Assignment 2: Numerical Linear Algebra Algorithms in NumPy

1 Instructions

In this assignment, you will implement three fundamental algorithms in numerical linear algebra using only **basic NumPy operations** (matrix-vector products, matrix-matrix products, norms, outer products, and slicing). You will then compare your outputs to NumPy's optimized "gold standard" routines to explore accuracy, correctness, and performance.

There are two components to this assignment:

1. **Implementation (The Notebook):** You will write code in a Jupyter Notebook to implement the algorithms and run experiments.
2. **Reporting (The Report):** You will summarize your findings, key code, and reflections in a structured Markdown report.

1.1 Submission Requirements

You must submit two components to Gradescope:

1. **The Report:** A plaintext Markdown document which you will paste directly into a Gradescope submission text box. This contains the **5 sections** described in Part 2 and is the **primary artifact for grading**.
    - **Character Limit:** 7,500 characters (roughly 1.5 - 2 pages of single-spaced text).

2. **Supplemental Material:** You must upload the raw materials used to create your report, specifically your `.ipynb` notebook containing all code and raw experimental results.

   - **Upload Method:** Please upload this file directly to the Gradescope assignment.
   - *Note: This supplemental material is not graded for content but is required for verification.*

2 Part 1: Implementation (The Notebook)

You will author a Jupyter Notebook (`.ipynb`) containing implementations of the three algorithms described below, along with validation and timing experiments.

2.1 Data Generation

For all algorithms, you must test your code on **symmetric matrices** ($A = A^\top$). You may generate these randomly (e.g., $A = BB^\top$). Ensure your test cases have at least $d = 10$ dimensions.

2.2 A Note on Indexing

Important: The pseudocode provided below uses standard mathematical **1-based indexing** (rows and columns $1 \dots m$). Python and NumPy use **0-based indexing** ($0 \dots m-1$).

- **Adjustment:** When translating math to code, shift indices down by 1. For example, loop variable k (1 to m) becomes `k` (0 to `m-1`) or `i` (0 to `n-1`).
- **Slicing:** Remember that Python slices `start:stop` include `start` but **exclude** `stop`.

2.3 Algorithm 1: Power Iteration

Power Iteration is an iterative method used to find a matrix's dominant eigenvector, which corresponds to the eigenvalue with the largest absolute value. The process involves repeated matrix-vector multiplication and normalization to ensure the result remains a unit vector. When applied to a sample covariance matrix, it identifies the first principal component, representing the direction of maximum variance in a dataset. Thus, making it a building block for understanding more complex dimensionality reduction techniques like PCA(Principal Component Analysis).

See also: Wikipedia - Power Iteration

Pseudocode:

Note: The initial vector v_0 is usually generated randomly (e.g., using a standard normal distribution for each entry).

Input: Symmetric matrix $A \in \mathbb{R}^{d \times d}$, initial nonzero $v_0 \in \mathbb{R}^d$, iterations T, tolerance $\varepsilon > 0$.

Output: Unit vector $\hat{v}$ approximating the dominant eigenvector and eigenvalue $\hat{\lambda}$.

1. $v \leftarrow \dfrac{v_0}{\|v_0\|_2}$
2. **for** $t = 1$ **to** T **do**
3. $\quad w \leftarrow Av$
4. $\quad v_{\text{new}} \leftarrow \dfrac{w}{\|w\|_2}$
5. $\quad$ **if** $\|v_{\text{new}} - v\|_2 < \varepsilon$ **then break**
6. $\quad v \leftarrow v_{\text{new}}$
7. **end for**
8. $\hat{v} \leftarrow v$
9. $\hat{\lambda} \leftarrow \hat{v}^\top A \hat{v}$
10. **return** $(\hat{v}, \hat{\lambda})$

2.4 Algorithm 2: QR Decomposition via Householder Reflectors

QR decomposition is a matrix factorization that breaks a matrix A into an orthogonal matrix Q and an upper triangular matrix R. You can think of it as the matrix-level version of the Gram-Schmidt process from MA 265 (Linear Algebra), which takes a set of vectors and makes them orthonormal. While Gram-Schmidt works by "subtraction", the Householder method used in this assignment works by "reflection," using linear transformations to flip vectors onto the axes. This approach is far more numerically stable and ensures that Q remains truly orthogonal and R remains accurately triangular.

