hodnoty, například jednotkové matice. Toto řešení je velice snadné, ale není zaručeno, že pro konkrétní případ se za použití této matice podmíněnost celé soustavy naopak nezhorší.

Druhým, lepším, ale také náročnějším způsobem je použití matice, která nějakým způsobem reflektuje strukturu matice A. Hledání takové matice je často časově náročnou operací, proto se příliš nevyužívá, pokud chceme řešit soustavu jen jedenkrát. Pokud však jsme postaveni do situace, kdy máme konstantní matici A a hledáme řešení pro různá b, je pro nás výhodné si vypočítat pomocnou matici P.

4.2 Používané algoritmy

Mezi jedny z nejpoužívanějších algoritmů pro generování preconditioneru patří algoritmy z rodiny ILU[2]. My se budeme zabývat těmito:

- Jacobiho preconditioner,
- ILU(0) – nejjednodušší varianta algoritmu z rodiny ILU,
- MILU – modified ILU, modifikovaná verze algoritmu ILU,
- ILUTP – ILU with pivoting, varianta ILU(0) s výběrem hlavního prvku.

Pro algoritmy ILU(0), MILU a ILUTP platí, že jsou si velice podobné. Všechny tyto funkce vychází z Gaussovy eliminační metody popsané v sekci 3.6.1. Výsledkem těchto metod je většinou horní trojúhelníková matice U a spodní trojúhelníkovou maticí L, pro které platí:

$$A = LU \tag{4.3}$$

Pro matice L a U platí, že nad (respektive pod) diagonálou mají samé nuly. Toho se v praxi často využívá a obě tyto matice jsou uloženy do jedné. Takto vygenerovaná matice je pro nás vhodná, jelikož praxe ukazuje, že jako preconditioner vykazuje dobré výsledky.

Časová složitost těchto algoritmů je obecně $O\left(n^3\right)$, což vychází z výše zmíněné Gaussovy eliminační metody, u které jsme si dříve řekli, že její časová složitost je právě $O\left(n^3\right)$.

[2]ILU: incomplete LU factorization, neboli rozklad matice na její dolní a horní trojúhelníkovou část.

4.2.1 Jacobiho preconditioner

Jacobiho preconditioner je jeden z nejsnáze vygenerovatelných. Z matice A ho získáme takto:

$$p_{ij} = \begin{cases} a_{ii} & \text{pro } i = j \\ 0 \end{cases} \tag{4.4}$$

Vybereme tedy pouze diagonální prvky matice A, zbylé prvky jsou nulové. Výhodou tohoto postupu, ve srovnání s ostatními zmíněnými metodami, je jeho malá časová náročnost, která je $O(n)$. Nevýhodou pak je, že ve srovnání s ostatními poskytuje jen málo kvalitní výsledky.

4.2.2 Procedury z rodiny ILU

Základem je funkce ILU(0), což je jedna z nejjednodušších variant LU rozkladu. Vychází z Gaussovy eliminační metody a její princip je založen na postupné modifikaci matice A. U tohoto algoritmu se nepracuje s nulovými členy původní matice A. Jako příklad implementace si můžeme uvést variantu popsanou v [5].

```
P = A;
for i = 2:n
  for k = 1:(i-1)
    if A[i,k]!=0
      P[i,k] = P[i,k]/P[k,k];
      for j = (k+1):n
        if A[i,j]!=0
          P[i,j] = P[i,j] - P[i,k]P[k,j];
        endif
      end
    endif
  end
end
```

Vidíme 3 krát vnořený for cyklus, z něhož vyplývá i časová složitost $O(n^3)$.

Varianty MILU a ILUTP si popíšeme jen velmi obecně–veškeré podrobnosti lze dohledat v [5] a [3]. První zmiňovaná funkce MILU se od funkce ILU liší v modifikaci vnitřních cyklů, kdy nepřeskakuje nulové prvky původní matice A, ale dále je zohledňuje.

Varianta ILUTP přidává výběr hlavního prvku, pivotu, do druhého vnořeného cyklu a tak se snaží dále zlepšit vlastnosti generované matice.

4.3 Implementace v Octave

Octave doposud postrádá kvalitní funkce pro generování matic vhodných pro preconditioning. Původně jsem tedy chtěl tyto funkce implementovat nad rámec původního zadání.

Po implementaci funkcí ILU(0) a MILU jsem provedl zátěžové testy a přišel jsem k závěru, že implementace v jazyku Octave není v současnosti rozumnou volbou. Důvodem je pomalost některých operací vykonávaných interpretem Octave. Více se o problému s výkonem rozepisuji v části 6, zde jenom uvedu, že hlavním problémem byla pomalost *for* cyklu, což je v případě jeho trojnásobného vnoření velice citelné. Řešením by bylo implementovat dané funkce v jiném jazyce, například v C++.

V následné diskuzi s vývojáři Octave však vyplynulo, že Octave začne pravděpodobně využívat některé již hotové knihovny právě pro generování preconditioneru. Hovoří se například o knihovně Hypre[3], která je již velmi dobře optimalizována a dosahuje tedy velmi dobrých výsledků.

