UNIVERZITA PALACKÉHO V OLOMOUCI
PŘÍRODOVĚDECKÁ FAKULTA
KATEDRA MATEMATICKÉ ANALÝZY A APLIKACÍ MATEMATIKY

BAKALÁŘSKÁ PRÁCE

Numerické algoritmy pro hledání vlastních čísel a vektorů matice

Vedoucí diplomové práce:
RNDr. Tomáš Fürst, Ph.D.
Rok odevzdání: 2010

Vypracoval:
Daniel Beneš
MAP, III. ročník

Prohlášení

Prohlašuji, že jsem vytvořil tuto diplomovou práci samostatně za vedení pana Tomáše Fürsta a že jsem v seznamu použité literatury uvedl všechny zdroje použité při jejím zpracování.

V Olomouci dne 13.3. 2010

Poděkování

Rád bych na tomto místě poděkoval vedoucímu diplomové práce Tomáši Fürstovi za obětavou spolupráci i za čas, který mi věnoval při konzultacích.

Obsah

1 Úvod

Cílem této práce je představení různých numerických algoritmů pro hledání vlastních čísel a vektorů matice. Problém hledání těchto čísel a vektorů k dané matici (dále Eigensystem[1]) je zajímavou syntézou zcela odlišně stavěných matematických disciplín algebry a numerické matematiky. Aplikace těchto algoritmů lze nalézt v různých odvětvích např. dynamika, kvantová chemie, elektrické sítě, chemické reakce, makroekonomika a přesné příklady použití jsou k nalezení v [6].

V práci analyzuji pouze algoritmy, které jsou obecně vhodné pro matice většího než pátého řádu. Navíc jsem se, přestože v praxi nás často zajímají vlastní čísla v nějakém intervalu nebo třeba největší vlastní číslo a pro tento případ jsou vhodné speciální metody, rozhodl zaměřit na rutiny, které řeší úplný eigensystem, čili všechny vlastní čísla a jim příslušné vektory. O jednotlivých procesech by toho šlo napsat daleko více, např. analýzy chyb, důkazy konvergence, možná vylepšení, ale pro účely této práce jsem se rozhodl na daném prostoru zpracovat a naprogramovat co nejvíce těchto algoritmů.

Po popisu algoritmů má tato práce také za cíl dané algoritmy naprogramovat, k tomuto účelu jsem zvolil použití objektově orientovaného jazyka VB.NET pod dnes velmi rozšířenou platformu .NET (reference např.[1]), díky této volbě bude výstupem práce zároveň zdokumentovaná a přehledná knihovna naprogramovaných algoritmů pro další použití developerů pod .NET. Tímto vznikne menší verze známého balíčku algoritmů ve Fortranu k řešení problémů Eigensystem, takzvaného EISPACKu[2]. EISPACK je vlastně programovým přepisem rutin popsaných v [4] a tato publikace je považována také za základní zdroj informací ohledně Eigensystem algoritmů. O tom, že podobné knihovny programátoři často používají, svědčí například fakt, že se vývojem matematických .NET knihoven zabývají i některé komerční firmy[3].

Součástí této práce je tedy CD, s kódy i již přeloženými algoritmy. Celé ře-

[1] anglický termín
[2] zkratka Eigensystem Pack více [3]
[3] například www.bluebit.gr nebo www.centerspace.net

šení nazvané EigensystemAlg na CD je rozděleno do dvou projektů a výsledkem jsou tedy dvě knihovny tříd. Projekt GenMath[4] je složen ze struktur a tříd, pro obecnou práci s vektory a maticemi nad $\mathbb{R}$ i $\mathbb{C}$ (díky objektově orientovanému přístupu by šlo efektivně naprogramovat tyto funkce nad libovolným tělesem, to by ale přesáhlo cíl této práce), přičemž implementovány jsou pouze základní a pro druhý projekt nezbytné operace a funkce. Projekt EISPACKNET je tedy složen z algoritmů analyzovaných v této práci a je referencován na projekt GenMath, naproti tomu projekt GenMath už žádnou referenci kromě těch, které jsou součástí .NET 3.5, neobsahuje.

Jako příklad použití mnou naprogramovaných knihoven jsem vytvořil webovou stránku volně přístupnou na numalg.aspone.cz , kde lze on-line hledat vlastní čísla matice. Zde jsem naprogramoval i pár doplňkových funkcí jako pro vlastní účely vytvořenou databázi použití algoritmů obsahující například řád matice, čas běhu algoritmu, atd.

2 Numerické algoritmy

Z definice vlastních čísel je zřejmé, že pro matice čtvrtého a menšího řádu lze nalézt přesné algebraické řešení Eigensystem (vlastní čísla: kořeny charakteristického polynomu matice, vlastní vektory: soustavy homogenních lineárních rovnic). Numerické algoritmy, ale postupují zcela odlišně a snaží se použít matematického modelu, který zaručuje konvergenci, ale protože nemohou pokračovat do nekonečna, musí se zastavit po splnění nějakého kritéria. Důležitou věcí v dobrém numerickém algoritmu je tzv. zaokrouhlovací chyba. K té vždy dochází a nelze se jí úplně vyvarovat (nastane prakticky při každé numerické operaci s desetinnými čísly a dál se šíří v celém algoritmu), jde tedy o její minimalizaci. Proto je například možné, ale numericky neefektivní, hledat vlastní čísla jako přibližná numerická řešení charakteristické rovnice. Takový algoritmus by navíc byl velmi pomalý a nikde v literatuře, zabývající se numerickými rutinami, se o něm ani nezmiňují.

[4] zkratka General Math

3 Jacobiho transformace reálné symetrické matice

3.1 Použití

Jacobiho transformace je jedna z nejjednodušších, ale také nejelegantnějších v současné době používaných metod. Je vhodná pro hledání všech vlastních čísel a zároveň vždy produkuje i ortogonální vlastní vektory. Tento algoritmus nelze nijak upravit k hledání pouze některých vlastních čísel a vektorů a pro matice většího řádu (větší než 10) je pomalejší než kombinace Householderovy tridiagonalizace a QR algoritmu, viz kapitola 3.

Samozřejmě jedná se reálnou symetrickou matici, která má vždy všechna vlastní čísla reálná a vlastní vektory ortogonální a algoritmus vždy konverguje bez jakkýchkoliv jiných požadavků na matici.

3.2 Popis algoritmu

Označíme zadanou symetrickou reálnou matici jako A_0 a její řád n. Jacobiho iterační proces lze popsat takto:

$$A_k = U_k^T \cdot A_{k-1} \cdot U_k \quad , k \in \mathbb{N} \tag{1}$$

kde U_k jsou ortogonální matice tvaru:

$$U(p,q,\Phi) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\Phi & \cdots & \sin\Phi & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin\Phi & \cdots & \cos\Phi & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ p \\ \\ q \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & p & & q & & \end{matrix}$$

tedy U_k jsou matice, které se liší od jednotkové matice pouze v prvcích $u_{pp} = u_{qq} = \cos\Phi$ a $u_{pq} = -u_{qp} = \sin\Phi$. Tyto ortogonální transformace se také nazývají Jacobiho rotace. Důležité je, že díky ortogonalitě (a tudíž i regularitě) jsme

provedli vlastně podobnostní transformaci a je triviální dokázat, že podobné[5] matice mají stejné spektrum (jsou maticí stejného lineárního operáturu, pouze v jiné bázi). Přes různé možnosti volby parametrů p, q, Φ, dále se budu zabývat tzv. klasickou Jacobiho metodou tj.: v k-tém kroku algoritmu nalezneme U_k tak, že parametry $p, q (p < q)$ zvolíme jako souřadnice prvku s největší absolutní hodnotou mimo diagonálu a úhel rotace Φ tak aby touto rotací byl anulován prvek (p, q) (a tím i (q, p)) matice A_{k-1} (Podotknu, že v dalším kroku, když se bude anulovat jiný prvek, se nám může předchozí nula zrušit). Takto postupujeme, až se matice A_x redukuje na diagonální tvar na strojovou přesnost (nebo jakoukoliv větší), proč to nastane je patrné z důkazu konvergence, který je naznačen níže.

Z roznásobení matic (1) a označením $A_k = \{a_{ij}^{(k)}\}$ plyne :

$$a_{ip}^{(k)} = a_{pi}^{(k)} = a_{ip}^{(k-1)} \cos\Phi + a_{iq}^{(k-1)} \sin\Phi \tag{2}$$

$$a_{iq}^{(k)} = a_{qi}^{(k)} = -a_{ip}^{(k-1)} \sin\Phi + a_{iq}^{(k-1)} \cos\Phi \tag{3}$$

pro $\forall i \neq p, q$ a pro zbývající čtyři změněné prvky matice A_k :

$$a_{pp}^{(k)} = a_{pp}^{(k-1)} \cos^2\Phi + 2a_{pq}^{(k-1)} \cos\Phi \sin\Phi + a_{qq}^{(k-1)} \sin^2\Phi \tag{4}$$

$$a_{qq}^{(k)} = a_{pp}^{(k-1)} \sin^2\Phi - 2a_{pq}^{(k-1)} \cos\Phi \sin\Phi + a_{qq}^{(k-1)} \cos^2\Phi \tag{5}$$

$$a_{pq}^{(k)} = a_{qp}^{(k)} = (a_{qq}^{(k-1)} - a_{pp}^{(k-1)}) \cos\Phi \sin\Phi + a_{pq}^{(k-1)}(\cos^2\Phi - \sin^2\Phi). \tag{6}$$

Hledáme Φ tak, aby $a_{pq}^{(k)}$ bylo rovno nule. Pokud $a_{pq}^{(k-1)} = a_{qp}^{(k-1)} = 0$, můžeme algoritmus ukončit, protože by to znamenalo, že prvek s největší absolutní hodnotou mimo diagonálu je anulovaný a tím pádem i všechny ostatní prvky mimo diagonálu, čili $A_{k-1} = diag(\lambda_1, \ldots, \lambda_n)$. Budeme tedy předpokládat opak a položíme

$$\tau = \frac{a_{qq}^{(k-1)} - a_{pp}^{(k-1)}}{2a_{pq}^{(k-1)}}$$

[5] A je podobná B, pokud ex. regulární Q tak, že $A = Q^{-1}BQ$

a

$$t = \tan\Phi = \frac{s}{c}.$$

Z řešené rovnice $a_{pq}^{(k)} = 0$ z (6) pak dostáváme, že t musí splňovat

$$t^2 + 2\tau t - 1 = 0.$$

Z této kvadratické rovnice musíme určit ten kořen, který je bližší nule, tedy aby $|\Phi| \leq \frac{\pi}{4}$. Pak už můžeme určit $c = \frac{1}{\sqrt{1+t^2}}$ a $s = tc$. A máme tedy všechny parametry k určení nové matice A_k. K minimalizování zaokrouhlovacích chyb pak použijeme místo (2),(3) ekvivalentní

$$a_{ip}^{(k)} = a_{pi}^{(k)} = a_{ip}^{(k-1)} - \sin\Phi(a_{iq}^{(k-1)} + a_{ip}^{(k-1)}\Gamma) \tag{7}$$

$$a_{iq}^{(k)} = a_{qi}^{(k)} = a_{iq}^{(k-1)} + \sin\Phi(a_{ip}^{(k-1)} - a_{iq}^{(k-1)}\Gamma) \tag{8}$$

kde $\Gamma = \tan\frac{\Phi}{2} = \frac{\sin\Phi}{1+\cos\Phi}$ a místo (4),(5),(6) :

