Lab 1

Noah Prentice

18 October 2024

Note.

Code for this assignment was done in Python. Only important code snippets and outputs are shown here, but complete files can be found at this GitHub repository.[a] All code for Exercise N is in `ExN.py`.

[a] URL: https://github.com/NoahPrentice/Numerical-Linear-Algebra-MTH551-F24/tree/main/Lab1

Exercise 1. Write an algorithm for matrix-vector multiplication $\vec{b} = A\vec{x}$ in MATLAB using two different ways:

(a) by computing inner products of rows and columns

(b) by representing the product as a linear combination of columns of A.

Solution. **(a)** The following Python code was used to compute $\vec{b} = A\vec{x}$ by computing inner products of rows and columns:

```python
def inner_product(u: np.ndarray, v: np.ndarray) -> float:
    """Calculates the inner product of two vectors of equal size.

    Parameters:
        u: first vector, as a "column vector" (i.e., a 2D np.ndarray with 1 column).
        v: second vector, also as a "column vector."

    Returns:
        The inner product of u and v.
    """
    assert u.size == v.size

    inner_product = 0
    for i in range(u.size):
        inner_product += u[i][0] * v[i][0]
    return inner_product


def multiplication_thru_inner_prodcut(A: np.ndarray, x: np.ndarray) -> np.ndarray:
    """Left-multiplies a vector by a matrix by computing inner products of the rows of
    the matrix with the vector.

    Parameters:
        A: matrix (2D np.ndarray) with, say, n columns.
        x: "column vector" (2D np.ndarray with 1 column) of length n.

    Returns:
        The product Ax, as a "column vector."
    """
    assert len(A.shape) == 2
    assert A.shape[1] == x.size

    Ax = []
```

```python
    for i in range(x.size):
        row_i_as_array = A[i, :][None]
        Ax.append([inner_product(row_i_as_array.T, x)])
    return np.array(Ax)
```

(b) The following Python code was used to compute $\vec{b} = A\vec{x}$ by by representing the product as a linear combination of columns of A:

```python
def multiplication_thru_sum_of_columns(A: np.ndarray, x: np.ndarray) -> np.ndarray:
    """Left-multiplies a vector by a matrix by computing a weighted sum of the columns of
    the matrix.

    Parameters:
        A: matrix (2D np.ndarray) with, say, n columns.
        x: "column vector" (2D np.ndarray with 1 column) of length n.

    Returns:
        The product Ax, as a "column vector."
    """
    assert len(A.shape) == 2
    assert A.shape[1] == x.size

    A_columns = [A[:, [i]] for i in range(A.shape[1])]
    weighted_sum_of_columns = A_columns[0] * x[0][0]
    for i in range(1, x.size):
        weighted_sum_of_columns += A_columns[i] * x[i][0]
    return weighted_sum_of_columns
```

These two methods were compared using random matrices in $\mathbb{R}^{m\times m}$ for $m = 2$ and $m = 100$, with 10 comparisons done for each m. The results are as follows:

- Performance. For $m = 2$, computing $A\vec{x}$ as a weighted sum of columns was 6.3 microseconds *slower* on average compared to using inner products. However, for $m = 100$ computing a weighted sum of columns was 3.75 milliseconds *faster* on average compared to the inner product method.
- Precision. The results of the two methods, $\vec{b}_1$ and $\vec{b}_2$, were compared by finding $\|\vec{b}_2 - \vec{b}_1\|_\infty$. This yielded 0.0 every time, showing that both methods produce equivalent results.

$\square$

Exercise 2. Write an algorithm for finding the residual of the "best" approximation to a vector $\vec{x}$ in the space spanned by n orthonormal m-vectors $\{q_i\}$ in MATLAB using two different ways, (as defined by solving Equation 2.7 in Trefethan-Bau for $\vec{r}$):

(a) $\vec{r} = \vec{v} - \sum_{i=1}^{n} (q_i^* v) q_i$

(b) $\vec{r} = \vec{v} - \sum_{i=1}^{n} (q_i q_i^*) v$

Solution. Note that the formula listed for (a) involves computing an inner product $q_i^* v$, whereas the formula listed for (b) involves computing an outer product $q_i^* q_i$. We therefore refer to the former as the *inner product method* and the latter as the *outer product method*.

(a) The Python code for the inner product method is as follows:[1]

```
def residual_through_inner_product(
    orthonormal_vectors: list[np.ndarray], v: np.ndarray
) -> np.ndarray:
    """Computes the residual of a vector with respect to a set of orthonormal vectors by
    computing the inner product of each orthonormal vector with v.

    Parameters:
        orthonormal_vectors: a list of orthonormal "column vectors" (2D np.ndarray
            objects with 1 column).
        v: a "column vector" of the same size as the vectors in orthonormal_vectors.

    Returns:
        The residual of v with respect to orthonormal_vectors, that is, the result after
        applying Gram-Schmidt to v using the vectors in orthonormal_vectors.
    """
    for q in orthonormal_vectors:
        assert q.shape == v.shape

    residual = v
    for q in orthonormal_vectors:
        residual -= inner_product(q, v) * q
    return residual
```