[3] http://acts.nersc.gov/hypre/

Kapitola 5

Implementace vybraných iteračních algoritmů

Nyní si můžeme blíže popsat vlastní implementaci iteračních metod pro řešení soustav lineárních rovnic, které byly vytvořeny a následně zařazeny do Octave.

Námi popsané a implementované funkce byly vybrány ze seznamu chybějících funkcí, který je dostupný na stránkách projektu[1]. V tomto seznamu jsou nejžádanější funkce vhodné k implementaci. Práci jsme navíc konzultovali a koordinovali s ostatními vývojáři. Toto se dělo prostřednictvím elektronické konference, kterou je možné sledovat na internetové adrese: *http://www.nabble.com/Octave-f1895.html*. Zamezily jsme tím duplicitní práci. Dalším důležitým krokem bylo nalezení vhodné studijní literatury. Zdroje, ze kterých jsme čerpaly, jsou uvedeny v příloze.

Prostá implementace vybraných funkcí je ale pouze částí našeho úkolu. Druhou je jejich začlenění do projektu Octave. Samotný proces začlenění je následující, po napsání funkce se příslušný soubor zašle do e-mailové konference. Zde je podroben kritice hlavních vývojářů a pokud jsou s jeho kvalitou spokojeni, zařadí ho do jádra Octave, v opačném případě navrhnou, co je potřeba změnit. Případné další úpravy se pak zasílají formou rozdílových aktualizací vygenerovaných programem diff. V open-source komunitě obecně platí „Vydávej hodně a vydávej často“. S postupem práce a implementací je pak seznámeno více lidí, kteří přináší vlastní poznatky a nápady.

Zdrojové soubory námi implementovaných funkcí jsou součástí přílohy, popřípadě projektu Octave.

5.1 Způsob implementace

Nové funkce je možné psát dvěma způsoby – buď přímo v jazyce Octave, nebo v některém z jiných podporovaných jazyků, které jsou kompilovány do jazyka cílové platformy. Většina funkcí vyšší úrovně je psána právě prvním způsobem prostřednictvím tzv. M-souborů (anglicky M-File). Daň za nižší výkon oproti kompilovaným funkcím je vykoupena vyšší mírou abstrakce a oproštění se od problémů, které jsou blízké jazykům nižší úrovně. Další výhodou je snadnější údržba a přístup k funkcím, které jsou distribuovány v textové podobě a jsou uloženy v adresářích k tomu určených.

Většina rozsáhlejších projektů se řídí tzv. Coding Standard. Coding Standard je vlastně souhrn rad, doporučení a pravidel říkající, jakým způsobem psát program, aby byl snadno

[1] http://www.gnu.org/software/octave/index.html

udržovatelný a čitelný. Souhrn pravidel zahrnuje v prvé řadě styl odsazování bloků kódu a závorek, konvence v pojmenovávání proměnných, ale i takové drobnosti, jako je styl zápisu řetězců. Tyto standardy je dobré dodržovat nejen pro zvýšení čitelnosti kódu, ale také proto, že jsou v nich často zahrnuty pro daný jazyk či projekt drobné specifikace, které mají za úkol zvýšit výkon, popřípadě potlačit konstrukce, které jsou často zdrojem chyb. Standard použitý v Octave je popsán v [8], kde je možné se s ním blíže seznámit.

Krom výše zmíněného je navíc dobré získat obecný přehled o silných a slabých stránkách jazyka Octave. Základní informace je možné opět najít na stránkách projektu. Bohužel tento seznam není úplný, jak jsme se přesvědčili při pokusech implementovat metody pro preconditioning. Zmiňuje pouze několik málo faktů, z nichž nejdůležitější je snaha omezit realokace paměti na nezbytné minimum.

Pokud například víme, že naše funkce vygeneruje matici o rozměru $n \times n$, je rychlejší vygenerovat nejprve prázdnou matici daných rozměrů a tu pak plnit, než vytvořit matici menší a tu pak zvětšovat postupným přidáváním řádků. Informace, která se však v těchto radách nezmiňuje, a která dle našeho názoru za zmínku stojí, je rozdíl v rychlosti těchto dvou funkcí:

```
function [out] = slow(A)
    out = zeros (1, rows(A));
    for i=1:rows(A)
        out(1,i) = A(i,i)+i*i;
    endfor
endfunction
```

Takováto funkce je řádově pomalejší než:

```
function [out] = quick(A)
    out = diag(A)' + (1:rows(A)).^2 ;
endfunction
```

Obě přitom vykonávají stejnou činnost. Na vstupu očekávají čtvercovou matici a výsledkem je vektor diagonálních prvků této matice, kde ke každému z těchto prvků je navíc přičtena druhá mocnina čísla odpovídající jeho pořadí.

Pro představu jsme provedli test na jednotkové matici o rozměru 10000×10000. Zde se ukázalo, že druhá varianta je více než $80\times$ rychlejší než varianta první. Obě funkce přitom vykonávají stejnou činnost. První využívá konstrukce `for`, druhá o mnoho rychlejší konstrukci popsanou v sekci 2.2.3. Pravdou ale je, že u druhé funkce je o poznání méně zřejmá podstata její činnosti, proto je nutné ji řádně okomentovat.