See also: Wikipedia - QR Decomposition

Pseudocode:

Input: Matrix $A \in \mathbb{R}^{m \times n}$.

Output: Orthogonal $Q \in \mathbb{R}^{m \times m}$ and upper-triangular $R \in \mathbb{R}^{m \times n}$ such that $A = QR$.

1. $R \leftarrow A$
2. $Q \leftarrow I_m$
3. **for** $k = 1$ **to** $\min(m, n)$ **do**
4. $\quad x \leftarrow R_{k:m,\,k}$
5. $\quad \alpha \leftarrow \|x\|_2$
6. $\quad$ **if** $\alpha = 0$ **then continue**
7. $\quad e_1 \leftarrow [1, 0, 0, \ldots, 0]^\top$
8. $\quad s \leftarrow \begin{cases} 1, & x_1 \geq 0 \\ -1, & x_1 < 0 \end{cases}$
9. $\quad u \leftarrow x + s\alpha e_1$
10. $\quad v \leftarrow \dfrac{u}{\|u\|_2}$
11. $\quad H \leftarrow I - 2vv^\top$
12. $\quad R_{k:m,\,k:n} \leftarrow H\, R_{k:m,\,k:n}$
13. $\quad Q_{:,\,k:m} \leftarrow Q_{:,\,k:m}\, H$
14. **end for**
15. **return** (Q, R)

2.5 Algorithm 3: QR Algorithm for Eigendecomposition

The QR Algorithm is an iterative numerical method used to calculate all the eigenvalues and eigenvectors of a square matrix. Unlike a simple factorization, it is a "looping" process that repeatedly uses the QR Decomposition as its engine. Implement the **QR algorithm** (iterative) for a symmetric matrix (A), using your Householder QR from Section 3.

See also: Wikipedia - QR Algorithm

Pseudocode:

Input: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, iterations T, tolerance $\varepsilon > 0$.
Output: Orthogonal $V \in \mathbb{R}^{n \times n}$ and diagonal $\Lambda \in \mathbb{R}^{n \times n}$ s.t. $A \approx V \Lambda V^\top$.

$A_{\text{curr}} \leftarrow A$
$V \leftarrow I_n$
for $t = 1$ **to** T **do**
  $(Q, R) \leftarrow \texttt{HouseholderQR}(A_{\text{curr}})$
  $A_{\text{new}} \leftarrow RQ$
  $V \leftarrow VQ$
  $\text{off} \leftarrow A_{\text{new}} - \text{diag}(\text{diag}(A_{\text{new}}))$
  if $\|\text{off}\|_F \leq \varepsilon \|A_{\text{new}}\|_F$ **then break**
  $A_{\text{curr}} \leftarrow A_{\text{new}}$
end for
$\Lambda \leftarrow \text{diag}(\text{diag}(A_{\text{curr}}))$
return (V, Λ)

2.6 Evaluation & Complexity Experiments

In your notebook, you must perform the following for **each** algorithm.

2.6.1 1. Required Metrics for Validation

You must compute and report the following error metrics in your report to justify the correctness of your implementations.

- **For Algorithm 1 (Power Iteration):** Compute the ground truth eigenvalues using `numpy.linalg.eigh(A)`. Let λ_{GT} be the eigenvalue with the largest *absolute* value. Report the **relative difference**:

$$\text{Error} = \frac{|\hat{\lambda} - \lambda_{GT}|}{|\lambda_{GT}|}$$

- **For Algorithm 2 (QR Decomposition):** To verify your decomposition, compute the **relative squared Frobenius norm** of the reconstruction error compared to the original matrix A:

$$\text{Error} = \frac{\|A - \hat{Q}\hat{R}\|_F^2}{\|A\|_F^2}$$

- **For Algorithm 3 (QR Algorithm):** Compute the ground truth eigenvalues λ_{GT} using `numpy.linalg.eigh(A)` (note that NumPy sorts these ascendingly). Take the diagonal of your result matrix, $\hat{\lambda} = \text{diag}(\Lambda)$. **Sort** $\hat{\lambda}$ in ascending order. Report the **relative L2 norm difference** between the sorted eigenvalue vectors:

$$\text{Error} = \frac{\|\text{sort}(\hat{\lambda}) - \lambda_{GT}\|_2}{\|\lambda_{GT}\|_2}$$

2.6.2 2. Timing Analysis

Measure the **wall-clock time** (actual seconds) required to run your implementation for matrix sizes $n = 10, 100, \text{and } 1000$.