$$a_{pp}^{(k)} = a_{pp}^{(k-1)} - a_{pq}^{(k-1)}\tan\Phi \tag{9}$$

$$a_{qq}^{(k)} = a_{qq}^{(k-1)} + a_{pq}^{(k-1)}\tan\Phi \tag{10}$$

$$a_{pq}^{(k)} = a_{qp}^{(k)} = 0. \tag{11}$$

Rychlost a důkaz konvergence: Pokud nalezneme vyjádření řešení výše zmíněné rovnice $t^2 + 2\tau t - 1 = 0$ pomocí klasické formule $t_{1,2} = -\tau \pm \sqrt{\tau^2 + 1}$ je vidět, že jedno řešení bude v absolutní hodnotě menší jedné. To ale znamená, že pro tento kořen bude vypočtené Φ splňovat výše zmíněné $|\Phi| \leq \frac{\pi}{4}$. Nyní nás bude zajímat vliv Jacobiho rotace na výraz $\sum_{j\neq i} a_{ij}^2$, pokud bychom dokázali, že aplikací Jacobiho rotace na matici A dojde vždy ke snížení hodnoty výrazu, vlastně bychom dokázali konvergenci k diagonální matici (součet čtverců je zjevně

nezáporný). Vzhledem k tomu, že transformací se mění pouze prvky v p,q-tém sloupci a řádku a vzhledem k anulaci a_{pq} můžeme psát

$$\sum_{j\neq i} {a_{ij}^{(k)}}^2 - \sum_{j\neq i} {a_{ij}^{(k+1)}}^2 = 2\sum_{i\neq p,q} ({a_{ip}^{(k)}}^2 + {a_{iq}^{(k)}}^2 - {a_{ip}^{(k+1)}}^2 - {a_{iq}^{(k+1)}}^2) + 2{a_{pq}^{(k)}}^2.$$

Pro každé $i \neq p,q$ platí z rovnic (2),(3) také:

$${a_{ip}^{(k+1)}}^2 + {a_{iq}^{(k+1)}}^2 = {a_{ip}^{(k)}}^2 cos^2(\Phi) + {a_{iq}^{(k)}}^2 sin^2(\Phi) + {a_{ip}^{(k)}}^2 sin^2(\Phi) + {a_{iq}^{(k)}}^2 cos^2(\Phi) = {a_{ip}^{(k)}}^2 + {a_{iq}^{(k)}}^2$$

Celkově tedy pak

$$\sum_{j\neq i} {a_{ij}^{(k)}}^2 - \sum_{j\neq i} {a_{ij}^{(k+1)}}^2 = 2{a_{pq}^{(k)}}^2,$$

odkud vidíme, že skutečně s každou iterací dochází ke zmenšování součtů druhých mocnin prvků mimo diagonálu a taky to zdůvodňuje volbu prvku a_{pq}. Označíme-li $S_k := \sum_{j\neq i} {a_{ij}^{(k)}}^2$, vychází

$$S_{k+1} = S_k - 2{a_{pq}^{(k)}}^2.$$

Uvážíme-li navíc, že prvek a_{pq} byl volen jako ten s největší absolutní hodnotou mimo diagonálu v matici A_k, můžeme odhadnout $(n^2-n){a_{pq}^{(k)}}^2 \geq S_k (\Leftarrow$ každý sčítanec v S_k je menší než ${a_{pq}^{(k)}}^2$, těch sčítanců je celkem n^2-n). Platí tedy

$$S_{k+1} = S_k - 2{a_{pq}^{(k)}}^2 \leq S_k - \frac{2S_k}{n^2-n} = S_k(1-\frac{2}{n^2-n}) \leq \ldots \leq S_0(1-\frac{2}{n^2-n})^{k+1}.$$

Dále pokud označíme $N := \frac{n^2-n}{2}$, platí

$$S_{rN} \leq (1-\frac{1}{N})^{rN} S_0 \leq e^{-r} S_0.$$

Což značí, že každých $(n^2-n)/2$ kroků iterace klesne součet čtverců prvků mimo diagonálu nejméně e-krát. Je to upravdu přísný odhad a podle [7] i [8] je ve skutečnosi konvergence rychlejší a odkazují na důkazy kvadratické konvergence (=pro dostatečně velké $k \quad \exists c : \sqrt{S_{k+N}} \leq cS_k$). Aby byl důkaz úplně kompletní muselo by se ještě dokázat, že je konvergence zaručena k jedné určité

diagonální matici, že takových matic nebude více, to se jeví jako jednoduché, ale důkaz není úplně triviální (využívá důsledků teorie perturbace), k nalezení je v [8] strana 268. Přestože, jsme v tomto důkazu dále nepoužili požadavek $|\Phi| \leq \frac{\pi}{4}$, tento je nezbytný právě k důkazům rychlejší konvergence, navíc pokud se podíváme na vzorce (9),(10) je zřejmé, že pokud bychom zvolili to větší t, mohlo by nám to zbytečně posunout už třeba dobře aproximované vl. číslo. □

Zbývá ještě vyřešit otázku vlastních vektorů, která ale v případě tohoto algoritmu není vůbec problém. Nechť tedy na konci Jacobiho procesu máme matici $A_x = diag(\lambda_1, \ldots, \lambda_n)$, která vznikla z původní A_0 výše popsanou metodou, pak tedy

$$diag(\lambda_1, \ldots, \lambda_n) = {U_x}^T \cdots {U_1}^T \cdot A_0 \cdot U_1 \cdots U_x,$$

a označíme-li $V = U_1 \cdots U_x$ pak

$$D := diag(\lambda_1, \ldots, \lambda_n) = V^T \cdot A_0 \cdot V \tag{12}$$

Pokud si uvědomíme, že V je ortogonální, a když tuto rovnici vynásobíme zleva maticí V, dostaneme $A_0V = VD$, z čehož je vidět, že sloupce matice V jsou právě vlastní vektory příslušné vlastním číslům $\lambda_1, \ldots, \lambda_n$. Při průběžném počítání právě této matice opět využijeme speciálního tvaru matic $U_\alpha : \forall i = 1 \ldots n$

$$v_{ip} := v_{ip} - \sin\Phi(v_{iq} + v_{ip}\Gamma)$$

$$v_{iq} := v_{iq} + \sin\Phi(v_{ip} - v_{iq}\Gamma).$$

4 Householderova tridiagonalizace reálné symetrické matice a QR algoritmus

4.1 Použití

Tato metoda je podle [4] nejefektivnější metoda, přesněji kombinace dvou metod, sloužící k nalezení všech vlastních čísel samotných nebo všech vlastních čísel a jím příslušným vlastním vektorům. Jak již bylo napsáno výše, pro matice vyšších řádů je tato metoda většinou rychlejší než Jacobiho.

4.2 Popis algoritmu

4.2.1 Tridiagonalizace

Proces tzv. tridiagonalizace, čili podobnostní transformace symetrické matice A do tridiagonálního tvaru vychází z toho, že pro každou takovou matici existuje ortogonální matice H tak, že

$$H^T \cdot A \cdot H = T, \tag{13}$$

kde T je tridiagonální matice. Pokud řád matice A je n, pak matici H získáme pomocí $n-2$ householderových transformací dle následujícího schématu.

Uvažujme iterativní proces diagonálních transformací matice $A = A_0$, kde v každém kroku bude matice A_{k-1} tridiagonální v prvních $k-1$ sloupcích a tvaru

$$A_{k-1} = \begin{array}{c} \left(\begin{array}{ccc} B_{11} & B_{12} & 0 \\ B_{12}^T & B_{22} & B_{23}^T \\ 0 & B_{23} & B_{33} \end{array} \right) \begin{array}{c} k-1 \\ 1 \\ n-k \end{array} \\ \begin{array}{ccc} k-1 & 1 & n-k \end{array} \end{array}, \tag{14}$$

kde submatice $\begin{pmatrix} B_{11} & B_{12} \\ B_{12}^T & B_{22} \end{pmatrix}$ je tridiagonální. Pak k její tridiagonalizaci v prvních k sloupcích a transformaci do podoby nové A_k stačí zvolit transformační matici tvaru:

$$P_k = \begin{array}{c} \begin{pmatrix} E_k & 0 \\ 0 & U \end{pmatrix} \begin{array}{c} k \\ n-k \end{array} \\ \begin{array}{cc} k & n-k \end{array} \end{array}, \tag{15}$$

kde U je Housholderova transformační matice (o její specifikaci později, nyní pouze vytknu, že platí $U = U^{-1} = U^T$), pro kterou musí platit :

$$B_{23}^T U = U B_{23} = \begin{pmatrix} l \\ 0 \\ \vdots \\ 0 \end{pmatrix} = l e_1. \tag{16}$$

Platí tedy

$$A_k = P_k A_{k-1} P_k = \begin{pmatrix} E_{k-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & U \end{pmatrix} . \begin{pmatrix} B_{11} & B_{12} & 0 \\ B_{12}^T & B_{22} & B_{23}^T \\ 0 & B_{23} & B_{33} \end{pmatrix} . \begin{pmatrix} E_{k-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & U \end{pmatrix} =$$

$$= \begin{pmatrix} B_{11} & B_{12} & 0 \\ B_{12}^T & B_{22} & B_{23}^T U \\ 0 & UB_{23} & UB_{33}U \end{pmatrix} \tag{17}$$

a evidentně $A_{n-2} = P_{n-2} \cdots P_1 \cdot A_0 \cdot P_1 \cdots P_{n-2} = H^T A H = T$ je tridiagonální matice.

Jak bylo řečeno, matice U řádu $n-k$ je tzv. Householderova matice a ta je definována jako

$$U = E - 2uu^T, \tag{18}$$

kde u je vektor o velikosti 1 a symbolem uu^T rozumíme vnější (tenzorový) součin těchto vektorů[6] (analogicky pak symbolem $u^T u$ rozumíme skalární součin, který je v tomto případě roven 1). Pak z komutativity součinu realných čísel je $U^T = U$. Platí ale také

$$U^2 = E - 4uu^T + 4(uu^T)^2 = E - 4uu^T + 4u(u^T u)u^T = E, \tag{19}$$

tedy výše zmíněné $U^T = U = U^{-1}$. V (18) je k definici matice U použit libovolný jednotkový vektor, to je ale dost omezující. Nechť tedy v je libovolný vektor s $n-k$ prvky, pokud zvolíme $u = \frac{v}{\sqrt{v^T v}} = \frac{v}{|v|}$, kde $|.|$ zde i dále značí Euklidovu normu $||.||_2$, pak lze U také napsat jako

$$U = E - \frac{2vv^T}{v^T v}. \tag{20}$$

Pro náš případ potřebujeme, aby $(E - \frac{2vv^T}{v^T v})B_{23} = le_1$ a tudíž si povšimneme následující vlastnosti všech Householderových transformací, dokázaných v [7], strana 209: pro každý vektor x platí

$$v = x \pm |x|e_1 \quad \Rightarrow \quad Ux = \pm |x| e_1. \tag{21}$$

[6] $\{uu^T\}_{ij} = u_i u_j$

Což vlastně znamená, že jestliže pro výpočet matice U z (20) zvolíme v jako $B_{23} \pm |B_{23}|e_1$, pak UB_{23} bude v žádané formě násobku e_1. Otázkou zůstává volba znaménka. Podle stejného zdroje dojde k minimalizaci numerické chyby pokud vypočteme vektor v jako

$$v_i = x_i \qquad \forall i = 2 \dots p \tag{22}$$

$$v_1 = \frac{-(x_2^2 + \dots + x_p^2)}{x_1 + |x|}, \tag{23}$$

samozřejmě u nás $x := B_{23}$.