5.2 Postup práce

Nejprve bylo nutné zvolit konkrétní funkce k implementaci. Toto se dělo v součinnosti s vývojáři a společností Red Hat. Následně jsme začali sbírat podklady pro jednotlivé funkce. Zde se vyskytl první problém, a to ten, že většina zdrojů nabízela algoritmy, které v nějakém ohledu neodpovídaly našim potřebám. Nejčastějším nedostatkem bylo, že tyto algoritmy nepočítaly s možností předpodmínění. Výhody předpodmínění jsme diskutovali v kapitole 4 a dle našeho názoru jsou poměrně zásadní pro rozumnou použitelnost námi psaných funkcí. Jako nejobsáhlejší zdroj dat se nám nakonec osvědčila publikace [5], kterou jsem dále konfrontovali převážně s [6]. Za osvědčený zdroj se také považuje stránka http://mathworld.wolfram.com/.

Po shromáždění potřebných materiálů následoval návrh rozhraní nových funkcí. Zde jsme měli situaci zjednodušenou, jelikož naším přáním bylo dosáhnout kompatibility s Matlabem, proto jsme přejali rozhraní, které má u těchto funkcí právě ten.

Následoval přepis algoritmů do jazyka Octave. Tato část sama o sobě nebyla až tak náročná. Náročnější byly následné optimalizace pro zvýšení výkonu, o tomto se více rozepisuji v kapitole 6.

Závěrečnou fází pak byla validace výsledků, kdy jsme museli prověřit, že výsledky vypočtené našimi funkcemi jsou správné.

5.3 Popis rozhraní implementovaných procedur

Implementované funkce, které jsme do této doby popsali mají stejné rozhraní. Na vstupu očekávají až sedm vstupních parametrů:

- A – čtvercová, symetrická a pozitivně definitní matice reálných čísel o rozměrech $n \times n$.
- b – vektor řešení o délce n.
- tol – specifikace cílové tolerance, implicitně 10^{-6}.
- maxit – specifikace maximálního počtu iterací, implicitně menší z čísel 20 a n.
- M1 – specifikace preconditioneru M, můžeme být použita i funkce, která vrací M\X.
- M2 – v kombinaci s M1 definuje preconditioner tak, že platí $M = M1 \times M2$.
- x0 – specifikace počátečního odhadu řešení, implicitně pak nulový vektor délky n.

První dva parametry jsou povinné. Zbylé parametry jsou volitelné.

Výsledkem funkcí pak je 5 hodnot:

- x – vypočítaný výsledek.
- flag – příznak, který nabývá hodnoty 0, pokud funkce dosáhne řešení v předem daném počtu iterací, 1 pokud řešení nedosáhne nebo 3 pokud funkce začne stagnovat tzn. v dvou následujících krocích se nezměnila hodnota relativního rezidua. V prostředí Matlabu může parametr *flag* nabývat i hodnoty 2, což značí špatný preconditioner, ale jak si popíšeme dále, tuto kontrolu z praktických důvodů neprovádíme.
- relres – obsahuje hodnotu relativního rezidua v posledním kroku, které je spočítáno jako $\frac{norm(b-Ax)}{norm(b)}$.
- iter – je číslo iterace, při kterém funkce skončila.
- resvec – je vektor relativních reziduí pro každou iteraci.

Použití námi vytvořených funkcí vypadá tedy následovně:

```
> A=[5 -1 3;-1 2 -2;3 -2 3];
> b=[7;-1;4];
> [x,flag,relres,iter,resvec] = cgs(A,b)
x =

   1.00000
   1.00000
   1.00000

flag = 0
relres =  1.2529e-14
iter =  3
resvec =

   1.0000e+00
   1.9178e-01
   7.2378e-03
   1.2529e-14
```

Místo funkce *cgs* je možné použít i *bicg* nebo *bicgstab*.

Kapitola 6

Testy

Všechny výše popsané algoritmy byly důkladně testovány. Základním problémem bylo najít vhodné testovací rovnice. Vzhledem k tomu, že pro malé matice je téměř jedno jaký algoritmus je použit, jelikož se ve větší míře neprojeví jejich časová efektivita, bylo nutno použít soustavy o několika stech až tisíci neznámých. Ideálně takové, které se skutečně reálně řeší. Takovéto matice můžeme najít třeba na stránce MatrixMarket[1], kde je na výběr několik desítek matic, včetně popisu jejich vlastností, jako je počet nulových členů a grafy rozložení hodnot.

Do testu jsme nakonec zařadili tyto soustavy:

- bcsstm11[2],
- bcsstm12[3],
- pores_3[4],
- watt_1[5].

Čas potřebný pro výpočet byl měřen na počítači Merlin, jehož aktuální hardwarová konfigurace je uvedena v příloze. Po prvních testech vyšlo najevo, že většina algoritmů má řádově horší výsledky než jejich ekvivalent v Matlabu. Bylo tedy nutné najít zdroj potíží a tento odstranit. Vhodný nástroj, použitelný pro profiling, který je v Octave dostupný, je funkce *cputime()*, která vrací procesorový čas uplynulý od započetí práce s Octave.