(b) The Python code for the outer product method is as follows:

```
def outer_product(u: np.ndarray, v: np.ndarray) -> np.ndarray:
    """Computes the outer product of two "column vectors" (2D np.ndarray with 1 column)

    Parameters:
        u: first "column vector"
        v: second "column vector"

    Returns:
        The outer product of u and v, uv^T.
    """
    matrix_list = []
    for row in u:
        row_i = []
        u_i = row[0]
        for column in v:
            v_j = column[0]
            row_i.append(u_i * v_j)
        matrix_list.append(row_i)
    return np.array(matrix_list)
```

[1] Note that this code uses the `inner_product()` function defined for Exercise 1.

```python
def residual_through_outer_product(
    orthonormal_vectors: list[np.ndarray], v: np.ndarray
) -> np.ndarray:
    """Computes the residual of a vector with respect to a set of orthonormal vectors by
    computing the outer product of each orthonormal vector with itself.

    Parameters:
        orthonormal_vectors: a list of orthonormal "column vectors" (2D np.ndarray
            objects with 1 column).
        v: a "column vector" of the same size as the vectors in orthonormal_vectors.

    Returns:
        The residual of v with respect to orthonormal_vectors, that is, the result after
        applying Gram-Schmidt to v using the vectors in orthonormal_vectors.
    """
    for q in orthonormal_vectors:
        assert q.size == v.size

    residual = v
    for q in orthonormal_vectors:
        q_matrix = outer_product(q, q)
        residual -= q_matrix @ v
    return residual
```

These two methods were compared using random vectors $\{q_i\}_{i=1}^n$ and v in $\mathbb{R}^{50}$ for $n = 30$ and $n = 50$, with 10 comparisons done for each n.

- Performance. The inner product method tested faster on average for both values of n: for $n = 30$, it was on average 32.4 milliseconds faster than the outer product method. This difference in performance was exaggerated for $n = 50$, where the inner product method was on average 53.4 milliseconds faster than the outer product method.
- Precision. As in Exercise 1, we measure the distance between the methods' results using the ∞-norm difference of the residuals. Doing so yielded 0.0 on every test, showing that the methods yield equivalent results.

$\square$

Exercise 4. Create a function `Aball` which takes as an additional input a matrix A and plots the image of the unit ball under the mapping defined by A.

Solution. The implementation of this function is as follows:[2]

```
def Aball(A: np.ndarray, M: int) -> None:
    """Plots the image of the unit ball under matrix A with resolution M."""
    t = [i / M for i in range(M)] + [0.0]
    x = [math.cos(2 * math.pi * t_i) for t_i in t]
    y = [math.sin(2 * math.pi * t_i) for t_i in t]
    ...
    for i in range(M + 1):
        old_x = x[i]
        old_y = y[i]
        vector = np.array([[old_x], [old_y]])
        vector = np.matmul(A, vector)
        new_x = vector[0][0]
        new_y = vector[1][0]
        x[i] = new_x
        y[i] = new_y
    ...

    plt.show()
```

Which yielded the following plots for AS_2 which were particularly skinny and fat:

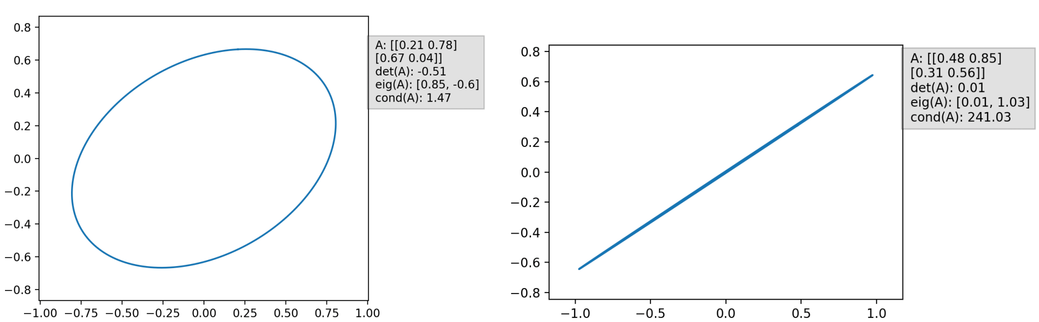

Figure 1: AS_2

As we can see, the fatter ellipse has a lower condition number (quite close to 1), and its eigenvalues and determinant are far from 0; the skinnier ellipse has a much higher condition number (by 2 orders of magnitude), and it has an eigenvalue and determinant very close to 0. I would wager that this is not coincidental:

Conjecture.

Suppose A_1 and A_2 are 2×2 real matrices and $S_2 = \{v \in \mathbb{R}^2 : \|v\|_2 = 1\}$ is the unit circle. If $A_1 S_2$ is a fatter ellipse than $A_2 S_2$ (where "fatness" is measured by an ellipse's eccentricity), then

1. $1 \leq \operatorname{cond}(A_1) < \operatorname{cond}(A_2)$
2. $0 \leq \min_{\lambda \in \operatorname{eig}(A_2)} |\lambda| < \min_{\lambda \in \operatorname{eig}(A_1)} |\lambda|$
3. $0 \leq |\det(A_2)| < |\det(A_1)|$

□

[2]Code for plotting omitted. The inquisitive reader should investigate the GitHub repository for more information.