V algoritmu musíme ošetřit případ, kdy dostaneme v některém kroku vektor B_{23} již v požadovaném tvaru a tento krok pak musíme přeskočit (kdybychom počítali transformační matici, dokonce by to nebyla jednotková matice). Teď ještě zbývá aplikace transformací $P_k A_{k-1} P_k$ na jednotlivé matice, kde s výhodou využijeme symetrie. Z rovnice (17) vidíme, že jediné, co nám zbývá vypočítat, je submatice $UB_{33}U$ (Připomenu, že $UB_{23} = |B_{23}|e_1$ a zbytek zůstává stejný). Označíme-li

$$\beta = \frac{2}{v^T v}, \qquad y = \beta B_{33} v, \qquad w = y - (\beta y^T v/2)v,$$

pak

$$UB_{33}U = B_{33} - vw^T - wv^T. \tag{24}$$

4.2.2 Symetrické tridiagonalizované matice

Teď již máme připravenou tridiagonální matici T a existuje nyní více způsobů, jak nalézt vlastní čísla a vektory. Pokud například hledáme pouze vlastní čísla v nějakém intervalu, je nejlepší využít vlastnosti, že hlavní minory polynomiální matice $T - \lambda E$ uspořádané podle řádu tvoří Sturmovu posloupnost[7] příslušnou determinantu matice $T - \lambda E$, a metody bisekce[8], dokázané například v [8]. Pro účely této práce jsem se ale rozhodl tento algoritmus vynechat.

[7] Pouze tehdy, když každý prvek v tridiagonálním pásu mimo diagonálu je nenulový, v opačném případě rozdělíme matici podle počtu nulových prvků na několik submatic a každou řešíme zvlášť (více v [8] strana 299).

[8] Výhodně upravené Givensem pro nalezení všech vl. čísel v intervalu

4.2.3 QR algoritmus

V této kapitole se budeme zabývat pouze upraveným QR algoritmem pro symetrické tridiagonální matice a ne obecným QR algoritmem. Nyní použijeme podobný algoritmus jako metoda Jacobiho, a sérií ortogonálních transformací dojdeme k diagonální matici. K tomu budeme potřebovat tzv. Givensovy rotace. Givensovou rotací rozumíme čtercovou matici řádu 2 tvaru $\begin{pmatrix} c & -s \\ s & c \end{pmatrix}$, kde $s = \sin\Phi, c = \cos\Phi$ takovou, že pokud máme dána dvě realná čísla a a b, platí

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

Tuto rotaci je v praxi nejlepší vypočítat podle [9] tak, že pokud $|b| > |a|$ tak :

$$s = \frac{1}{\sqrt{1 + (\frac{-a}{b})^2}}, \qquad c = s\frac{-a}{b}, \tag{25}$$

v opačném případě

$$c = \frac{1}{\sqrt{1 + (\frac{-b}{a})^2}}, \qquad s = c\frac{-b}{a}. \tag{26}$$

Nyní uvažujme iterační proces:

$$S_k = B_k \cdot S_{k-1},$$

kde $k = 1, \ldots, n-1, \quad S_0 = T$ a

$$B_k = \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & & \vdots & & \vdots \\ 0 & \cdots & c & -s & \cdots & 0 \\ 0 & \cdots & s & c & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ k \\ k+1 \\ \\ \\ \end{matrix}, \qquad \begin{matrix} \\ \\ k & k+1 \\ \\ \\ \\ \end{matrix}$$

s použitím c a s v každém kroku pomocí výše popsaného subalgoritmu (25) nebo (26) pro $a = s_{k,k}^{(k-1)}, \quad b = s_{k+1,k}^{(k-1)}$, pokud značíme $S_k = \{s_{ij}^k\}$. Evidentně je matice B_k vždy ortogonální. Tímto vznikne takzvaný QR rozklad matice

$$T = QR = {B_{n-1}}^T \cdots {B_1}^T \cdot S_{n-1} = (B_1 \cdots B_{n-1})^T \cdot S_{n-1},$$

kde Q je ortogonální a R je horní pásová matice s šírkou pásma 3, protože v každém kromě posledního kroku vlastně provadíme pouze násobení

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} . \begin{pmatrix} a & x & 0 \\ b & x & x \end{pmatrix} = \begin{pmatrix} x & x & x \\ 0 & x & x \end{pmatrix}$$

a v posledním kroku

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} . \begin{pmatrix} a & x \\ b & x \end{pmatrix} = \begin{pmatrix} x & x \\ 0 & x \end{pmatrix}.$$

Nyní již víme jak setrojit k libovolné tridiagonální matici její QR rozklad a toho dále využijeme, protože pokud označíme $A_0 := T$ platí:

$$A_s = Q_s R_s \quad \Rightarrow \quad A_{s+1} = Q_s^T A_s Q_s = Q_s^T Q_s R_s Q_s = R_s Q_s. \tag{27}$$

Pokud budeme uvažovat takovouto iteraci od $s = 0$, lze dokázat[9], že matice A_s konverguje k diagonální matici. Konvergenci lze ale ještě urychlit pomocí takzvaných shiftů[10].

Shifty: Lze velmi snadno dokázat:

$$((\omega \in \mathbb{R}) \wedge (T - \omega E = QR)) \Rightarrow T_+ = RQ + \omega E = Q^T T Q \tag{28}$$

a T_+ je také tridiagonální. Shift je tedy právě to, že místo QR rozkladu matice A_s uděláme v každém kroku rozklad matice $A_s - \omega E$ a další matici A_{s+1} vypočteme jako $RQ + \omega E$ a podle předchozího tedy opět provádíme pouze ortogonální transformaci. Toho tedy využijeme k urychlení konvergence.

[9] například pro obecné matice v [8]

[10] možno přeložit jako posunutí, ale já se budu dále držet anglického termínu s českým skloňováním.

Pokud nalezne ω jako vlastní číslo matice

$$T(n-1:n, n-1:n) = \begin{pmatrix} t_{n-1,n-1} & t_{n-1,n} \\ t_{n,n-1} & t_{n,n} \end{pmatrix}, \tag{29}$$

to které je blíže $t_{n,n}$, tedy

$$\omega = t_{n,n} + \frac{t_{n-1,n-1} - t_{n,n}}{2} - sign(t_{n-1,n-1} - t_{n,n}) \sqrt{(\frac{t_{n-1,n-1} - t_{n,n}}{2})^2 + t_{n,n-1}^2} \tag{30}$$

pak po iteraci provedené s tímto shiftem se rapidně zmenší absolutní hodnota prvku $t_{n,n-1} = t_{n-1,n}$ (více o volbě shifu v kapitole 6.2.2) a po několika takových iteracích bude tato hodnota zanedbatelná. Pak také $t_{n,n}$ je již přibližné vlastní číslo. Potom můžeme tedy v dalších iteracích poslední řádek (a sloupec) vynechat a tudíž dělat QR rozklad matice řádu $n-1$:

$$T(1:n-1, 1:n-1) - \omega E = QR$$

a poté počítat shift ω jako vlastní číslo matice

$$T(n-2:n-1, n-2:n-1)$$

, to blíže $t_{n-1,n-1}$ a tak pokračovat dokud nevynecháme všechny kromě prvního řádku (sloupce). A tedy budeme mít již všechny vlastní čísla zadané matice. Zbývá pouze otázka vlastních vektorů, které se budou věnovat v následující podkapitole.

Numerická aplikace Givensovy rotace Pro samotný numerický algoritmus je ještě důležité využít struktury matic B_k, protože počítat Q jako součin matic B_k^T je z praktického a numerického hlediska nemožné (vyzkoušeno, zpomaluje celý algoritmus asi 60 krát). Snadno lze vidět, že je-li L libovolná matice, tak aby měl uvažovaný součin smysl, pak aplikací $L^\dagger := LB_k^T$ se změní pouze k-tý a $(k+1)$-tý sloupec matice L a to následovně:

$$L^\dagger(:, [k, k+1]) = L(:, [k, k+1]) \begin{pmatrix} c & -s \\ s & c \end{pmatrix}^T \tag{31}$$

Toho tedy využijeme tak, že při každém QR rozkladu nebudeme vůbec matici Q počítat, ale budeme hodnoty c, s ukládat do nějakého pole. Pak s použitím (31) můžeme rovnou najít matici $RQ = R \cdot {B_1}^T \cdots {B_{n-1}}^T$ postupnou iterací a neztratíme tím informaci o tom, jak matice Q vznikla. Podobně toho využijeme i pro počítání vlastních vektorů, viz (3.2.4). V [7] je dokonce navrhnuto, jak lze místo obou hodnot c, s ukládat pouze jednu a tu pak zpětně převést, dochází tam ale k malé chybě při zpětném počítání s, která je podle mě při dnešní paměťové vybavenosti počítačů zbytečná.

Implicitní QR: Určitou modifikací, lze provádět QR iterace s shiftem pomocí tzv. metody implicitní QR. Jde vlastně o ekvivalentní úpravu výše zmíněného algoritmu vycházejícího z (28) s tím, že se explicitně v každém kroku neformuluje matice $T - \omega E$. Takto upravený proces by při přesném počítání měl úplně stejný výsledek, k výraznému rozdílu podle [4] dochází v zaokrouhlovací chybě, která bude u implicitního QR menší a implicitní QR je i daleko rychlejší. V této práci jsem se rozhodl tento algoritmus vynechat, více o něm lze najít v [4],[7],[8] nebo se inspirovat kapitolou 4.2.2, kde popisuji implicitní QR algoritmus pro obecné reálné matice.

4.2.4 Vlastní vektory

Protože v celém algoritmu šlo pouze o ortogonální transformace dané matice, lze opět podobně jako u Jacobiho metody (viz. rovnost (12) a následující odstavec) najít vlastní vektory jako sloupce matice průběžně počítané jako součin matic ortogonálních transformací. Toto jsem sám nejprve považoval pro tuto práci za postačující způsob výpočtu vlastních vektorů, jenže opravdu dochází ke zbytečnému prodloužení času běhu programu (rozdíl v přesnosti oběma způsoby vypočtených vlastních vektorů byl irelevantní, ale rozdíl délky běhu programu na matici 80x80 asi půl sekundy, což je asi třetina běhu celého algoritmu).

V první části algoritmu čili tridiagonalizaci, provedeme výpočet ortogonální matice transformací až po samotné tridiagonalizaci, při krocích tridiagonalizace si budeme do nějaké matice ukládat vektory $v^{(j)}$ (pro ušetření paměti lze ukládat

$v^{(j)}$ bez prvního prvku, který je vždy roven 1) jednotlivých Householderových transformací a koeficienty $\beta^{(j)}$ do nějakého pole (vektoru). Vzhledem k tomu, že každá Householderova transformace byla do prvních j řádků a sloupců identitou, je pro výpočet celkové matice V nejefektnější použít zpětnou iteraci a v každém z $n-2$ kroků vypočítat (j je iterační prom.):

$$y = \beta^{(j)} \cdot V(j:n, j:n)^T \cdot v^{(j)} \tag{32}$$

$$V(j:n, j:n) = V(j:n, j:n) - v^{(j)} y^T, \tag{33}$$

kde je zároveň použit vzorec[11] pro zrychlený výpočet $P \cdot V(j:n, j:n)$, kde P je Householderova matice.

