Usporedba Jacobijevog algoritma i OR algoritma za računanje svojstvenih vrijednosti

Pavić, Hrvoje

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Hrvoje Pavić

Matični broj: 45027/16-R

Studij: Poslovni sustavi

USPOREDBA JACOBIJEVOG ALGORITMA I QR ALGORITMA ZA RAČUNANJE SVOJSTVENIH VRIJEDNOSTI

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Bojan Žugec

Varaždin, rujan 2021.

Hrvoje Pavić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/ica potvrdio/la prihvaćanjem odredbi u sustavu FOI-radovi

---

Sažetak

U ovom radu napravljena je usporedba Jacobijevog i QR algoritma za određivanje svojstvenih vrijednosti. Prvo je općenito objašnjen problem svojstvenih vrijednosti, te je prikazan na primjeru. Glavni dio završnog rada čine analiza Jacobijevog algoritma i QR algoritma. Svaki algoritam je matematički objašnjen, a zatim i prikazan na konkretnom primjeru. Također, programski kod za svaki algoritam realiziran je u programskom jeziku Python. Na kraju je napravljena usporedba između Jacobijevog i QR algoritma.

Ključne riječi: svojstvene vrijednosti; Jacobijev algoritam; QR algoritam; Givensova rotacija; simetrične matrice; svojstveni vektor; usporedba;

Sadržaj

1. Uvod

Svojstveni problem(eng. Eigenproblem) je problem pronalaska svojstvenih vektora(eng. Eigenvector) prilikom linearne transformacije te odgovarajuće svojstvene vrijednosti(eng. Eigenvalue). Nas zanima postoje li vektori čiji smjer nakon primjene linearne transformacije ostaje nepromijenjen. Ako postoje, takve vektore nazivamo svojstvenim vektorima, a rezultantni vektor je umnožak svojstvenog vektora i svojstvene vrijednosti.

Svojstvene vrijednosti danas povezujemo s linearnom algebrom i teorijom matrica. Povijesno gledano svojstvene vrijednosti su se prvotno koristile u druge svrhe. U 18. stoljeću Leonhard Euler je prvi otkrio važnost nepromijenjenih osi prilikom rotacije krutih tijela. Joseph-Louis Lagrange je kasnije povezao nepromijenjene osi prilikom rotacije krutog tijela i svojstvene vektore.

Računanje svojstvenih vrijednosti ručno bio je jako veliki problem prije pojave računala. No s dolaskom prvih računala razvijeni su razni algoritmi za računanje svojstvenih vrijednosti. Većina algoritama je bila razvijena i prije dolaska računala, ali tek rješavanjem računala omogućeno je odrediti svojstvene vrijednosti za jako velike matrice.

Prvi numerički algoritam za računanje svojstvenih vrijednosti predložio je Richard von Mises 1929. godine a zove se Power metoda. U ovom radu Power metoda nije obrađena, već Jacobijev i QR algoritam. QR algoritam je najpoznatiji algoritam za računanje svojstvenih vrijednosti danas.

Svojstvene vrijednosti se danas primjenjuju u raznim granama znanosti. U fizici svojstvene vrijednosti povezujemo s vibracijama, npr. prilikom titranja mosta ili neke druge građevine. U matematici svojstvene vrijednosti imaju jako važnu ulogu u rotaciji tijela. U ovom radu ćemo pokazati svojstvene vrijednosti na primjerima rotacije likova u 2D i rotaciji tijela u 3D. Nadalje navedeni algoritmi će nam pomoći odrediti svojstvene vrijednosti za $n \times n$ matrice.

U nastavku rada detaljno su opisani Jacobijev i QR algoritam, te je prikazana primjena istih.

2. Metode i tehnike rada

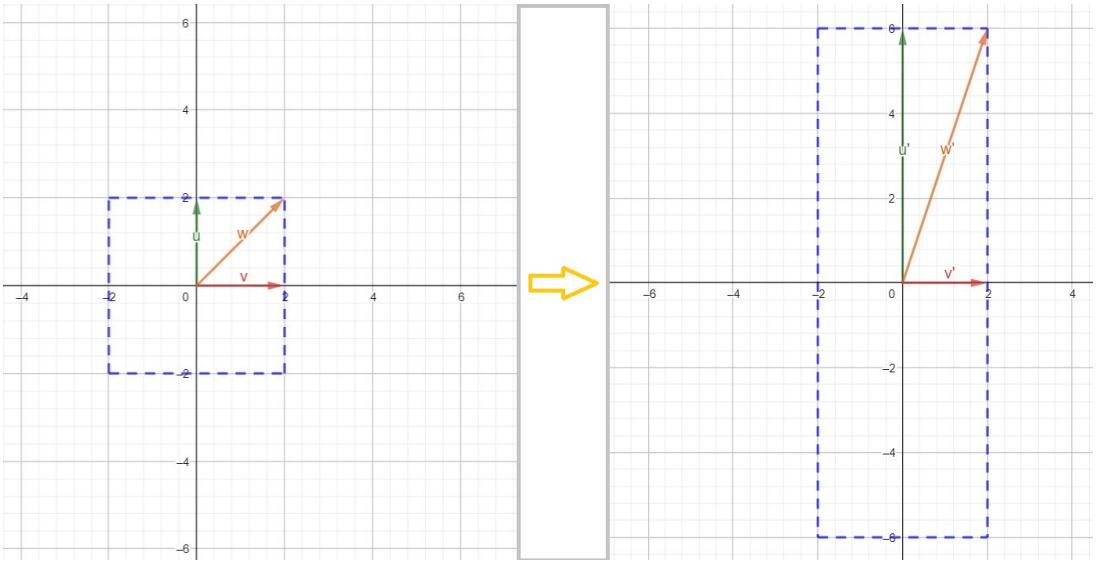
Literatura proučena u ovom završnom radu vezana je za svojstvene vrijednosti. Prikazana su 2 numerička algoritma za računanje svojstvenih vrijednosti, Jacobijev i QR algoritam. Definirani su koraci prolaska kroz svaki algoritam uz primjere. Matematičke operacije u primjerima su provjeren u online kalkulatoru Symbolab. Implementacija samih algoritama je napravljena pomoću programskog jezika Python. Implementirani kod prikazan je pomoću biblioteke listings. Rad je napravljen u Online LaTeX Editoru Overleaf.

3. Problem svojstvenih vrijednosti

Neka je V vektorski prostor nad poljem F, a $f : V \to V$ linearni operator. Problem svojstvenih vrijednosti za navedeni linearni operator je problem određivanja skalara $\lambda \in F$ te pripadajućeg vektora $x \in V$ za koje vrijedi :

$$f(x) = \lambda x \tag{3.1}$$

Vektor $x \in V$ naziva se svojstveni vektor akko vrijedi da je $x \neq 0$, a skalar $\lambda \in F$ je svojstvena vrijednost navedenog operatora[1].



Slika 1: Svojstveni vektor i svojstvena vrijednost u 2D

Slika (1), lijevi dio, prikazuje kvadrat i pripadajuće vektore : $\vec{u}$, $\vec{w}$ i $\vec{v}$. Sada pretpostavimo da smo linearno transformirali geometrijski lik s lijeve strane slike, te smo kao rezultat dobili geometrijski lik na desnoj strani slike[2]. Linearnu transformaciju možemo prikazati pomoću matrice transformacije T. Nakon djelovanja matrice transformacije T povećali smo(u ovom primjeru, općenito možemo i smanjiti, izobličiti ili rotirati) početni geometrijski lik po osi ordinate te rezultat vidimo na desnoj strani slike. Novi lik ima svoje pripadajuće vektore : $\vec{u'}$, $\vec{w'}$ i $\vec{v'}$. Nas zanima smjer vektora, odnosno, nakon što smo izvršili linearnu transformaciju postoje li i dalje vektori čiji je smjer ostao nepromijenjen. Vidimo da smjer vektora $\vec{w}$ i $\vec{w'}$ više nije isti, međutim smjer vektora $\vec{u}$ odgovara smjeru vektora $\vec{u'}$ i smjer vektora $\vec{v}$ odgovara smjeru vektora $\vec{v'}$. Ako je smjer vektora nakon linearne transformacije isti smjeru početnog vektora, onda je početni vektor **svojstveni vektor**. U ovom slučaju vektori $\vec{u}$ i $\vec{v}$ su svojstveni vektori. Primijetimo da je vektor $\vec{u'}$ duži u odnosu na vektor $\vec{u}$. Omjer duljine vektora $\vec{u'}$ i $\vec{u}$ označavamo sa λ, gdje je λ oznaka za **svojstvenu vrijednost**, a vrijedi $\lambda \in \mathbb{R}$.
Dakle djelovanjem matrice transformacije T na vektor $\vec{u}$ kao rezultat dobili smo transformirani vektor $\vec{u'}$ koji je λ puta veći(u ovom primjeru, općenito može biti i manji ili jednak) od početnog

vektora $\vec{u}$, za kojeg smo utvrdili da je svojstveni vektor[3]. Taj izraz možemo zapisati kao :

$$T\vec{u} = \lambda\vec{u} \tag{3.2}$$

Za ovaj primjer rekli smo da je i $\vec{v}$ svojstveni vektor, gdje je svojstvena vrijednost $\lambda = 1$ jer se duljina vektora $\vec{v'}$ nije mijenjala nakon linearne transformacije, a smjer je ostao netaknut. Valja još spomenuti da λ može biti negativan broj, što znači da je smjer transformiranog vektora ostao isti, ali mu se orijentacija promijenila.