3 Part 2: Content Requirements (The Report)

Your report must be organized into **exactly five sections** with Markdown headers (e.g., `# Section 1: ...`). This text is what you will paste into Gradescope.

3.1 Section 1: Executive Summary & Key Insight

- **One Sentence Takeaway:** Start with a single sentence that captures the most surprising, confusing, or impactful insight you gained.
- **Summary Paragraph:** Write a short paragraph (3–5 sentences) summarizing what you implemented, how you validated against NumPy (citing the specific metrics defined above), and what you learned from the study.

3.2 Sections 2, 3, and 4: The Algorithms

For each algorithm (Power Iteration, Householder QR, QR Algorithm), provide a dedicated section containing:

1. **Core Code:** Paste **2–4 lines** of your actual Python code that represent the "most important" update or calculation.
2. **Explanation:** Briefly explain (1-3 sentences) how these specific lines map to the mathematical pseudocode.
3. **Gold Standard Comparison:** Explicitly report the **relative error metrics** defined in the instructions above. (e.g., "The relative L2 difference for the eigenvalues was 4.5×10^{-15}").

3.3 Section 5: Empirical Complexity & Reflection

This section must include **both**:

A) Empirical Complexity Run timing experiments for different matrix sizes $n = 10, 100, 1000$ for all three algorithms. Include:

- A **Markdown table** showing the wall-clock execution times (in seconds).
- A short discussion comparing observed scaling to expectations. What behavior did you observe when you increased n by a factor of 10 in your experiments?

B) Reflection Write a short reflection (at least one paragraph) addressing:

- What was hardest to debug and how you verified correctness.
- Any issues you observed, and the difference between the "gold standard" and your implementation.

- What you learned about algorithm design vs practical performance.

4 Grading Rubric

Each of the five sections will be weighted equally (20% each).

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 1: Insight & Summary	Takeaway is specific, memorable, and technically grounded. Summary perfectly captures the report's core results.	Takeaway is clear and relevant. Summary provides a solid overview of validation and results.	Takeaway is somewhat generic. Summary exists but is vague regarding metrics or methods.	Takeaway is missing or summary is disconnected from the work.	Section is missing or unintelligible.
Section 2: Power Iteration	Key lines are well chosen. Comparison uses the required **relative difference** metric and is precise.	Code selection is relevant. Comparison is present but might lack quantitative precision.	Code lines are present but weak/loose. Comparison is vague (e.g., "it looked close") or uses wrong metric.	Key lines or comparisons are miss-ing/incorrect.	Section is missing or fails to implement the algorithm.
Section 3: Householder QR	Key lines capture reflector formation/updates. Comparison uses the required **relative squared Frobenius norm**.	Code selection is correct. Validation is reasonable but discussion is minor.	Code selection is weak or confusing. Validation metrics are incomplete or incorrect.	Comparison or code evidence is largely incorrect or missing.	Section is missing or fails to implement the algorithm.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 4: QR Algorithm	Core lines show QR step and similarity update. Comparison uses the required **relative L2 norm difference**.	Code selection is appropriate. Comparisons are present with minor issues in discussion.	Code or comparisons are weak/confusing. Connection to Householder QR is unclear.	Implementation is incorrect or evidence is missing.	Section is missing or fails to implement the algorithm.
Section 5: Complexity & Reflection	Timing experiments cover all sizes $(10, 100, 1000)$ with a clear table. Reflection is deep, specific, and honest about debugging.	Timing results are complete. Reflection is thoughtful but may lack specific debugging details.	Timing results are present but noisy/incomplete. Reflection is generic or dutiful.	Major sizes missing in timing. Reflection is superficial or ignores the process.	Section is missing.