Assignment 3 - Numerical Linear Algebra

Arthur Rabello Oliveira[1]

28/05/2025

Abstract

We design and test a function `to_hessemberg(A)` that reduces an arbitrary square matrix to (upper) Hessenberg form with Householder reflectors, returns the reflector vectors, the compact Hessenberg matrix H, and the accumulated orthogonal factor Q, verifying numerically that $A = QHQ^*$ and $Q^*Q = I$ for symmetric and nonsymmetric inputs of orders $10 - 10000$. Timings confirm the expected $O(\text{something})$ cost and reveal the $2\times$ speed-up attainable for symmetric matrices through trivial bandwidth savings. Leveraging this routine, we investigate the spectral structure of orthogonal matrices: we show that all eigenvalues lie on the unit circle, analyse the consequences for the power method and inverse iteration, and obtain a closed-form spectrum for generic 2×2 orthogonals. Random 4×4 orthogonal matrices generated via QR factorisation are then reduced to Hessenberg form; the eigenvalues of their trailing 2×2 blocks are computed analytically and reused as fixed shifts in the QR iteration, where experiments demonstrate markedly faster convergence. Throughout, every algorithm is documented and supported by commented plots that corroborate the theoretical claims.

Contents

[1] Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: arthur.oliveira.1@fgv.edu.br

1. Introduction

One could calculate the eigenvalues of a square matrix using the following algorithm:

1. Compute the n-th degree polynomial $\det(A - \lambda I) = 0$,
2. Solve for λ (somehow).

On step 2, the eigenvalue problem would have been reduced to a polynomial root-finding problem, which is awful and extremely ill-conditioned. From the previous assignment we know that in the denominator of the relative condition number $\kappa(x)$ there's a $|x - n|$. So $\kappa(x) \to \infty$ when $x \to 0$. As an example, consider the polynomial

$$\begin{gathered} p(x) = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 \\ -4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512 \end{gathered} \tag{1}$$

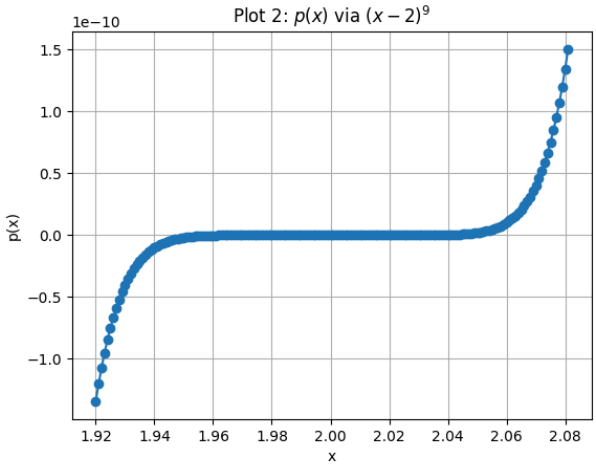

Figure 1: $p(x)$ via the coefficients in eq. (1)

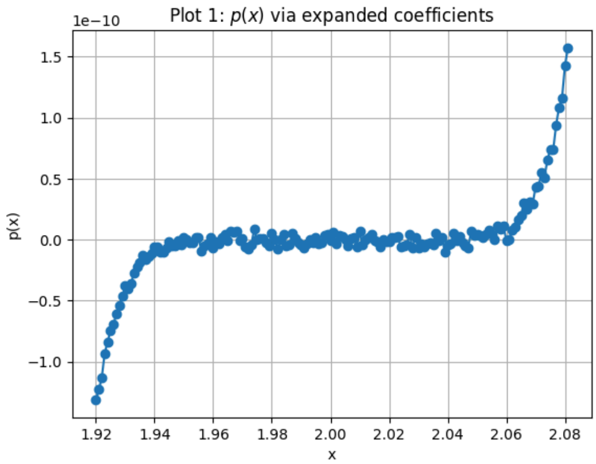

Figure 2: $p(x)$ via $(x-2)^9$

Figure 2 shows a smooth curve, while Figure 1 shows a weird oscillation around $x = 0$ (And pretty much everywhere else if the reader is sufficiently persistent).

This is due to the round-off errors when $x \approx 0$ and the big coefficients of the polynomial. In general, polynomial are very sensitive to perturbations in the coefficients, which is why rootfinding is a bad idea to find eigenvalues.