V druhé části, čili QR algoritmu, použijeme stejného prostředku jako pro výpočet matice RQ dle (2.3.3) – Numerická aplikace Givensovy rotace. Novou matici $V = V \cdot {B_1}^T \cdots {B_{n-1}}^T$ ortogonálních transformací vypočteme vždy až na konci QR rozkladu postupnou iterací s využitím (31).

5 Redukce reálné matice na Hessenbergův tvar a QR algoritmus

5.1 Použití

Tento algoritmus je dosti podobný tomu, kterým se zabývá kapitola třetí. Jsou zde ale důležité rozdíly, které jsou převážně způsobeny tím, že u nesymetrické reálné matice nemáme zaručenou existenci kompletního systému vlastních vektorů, a samozřejně vlastní čísla a vektory takové matice jsou obecně komplexní. Neexistence úplného systému vlastních vektorů (= defektnost matice) není v samotném algoritmu nijak ošetřena, protože s nutnou existencí zaokrouhlovací chyby neexistuje podle [4] žádná přesná rutina, která by jednoznačně určila, zda je matice defektivní nebo ne. Proto se tento algoritmus tímto nezabývá, pokusí se najít všechny vlastní vektory a je na uživateli, jak výsledek posoudí, protože v případě

[11] [7] strana 211

defektnosti matice bodou nějaké vektory (příslušné stejnému vl. číslu) přibližně rovnoběžné. Navíc u nesymetrických matic lze pozorovat obrovské změny vlastních čísel při malé změně dané matice.

5.2 Popis algoritmu

5.2.1 Redukce na Hessenbergův tvar

Dále v této kapitole, bude matice jejíž eigensystem řešíme označena jako

$$A = \{a_{ij}\}, \quad i, j = 1 \dots n$$

O matici $H = \{h_{ij}\}, \quad i, j = 1 \dots n$ řekneme, že je v Hessenbergově tvaru, pokud platí

$$h_{ij} = 0 \qquad \forall i, j = 1 \dots n : i > j + 1$$

O matici v Hessenbergově tvaru řekneme, že je redukovaná pokud existuje $i \in \mathbb{N}, i \leq n : h_{i+1,i} = 0$

Prostudováním kapitoly 3.2.1, lze snadno vidět, že například právě aplikací $n - 2$ Householderových transformací na reálnou nesymetrickou matici vznikne Hessenbergova matice (právě nesymetrií matice nám nad diagonálou vznikne obecně více pásů matice než v případě symetrickém). Existuje ale rychlejší metoda, která je odvozena od Gaussovy eliminační metody.

Pro další použití si zadefinuji řádkovou permutační matici $E(p, q), \quad \forall p, q = 1 \dots n, i \neq j$:

$$E(p,q) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ p \\ \\ q \\ \\ \\ \end{matrix} ,$$
$$\begin{matrix} & & p & & q & & \end{matrix}$$

což je matice, která se liší od jednotkové matice pouze v prvcích $E_{pp}(p, q) = E_{qq}(p, q) = 0$ a $E_{pq}(p, q) = E_{qp}(p, q) = 1$. Evidentně $E(p, q)$ je regulární a

$E^{-1}(p,q) = E(p,q)$, dále pak násobení libovolné čtvercové matice maticí $E(p,q)$ zleva způsobuje pouze výměnu p-tého a q-tého řádku dané matice a násobení zprava pak výměnu p-tého a q-tého sloupce.

Uvažujme, že v r-tém kroku iteračního procesu byla matice $A_1 := A$ redukována na matici A_r, která je v Hessenbergově tvaru ve svých prvních $r-1$ sloupcích. Označme $A_k = \{a_{ij}^{(k)}\}, \quad i,j = 1 \dots n, \quad k = 1 \dots n-2$ (Někde ještě pro lepší přehlednost odděluji indexy čárkou). V tomto kroku provedeme následující[12]:

1. Najdeme nějvětší z $|a_{ir}^{(r)}|, \quad i = r+1 \dots n$, pokud je takových více, vezmeme první z nich. Pokud toto maximum je nula, tento (r-tý) krok vynecháme a pokračujeme dalším. Pokud toto maximum je větší jak nula, označíme řádkový index tohoto prvku v matici A_r jako $(r+1)^\dagger$. Následně zaměníme v této matici řádky $r+1$ a $(r+1)^\dagger$ a sloupce $r+1$ a $(r+1)^\dagger$.

2. $\forall i = r+2, \dots, n$:

   - Spočítáme $n_{i,r+1} := \frac{a_{ir}^{(r)}}{a_{r+1,r}^{(r)}} \qquad \Rightarrow |n_{i,r+1}| \leq 1.$
   - Od i-tého řádku matice A_r odečteme $n_{i,r+1}$ násobek řádku $r+1$-tého.
   - K $r+1$-tému sloupci matice A_r přičteme $n_{i,r+1}$ násobek sloupce i-tého.

Tedy, s uvážením co v každém výše zmíněném kroku provádíme v jazyku maticových multiplikací, platí :

$$A_{r+1} = N_{r+1}^{-1} \cdot E(r+1,(r+1)^\dagger) \cdot A_r \cdot E(r+1,(r+1)^\dagger) \cdot N_{r+1}, \qquad (34)$$

kde matice N_{r+1} je tvořena prvky:

$(N_{r+1})_{i,r+1} = n_{i,r+1} \quad \forall i = r+2, \dots, n$
$(N_{r+1})_{i,j} = \delta_{ij} \qquad$ všude jinde

Z rovnosti (34) plyne, že tentokrát neprovádíme ortogonální transformace, nicméně pořád jde alespoň o podobnostní transformace, takže spektrum matic bude stejné.

[12] [4],[5]

5.2.2 QR algoritmus pro reálné matice v Hessenbergově tvaru

V této kapitole bude vstupní Hessenbergova matice označena $H = \{h_{ij}\}_{i,j=1\ldots n}$. Dále taky horní index H u libovolné matice bude značit Hermitovskou transpozici této matice.

QR algoritmus pro obecné reálné matice (v Hessenbergově tvaru) je podobný tomu pro matice symetrické (v tridiagonálním tvaru) popsanému v kapitole 3.2.3. Můžeme tedy přejít rovnou k modelu s shifty, vycházíme opět z (28), s tím rozdílem, že máme matici v Hessenbergově tvaru a tudíž lze dokázat, že i matice T_+ bude v tomto maticovém tvaru ([8] strana 524). Opět vyvstává otázka volby shiftu ω, algoritmus by konvergoval i s volbou $\omega := 0$, ale nejrychleji bude konvergovat, pokud budeme volit ω jako odhad vlastní hodnoty. Z kapitoly 3.2.3 plyne, že iterační proces přes s s shifty lze zapsat takto :

$$H_s - \omega_s E = U_s R_s \quad \Rightarrow \quad H_{s+1} = R_s U_s + \omega_s E, \tag{35}$$

kde U je ortogonální (v kapitole 3.2.3 jako Q) a R horní trojúhelníková matice.

Jak ale víme, vlastní číslo reálné matice je obecně komplexní, a volit shift jako komplexní číslo by vedlo k tomu, že bychom měli v dalším kroku matici komplexní, což by dost celou věc zkomplikovalo. Teoreticky a s přesným počítáním (což je v praxi nemožné) by to šlo obejít tím, že bychom místo jednoho kroku s shiftem ω provedli hned dva[13], a to postupně s ω a $\overline{\omega}$ (tedy jeden krok navíc s shiftem komplexně sdruženého k tomu předchozímu), ale v praxi by pak v matici byly prvky se zanedbatelnou, ale nenulovou imaginární částí (a jejich zanedbání by podle [4] zapříčinilo vznik několika chyb).

V tomto odstavci ukážu, proč by to tak mělo teoreticky fungovat. V kapitole 3.2.3 jsme pro výpočet shiftu použili to vlastní číslo submatice $G := H(n-1 : n, n-1 : n)$, které bylo blíže $h_{n,n}$. Protože ale matice G není obecně symetrická, tyto vlastní čísla jsou buď jako předtím dvě reálná čísla (pak je všechno v pořádku, nebudu dále uvažovat) nebo jsou to dvě komplexně sdružená komplexní čísla,

[13] Někde se proto tomu říká double-shift, dvojitý shift

označme je a_1, a_2. Pak výše zmíněný double shift vypadá takto :

$$\begin{aligned} H_k - a_1 E &= U_k R_k \\ H_{k+1} &= R_k U_k + a_1 E \\ H_{k+1} - a_2 E &= U_{k+1} R_{k+1} \\ H_{k+2} &= R_{k+1} U_{k+1} + a_2 E. \end{aligned}$$

Z těchto rovnic lze lehce ukázat, že

$$(U_k U_{k+1})(R_{k+1} R_k) = (H_k - a_1 E)(H_k - a_2 E) = H^2 - sH + tE \tag{36}$$

kde $s := a_1 + a_2 \in \mathbb{R}$ a $t := a_1 a_2 \in \mathbb{R}$, tudíž (s ohledem na to, že součin dvou ortogonálních matic je matice ortogonální a součin dvou horních trojúhelníkových matic je opět horní trojúhelníková. matice) je $(U_k U_{k+1})(R_{k+1} R_k)$ QR faktorizace realné matice a můžeme volit U_k, U_{k+1} tak aby $U := U_k U_{k+1}$ byla ortogonální reálná. Z předchozího tedy plyne:

$$\begin{aligned} H_{k+2} = U_{k+1}^H H_{k+1} U_{k+1} = U_{k+1}^H (U_k^H H_k U_k) U_{k+1} = \\ = (U_k U_{k+1})^H H_k (U_k U_{k+1}) = U^T H_k U, \end{aligned} \tag{37}$$

označením $R := R_{k+1} R_k, M := (H_k - a_1 E)(H_k - a_2 E)$

$$R = U^T M \tag{38}$$

Z výpočtu kvadratické charakteristické rovnice matice G, a známých vzorců je zřejmé, že:

$$s = h_{n,n} + h_{n-1,n-1}$$

$$t = h_{n-1,n-1} h_{n,n} - h_{n,n-1} h_{n-1,n}$$

V [4], s odkazem na práci J. C. F. Francise o QR algoritmech, je navržena metoda provedení výše zmíněného dvojkroku, která se efektně vyhne komplexní aritmetice a je založena na následujícím[14]: pokud jsou matice L a S ortogonální, takové že obě $S^T A S$ a $L^T A L$ jsou v Hessenbergově tvaru, pak pokud matice L a

[14] tzv. věta o implicitním QR pro Hessenbergovy matice, v plném znění i s důkazem v [7] strana 346, platí pouze pro neredukované matice

S mají stejný první sloupec, tak L a S jsou „v podstatě“[15] stejné. Zde zopakuji (37):

$$H_{k+2} = U^T H_k U. \tag{39}$$

Nechť nyní je matice C odvozena nějakou metodou z H_k tak, že

$$U^{\dagger T} H_k U = C, \tag{40}$$

kde $U^\dagger$ je ortogonální a C je v Hessenbergově tvaru, pak pokud $U^\dagger$ má stejný první sloupec jako U, tak $U = U^\dagger$ a $H_{k+2} = C$.