3.1. Računanje svojstvene vrijednosti i svojstvenog vektora

Raspišimo sada jednadžbu (3.2)[4]. Primijetimo da s lijeve strane jednadžbe (3.2) vektor $\vec{u}$ množimo s matricom transformacije T dimenzije $n \times n$, a na desnoj strani vektor $\vec{u}$ množimo sa skalarom λ. Kako bi mogli raspisati ovaj izraz prvo moramo skalar λ raspisati kao umnožak jedinične matrice I dimenzije $n \times n$ i samog sebe, pa dobivamo :

$$\begin{aligned} T\vec{u} &= \lambda I\vec{u} \\ T\vec{u} - \lambda I\vec{u} &= \vec{0} \\ (T - \lambda I)\vec{u} &= \vec{0} \end{aligned} \tag{3.3}$$

Nakon sređivanja izraza, u zagradi smo dobili $(T-\lambda I)$, što je obično oduzimanje matrica. Kada determinanta izraza $(T - \lambda I)$ ne bi bila jednaka nuli, to znači da izraz $(T - \lambda I)$ ima svoj inverz. Prema tome mogli bi cijelu jednadžbu pomnožiti sa $(T - \lambda I)^{-1}$, što bi značilo da je $\vec{u} = \vec{0}$. Uvrštavanje $\vec{u} = \vec{0}$ u jednadžbu (3.2) bi bilo valjano, ali kako je rješenje te jednadžbe $0 = 0$, rješenje $\vec{u} = \vec{0}$ nazivamo trivijalnim i ne govori nam ništa o svojstvenih vrijednostima, tako da nas takvo rješenje ne zanima. Zbog toga determinanta matrice $(T - \lambda I)$ mora biti jednaka nuli, pa možemo zapisati :

$$\det(T - \lambda I) = 0 \tag{3.4}$$

Izraz (3.4) ima svoj naziv, karakteristična ili **svojstvena jednadžba**. Rezultat sređivanja jednadžbe jest karakteristični ili **svojstveni polinom**, a oznaka je $k_T(\lambda)$. Izraz (3.4) možemo zapisati kao :

$$k_T(\lambda) = \det(T - \lambda I) = 0 \tag{3.5}$$

Rješavanjem svojstvene jednadžbe dobivamo odgovarajuće vrijednosti za λ, odnosno **svojstvene vrijednosti**. Nakon što saznamo svojstvene vrijednosti možemo lako izračunati i svojstvene vektore. U nastavku ćemo riješiti jedan primjer te konkretno pokazati kako se određuju svojstvene vrijednosti i svojstveni vektori, te opisati još neke zanimljive termine vezane za problem svojstvenih vrijednosti.

Primjer 1. Neka je $f : V \to V$ linearni operator. Izračunajmo svojstvene vrijednosti i svojstvene vektore za sljedeću simetričnu matricu transformacije $M(3 \times 3)$[5] :

$$M = \begin{bmatrix} -1 & 2 & 2 \\ 2 & 2 & -1 \\ 2 & -1 & 2 \end{bmatrix}$$

Prvo moramo riješiti svojstvenu jednadžbu (3.5) $k_M(\lambda) = \det(M - \lambda I) = 0$:

$$\det\left(\begin{bmatrix} -1 & 2 & 2 \\ 2 & 2 & -1 \\ 2 & -1 & 2 \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix}\right) = 0$$

$$\begin{vmatrix} -1-\lambda & 2 & 2 \\ 2 & 2-\lambda & -1 \\ 2 & -1 & 2-\lambda \end{vmatrix} = 0$$

$$(-1-\lambda)[(2-\lambda)(2-\lambda) - (-1)(-1)] - 2[2(2-\lambda) - 2(-1)] + 2[2(-1) - 2(2-\lambda)] = 0$$

$$(-1-\lambda)[\lambda^2 - 4\lambda + 3] - 2[-2\lambda + 6] + 2[2\lambda - 6] = 0$$

$$-\lambda^2 + 4\lambda - 3 - \lambda^3 + 4\lambda^2 - 3\lambda + 4\lambda - 12 + 4\lambda - 12 = 0$$

$$-\lambda^3 + 3\lambda^2 + 9\lambda - 27 = 0$$

$$-\lambda^2(\lambda - 3) + 9(\lambda - 3) = 0$$

$$(\lambda - 3)(-\lambda^2 + 9) = 0$$

$$(\lambda - 3)(-\lambda + 3)(\lambda + 3) = 0$$

$$\lambda_1 = 3, \lambda_2 = -3$$

Dobili smo 2 rješenja, $\lambda_1 = 3$ i $\lambda_2 = -3$ su svojstvene vrijednosti. Prije nego li odredimo svojstvene vektore, dobivena rješenja možemo bolje zapisati. Skup svih svojstvenih vrijednosti linearnog operatora f nazivamo **spektar**, a oznaka je $\sigma(f)$[6]. Dobivena rješenja sada možemo zapisati kao :

$$\sigma(f) = \{3, -3\}$$

Sada možemo odrediti svojstvene vektore, za obje svojstvene vrijednosti. Uvrštavanjem dobivenih vrijednosti u jednadžbu (3.3) dobivamo :

$$\begin{bmatrix} -1-3 & 2 & 2 \\ 2 & 2-3 & -1 \\ 2 & -1 & 2-3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -4x + 2y + 2z \\ 2x - 1y - 1z \\ 2x - 1y - 1z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\left[\begin{array}{ccc|c} -4 & 2 & 2 & 0 \\ 2 & -1 & -1 & 0 \\ 2 & -1 & -1 & 0 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} -4 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Nakon Gauss-Jordanove eliminacije iz sređene matrice očitavamo :

$$-4x + 2y + 2z = 0| : 2$$
$$-2x + y + z = 0$$

U ovom slučaju imamo 2 parametra, $p \in \mathbb{R}$ gdje je $y = p$ i $u \in \mathbb{R}$ gdje je $z = u$:

$$x = \frac{p+u}{2}$$

Prvo ćemo definirati i odrediti **svojstveni potprostor**[7]. Potprostor svih svojstvenih vektora $\vec{x}$ koji pripadaju svojstvenoj vrijednosti λ, odnosno za koje vrijedi jednadžba (3.2) zove se svojstveni potprostor od T koji pripada svojstvenoj vrijednosti λ. Svojstveni potprostor od λ označavamo sa $S(\lambda)$. Konkretno za ovaj primjer gdje je $\lambda = 3$ svojstveni potprostor od M je :

$$S(3) = \left\{ \left(\frac{y+z}{2}, y, z \right) : y, z \in \mathbb{R} \right\}$$

Kako bi odredili **bazu** za $S(3)$, izraz $\left(\frac{y+z}{2}, y, z\right)$ možemo raspisati :

$$\left(\frac{y+z}{2}, y, z \right) = y(\frac{1}{2}, 1, 0) + z(\frac{1}{2}, 0, 1)$$

Iz čega slijedi da je jedna baza za $S(3)$:

$$B_{S(3)} = \left\{ \left(\frac{1}{2}, 1, 0 \right), \left(\frac{1}{2}, 0, 1 \right) \right\}$$

Definirajmo sada geometrijsku kratnost svojstvene vrijednosti λ. **Geometrijska kratnost**[8] od λ je dimenzija svojstvenog potprostora $S(\lambda)$. Dimenziju najlakše očitamo iz rješenja za jednu bazu od $S(\lambda)$. U ovom primjeru vidimo da našu bazu čine dvije dimenzije. Oznaka za dimenziju je $d(\lambda)$, a u ovom primjeru za $\lambda = 3$ ona iznosi :

$$d(3) = 2$$

Što povlači da i geometrijska kratnost iznosi 2. Još možemo i definirati **algebarsku kratnost**[8], koja se očitava iz svojstvene jednadžbe koju smo dobili prije nego li smo izračunali svojstvene vrijednosti λ. Za ovaj primjer to je bila jednadžba koju ćemo malo preurediti :

$$(\lambda - 3)(-\lambda + 3)(\lambda + 3) = 0$$
$$(-\lambda + 3)(\lambda - 3)(\lambda + 3) = 0$$
$$-1(\lambda - 3)(\lambda - 3)(\lambda + 3) = 0$$
$$-1(\lambda - 3)^2(\lambda + 3) = 0$$

Algebarska kratnost jednaka je potenciji u ovoj jednadžbi koja se ne može više srediti, uz element za koji je λ nultočka. U ovom slučaju to je potencija uz $(\lambda - 3)^2$. Za $\lambda = -3$ algebarska kratnost će iznositi 1 jer $(\lambda + 3)^1$. Također bitno je naglasiti da je algebarska kratnost uvijek manja ili jednaka od geometrijske kratnosti, odnosno $l(\lambda) \leq d(\lambda)$. Označavamo ju sa $l(\lambda)$,

dakle pišemo :

$$l(3) = 2$$

Sada napokon možemo odrediti svojstvene vektore za $\lambda = 3$. Svojstvenih vektora ima ∞ mnogo, a kako bi odredili barem jedan moramo jednom od parametara p ili u odrediti bilo koju proizvoljnu vrijednost. Recimo da je $u = p$, iz toga slijedi da je svojstveni vektor $(\frac{2p}{2}, p, p)$ odnosno $p(1, 1, 1)$. Svojstveni vektor u ovom slučaju za $\lambda = 3$ je $(1, 1, 1)$.
Za provjeru možemo uvrstiti svojstveni vektor u jednadžbu (3.3) te potvrditi da su lijeva i desna strana jednake.
Odredimo sada sve ove vrijednosti redom za $\lambda = -3$:

$$S(-3) = \{(-(y + z), y, z) : y, z \in \mathbb{R}\}$$
$$B_{S(-3)} = \{(-1, 1, 0), (-1, 0, 1)\}$$
$$d(-3) = 2$$
$$l(-3) = 1$$

Zbog 2 parametra opet moramo odrediti barem jedan. I dalje će biti ∞ mnogo svojstvenih vektora za svojstvenu vrijednost $\lambda = -3$. Recimo da je $u = p$, gdje slijedi da je svojstveni vektor $(-2p, p, p)$ odnosno $p(-2, 1, 1)$. Svojstveni vektor u ovom slučaju za $\lambda = -3$ iznosi $(-2, 1, 1)$.