Here we discuss aspects of some iterative eigenvalue algorithms, such as power iteration, inverse iteration, and QR iteration.

2. Hessemberg Reduction (Problem 1)

2.1. Calculating the Householder Reflectors (a)

The following function calculates the Householder reflectors that reduce a matrix to Hessenberg form. It returns the reflector vectors, the compact Hessenberg matrix H, and the accumulated orthogonal factor Q.

```python
import numpy as np
import time
from typing import List, Tuple


def build_householder_unit_vector(
        target_vector: np.ndarray
) -> np.ndarray:

    """
    Builds a Householder unit vector

    Args:
        1. target_vector (np.ndarray): Column vector that we want to annihilate (size ≥ 1).

    Returns:
        np.ndarray:
            The normalised Householder vector (||v||₂ = 1) with a real first component.

    Raises:
        1. ValueError: If 'target_vector' has zero length.
    """

    if target_vector.size == 0:
        raise ValueError("The target vector is empty; no reflector needed.")

    vector_norm: float = np.linalg.norm(target_vector)

    if vector_norm == 0.0: #nothing to annihilate — return canonical basis vector
        householder_vector: np.ndarray = np.zeros_like(target_vector)
        householder_vector[0] = 1.0
        return householder_vector

    sign_correction: float = (
```

```python
            1.0 if target_vector[0].real >= 0.0 else -1.0
        )
        copy_of_target_vector: np.ndarray = target_vector.copy()
        copy_of_target_vector[0] += sign_correction * vector_norm
        householder_vector: np.ndarray = (
            copy_of_target_vector / np.linalg.norm(copy_of_target_vector)
        )
        return householder_vector


    def to_hessenberg(
            original_matrix: np.ndarray,
    ) -> Tuple[List[np.ndarray], np.ndarray, np.ndarray]:

        """
        Reduce 'original_matrix' to upper Hessenberg form by Householder reflections.

        Args
            1. original_matrix (np.ndarray): Real or complex square matrix of order 'matrix_order'.

        Returns
            Tuple consisting of:

            1. householder_reflectors_list (List[np.ndarray])
            2. hessenberg_matrix (np.ndarray)
            3. accumulated_orthogonal_matrix (np.ndarray)  s.t.
              original_matrix = Q · H · Qᴴ

        Raises
            1. ValueError: If 'original_matrix' is not square.
        """

        working_matrix: np.ndarray = np.asarray(original_matrix).copy()

        if working_matrix.shape[0] != working_matrix.shape[1]:
            raise ValueError("Input matrix must be square.")

        matrix_order: int = working_matrix.shape[0]
        accumulated_orthogonal_matrix: np.ndarray = np.eye(
            matrix_order, dtype=working_matrix.dtype
        )
        householder_reflectors_list: List[np.ndarray] = []

```

```python
    for column_index in range(matrix_order - 2): #extract the part of column 'column_index' that we want to zero out
        target_column_segment: np.ndarray = working_matrix[
            column_index + 1 :, column_index
        ]

        householder_vector: np.ndarray = build_householder_unit_vector(
            target_column_segment
        )  #build Householder vector for this segment
        householder_reflectors_list.append(householder_vector)

        #expand it to the full matrix dimension
        expanded_householder_vector: np.ndarray = np.zeros(
            matrix_order, dtype=working_matrix.dtype
        )
        expanded_householder_vector[column_index + 1 :] = householder_vector


        working_matrix -= 2.0 * np.outer(
            expanded_householder_vector,
            expanded_householder_vector.conj().T @ working_matrix,
        ) #apply reflector from BOTH sides
        working_matrix -= 2.0 * np.outer(
            working_matrix @ expanded_householder_vector,
            expanded_householder_vector.conj().T,
        )

        #accumulate Q
        accumulated_orthogonal_matrix -= 2.0 * np.outer(
            accumulated_orthogonal_matrix @ expanded_householder_vector,
            expanded_householder_vector.conj().T,
        )

    hessenberg_matrix: np.ndarray = working_matrix
    return (
        householder_reflectors_list,
        hessenberg_matrix,
        accumulated_orthogonal_matrix,
    )
```

We will evaluate this function in Section 2.2.

