Computational Physics

Exercise Sheet 07

Rene-Marcel Lehner

Summer Term 2023

ASSIGNMENT N°

07

Vorwort

Aufgabe 1 lief gut, Aufgabe 2 ist leider nicht vollständig. Ich glaube ich habe einen kleinen Fehler im Algorithmus, den ich aber nicht finde; vielleicht in der Theorie, vielleicht im Code. Das steht aber alles in der Aufgabe.

Hinzu zu meinem Frust kommt, dass mein iCloud-Sync meinen git Pull/Push zerschossen hat und ich die Python-Datei für Aufgabe 1 verloren habe.. Daraus ziehe ich zumindest die Lehre, automatisches File-Cloud-Sharing und git niemals wieder miteinander zu kombinieren.

Dafür habe ich aber die Auswertung für Aufgabe 1 etwas erweitert gehabt und für ein wenig Abwechselung mal das style package `matplotx` ausprobiert.

Bezüglich des Codes für Aufgabe 1:
Die Klassen `Householder` und `QR` kann man sehr gut in entsprechende `.h` und `.cpp` Dateien aufteilen. Dies war eigentlich auch der Fall (und Plan), aber das mochte der Debugger `dbg` nicht so sehr.
Da ich bis zum Schluss ge-debugged habe, habe ich die Klassen nicht mehr ausgelagert.

E0: Matrix diagonalization - Power method

Task:

Many physical systems can be described by matrices whose eigenvalues correspond to the eigenfrequencies of these systems. Therefore, matrix diagonalization is an important tool in physics, and the power method is a simple and significant technique.
Given the symmetric 4×4-matrix:

$$A = \begin{pmatrix} 1 & -2 & -3 & 4 \\ -2 & 2 & -1 & 7 \\ -3 & -1 & 3 & 6 \\ 4 & 7 & 6 & 4 \end{pmatrix}$$

(a) Determine the eigenvalues of the matrix using `numpy.linalg.eig` or the C++ library Eigen (`http://www.eigen.tuxfamily.org`).

(b) Determine the eigenvalues a second time by implementing the power method. Choose a suitable initial vector $\boldsymbol{v}_0$ and an appropriate number of iterations. Compare your results with the ones from the first part of the task.

a) Ausführung der Methode `np.linalg.eig(A)` liefert

```
Numpy eigenvalues:
(array( [-9.27793969, 12.08505811,  4.59704881,  2.59583277]),

array([ [-0.45657804, -0.07467658,  0.64264911,  0.61070616],
        [-0.49016739, -0.47399743,  0.24995413, -0.6874484 ],
        [-0.44274934, -0.42870916, -0.70253377,  0.35584796],
        [ 0.5960247 , -0.76547913,  0.17598679,  0.166808  ]]))
```

Die Eigenwerte der Matrix sind also:

$$\lambda_1 \approx -9.27793969$$
$$\lambda_2 \approx 12.08505811$$
$$\lambda_3 \approx 4.59704881$$
$$\lambda_4 \approx 2.59583277$$

und die zugehörigen Eigenvektoren sind:

$$\boldsymbol{v}_1 \approx \begin{pmatrix} -0.45657804 \\ -0.49016739 \\ -0.44274934 \\ 0.5960247 \end{pmatrix}, \boldsymbol{v}_2 \approx \begin{pmatrix} -0.07467658 \\ -0.47399743 \\ -0.42870916 \\ -0.76547913 \end{pmatrix}, \boldsymbol{v}_3 \approx \begin{pmatrix} 0.64264911 \\ 0.24995413 \\ -0.70253377 \\ 0.17598679 \end{pmatrix}, \boldsymbol{v}_4 \approx \begin{pmatrix} 0.61070616 \\ -0.6874484 \\ 0.35584796 \\ 0.166808 \end{pmatrix}$$

Man beachte, dass die **Spalten** der ausgegebenen Matrix (das 2. Array) die entsprechenden Eigenvektoren sind.

b) Die Potenzmethode liefert unter bestimmten Bedingungen (Abschnitt 1) den Eigenvektor zum **betragsmäßig größten Eigenwert**. Um weitere Eigenwerte zu erhalten, wird die Potenzmethode um das sogenannte **"Deflating"** erweitert.

