Assignment 3 - Numerical Linear Algebra

Arthur Rabello Oliveira[1]

27/05/2025

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. <u>From the previous assignment</u> 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

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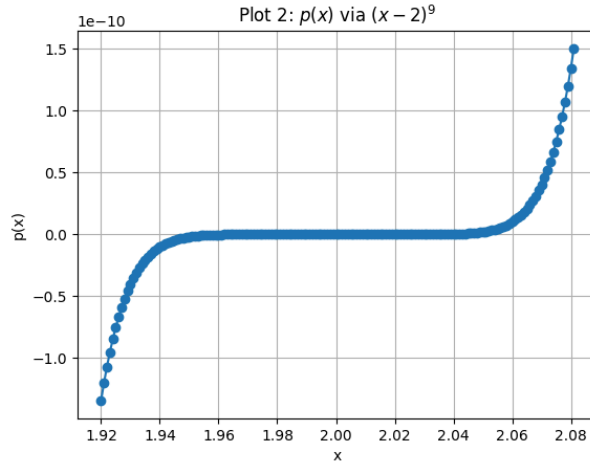$$-4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512 \tag{1}$$

Figure 1: $p(x)$ via the coefficients in <u>eq. (1)</u>
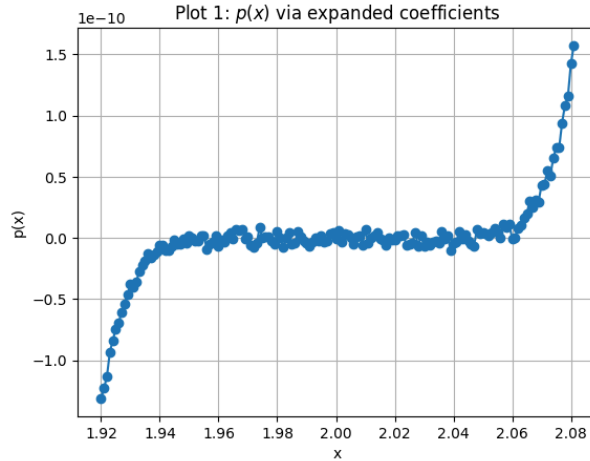
Figure 2: $p(x)$ via $(x - 2)^9$

<u>Figure 2</u> shows a smooth curve, while <u>Figure 1</u> 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
35            1.0 if target_vector[0].real >= 0.0 else -1.0
36        )
37        copy_of_target_vector: np.ndarray = target_vector.copy()
38        copy_of_target_vector[0] += sign_correction * vector_norm
39        householder_vector: np.ndarray = (
40            copy_of_target_vector / np.linalg.norm(copy_of_target_vector)
41        )
42        return householder_vector


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

49        """
50        Reduce 'original_matrix' to upper Hessenberg form by Householder
          reflections.

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

55        Returns
56            Tuple consisting of:

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

63        Raises
64            1. ValueError: If 'original_matrix' is not square.
65        """

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

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

72        matrix_order: int = working_matrix.shape[0]
73        accumulated_orthogonal_matrix: np.ndarray = np.eye(
74            matrix_order, dtype=working_matrix.dtype
75        )
76        householder_reflectors_list: List[np.ndarray] = []
77
```