2.2. Evaluating the Function (b), (c), (d)

We present another algorithm for evaluating the function `to_hessenberg(A)` for random matrices of various sizes, inputed by the user, which also gets to choose if symmetric matrices will be generated or not.

```python
import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt
import math
from IPython.display import display, Markdown
from ast import literal_eval

#RANDOM MATRIX GENERATOR
def generate_random_matrix(n:int, distribution:str="normal",
                           symmetric:bool=False, seed:int|None=None):
    rng = np.random.default_rng(seed)
    if distribution == "normal":
        A = rng.standard_normal((n, n))
    elif distribution == "uniform":
        A = rng.uniform(-1.0, 1.0, size=(n, n))
    else:
        raise ValueError("distribution must be 'normal' or 'uniform'")
    return (A + A.T) / 2.0 if symmetric else A


#REFLECTOR CALCULATOR
def _house_vec(x:np.ndarray) -> np.ndarray:

    """
    Builds a Householder reflector for a given column vector x.
    Args:
        x (np.ndarray): Column vector to be transformed.
    Returns:
        np.ndarray: Normalised Householder vector with a real first component.
    Raises:
        None
    """

    sigma = np.linalg.norm(x)
    if sigma == 0.0:
        e1 = np.zeros_like(x)
        e1[0] = 1.0
        return e1
    sign = 1.0 if x[0].real >= 0.0 else -1.0
    v = x.copy()
    v[0] += sign * sigma
    return v / np.linalg.norm(v)

def hessenberg_reduction(A_in:np.ndarray, symmetric:bool=False,
accumulate_q:bool=True):
```

```python

    """
    Reduces a matrix to upper Hessenberg form using Householder reflections.
    Args:
        A_in (np.ndarray): Input matrix to be reduced.
        symmetric (bool): If True, treat the matrix as symmetric and reduce to tridiagonal form.
        accumulate_q (bool): If True, accumulate the orthogonal matrix Q.
    Returns:
        Tuple[np.ndarray, np.ndarray]: The reduced matrix in Hessenberg form and the orthogonal matrix Q.
    Raises:
        None
    """

    A = A_in.copy()
    n = A.shape[0]
    Q = np.eye(n, dtype=A.dtype)

    if not symmetric:    #GENERAL caSe
        for k in range(n-2):
            v = _house_vec(A[k+1:, k])
            w = np.zeros(n, dtype=A.dtype)
            w[k+1:] = v
            A -= 2.0 * np.outer(w, w.conj().T @ A)
            A -= 2.0 * np.outer(A @ w, w.conj().T)
            if accumulate_q:
                Q -= 2.0 * np.outer(Q @ w, w.conj().T)
        return A, Q

    #SYMMETRIC TRIDIAGONAL CASE
    for k in range(n-2):
        x = A[k+1:, k]
        v = _house_vec(x)
        beta = 2.0

        w = A[k+1:, k+1:] @ v   #trailing submatrix rank-2 update (A ← A − v wᵀ − w vᵀ)
        tau = beta * 0.5 * (v @ w)
        w -= tau * v
        A[k+1:, k+1:] -= beta * np.outer(v, w) + beta * np.outer(w, v)

        new_val = -np.sign(x[0]) * np.linalg.norm(x)   #store the single sub-diagonal element, zero the rest
        A[k+1, k] = new_val
        A[k, k+1] = new_val
```

```python
        A[k+2:, k] = 0.0
        A[k, k+2:] = 0.0

        if accumulate_q:  #accumulate Q if requested
            Q[:, k+1:] -= beta * np.outer(Q[:, k+1:] @ v, v)

    A = np.triu(A) + np.triu(A, 1).T  #force symmetry
    return A, Q


#VERIFYING PART
def verify_factorisation_once(n:int, dist:str, symmetric:bool, seed:int|None):

    """
    Verifies the factorisation of a random matrix of size n.
    Args:
        n (int): Size of the matrix.
        dist (str): Distribution type ('normal' or 'uniform').
        symmetric (bool): Whether the matrix is symmetric.
        seed (int | None): Random seed for reproducibility.
    Returns:
        None
    Raises:
        None
    """

