8. Algorytm QR obliczania wartości własnych

Maciej Szymkat

8.1. Wprowadzenie

Na początku należy dokonać podstawowego rozróżnienia: czym innym jest rozkład QR macierzy, a czym innym metoda obliczania wartości własnych zwana algorytmem QR. W metodzie tej, w kolejnych iteracjach, konstruowane są rozkłady QR.

W ujęciu historycznym pierwsze prace, w których autorzy posługują się metodą kolejnego rzutowania wektorów, pojawiały się już w końcu XVIII i na początku XIX wieku. Prekursorami byli Carl Friedrich Gauss, Pierre Simon de Laplace i inni matematycy, u których pomysły te stosowane były do obliczania orbit ciał niebieskich, przy niedokładnych danych pomiarowych. Można powiedzieć, że w tym okresie narodziła się przyszła metoda najmniejszych kwadratów (regresja liniowa). Ówcześni matematycy stosowali algorytm zbliżony do metody Grama–Schmidta, który pod tą nazwą pojawił się dopiero pod koniec XIX i na początku XX wieku (zob. [1]). Wraz z rozwojem metod obliczeniowych stwierdzono, że metody Grama–Schmidta pod względem dokładności zawodzą w przypadku niektórych macierzy (dziś powiedzielibyśmy – źle uwarunkowanych), przy większych wymiarach w kolejnych iteracjach następuje stopniowa utrata ortogonalności generowanych wektorów. Rozwiązaniem okazały się algorytmy wykorzystujące transformacje zdefiniowane przez macierze dobrze uwarunkowane numerycznie, które zostały zaproponowane w pracach, najpierw Wallesa Givensa, a potem Alstona Scotta Householdera, w latach pięćdziesiątych XX wieku. Algorytmy te, w odróżnieniu od metod Grama–Schmidta, wymagają obliczania pierwiastków.

W zasadzie konstruowanie rozkładów QR jest dziś standardem w rozwiązywaniu prostokątnych układów równań liniowych (dla kwadratowych wykorzystuje się specjalizowane solwery LU dla różnych typów macierzy, np. dla macierzy symetrycznych, trójkątnych, pasmowych czy trójprzekątniowych). Prostokątne układy liniowe występują w zadaniach regresji liniowej. Rozkład QR jest wykorzystywany także w wariancie symetrycznym, jawnym lub niejawnym, do budowy rozkładu według wartości osobliwych SVD, o podstawowym znaczeniu w matematyce stosowanej. Szczególnie ważne jest zastosowanie dekompozycji QR do rozwiązywania problemu znajdowania wartości i wektorów własnych w tzw. algorytmie QR.

Wprowadzenie funkcji własnych (wektorów) przypisuje się Cauchy'emu, który w swoich pracach z pierwszej połowy XIX wieku do opisania mechaniki ciał niebieskich zastosował rozwiązanie liniowych równań różniczkowych. Sam termin „Eigenwerte" pochodzi z pracy Davida Hilberta z początku XX wieku [2]. Zadanie wyznaczania wartości i wektorów własnych jest jednym z najbardziej podstawowych problemów liniowej algebry numerycznej [3]. Na początku lat sześćdziesiątych zaproponowany został algorytm QR niezależnie przez Johna Francisa i Wierę Kubłanowską. Francis zauważył, że iteracja QR zachowuje postać macierzy Hessenberga. Dodanie tzw. przesunięć Wilkinsona i podwójnych przesunięć Francisa, korzystających z niejawnego wyliczania macierzy ortogonalnej rozkładu spowodowało istotny wzrost efektywności algorytmu [4]. Rząd zbieżności algorytmu QR dla macierzy symetrycznych jest sześcienny, a dla niesymetrycznych kwadratowy. Więcej informacji na temat rozkładu QR i algorytmu QR można znaleźć w podstawowych pozycjach [5–7].

Algorytm QR został uznany za jeden z dziesięciu najważniejszych algorytmów powstałych w XX wieku [8]. Oprócz zalet samego algorytmu ważne są szerokie zastosowania wartości i wektorów własnych. Wikipedia wylicza następujące: przekształcenia geometryczne, równania Schrödingera, zjawiska rozchodzenia się fal, modelowanie orbitali cząsteczkowych, analiza składowych głównych (PCA) w statystyce, analiza drgań mechanicznych, analiza obrazów, geologia i glacjologia, tensory naprężeń czy epidemiologia (współczynnik R_0). Do niewymienionych zalet należałoby jeszcze dodać zastosowania w układach sterowania, gdzie za pomocą własności wartości własnych określa się własności dynamiczne układów liniowych. Są to dość chaotycznie wybrane przykłady, ale dają pojęcie o wielkości zakresu zastosowań. Właściwie wszędzie, gdzie coś jest charakteryzowane jako widmowe, w gruncie rzeczy chodzi o wartości i wektory własne. W dziedzinie numerycznej algebry liniowej sporą karierę zrobiły też wartości osobliwe macierzy, ale w końcu ich kwadraty są właśnie wartościami własnymi macierzy pomnożonej przez swoją transpozycję. W interesującym wpisie z 2016 roku Higham ze współautorami [9] przeanalizowali ważność poszczególnych algorytmów pod względem liczby odwołań w ponad 1000-stronnicowym przewodniku *Princeton Companion to Applied Mathematics* [10]. Algorytm QR znalazł się na drugim i trzecim miejscu: jako algorytm dekompozycji macierzy razem z rozkładami LU i Choleskiego oraz w metodach obliczania wartości osobliwych razem z algorytmem QZ. Pierwsze miejsce zajęła metoda Newtona i metody quasi-Newtonowskie.

Kiedy algorytm QR może zawieść i co wtedy? Dzieje się tak, jeśli problem obliczania wartości własnych jest źle uwarunkowany numerycznie. Rozwiązaniem jest wzrost dokładności obliczeń. Dla środowisk MATLAB-a i Pythona dostępne są specjalizowane biblioteki, o których więcej informacji podano w poniższym akapicie.

W niniejszym rozdziale postaramy się odpowiedzieć na pytanie, czy możliwe jest zbudowanie prostego, ale na tyle wiernego modelu algorytmu QR, aby w konkretnych eksperymentach numerycznych można było zilustrować jego własności. Będziemy analizować: czas obliczeń, dokładność oraz zachowanie różnych wariantów (liczbę iteracji i ich przebieg). Wybrane

zostało środowisko programowe MATLAB-a, gdyż wydawało się dogodne do prowadzenia testów numerycznych, a język wysokiego poziomu pozwalał zapisać kod w sposób zwarty i czytelny. Procedury numeryczne algebry liniowej wbudowane w MATLAB-ie początkowo oparte były na bibliotekach LINPACK i EISPACK, od 2000 roku zostały zastąpione przez pakiet funkcji LAPACK ze wsparciem dla procedur wektorowych i macierzowych biblioteki BLAS. Alternatywą mógłby być język Python uruchomiany np. w środowisku Amazon SageMaker (zwłaszcza w kontekście uczenia maszynowego i PCA), z pakietami NumPy lub SciPy. Biblioteka scipy.linalg zawsze kompilowana jest ze wsparciem dla procedur BLAS/LAPACK, w przypadku numpy.linalg opcjonalnie.

Przyjmujemy ogólne założenie, że ograniczamy się do przypadku macierzy rzeczywistych, niesymetrycznych, pełnych, o różnych wartościach własnych. Eksperymenty numeryczne były prowadzone na standardowym komputerze z systemem operacyjnym Windows 10 i środowiskiem MATLAB-a w wersji R2020b. Komputer wyposażony był w procesor Intel Core i7 8550 (o parametrach: cztery rdzenie, osiem wątków, 1,80 GHz). Będziemy zajmowali się tylko problemem wyznaczania wartości własnych. Jeśli zależałoby nam na wyliczeniu wektorów własnych, moglibyśmy skorzystać z tego, że algorytmy QR w trakcie sprowadzania macierzy do postaci trójkątnej, z ewentualnymi blokami 2×2 na przekątnej, mogą akumulować informacje o macierzy ortogonalnej rozkładu QR, z której można odtworzyć wektory własne. Można by też, po wyznaczeniu pewnej wartości własnej λ macierzy A, wyznaczyć bazę przestrzeni zerowej macierzy $A - \lambda I$, np. w MATLAB-ie za pomocą funkcji `null`. Dokładność wyznaczenia wektorów własnych może poprawić tzw. wyważenie macierzy, to jest sprowadzenie jej, przez odpowiednią permutację wierszy i kolumn oraz transformację podobieństwa z diagonalną macierzą przejścia, do macierzy o możliwie najbardziej wyrównanych normach wierszy i kolumn, zob. https://www.mathworks.com/help/matlab/ref/balance.html i przedstawiony tam przykład.

8.2. Wektory i wartości własne

Niech $A \in \mathbb{C}^{n\times n}$. Z definicji liczba $\lambda \in \mathbb{C}$ jest wartością własną macierzy A, a niezerowy wektor $x \in \mathbb{C}^n$ skojarzonym z nią wektorem własnym, jeśli

$$Ax = \lambda x,$$

innymi słowy, jeśli równanie $(A - \lambda I)x = 0$ ma rozwiązanie niezerowe.

Warunkiem koniecznym i wystarczającym jest, aby

$$\det(A - \lambda I) = 0.$$

Inaczej mówiąc, liczby λ są pierwiastkami wielomianu charakterystycznego macierzy A. Widmo macierzy to zbiór jej wartości własnych. Podstawową własnością wartości własnych

i wektorów własnych jest liniowa niezależność wektorów własnych odpowiadających różnym wartościom własnym.

Załóżmy, że wszystkie wartości własne są różne. Niech wektory własne $\{x_1, x_2, \ldots, x_k\}$ tworzą maksymalny zbiór wektorów liniowo niezależnych. Przypuśćmy, że $k < n$. Wtedy

$$x_{k+1} = \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_k x_k,$$

z czego wynika

$$Ax_{k+1} = \alpha_1 Ax_1 + \alpha_2 Ax_2 + \cdots + \alpha_k Ax_k,$$

a więc

$$\lambda_{k+1} x_{k+1} = \alpha_1 \lambda_1 x_1 + \alpha_2 \lambda_2 x_2 + \cdots + \alpha_k \lambda_k x_k$$

$$\lambda_{k+1}(\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_k x_k) = \alpha_1 \lambda_1 x_1 + \alpha_2 \lambda_2 x_2 + \cdots + \alpha_k \lambda_k x_k$$

$$0 = \alpha_1(\lambda_1 - \lambda_{k+1})x_1 + \alpha_2(\lambda_2 - \lambda_{k+1})x_2 + \cdots + \alpha_k(\lambda_k - \lambda_{k+1})x_k,$$

co jest sprzeczne z założeniem o liniowej niezależności $\{x_1, x_2, \ldots, x_k\}$.

W ogólnym przypadku każda macierz $n \times n$, która ma n liniowo niezależnych wektorów własnych, jest podobna do macierzy diagonalnej z wartościami własnymi na przekątnej. Macierz przejścia realizująca to podobieństwo jest macierzą zbudowaną z wektorów własnych. Nie musi być ona ortogonalna (macierz odwrotna równa jest macierzy transponowanej do niej $A^{-1} = A^\top$) czy unitarna w przypadku zespolonym (macierz odwrotna jest równa macierzy hermitowsko sprzężonej do niej $A^{-1} = A^H$). Warunkiem koniecznym i wystarczającym ortogonalności macierzy przejścia jest normalność macierzy A ($A^\top A = AA^\top$, w przypadku zespolonym $A^H A = AA^H$).

Macierze, które są podobne do macierzy diagonalnych, nazywane są diagonalizowalnymi. Dla każdej macierzy można natomiast skonstruować rozkład Schura

$$A = U^\dagger T U,$$

gdzie U jest macierzą unitarną ($UU^\dagger = U^\dagger U = I, U^\dagger$ oznacza macierz zespoloną, hermitowsko sprzężoną do macierzy U, czyli sprzężoną i transponowaną), a T macierzą trójkątną górną.

Warunkiem koniecznym i wystarczającym zerowania się elementów pozadiagonalnych w T jest normalność macierzy A ($AA^\dagger = A^\dagger A$). W przypadku rzeczywistym rozkład Schura można zapisać w postaci

$$A = Q^\top T Q,$$

gdzie Q jest macierzą ortogonalną ($QQ^\top = Q^\top Q = I$), a T macierzą blokowotrójkątną górną.

Na przekątnej macierzy T występują bloki 1×1, odpowiadające rzeczywistym wartościom własnym, lub bloki 2×2 postaci $\begin{bmatrix} a & b \\ -b & a \end{bmatrix}$, odpowiadające parom sprzężonym wartości własnych ($a + bi$ i $a - bi$). Wartości własne macierzy ortogonalnych leżą na okręgu jednostkowym na płaszczyźnie zespolonej.