Es wird der Einfluss des Eigenvektors (zum errechneten Eigenwert) von der ursprünglichen Matrix entfernt, sodass λ_2 der betragsmäßig größte Eigenwert wird. Die Potenzmethode liefert dann den Eigenvektor zu λ_2. Auf diese Weise können alle Eigenwerte und -vektoren bestimmt werden.

Die gesamte Programmausführung wird also zusammengefasst zu

```
    eigenvector_power = []
    eigenvalue_power = []

    for i in range(A.shape[0]):
        eigenvector_power.append(power_iteration(A, 10000))
        eigenvalue_power.append(eigenvalue(A, eigenvector_power[i]))

        A = deflating(A, eigenvector_power[i], eigenvalue_power[i])

```

wobei $\boldsymbol{A}$ als quadratisch vorausgesetzt wird.

Die berechneten Eigenwerte und -vektoren sind nach

```
    Power method eigenvalues: [12.085058112675243, -9.27793969008451,
    4.597048810634316, 2.5958327667749517]
    Power method eigenvectors:
        [array([0.07467658, 0.47399743, 0.42870916, 0.76547913]),
        array([ 0.45657804,  0.49016739,  0.44274934, -0.5960247 ]),
        array([-0.64264911, -0.24995413,  0.70253377, -0.17598679]),
        array([ 0.61070616, -0.6874484 ,  0.35584796,  0.166808  ])]

```

Eigenwerte

$$\begin{aligned}\lambda_1 &\approx -9.27793969008451\\ \lambda_2 &\approx 12.085058112675243\\ \lambda_3 &\approx 4.597048810634317\\ \lambda_4 &\approx 2.595832766774952\end{aligned}$$

Eigenvektoren

$$\boldsymbol{v}_1 \approx \begin{pmatrix}0.45657804\\0.49016739\\0.44274934\\-0.5960247\end{pmatrix} \boldsymbol{v}_2 \approx \begin{pmatrix}0.07467658\\0.47399743\\0.42870916\\0.76547913\end{pmatrix} \boldsymbol{v}_3 \approx \begin{pmatrix}-0.64264911\\-0.24995413\\0.70253377\\-0.17598679\end{pmatrix} \boldsymbol{v}_4 \approx \begin{pmatrix}0.61070616\\-0.6874484\\0.35584796\\0.166808\end{pmatrix}$$

Die Eigenwerte stimmen mit denen von `Numpy` **exakt** überein, genauso wie die Eigenvektoren (bis auf ein Vielfaches von -1). Man beachte, dass die Eigenwerte der Potenzmethode **genauer** als die Werte von `Numpy` bestimmt sind und die `float`-Präzision vollständig ausschöpfen.

Die Aufgabe könnte hier enden.
Da jedoch einige Fragen offen bleiben und eine Untersuchung des Konvergenzverhaltens aussteht, wird die Aufgabe etwas erweitert.

1. Was ist das charakteristische Polynom und gibt es eine analytische Lösung für die Eigenwerte? Falls nicht, kann eine genauere Lösung berechnet werden, die als Vergleich mit den Werten der Potenzmethode dient?
2. Wie gut konvergiert die Potenzmethode gegen die Eigenwerte? Wie viele Iterationen sind in etwa nötig, um einen genauen Eigenwert zu erhalten?
3. Wie hängt die Größe der Matrix mit der Konvergenzgeschwindigkeit und -genauigkeit ab?