    A = generate_random_matrix(n, dist, symmetric, seed)
    T, Q = hessenberg_reduction(A, symmetric=symmetric)
    res_fact = np.linalg.norm(A - Q @ T @ Q.T)
    res_orth = np.linalg.norm(Q.T @ Q - np.eye(n))
    colour = "green" if res_fact < 1e-11 else "red"
    typ = "symmetric" if symmetric else "general"
    display(Markdown(
        f"**{n}×{n} {typ}**  \n"
        f"<span style='color:{colour}'>‖A − Q T Qᵀ‖ = {res_fact:.2e}</span> \n"
        f"‖QᵀQ − I‖ = {res_orth:.2e}"
    ))


def benchmark_hessenberg(size_list, dist:str, mode:str, seed:int|None, reps_small:int=5):

    """
    Benchmark the Hessenberg reduction for various matrix sizes and types.
    Args:
```

```
        size_list (list of int): List of matrix sizes to test.
        dist (str): Distribution type ('normal' or 'uniform').
        mode (str): Matrix type ('general', 'symmetric', or 'both').
        seed (int | None): Random seed for reproducibility.
        reps_small (int): Number of repetitions for small matrices.
    Returns:
        pd.DataFrame: DataFrame containing the benchmark results.
    Raises:
        None
    """

    records = []
    for n in size_list:
        for sym in ([False, True] if mode=="both" else [mode=="symmetric"]):
            A = generate_random_matrix(n, dist, sym, seed)

            t0 = time.perf_counter()
            hessenberg_reduction(A, symmetric=sym, accumulate_q=False)
            probe = time.perf_counter() - t0
            reps = reps_small if probe*reps_small >= 1.0 else math.ceil(1.0 / probe)

            times = []
            for _ in range(reps):
                start = time.perf_counter()
                hessenberg_reduction(A, symmetric=sym, accumulate_q=False)
                times.append(time.perf_counter() - start)

            records.append(dict(size=n,
                                type="symmetric" if sym else "general",
                                reps=reps,
                                avg=np.mean(times)))

    df = pd.DataFrame(records)
    display(df.style.format({"avg":"{:.3e}"}).hide(axis="index"))

    plt.figure(figsize=(7,5))
    mark = {"general":"o", "symmetric":"s"}
    for label, sub in df.groupby("type"):
        plt.loglog(sub["size"], sub["avg"], marker=mark[label], ls="-", label=label)
        if len(sub) > 1:
            a,b = np.polyfit(np.log10(sub["size"]), np.log10(sub["avg"]), 1)
            plt.loglog(sub["size"], 10**(b+a*np.log10(sub["size"])),
                       "--", label=f"{label} fit ~ $n^{a:.2f}$")
    plt.xlabel("matrix size  (log)")
```

```python
    plt.ylabel("runtime [s]  (log)")
    plt.title("Hessenberg (general)  vs  Tridiagonal (symmetric)")
    plt.grid(True, which="both", ls=":")
    plt.legend(); plt.tight_layout(); plt.show()
    return df


#===INTERACTIVE PART=========================================================
try:
    raw = input("\nMatrix sizes (Python list) (e.g): [64,128,256,512,1024]: ")
    sizes = literal_eval(raw) if raw.strip() else [64,128,256,512,1024]
except Exception:
    print("Bad list -> using default.")
    sizes = [64,128,256,512,1024]

dist = input("Distribution ('normal'/'uniform')  [normal]: ").strip().lower() or "normal"
mode_txt = input("Matrix type g=general, s=symmetric, b=both  [g]: ").strip().lower() or "g"
mode = "symmetric" if mode_txt=="s" else "both" if mode_txt=="b" else "general"
seed_txt = input("Random seed (None/int) [None]: ").strip()
seed_val = None if seed_txt.lower() in {"", "none"} else int(seed_txt)

for sym in ([False, True] if mode=="both" else [mode=="symmetric"]): #accuracy on largest size
    verify_factorisation_once(max(sizes), dist, sym, seed_val)

benchmark_hessenberg(sizes, dist, mode, seed_val) #timings
```