Macierze A i B są podobne ($A \sim B$), jeśli istnieje macierz nieosobliwa P taka, że $B = P^{-1}AP$. Macierze podobne mają te same wartości własne. Dla danego odwzorowania liniowego $x \to Ax$ wartości własne są te same, jedynie jego reprezentacja macierzowa i wektory własne zależą od wyboru bazy. Macierze reprezentujące to samo odwzorowanie liniowe są podobne. Macierz P reprezentująca to podobieństwo jest macierzą przejścia od jednej bazy do drugiej.

Mówimy, że macierz jest macierzą Hessenberga, jeżeli jej wszystkie elementy pod pierwszą podprzekątną zerują się

$$H = [h_{ij}] = \begin{bmatrix} * & * & * & \cdots & * & * \\ * & * & * & \ddots & * & * \\ 0 & * & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & * & * \\ 0 & 0 & \ddots & * & * & * \\ 0 & 0 & \cdots & 0 & * & * \end{bmatrix},$$

czyli $h_{ij} = 0$ dla $j \leq i-2$.

Dowolną macierz można sprowadzić do postaci macierzy Hessenberga z zachowaniem relacji podobieństwa, np. za pomocą przekształceń Householdera. Niech $w \in \mathbb{R}^n$ spełnia $\langle w, w \rangle = w^\top w = \|w\|^2 = 1$. Zdefiniujmy macierz Householdera

$$P = I - 2ww^\top.$$

Macierz ta jest symetryczna ($P^\top = I^\top - 2\left(ww^\top\right)^\top = I - \left(w^\top\right)^\top w^\top = I - 2ww^\top = P$), unitarna ($PP^\top = PP = P^\top P$), zachodzi ciąg równości

$$PP = \left(I - 2ww^\top\right)\left(I - 2ww^\top\right) = I - 2ww^\top - 2ww^\top + 4ww^\top ww^\top$$

$$= I - 4ww^\top + 4ww^\top = I.$$

Transformacja Householdera $x \to Px$ zachowuje normę wektora

$$\|Px\|^2 = x^\top P^\top Px = x^\top x = \|x\|^2.$$

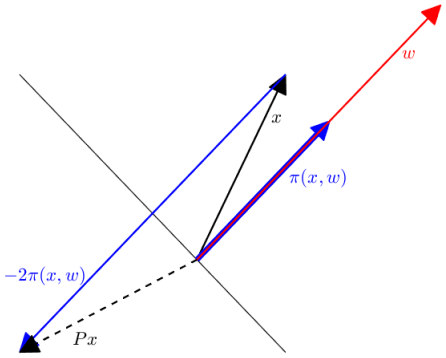
Dla $a, b \in \mathbb{R}^{n\times 1}$ wprowadźmy następującą notację. Definiujemy iloczyn skalarny $\langle a, b \rangle = a^\top b$. W przypadku zespolonym $\langle a, b \rangle = a^H b$, gdzie $a^H = \overline{a}^\top$. Zachodzi $\langle a, b \rangle = \|a\| \, \|b\| \cos\varphi_{a,b}$, gdzie $\varphi_{a,b}$ jest kątem pomiędzy wektorami a i b. Wektory są ortogonalne, jeśli $\varphi_{a,b} = \frac{\pi}{2}$, wtedy też $\langle a, b \rangle = 0$. Rzut wektora b na kierunek wektora a zapiszemy jako

$$\pi(b, a) = \|b\| \cos\varphi_{a,b} \frac{a}{\|a\|} = \frac{\langle a, b \rangle}{\|a\|} \frac{a}{\|a\|} = \frac{\langle a, b \rangle}{\|a\|^2} a \, .$$

Obraz przekształcenia Householdera możemy zapisać w postaci

$$Px = x - 2ww^\top x = x - 2w \langle w, x \rangle = x - 2 \langle w, x \rangle w = x - 2\pi(x, w).$$

Stąd prosta interpretacja geometryczna tego przekształcenia. Wynikiem jego działania jest wektor leżący po przeciwnej stronie hiperpłaszczyzny ortogonalnej do wektora w, na prostej równoległej do wektora w przechodzącej przez koniec wektora x (rys. 8.1). Przekształcenie Householdera nazywane jest też odbiciem Householdera (*Householder reflection*).



Rys. 8.1. Odbicie Householdera na płaszczyźnie

Niech $x \in \mathbb{R}^n$, $e_1 = [1,0,\ldots,0] \in \mathbb{R}^n$, jako w weźmy

$$w = \frac{x - \|x\| e_1}{\sqrt{(x - \|x\| e_1)^\top (x - \|x\| e_1)}}.$$

Łatwo sprawdzić, że

$$Px = \|x\| e_1.$$

Niech dana będzie macierz $A_n = [a_{ik}] \in \mathbb{R}^{n\times n}$. Zapiszmy ją w postaci blokowej

$$A_n = \begin{bmatrix} a_{11} & v \\ u & \tilde{A}_{n-1} \end{bmatrix},$$

gdzie u i v są odpowiednio wektorami kolumnowym i wierszowym. Niech P_{n-1} będzie macierzą realizującą przekształcenie Householdera, dobraną do wektora u tak, żeby $P_{n-1}u = \|u\| e_1$.

Rozważmy następujące podobieństwo unitarne

$$\begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{0} & P_{n-1} \end{bmatrix} A_n \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{0} & P_{n-1}^\top \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{0} & P_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & v \\ u & \tilde{A}_{n-1} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0}^\top \\ \mathbf{0} & P_{n-1}^\top \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} & vP_{n-1}^\top \\ \|u\| u_1 & \\ 0 & \\ 0 & P_{n-1}\tilde{A}_{n-1}P_{n-1}^\top \\ \vdots & \\ 0 & \end{bmatrix}.$$

Postępując analogicznie, do $A_{n-1} = P_{n-1}\tilde{A}_{n-1}P_{n-1}^{\top}$ dobieramy P_{n-2} itd. W rezultacie po $n-2$ krokach otrzymujemy macierz Hessenberga podobną do macierzy A_n.

Poniżej prosta implementacja w MATLAB-ie przekształcenia do postaci Hessenberga za pomocą transformacji Householdera.

```
function A=hhess(A)
n=size(A,1);
for k=1:n-2
  v=A(k+1:n,k);
  alpha=-norm(v);
  if v(1)<0, alpha=-alpha; end
  v(1)=v(1)-alpha;
  v=v/norm(v);
  A(k+1:n,k+1:n)=A(k+1:n,k+1:n)-2*v*(v.'*A(k+1:n,k+1:n));
  A(k+1,k)=alpha;
  A(k+2:n,k)=0;
  A(1:n,k+1:n)=A(1:n,k+1:n)-2*(A(1:n,k+1:n)*v)*v.';
end
end
```

8.3. Konstrukcja rozkładu QR

Macierze $Q, R \in \mathbb{R}^{n\times n}$ tworzą rozkład ortogonalnotrójkątny (QR) macierzy $A \in \mathbb{R}^{n\times n}$ wtedy, gdy

$$A = QR,$$

gdzie Q jest macierzą ortogonalną, a R macierzą trójkątną górną.

W tej części zajmiemy się algorytmami wyznaczania macierzy Q i R dla zadanej macierzy A.

Rzut b na podprzestrzeń ortogonalną do wektora a wynosi

$$\pi^{\perp}(b,a) = b - \pi(b,a) = b - \frac{\langle a,b\rangle}{\|a\|^2}a.$$

Przy założeniu, że wektor a jest znormalizowany $\left(\|a\| = \|a\|^2 = \langle a,a\rangle = 1\right)$,

$$\pi^{\perp}(b,a) = b - \langle a,b\rangle\ a.$$

Zachodzi oczywiście

$$\left\langle a, \pi^{\perp}(b,a)\right\rangle = \left\langle a, b - \frac{\langle a,b\rangle}{\|a\|^2}a\right\rangle = \langle a,b\rangle - \langle a,b\rangle\frac{\langle a,a\rangle}{\|a\|^2} = 0.$$

Niech $c \in \mathbb{R}^{n\times 1}$. Zakładamy, że wektor $\pi^{\perp}(b,a)$ jest znormalizowany. Definiujemy

$$\pi^{\perp}(c,b,a) = \pi^{\perp}\left(\pi^{\perp}(c,a),\ \pi^{\perp}(b,a)\right)$$

$$\pi^{\perp}(c,b,a) = \pi^{\perp}(c,a) - \left\langle \pi^{\perp}(b,a),\, \pi^{\perp}(c,a) \right\rangle \pi^{\perp}(b,a)$$

$$= c - \langle a,c\rangle a - \left\langle \pi^{\perp}(b,a),\, \pi^{\perp}(c,a) \right\rangle (b - \langle a,b\rangle a)$$

$$= c - \pi(c,a) - \left\langle \pi^{\perp}(b,a), c - \langle a,c\rangle a \right\rangle \pi^{\perp}(b,a)$$

$$= c - \pi(c,a) - \pi\left(c, \pi^{\perp}(b,a)\right).$$

Pokażemy, że wektor $\pi^{\perp}(c,b,a)$ jest ortogonalny do wektorów a i $\pi^{\perp}(b,a)$:

$$\left\langle a, \pi^{\perp}(c,b,a) \right\rangle = \left\langle a, c - \langle a,c\rangle a - \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,a) \right\rangle (b - \langle a,b\rangle a) \right\rangle$$

$$= \langle a,c\rangle - \langle a,c\rangle \ \|a\|^2 - \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,a) \right\rangle \left(\langle a,b\rangle - \langle a,b\rangle \ \|a\|^2 \right) = 0$$

$$\left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,b,a) \right\rangle = \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,a) - \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,a) \right\rangle \pi^{\perp}(b,a) \right\rangle$$

$$= \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,a) \right\rangle - \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,a) \right\rangle \left\| \pi^{\perp}(b,a) \right\|^2 = 0$$

Z warunku $\left\langle a, \pi^{\perp}(c,b,a) \right\rangle = 0$ i obserwacji, że

$$0 = \left\langle \pi^{\perp}(b,a), \pi^{\perp}(c,b,a) \right\rangle = \left\langle b - \langle a,b\rangle \ a, \pi^{\perp}(c,b,a) \right\rangle = \left\langle b, \pi^{\perp}(c,b,a) \right\rangle,$$

wynika, że skonstruowany wektor $\pi^{\perp}(c,b,a)$ jest jednocześnie ortogonalny do wektora a oraz do wektora b.

Niech A będzie macierzą zbudowaną z wektorów kolumnowych $a,\ b,\ c,\ d,\ e,\ \ldots$. W konstrukcji rozkładu ortogonalnotrójkątnego QR metodą Grama–Schmidta analogicznie obliczamy i normalizujemy kolejne rzuty dla dalszych kolumn A

$$\pi^{\perp}(d,c,b,a) = \pi^{\perp}\left(\pi^{\perp}(d,b,a),\ \pi^{\perp}(c,b,a)\right)$$

$$= d - \pi(d,a) - \pi\left(d, \pi^{\perp}(b,a)\right) - \pi\left(d, \pi^{\perp}(c,b,a)\right)$$

$$\pi^{\perp}(e,d,c,b,a) = \pi^{\perp}\left(\pi^{\perp}(e,c,b,a),\ \pi^{\perp}(d,c,b,a)\right)$$

$$= e - \pi(e,a) - \pi\left(e, \pi^{\perp}(b,a)\right) - \pi\left(e, \pi^{\perp}(c,b,a)\right) - \pi\left(e, \pi^{\perp}(d,c,b,a)\right) \text{ itd.}$$

Przeskakujemy rzuty zerowe, których ewentualne pojawienie się oznacza spadek rzędu macierzy.