Theorie

Charakteristisches Polynom

Das Charakteristisches Polynom lautet

$$\lambda^4 - 10\lambda^3 - 80\lambda^2 + 773\lambda - 1338 = 0 \quad .$$

In der Tat **gibt** es dazu eine analytische Lösung. Der **Satz von Abel-Ruffini** besagt, dass *keine geschlossene Formel für Gleichungen 5. Grades oder höher* gefunden werden kann. Im Umkehrschluss bedeutet dies, dass es für alle Gleichungen des Grades 4 oder kleiner geschlossene Formeln geben **muss**.

Eine geschlossene Formel für die vorliegende Quartische Gleichung wurde bereits vor vielen Jahrhunderten (*Lodovico Ferrari (1522–1565)*) gefunden. Da dort viele Wurzelterme auftreten, sind die Ergebnisse mit hoher Wahrscheinlichkeit **irrationale Zahlen**, weswegen auch die Ergebnisse der geschlossenen Formel die Maschinenpräzision vollständig ausschöpfen und nicht von den Ergebnissen einer geeigneten numerischen Methode abweichen würden.

Daher entspricht unser numerisches Ergebnis, das die maximale Präzision ausschöpft und vollständig konvergiert ist, der **Referenzlösung**, die für den relativen Fehler verwendet wird.

Potenzmethode

1. Wähle einen Startvektor $\boldsymbol{v}_0$.
2. Definiere eine Iteration durch $\boldsymbol{w}_n = \boldsymbol{A} \cdot \boldsymbol{v}_{n-1}$ und Normierung $\boldsymbol{v}_n = \frac{\boldsymbol{w}_n}{|\boldsymbol{w}_n|}$.

   Bedingungen: Die Eigenwerte sind **nicht** entartet, $\boldsymbol{A}$ ist diagonalisierbar und $\boldsymbol{v}_0$ ist **nicht orthogonal** zum Eigenvektor.

Es gilt auch, dass die Konvergenzgeschwindigkeit von dem **Verhältnis der zwei größten Eigenwerte** abhängt und **linear** ist:

$$\frac{\boldsymbol{w}_1}{|\boldsymbol{w}_1|} = \frac{\boldsymbol{A}^n \boldsymbol{v}_0}{|\boldsymbol{A}^n \boldsymbol{v}_0|} = \boldsymbol{x}_1 + \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|\right)$$