4. Jacobijev algoritam

Carl Gustav Jacob Jacobi bio je njemački matematičar koji je sredinom 19. stoljeća predložio ideju za računanje svojstvenih vrijednosti, no njegova ideja doživjela je uspjeh sredinom 20. stoljeća pojavom računala.

Jacobijev algoritam je iterativna metoda računanja svojstvenih vrijednosti za simetrične realne matrice. Kako algoritam ima puno iteracija (ovisno o matrici) nije čudno da je ovaj algoritam doživio uspjeh tek 100 godina nakon što je predložen, pojavom računala.

U nastavku ćemo proći kroz algoritam na primjeru koji nema puno iteracija, te zatim u Pythonu riješiti zadatak koji ima više iteracija.

Givensova rotacija je rotacija u prostoru razapeta sa 2 koordinatne osi[9]. Matrica Givensove rotacija nam pomaže u provođenju algoritma za Jacobijev i QR algoritam. Računski, to je ortogonalna matrica koja ima svoju svrhu prilikom svake iteracije. Inače Carl Gustav Jacobi je sam predložio Jacobijevu rotaciju, ali ona je ekvivalentna Givensovoj, a kako nam je potrebna za računanje oba algoritma onda ćemo ju i opisati. Prikazujemo ju pomoću matrice u sljedećoj formi :

$$G(i,j,\Theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos(\Theta) & \cdots & -\sin(\Theta) & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \sin(\Theta) & \cdots & \cos(\Theta) & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}. \tag{4.1}$$

Givensova rotacija je rotacija za kut Θ između i i j te je potrebna za računanje Jacobijevog i QR algoritma. Neka je zadana simetrična matrica transformacije A_n gdje je $n \in \mathbb{N}^0$:

$$A_n = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Matrica je simetrična što povlači : $a_{21} = a_{12}$. Kako bi proveli ortogonalnu transformaciju mora vrijediti formula :

$$A_{n+1} = G_n^T A_n G_n = \begin{bmatrix} a_{11}^{'} & a_{12}^{'} \\ a_{21}^{'} & a_{22}^{'} \end{bmatrix} \tag{4.2}$$

Gdje je A_{n+1} transformirana matrica A_n nakon n iteracija. G_n je Givensova matrica rotacije (4.1) nakon n iteracija također. Sada možemo raspisati izraz (4.2) :

$$A_{n+1} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}\cos(\theta)^2 + 2a_{12}\sin(\theta)\cos(\theta) + a_{22}\sin(\theta)^2 & \left(\cos(\theta)^2 - \sin(\theta)^2\right)a_{12} + \sin(\theta)\cos(\theta)\left(a_{22} - a_{11}\right) \\ \left(\cos(\theta)^2 - \sin(\theta)^2\right)a_{12} + \sin(\theta)\cos(\theta)\left(a_{22} - a_{11}\right) & a_{11}\sin(\theta)^2 - 2a_{12}\cos(\theta)\sin(\theta) + a_{22}\cos(\theta)^2 \end{bmatrix} \tag{4.3}$$

Znamo da nam vrijednosti u matrici A_{n+1} izvan glavne dijagonale moraju biti nula :

$$\begin{gathered} a_{12}^{'} = a_{21}^{'} = 0 \\ \left(\cos(\theta)^2 - \sin(\theta)^2\right) a_{12} + \sin(\theta)\cos(\theta)\left(a_{22} - a_{11}\right) = 0 \\ gdje\ vrijedi : \left(\cos(\theta)^2 - \sin(\theta)^2\right) = \cos(2\theta)\ i\ 2\sin(\theta)\cos(\theta) = \sin(2\theta) \\ \cos(2\theta)a_{12} + \frac{\sin(2\theta)}{2}\left(a_{22} - a_{11}\right) = 0\ :\ |\ \cos(2\theta) \\ a_{12} + \frac{\tan(2\theta)}{2}\left(a_{22} - a_{11}\right) = 0 \\ \tan(2\theta) = \frac{-2a_{12}}{a_{22} - a_{11}} = \frac{2a_{12}}{a_{11} - a_{22}} \end{gathered} \tag{4.4}$$

U nastavku ćemo objasniti Jacobijev algoritam kroz primjer, te svaki korak detaljno opisati :

Primjer 2. Neka je $f : V \to V$ linearni operator. Koristeći Jacobijev algoritam[10] pronađite svojstvene vrijednosti za danu matricu simetričnu matricu M:

$$M = \begin{bmatrix} -2 & 6 & 10 \\ 6 & 8 & 6 \\ 10 & 6 & -2 \end{bmatrix}$$

Korak 1) Za zadanu matricu M moramo odrediti najveću vrijednost koja se ne nalazi na dijagonali. Kako je riječ o simetričnim matricama uvijek ćemo imati barem dvije takve vrijednosti. U ovom slučaju to su vrijednosti M_{13} i M_{31}. Svejedno je koju vrijednost odaberemo.

$$M_{max} = M_{13} = 10$$

Korak 2) Sada kreiramo Givensovu matricu (4.1). Na mjestu najveće vrijednosti M_{13} pišemo $-\sin(\Theta)$, a na mjesto simetrično od M_{13}, odnosno M_{31} pišemo $\sin(\Theta)$. Još stavljamo $\cos(\Theta)$ na mjesta dijagonale u redcima gdje su najveće vrijednosti. Prazna mjesta na dijagonali popunimo sa 1, a ostala mjesta sa 0. Givensova matrica zatim izgleda :

$$G = \begin{bmatrix} \cos(\Theta) & 0 & -\sin(\Theta) \\ 0 & 1 & 0 \\ \sin(\Theta) & 0 & \cos(\Theta) \end{bmatrix}$$

Korak 3) Zatim računamo kut transformacije Θ, gdje gledamo vrijednosti iz matrice M koje po pozicijama odgovaraju vrijednostima iz matrice G po formuli (4.4) :

$$\tan(2\theta) = \frac{a_{12}}{a_{11} - a_{22}}$$

$$\tan(2\Theta) = \frac{2 \cdot 10}{-2 - (-2)} = \frac{20}{0} = \infty$$

$$\Theta = \frac{\pi}{4}$$

Korak 4) Sada uvrštavamo vrijednost kuta Θ u Givensovu matricu :

$$G = \begin{bmatrix} \cos(\frac{\pi}{4}) & 0 & -\sin(\frac{\pi}{4}) \\ 0 & 1 & 0 \\ \sin(\frac{\pi}{4}) & 0 & \cos(\frac{\pi}{4}) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}$$

Korak 5) Imamo sve potrebne vrijednosti da provedemo ortogonalnu transformaciju (1. iteraciju), po formuli (4.2) :

$$M_1 = G^T M G$$

$$M_1 = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} -2 & 6 & 10 \\ 6 & 8 & 6 \\ 10 & 6 & -2 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}$$

$$M_1 = \begin{bmatrix} 8 & 6\sqrt{2} & 0 \\ 6\sqrt{2} & 8 & 0 \\ 0 & 0 & -12 \end{bmatrix}$$

Korak 6) Sada ponavljamo postupak. Postupak će biti gotov kada su na dijagonali ostale neke vrijednosti, a svi ostali elementi matrice su jednaki nuli. Te preostale vrijednosti na dijagonali su svojstvene vrijednosti.

$$M_{1max} = M_{1(12)} = 6\sqrt{2}$$

$$\tan(2\Theta_1) = \frac{2M_{1(12)}}{M_{1(11)} - M_{1(22)}} = \frac{2 \cdot 6\sqrt{2}}{8-8} = \frac{12\sqrt{2}}{0} = \infty$$

$$\Theta_1 = \frac{\pi}{4}$$

$$G_1 = \begin{bmatrix} \cos(\frac{\pi}{4}) & -\sin(\frac{\pi}{4}) & 0 \\ \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_2 = G_1^T M_1 G_1$$

$$M_2 = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 6\sqrt{2} & 0 \\ 6\sqrt{2} & 8 & 0 \\ 0 & 0 & -12 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} \sqrt{2}(4\sqrt{2}+6) & 0 & 0 \\ 0 & 8-6\sqrt{2} & 0 \\ 0 & 0 & -12 \end{bmatrix}$$

Korak 7) Nakon 2. iteracije dobili smo da su svi elementi matrice jednaki nuli, osim elemenata na dijagonali. To smo i željeli, te je ovo kraj Jacobijevog algoritma. Dobivene vrijednosti na dijagonali su svojstvene vrijednosti matrice M. Konačan rezultat možemo zapisati :

$$\sigma(f) = \{\sqrt{2}(4\sqrt{2}+6), 8-6\sqrt{2}, -12\}$$

Svojstvene vektore računamo tako da pomnožimo sve Givensove matrice koje smo koristili u

svakoj iteraciji, te zatim iščitamo vrijednosti svojstvenih vektora :

$$G_0 = G \cdot G_1 = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

$$\vec{v_1} = \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{2}}{2} \\ \frac{1}{2} \end{pmatrix}, \vec{v_2} = \begin{pmatrix} -\frac{1}{2} \\ \frac{\sqrt{2}}{2} \\ -\frac{1}{2} \end{pmatrix}, \vec{v_3} = \begin{pmatrix} -\frac{\sqrt{2}}{2} \\ 0 \\ \frac{\sqrt{2}}{2} \end{pmatrix}$$

Prikazani primjer je u najmanju ruku trivijalan ako uzmemo u obzir da je bilo potrebno provesti samo dvije iteracije. Sada ćemo u Pythonu pokazati kako za bilo koje random simetrične matrice ovaj algoritam izračunava svojstvene vrijednosti.