The reader should be aware that my poor Dell Inspiro 5590 has crashed precisely 5 times while i was writing this. The runtime was around 4 minutes for a matrix $A \approx 10^3 \times 10^3$.

An expected output is:

```python
64×64 general
‖A − Q T Qᵀ‖ = 7.51e-14
‖QᵀQ − I‖ = 7.07e-15

64×64 symmetric
‖A − Q T Qᵀ‖ = 4.83e-14
‖QᵀQ − I‖ = 7.39e-15

128×128 general
‖A − Q T Qᵀ‖ = 1.84e-13
‖QᵀQ − I‖ = 1.26e-14

128×128 symmetric
‖A − Q T Qᵀ‖ = 1.14e-13
```

```
||QᵀQ − I|| = 1.25e-14

256×256 general
||A − Q T Qᵀ|| = 4.70e-13
||QᵀQ − I|| = 2.28e-14

256×256 symmetric
||A − Q T Qᵀ|| = 2.78e-13
||QᵀQ − I|| = 2.25e-14

512×512 general
||A − Q T Qᵀ|| = 1.16e-12
||QᵀQ − I|| = 4.10e-14

512×512 symmetric
||A − Q T Qᵀ|| = 7.10e-13
||QᵀQ − I|| = 4.09e-14

1024×1024 general
||A − Q T Qᵀ|| = 3.05e-12
||QᵀQ − I|| = 7.57e-14

1024×1024 symmetric
||A − Q T Qᵀ|| = 1.84e-12
||QᵀQ − I|| = 7.64e-14
```

As n grows, we observe that the residuals also grow, but still in machine precision. The difference between the symmetric and nonsymmetric cases are more pronounced in larger matrices.

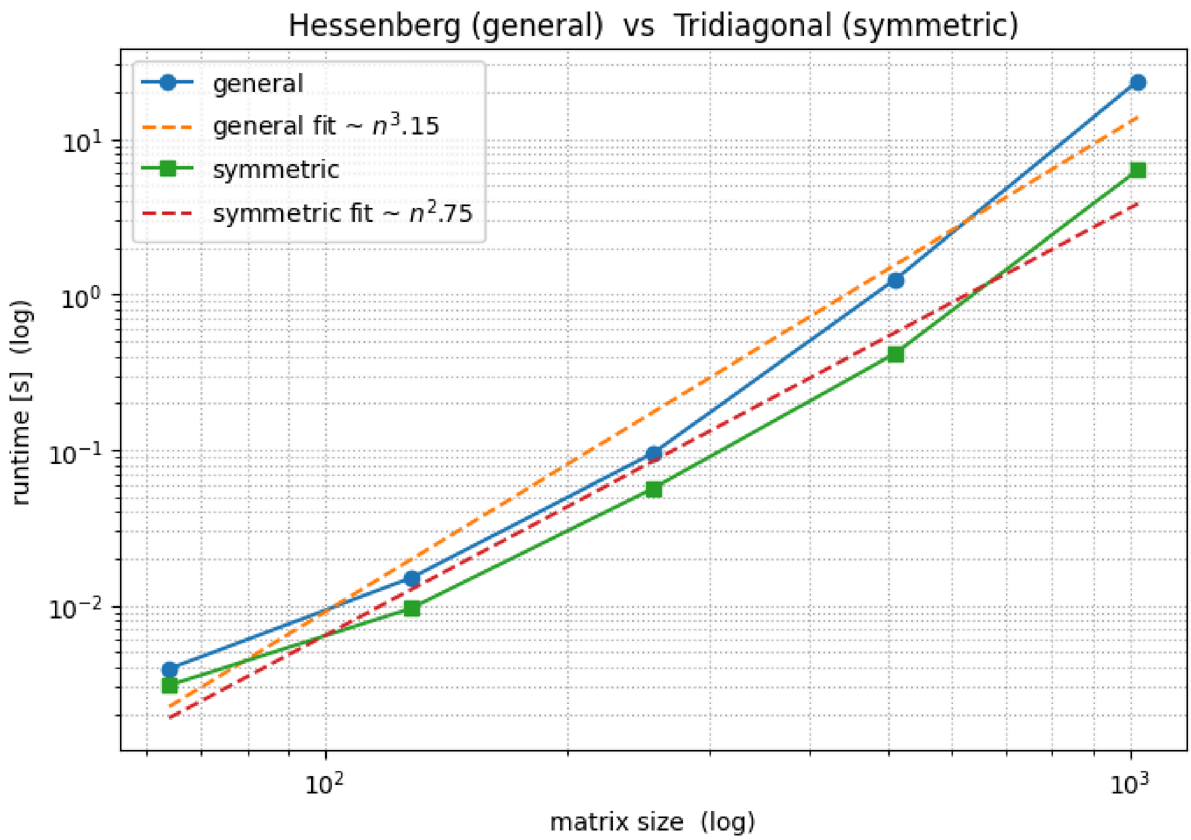

Figure 3: Runtime of the Hessenberg reduction for ordinary and symmetric matrices

2.2.1. Complexity (c)

Figure 3 shows the expected $O(n^3)$ complexity for the general case and $O(n^2)$ for the symmetric case. The latter is better discussed in Section 2.2.2.