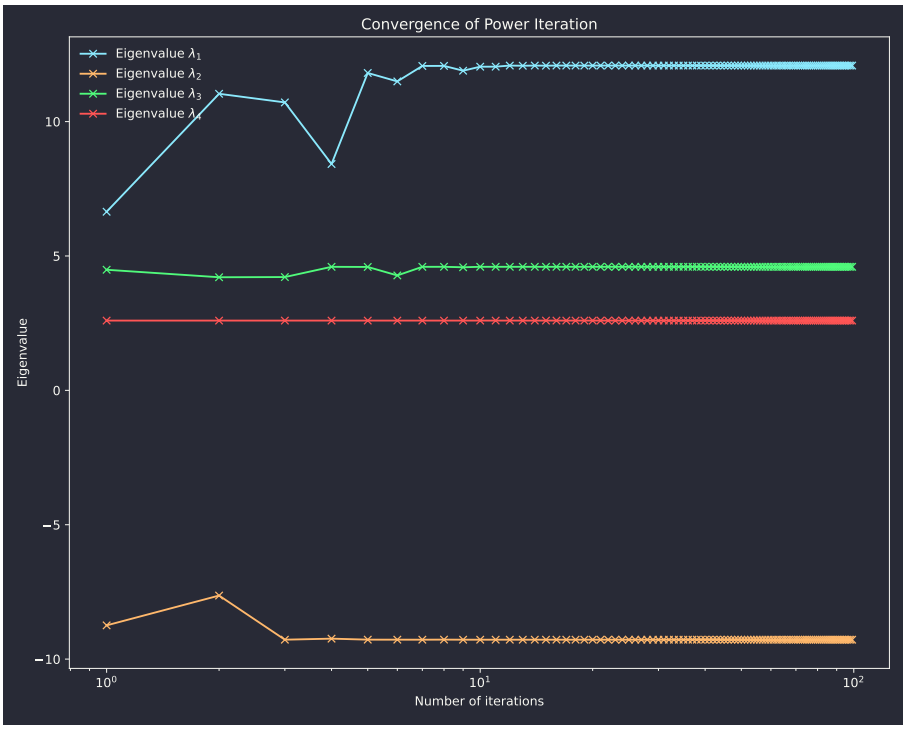

Figure 1.1: Approximierte Eigenwerte in Abhängigkeit der Iterationsschritte. Die Potenzmethode konvergiert schon für wenige Iterationsschritte.

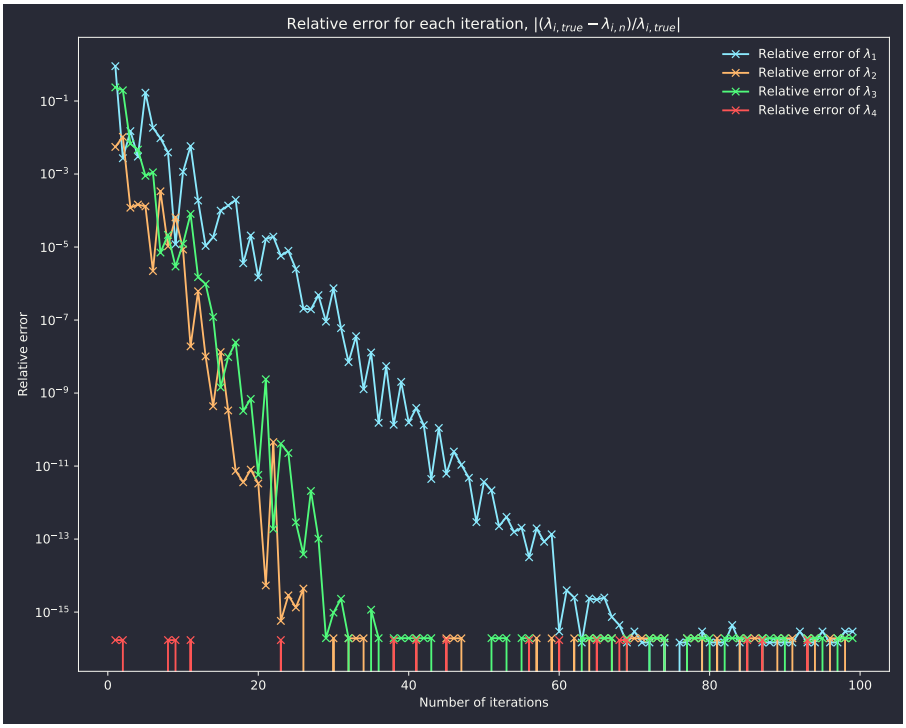

Figure 1.2: Relative Fehler in Abhängigkeit der Iterationsschritte. Die höchstmögliche Genauigkeit bzgl. der Maschinenpräzision wird in unter 100 Iterationsschritten erreicht.