4.1. Jacobijev algoritam u Pythonu

Do sada smo vidjeli Jacobijev algoritam na djelu za dvije iteracije s točno zadanom matricom. Međutim, poanta Jacobijevog algoritma je da saznamo svojstvene vrijednosti za bilo koje realne simetrične matrice većih dimenzija. U ovom poglavlju to je i realizirano. Opisat ćemo svaku funkciju detaljno za realizaciju Jacobijevog algoritma u programskom jeziku Python.

Prikaz korištenih biblioteka i funkcije koja unosi broj za pseudo-random generiranje matrica :

Listing 4.1: Korištene biblioteke

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
np.random.seed(1)
np.set_printoptions(3)
```

Funkcija koja generira nasumičnu simetričnu matricu, ovisno o odabranoj dimenziji prilikom pokretanja programa :

Listing 4.2: Funkcija za generiranje nasumične simetrične matrice

```
def simetricna_matrica(n):
    pocetna = np.random.randint(1, 10, size=(n,n))
    return (pocetna + pocetna.T)/2
```

Ova funkcija omogućuje provjeru svojstvenih vrijednosti za generiranu matricu, ali ne preko Jacobijevog algoritma, već algoritma koji je ugrađen u Pythonu. Glavna svrha ove funkcije je da usporedi dobivene svojstvene vrijednosti Jacobijevim algoritmom te točne svojstvene vrijednosti, kao svojevrsna provjera koda. Funkcija se poziva naredbom $numpy\_test(sim)$ u IDLE Shellu :

Listing 4.3: Funkcija za provjeru svojstvenih vrijednosti

```
def numpy_test(matrica):
    eigval, eigvec = la.eig(matrica)
    print('Svojstvene vrijednosti: ', ' '.join(['{:.2f}'.format(val) for val in
        eigval]))
```

Kako bi provjerili da se konvergencija smanjuje prilikom iteriranja programa, koristimo Vandijagonalnu Frobeniusovu normu [11]. Bez ove funkcije program bi se u 99.99% slučajeva beskonačno vrtio. Iterativna procedura se zaustavlja kada Frobeniusova norma postigne vrijednost manju od zadane tolerancije. Vandijagonalnu Frobeniusovu normu računamo :

$$off(A) = \sqrt{\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} a_{ij}^2} \tag{4.5}$$

Sumu svih vandijagonalnih elemenata kvadriramo, a zatim korjenujemo. Rezultat je broj koji predstavlja konvergenciju, a uspoređujemo ju s našom definiranom tolerancijom. Prilikom svake iteracije vrijednosti elemenata s vanjske dijagonale se smanjuju, međutim oni najčešće nikada neće konvergirati točno u nula, nego neku jako malu vrijednost. Korištenjem ove norme mi određujemo kada je suma svih elemenata van dijagonale dovoljno mala da kažemo da je jednaka nuli. Kada je konvergencija manja od tolerancije tu je kraj programa :

Listing 4.4: Vandijagonalna Frobeniusova norma

```
def odstupanje(matrica):
    # vandijagonalna Frobeniusova norma
    test_matrica = matrica.copy()
    velicina = test_matrica.shape[0]
    for i in range(velicina):
        test_matrica[i,i] = 0
    return np.sqrt(la.norm(test_matrica))
```

U koraku $2)$ riješenog primjera Jacobijevog algoritma moramo odrediti na kojima se pozicijama u matrici nalaze najveće vrijednosti. To omogućuje ova funkcija kako bi onda mogli pravilno primijeniti $sinus$ i $kosinus$ na odgovarajuću poziciju :

Listing 4.5: Funkcija za određivanje pozicije sinusa i kosinusa

```
def pozicije(matrica):
    #pozicija najveceg elementa
    test_matrica = matrica.copy()
    velicina = test_matrica.shape[0]
    for i in range(velicina):
        test_matrica[i,i] = 0
    ind_sin1 = np.unravel_index(np.argmax(test_matrica, axis=None), test_matrica.
        shape)
    ind_sin2 = (ind_sin1[1], ind_sin1[0])
    # pozicija kosinusa
    ind_cos1 = (ind_sin1[0], ind_sin1[0])
    ind_cos2 = (ind_sin1[1], ind_sin1[1])
    return (ind_sin1, ind_sin2, ind_cos1, ind_cos2)
```

U koraku $3)$ riješenog primjera Jacobijevog algoritma moramo odrediti kut. U primjeru smo kut riješili preko $arkustangens$ funkcije. Međutim, primjenom tog načina izračuna kuta nije bilo moguće dobiti rješenje. Moguće da se velike numeričke greške pojavljuju zbog jako malih brojeva prilikom evaluacije $arkustangens$ funkcije. Samim time je ovom metodom nemoguće postići zadovoljnu konvergenciju. Zato koristimo Schurovu dekompoziciju :

Listing 4.6: Schurova dekomopzicija

```
def schurova_dekompozicija(matrica, indeksi):
    kosinus = 1
    sinus = 0
    if matrica[indeksi] != 0:
        tau = (matrica[indeksi[1],indeksi[1]]-matrica[indeksi[0],indeksi[0]])/(2*
            matrica[indeksi])
        if tau >= 0:
            t = 1/(tau+np.sqrt(1+tau**2))
        else:
            t = 1/(tau-np.sqrt(1+tau**2))
    kosinus = t/np.sqrt(1+t**2)
    sinus = 1/np.sqrt(1+t**2)
    return sinus, kosinus
```

U glavnom dijelu programskog koda provodimo Jacobijev algoritam. Nakon provedbe Jacobijevog algoritma dobijemo svojstvene vrijednosti. Program je napravljen da računa svojstvene vrijednosti za bilo koje $n \times n$ matrice, a kao bitan rezultat provođenja algoritma ispisujemo broj iteracija koje su potrebne za određenu matricu. Broj iteracija ćemo i tablično zapisati, kako bi ga mogli usporediti s QR algoritmom :

Listing 4.7: Jacobijev algoritam

```
def Jacobi(matrica, tolerancija):

    velicina = matrica.shape[0]
    iteracija = matrica.copy()
    vektor = np.identity(velicina)
    x = [[0, odstupanje(iteracija)]]
    i = 0
    while (odstupanje(iteracija) > tolerancija):

        # TRAZENJE NAJVECEG ELEMENTA
        ind_sin1, ind_sin2, ind_cos1, ind_cos2 = pozicije(iteracija)
        sinus, kosinus = schurova_dekompozicija(iteracija, ind_sin1)

        #kut = 0.5*np.arctan(2*iteracija[ind_sin1]/(iteracija[ind_cos1]-iteracija[
            ind_cos2]))
        #sinus, kosinus = (np.sin(kut), np.cos(kut))

        # GIVENSOVA MATRICA
        givens = np.identity(velicina)
        givens[ind_sin1] = -sinus
        givens[ind_sin2] = sinus
        givens[ind_cos1] = kosinus
        givens[ind_cos2] = kosinus
```

```
        # KORAK ITERACIJE
        iteracija = givens.T@iteracija@givens
        vektor = vektor@givens

        #print(givens, end='\n\n')
        print(iteracija, end='\n\n')

        # SPREMANJE ODSTUPANJA
        i+=1
        x.append([i, odstupanje(iteracija)])

    # ISPIS DIJAGONALNIH ELEMENTA (EIGENVRIJEDNOSTI)
    for i in range(velicina):
        print(iteracija[i,i], end=', ')
    print()

    # LISTA ODSTUPANJA ZA GRAF
    return x


sim = simetricna_matrica(5)

print(sim, end='\n\n')