Nyní z (38) plyne, že U je matice, která triangularizuje matici (maticový součin) M, což, jak jsme ukázali výše, je realná matice. Za použití Householderových transformací lze triangularizovat jakoukoliv reálnou matici (viz kapitola 3.2.1 z ohledem na nesymetrii a snahu matici triangularizovat, ne tridiagonalizovat. Navíc, zde nejde o podobnostní transformaci. Kroků bude v tomto případě $n-1$): $P_{n-1} \cdots P_1 = U$. Vzhledem k tomu, že matice $P_2, \ldots, P_{n-1}$ jsou určitě nejméně v prvním sloupci identity, je první sloupec U stejný jako ten P_1. Proto nám stačí najít matici $U^\dagger$ splňující rovnici (40), která bude mít jako svůj první sloupec první sloupec P_1, a tím pádem bychom získali matici $U = U^\dagger$, čili ortogonální matici transformace double-shiftu z H_k na H_{k+2}, a zároveň i matici $C = H_{k+2}$.

Vraťme se nyní k triangularizaci matice. Matice první Householderovy transformace P_1 je určena pouze prvním sloupce matice M, který z (36) je ve tvaru $(x, y, z, 0, \ldots, 0)^T$, kde

$$\begin{aligned} x &= h_{11}^2 + h_{21}h_{12} - sh_{11} + t, \\ y &= h_{21}h_{11} + h_{22}h_{21} - sh_{21} = h_{21}(h_{11} + h_{22} - s), \\ z &= h_{21}h_{32}, \end{aligned}$$

kde pro jednoduchost vynechávám horní iterační indexy u prvků matice H_k. Protože tento sloupec má pouze tři nenulové prvky, vektor u z (18) (potažmo vektor v), čili vektor Householderovy transformace pro výpočet matice P_1 bude mít také pouze tři nenulové prvky a to první tři[16]. To ale z tvaru Hoseholderovy matice musí znamenat, že

[15] $\exists D = diag(\pm 1, \ldots, \pm 1) : S^T AS = D^{-1} L^T ALD$

[16] $u = \frac{v}{|v|}, v = (x \pm \sqrt{x^2 + y^2 + x^2}, y, z, 0, \ldots, 0)^T$

matice P_1 bude všude, kromě prvních tří sloupců maticí jednotkovou. Pak

$$P_1 H_k P_1^T = \begin{pmatrix} \heartsuit & \heartsuit & \heartsuit & \heartsuit & \heartsuit & \dots & \heartsuit \\ \heartsuit & \heartsuit & \heartsuit & \heartsuit & \heartsuit & \dots & \heartsuit \\ \diamondsuit & \heartsuit & \heartsuit & \heartsuit & \heartsuit & \dots & \heartsuit \\ \diamondsuit & \diamondsuit & \heartsuit & \heartsuit & \heartsuit & \dots & \heartsuit \\ 0 & 0 & 0 & \heartsuit & \heartsuit & \dots & \heartsuit \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 0 & \heartsuit & \heartsuit \end{pmatrix} \tag{41}$$

tedy je to matice v Hessenbergově tvaru se třemi prvky navíc (=$\diamondsuit$). Připomenu, že ostatní matice triangularizace matice M vůbec nebudeme hledat, protože nám neovlivní první sloupec jejich součinu. A protože víme jaký bude první sloupec a víme o jednoznačnosti je daleko výhodnější než najít matici U jako tu co triangularizuje M, najít $U^\dagger$ z (40), protože pak už budeme mít přímo novou matici $C = H_{k+2}$.

Nyní je tedy úkolem matic $P_2 \dots P_{n-1}$ ortogonálně transformovat tuto matici (=$P_1 H_k P_1^T$) na matici v Hessenbergově tvaru (a neovlivnit první sloupec součinu, navíc transformace musí být ortogonální, takže použití algoritmu z 5.2.1 nepřipadá v úvahu). Toho se dosáhne tak, že každá z těchto matic posune ty tři prvky navíc o jeden sloupec a řádek níže (ilustrační matice vynechám, přehledně pro $n = 6$ jsou vidět v [7], strana 357). Tedy použijeme $n - 2$ Householderových transformací

$$P_{n-1} \dots P_2 P_1 H_k P_1^T P_2^T \dots P_{n-1}^T = C, \tag{42}$$

podle předchozího tedy $U^T = P_{n-1} \dots P_2 P_1$, $H_{k+2} = C$.

Teď ještě musíme zjistit jak budou vypadat matice $P_{n-1} \dots P_2$. Tyto matice P_r budou Householderovy, příslušné vektorům v_r, nebudu zde zbytečně odvozovat proč, ale

$$v_r = \begin{matrix} (0, \dots, 0, & \alpha_r, \beta_r, \gamma_r, & 0, \dots, 0) \\ r-1 & & n-r-2 \end{matrix}, \tag{43}$$

kde tyto tři nenulové prvky budeme počítat již známou metodou podle (22), (23), (24) pro x jako vektor tvořený postupně prvky na $(r, r-1), (r+1, r-1), (r+2, r-$

1)-tém místě současné matice (= právě dvě ze tří nově vzniklých (posunutých) nenulových prvků pod subdiagonálou a prvku na subdiagonále nad nimi)[17] .

Teda už máme kostru QR dvojkroku, celé to shrnu:

1. Vypočteme s,t.
2. Vypočteme první sloupec matice M.
3. Zjistíme P_1 a provedeme ortogonální transformaci na H_k.
4. $\forall k = 2, \ldots, n-1$ nalezneme P_k a provedeme ortogonální transformaci.
5. Zde už máme novou Hessenbergovu matici H_{k+2} a zároveň matici U.

Pokud budeme takto postupovat v celém algoritmu iteračně přes k, po několika dvojkrocích by měl[18] buď $h_{n,n-1}$ nebo $h_{n-1,n-2}$ být zanedbatelný. Pokud to bude $h_{n,n-1}$ tak na $h_{n,n}$ máme přibližné reálné vlastní číslo matice a můžeme vynechat poslední řádek a sloupec a pokračovat dál. Pokud to bude $h_{n-1,n-2}$, tak máme dvě vlastní čísla (buď komplexně sdružená nebo reálná) jako vlastní čísla matice G a vynecháme dva řádky a sloupce.

Vzhledem k tomu, že jsme v algoritmu předpokládali neredukovanou matici, musíme po každém dvojkroku zjistit, jestli se nám matice nezredukovala (= nestala redukovanou na zadanou přesnost). Pokud ano, vzhledem k tvaru matice se nám matice rozdělí na dvě neredukované, jejichž vlastní čísla jsou stejná jako ta matice celé redukované.

Poslední věcí k algoritmickém ošetření je, že po 30 (dle [4]) dvojkrocích nenalezneme žádné vlastní číslo a pak musíme algoritmus ukončit s neúspěchem, jinak by nejspíš byl nekonečný.

5.2.3 Vlastní vektory

Tento problém je nečekaně trochu složitější než pro LR algoritmus, a proto zde užiji výsledky odvozené v příslušné kapitole: doporučuji před přečtením této

[17] V [4] je ještě doporučeno tento vektor normovat vydělením součtem absolutních hodnot jeho prvků.

[18] Podle [4] je divergence docela vzácná událost.

kapitoly přečíst kapitolu 6.2.3.

V kapitole 6.2.3 jsem ukázal (ukáži), jak k horní trojúhelníkové matici najít vlastní vektory a jak je posléze transformovat na vlastní vektory původní matice. Problém ale je, že v QR algoritmu narozdíl od komplexního LR, na konci běhu algoritmu máme obecně na diagonále 2x2 bloky. Přestože v mém hlavním zdroji informací, co se týče toho, které algoritmy použít - [4], je tento problém řešen složitější zpětnou susbstitucí, já se rozhodl si sám odvodit jednodušší řešení, které se přímo nabízí (a tudíž nejspíš nebude numericky příliš efektivní). Jde o to, že použijeme ještě další podobnostní transformaci matice T, která vznikla na konci QR algoritmu, transformační matici označíme G. Cílem této transformace bude triangularizace matice $T = P^{-1}S^{-1}ASP$ na $T^{\dagger} = G^{-1}P^{-1}S^{-1}ASPG$ a pak postup dle kapitoly 6.2.3. Zřejmě triangularizace nedosáhneme obecně bez přesunutí do komplexní aritmetiky.

Matice G vznikne součinem $G = G_1 \cdots G_l$, l záleží na dané matici a bude patrné z vysvětlení. Účelem transformace G_1 bude evidentně triangularizace prvního 2x2 bloku na diagonále, atd. Tedy chceme aby:

$$G_1^{-1} \cdot \begin{pmatrix} a & b \\ d & e \end{pmatrix} G_1 = \begin{pmatrix} \lambda_p & f \\ 0 & \lambda_{p+1} \end{pmatrix}, \tag{44}$$

přičemž zde a dále nerozlišuji mezi maticí G_i jako 2x2 a nxn, která je všude jinde kromě určitého diagonálního 2x2 bloku jednotková.

Analogicky k Jacobiho metodě s odlišností týkající se nesymetrie a možnosti komplexních vl. čísel, matici G_i budeme hledat ve tvaru:

$$G_i = \begin{pmatrix} \cos z & \sin z \\ -\sin z & \cos z \end{pmatrix}, \tag{45}$$

kde z bude hledaný komplexní nebo reálný parametr. V obou případech platí $\cos^2 z + \sin^2 z = 1, \operatorname{tg} z = \frac{\sin z}{\cos z}$, odkud rovnou $\cos z = \pm\frac{1}{\sqrt{1+\operatorname{tg}^2 z}}$. Z toho také v obou případech plyne, že $G^{-1} = G^T$ (To znamená ortogonalitu, ale pouze v

reálném případě). Roznásobením (44) s použitím (43) dostaváme rovnici

$$(a-e)\cos z \sin z + d\cos^2 z - b\sin^2 z = 0, \tag{46}$$

vydělením $\cos^2 z$

$$(a-e)\,\mathrm{tg}\, z + d - b\,\mathrm{tg}^2 z = 0. \tag{47}$$

Tuto rovnici vyřešíme a vezmeme ten kořen, který nám po transformaci neprohodí vlastní čísla.

Musíme ale samozřejmě při podobnostní transformaci matice T využít její tvar i tvar matice G_i, jinak by to celý algoritmus výrazně zpomalilo. Transformace G_i tedy s ohledem na oba tvary, s předpokladem, že G_i má triagularizovat diagonální blok na pozicích $l, l+1$ průběžně počítané matice $T^{G_{i-1}} = \{t_{jk}^{(i-1)}\}_{j,k=1\ldots n}$, roznásobením provádí pouze následující změny:

$$\begin{array}{ll}
t_{ll}^{(i)} = \lambda_l & \\
t_{l+1,l+1}^{(i)} = \lambda_{l+1} & \\
t_{l,l+1}^{(i)} = cs(t_{ll}^{(i-1)} - t_{l+1,l+1}^{(i-1)}) - s^2 t_{l+1,l}^{(i-1)} + c^2 t_{l,l+1}^{(i-1)} & \\
t_{l+1,l}^{(i)} = 0 & \\
t_{j,l}^{(i)} = ct_{j,l}^{(i-1)} - st_{j,l+1}^{(i-1)} & \forall j = 1\ldots l-1 \\
t_{j,l+1}^{(i)} = st_{j,l}^{(i-1)} + ct_{j,l+1}^{(i-1)} & \forall j = 1\ldots l-1 \\
t_{l,j}^{(i)} = ct_{l,j}^{(i-1)} - st_{l+1,j}^{(i-1)} & \forall j = l+2\ldots n \\
t_{l+1,j}^{(i)} = st_{l,j}^{(i-1)} + ct_{l+1,j}^{(i-1)} & \forall j = l+2\ldots n
\end{array}, \tag{48}$$

kde c, s je zkrácený zápis pro příslušný $\cos z, \sin z$. Analogicky lze také odvodit, jak efektivně průběžně počítat samotnou matici G

Další věc, kterou jsem ještě nezmínil je, jak počítat matici P, to ale není příliš složité a je to analogické ke kapitole 3.2.4. O matici S viz kapitola 6.2.3.