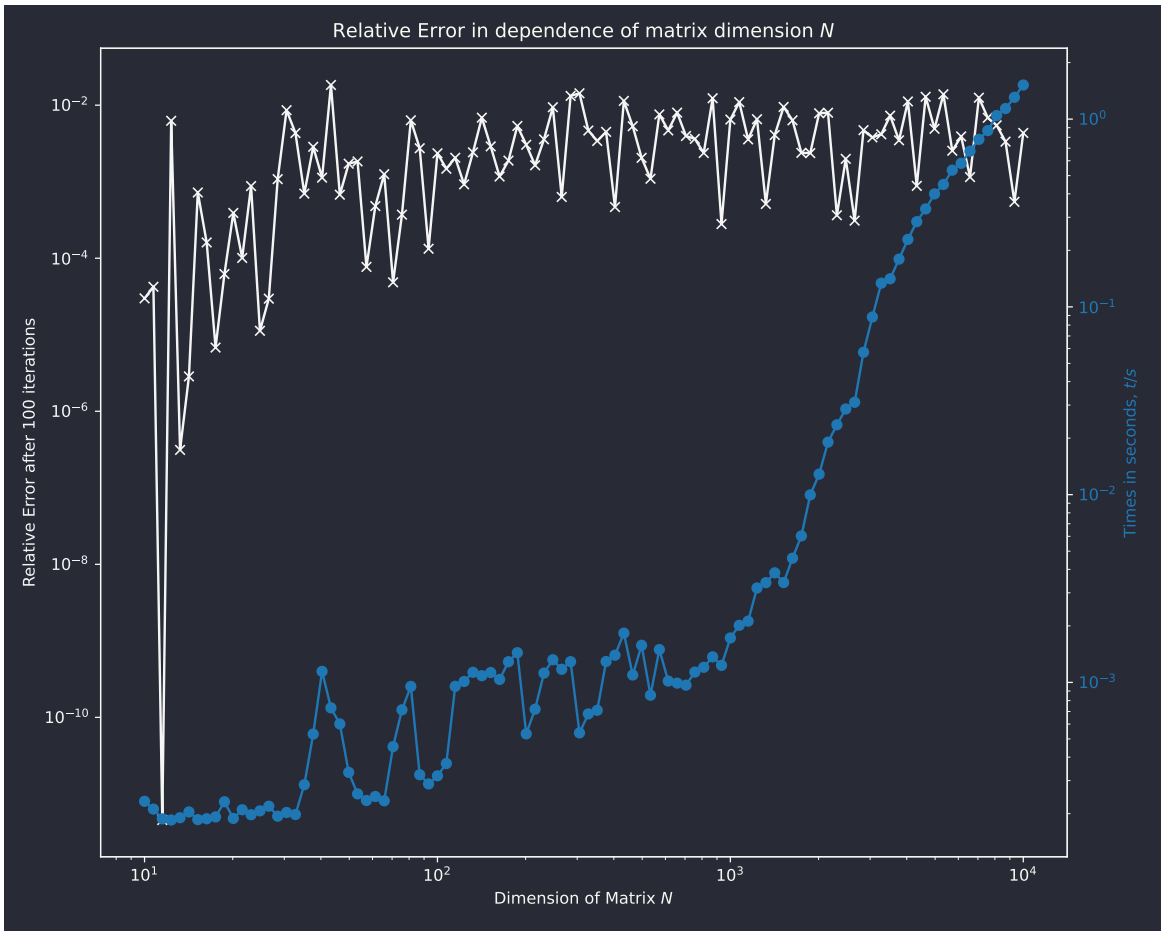

Figure 1.3: 100 Iterationsschritte für alle Matrixdimensionen. Der relative Fehler pendelt sich sehr schnell bei etwa 10^{-2} ein, unabhängig von der Größe der Matrix. Die Präzision aus Abbildung 1.2 wird bis auf einen Ausreißer nicht erreicht. Trotz der nicht steigenden Genauigkeit wird aber ein **exponentieller** Anstieg in der Laufzeit beobachtet, was ohne weitere Untersuchung auf die quadratisch wachsende Natur der Werte $(N \times N)$ zurückgeführt wird.