Algorytm Grama–Schmidta (wersja klasyczna) można zapisać następująco w MATLAB-ie (funkcja `proj` odpowiada funkcji $\pi^{\perp}$, funkcja `dot` jest standardową implementacją iloczynu skalarnego).

```
function [Q,R] = cgs(A)
  n=size(A,1); Q=A;
  for k=1:n-1
    Q(:,k)=Q(:,k)/norm(Q(:,k));
    for i=k+1:n, Q(:,i)=proj(Q(:,i),Q(:,k)); end
  end
  Q(:,n)=Q(:,n)/norm(Q(:,n));
  R=Q'*A;
  function pv=proj(v1,v2)
    pv=v1-dot(v1,v2)*v2;
  end
end
```

Lepsze własności numeryczne ma tzw. wersja zmodyfikowana:

```
function [Q,R]=mgs(A)
  n=size(A,1); Q=A; R=zeros(n);
  for k=1:n
    R(k,k)=norm(Q(:,k));
    Q(:,k)=Q(:,k)/R(k,k);
    R(k,k+1:n)=Q(:,k)'*Q(:,k+1:n);
    Q(:,k+1:n)=Q(:,k+1:n)-Q(:,k)*R(k,k+1:n);
  end
end
```

Odmienny sposób obliczenia rozkładu QR polega na wykorzystaniu przekształceń Householdera do wprowadzania kolejnych zer pod przekątną, podobnie jak w metodzie eliminacji Gaussa. Odpowiada to mnożeniu od prawej strony kolejnych podmacierzy przez macierze eliminacyjne, które w tym przypadku są ortogonalne. Ich iloczyn jest szukaną macierzą Q. Poniżej prosta implementacja tej metody w MATLAB-ie.

```
function [Q,R] = hqr(A)
[~,n]=size(A); Q=eye(n); R=A;
sig=@(u) sign(u)+(u==0);
for k=1:n
  v=R(k:n,k);
  v(1)=v(1)+sig(v(1))*norm(v);
  s=norm(v);
  if s~=0
    v=v/s;
    R(k:n,k:n)=R(k:n,k:n)-2*v*v'*R(k:n,k:n);
    Q(1:n,k:n)=Q(1:n,k:n)-2*Q(1:n,k:n)*v*v';
  end
end
```

Eksperyment 8.1

W tym i w kolejnym eksperymencie zastosowano następujące podejście do dokładności metody wyznaczania rozkładów QR, a później także wartości własnych. Ponieważ w punkcie wejściowym mamy macierz, której wartości własnych nie znamy, nie możemy ocenić wprost, czy wartości wyliczone odbiegają od dokładnych. Oczywiście można by zastosować inną dokładniejszą metodę ich wyliczenia i porównać wyniki. Efektywność takiego postępowania zostanie omówiona w podrozdziale 8.5. W tym miejscu zastosujemy prostsze rozwiązanie.

Proces przygotowania serii macierzy testowych rozpoczniemy od utworzenia ciągów wartości własnych (rzeczywistych i par sprzężonych) za pomocą generatora liczb pseudolosowych. Ciągi te zapamiętujemy i umieszczamy na przekątnych macierzy wyjściowych, które następnie zostaną przekształcone za pomocą transformacji podobieństwa z wykorzystaniem losowych, ortogonalnych macierzy przejścia. Uzyskujemy wtedy macierze pełne. Losujemy macierze rzeczywiste o zadanej lub losowej liczbie par sprzężonych wartości własnych. Moduły wartości własnych wyliczamy jako liczby $e^{\kappa m}$, gdzie m wyznaczane jest z rozkładu normalnego, a κ jest parametrem. Parametr ten ma silny wpływ na uwarunkowanie numeryczne generowanych macierzy. Argumenty wyliczane są z rozkładu równomiernego z przedziałów $(-\pi, 0)$ i $(0,\ \pi)$. Funkcja zwraca rzeczywistą macierz blokowodiagonalną i wektor wartości własnych uporządkowany według wartości części rzeczywistych i części urojonych, przy równych wartościach części rzeczywistych (funkcja pomocnicza `eig_sort`).

W celu zapewnienia powtarzalności generowanych serii macierzy trzeba inicjować standardowy generator liczb pseudolosowych (`rng`) za pomocą zapamiętanego wcześniej zestawu parametrów startowych. Poniższy kod jest implementacją opisanej techniki generacji macierzy testowych.

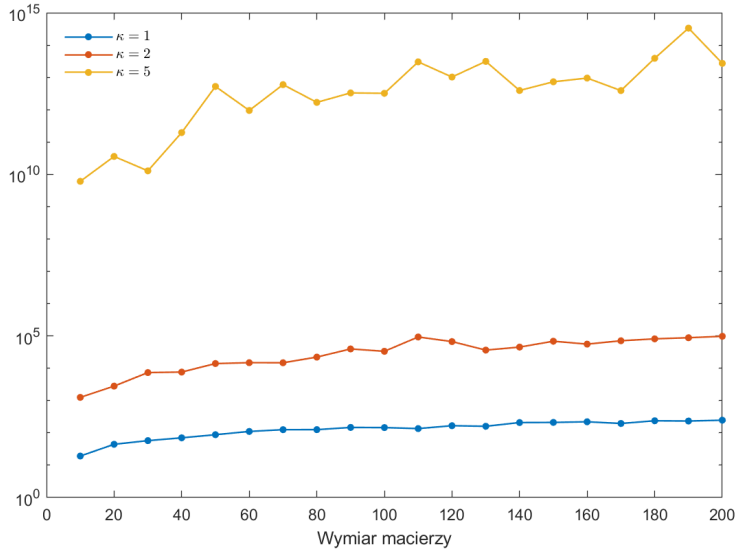
```
function [A,lambda]=gen_mat(n,kappa,n_cp)
sig=@() 2*randi(2)-3;
A0=zeros(n,n);

if nargin<3, n_cp=randi(n/2); end
if nargin<2, kappa=1; end
n_r=n-2*n_cp;
for i=1:n_r, A0(i,i)=rnd1; end
for i=n_r+1:2:n
  [M,phi]=rnd2; re=real(M*exp(phi*1i)); im=imag(M*exp(phi*1i));
  A0(i,i)=re; A0(i+1,i+1)=A0(i,i); A0(i,i+1)=im; A0(i+1,i)=-A0(i,i+1);
end

lambda=eig_sort(eig_diag(A0));
P=randn(n,n); [P,~]=qr(P); A=P*A0*inv(P);

function Ms=rnd1(), Ms=sig()*(exp(randn()))^kappa; end

function [M,phi]=rnd2(), M=(exp(randn()))^kappa; phi=sig()*pi*rand(); end
end
```

Rozpoczniemy od zbadania wskaźników uwarunkowania macierzy dla serii 20 tysięcy macierzy o wymiarach od 10×10 do 200×200, przy wartościach parametru $\kappa = 1, 2, 5$. Za pomocą standardowej funkcji `cond` wyliczany jest wskaźnik uwarunkowania macierzy A równy $\|A\| \left\|A^{-1}\right\|$. Charakteryzuje on wrażliwość odwrotności macierzy ze względu na zaburzenia wartości jej elementów.

Posłużymy się następującym skryptem:

```
Nmax=200; imax=Nmax/10; Nruns=100; mcn=zeros(imax,4); kappa=[1 2 5];
load rng_set_qr; rng(s);
for i=1:imax
  for ii=1:Nruns
    for k=1:numel(kappa)
      A=gen_mat(10*i,kappa(k));  mcn(i,k)=mcn(i,k)+cond(A);
    end
  end
end
```

Wyniki przedstawia rysunek 8.2.



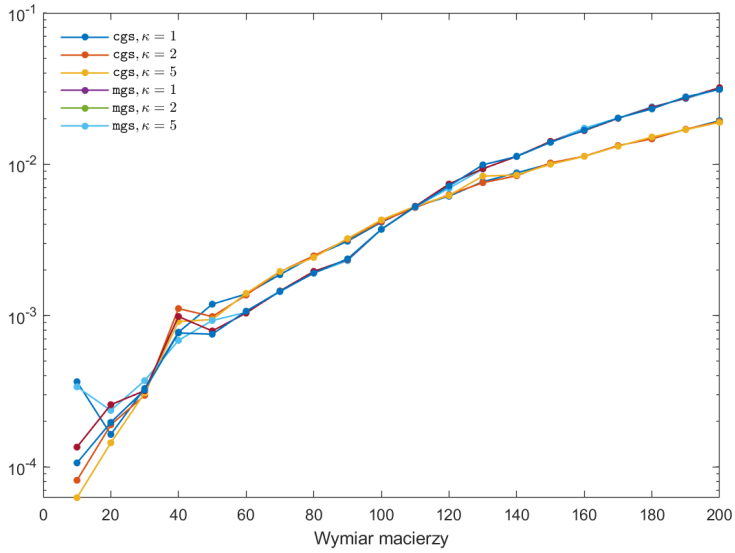
Rys. 8.2. Średni wskaźnik uwarunkowania macierzy

Najpierw porównamy wyniki dwu implementacji metody Grama–Schmidta: `cgs` i `mgs`, dla tej samej serii macierzy. Jakość obliczeń ocenimy na podstawie wielkości błędu ortogonalności, wyrażonego jako średnia maksymalna wartość bezwzględna elementów macierzy $Q\,Q^\top - I$, oraz błędu rozkładu, wyrażonego jako średnia stosunku maksymalnej wartości bezwzględnej elementów macierzy $QR - A$ do maksymalnej wartości bezwzględnej elementów macierzy A.

W środowisku MATLAB-a został uruchomimy następujący skrypt.

```
Nmax=200; imax=Nmax/10;
et=zeros(imax,2,5); qr_err=zeros(imax,2,3); orth_err=zeros(imax,2,3);
fun={@cgs,@mgs}; Nruns=100; kappa=[1 2 5];
load rng_set_qr; rng(s);
for i=1:imax
  n=10*i;
  for k=1:3
    for ii=1:Nruns
      A=gen_mat(10*i,kappa(k));
      for kk=1:2
        tic; [Q,R]=feval(fun{kk},A); et(i,kk,k)=et(i,kk,k)+toc;
        qr_err(i,kk,k)=qr_err(i,kk,k)+norm(Q*R-A,inf)/norm(A,inf);
        orth_err(i,kk,k)=orth_err(i,kk,k)+norm(Q*Q'-eye(n),inf);
      end
    end
  end
end
```

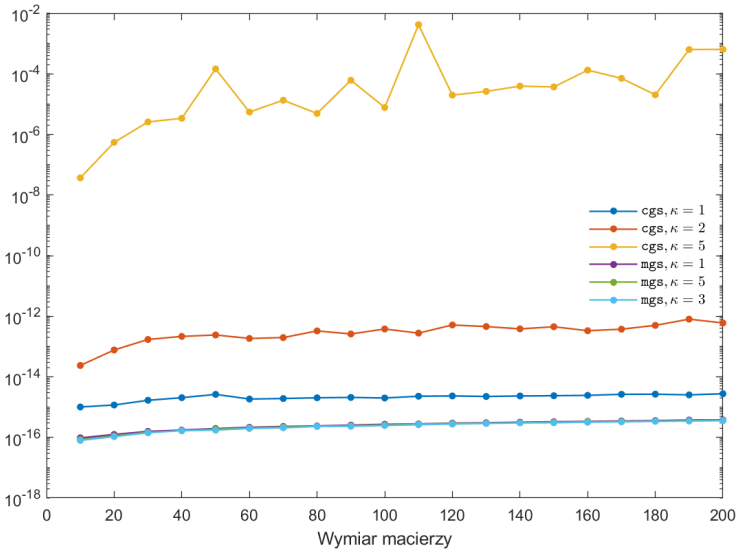
Wyniki przedstawiono na rysunkach 8.3–8.5.



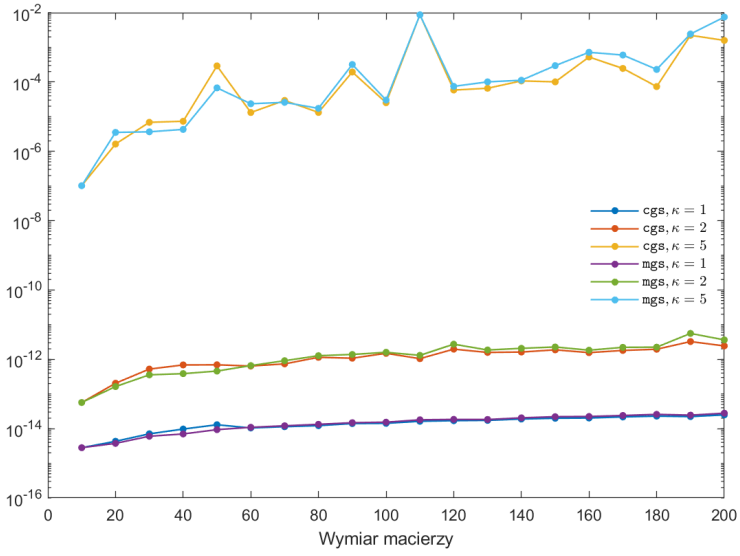
Rys. 8.3. Średni czas wykonania w sekundach

Jak widać, oba warianty mają porównywalne czasy obliczeń. Wariant `cgs` ma wyższy błąd rozkładu, zwłaszcza przy gorzej uwarunkowanych macierzach. Błąd rozkładu dla wariantu `mgs`, ze względu na sposób przekształcania macierzy w kolejnych etapach metody, jest bardzo niski. Jeśli chodzi o błąd ortogonalności, to obie metody osiągają podobne wyniki, bardzo słabe przy macierzach gorzej uwarunkowanych i fakt ten właściwie dyskwalifikuje

te metody we współczesnych zastosowaniach praktycznych. Są one ważne historycznie i ich znajomość może być istotna dla zrozumienia natury problemu.