6 Komplexní hermitovské matice

Hledáme vlastní čísla a vektory komplexní hermitovské matice $C = A + Bi$, kde matice A, B jsou obě reálné řádu n, platí : $C^H = C \Rightarrow A^T = A, \quad B^T = -B$. Pokud

$$Cx = \lambda x,$$

pak

$$x^H C x = \lambda x^H x,$$

dále také

$$(x^H C x)^H = x^H C^H x^{HH} = x^H C x,$$

a protože $x^H x \in \mathbb{R}$ musí být $\lambda \in \mathbb{R}$, protože hermitovská transpozice skaláru je rovna právě tomu skaláru, tehdy a jen tehdy, pokud má nulovou imaginární část. Tím jsem dokázal, že vlastní čísla komplexní hermitovské matice jsou reálná.

Uvažujme nyní matici $C^\dagger$ definovanou jako:

$$C^\dagger := \begin{pmatrix} A & -B \\ B & A \end{pmatrix}, \tag{49}$$

což je reálná symetrická matice a její eigensystem můžeme nalézt jedním ze dvou algoritmů popsaných v kapitolách 2 a 3. Jak to souvisí s vlastním systémem původní matice ukazuje následující. Nechť $(u, v)^T$ je vlastní vektor matice $C^\dagger$ příslušný λ_k, u i v mají n prvků, tedy

$$\begin{pmatrix} A & -B \\ B & A \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix} = \lambda_k \begin{pmatrix} u \\ v \end{pmatrix}. \tag{50}$$

Z toho ihned plyne

$$(A + Bi)(u + iv) = \lambda_k (u + iv). \tag{51}$$

Což nám ukazuje, jak postupovat při řešení eigensystému komplexní hermitovské matice převedením na řešení reálného symetrického problému matice dvojnásobného řádu, protože matice $C^\dagger$, bude mít každé vlastní číslo matice C dvakrát.

7 Obecné komplexní matice

7.1 Možné přístupy

Pohledem na (50) a (51) je vidět, že zde nebyl použit předpoklad na hermitovskost matice a tudíž lze opět řešit eigensystem převodem na reálnou matici

dvojnásobného řádu, tentokrát nesymetrickou, a použitím algoritmu popsaného v kapitole 4.

Zároveň lze použít něco podobného jako je QR alg. z kapitoly 4, ale pro přehlednost trochu předělaného na tzv. LR.

7.2 Redukce na Hessenbergův tvar a LR

7.2.1 Redukce na Hessenbergův tvar

Tuto podobnostní transformaci lze provést úplně stejně jako pro reálné matice, viz 5.2.1, jediným rozdílem bude použití komplexní aritmetiky, a že při hledání největšího prvku nebudeme hledat prvek s nějvětší normou, ale s největším součtem abolutních hodnot obou složek[19] (reálné a imaginární).

7.2.2 LR algoritmus

Jednoduchý LR: Jednoduchý (bez shiftů) LR lze popsat následujícím iteračním procesem přes s:

$$A_s = L_s R_s \qquad A_{s+1} = R_s L_s, \tag{52}$$

kde A_0 je daná matice v Hessenbergově tvaru, L_s je ortogonální($L^H L = E$) dolní trojúhelníková a R_s je horní trojúhelníková. V [8] strany 487-492 je dokázána konvergence A_s k horní trojúhelníkové matici, za jistých podmínek. Lze velice snadno dokázat, že matice A_0 a A_s jsou si podobné a matice A_s jsou v Hessenbergově tvaru pro každé s, tedy vlastní čísla matice A_0 budou přibližně čísla na diagonále konvergenční matice. Tento algoritmus tvoří teoretický základ aplikovaného LR alg., je však nutno provést několik změn, protože podmínky konvergence jsou takto těžko splnitelné, konvergence pomalá a algoritmus numericky nestabilní.

Zlepšení stability: V kapitole 4.2.1 bylo ukázáno jak pomocí numericky stabilních transformací nalézt k dané matici podobnou matici v Hessenbergově tvaru.

[19] podle [4] to je numericky dostačující a evidentně snadněji spočitatelné

Podobný algoritmus použijeme i zde, s tím rozdílem, že zde se jedná o triangularizaci, a nejedná se o podobnostní transformaci. V podstatě už nepůjde o LR rozklad matice A_s, protože kvůli stabilitě budeme zaměňovat řádky (v podstatě provedeme LR rozklad, ale matice s permutovanými prvky). Opět by zde bylo nutné nově provést důkaz konvergence takto upraveného LR a dalo by se ukázat, že některé podmínky jsou stejné, některé nové a některé předchozí nemusí být nutně splněny. To je ale daň za numerickou stabilitu, která je v podobném algoritmu nutná. Invariance Hessenbergova tvaru zůstává modifikovaným LR zachována. Triangularizace matice A_s bude dosažena $n-1$ kroky, analogicky ke kapitole 4.2.1:

$$M_{n,n-1}E'_{n-1,n}\dots M_{3,2}E'_{2,3}M_{2,1}E'_{1,2}A_s = R_s, \tag{53}$$

kde $M_{i,j}$ je jednotková matice kromě prvku $M_{i,j} = m_{ij}$, $E'_{i,j}$ je buď $E(i,j)$ dle kap. 4.2.1 nebo jednotková matice E. Toto plyne z tvaru matice A_s : Při výběru pivota vybíráme pouze ze dvou nenulových prvků, prvku na diagonále a prvku pod ním. A $m_{i+1,i}$ bude právě podíl těchto dvou prvků (s normou menší jak $\sqrt{2}$, protože nebudeme zdlouhavě porovnávat normy, ale součet absolutních hodnot obou složek).

$$A_{s+1} = R_s E'_{1,2}M^{-1}_{2,1}E'_{2,3}M^{-1}_{3,2}\dots E'_{n-1,n}M^{-1}_{n,n-1} \tag{54}$$

Snadno lze vidět, že inverzní matice k $M_{i+1,i}$ je s $M_{i+1,i}$ stejná všude kromě $(i+1,i)$-té pozice, kde se liší pouze znaménkem.

Zrychlení konvergence: Konvergenci ještě zrychlíme zavedením shiftů. Iterační proces s shifty plus tzv. obnovováním (vysvětlím později) lze zapsat

$$A_s - \omega_s E = L_s R_s \qquad A_{s+1} = R_s L_s + \omega_s E. \tag{55}$$

Je triviální ukázat, že jde o podobnostní transformaci. K naprogramování je ale vhodňejší použít iteraci dle následujícího

$$A_s - \omega_s E = L_s R_s \qquad A_{s+1} = R_s L_s, \tag{56}$$

čili jde o to samé, s tím rozdílem, že $\omega_s E$ k A_{s+1} nepřidáváme, a pak je A_{s+1} podobná k $A_0 - \sum_{i=0}^{s} \omega_s E$, neboli vlastní čísla matice A_{s+1} se liší od vlastních čísel matice A_0 o $\sum_{i=0}^{s} \omega_s$. Takto upravený proces se nazývá bez obnovování.

Zbývá otázka volby ω_s. Zde, analogicky jako v QR algoritmech, se snažíme zvolit shift jako aproximaci vlastního čísla, v předchozích kapitolách jsem tomu nevěnoval pozornost, ale zde alespoň načrtnu lépe proč a jak.

Jak již bylo řečeno, konvergence algoritmu je dokázána, a tedy $A_s = \{a_{ij}^{(s)}\}_{i,j=1\ldots n}$ konverguje k horní troj. matici pro $s \to \infty$. Označme λ_i prvky na hlavní diagonále konvergenční matice. V samotném důkazu je ještě pro obecné matice dokázáno

$$a_{ij}^{(s)} = O(\frac{\lambda_i}{\lambda_j})^s \qquad s \to \infty \quad (i > j), \tag{57}$$

neboli, že v případě Hessenbergovy matice konvergují prvky na pozicích $(r+1, r)$ k nule stejně nebo rychleji než $(\frac{\lambda_{r+1}}{\lambda_r})^s$. Uvažujeme nyní matici $(A - pE)$. Tato matice má vlastní čísla $(\lambda_i - p)$ a prvek $a_{n,n-1}^{(s)}$ konverguje k nule jako $(\lambda_n - p)/(\lambda_{n-1} - p)^s$. Odkud už je snadno vidět motivace volby shiftu jako aproximaci toho vlastního čísla matice, které by nám konvergenčně vykrystalizovalo na pozici (n, n).

V praxi je nejlepší jako ω_s volit vlastní číslo matice:

$$\begin{pmatrix} a_{n-1,n-1}^{(s)} & a_{n-1,n}^{(s)} \\ a_{n,n-1}^{(s)} & a_{n,n}^{(s)} \end{pmatrix},$$

to blíže $a_{n,n}^{(s)}$.

Celkový proces: V celkovém procesu ještě musíme kontrolovat několik věcí. Zaprvé je to problém, pokud se nám matice stane redukovanou. Pak dojde k rozdvojení problému na dvě menší matice, obě v Hessenbergově tvaru. Pro kontrolu redukovanosti ale neexistuje úplně uspokojivé kritérium, podle [4] je nejvhodnější považovat prvek $a_{i+1,i}^{(s)}$ na subdiagonále za zanedbatelný pokud

$$m(a_{i+1,i}^{(s)}) \leq \epsilon[m(a_{i+1,i+1}^{(s)}) + m(a_{i,i}^{(s)})],$$

kde

$$m(x) = |Re(x)| + |Im(x)|,$$

a ϵ je strojová nebo menší daná přesnost. Pokud se nám tedy stane zanedbatelným prvek $a^{(s)}_{n,n-1}$, máme na $a^{(s)}_{n,n}$ přibližnou hodnotu vlastního čísla a v algoritmu můžeme pokračovat bez posledního sloupce a řádku, pokud se nám ale stane zanedbatelným jiný prvek na subdiagonále, musíme matici rozdělit na dvě. Za druhé je to počet iterací, takže pokud bez nalezení vlastního čísla uděláme třicet iterací, musíme algoritmus ukončit neúspěchem.

7.2.3 Vlastní vektory

V předchozích dvou kapitolách jsem ukázal jak najít k matici A regulární matice S, P takové (přibližně) ,že

$$P^{-1}S^{-1}ASP = T, \tag{58}$$

kde T je matice horní trojúhelníková. Ve vysvětlování jednotlivých subalgoritmů budu postupovat zpětně.