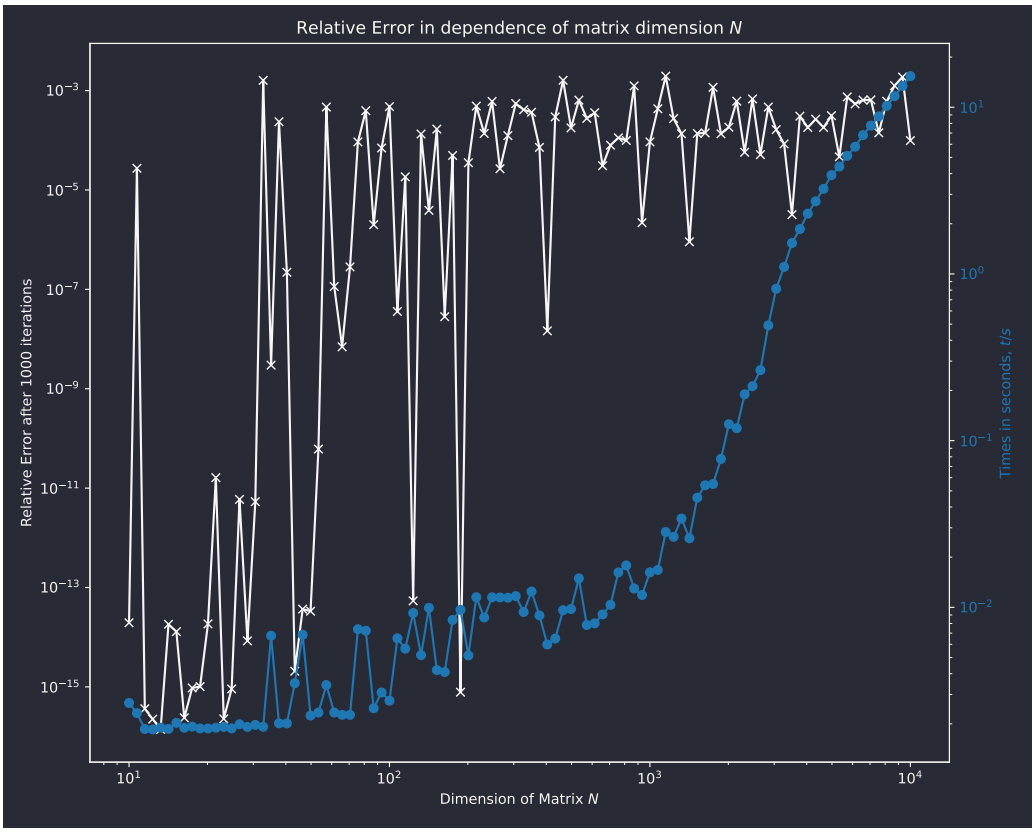

Figure 1.4: 1000 Iterationsschritte für alle Matrixdimensionen. Der relative Fehler pendelt sich wieder sehr schnell, dieses mal aber bei etwa 10^{-3} ein, unabhängig von der Größe der Matrix. Die Präzision aus Abbildung 1.2 wird (bis auf ein paar Ausreißer) noch bis 50×50-Matrizen erreicht. Auch hier wird ein exponentieller Anstieg der Laufzeit beobachtet. Unter grober Beobachtung wird auch festgestellt, dass die Laufzeit linear mit den Iterationsschritten ansteigt, was zu erwarten ist, da der Rechenaufwand pro Iteration unverändert bleibt.

Fazit

Die Potenzmethode konvergiert schnell und recht zuverlässig gegen die gesuchten Eigenwerte und -vektoren.

Auf den ersten Blick kann der relative Fehler einer hochdimensionalen Matrix reziprok mit einer Vergößerung der Iterationsschritte und damit einer Verlängerung der Laufzeit abgefangen werden (eine Zehnfachsteigerung der Iterationsschritte, und somit auch der Laufzeit, resultiert in einer Zehntelreduzierung des relativen Fehlers.).

Es müsste aber weiter untersucht werden, unter welchen Bedingungen und wie weitreichend diese Aussage gültig ist.

Es muss vor allem beachtet werden, dass die Konvergenzgeschwindigkeit des relativen Fehlers auch sehr stark von dem Verhältnis der beiden betragsmäßig größten Eigenwerte abhängig ist. Würde dieser Faktor mit einberechnet werden, könnte man eine einheitlichere Aussage über die Konvergenz- und Laufzeitabhängigkeiten treffen.