6.1 Optimalizace kódu

V našich funkcích jsme oběvili několik slabých míst. Asi nejvýraznějším je funkce *cond()*. Tato funkce vrátí podmíněnost matice. V našich funkcích sloužila pro kontrolu toho, že předávaný preconditioner má dobré charakteristiky a nezhorší řešitelnost soustavy. Při testech na náhodných maticích se ukázalo, že pro klasické matice je tato funkce zhruba dvakrát pomalejší než implementace v matlabu. Pro matice řídké, které jsou velmi často předávány jako preconditioner, je ale v Matlabu použita funkce *condest()*, která je pro tuto skupinu matic optimalizována. Tato funkce však v Octave dosud není k dispozici a použití metody

[1] http://math.nist.gov/MatrixMarket/
[2] http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc1/bcsstm11.html
[3] http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc1/bcsstm12.html
[4] http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/pores/pores_3.html
[5] http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/watt/watt__1.html

cond() se ukázalo být fatální chybou. Pro rozumnou použitelnost tedy bylo nutné kontrolu na podmíněnost preconditioneru odstranit a spoléhat se na to, že uživatel zadá rozumné hodnoty.

Po odstranění volání *cond()* se všechny algoritmy řádově zrychlily, avšak stále byly patrné rozdíly. Při následném profilingu jsme odhalili další důvody zpomalení námi psaných algoritmů, konkrétně pak funkce *inv()* a *det()* z nichž obě jsou zhruba 10krát pomalejší než jejich implementace v Matlabu. Vzhledem k charakteru práce jsme se však rozhodli, že obě funkce ve funkcích ponecháme s patřičným komentářem a je možné, že v další verzi Octave budou optimalizovány.

Posledním značným zklamáním bylo zjištění, že Octave samo o sobě není tak rychlé jako Matlab. Demonstrační cyklus:

```
a = 1
for iter = 1:5000
   a = a*iter
end
```

proběhne v Matlabu v řádu milisekund, zatímco Octave trvá zhruba desetiny sekundy. Při hledání důvodu tohoto rozdílu vyšlo najevo, že Matlab používá techniky JIT[6] kompilace. Tato technika spočívá v tom, že zdrojový skript je při spuštění převeden nikoli do kódu virtuálního stroje, ale přímo do kódu cílové platformy. Tím je možné několikanásobně zrychlit prováděný kód. U cyklů je pak tento rozdíl ještě zřetelnější. Matlab si navíc udržuje cache takto již jednou zkompilovaných funkcí, a proto voláme-li jednu funkci vícekrát, je druhé a další volání mnohonásobně rychlejší.

6.2 Testovací metodika

Z výše zmíněného vyplývá, že za současných podmínek bude drtivá většina operací rychlejší v prostředí Matlabu. Srovnání časů tedy není nejlepší volbou pro srovnání kvality implementace našich funkcí. K tomu se mnohem více hodí počet iterací nutný pro nalezení správného řešení.

Testovací metodika byla následující. Pro každou matici a funkci bylo provedeno celkem 5 měření, a byla vybrána nejmenší hodnota. Na první pohled je zřejmé že rozdíly v časech jsou značné ve prospěch Matlabu. O něco zajímavější je počet iterací, kterých bylo potřeba k dosažení správného výsledku. Je patrné, že zde jsou si výsledky velice blízké. Lze tedy předpokládat, že pokud bude Octave dále optimalizováno, přiblíží se námi implementované funkce rychlostí Matlabu.

Pro měření jsme tentokráte nepoužili funkce *cputime*(), ale dle doporučení na stránkách Matlabu jsme použili funkce *tic*() a *toc*(). Voláním funkce *tic*() zapneme časovač a voláním funkce *toc*() získáme čas, jenž uběhl od posledního volání funkce *tic*().

Jelikož ne všechny testované soustavy nalezené na MatrixMarket obsahovaly i vektor řešení, byl tento vypočten jako $A * ones(rows(A), 1)$, kde A byla testovaná soustava , druhý výraz odpovídá vektoru jedniček o počtu řádků shodném s maticí A. Výsledky pro všechny neznámé by se tedy měly přibližovat 1. Pro zajímavost jsme do měření zahrnuli i prosté dělení. Maximální počet iterací byl nastaven na 1 000.

[6] Just-in-time

6.3 Výsledky měření

Veškeré naměřené údaje shrnují tabulky 6.1 - 6.4. Sloupec *Čas* obsahuje dobu výpočtu příslušného algoritmu ve vteřinách, sloupec *It* pak příslušný krok, během kterého metoda našla řešení. V případě, že metoda nenašla řešení v daném počtu iterací, je zde *NA*.

Rovnou zde upozorníme, že implementované algoritmy mají více variant. Liší se vzájemně pořadím operací a i prostředí Octave se od prostředí Matlabu může lišit, například v přesnosti prováděných výpočtů. Tímto si vysvětlujeme drobné rozdíly v počtu iterací určitých metod v Octave a Matlabu.