```python
78      for column_index in range(matrix_order - 2): #extract the part of column
        'column_index' that we want to zero out
79          target_column_segment: np.ndarray = working_matrix[
80              column_index + 1 :, column_index
81          ]
82
83          householder_vector: np.ndarray = build_householder_unit_vector(
84              target_column_segment
85          )  #build Householder vector for this segment
86          householder_reflectors_list.append(householder_vector)
87
88          #expand it to the full matrix dimension
89          expanded_householder_vector: np.ndarray = np.zeros(
90              matrix_order, dtype=working_matrix.dtype
91          )
92          expanded_householder_vector[column_index + 1 :] = householder_vector
93
94
95          working_matrix -= 2.0 * np.outer(
96              expanded_householder_vector,
97              expanded_householder_vector.conj().T @ working_matrix,
98          ) #apply reflector from BOTH sides
99          working_matrix -= 2.0 * np.outer(
100             working_matrix @ expanded_householder_vector,
101             expanded_householder_vector.conj().T,
102         )
103
104         #accumulate Q
105         accumulated_orthogonal_matrix -= 2.0 * np.outer(
106             accumulated_orthogonal_matrix @ expanded_householder_vector,
107             expanded_householder_vector.conj().T,
108         )
109
110     hessenberg_matrix: np.ndarray = working_matrix
111     return (
112         householder_reflectors_list,
113         hessenberg_matrix,
114         accumulated_orthogonal_matrix,
115     )
```

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
46
47          """
48          Reduces a matrix to upper Hessenberg form using Householder reflections.
49          Args:
50              A_in (np.ndarray): Input matrix to be reduced.
51              symmetric (bool): If True, treat the matrix as symmetric and reduce to
                    tridiagonal form.
52              accumulate_q (bool): If True, accumulate the orthogonal matrix Q.
53          Returns:
54              Tuple[np.ndarray, np.ndarray]: The reduced matrix in Hessenberg form
                    and the orthogonal matrix Q.
55          Raises:
56              None
57          """
58
59          A = A_in.copy()
60          n = A.shape[0]
61          Q = np.eye(n, dtype=A.dtype)
62
63          if not symmetric:     #GENERAL caSe
64              for k in range(n-2):
65                  v = _house_vec(A[k+1:, k])
66                  w = np.zeros(n, dtype=A.dtype)
67                  w[k+1:] = v
68                  A -= 2.0 * np.outer(w, w.conj().T @ A)
69                  A -= 2.0 * np.outer(A @ w, w.conj().T)
70                  if accumulate_q:
71                      Q -= 2.0 * np.outer(Q @ w, w.conj().T)
72              return A, Q
73
74          #SYMMETRIC TRIDIAGONAL CASE
75          for k in range(n-2):
76              x = A[k+1:, k]
77              v = _house_vec(x)
78              beta = 2.0
79
80              w = A[k+1:, k+1:] @ v    #trailing submatrix rank-2 update (A ← A − v wᵀ
                    − w vᵀ)
81              tau = beta * 0.5 * (v @ w)
82              w -= tau * v
83              A[k+1:, k+1:] -= beta * np.outer(v, w) + beta * np.outer(w, v)
84
85              new_val = -np.sign(x[0]) * np.linalg.norm(x)   #store the single sub-
                    diagonal element, zero the rest
86              A[k+1, k] = new_val
87              A[k, k+1] = new_val
```

```python
88          A[k+2:, k] = 0.0
89          A[k, k+2:] = 0.0
90
91          if accumulate_q:   #accumulate Q if requested
92              Q[:, k+1:] -= beta * np.outer(Q[:, k+1:] @ v, v)
93
94      A = np.triu(A) + np.triu(A, 1).T   #force symmetry
95      return A, Q
96
97
98  #VERIFYING PART
99  def verify_factorisation_once(n:int, dist:str, symmetric:bool, seed:int|None):
100
101      """
102      Verifies the factorisation of a random matrix of size n.
103      Args:
104          n (int): Size of the matrix.
105          dist (str): Distribution type ('normal' or 'uniform').
106          symmetric (bool): Whether the matrix is symmetric.
107          seed (int | None): Random seed for reproducibility.
108      Returns:
109          None
110      Raises:
111          None
112      """
113