E1: Householder with QR Iteration

Task:

In the following, you are asked to become familiar with the Householder algorithm and the QR iteration. Using this method, it is possible to efficiently diagonalize even large matrices.
Given the symmetric $\mathrm{N} \times \mathrm{N}$ matrix:

$$A_{k,l} = k + l + k\delta_{k,l} \quad 0 \leq k, l < N.$$

(a) Use Householder transformations to transform matrix A into tridiagonal form.

(b) Determine the eigenvalues of A using QR iteration.

(c) Diagonalize matrix A also using the power method that you implemented in Task 1. Compare the results and runtimes.

Do this for $N = 10, N = 100$ and $N = 1000$.

Diese Aufgabe ist mit großem Frust verbunden, denn es scheint als würde es nur einen marginalen Fehler in dem Algorithmus geben. Daher ist die Auswertung für c) leider nicht möglich; wenn auch Laufzeiten in E0 bereits untersucht wurden.

a) Die Householder-Transformation überführt **reelle und symmetrische** Matrizen auf eine **Tridiagonalform**. Besonders an diesem Algorithmus ist, dass er nach einer festen Anzahl an Schritten ($N-2$ mit $\boldsymbol{A} \in \mathbb{R}^{N \times N}$) beendet ist.

Ganz allgemein gilt

$$\underline{\underline{P}}_1^T \cdot \underline{\underline{A}} \cdot \underline{\underline{P}}_1 = \left(\begin{array}{c|ccc} a_{11} & k_1 & & 0 \\ \hline k_1 & a'_{22} & a'_{23} \ \dots & a'_{2N} \\ & a'_{32} & & a'_{3N} \\ 0 & \vdots & & \vdots \\ & a'_{N2} & \dots & a'_{NN} \end{array}\right)$$

mit

$$\underline{\underline{P}}_n = \left(\begin{array}{c|c} \underline{\underline{I}}_{n \times n} & 0 \\ \hline 0 & \underline{\underline{S}}_n \end{array}\right),$$

$$\underline{\underline{S}}_n = \underline{\underline{1}} - 2\boldsymbol{u}_n \otimes \boldsymbol{u}_n,$$

$$\underline{\underline{P}}_1^T \cdot \underline{\underline{A}} \cdot \underline{\underline{P}}_1 = \left(\begin{array}{c|c} a_{11} & \left(\underline{\underline{S}}_1 \cdot \boldsymbol{v}\right)^T \\ \hline \underline{\underline{S}}_1 \cdot \boldsymbol{v} & \underline{\underline{A}}' \end{array}\right) \quad \text{mit} \quad \boldsymbol{v} \equiv \begin{pmatrix} a_{21} \\ \vdots \\ a_{N1} \end{pmatrix},$$

$$k_1 = \pm|\boldsymbol{v}|$$

und

$$\boldsymbol{u}_1 = \frac{\boldsymbol{v} - k_1 \boldsymbol{e}_1}{|\boldsymbol{v} - k_1 \boldsymbol{e}_1|} \quad .$$

Es wird dann $\underline{\underline{A}}' = \underline{\underline{P}}_1^T \cdot \underline{\underline{A}} \cdot \underline{\underline{P}}_1$ berechnet und die Schritte mit $\underline{\underline{A}}'$ wiederholt, bis $n = N - 2$.

Unklarheiten

In der Aufgabenstellung ist die Formel $A_{k,l} = k + l + k\delta_{k,l} \quad 0 \leq k, l < N$ gegeben. Für gewöhnlich fangen die Indizes in der Matrix-Notation bei a_{11} an. Da jedoch $0 \leq k, l < N$ angegeben ist, scheint die Notation in der Tat bei a_{00} anzufangen.