diag = np.array(Jacobi(sim,0.00001))
print('Broj iteracija: ', len(diag))
plt.plot(diag[:,0], diag[:,1])
plt.show()
```

Preostalo je pokazati rezultate provedbe Jacobijevog algoritma. To ćemo uraditi tablično i grafički.

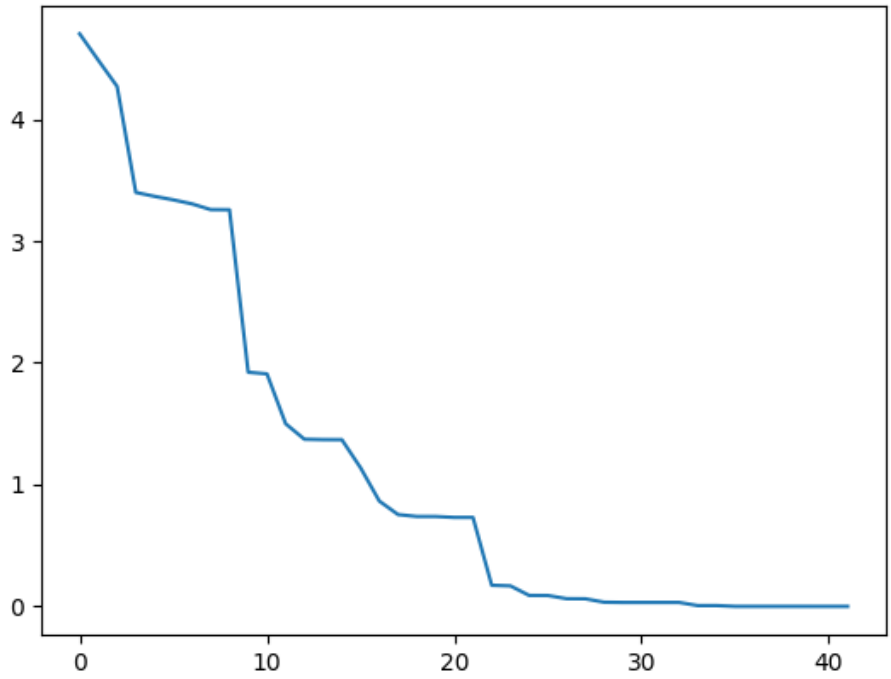
4.1.1. Rezultati Jacobijevog algoritma u Pythonu

Tablica prikazuje dimenziju matrice i broj iteracija koji je bilo potrebno provesti kako bi se izračunale svojstvene vrijednosti :

Dimenzija matrice	Broj iteracija
4x4	21
5x5	42
6x6	69
7x7	85
8x8	108
9x9	136
10x10	176
11x11	207
12x12	248
13x13	311
14x14	342
15x15	412
20x20	772
50x50	4803
100x100	19929

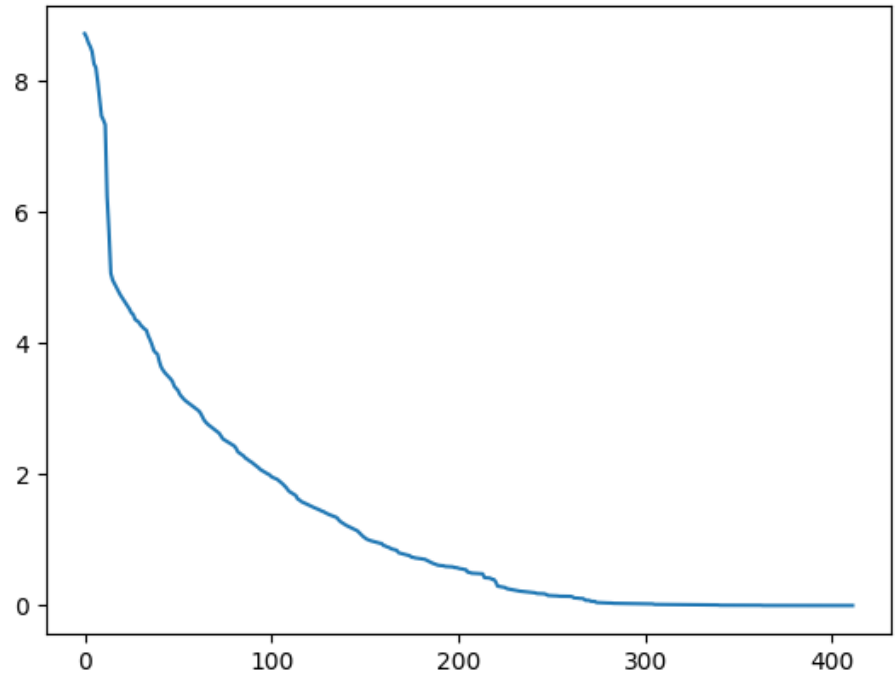
Tablica 1: Tablični prikaz potrebnog broja iteracija da se dobiju svojstvene vrijednosti

Još ćemo grafički prikazati krivulju broja iteracija (os apscise) te konvergencije (os ordinate). Možemo primljetiti da krivulja poprima sve ljepši oblik ponavljanjem više iteracija :



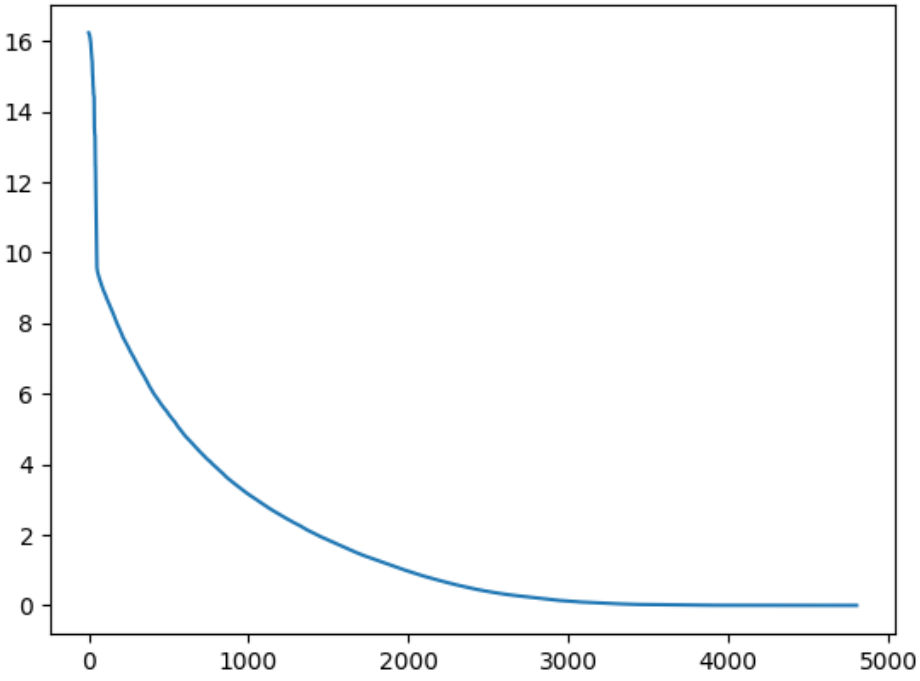
Slika 2: Broj iteracija potreban da izračunamo λ za matricu 5×5

Na grafu (2) možemo uočiti kako povećanjem iteracija, konvergencija pada. Konvergenciju računamo preko (4.5), odnosno suma vandijagonalnih elemenata se smanjuje. Kako bi izračunali svojstvene vrijednosti, suma svih vandijagonalnih elemenata mora biti približna nuli. Ovaj graf prikazuje koliko je iteracija potrebno da bi transformirali početnu matricu, do matrice gdje su vandijagonalni elementi nule, a elementi na dijagonali su u tom slučaju svojstvene vrijednosti.



Slika 3: Broj iteracija potreban da izračunamo λ za matricu 15×15

Izgled grafa (3) poprima pravilniji oblik krivulje povećanjem broja iteracija.
Izgled grafa (4) kod matrica velikih dimenzija poprima skoro pa savršen oblik, jer je potrebno provesti puno više iteracije za matrice većih dimenzija.



Slika 4: Broj iteracija potreban da izračunamo λ za matricu 50×50

5. QR algoritam

QR algoritam razvijen je sredinom 20. stoljeća, a glavna ideja ovog algoritma je da zadanu matricu dekomponiramo, odnosno zapišemo kao produkt ortogonalne matrice (Q) i gornje trokutaste matrice (R). Postupak je detaljnije objašnjen u nastavku.

Neka je zadana matrica A :

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

QR algoritmom rastavljamo matricu A na matrice QR. Matrica Q je ortogonalna matrica, a matrica R gornje trokutasta[12]. Njih matrično zapisujemo :

$$Q = \begin{bmatrix} \cos(\Theta) & \sin(\Theta) \\ -\sin(\Theta) & \cos(\Theta) \end{bmatrix}, R = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

Raspišimo sada matricu A :

$$\begin{gathered} A = QR \\ Q^{-1} \mid A = QR \\ Q^{-1}A = Q^{-1}QR \\ Q^{-1}A = IR \\ Q^{-1}A = R \end{gathered} \tag{5.1}$$

Za ortogonalne matrice vrijedi da je $Q^{-1} = Q^T$, pa konačna formula glasi :

$$Q^T A = R \tag{5.2}$$

Uvrštavanjem gore navedenih matrica u (5.2) dobivamo :

$$\begin{gathered} \begin{bmatrix} \cos(\Theta) & -\sin(\Theta) \\ \sin(\Theta) & \cos(\Theta) \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \\ \begin{bmatrix} a_{11}\cos(\Theta) - a_{21}\sin(\Theta) & a_{12}\cos(\Theta) - a_{22}\sin(\Theta) \\ a_{11}\sin(\Theta) + a_{21}\cos(\Theta) & a_{12}\sin(\Theta) + a_{22}\cos(\Theta) \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \end{gathered} \tag{5.3}$$

Matrica R je gornje trokutasta pa znamo da su sve vrijednosti ispod glavne dijagonale jednake nuli :

$$\begin{gathered} h_{21} = a_{11}\sin(\Theta) + a_{21}\cos(\Theta) = 0 \\ a_{11}\sin(\Theta) = -a_{21}\cos(\Theta) | : \cos(\Theta) \\ a_{11}\tan(\Theta) = -a_{21} \\ \tan(\Theta) = \frac{-a_{21}}{a_{11}} \end{gathered} \tag{5.4}$$

A sinuse i kosinuse kuta računamo prema formuli :

$$\sin(\Theta) = \frac{c \cdot |a_{21}|}{\sqrt{(a_{11})^2 + (a_{21})^2}},\ \cos(\Theta) = \frac{|a_{11}|}{\sqrt{(a_{11})^2 + (a_{21})^2}} \tag{5.5}$$

Gdje c ovisi o $\tan(\Theta)$, ako je $\tan(\Theta) \geq 0$ onda je $c = 1$, a ako je $\tan(\Theta) < 0$ onda je $c = -1$. Sada imamo sve potrebno da izračunamo ortogonalnu matricu Q i gornje trokutastu matricu R. Zatim računamo :

$$A_1 = R_1 Q_1 \tag{5.6}$$

Kada izračunamo A_1 završili smo s prvom iteracijom. Postupak ponavljamo sve dok je razlika vrijednosti dijagonala za A_{n+1} i A_n veća od 0.05, to ćemo najbolje objasniti na sljedećem primjeru.

Primjer 3. Neka je $f : V \to V$ linearni operator. Izračunajmo svojstvene vrijednosti koristeći QR algoritam[13] za danu matricu A :

$$A = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

Korak 1) Računamo trigonometrijske vrijednosti (5.4) (5.5) :

$$\tan(\Theta) = \frac{-1}{3}\,, \tan(\Theta) < 0 \to c = -1$$
$$\sin(\Theta) = \frac{-1 \cdot 1}{\sqrt{(3)^2 + (1)^2}} = \frac{-\sqrt{10}}{10}$$
$$\cos(\Theta) = \frac{3}{\sqrt{(3)^2 + (1)^2}} = \frac{3\sqrt{10}}{10}$$

Korak 2) Zatim računamo vrijednosti gornje trokutaste matrice (5.3) :

$$h_{11} = 3\frac{3\sqrt{10}}{10} - 1\frac{-\sqrt{10}}{10} = \frac{10\sqrt{10}}{10} = \sqrt{10}$$
$$h_{12} = \sqrt{10}$$
$$h_{21} = 0$$
$$h_{22} = \sqrt{10}$$

Korak 3) Sada smo izračunali potrebne vrijednosti da kreiramo matrice Q_1 i R_1, to su matrice koje nastaju dekompozicijom početne matrice A što možemo i računski potvrditi (5.1) :

$$A = \begin{bmatrix} \frac{3\sqrt{10}}{10} & 1\frac{-\sqrt{10}}{10} \\ 1\frac{\sqrt{10}}{10} & \frac{3\sqrt{10}}{10} \end{bmatrix} \begin{bmatrix} \sqrt{10} & \sqrt{10} \\ 0 & \sqrt{10} \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

Korak 4) Kada smo se uvjerili da je dekompozicija dobra, računamo vrijednost prve iteracije QR algoritma, odnosno vrijednost A_1 (5.6) :

$$A_1 = R_1 Q_1 = \begin{bmatrix} \sqrt{10} & \sqrt{10} \\ 0 & \sqrt{10} \end{bmatrix} \begin{bmatrix} \frac{3\sqrt{10}}{10} & 1\frac{-\sqrt{10}}{10} \\ 1\frac{\sqrt{10}}{10} & \frac{3\sqrt{10}}{10} \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix} \tag{5.7}$$

Korak 5) Završili smo s prvom iteracijom, te sada koristimo vrijednosti dobivene matrice A_1 u

2. iteraciji, gdje na kraju dobivamo (5.6) :

$$A_2 = R_2Q_2 = \begin{bmatrix} \frac{79}{17} & \frac{27}{17} \\ \frac{10}{17} & \frac{40}{17} \end{bmatrix}$$

Korak 6) Kako bi znali kada je kraj algoritmu, nakon svake iteracije oduzimamo vrijednosti po dijagonali za $|A_{n+1}$ - $A_n|$, gdje je n broj iteracija. Konkretno za prethodni korak smo dobili A_2 te sada provjeravamo da li je razlika dijagonala dovoljno mala, u ovom slučaju da li je vrijednost manja od 0.05. To je proizvoljno odabran mali broj, moguće je odabrati bilo koji dovoljno mali broj da rješenje na kraju ima smisla. Što manji broj za provjeru se odabere, to je rezultat točniji:

$$|A_{2_{11}} - A_{1_{11}}| = |\frac{79}{17} - 4| = \frac{11}{17} > 0.05$$
$$|A_{2_{22}} - A_{1_{22}}| = |\frac{40}{17} - 3| = \frac{11}{17} > 0.05$$

Korak 7) Oba uvjeta nisu zadovoljena, te se postupak nastavlja. Za ovaj konkretni zadatak kraj je na 5. iteraciji, kada dobijemo :

$$A_5 = \begin{bmatrix} 4.9819 & 1.0272 \\ 0.0456 & 2.0156 \end{bmatrix}$$

Svojstvene vrijednosti očitamo s glavne dijagonale te iznose : $\sigma(f) = \{4.9819, 2.0156\}$

5.1. QR algoritam u Pythonu

U ovom poglavlju smo implementirali QR algoritam u Pythonu. Prvo je pokušano implementirati algoritam za bilo koje matrice, ne samo simetrične. Jer to i je poanta ovog algoritma, da matrica može biti bilo kakva. No problem je nastao kada bi se random generirala matrica koja nema realne svojstvene vrijednosti. Zato smo odabrali samo simetrične matrice, jer znamo da one samo i mogu dati realne svojstvene vrijednosti. Ostavljena je mogućnost u kodu da se implementira bilo koja matrica, te ako ona ima samo realne svojstvene vrijednosti algoritam će doći do kraja. QR dekompozicija je napravljena na 2 načina, preko Givensove matrice rotacije i preko Householderovog reflektora. Samo provođenje algoritma je isto za obje dekompozicije, broj iteracija je isti, ali vrijeme potrebno da se pojedini algoritam izvede nije, te će to biti prikazano u rezultatima.

Prikaz korištenih biblioteka u programu te mogućnost mijenjanja nasumičnog generiranja :

Listing 5.1: Korištene biblioteke u QR

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import timeit
np.random.seed(2)
np.set_printoptions(2)
```