To understand why the complexity is $O(n^3)$ in the general case, we can look at the algorithm. The outer loop runs $n-2$ times, and inside it, we have two matrix-vector products and two outer products, which are all $O(n^2)$. Thus, the total complexity is $O(n^3)$.

2.2.2. The Symmetric Case (d)

On the symmetric case we know that reflectors will be applied in only one side of the matrix, since $v^T A = Av^T$. That is precisely what the function `generate_random_matrix` does. Which cuts complexity from the expected $O(n^3)$ seen in the previous section to a $O(n^2)$[2].

3. Eigenvalues and Iterative Methods

3.1. Power iteration

The power iteration consists on computing large powers of the sequence:

$$\frac{x}{\|x\|}, \frac{Ax}{\|Ax\|}, \frac{A^2x}{\|A^2x\|}, ..., A \in \mathbb{C}^{m \times m} \tag{2}$$

To see why this sequence converges(under good assumptions), let A be diagonalizable. And write:

[2]See page 194 of Trefethen & Bau's Numerical Linear Algebra book

$$x = \sum_{i=1}^{m} \varphi_i v_i \tag{3}$$

In a basis of eigenvectors v_i with respective eigenvalues λ_i. Then for $x \in \mathbb{C}^m$ we have:

$$Ax = \sum_{i=1}^{m} \lambda_i \varphi_i v_i \tag{4}$$

Or even better:

$$A^n x = \sum_{i=1}^{m} \lambda_i^n \varphi_i v_i \tag{5}$$

Let v_j be the eigenvector associated to the biggest eigenvalue λ_j, then we have:

$$A^n x = \frac{1}{\lambda_j^n} \cdot \sum_{i=1}^{m} \lambda_i^n \varphi_i v_i = \frac{\lambda_1^n}{\lambda_j^n} \varphi_1 v_1 + \ldots + \varphi_j v_j + \ldots + \frac{\lambda_m^n}{\lambda_j^n} \varphi_m v_m \tag{6}$$

When $n \to \infty$ all of the smaller $\frac{\lambda_k}{\lambda_j}$ will approach 0, so we have:

$$\lim_{n\to\infty} A^n x = \varphi_j v_j \tag{7}$$

So the denominator on the original expression becomes

$$\|A^n x\| = \|\varphi_j v_j\| = |\varphi_j| \|v_j\| \tag{8}$$

And the limit is:

$$\lim_{n\to\infty} \frac{A^n x}{\|A^n x\|} = \frac{\varphi_j v_j}{|\varphi_j| \|v_j\|} \tag{9}$$

Since $\frac{\varphi_j}{|\varphi_j|} = \pm 1$, the sequence converges to $\pm v_j$ uga buga

3.2. Inverse Iteration

Consider $\mu \in \mathbb{R} \setminus \Lambda$, where Λ is the set of eigenvalues of A. The eigenvalues $\hat{\lambda}$ of $(A - \mu I)^{-1}$ are:

$$\begin{gathered} \det\left(A - \mu I - \hat{\lambda} I\right) = 0 \Leftrightarrow \det\left(A - \left(\mu + \hat{\lambda}\right) I\right) = 0 \\ \Leftrightarrow \hat{\lambda}_j = \frac{1}{\lambda_j - \mu} \end{gathered} \tag{10}$$

Where λ_j are the eigenvalues of A. So if μ is close to an eigenvalue, then $\hat{\lambda}$ will be large. Power iteration seems interesting here, so the sequence:

$$\frac{x}{\|x\|}, \frac{(A - \mu I)^{-1} x}{\|(A - \mu I)^{-1} x\|}, \frac{(A - \mu I)^{-2} x}{\|(A - \mu I)^{-2} x\|}, \ldots \tag{11}$$

Converges to the eigenvector associated to the eigenvalue $\hat{\lambda}$.

3.3. QR Iteration

3.4. QR Iteration with Shifts

4. Orthogonal Matrices (Problem 2) (a)

Here we will discuss how orthogonal matrices behave when we appluy the iterations discussed in Section 3.1, Section 3.2 and Section 3.4.