Dies ist später insofern relevant, als dass die QR-Zerlegung den Wert a_{00} verwendet und durch ihn teilt. Dies kann umgangen werden, indem die Matrix doppelt gespiegelt wird, sodass der Wert an die Stelle a_{NN} wandert. Dieser wird in der Berechnung nämlich übersprungen, da für die P-Matrizen nur bis $n = N - 1$ iteriert wird. Die Eigenwerte sollten bei dieser Spiegelung unverändert bleiben, da diese ebenfalls durch orthogonale Matrizen beschrieben wird. Bei solch einer Ähnlichkeitstransformation bleiben die Eigenwerte unberührt.

Die Rechnungen wurden mit beiden Varianten (a_{11} und a_{00}, also verschiedene Startwerte für die Gleichung) durchgeführt. Es wird auch erfolgreich (via Householder) eine Tridiagonalmatrix berechnet. Doch konvergiert diese in der QR-Zerlegung dann leider nicht gegen eine Diagonalmatrix.

Die berechnete Tridiagonalmatrix lautet:

$$\begin{pmatrix} 0 & -17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -17 & 97 & 22 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 22 & 3 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 5 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 5 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 5 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 5 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 5 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 5 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 5 \end{pmatrix}$$

(Ergebnisse sind stark gerundet. Im Algorithmus wird ungerundet weitergerechnet).

b) Der Algorithmus ist vermutlich bis auf einen kleinen Fehler vollständig implementiert. Der Code ist in zwei Klassen aufgeteilt und sollte dadurch (hoffentlich) sehr lesbar sein. Ich habe nun nach vielen Stunden Debugging aufgegeben, da ich ahne, dass der Fehler eher in den Formeln oder dem allgemeinen Algorithmus liegt. Ich ziehe aus der Fehlersuche leider keinen Mehrwert, auch wenn ich die Aufgabe sehr gerne vervollständigt hätte.

Hier sind die verwendeten Formeln.

Ich habe alles aus dem Kierfeld-Skript (vielleicht liegt da der Fehler):

$$\underline{\underline{A}} = \begin{pmatrix} a_{11} & k_1 & & & 0 & \\ k_1 & a_{22} & k_2 & & & \\ & k_2 & a_{33} & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & k_{N-1} \\ & 0 & & & k_{N-1} & a_{NN} \end{pmatrix}$$

$$\underline{\underline{Q}}^t = \underline{P}_{N-1\,N} \cdot \ldots \cdot \underline{P}_{23} \cdot \underline{P}_{12}$$

$$\underline{P}_{12} \begin{pmatrix} a_{11} \\ k_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} r_{11} \\ 0 \\ \vdots \\ \vdots \\ 0 \end{pmatrix} \Longleftrightarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_{11} \\ k_1 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} r_{11} \\ 0 \end{pmatrix}$$

$$\Rightarrow t = \frac{s}{c} = \frac{k_1}{a_{11}}$$

$$c = \frac{1}{\sqrt{t^2+1}}, \quad s = tc.$$

Mit dem Ziel: $\underline{\underline{Q}}^t \cdot \underline{\underline{A}} \cdot \underline{\underline{Q}} = \underline{\underline{A'}} =$ QR-Trafo von $\underline{\underline{A}}$

Allgemeine Form der Drehmatrizen, wobei $\cos$ und $\sin$ entsprechend c und s sind:

$$\underline{\underline{P}}_{pq} \equiv \begin{pmatrix} 1 & & & & & & & 0 \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & \cos\phi & \dots & \sin\phi & & \\ & & & & \ddots & & & \\ & & & -\sin\phi & \dots & \cos\phi & & \\ & & & & & & 1 & \\ & & 0 & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}$$

Und dann allgemein:

$$t = \frac{s}{c} = \frac{k_n}{a_{nn}}$$

$$c = \frac{1}{\sqrt{t^2+1}}, \quad s = tc.$$