Generiranje simetrične matrice kako bi bili sigurni da uvijek kao rezultat dobijemo realne svojstvene vrijednosti :

Listing 5.2: Generiranje simetrične matrice

```
def simetricna_matrica(n):
    pocetna = np.random.randint(1, 10, size=(n,n))
    return (pocetna + pocetna.T)/2
```

Ova funkcija računa svojstvene vrijednosti s već ugrađenim algoritmom u Pythonu, a nama služi kao provjera dobivenih rezultata :

Listing 5.3: Test za provjeru svojstvenih vrijednosti

```
def numpy_test(matrica):
    eigval, eigvec = la.eig(matrica)
    print('Svojstvene vrijednosti: ', ' '.join(['{:.2f}'.format(val) for val in
        eigval]))
```

Kako bi mogli odrediti jednu matricu dekompozicije Q potrebno je prvo izračunati sinuse i kosinuse :

Listing 5.4: Sinus i kosinus za Givensovu matricu rotacije

```
def sinus_kosinus(i, j, matrica):
    sinus = 0
    kosinus = 1
    if matrica[i, j] != 0:
        sinus = -matrica[i,j]/np.sqrt(matrica[j,j]**2+matrica[i,j]**2)
        kosinus = matrica[j,j]/np.sqrt(matrica[j,j]**2+matrica[i,j]**2)
    return sinus, kosinus
```

Nakon što smo izračunali sinuse i kosinuse sada možemo generirati Givensovu matricu rotacije, odnosno matricu Q :

Listing 5.5: Generira Givensovu matricu

```
def givens(i, j, matrica):
    velicina = matrica.shape[0]
    g = np.identity(velicina)
    sinus, kosinus = sinus_kosinus(i, j, matrica)
    g[i, j] = -sinus
    g[j, i] = sinus
    g[i, i] = kosinus
    g[j, j] = kosinus
    return g.T
```

Vraća vrijednosti Q i R, potrebne za provođenje QR algoritma :

Listing 5.6: QR dekompozicija preko Givensa

```
def qr_dekompozicija_givens(matrica):
    velicina = matrica.shape[0]
    r = matrica.copy()
    q = np.identity(velicina)
```

```
    # Provrti kroz sve i,j ispod dijagonale
    for i in range(velicina):
        for j in range(i):
            # Generira Givensa koji ponistava i,j u matrici
            giv = givens(i, j, r)
            # Ponisti i,j u matrici r
            r = giv@r
            # Svaki korak dekompozicije
            # print(r)
            # print()
            # Umnozak svih Givensa koji su bili potrebni za ponistavanje i,j
            q = q@giv.T
    return q, r
```

Sama provedba QR algoritma :

Listing 5.7: Provodi QR algoritam

```
def qr_diagonalizacija_givens(matrica, tolerancija):
    nova = matrica.copy()
    zadnja = np.zeros(nova.shape)
    x = []
    i = 0
    # Radi dok je bilo koja razlika na dijagonalama nove i zadnje matrice veca od
        tolerancije
    while np.any(np.diag(nova-zadnja)>tolerancija):
        i+=1
        zadnja = nova
        q, r = qr_dekompozicija_givens(zadnja)
        nova = r@q
        # Svaka iteracija svojstvenih vrijednosti
        #print(np.diag(nova))
        x.append(np.linalg.norm(np.diag(nova-zadnja)))
    print('Broj iteracija Givens: ', i)
    return np.diag(nova), x
```

Ovo je drugačiji način na koji možemo napraviti QR dekompoziciju. Ovo je ujedno i brži način provedbe same dekompozicije, a onda i algoritma. Cilj ove provedbe je pronaći operator H koji za vektor-stupac a poništava sve elemente ispod dijagonale. Pokazat će se da je operator H upravo Householderov reflektor :

Listing 5.8: QR dekompozicija preko Householdera

```
def qr_dekompozicija_hh(matrica):
    velicina = matrica.shape[0]
    r = matrica.copy()
    q = np.identity(velicina)
    for i in range(velicina-1):
        v = r[:,i].copy()
        v[:i] = 0
        norma = np.linalg.norm(v)
        v[i] += np.sign(r[i,i])*norma
        u = 1/np.sqrt(2*norma*(norma+abs(r[i,i])))*v
```

```
        hh = np.identity(velicina)-2*np.outer(u,u)
        r = hh@r
        q = q@hh
        # Svaki korak dekompozicije
        # print(r)
        # print()
    return q, r
```

Provedba QR algoritma pozivanjem Householderove dekompozicije[14] :

Listing 5.9: Provodi QR algoritam

```
def qr_diagonalizacija_hh(matrica, tolerancija):
    nova = matrica.copy()
    zadnja = np.zeros(nova.shape)
    x = []
    i = 0
    while np.any(np.diag(nova-zadnja)>tolerancija):
        i+=1
        zadnja = nova
        q, r = qr_dekompozicija_hh(zadnja)
        nova = r@q
        #print(np.diag(nova))
        x.append(np.linalg.norm(np.diag(nova-zadnja)))
    print('Broj iteracija Householder: ', i)
    return np.diag(nova), x
```

Na kraju je mogućnost da se računaju vrijednosti za točno odabrane matrice koje ne moraju biti simetrične, no rezultati će se prikazati samo ako su sve svojstvene vrijednosti za odabranu matricu realne. Također računamo i zasebno vrijeme za provedbu svakog algoritma, te prikazujemo rezultate grafički :

Listing 5.10: Pozivanje algoritama i ostalo

```
# Za tocno zadanu matricu
fixna_matrica = np.array([[ 3, 2, 5],
                  [6, -10, -10],
                  [6, -15, 23]])

m = simetricna_matrica(4)
#print(m)
#print()
eig1, x1 = qr_diagonalizacija_givens(m, 0.001)
eig2, x2 = qr_diagonalizacija_hh(m, 0.001)
#print(eig1)
#print(eig2)
qr_dekompozicija_hh(m)

number = 1000
print(timeit.timeit(lambda: qr_dekompozicija_hh(m), number=number)/number*1000)
print(timeit.timeit(lambda: qr_dekompozicija_givens(m), number=number)/number*1000)

plt.plot(x1)
```

```
plt.plot(x2)
plt.show()
```

5.1.1. Prikaz dobivanja gornje trokutaste matrice R

Zanimljivo je prikazati na koji način prilikom provedbe iteracije se mijenjaju vrijednosti elemenata u matrici R. Nema smisla prikazivati promjene vrijednosti elemenata u matrici Q kada je ona samo pomoćna u računu. U nastavku je prikazano konkretno na zadanom primjeru kako izgleda traženje gornje trokutaste matrice R preko 2 različite metode.