Rys. 8.4. Średni błąd rozkładu

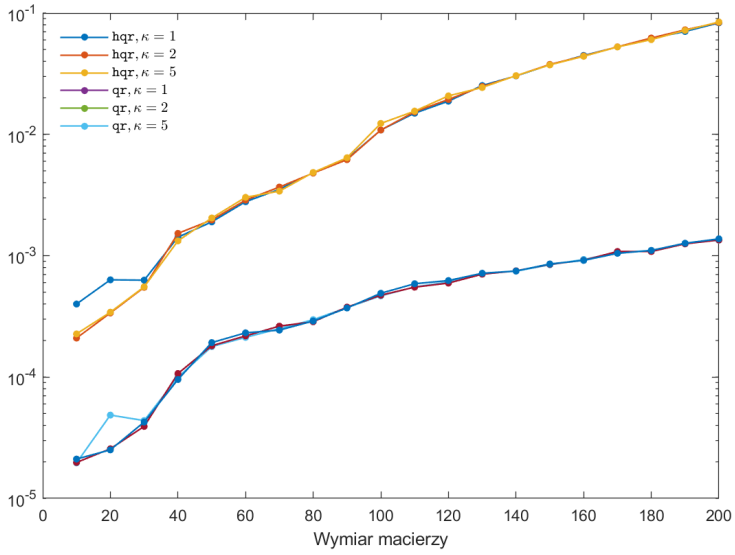


Rys. 8.5. Średni błąd ortogonalności

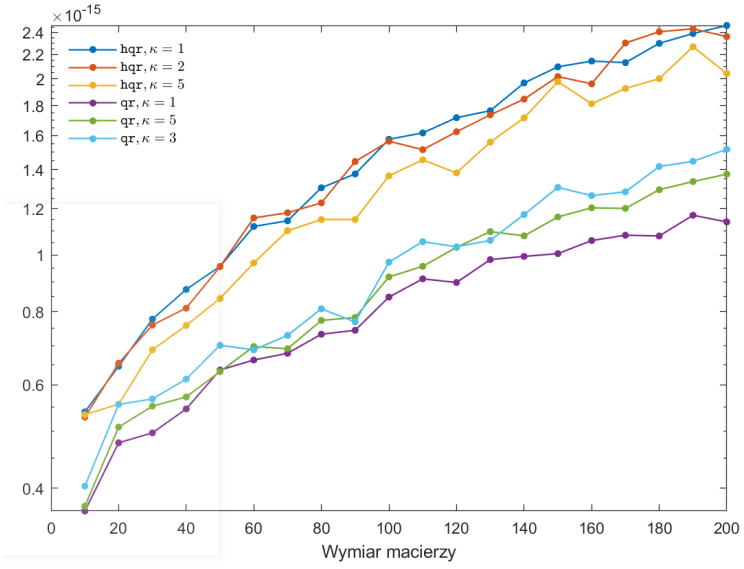
Eksperyment 8.2

W dalszych częściach skupimy się na, posiadających lepsze własności numeryczne, metodach bazujących na zastosowaniu przekształceń Householdera. Rozważymy przedstawiony

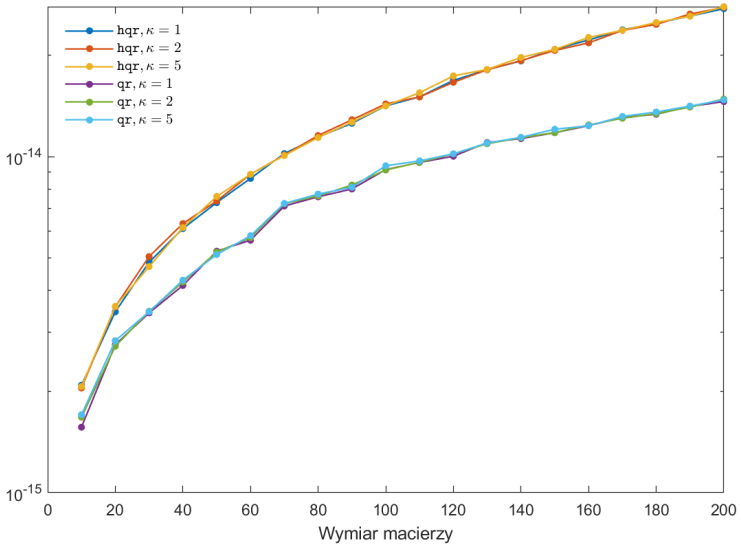
już model implementacji `hqr` i porównamy go z wbudowaną procedurą środowiska MATLAB-a `qr`. Wykorzystamy ten sam skrypt – z jedyną różnicą, polegającą na zastąpieniu definicji `fun={@cgs,@mgs}` instrukcją `fun={@hqr,@qr}`. Odpowiednie wykresy przedstawiono na rysunkach 8.6–8.8.



Rys. 8.6. Średni czas wykonania w sekundach



Rys. 8.7. Średni błąd rozkładu



Rys. 8.8. Średni błąd ortogonalności

Funkcja `qr` charakteryzuje się wyraźnie niższym czasem obliczeń, co wydaje się oczywiste, ze względu na to, że jej kod jest wykonywany w postaci skompilowanej, a nie interpretowanej. Błędy rozkładu i ortogonalności są bardzo niskie, nawet dla macierzy źle uwarunkowanych, dla procedury `hqr` tylko nieco wyższe.

8.4. Algorytm QR obliczania wartości własnych

Ogólną ideę prostego wariantu algorytmu QR można zapisać następująco.

0. Sprowadzamy macierz A do postaci Hessenberga A_0, $i := 0$, $k := n$.
1. Jeśli element $a_{k,k-1}$ macierzy A_i jest bliski zeru, jako przybliżenie wartości własnej λ_k przyjmujemy $a_{k,k}$, macierz A_i zastępujemy macierzą zbudowaną z jej pierwszych $k-1$ wierszy i kolumn (deflacja), $k := k-1$, $\rightarrow 2$.
   Jeśli wartości własne macierzy $\begin{bmatrix} a_{k-1,k-1} & a_{k-1,k} \\ a_{k,k-1} & a_{k,k} \end{bmatrix}$ są zespolone i element $a_{k-1,k-2}$ macierzy A_i jest bliski zeru, jako przybliżenie wartości własnych λ_k i λ_{k-1} przyjmujemy wartości własne tego bloku, macierz A_i zastępujemy macierzą zbudowaną z jej pierwszych $k-2$ wierszy i kolumn (deflacja), $k := k-2$, $\rightarrow 2$.
2. Jeśli k jest równe 1, jako przybliżenie wartości własnej λ_1 przyjmujemy $a_{1,1} \rightarrow$ stop.
3. Konstruujemy rozkład QR macierzy $A_i = Q_i R_i$.
4. Obliczamy $A_{i+1} := R_i Q_i$, $i := i+1$, $\rightarrow 1$.

Odwrócenie kolejności mnożenia macierzy w rozkładzie QR i w kroku 4. algorytmu QR wiąże się z tym, że macierze A_i i A_{i+1} mają te same wartości własne. Zachodzi bowiem

$$A_{i+1} = R_i Q_i = Q_i^{-1} Q_i R_i Q_i = Q_i^{-1} A_i Q_i,$$

a więc macierze A_i i A_{i+1} są podobne. Zauważmy też, że dla macierzy A, która jest macierzą Hessenberga, rozkład ortogonalny $QR = A$ zachowuje postać Hessenberga w macierzy Q. Co więcej, macierz RQ także jest macierzą Hessenberga, ponieważ przy mnożeniu macierzy R i Q, dla elementów pod pierwszą podprzekątną iloczynu, w odpowiednich wierszach macierzy Q i kolumnach macierzy R, niezerowe elementy w jednym wektorze „spotykają się" z zerowymi elementami, a elementy o tym samym indeksie w wektorze drugim.

Rozpoczniemy od zbadania poniższej, prostej implementacji.

```
% Algorytm QR bez przesunięć
function [lambda,it]=eig_qr0(A)
it=0; n=size(A,1); A=hess(A); k=n;
while k>1
  if abs(A(k,k-1))<=2*eps*(abs(A(k-1,k-1))+abs(A(k,k)))
    A(k,k-1)=0; k=k-1;
  elseif k>2 && abs(A(k-1,k-2))<=2*eps*(abs(A(k-2,k-2))+abs(A(k-1,k-1)))
    A(k-1,k-2)=0; k=k-2;
  else
    if it>=1000*n, lambda=[]; return; end
    [Q,R]=qr(A(1:k,1:k)); A(1:k,1:k)=R*Q;
    it=it+1;
  end
end
lambda=eig_diag(A);
end
```

Funkcja `eig_diag`, której kod zamieszczono poniżej, zwraca wektor wartości własnych rzeczywistej macierzy blokowodiagonalnej o blokach o wymiarze 1 lub 2, a funkcja pomocnicza `eig2` wylicza, na podstawie wzorów analitycznych, wartości własne dla bloków o wymiarze równym 2.

```
function lambda=eig_diag(A)
n=size(A,1); lambda=zeros(n,1); d=diag(A,-1); k=1;
while k<=n-1
  if d(k), lambda(k:k+1)=eig2(A(k:k+1,k:k+1)); k=k+2;
  else lambda(k)=A(k,k); k=k+1; end
end
if k==n,lambda(k)=A(k,k); end
  function lambda=eig2(A2)
    if size(A2,1)==1, lambda=A2; return; end
    b=trace(A2); c=det(A2); delta=b^2-4*c;
    if delta>=0, sq=sqrt(delta); else sq=1i*sqrt(-delta); end
    lambda=[(b+sq)/2;(b-sq)/2];
  end
end
```

Dla czterech serii po sto macierzy o wymiarze 10×10, dla których liczba par sprzężonych zespolonych wartości własnych wynosiła odpowiednio 0, 1, 2 i 3, wykonamy prosty test. Przypadki, dla których liczba iteracji przekroczy 10 000, uznamy za niezbieżne, dla pozostałych obliczymy średnią liczbę iteracji.

```
n=10; nc=[0 0 0 0]; it=[0 0 0 0]; load rng_set_eiq_qr0 s; rng(s);
for k=0:1:3
  for i=1:100
     A=gen_mat(n,1,k); [~,it_]=eig_qr0(A);
    if it_<1000*n, nc(k+1)=nc(k+1)+1; it(k+1)=it(k+1)+it_;
    end
  end
end
n_conv=nc, aver_it=round(it./nc)
```

Wynik jest następujący:

```
n_conv  =   100    86    72    55
aver_it =   443   942  1048  1269
```

Algorytm jest w miarę zbieżny tylko dla przypadku wyłącznie rzeczywistych wartości własnych. Im większa była liczba par zespolonych, tym liczba przypadków zbieżnych była mniejsza, a średnia liczba iteracji narastała. Pewnym rozwiązaniem jest zastosowanie przesunięć. Zaobserwowano, że algorytm QR jest znacznie lepiej zbieżny, jeśli w krokach 3. i 4. zastosuje się przesunięcia pojedyncze o tzw. iloraz Rayleigha.

$$\mu = a_{k,k},\ A_i - \mu I = Q_i R_i,\ A_{i+1} = R_i Q_i + \mu.$$

Modyfikujemy kod naszego testu, zamieniając wywołania funkcji `eig_qr0` na `eig_qr1`, gdzie nowa funkcja jest identyczna z poprzednią, z wyjątkiem linii, w której wyznaczane są rozkłady QR, która w `eig_qr1` ma postać:

```
sI=A(k,k)*eye(k); [Q,R]=qr(A(1:k,1:k)-sI); A(1:k,1:k)=R*Q+sI;
```

Po wykonaniu zmodyfikowanego kodu otrzymujemy wynik:

```
n_conv  =  100    85    73    52
aver_it =   24    65   122   197
```

który jest wyraźnie lepszy, ale nowa wersja także zawodzi przy większej liczbie par zespolonych wartości własnych. Dla niektórych problemów, np. z macierzami symetrycznymi, ten algorytm może być uznany za zadowalający. Nieco uprzedzając dalsze rozważania, w celu porównania, podajemy wynik analogicznego testu dla wariantu z przesunięciami podwójnymi (`eig_qr`), który będzie szczegółowo analizowany w dalszej części rozdziału:

```
n_conv  =  100   100   100   100
aver_it =   19    21    22    23
```