Octave bez předpodmínění								
	bcsstm11		**bcsstm12**		**pores_3**		**watt_1**	
Algoritmus	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**
bcg	0.0964	20	0.0778	3	0.6188	NA	0.0386	1
cgs	0.0817	20	0.0644	3	0.4020	NA	0.0347	1
bicgstab	0.0767	16	0.0798	3	0.5863	961	0.0346	1
prosté dělení	0.0183	-	0.0173	-	0.0293	-	0.0392	-

Tabulka 6.1: Octave bez předpodmínění.

Matlab bez předpodmínění								
	bcsstm11		**bcsstm12**		**pores_3**		**watt_1**	
Algoritmus	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**
bcg	0.0182	20	0.0088	3	0.5382	NA	0.0045	1
cgs	0.0186	18	0.0077	3	0.5820	NA	0.0037	1
bicgstab	0.0252	16.5	0.0097	2.5	0.5663	NA	0.0033	1
prosté dělení	0.0004	-	0.0005	-	0.0090	-	0.2467	-

Tabulka 6.2: Matlab bez předpodmínění.

V tabulce 6.2 nás může zaujmout počet iterací u metody *bicgstab*, který ve dvou případech nabývá hodnot 16.5 a 2.5. Toto je pravděpodobně způsobeno tím, že u této funkce v závislosti na konkrétní implementaci můžeme provádět změnu výsledku dvoukrokově s postupným vyhodnocením podmínky, nebo jednokrokově. Matlab pravděpodobně zvolil dvoukrokový postup a výsledek 16.5 značí, že požadované přesnosti bylo dosaženo již v mezikroku.

Asi nejzajímavější jsou matice pores_3, kde se k řešení blížíme jen velmi pomalu. Tato matice tedy není příliš vhodná pro řešení danými metodami. Oproti tomu matice watt_1 se zdá být ideálním kandidátem pro řešení novými funkcemi. Jeden z důvodů je pravděpodobně i ten, že soustava má více správných řešení.

V následujících měřeních shrnujeme výsledky dosažené za pomocí předpodmínění. Jak jsme si řekli dříve, Octave postrádá metody pro preconditining, proto byl preconditioner vypočítán v Matlabu takto:

```
> setup.type = 'nofill';
> P = ilu(A,setup)
```

Octave s předpodmíněním								
	bcsstm11		bcsstm12		pores_3		watt_1	
Algoritmus	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**
bcg	0.1043	35	0.0824	3	3.5353	NA	NA	NA
cgs	0.0306	1	0.0641	1	3.6968	NA	NA	NA
bicgstab	0.0316	1	0.0312	1	1.6126	81	NA	NA

Tabulka 6.3: Octave s předpodmíněním.

Matlab s předpodmíněním								
	bcsstm11		bcsstm12		pores_3		watt_1	
Algoritmus	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**	**Čas**	**It**
bcg	0.0017	1	0.0055	1	1.2158	133	NA	NA
cgs	0.0017	1	0.0045	1	0.7990	121	NA	NA
bicgstab	0.0018	0.5	0.0020	0.5	0.5576	79.5	NA	NA

Tabulka 6.4: Matlab s předpodmíněním.

kde A je řešená matice a následně exportován do Octave.

V tabulkách 6.4 a 6.3 můžeme pozorovat, pozitivní vliv předpodmínění na rychlost konvergence. U většiny soustav bylo řešení nalezeno po menším počtu iterací. V prostředí Matlabu můžeme pozorovat i zkrácení běhu programu. V Octave je situace složitější. S preconditionerem jsou totiž nejprve provedeny operace, které mají usnadnit následnou řešitelnost, jedná se právě o v sekci 6.1 zmíněný výpočet inverze a determinantu. Tyto funkce jsou však naneštěstí v Octave v současné době poměrně pomalé a přestože můžeme sledovat zmenšení počtu iterací, čas nutný pro nalezení řešení se nezkrátil. Na mailing listu však již padla zmínka o probíhajících optimalizacích těchto funkcí a proto jsme se je rozhodli ponechat.

Ve většině měření bylo prosté dělení rychlejší než námi implementované algoritmy. Toto je však možné přisoudit tomu, že dělení je velice častá operace a je tedy již velice dobře optimalizovaná, navíc pravděpodobně napsána v nativním kódu. Navíc, jak již bylo zmíněno výše, ne pro všechny typy soustav jsou iterační metody vhodné. Nám se podařilo najít pouze jednu takovou, kde by bylo použití námi implementovaných funkcí přínosné.

Při srovnání s funkcemi napsanými v Matlabu si, dle našeho názoru, vedeme dobře. Počty iterací jsou vyrovnané a lze tedy předpokládat, že s postupným vylepšením interpretu Octave dosáhneme i srovnatelných časů.

Kapitola 7

Další implementované funkce

Nad rámec naší původní práce jsme navíc implementovali některé další funkce, konkrétně pak skupinu funkcí schopnou snadno vykreslovat orientované grafy a funkci *treelayout* pro generování souřadnic, které mohou sloužit jako podklad pro vykreslení grafu. Procedura *treelayout* je prostou modifikací již hotové funkce *treeplot*, proto jí zde nebudeme věnovat větší pozornost a zaměříme se rovnou na problematiku vykreslení orientovaného grafu.