Prikaz dobivanja 0 ispod glavne dijagonale. Radi se korak po korak, odnosno ćelija po ćelija :

Listing 5.11: R preko Givensa

```
[[9.   9.   6.5  5.5]
 [9.   8.   4.   3. ]
 [6.5  4.   5.   6.5]
 [5.5  3.   6.5  5. ]]

[[ 1.27e+01  1.20e+01  7.42e+00  6.01e+00]
 [-3.33e-16 -7.07e-01 -1.77e+00 -1.77e+00]
 [ 6.50e+00  4.00e+00  5.00e+00  6.50e+00]
 [ 5.50e+00  3.00e+00  6.50e+00  5.00e+00]]

[[ 1.43e+01  1.25e+01  8.89e+00  8.31e+00]
 [-3.33e-16 -7.07e-01 -1.77e+00 -1.77e+00]
 [ 0.00e+00 -1.90e+00  1.08e+00  3.06e+00]
 [ 5.50e+00  3.00e+00  6.50e+00  5.00e+00]]

[[ 1.43e+01  1.25e+01  8.89e+00  8.31e+00]
 [ 1.16e-16  2.03e+00 -3.94e-01 -2.25e+00]
 [-3.12e-16 -8.20e-17 -2.03e+00 -2.72e+00]
 [ 5.50e+00  3.00e+00  6.50e+00  5.00e+00]]

[[ 1.53e+01  1.28e+01  1.06e+01  9.55e+00]
 [ 1.16e-16  2.03e+00 -3.94e-01 -2.25e+00]
 [-3.12e-16 -8.20e-17 -2.03e+00 -2.72e+00]
 [ 6.11e-16 -1.70e+00  2.87e+00  1.68e+00]]

[[ 1.53e+01  1.28e+01  1.06e+01  9.55e+00]
 [-3.03e-16  2.65e+00 -2.15e+00 -2.80e+00]
 [-3.12e-16 -8.20e-17 -2.03e+00 -2.72e+00]
 [ 5.43e-16  1.66e-17  1.95e+00 -1.52e-01]]

[[ 1.53e+01  1.28e+01  1.06e+01  9.55e+00]
 [-3.03e-16  2.65e+00 -2.15e+00 -2.80e+00]
 [ 6.01e-16  7.06e-17  2.82e+00  1.86e+00]
 [-1.75e-16  4.49e-17  3.67e-17  1.99e+00]]
```

Prikaz dobivanja 0 ispod glavne dijagonale. S time da za razliku od Givensove dekom-

pozicije, Householderova ne radi ćeliju po ćeliju već stupac po stupac, pa je ujedno i puno brža metoda :

Listing 5.12: R preko Housenholdera

```
[[9.   9.   6.5  5.5]
 [9.   8.   4.   3. ]
 [6.5  4.   5.   6.5]
 [5.5  3.   6.5  5. ]]

[[-1.53e+01 -1.28e+01 -1.06e+01 -9.55e+00]
 [ 1.29e-15 -5.73e-02 -2.34e+00 -2.57e+00]
 [ 1.39e-15 -1.82e+00  4.21e-01  2.48e+00]
 [ 1.55e-15 -1.92e+00  2.63e+00  1.60e+00]]

[[-1.53e+01 -1.28e+01 -1.06e+01 -9.55e+00]
 [-2.11e-15  2.65e+00 -2.15e+00 -2.80e+00]
 [-8.99e-16 -4.12e-16  5.52e-01  2.32e+00]
 [-8.64e-16 -2.54e-16  2.76e+00  1.43e+00]]

[[-1.53e+01 -1.28e+01 -1.06e+01 -9.55e+00]
 [-2.11e-15  2.65e+00 -2.15e+00 -2.80e+00]
 [ 1.02e-15  3.30e-16 -2.82e+00 -1.86e+00]
 [ 7.12e-16  3.54e-16  4.60e-16 -1.99e+00]]
```

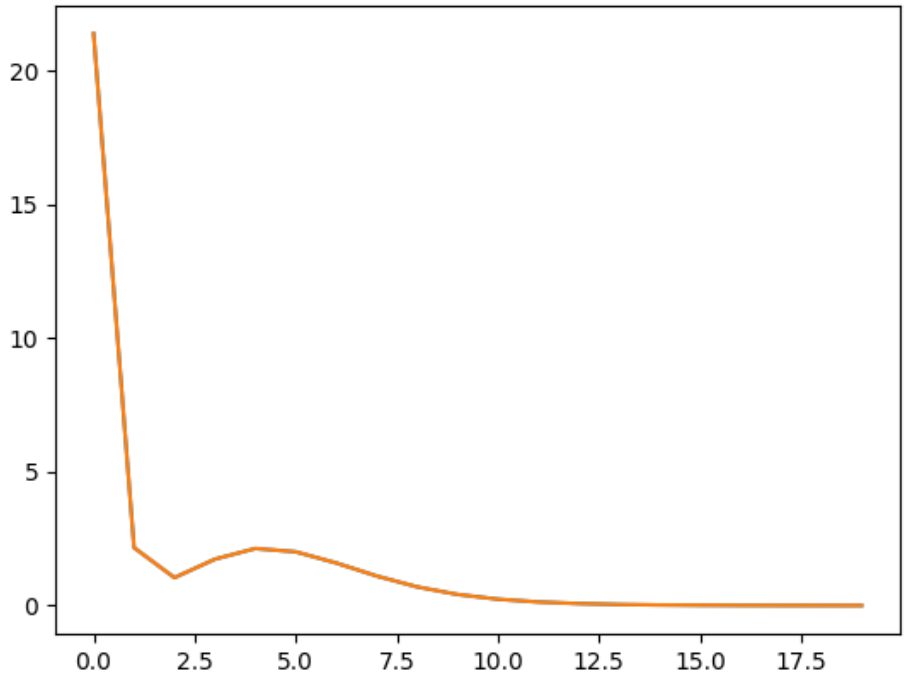
5.1.2. Rezultati QR algoritma u Pythonu

Tablično ćemo prikazati rezultate provedbe QR algoritma u Pythonu. Broj iteracija za svaku metodu je isti, ali način dobivanja matrica Q i R nije, tako da ćemo uključiti i vrijeme kao element usporedbe :

Dimenzija matrice	Givens/s	Householder/s	Broj iteracija
4x4	0.0749107999999	0.0666489000000	11
5x5	0.1066333000000	0.0912770000000	20
6x6	0.1629722000000	0.1115146999999	33
7x7	0.2339137999999	0.1326841999999	53
8x8	0.3051479999999	0.1542475000000	168
9x9	0.3985474000000	0.1754448000000	114
10x10	0.4922967999999	0.2512760999999	338
15x15	1.2885328000000	0.3633000999999	62
20x20	2.6786228000000	0.5026335000000	258

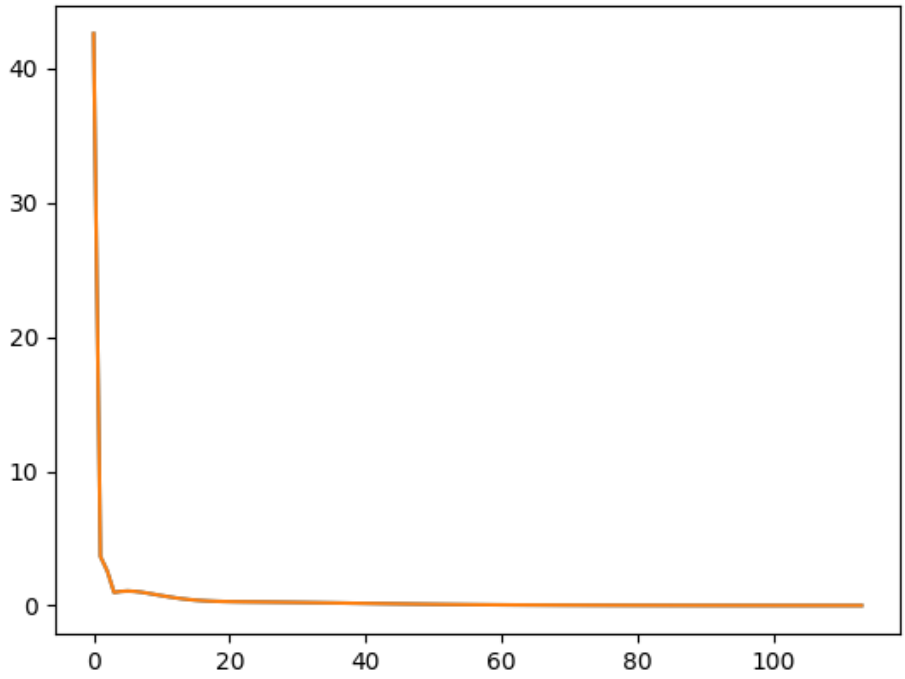
Tablica 2: Tablični prikaz broja iteracija i vremena za dobivanje svojstvenih vrijednosti preko QR

Grafički ćemo prikazati krivulju broja iteracija (os apscise) i konvergencije (os ordinate). Možemo primijetiti da u odnosu na Jacobijev algoritam, u QR algoritmu puno brže se odrede približne vrijednosti svojstvenih vrijednosti (graf brže pada) ali se i dalje provodi iteracija kako bi se dobila što preciznija vrijednost.