Wariant ten wykorzystuje następujący pomysł. Jeśli wartości własne macierzy

$$H_0 = \begin{bmatrix} a_{k-1,k-1} & a_{k-1,k} \\ a_{k,k-1} & a_{k,k} \end{bmatrix}$$

są rzeczywiste, to stosujemy tzw. przesunięcie Wilkinsona, czyli przesunięcie pojedyncze z μ równym wartości własnej H_0 bliższej $a_{k,k}$. W przypadku macierzy, która posiada parę sprzężonych, zespolonych wartości własnych, μ i $\overline{\mu}$, realizujemy tzw. przesunięcie podwójne Francisa. Będziemy chcieli od razu osiągnąć wynik dwu kolejnych iteracji QR z przesunięciami pojedynczymi dla macierzy zespolonych

$$A_i - \mu I = Q_i R_i$$

$$A_{i+1}^* = R_i Q_i + \mu I$$

$$A_{i+1}^* - \overline{\mu} I = Q_i^* R_i^*$$

$$A_{i+2} = R_i^* Q_i^* + \overline{\mu} I.$$

Po odjęciu stronami równania drugiego i trzeciego otrzymujemy związek

$$Q_i^* R_i^* = R_i Q_i + (\mu - \overline{\mu}) I.$$

Mnożymy stronami przez Q_i z lewej i R_i z prawej

$$Q_i Q_i^* R_i^* R_i = Q_i R_i (Q_i R_i + (\mu - \overline{\mu}) I) = (A_i - \mu I)(A_i - \overline{\mu} I) = A_i^2 - 2\mathrm{Re}(\mu) A_i + |\mu|^2 I.$$

Oznaczmy

$$M = A_i^2 - 2\mathrm{Re}(\mu) A_i + |\mu|^2 I.$$

M jest macierzą rzeczywistą, dla której macierze $Q_i Q_i^*$ i $R_i^* R_i$ tworzą rozkład QR. Ponieważ dla macierzy nieosobliwych rozkład ortogonalny jest jednoznaczny z dokładnością do jednoczesnego pomnożenia wierszy macierzy ortogonalnej i kolumn macierzy trójkątnej przez elementy dowolnego ciągu złożonego z wartości 1 lub -1, macierze $Q_i Q_i^*$ i $R_i^* R_i$ są rzeczywiste. Zauważmy, że

$$Q_i^* A_{i+2} = Q_i^* \left(R_i^* Q_i^* + \overline{\mu} I \right) = Q_i^* R_i^* Q_i^* + \overline{\mu} Q_i^* = (A_{i+1}^* - \overline{\mu} I) Q_i^* + \overline{\mu} Q_i^* = A_{i+1}^* Q_i^*$$

i analogicznie

$$Q_i A_{i+1}^* = A_i Q_i,$$

a stąd

$$Q_i Q_i^* A_{i+2} = Q_i A_{i+1}^* Q_i^* = A_i Q_i Q_i^*,$$

co oznacza, że macierze A_i, A_{i+1}^* i A_{i+2} tworzą ciąg ortogonalnie podobnych macierzy Hessenberga. Macierz M nie musi być macierzą Hessenberga, ze względu na składnik A_i^2, w którym

elementy niezerowe „przesuwają się" o jedną podprzekątną w dół. Oznaczmy przez G macierz ortogonalną rozkładu QR macierzy M. Poprzednią definicję A_{i+2} zastępujemy formułą

$$A_{i+2} = G^{\top} A_i G.$$

Oczywiście A_{i+2} jest macierzą Hessenberga. W sumie zamiast dwóch rozkładów QR konstruujemy tylko jeden, i to bez obliczania macierzy R, a także realizujemy jedno przekształcenie podobieństwa macierzy Hessenberga, przy znanej odwrotności macierzy przejścia. Postępowanie to pozwala uniknąć obliczeń na liczbach zespolonych. Uzasadnione jest pytanie, czy działanie macierzy G na A_i jest identyczne z działaniem macierzy $Q_i Q_i^*$. Odpowiedź daje twierdzenie o niejawnym obliczaniu macierzy Q w rozkładzie Q ([6], tw. 7.4.2). W skrócie twierdzenie to mówi, że jeśli dwie rzeczywiste macierze ortogonalne sprowadzają zadaną macierz Hessenberga do postaci trójkątnej, to – pod warunkiem, że mają identyczne pierwsze kolumny – wszystkie ich pozostałe kolumny są sobie równe z dokładnością do czynnika ± 1. W naszym przypadku warunek występujący w tym twierdzeniu jest spełniony, gdy w konstrukcji pierwszej kolumny macierzy ortogonalnej w rozkładzie QR zastosujemy odbicie Householdera.

Algorytm przesunięć Francisa jest bardzo skuteczny numerycznie. W bardziej zaawansowanych implementacjach, dla macierzy o większym wymiarze n, obliczenia organizowane są w taki sposób, że udaje się osiągnąć złożoność obliczeniową rzędu n^2. Algorytm QR wykorzystywany jest w zasadzie w przypadku tzw. macierzy pełnych (lub gęstych). W przypadku macierzy rzadkich (o proporcjonalnie małej liczbie elementów niezerowych) lub macierzy o znanej strukturze stosuje się inne wyspecjalizowane algorytmy. Poniżej podano przykładową implementację metody obliczania wartości własnych QR z przesunięciami pojedynczymi i podwójnymi.

```
% Algorytm QR z przesunięciami podwójnymi

function [lambda,it,D]=eig_qr(A,qr_fun)
it=0; n=size(A,1); A=hess(A); k=n;
while k>1
  if abs(A(k,k-1))<=2*eps*(abs(A(k-1,k-1))+abs(A(k,k)))
    A(k,k-1)=0; k=k-1;
  else
    r=(A(k,k)-A(k-1,k-1))/(2*A(k,k-1));
    s=r^2+A(k-1,k)/A(k,k-1);
    if s>=0 % dwa pierwiastki rzeczywiste
      s=sqrt(s);
      if r<0, s=-s; end
      if r+s~=0, s=A(k,k)+A(k-1,k)/(r+s); end
      % pojedyncze przesunięcie Wilkinsona
      sI=s*eye(k); [Q,R]=feval(qr_fun,A(1:k,1:k)-sI); A(1:k,1:k)=R*Q+sI;
      it=it+1;
    else % para sprzężonych pierwiastków zespolonych
      if k==2, k=0;
```

```
      elseif abs(A(k-1,k-2))<=2*eps*(abs(A(k-2,k-2))+abs(A(k-1,k-1)))
        A(k-1,k-2)=0; k=k-2;
      else
        % ślad i wyznacznik macierzy 2x2
        t=A(k-1,k-1)+A(k,k); d=A(k-1,k-1)*A(k,k)-A(k,k-1)*A(k-1,k);
        % podwójne przesunięcie Francisa
        M=A(1:k,1:k)^2-t*A(1:k,1:k)+d*eye(k);
        it=it+1;
        [G,~]=feval(qr_fun,M); A(1:k,1:k)=triu(G'*A(1:k,1:k)*G,-1);
        it=it+1;
      end
    end
  end
end
D=A; lambda=eig_diag(D);
end
```

Drugim argumentem funkcji jest wskaźnik do funkcji, która ma być używana do generowania rozkładów QR. W dalszych eksperymentach będą to funkcje `hqr` i MATLAB-owska funkcja `qr`. W porównaniach zostanie też użyta, do wyliczania uogólnionych wartości własnych, wbudowana funkcja MATLAB-a `eig`, która korzysta z rozkładów Choleskiego i w miejsce metody bazującej na algorytmie QR używa analogicznej metody QZ.

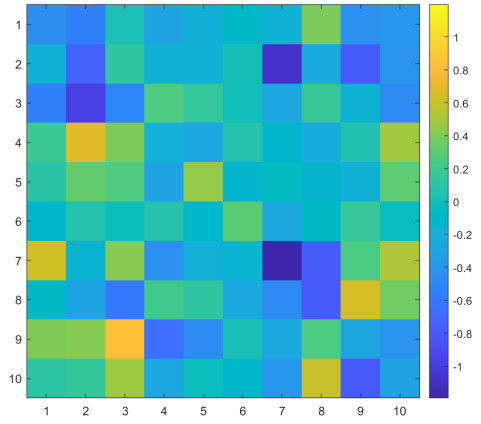
Eksperyment 8.3

Zajmiemy się oceną własności przedstawionej prostej implementacji metody obliczania wartości własnych macierzy. Rozpoczniemy od zastosowania procedury `eig_qr` do przykładowej macierzy 10×10. Aby zilustrować działanie procedury, przy kończeniu każdej iteracji, po instrukcjach `it = it + 1`, zostały wstawione wywołania `mat_image(A,1e-14)` funkcji służącej do wizualizacji struktury macierzy.

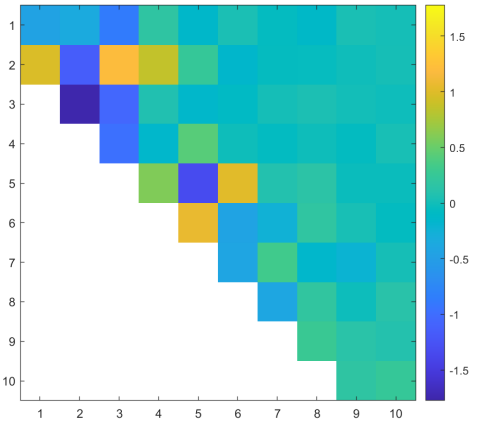
```
function mat_image(A,thr)
colormap parula(1024);
ma=1.01*max(max(abs(A)));
if nargin==1, imagesc(A,[-ma ma]);
else
  A(abs(A)<thr)=-1.01*ma;
  h=imagesc(A,[-ma ma]); hf=gcf; cmap=hf.Colormap;
  cmap(1,:)=1; % na biało zaznaczamy elementy o małych modułach
  hf.Colormap=cmap;
end
colorbar; daspect([1 1 1])
end
```

Ewolucję macierzy obrazują rysunki 8.9–8.13. Na początku macierz sprowadzana jest do postaci Hessenberga. Do iteracji dziesiątej wykrywane są trzy rzeczywiste wartości własne, a na pozycjach 6. i 7. na przekątnej pojawia się blok 2×2 ze sprzężonymi zespolonymi wartościami własnymi. Następnie po czterech iteracjach podwójnych z przesunięciami Francisa

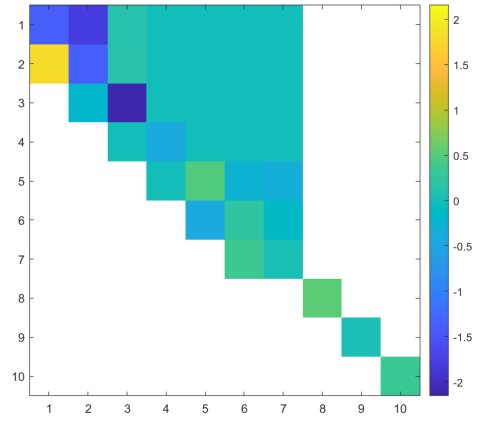
w iteracji 18., wyodrębnia się on w pełni w diagonalnej strukturze macierzy. Kolejne pięć iteracji z przesunięciami pojedynczymi Wilkinsona pozwala umiejscowić na przekątnej kolejne trzy wartości rzeczywiste oraz, na końcu, blok 2×2 odpowiadający drugiej parze sprzężonej. Ponieważ kolejne rozkłady QR generowane są dla macierzy w postaci Hessenberga, które w wyniku stopniowych transformacji zbliżają się do macierzy blokowodiagonalnej, to kluczowa w ocenie zbieżności jest obserwacja modułów elementów na podprzekątnej (rys. 8.14). W iteracjach od 10. do 18. możemy zaobserwować efekt działania czterech podwójnych przesunięć Francisa i wyodrębnienie się bloku 2×2 z parą sprzężoną zespolonych wartości własnych. W sumie zbieżność uzyskano po wykonaniu łącznie 23 iteracji, czyli średnio po 2–3 iteracje na jedną wartość własną.