7.1 Analýza problému

Našim cílem bylo vytvořit funkci schopnou vykreslit co nejširší skupinu orientovaných grafů. Následkem toho musí být schopna přijímat co možná nejvíce vstupních parametrů, z nichž ale povinný bude pouze jediný. Tímto jediným parametrem je přechodová matice popisující hrany mezi jednotlivými vrcholy. Následně by tato funkce měla umožnit specifikovat popisky jednotlivích vrcholů, pozice těchto vrcholů a mnoho dalších.

V současné verzi používá Octave pro vykreslování grafů nástroje GNU Plot. Po nastudování dokumentace[1] k tomuto nástroji vyšlo najevo, že hlavním cílem tohoto nástroje je kresba statistických grafů. Pokud bychom chtěli kreslit graf orientovaný, museli bychom ho vykreslit z jednotlivých grafických primitiv, což by bylo náročné.

Naší snahou bylo najít jiný open-source program, který by více vyhovoval našim potřebám. Všechny naše požadavky naplnila skupina utilit z projektu graphviz[2], pro který jsme se rozhodli napsat v Octave interface, abychom následně mohli využít služeb tohoto programu.

7.1.1 Graphviz

Graphviz se skládá celkem ze čtyř utilit. Pro nás připadají v úvahu dvě:

- neato a
- dot.

Obě jsou schopny vykreslit orientovaný graf, ale mírně se liší. První umožňuje přesně definovat pozici jednotlivých vrcholů a má širší možnosti v kreslení neorientovaných grafů. Druhá jmenovaná vrcholy pozicovat neumí a je více optimalizována pro orientované grafy. Z těchto

[1] http://www.gnuplot.info/documentation.html
[2] http://www.graphviz.org/

dvou jsme nakonec zvolili utilitu *neato*, hlavně z důvodu, že umožňuje přesně specifikovat pozice vrcholů.

7.2 Návrh řešení

Utilita *neato* se ovládá prostřednictvím jazyka *dot*. Našim úkolem tedy v prvé řadě je napsat modul, který bude schopen generovat popis grafu v tomto jazyce, předávat ho na vstup programu *neato* a zobrazovat výsledný graf. Následně pak můžeme napsat funkce, které budou tento modul využívat a budou zobrazovat některé speciální typy grafů. Pro zobrazení grafů jsme se nakonec rozhodli použít utility display z programu imagemagick, jelikož je značně rozšířená a pravděpodobně bude nainstalována na velkém množství unixových počítačů.

7.3 Vlastní implementace

Prvním problémem, který musíme řešit, je způsob předávání parametrů. Těchto parametrů je velmi mnoho, jsou různorodé a jen málokdy povinné. Ideální se nám pro tento účel zdálo použít datového typu *struktura* popsaného v 2.2.4. Tím naplníme všechny naše potřeby a v budoucnu umožníme velice snadno nové parametry přidat. V rámci řešení jsme se navíc rozhodli problém dekomponovat na jednotlivé dílčí úkoly, které jsme dále rozdělili do samostatných funkcí. Těmito problémy jsou:

- vygenerování obecného popisu grafu, vrcholu a hran.
- vygenerování popisu jednotlivích hran.
- vygenerování popisu jednotlivích vrcholů.
- přebrání vygenerovaného kódu v jazyce dot a zobrazení vlastního grafu.

Každý z těchto problémů je řešen v samostatné funkci. Samotné zdrojové soubory jsou součástí přílohy.

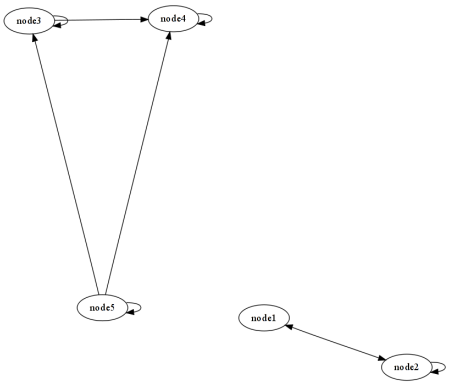
7.4 Porovnání a ukázky grafů

Zde porovnáme současnou funkci *gplot* s námi napsanou funkcí *dotplot*. První výhodou nově implementované funkcí *dotplot* je, že i bez explicitního uvedení souřadnic je schopna vykreslit přehledný orientovaný graf. Pro další příklady využijeme přechodové matice:

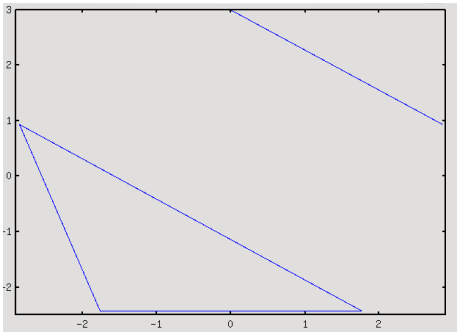
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{7.1}$$

Graf vykreslený metodou neatoplot bez doplňujících parametrů je zobrazen na obrázku 7.1.