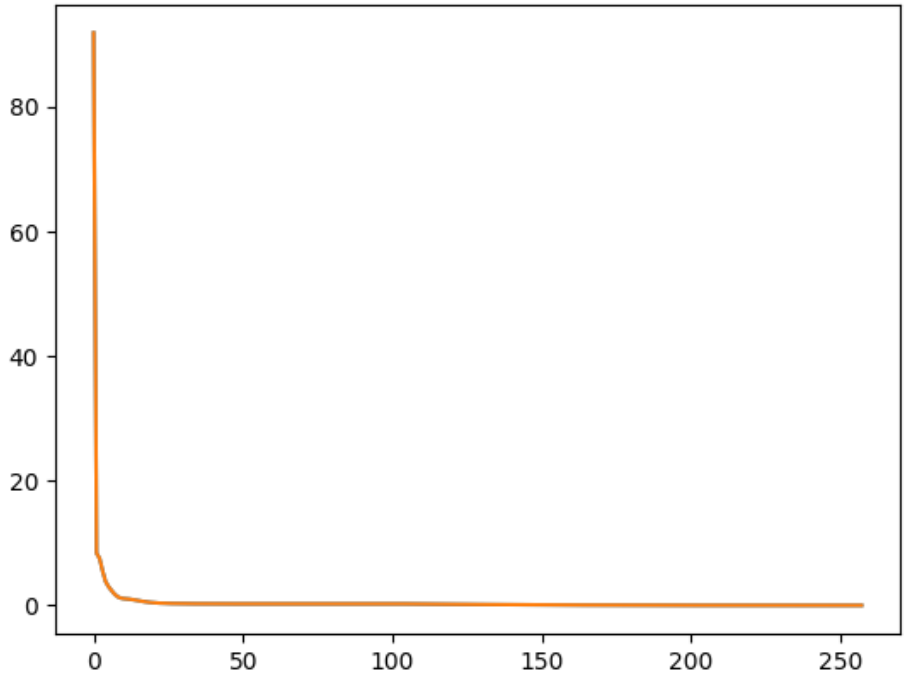


Slika 5: Broj iteracija potreban da izračunamo λ za matricu 5×5

Graf (5) prikazuje pad konvergencije za povećanje broja iteracija. Zanimljivo je i primijetiti kako provedbom QR algoritma konvergencija brže pada u odnosu na Jacobijev algoritam, ali vidimo da se za svaku matricu konvergencija naglo smanji, a zatim poveća. Zatim se opet smanjuje sve dok ukupna suma vandijagonalnih elemenata nije manja od zadane tolerancije.
Na grafu (6) vidimo kako je razlika u odnosu na graf (5) izraženija, a sama matrica nije toliko veća. Ovim algoritmom niska konvergencija se brzo postigne, nakon dosta malog broja iteriranja. Kako smo zadali jako nisku toleranciju onda je potrebno vrtjeti još dosta iteracija da bi se ta najmanja razlika dovoljno smanjila da vrijednosti na vandijagonalnim elementima postanu nula.
Graf (7) najbolje predočava kako smo već nakon vrlo malog broja iteracija došli do približnog rješenja, odnosno elementi na glavnoj dijagonali matrice su slični svojstvenim vrijednostima, ali potrebno je provesti još dosta iteracija kako bi se zadovoljila razlika konvergencije i zadane tolerancije za ovaj algoritam.



Slika 6: Broj iteracija potreban da izračunamo λ za matricu 9×9



Slika 7: Broj iteracija potreban da izračunamo λ za matricu 20×20

6. Usporedba algoritama

Dimenzija matrice	Broj iteracija Jacobi	Broj iteracija QR
4x4	21	11
5x5	42	20
6x6	69	33
7x7	85	53
8x8	108	168
9x9	136	114
10x10	176	338
15x15	412	62
20x20	772	258

Tablica 3: Tablični prikaz usporedbe broja iteracija za pojedine algoritme

7. Zaključak

U ovom radu definirali smo i proveli Jacobi i QR algoritme za računanje svojstvenih vrijednosti. Na pitanje koji je algoritam bolji, nema odgovora. Svaki algoritam ima svoje prednosti i mane. Jacobijev algoritam računa svojstvene vrijednosti samo za simetrične matrice. QR algoritam računa svojstvene vrijednosti za bilo koje $(n \times n)$ matrice. Kod računa svojstvenih vrijednosti za $(n \times n)$ matrice koje nisu simetrične velika je šansa da ćemo pogoditi matricu koja nema samo realna rješenja, nego i kompleksa. U ovom radu fokusirali smo se samo na realna rješenja, realne svojstvene vrijednosti.

Ako pogledamo tablicu (3) možemo očitati broj iteracija koji je bio potreban za pojedini algoritam kako bi odredili svojstvene vrijednosti za matrice danih $(n \times n)$ dimenzija. Valja naglasiti da te matrice nisu bile iste, već je rad napravljen na nasumičnim simetričnim matricama. Matrice nisu iste zato što nas konačno rješenje svojstvenih vrijednosti ne interesira. Ako je matrica ista, koji god algoritam provedemo dobit ćemo isto svojstveno rješenje, međutim mi uspoređujemo algoritme. A to znači da nas prije svega zanima koji je algoritam brži i točniji. Iz tablice (3) možemo očitati da je bilo potrebno provesti manje iteracija za QR algoritam u odnosu na Jacobijev. Postoje i neka odstupanja, koja su moguća ako se generirala simetrična matrica za koju se puno brže dobiju svojstvene vrijednosti. Međutim glavnu ulogu u dobivanju rješenja igra tolerancija. Kako kod provedbe algoritama bez unaprijed definirane tolerancije nikada ne bi dobili rješenje, jer vrijednost nikada ne bi bila točno nula na vandijagonalnim elementima, samo definiranje tolerancije igra jako veliku ulogu u tome koje će algoritam brže doći do kraja iteriranja.

Dekompozicija kod QR algoritma se može provesti na više načina, a u ovom radu je prikazano preko Givensove matrice rotacije i Householderovog reflektora, tablica (2). Po rezultatima možemo zaključiti da je metoda Householderovog reflektora puno brža, jer ipak eliminira više elemenata ispod glavne dijagonale odjednom. Međutim broj iteracija da se provede algoritam ostaje isti za obje dekompozicije, ili bilo koju drugu dekompoziciju koja postoji.

Na kraju možemo zaključiti da su oba algoritma za računanje svojstvenih vrijednosti pouzdani, a koje je rješenje točnije, ovisi o toleranciji koju sami definiramo, gdje je svakako cilj zadati što nižu toleranciju, ali bez super računala moguće je da konvergencija nikada ne dođe do jako niske tolerancije.

Popis literature

[1] R. K. Gupta, *Chapter 6 - Eigenvalues and Eigenvectors*. 2019., str. 283–297.

[2] I. C. London, *Linear Algebra – What are eigenvalues and eigenvectors*, https://www.youtube.com/watch?v=kwA3qM0rm7c, Na dan : 17.09.2021.

[3] J. Chasnov, *The eigenvalue problem | Lecture 32 | Matrix Algebra for Engineers*, https://www.youtube.com/watch?v=29keVZGvqME, Na dan : 17.09.2021.

[4] S. Khan, *Introduction to eigenvalues and eigenvectors*, https://www.khanacademy.org/math/linear-algebra/alternate-bases/eigen-everything/v/linear-algebra-introduction-to-eigenvalues-and-eigenvectors, Na dan : 13.09.2021.

[5] V. Krčadinac, *12.1 Linearni operatori i svojstvene vrijednosti*, https://www.youtube.com/watch?v=vUHXr-vTjyU, Na dan : 10.09.2021.

[6] prof.dr.sc. Blaženka Divjak, *Odabrana poglavlja matematike : Dio IV Linearni operatori*, str. 124–173.

[7] Z. Katedra za matematiku (FSB, *Svojstveni vektori i svojstvene vrijednosti*. 2019.

[8] Z. Drmač, V. Hari, M. Marušić, S. Singer, M. Rogina i S. Singer, *Numerička analiza : Predavanja i vježbe*. 2003., str. 42–45.

[9] I. A. Yowetu, *Givens Rotation Method*, https://www.youtube.com/watch?v=MxZy0LLEDLY, Na dan : 14.09.2021.

[10] D. V. S. Chauhan, *JACOBI'S METHOD | NUMERICAL ANALYSIS (B.Sc 3rd year)Algebraic eigen value problems|*, https://www.youtube.com/watch?v=QOouCruB2hA, Na dan : 14.09.2021.

[11] E. W. Weisstein, *"Frobenius Norm." From MathWorld–A Wolfram Web Resource*, https://mathworld.wolfram.com/FrobeniusNorm.html, Na dan : 17.09.2021.

[12] E. Mikida, *The QR Algorithm for Finding Eigenvectors*, https://www.r-bloggers.com/2017/04/qr-decomposition-with-householder-reflections/, Na dan : 15.09.2021.

[13] D. V. S. Chauhan, *QR METHOD | NUMERICAL ANALYSIS |Algebraic eigenvalue problems|*, https://www.youtube.com/watch?v=w0Dp92ZNf3U, Na dan : 15.09.2021.

[14] A. Schlegel, *QR Decomposition with Householder Reflections*, https://www.r-bloggers.com/2017/04/qr-decomposition-with-householder-reflections/, Na dan : 16.09.2021.

Popis slika

Popis tablica