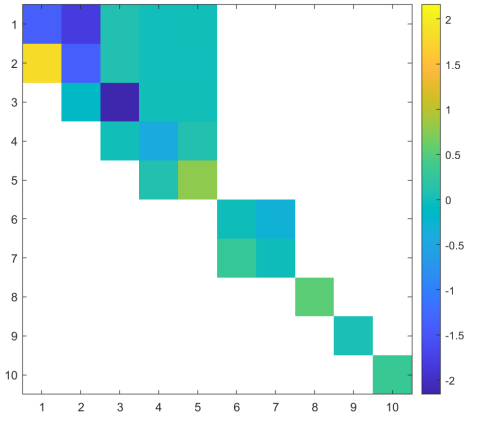
Rys. 8.9. Macierz początkowa



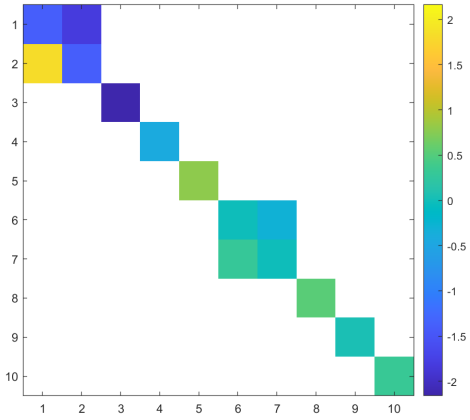
Rys. 8.10. Iteracja 0: macierz w postaci Hessenberga



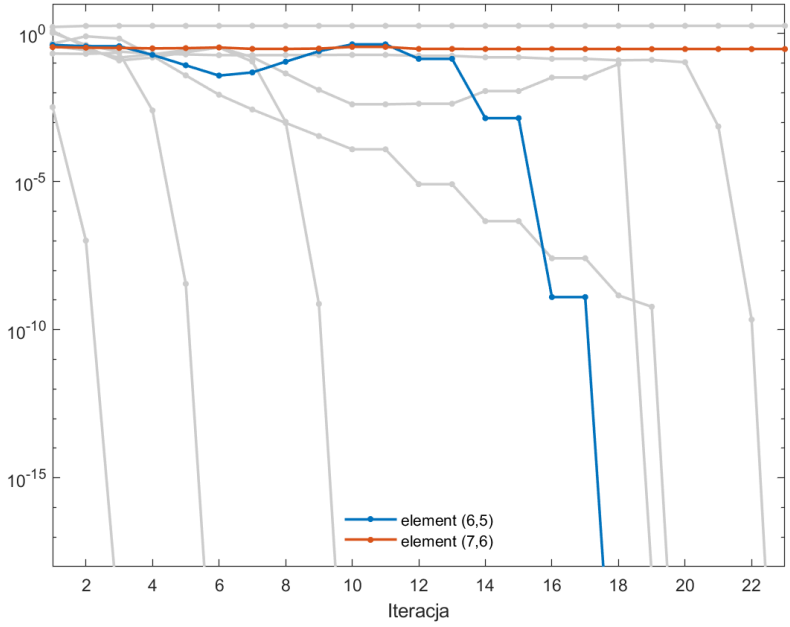
Rys. 8.11. Iteracja 10



Rys. 8.12. Iteracja 18



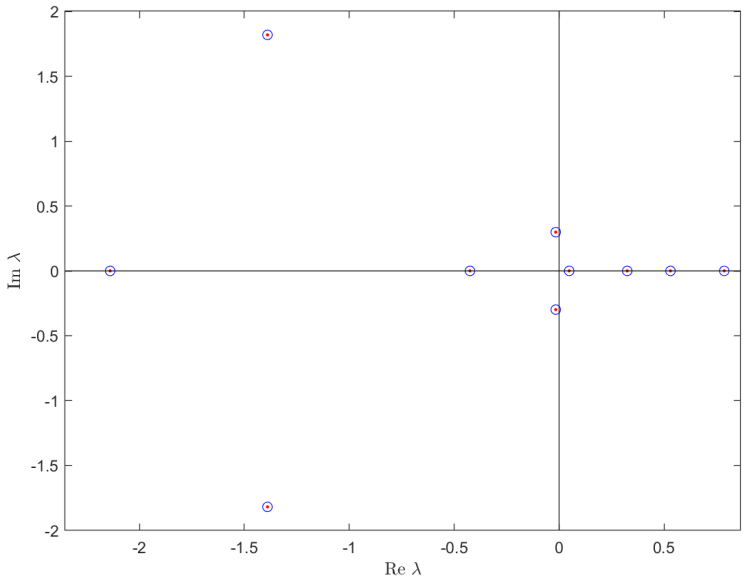
Rys. 8.13. Iteracja 23: macierz końcowa



Rys. 8.14. Moduły elementów na podprzekątnej

Rysunek 8.15 przedstawia porównanie położenia na płaszczyźnie zespolonej wartości własnych macierzy początkowej i końcowej. Został on utworzony za pomocą funkcji `eig_plot`. Czerwonymi kropkami zaznaczono wylosowane wartości własne, a niebieskimi kółkami wartości otrzymane za pomocą analizowanych procedur numerycznych, zastosowanych do macierzy wyjściowej.

```
function eig_plot(lam,lam_)
if nargin==1
  llr=double(real(lam)); lli=double(imag(lam)); plot(llr,lli,'bo'); drawnow;
elseif nargin==2
  clf; plot(real(lam),imag(lam),'bo',real(lam_),imag(lam_),'r.');
  llr=real([lam;lam_]); lli=imag([lam;lam_]);
else error('wrong number of input arguments'); end
end
```



Rys. 8.15. Wartości własne macierzy początkowej i końcowej

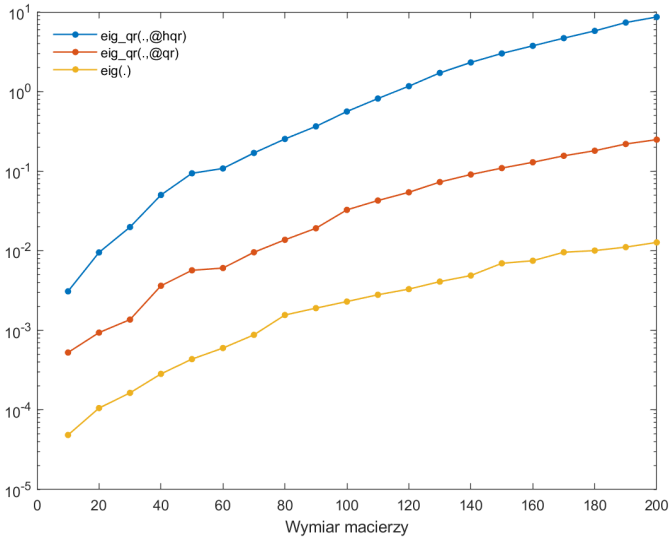
Eksperyment 8.4

W kolejnym przypadku wywołamy funkcję `eig_qr` i `eig` dla serii macierzy o danym wymiarze. Błąd wyznaczenia wartości własnych określono jako normę różnicy wektorów zespolonych utworzonych po ich uporządkowaniu według wartości części rzeczywistych i zespolonych, jeśli części rzeczywiste są równe. Posługujemy się do tego funkcją `eig_sort`:

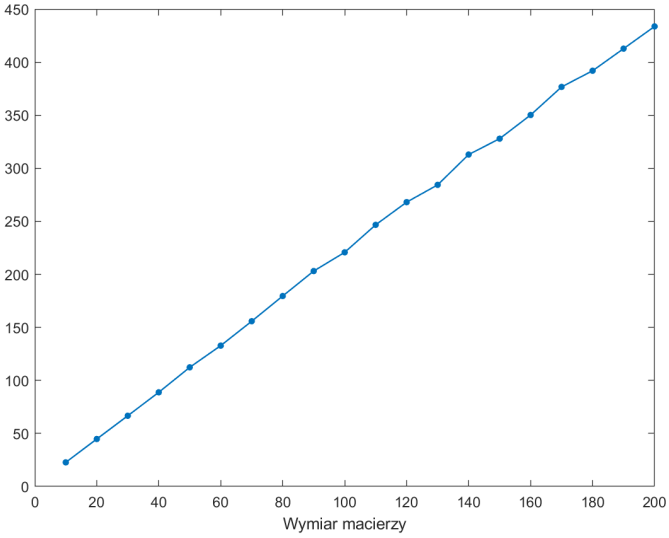
```
Nmax=200; imax=Nmax/10; Nruns=100; load rng_set_eig_2 s; rng(s);
et=zeros(imax,3); qr_it=zeros(imax,2); eig_err=zeros(imax,3);
for i=1:imax
  disp(['i      = ' num2str(i)]); t0=clock;
  for ii=1:Nruns
    [A,lambda_]=gen_mat(10*i,1);
    tic; [lambda,it]=eig_qr(A,@hqr); et(i,1)=et(i,1)+toc;
    qr_it(i,1)=qr_it(i,1)+it;
    eig_err(i,1)=eig_err(i,1)+norm(eig_sort(lambda)-lambda_)/norm(lambda_);
    tic; [lambda,it]=eig_qr(A,@qr); et(i,2)=et(i,2)+toc;
    qr_it(i,2)=qr_it(i,2)+it;
```

```
    if qr_it(i,1)~=qr_it(i,1), disp(qr_it(i,:)); end
    eig_err(i,2)=eig_err(i,2)+norm(eig_sort(lambda)-lambda_)/norm(lambda_);
    tic; lambda=eig(A); et(i,3)=et(i,3)+toc;
    eig_err(i,3)=eig_err(i,3)+norm(eig_sort(lambda)-lambda_)/norm(lambda);
  end
  disp(['etime = ' num2str(etime(clock,t0))]);
end
```

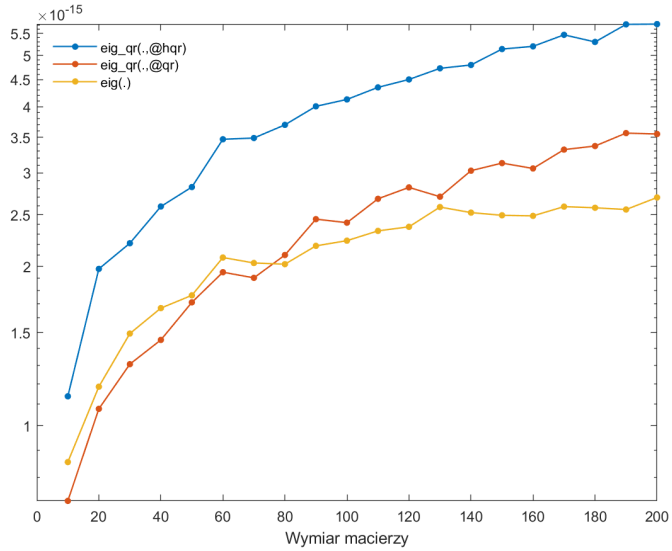
Uzyskane wyniki przedstawiono na rysunkach 8.16–8.18.



Rys. 8.16. Średni czas wykonania w sekundach



Rys. 8.17. Średnia liczba iteracji QR



Rys. 8.18. Średni błąd wartości własnych

Co ciekawe, mimo że rozmiar macierzy wzrósł wielokrotnie, to dokładność wyznaczenia wartości własnych pozostała na podobnym poziomie. Poza zrozumiałymi różnicami w czasie wykonania, mamy w zasadzie do czynienia z procedurami o zbliżonej jakości.

Dokonaliśmy porównania (pod względem czasów obliczeń i uzyskiwanej dokładności) prostych wersji interpretowanych (opisanych wcześniej, teoretycznie poprawnych) z wersjami kompilowanymi, wbudowanymi w środowisko MATLAB-a. Plusem tych pierwszych jest „transparentność", tzn. możliwość sprawdzenia, „co się dzieje" w trakcie wykonania, zaletami tych drugich są wydajność i jakość (stabilność numeryczna). Dla przykładu, na podstawie wyników zaprezentowanego eksperymentu możemy stwierdzić, że wyliczenie jednej wartości własnej wymaga średnio 2,5 iteracji QR. Wydaje się to bardzo niewiele, przy uwzględnieniu dokładności, z jaką wartości własne są wyliczane. Pokazuje to w istocie, jak silna jest zbieżność algorytmu QR. Tłumaczy to zarówno jego popularność, jak i znaczenie dla współczesnych metod numerycznych.