114      A = generate_random_matrix(n, dist, symmetric, seed)
115      T, Q = hessenberg_reduction(A, symmetric=symmetric)
116      res_fact = np.linalg.norm(A - Q @ T @ Q.T)
117      res_orth = np.linalg.norm(Q.T @ Q - np.eye(n))
118      colour = "green" if res_fact < 1e-11 else "red"
119      typ = "symmetric" if symmetric else "general"
120      display(Markdown(
121          f"**{n}×{n} {typ}**  \n"
122          f"<span style='color:{colour}'>‖A − Q T Qᵀ‖ = {res_fact:.2e}</span>
              \n"
123          f"‖QᵀQ − I‖ = {res_orth:.2e}"
124      ))
125
126
127  def benchmark_hessenberg(size_list, dist:str, mode:str, seed:int|None,
      reps_small:int=5):
128
129      """
130      Benchmark the Hessenberg reduction for various matrix sizes and types.
131      Args:
```

```python
132            size_list (list of int): List of matrix sizes to test.
133            dist (str): Distribution type ('normal' or 'uniform').
134            mode (str): Matrix type ('general', 'symmetric', or 'both').
135            seed (int | None): Random seed for reproducibility.
136            reps_small (int): Number of repetitions for small matrices.
137        Returns:
138            pd.DataFrame: DataFrame containing the benchmark results.
139        Raises:
140            None
141        """
142
143        records = []
144        for n in size_list:
145            for sym in ([False, True] if mode=="both" else [mode=="symmetric"]):
146                A = generate_random_matrix(n, dist, sym, seed)
147
148                t0 = time.perf_counter()
149                hessenberg_reduction(A, symmetric=sym, accumulate_q=False)
150                probe = time.perf_counter() - t0
151                reps = reps_small if probe*reps_small >= 1.0 else math.ceil(1.0 /
                       probe)
152
153                times = []
154                for _ in range(reps):
155                    start = time.perf_counter()
156                    hessenberg_reduction(A, symmetric=sym, accumulate_q=False)
157                    times.append(time.perf_counter() - start)
158
159                records.append(dict(size=n,
160                                    type="symmetric" if sym else "general",
161                                    reps=reps,
162                                    avg=np.mean(times)))
163
164        df = pd.DataFrame(records)
165        display(df.style.format({"avg":"{:.3e}"}).hide(axis="index"))
166
167        plt.figure(figsize=(7,5))
168        mark = {"general":"o", "symmetric":"s"}
169        for label, sub in df.groupby("type"):
170            plt.loglog(sub["size"], sub["avg"], marker=mark[label], ls="-",
                   label=label)
171            if len(sub) > 1:
172                a,b = np.polyfit(np.log10(sub["size"]), np.log10(sub["avg"]), 1)
173                plt.loglog(sub["size"], 10**(b+a*np.log10(sub["size"])),
174                           "--", label=f"{label} fit ~ $n^{a:.2f}$")
175        plt.xlabel("matrix size  (log)")
```

```python
176        plt.ylabel("runtime [s]  (log)")
177        plt.title("Hessenberg (general)  vs  Tridiagonal (symmetric)")
178        plt.grid(True, which="both", ls=":")
179        plt.legend(); plt.tight_layout(); plt.show()
180        return df
181
182
183  #===INTERACTIVE PART=========================================================
184  try:
185        raw = input("\nMatrix sizes (Python list) (e.g): [64,128,256,512,1024]: ")
186        sizes = literal_eval(raw) if raw.strip() else [64,128,256,512,1024]
187  except Exception:
188        print("Bad list -> using default.")
189        sizes = [64,128,256,512,1024]
190
191  dist = input("Distribution ('normal'/'uniform')  [normal]: ").strip().lower()
     or "normal"
192  mode_txt = input("Matrix type g=general, s=symmetric, b=both  [g]:
     ").strip().lower() or "g"
193  mode = "symmetric" if mode_txt=="s" else "both" if mode_txt=="b" else "general"
194  seed_txt = input("Random seed (None/int) [None]: ").strip()
195  seed_val = None if seed_txt.lower() in {"", "none"} else int(seed_txt)
196
197  for sym in ([False, True] if mode=="both" else [mode=="symmetric"]): #accuracy
     on largest size
198        verify_factorisation_once(max(sizes), dist, sym, seed_val)
199
200  benchmark_hessenberg(sizes, dist, mode, seed_val) #timings
```