So let $Q \in \mathbb{C}^{m \times n}$ be an orthogonal matrix. We are interested in its eigenvalues λ. We know that:

$$\begin{aligned} Qx = \lambda x \Leftrightarrow x^T Q x = \lambda x^T x \\ \Leftrightarrow Q\langle x, x\rangle = \lambda \langle x, x\rangle \end{aligned} \tag{12}$$

Since Q preserves inner product, we have:

$$\begin{aligned} Q\langle x, x\rangle = \lambda\langle x, x\rangle \Leftrightarrow \langle x, x\rangle = \lambda\langle x, x\rangle \\ \Leftrightarrow |\lambda| = 1 \end{aligned} \tag{13}$$

So λ lies in the unit circle, i.e $\lambda = e^{i\varphi}, \varphi \in \mathbb{R}$. We now discuss how this affects efficiency of some iterative methods

4.1. Orthogonal Matrices and the Power Iteration

The power method is better discussed in Section 3.1. Here we will write straight forward the result:

$$Q^n x = \frac{1}{\lambda_j^n} \cdot \sum_{i=1}^{m} \lambda_i^n \varphi_i v_i \tag{14}$$

Where λ_i are the eigenvalues of Q, φ_i are the coefficients of the expansion of x in the basis of eigenvectors v_i. Since we have that $|\lambda_i| = 1$, we have:

The fact that $|\lambda_i| = 1 \Rightarrow |\lambda_i^n| = 1$ is sufficiently enough for one to be convinced that power iteration does not converge.

Let $\lambda_k = e^{i\psi_k}$, where $\psi_k \in \mathbb{R}$. Then expanding eq. (14):

$$Q^n x = \frac{1}{e^{i\psi_j \cdot n}} \cdot \sum_{\tau=1}^{m} e^{i\psi_\tau n} \varphi_\tau v_\tau \tag{15}$$

When $n \to \infty$ if $\lambda_j = 1$ then we have:

$$Q^n x = \varphi_j v_j + \sum_{\tau \neq j} e^{i\psi_\tau n} \varphi_\tau v_\tau \tag{16}$$

Since no eigenvalue dominates other eigenvalues in the orthogonal case, usually power iteration fails.

4.2. Orthogonal Matrices and Inverse Iteration

If we apply inverse iteration to an orthogonal matrix with a shift μ, we have:

$$\begin{aligned} \det\left(Q - \mu I - \hat{\lambda} I\right) = 0 \Leftrightarrow \det\left(Q - \left(\mu + \hat{\lambda}\right) I\right) = 0 \\ \Leftrightarrow \hat{\lambda}_j = \frac{1}{\lambda_j - \mu} \end{aligned} \tag{17}$$

We know that the eigenvalues of Q are on the unit circle, so if μ is close to an eigenvalue λ_j, $\hat{\lambda_j}$ will be huge (dominant), which makes power iteration converge to the eigenvector associated to $\hat{\lambda_j}_j$, which is the eigenvector associated to λ_j. The fact that the eigenvalues are on the unit circle also contributes to the convergence of the method.

So we concude that inverse iteration works well for orthogonal matrices, *if μ is close to an eigenvalue of Q*.

4.3. The 2×2 Case (b)

We will calculate the eigenvalues of:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{18}$$

With $a, b, c, d \in \mathbb{R}$. The characteristic polynomial gives us:

$$\begin{gathered} \det(A - \lambda I) = 0 \Leftrightarrow \det \begin{pmatrix} a - \lambda & b \\ c & d - \lambda \end{pmatrix} = 0 \\ \Leftrightarrow (a - \lambda)(d - \lambda) - bc = 0 \Leftrightarrow \lambda^2 + \lambda(-a - d) + (ad - bc) = 0 \\ \Leftrightarrow \lambda = (a + d) \pm \frac{\sqrt{(a + d)^2 - 4(ad - bc)}}{2} \end{gathered} \tag{19}$$

So the eigenvalues are:

$$\begin{aligned} \lambda_1 &= \frac{a + d + \sqrt{(a + d)^2 - 4(ad - bc)}}{2} \\ \lambda_2 &= \frac{a + d - \sqrt{(a + d)^2 - 4(ad - bc)}}{2} \end{aligned} \tag{20}$$

4.4. Random Orthogonal Matrices (c)

4.5. Orthogonal Matrices and QR Iteration With A Specific Shift

4.6. Shift With an Eigenvalue (d)

5. Conclusion

Bibliography