Nyní tedy máme matice P, S a samozřejmně i vlastní čísla matice $T = \{t_{i,j}\}_{i,j}$, která jsou shodná s vlastními čísly matice A. Prvním subalgoritmem bude nalezení vlastních vektorů $u_i = (u_{i,1}, \ldots, u_{i,n})^T$ matice T příslušné vlastním číslům λ_i. Tyto vektory, aby byly vlastní, musí splňovat $\forall i$:

$$\begin{pmatrix} \lambda_1 & t_{1,2} & \ldots & \ldots & \ldots & \ldots & t_{1,n} \\ 0 & \lambda_2 & t_{2,3} & \ldots & \ldots & \ldots & t_{2,n} \\ 0 & 0 & \ddots & t_{**} & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & \lambda_i & t_{i,i+1} & \ldots & t_{i,n} \\ 0 & \ldots & \ldots & 0 & \ddots & t_{**} & \ldots \\ 0 & \ldots & \ldots & \ldots & 0 & \ddots & t_{**} \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 & \lambda_n \end{pmatrix} \cdot \begin{pmatrix} u_{i,1} \\ u_{i,2} \\ \vdots \\ u_{i,i} \\ \vdots \\ \vdots \\ u_{i,n} \end{pmatrix} = \lambda_i \begin{pmatrix} u_{i,1} \\ u_{i,2} \\ \vdots \\ u_{i,i} \\ \vdots \\ \vdots \\ u_{i,n} \end{pmatrix} \tag{59}$$

Jde tedy o homogenní lineární soustavu rovnic s maticí:

$$\begin{pmatrix} \lambda_1 - \lambda_i & t_{1,2} & \dots & \dots & \dots & \dots & t_{1,n} \\ 0 & \lambda_2 - \lambda_i & t_{2,3} & \dots & \dots & \dots & t_{2,n} \\ 0 & 0 & \ddots & t_{**} & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & t_{i,i+1} & \dots & t_{i,n} \\ 0 & \dots & \dots & 0 & \ddots & t_{**} & \dots \\ 0 & \dots & \dots & \dots & 0 & \ddots & t_{**} \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n - \lambda_i \end{pmatrix} \tag{60}$$

Tato matice má hodnost $n-1$ (předpokládáme různá vlastní čísla, pokud jsou některá stejná, obecně se nelze na LR algoritmus moc spolehnout, ale zde můžeme místo nulového prvku na diagonále počítat s epsilon násobkem normy matice, protože přesně takové chyby, už jsme se v algoritmu stejně dopustili) a zvolením jednoho parametru $u_{i,i} = 1$ ihned dostáváme:

$$\begin{array}{ll} u_{i,k} = -\sum_{j=k+1}^{n}(t_{kj}u_{ij})/(\lambda_k - \lambda_i) & \forall k = 1 \dots i-1 \\ u_{i,i} = 1 & \\ u_{i,k} = 0 & \forall k = i+1 \dots n \end{array} \tag{61}$$

Další předpokládá, že jsme si v průběhu samotného LR algoritmu počítali matici P. Připomenu, tato matice vznikne jako součin matic P_s

$$P_s = E'_{1,2}M^{-1}_{2,1}E'_{2,3}M^{-1}_{3,2} \dots E'_{n-1,n}M^{-1}_{n,n-1}, \tag{62}$$

to znamená, že v každém kroku budeme matici P (začneme s jednotkovou) násobit zprava maticí P_s, a tedy v každém LR kroku budeme nejprve zaměňovat sloupce (pokud nutno) a pak budeme odečítat násobky sloupců.

Teď ještě budeme muset matici P vynásobit maticí S zleva. To znamená, že P násobíme maticí

$$S = E_{2,2^\dagger}N_2E_{3,3^\dagger}N_3 \dots E_{n-2,n-2^\dagger}N_{n-2}E_{n-1,n-1^\dagger}N_{n-1} \tag{63}$$

zleva, takže výpočet provedeme zpětnou iterací od $n-1$ do 2, v každém kroku budeme nejprve přičítat násobky řádků a poté zaměňovat řádky.

Poslední, co tedy uděláme pro nalezení přibližných vlastních vektorů matice A, bude zřejmé z následujícího:

$$(P^{-1}S^{-1}ASP = T) \wedge (Tx = \lambda x) \Rightarrow A(SPx) = SP(Tx) = \lambda(SPx), \qquad (64)$$

tedy je vypočteme jako SPu_i.

7.3 Převod na reálný problém

Zde je ale nutná pozornost, protože uvažovaná matice $C^\dagger$ bude mít s každým reálným vlastním číslem matice C toto vlastní číslo dvakrát a s každým komplexním vlastním číslem toto číslo a k němu sdružené. Problém pak nastává v určení, které z komplexně sdružených čísel je vlastní číslo matice C, a které pouze matice $C^\dagger$. Tento problém lze vyřešit pouze, pokud jsme počítali vlastní vektory. Protože pak nám u jednoho z komplexně sdruženého páru vyjde vlastní vektor jemu příslušný $u + iv$ jako přibližně nulový (pozor u i v jsou obecně komplexní vektory), a tím jsme zjistili, že hledané vlastní číslo původní matice je to druhé z páru. V praxi se mi pak nulovosti na stejnou přesnost s jakou byl prováděn algoritmus nepovedlo dosáhnout (většinou tak o dva řády - epsilon jako $e-15$ ale prvky vektoru $e-13+ie-13$), takže tam hledám ten vektor s menší normou (bez zbytečného odmocňování).

Nutnost počítání vlastních vektorů i pokud nejsou požadovány úměrně zpomaluje algoritmus a je tedy zřejmé, že je v tomto případě lepší použít přístup z předchozí kapitoly.

8 Poznámky ke kódu

8.1 Platforma .net

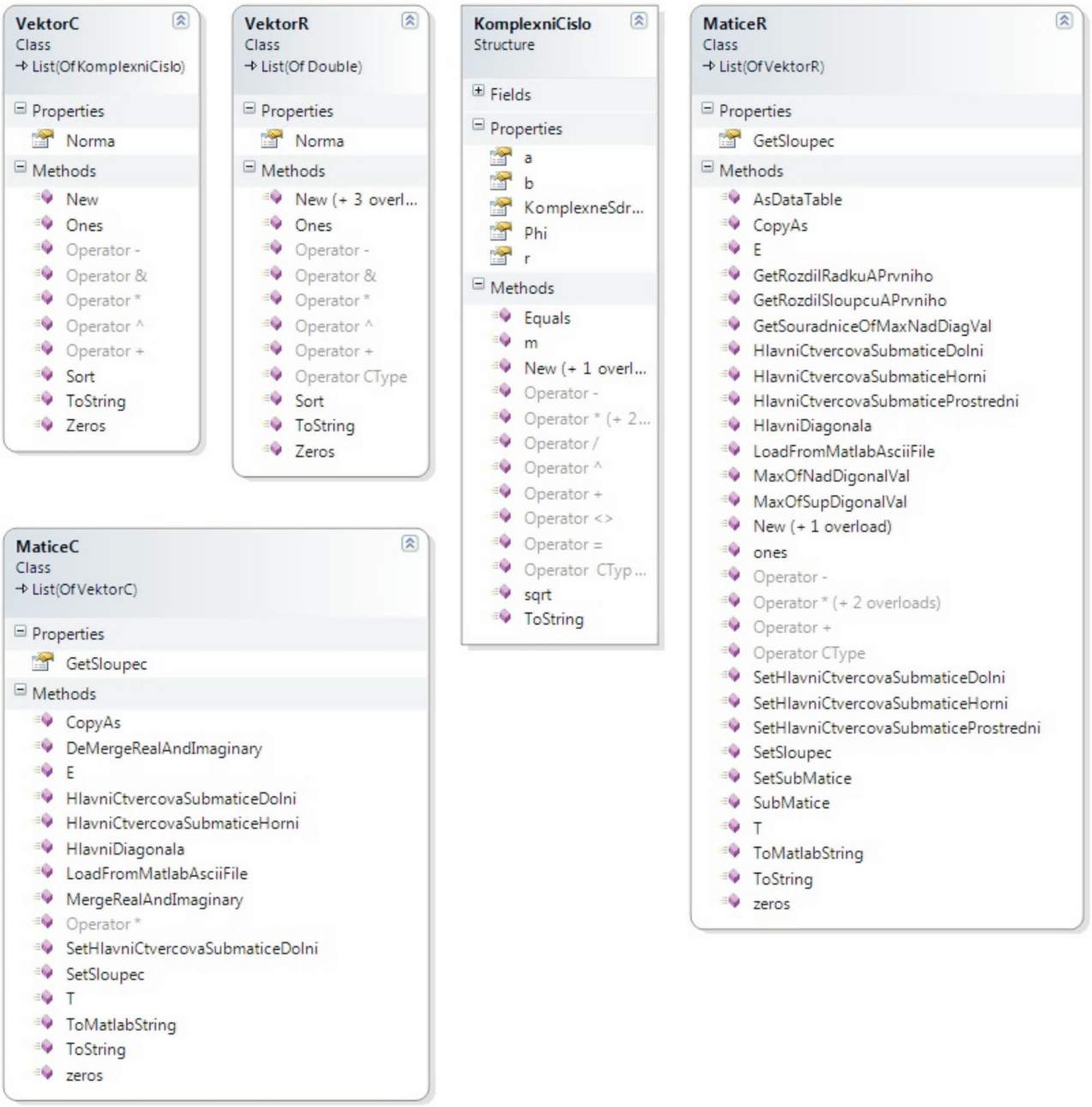
Platforma .net je programové prostředí pro vývoj programů, knihoven kódu, databázových aplikací i webových stránek pro Windows, vytvářené Microsoftem. V celém konceptu jde o soustředění na jeden operační systém, tedy Windows, a na více programovacích jazyků, které jsou pod touto platformou kompatibilní.

Tedy, přestože jsem já programoval ve VB.net, tak mé knihovny může používat C# developer stejně, jako kdyby byly taky naprogramovány v tomto jazyce.

Podle mě nejdůležitějším prvkem celé platformy je její přísně objektově orientované zaměření. Objektové programování je nejnovějším způsobem programování, stojící na třech pilířech, které si nedovolím překládat: encapsulation, inheritance, polymorphism. Nebudu to zde příliš rozebírat, to už by opravdu přesáhlo účel práce, ale shrnu to a zjednoduším. Jde o prvky programovacího jazyku, které vám dovolují vytvářet objekty, jejich instance, přes jejich instance nebo přes samotné objekty pak použijeme jejich členy, tedy zapouzdřené rutiny. Dále pak nám to dovoluje používat znovu náš kód, protože pokud vytvoříme jiný objekt, který ale derivuje své vlastnosti z již vytvořeného, nebudeme muset vše programovat znovu, ale vytvoříme dítě původního objektu. A pomocí posledního pilíře můžeme zacházet s objekty, které mají něco společného, podle té společné věci, nehledě na to o který objekt se jedná.