The reader should be aware that my poor <u>Dell Inspiro 5590</u> 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
1   64×64 general
2   ||A − Q T Qᵀ|| = 7.51e-14
3   ||QᵀQ − I|| = 7.07e-15
4
5   64×64 symmetric
6   ||A − Q T Qᵀ|| = 4.83e-14
7   ||QᵀQ − I|| = 7.39e-15
8
9   128×128 general
10  ||A − Q T Qᵀ|| = 1.84e-13
11  ||QᵀQ − I|| = 1.26e-14
12
13  128×128 symmetric
14  ||A − Q T Qᵀ|| = 1.14e-13
```

```
15  ||QᵀQ − I|| = 1.25e-14
16
17  256×256 general
18  ||A − Q T Qᵀ|| = 4.70e-13
19  ||QᵀQ − I|| = 2.28e-14
20
21  256×256 symmetric
22  ||A − Q T Qᵀ|| = 2.78e-13
23  ||QᵀQ − I|| = 2.25e-14
24
25  512×512 general
26  ||A − Q T Qᵀ|| = 1.16e-12
27  ||QᵀQ − I|| = 4.10e-14
28
29  512×512 symmetric
30  ||A − Q T Qᵀ|| = 7.10e-13
31  ||QᵀQ − I|| = 4.09e-14
32
33  1024×1024 general
34  ||A − Q T Qᵀ|| = 3.05e-12
35  ||QᵀQ − I|| = 7.57e-14
36
37  1024×1024 symmetric
38  ||A − Q T Qᵀ|| = 1.84e-12
39  ||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.
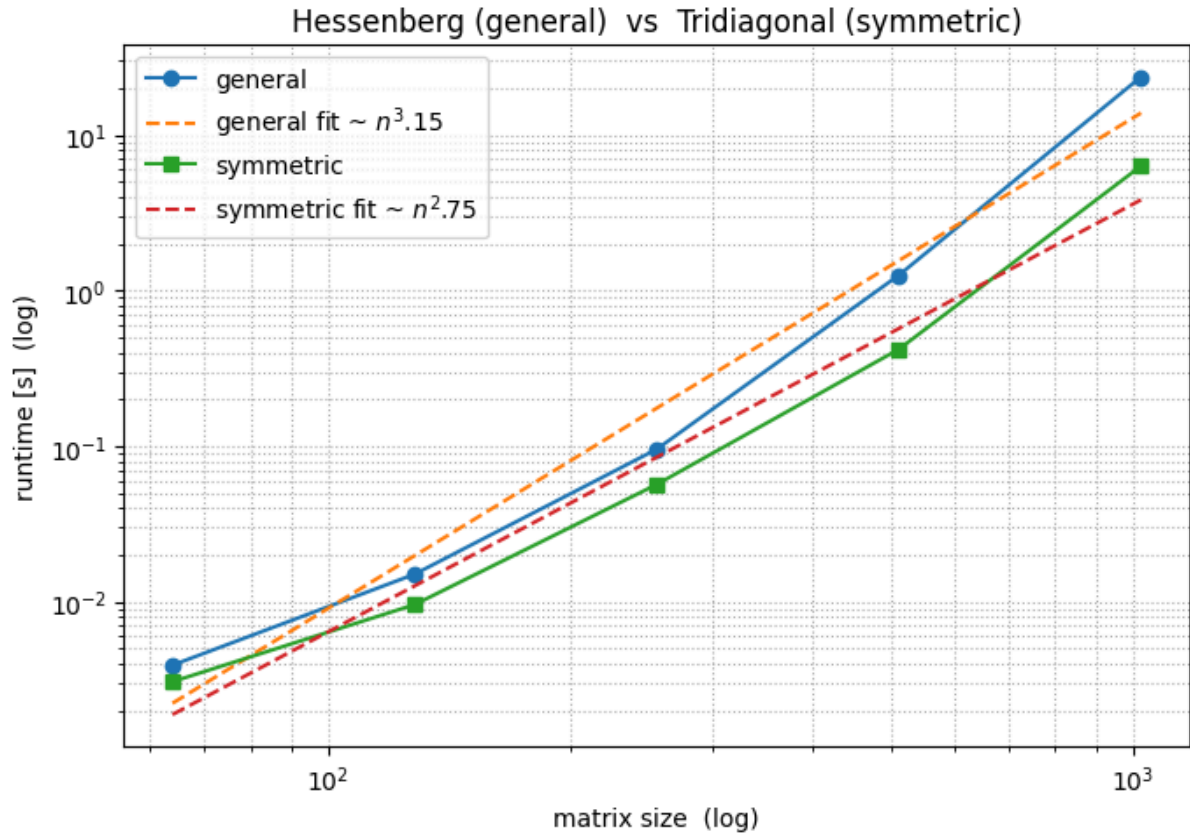
Figure 3: Runtime of the Hessenberg reduction for ordinary and symmetric matrices

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].

---

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

3. Orthogonal Matrices (Problem 2)

3.1. Eigenvalues and Iterative Methods (a)

3.2. The 2×2 Case (b)

3.3. Random Orthogonal Matrices (c)

3.4. Shift With an Eigenvalue (d)

4. Conclusion

Bibliography