8.5. Obliczenia symboliczne i arytmetyka zmiennej precyzji

W tej części przyjrzymy się kwestii wrażliwości numerycznej algorytmów do wyznaczania wartości własnych względem wartości macierzy, dla której są one obliczane. W tym celu przygotujemy ciąg macierzy Toeplitza o wymiarach od 10×10 do 200×200, zwiększanych z krokiem 10. Wybrana została szczególna struktura macierzy Toeplitza z elementami o wartości 1 ponad przekątną i stałych wartościach na przekątnej i kolejnych podprzekątnych,

do dziewiątej włącznie, które tworzą ciąg $\{1,-1,-1,1,1,1,1,-1,-1,-1\}$. Wszystkie wartości poniżej dziewiątej podprzekątnej są równe zeru. Poniżej macierz o wymiarze 15×15.

```
col=[1 -1 -1  1  1  1  1  1 -1 -1 zeros(1,n-10)];
n=15; disp(toeplitz(col(1:n),ones(1,n)));
   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  -1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  -1  -1   1   1   1   1   1   1   1   1   1   1   1   1   1
   1  -1  -1   1   1   1   1   1   1   1   1   1   1   1   1
   1   1  -1  -1   1   1   1   1   1   1   1   1   1   1   1
   1   1   1  -1  -1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1  -1  -1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1  -1  -1   1   1   1   1   1   1   1   1
  -1   1   1   1   1   1  -1  -1   1   1   1   1   1   1   1
  -1  -1   1   1   1   1   1  -1  -1   1   1   1   1   1   1
   0  -1  -1   1   1   1   1   1  -1  -1   1   1   1   1   1
   0   0  -1  -1   1   1   1   1   1  -1  -1   1   1   1   1
   0   0   0  -1  -1   1   1   1   1   1  -1  -1   1   1   1
   0   0   0   0  -1  -1   1   1   1   1   1  -1  -1   1   1
   0   0   0   0   0  -1  -1   1   1   1   1   1  -1  -1   1
```

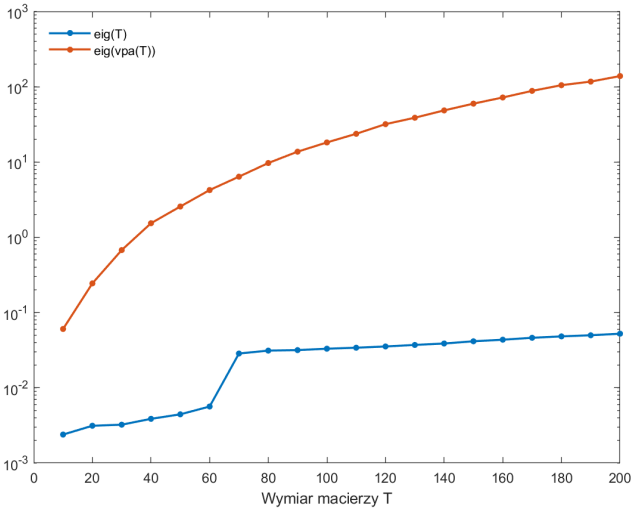
W MATLAB-ie wszystkie obliczenia wykonywane są domyślnie na liczbach w 64-bitowej reprezentacji zmiennoprzecinkowej, typu `double`. Jest to tzw. arytmetyka podwójnej precyzji. Odpowiada jej dokładność maszynowa na poziomie `eps` równym w przybliżeniu $2,22 \cdot 10^{-16}$, co oznacza dokładność reprezentacji na poziomie 16 cyfr dziesiętnych mantysy. W kolejnych eksperymentach numerycznych skorzystamy z możliwości dostarczanych przez bibliotekę MATLAB-a Symbolic Math Toolbox, przeznaczoną do obliczeń symbolicznych. Biblioteka ta pozwala na prowadzenie obliczeń w zadanej precyzji, ustalanej za pomocą funkcji `digits`. Do reprezentacji zmiennych w arytmetyce zmiennej precyzji służy specjalny typ `vpa` (*variable precision arithmetic*). Do rzutowania zmiennych innych typów numerycznych służy funkcja o tej samej nazwie. W przedstawianych dalej eksperymentach będzie używana reprezentacja 128-bitowa, czyli tzw. arytmetyka poczwórnej precyzji, zapewniająca w przybliżeniu dokładność 34 cyfr dziesiętnych mantysy lub lepszą.

Eksperyment 8.5

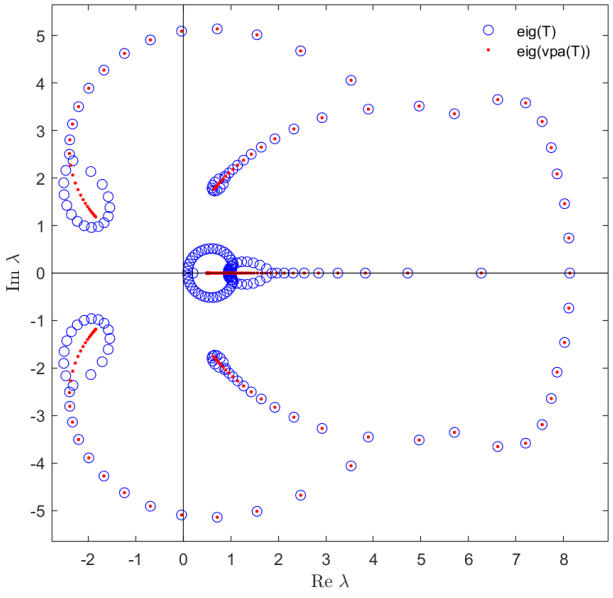
Za pomocą funkcji `eig` w podwójnej i w poczwórnej precyzji, obliczymy teraz wartości własne dla wybranego ciągu macierzy Toeplitza.

```
N=200; imax=N/10; lambda=zeros(imax,N); et=zeros(2,imax);
digits=32; lambda_=vpa(zeros(imax,N));
col=[1 -1 -1 1 1 1 1 1 -1 -1 zeros(1,N-10)];
for i=10:imax
  T=toeplitz(col(1:10*i),ones(1,10*i));
  tic; lambda(1:n,i)=eig(T); et(1,i)=toc;
  tic; lambda_(1:n,i)=eig(vpa(T)); et(2,i)=toc;
end
save lam_vpa_1 lambda_;
```

Jak widać na rysunku 8.19, w przypadku obliczeń z poczwórną precyzją, przy większych wymiarach macierzy, czasy obliczeń są raczej rzędu minut, podczas gdy dla podwójnej precyzji utrzymują się poniżej jednej dziesiątej sekundy.



Rys. 8.19. Czas wykonania w sekundach



Rys. 8.20. Wyliczone wartości własne macierzy T o wymiarach 200×200

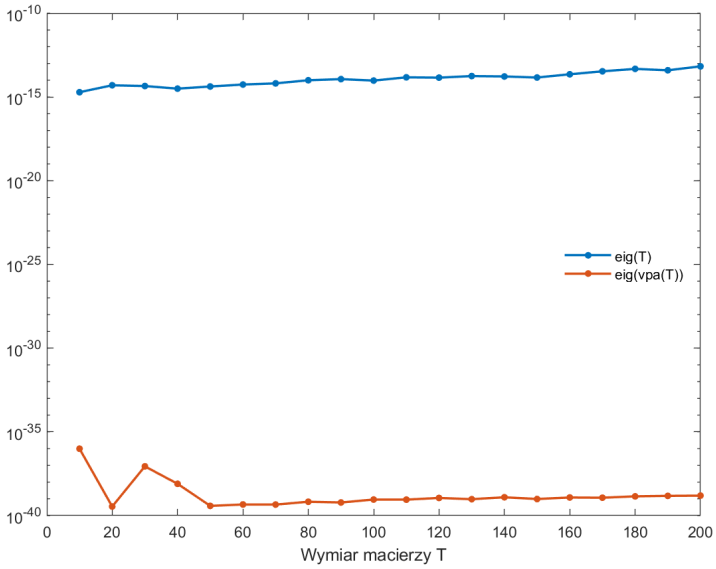
Przedstawione na rysunku 8.20 położenia wartości własnych w kilku obszarach są wyraźnie różne. Powstaje pytanie, który zestaw odpowiada lepszemu przybliżeniu dokładnych

wartości własnych. Oczywiście w odróżnieniu od analiz w poprzednich częściach, opartych na „syntetycznych" seriach macierzy, o zadanych pseudolosowych wartościach własnych, musimy teraz zastosować inne kryterium dokładności. Odwołamy się do definicji wartości własnych i skorzystamy z faktu, że funkcja `eig` może być wywołana także w następujący sposób: `[V,D]=eig(A)` i zwracana jest wtedy macierz wektorów własnych V oraz macierz diagonalna D z wartościami własnymi na przekątnej. Gdyby wektory i wartości własne były dokładne, tożsamościowo spełnione byłoby równanie $AV = VD$. Dlatego jako praktyczną miarę błędu wyliczenia wartości własnych przyjmiemy wielkość `norm(A*V-V*D)`.

Do obliczeń użyjemy poniższego skryptu.

```
N=200; imax=N/10; eig_err=zeros(2,imax); digits=32;
col=[1 -1 -1 1 1 1 1 -1 -1 zeros(1,N-10)];
for i=1:imax
  T=toeplitz(col(1:10*i),ones(1,10*i));
  [V,D]=eig(T); eig_err(1,n_)=norm(T*V-V*D)/norm(T*V);
  [V_,D_]=eig(vpa(T)); eig_err(2,n_)=norm(T*V_-V_*D_)/norm(T*V_);
end
```

Uzyskane wartości przedstawiono na rysunku 8.21.



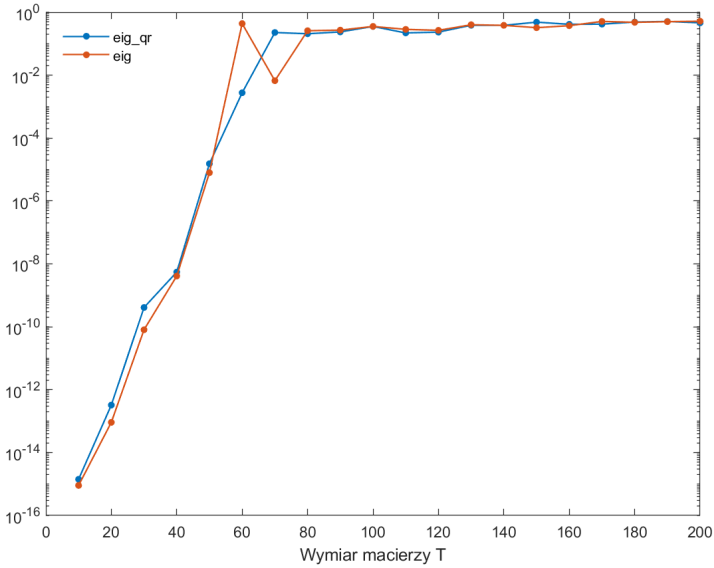
Rys. 8.21. Błąd spełnienia definicji wartości własnych

Wyniki nie pozostawiają wątpliwości, że metoda obliczania wartości własnych z użyciem poczwórnej precyzji wygenerowała znacznie lepsze przybliżenie wartości własnych dla analizowanej serii macierzy. Przyjmując to przybliżenie jako punkt odniesienia, za pomocą metody stosowanej we wcześniejszych częściach, zbadamy dokładność wyznaczenia wartości własnych przez funkcje `eig_qr` i `eig`.

Posłużymy się do tego następującym skryptem.

```
load lam_vpa_1 lambda_;
N=200; imax=N/10; eig_err=zeros(2,imax);
col=[1 -1 -1 1 1 1 1 -1 -1 zeros(1,N-10)];
fun={@eig_qr,@eig};
for i=1:imax
  T=toeplitz(col(1:10*i),ones(1,10*i));
  lambda__=eig_sort(double(lambda_(1:10*i,i))); nl__=norm(lambda__);
  eig_err(i,1)=norm(eig_sort(eig_qr(T,@qr))-lambda__)/nl__;
  eig_err(i,2)=norm(eig_sort(eig(T))-lambda__)/nl__;
end
```

Otrzymane rezultaty obrazuje rysunek 8.22.

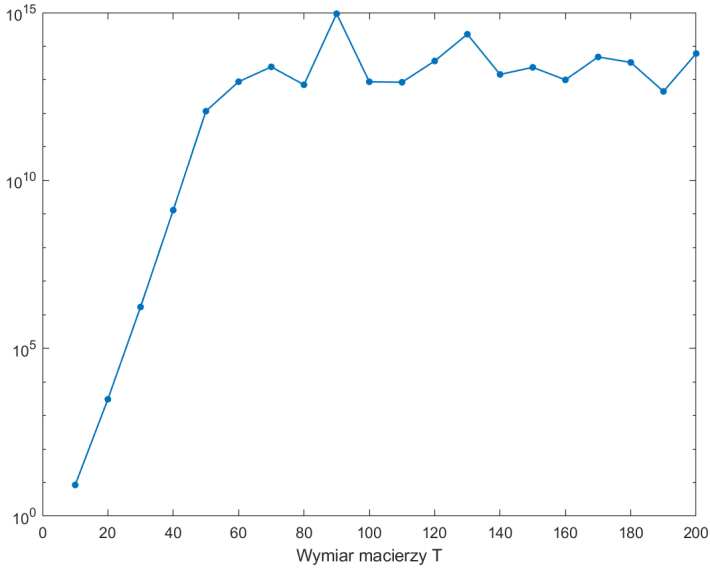


Rys. 8.22. Błąd wartości własnych

Widzimy, że mały błąd występuje właściwie tylko dla wymiaru macierzy równego 10, potem bardzo szybko narasta i już około wymiaru 60 osiąga poziom powyżej 1. Tłumaczy to wyraźny skok czasu wykonania funkcji `eig` widoczny na rysunku 8.19. Podobny poziom błędu funkcji `eig_qr` i `eig` oznacza, że problem tkwi nie w algorytmie wyznaczania wartości własnych, a raczej w metodzie wyznaczania rozkładów QR używanej w każdej iteracji.