Zde vyvstává první nedostatek funkce *gplot*, která potřebuje parametry minimálně dva, přičemž druhým parametrem je pozice jednotlivých vrcholů. Navíc, jak vidíme na obrázku 7.2, takto vykreslený graf nedosahuje zdaleka přehlednosti grafu našeho, nehledě na to, že například hrana z vrcholu 5 do vrcholu 5 není vykreslena vůbec.



Obrázek 7.1: Základní výstup funkce neatoplot



Obrázek 7.2: Základní výstup funkce gplot

Nová funkce *dotplot* navíc přidává mnoho volitelných parametrů. Podrobný popis je možné získat v Octave po vypsání příkazu:

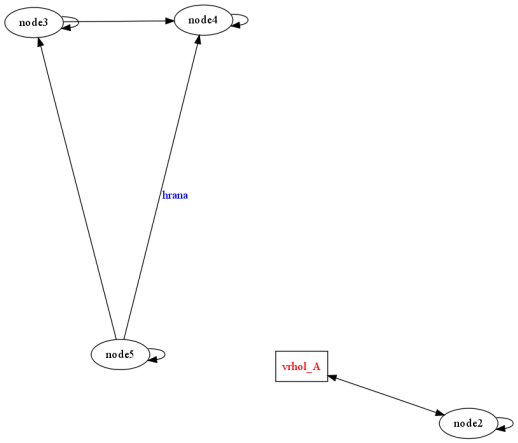
```
>help dotplot
@deftypefn {Function File} {} dotplot (@var{matrix})
@deftypefnx {Function File} {} dotplot (@var{matrix},@var{position},@var{options})
```

Zde uvádíme pouze část výpisu, kompletní popis dané procedůry je součastí jejího zdrojového souboru.

Zde si předvedeme jenom jednoduchou ukázku, čeho je naše funkce schopná po zadání těchto doplňujících parametrů. Předpokládejme opětovné použití přechodové matice 7.1:

```
> options.node{1}.shape = 'box';
> options.node{1}.label = 'vrhol_A';
> options.node{1}.fontcolor = 'red';
> options.edge{5,4}.fontcolor = 'blue';
> options.edge{5,4}.label = 'hrana';
```

Na obrázku 7.3 vidíme změny způsobené pravě dodatečnými agumenty.



Obrázek 7.3: Ukázka formátování

7.5 Podpůrné funkce

Přestože funkce *dotplot* poskytuje velmi dobré výsledky a je schopna mnoho grafů sama naformátovat do přehledné podoby, můžeme se setkat s potřebou zvolit si vlastní rozložení vrcholů. Abychom nemuseli vypisovat pozici vrcholů ručně, implementovali jsme čtyři pomocné funkce:

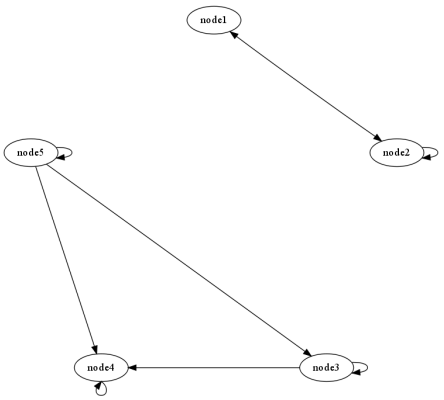
gplotgen: generuje souřadnice pro funkci gplot a neatoplot, a to takovým způsobem, aby vrcholy ležely na kružnici (obrázek 7.4).

gplotgensplit: podobné funkci *gplotgen*; navíc částí grafu, které nejsou spojité, budou ležet na samostatných kružnicích (obrázek 7.5).

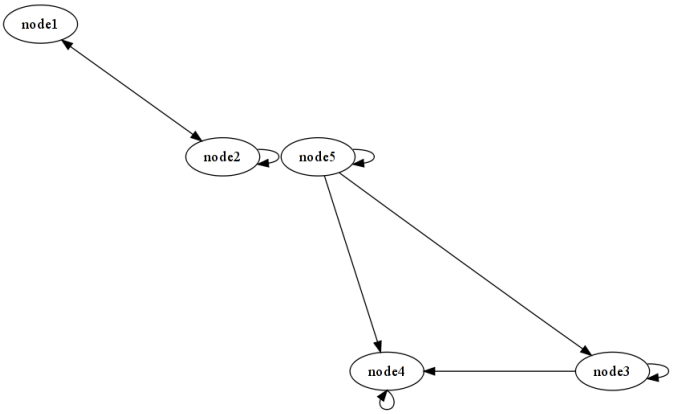
gplotgenline: generuje souřadnice pro funkci gplot a neatoplot, a to takovým způsobem, aby vrcholy ležely na jedné přímce (obrázek 7.6).

gplotgenlinesplit: podobné funkci *gplotgenline*; navíc částí grafu, které nejsou spojité, budou ležet na samostatných přímkách (obrázek 7.7).

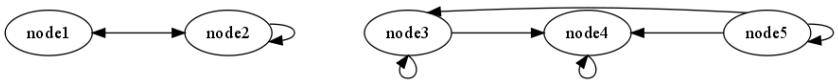
Všechny výše jmenované funkce najdou uplatněné hlavně při vizualizaci orientovaných grafů. Je možné je přitom využit pro funkci *dotplot* i *gplot*.