8.2 GenMath

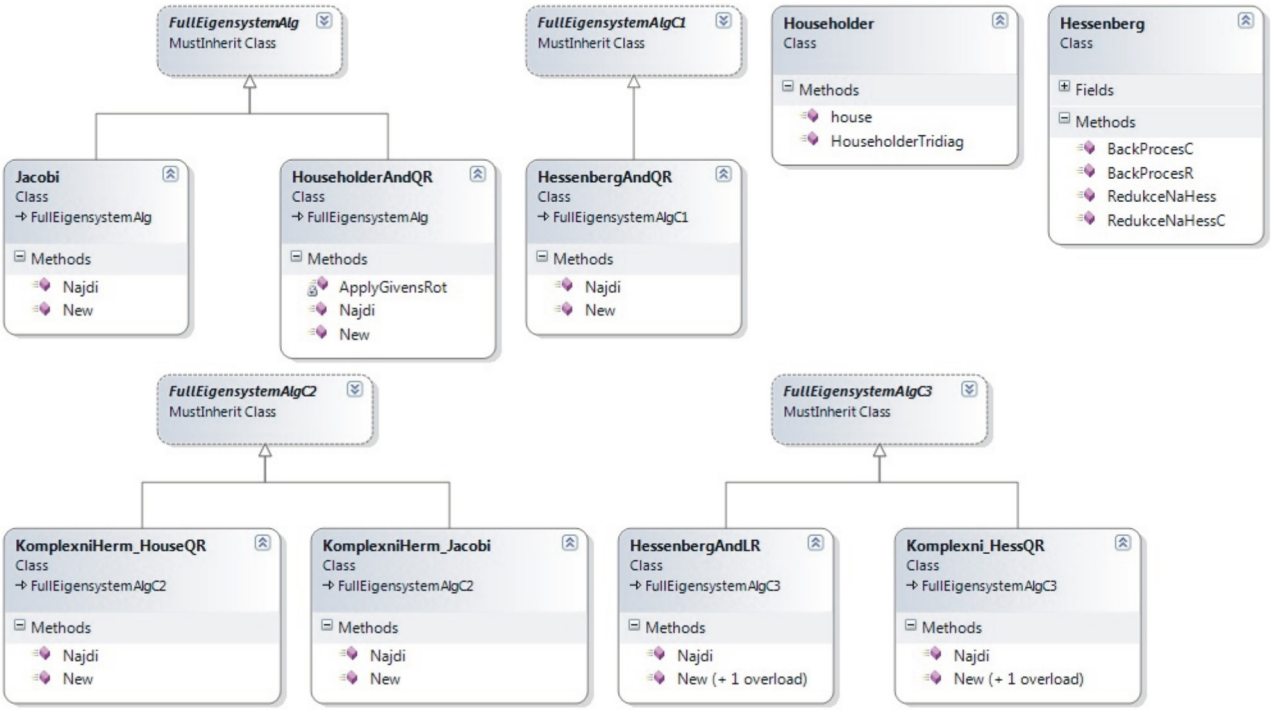
Schéma objektů :



Tento projekt tedy obsahuje základní operace a funkce pro práci s komplexními čísly a maticemi, většinou přesně podle definic. Naprogramovat tyhle třídy nebylo cílem této práce, ale nutným předpokladem a proto si subalgoritmy zde nekladou za cíl efektivitu, ale funkčnost.

8.3 EISPACKNET

Schéma objektů :



Třídy:

- Jacobi
  - Jacobiho transformace reálné symetrické matice
  - Kapitola 2
- Householder
  - Householderova tridiagonalizace reálné symetrické matice
  - Kapitola 3.2.1
- HouseholderAndQR
  - Householderova tridiagonalizace a QR algoritmus reálné symetrické matice

  - Kapitola 3

- Hessenberg
  - Redukce na Hessenbergův tvar reálné matice
  - Kapitola 4.2.1

- HessenbergAndQR
  - Redukce na Hessenbergův tvar a QR algoritmus reálné matice
  - Kapitola 4

- KomplexniHerm_HouseQR
  - Převod na symetrický reálný problém a požití třídy HouseholderQR na hermitovské komplexní matice
  - Kapitoly 5 a 3

- KomplexniHerm_Jacobi
  - Převod na symetrický reálný problém a požití třídy Jacobi na hermitovské komplexní matice
  - Kapitoly 5 a 2

- HessenbergAndLR
  - Redukce na Hessenbergův tvar a LR algoritmua na komplexní matice
  - Kapitola 6.2

- Komplexni_HessQR
  - Převod na reálný problém a HessenbergAndQR komplexní matice
  - Kapitoly 6.3 a 4

Všechny třídy, které dědí FullEigensystemAlg# se ve VB.net použijí například takto:

```
dim A as MaticeC = MaticeC.LoadFromMatlabAsciiFile("MR.txt", "MI.txt")
dim algoritmus as new HessenbergAndLR(A)
dim VlastniCisla as VektorC = algoritmus.Najdi()
dim VlastniVektorPrislusnyPrvnimuVlCislu as VektorC = algoritmus.V.GetSloupec(0)
```

v C#.net

```
MaticeC A = MaticeC.LoadFromMatlabAsciiFile("MR.txt", "MI.txt");
HessenbergAndLR algoritmus = new HessenbergAndLR(A);
VektorC VlastniCisla = algoritmus.Najdi();
VektorC VlastniVektorPrislusnyPrvnimuVlCislu = algoritmus.V.GetSloupec(0);
```

9 Testy

Jako poslední součást této práce jsem se snažil porovnat mé algoritmy s matlabem, především porovnat čas běhu jednotlivých algoritmů. Matlab pro řešení eigensystému příkazem eig používá rutiny knihovny LAPACK. Knihovna LAPACK je vysoce optimalizovaná a velice rozsáhlá volně dostupná knihovna původně Fortranovského kódu (http://www.netlib.org/lapack/) vyvíjená pod grantem National Science Foundation. Tato knihovna obsahuje velké množství jednotlivých algoritmů, které se často navzájem volají a proto pro lepší orientaci vznikla celá kniha LAPACK Users' Guide, Third Edition. Při rozplétání spletitého systému rutin, jsem došel k závěru, že matlab na symetrické matice používá Householderovu tridiagonalizaci a QR algoritmus v upravené verzi nazvané Pal-Walker-Kahan, a pro obecné reálné i komplexní matice redukci na Hessenbergův tvar a opět nějaké verze QR algoritmu, s tím, že implicitně ještě matici v tomto případě tzv. balancuje (jedná se o podobnostní transformaci za účelem snížení normy matice, protože v QR algoritmech je zaokrouhlovací chyba násobkem normy matice, více se tímto subalgoritmem v této práci nezabývám).

Teď už tedy vím s čím budu srovnávat a můžu tedy přistoupit k samotným testům. Testoval jsem husté matice 300x300 vygenerované matlabem z rovnoměrného rozdělení. Mé algoritmy jsem spouštěl s implicitní přesností $1e-15$. Testy byly provedeny bez hledání vlastních vektorů. Časy v tabulce jsou ve formátu MM:SS.SSSS.

Algoritmus	Můj čas	Matlab	Typ matice
Jacobi	04:23.7438	00:00.1584	$\mathbb{R}$ - symetrická
HouseholderAndQR	01:08.6400	00.00.1584	
HessenbergAndQR	00:46.7064	00:00.2522	$\mathbb{R}$
HessenbergAndLR	00:32.6750	00:02.0864	$\mathbb{C}$

Z toho, ale i z celé práce, je zřejmé, že algoritmy jsem rozebíral s progresí v jejich chápání a že je i v samotných algoritmech. Jacobiho metoda používá pouze sérii ortogonálních transformací, v každé se stanovuje matice transformace. Algoritmus HouseholderAndQR si nejprve matici transformuje na vhodnější maticový tvar a pak opět konstruuje sérii ortogonálních transformací, nyní je ale tato transformace stanovena pomocí QR rozkladu matice, čili provedením $n-1$ subkroků. HessenbergAndQR, kromě toho, že je určen pro obecné reálné matice a je inicializován transformací na jiný maticový tvar jiným subalgoritmem, je odlišný od toho předchozího ve dvou hlavních bodech: prováděním dvojkroků se vyhne komplexní aritmetice a je ve formě tazvané implicitní QR, tedy že v jednotlivých QR krocích nedochází k formulaci shiftnuté matice a jejímu QR rozkladu. Nakonec LR algoritmus používá hodně podobných prvků se všemi algoritmy, ale na trochu jiné bázi než QR.

Samozřejmě z časů je evidentní, že všechny kromě posledního algoritmu, v takové formě v jaké jsem je naprogramoval, jsou z praktického hlediska nepoužitelné, že použiji HessenbergAndLR i na klidně symetrickou $\mathbb{R}$ matici. Z předchozího odstavce, ale vyplývá na mě otestovaný edukativní přínos rozboru všech předchozích algoritmů.

10 Závěr

Cílem této práce bylo hlavně seznámit se se zajímavými algoritmy používanými k řešení eigensystémů matic a pochopit jak a proč fungují a dále pak naprogramovat je na perspektivní platformě .net. Samozřejmě tento problém je velice rozsáhlý a nepopsaných algoritmů je ještě mnoho, stejně jako různých vylepšení popsaných. Při samotné tvorbě kódu jsem pak následně použil pouze tuto práci a tedy mým cílem zřejmě nebylo přepsat nějaký kód do jiného jazyku, ale vytvořit kód svůj. Mnou naprogramované kódy si nekladou za cíl vždy tak úplně efektivnost, jako spíše přehlednost a použití nástrojů objektově orientovaného programovacího jazyku.

Takto stanovený cíl jsem ve své práci dosáhl, všechny algoritmy pochopil a

naprogramoval, na výraznější problémy jsem nenarazil, snad jen že ve většině mých zdrojů se otázce vlastních vektorů příliš nevěnují a zaměřují se na vlastní čísla a proto bylo nejobtížnější částí mé práce právě otázky vlastních vektorů a samozřejmě ladění samotného kódu. Myslím, že jsem se zdokonalil v tvorbě matematického a numerického zdrojového kódu, a zjistil, že naprogramovat *fungující* složitější numerický algoritmus na potřicáté je dost velký úspěch, nemluvě o tom, že *fungující* nemusí být dostačující.

Při vypracování jsem postupoval následovně. Nejprve jsem si vybral určitý algoritmus podle úvodu k [4] a zaměření mé práce (viz úvod). Pak jsem si ve stejné publikaci zjistil, co je to přesně za algoritmus, abych si ho mohl najít primárně v [7], někdy pak v [8], protože v [4] se příliš nezabývají matematickým podkladem algoritmů. Po principiálním pochopení rutiny jsem o ní začal psát do této práce, pokud jsem opravdu nemohl něco pochopit, nahlédl jsem do [2], kde je většina věcí vysvětlena velmi jednodušše (je zde evidentní, že jde o publikaci určenou i nematematikům, ale jinak jde v podstatě o přepis [4], co se týče eigensystemů). Když jsem dopsal teoretickou část, začal jsem psát program, pokud jsem narazil na nějaký problém, hledal jsem jeho řešení ve všech [1]-[9], a pokud se jednalo o složitější problém, který jsem nebyl schopen vyřešit sám, dopsal jsem ho pak do této práce.

Literatura

[1] Troelsen A., Pro VB 2008 and the .NET 3.5 Platform,1. vydání. Apress, 2008.

[2] Press W. a kol., Numerical Recipes in C: The Art of Scientific Computing,2. vydání. Cambridge University Press, 1992.

[3] Smith B.T., Boyle J.M., Dongarra, J.J., Matrix Eigensystem routines - Eispack guide,2. vydání. Springer, Berlin, 1976.

[4] Wilkinson J.H., Reinsch C.,Handbook for Automatic Computation,1. vydání. Springer-Verlag, 1971.

[5] Horn R. A., Johnson C. R., Matrix Analysis,4. vydání. Cambridge University Press, 1990.

[6] Saad Y.,Numerical Methods for Large Eigenvalue Problems. Manchester University Press.

[7] Golub G.H., Val Loan C.F., Matrix Computations. The Johns Hopkins University Press, 3. vydání, Baltimore, 1996.

[8] Wilkinson J.H., The Algebraic Eigenvalue Problem. Oxford University Press, London, 1965.

[9] Stewart G.W., Matrix Algorithms, Volume II - Eigensystems. Society for Industrial and Applied Mathematics Philadelphia.