Ogólnie wiadomo, że algorytm QR wykazuje bardzo dobrą zbieżność w przypadku macierzy o wyraźnie różnych wartościach własnych. Zbieżność ulega pogorszeniu, kiedy macierz zbliża się do macierzy z wielokrotnymi wartościami własnymi. Na rysunku 8.20 pokazującym rozmieszczenie wartości własnych obliczanych z podwójną i poczwórną precyzją można zauważyć pięć punktów zagęszczania się dokładniejszego przybliżenia wartości własnych i tam właśnie metoda mniej dokładna wykazuje znacznie gorszą zbieżność.

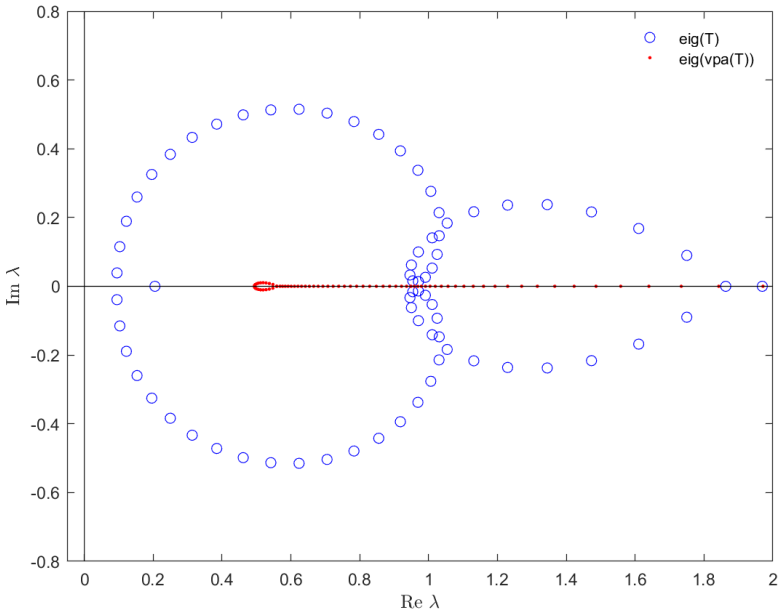
Wrażliwość problemu wyznaczania wartości własnych można ocenić za pomocą wskaźnika uwarunkowania wartości własnych wyliczanego w MATLAB-ie za pomocą funkcji `condeig`. Funkcja ta zwraca wektor zbudowany z elementów równych $\sec(\theta(v_i, w_i))$ (odwrotność funkcji $\cos(\theta(v_i, w_i))$), gdzie v_i i w_i są prawym i lewym wektorem własnym odpowiadającym wartości własnej λ_i. W przypadku macierzy o różnych wartościach własnych wektor `condeig` zawiera elementy większe lub równe 1. W przypadku macierzy o wartościach krotnych z lewą i prawą przestrzenią własną o wymiarze 1 elementy wektora `condeig` odpowiadające wartościom krotnym teoretycznie zmierzają do nieskończoności. Rysunek 8.23 przedstawia maksymalne wskaźniki uwarunkowania wartości własnych dla analizowanego ciągu macierzy Toeplitza. Wskaźnik uwarunkowania silnie rośnie i osiąga wartości powyżej 10^{13} już około wymiaru macierzy równego 60. Mamy tu pełną zgodność z wcześniej opisanymi efektami eksperymentów numerycznych.



Rys. 8.23. Maksymalny wskaźnik uwarunkowania wartości własnych

Powstaje pytanie, czy zastosowana poczwórna precyzja gwarantuje dokładność obliczenia wartości własnych na poziomie 10^{-32}. Przyjrzyjmy się bliżej powiększeniu położenia wartości własnych, uzyskanych dla macierzy Toeplitza `T` o wymiarach 200×200, za pomocą funkcji `vpa(T)` i `eig(vpa(T))`. Na rysunku 8.24 możemy dostrzec, że proces formowania się kolejnego owalu wokół punktu skupiania się, zaznaczonych na czerwono, wartości własnych, obliczonych przy poczwórnej precyzji, został zainicjalizowany. Prawdopodobnie efekt ten można by skorygować, stosując sześciokrotną precyzję. W praktyce postępowanie zależeć będzie od tego, do czego potrzebna jest aż tak dokładna znajomość położenia wartości własnych i jaki koszt obliczeniowy uznajemy za dopuszczalny.

Jak pokazało porównanie przedstawione na początku tego podrozdziału, przejście z podwójnej precyzji na poczwórna precyzję, za pomocą obliczeń symbolicznych połączonych z kontrolą błędu, wiąże się z ponad 1000-krotnym wzrostem czasu wykonania przy macierzach o wymiarach rzędu 200×200. Możliwe jest jednak nawet kilkudziesięciokrotne przyspieszenie obliczeń wysokiej precyzji przy zastosowaniu jednego z alternatywnych podejść. Przykładem takiego podejścia jest wykorzystanie dodatkowej, zewnętrznej, biblioteki MATLAB-a: Advanpix Multiprecision Computing Toolbox for MATLAB [11]. W zasadzie biblioteka bazuje na kompilowanym kodzie C/C++ najlepszych dostępnych obecnie algorytmów z różnych obszarów matematyki stosowanej, w tym metod numerycznych algebry liniowej.



Rys. 8.24. Wyliczone wartości własne macierzy T o wymiarach 200×200 (powiększenie)

Aby ocenić efektywność kompilowanego kodu wysokiej precyzji, odwołamy się do porównania następujących środowisk

- MATLAB/Multiprecision Computing Toolbox (MCT),
- MATLAB/Symbolic Math Toolbox (SMT),
- Julia z biblioteką `GenericSchur` (GS),
- Mathematica,
- Python z rozszerzeniem `scipy.linalg` i biblioteką `mpmath`.

Obliczano wartości własne szczególnej macierzy Toeplitza, tzw. circus matrix (zob. [12]), przy różnych jej wymiarach oraz precyzji 100 i 1000 cyfr dziesiętnych. Wyniki testów podano na stronie qiita.com/hanaata/. Kody odpowiednich programów umieszczono w otwartym

repozytorium [13]. W tabeli 8.1 podano uzyskane czasy obliczeń w sekundach, dla procesora Intel Core i9 9900k (osiem rdzeni, szesnaście wątków, 4,7 GHz), w nawiasie wartości dla procesora Intel Xeon W-2195 (osiemnaście rdzeni, trzydzieści dwa wątki, 3,2 GHz).

Tabela 8.1
Porównanie czasów obliczeń wartości własnych przy różnej precyzji i różnych wymiarach macierzy

Dokładność	**Wymiar macierzy**	**MATLAB /MCT**	**Mathematica**	**Julia/GS**	**MATLAB /SMT**
100 cyfr dziesiętnych	100×100	1,65	2,99	4,15	29,1
	500×500	142,0	365,5	665,0	3376,7
1000 cyfr dziesiętnych	100×100	6,52(6,33)	25,6	46,807	70,4
	500×500	541,3(406,3)	3048,4	5072,5	8413,6

W obliczeniach wykorzystano wersje oprogramowania aktualne w lipcu 2020 roku. Zestawienie nie obejmuje czasów obliczeń w Pythonie, kilkukrotnie większych niż dla wariantu MATLAB/SMT, prawdopodobnie dlatego, że użyto interpretowanych wersji kodu. Jak widać, czasy obliczeń dla testu MATLAB/MCT są od około 10 do ponad 20 razy krótsze niż dla testu MATLAB/SMT. W przypadku macierzy lepiej uwarunkowanych ze względu na obliczanie wartości własnych przyspieszenie jest kilkusetkrotne.

8.6. Podsumowanie

Głównym celem rozdziału było zaprezentowanie technik prowadzenia eksperymentów numerycznych, takich jak użycie powtarzalnych serii macierzy o znanych, w dobrym przybliżeniu, wartościach własnych czy użycie w tym celu obliczeń o zadanej wysokiej precyzji. Niejako przy okazji pokazano, że – dysponując dzisiejszą wiedzą i środkami obliczeniowymi – zadanie dokładnego wyznaczenia wartości własnych macierzy, o wymiarze rzędu setek, może być rozwiązane za pomocą kodu liczącego nie więcej niż kilkadziesiąt instrukcji (przekształcenie do postaci Hessenberga, rozkład QR i algorytm QR). Oczywiście mówimy tu o języku wysokiego poziomu typu MATLAB czy Python, gdzie operacje na macierzach zapisujemy pojedynczymi poleceniami.

Jeśli problem nie jest źle uwarunkowany, to kod może zachować dokładność niewiele odbiegającą od maszynowej. Trzeba pamiętać, że przedstawiona wersja jest bardzo prosta, daleka od profesjonalnego oprogramowania BLAS/LAPACK, w której zaimplementowano wiele udoskonaleń dokładności i efektywności obliczeń, jak np. adaptacyjne przesunięcia wielokrotne.

Rozwój metod numerycznych oczywiście nie zatrzymał się na algorytmie QR. Spośród innych metod zasługujących na uwagę wyróżniają się metody bazujące na podprzestrzeniach Kryłowa, np. metoda Lanczosa czy Arnoldiego (zob. [3]). Jeśli chodzi o stronę programistyczną, poza językami Fortran i C, pojawiło się zainteresowanie językiem C++, np. pakiet Armadillo.

Na zakończenie trzeba podkreślić, że o ile metody QR pozwalają efektywnie rozwiązywać zadania regresji liniowej, o tyle dzięki rozkładom według wartości osobliwych można budować macierze pseudoodwrotne i redukować modele liniowe do ich „istotnej" części. W wielu zastosowaniach dokładna znajomość wszystkich wartości i wektorów własnych nie jest konieczna do oceny stabilności modeli (zbieżności ich rozwiązań). Stąd wziął się nowy pomysł charakteryzacji przybliżonego położenia wartości własnych, zwłaszcza w obecności perturbacji, za pomocą tzw. pseudowidma macierzy [6, podrozdz. 7.9].

Bibliografia

[1] Leon S. J., Björck Å., Gander W., *Gram–Schmidt orthogonalization: 100 years and more*, „Numerical Linear Algebra with Applications", 2012, 20 (3), 492–532, doi: https://doi.org/10.1002/nla.1839.

[2] Tou E. R., *Math Origins: Eigenvectors and Eigenvalues. Convergence, Mathematical Association of America, 2018*, url: https://www.maa.org/press/periodicals/convergence/math-origins-eigenvectors-and-eigenvalues [31.03.2021].

[3] Golub G. H., Vorst H. A. van der, *Eigenvalue computation in the 20th century*, „Journal of Computational and Applied Mathematics", 2000, 123 (1–2), 35–65, doi: https://doi.org/10.1016/s0377-0427(00)00413-1.

[4] Golub G., Uhlig F., *The QR algorithm: 50 years later its genesis by John Francis and Vera Kublanovskaya and subsequent developments*, „IMA Journal of Numerical Analysis", 2009, 29 (3), 467–485, doi: https://doi.org/10.1093/imanum/drp012.

[5] Stewart G. W., *Matrix Algorithms. Volume 1: Basic Decompositions*, Society for Industrial and Applied Mathematics, 1998, doi: https://doi.org/10.1137/1.9781611971408.

[6] Golub G. H., Loan C. F. V., *Matrix computations*, 4th ed., Johns Hopkins University Press, Baltimore 2013.

[7] Björck Å., *Numerical Methods in Matrix Computations*, Springer International Publishing, 2015, doi: https://doi.org/10.1007/978-3-319-05089-8.

[8] Parlett B. N., *The QR algorithm*, „Computing in Science & Engineering", 2000, 2 (1), 38–42, doi: https://doi.org/10.1109/5992.814656.

[9] Higham N. J., *The Top 10 Algorithms in Applied Mathematics*, url: https://nhigham.com/2016/03/29/the-top-10-algorithms-in-applied-mathematics [31.03.2021].

[10] Higham N. J., Dennis M. R., Glendinning P., Martin P. A., Santosa F., Tanner J., red., *The Princeton Companion to Applied Mathematics*, Princeton University Press, 2016, doi: https://doi.org/10.1515/9781400874477.

[11] *Multiprecision Computing Toolbox for MATLAB. Advanpix LLC, Yokohama, Japan*, url: www.advanpix.com [31.03.2021].