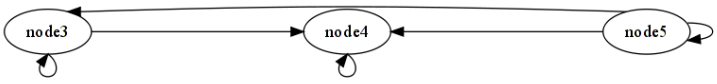
Obrázek 7.4: Ukázka výstupu funkce gplotgen



Obrázek 7.5: Ukázka výstupu funkce gplotgensplit



Obrázek 7.6: Ukázka výstupu funkce gplotgenline



Obrázek 7.7: Ukázka výstupu funkce gplotgenlinesplit

Kapitola 8

Závěr

V úvodu naší práce jsme se seznámili s programem Octave a jeho historií. Naučili jsme se jej obsluhovat a řešit jeho prosřednictvím základní algebraické úlohy. Položili jsme si tak základ k následnému rozšíření programu Octave o nové funkce, které jsou zapsány právě v jazyce Octave.

Následně jsme se seznámili s problematikou řešení soustav linárních rovnic. Popsali jsme si některé postupy a seznámili se s jejich výhodami i nevýhodami, ukázali jsme si, že některé funkce jsou vhodné pro určitou třídu problémů, zatímco pro jinou jsou téměř nepoužitelné s ohledem na časovou složitost.

V další kapitole jsme zužitkovali nově nabyté vědomosti a přikročili jsme k praktické části naší práce. Našim cílem bylo implementovat chybějící funkce, konkrétně pak: *treelayout*, *cgs*, *bicg* a *bicgstab*. Toto se nám podařilo a všechny jmenovanné se staly součástí jádra Octave. Tím je zaručeno, že budou šířeny spolu s programem a dostanou se ke koncovým uživatelům. Napomůžou tak další kompatibilitě a přenositelnosti kódu mezi Octave a Matlabem. V průběhu této práce jsme navíc objevili drobnou chybu ve stávajicím kódu funkce treeplot, na kterou jsme upozornili jejiho autora, který následně provedl opravu.

Přestože jsme se snažili námi implementované funkce optimalizovat jak jen to bylo možné, některé překážky se nám překonat nepodařilo. Asi největším zklamáním pro nás bylo zjištění, že interpret jazyka Octave nedosahuje rychlosti interpretu Matlabu. Jedním z důvodů je již dříve zmíněná absence JIT kompilace. I zde se však situace postupně zlepšuje. Do Octave jsou neustále začleňovány patche ostatních vývojářů z nichž některé jsou zaměřeny právě na zlepšení výkonu. Mnoho práce však ještě zbývá a jakákoli pomoc je ze strany vývojářů Octave vítaná.

Octave má ale i silné stránky, mezi nejvýraznější bezpochyby patří svobodná licence, pod kterou je tento program šířen. Tato licence zaručuje právo na přístup ke zdrojovým kódům Octave, což umožňuje jejich studium a napomáhá dalšímu rozvoji.

Součastí naší práce se nakonec stalo i rozšíření Octave o zcela novou funkčnost, a to generování popisu orientovaných grafů. Toto je nad rámec současné funkčnosti Matlabu, proto nebyly tyto funkce zařazeny přímo do Octave, ale budou zařazeny do podobného projektu Octave Forge. Zde jsou udržovány funkce, jenž nejsou součásti hlavní distribuce Octave, ale jsou s ní kompatibilní, což umožňuje uživatelům jejich snadnou instalaci a použití.

Literatura

[1] Barrett, R.; Berry, M.; Chan, T. F.; aj.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, ISBN 0-89871-328-5.

[2] Farebrother, R.: *Linear Least Squares Computations*. CRC, 1988.

[3] Hysom, D.; Pothen, A.: *Level-based Incomplete LU Factorization: Graph Model and Algorithms*. SIAM Journal On Matrix Analysis and Applications, 2002.

[4] Kovár, M.: *Maticový a tenzorový počet*. Vysoké učení technické v Brně, 2008.

[5] Saad, Y.: *Iterative Methods for Sparse Linear Systems (2nd edition)*. Society for Industrial and Applied Mathematics, 2000, ISBN 0-89871-534-2.

[6] Shewchuk, J. R.: *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. School of Computer Science Carnegie Mellon University, 1994.

[7] Tichý, P.: *O některých otevřených problémech v krylovovských metodách*. Dizertační práce, Matematicko - fyzikální fakulta Univerzity Karlovy, Praha, 2002.

[8] WWW stránky: Octave-Forge: StandardsMFiles [online]. `http://wiki.octave.org/wiki.pl?action=browse&diff=1&id=StandardsMFiles`, [cit. 2009-04-17].

Seznam příloh

Dodatek A – Konfigurace studentského serveru Merlin.

Dodatek B – Obsah CD.

Dodatek C – CD.

Dodatek A

Konfigurace studentského serveru Merlin

OS – CentOS 5.3 64bit

Pamět – 4 GB RAM

CPU – 2xDual Core Opteron 2216

Pevné disky – 2x150 GB + 2x 300GB HDD

Dodatek B

Obsah CD

- Zdrojové soubory
- Zdrojové soubory textové zprávy
- Textová zpráva