Challenge 1 APC 2024

October 2024

1 Introduction

In the Linear Algebra course you learned how to calculate eigenvalues of square matrices of size $n \times n$ by solving the characteristic equation:

$$\lambda^n + c_{n-1}\lambda^{n-1} + c_{n-2}\lambda^{n-2} + \cdots + c_0 = 0. \tag{1}$$

For large values of n, polynomial equations like (1) are difficult and time-consuming to solve. Moreover, numerical techniques for approximating roots of polynomial equations of high degree are sensitive to rounding errors.

The *Power Iteration* method is an iterative algorithm used to find the *dominant eigenvalue* (the eigenvalue with the largest absolute value) and its corresponding eigenvector of a matrix. Although this restriction may seem severe, dominant eigenvalues are of primary interest in many real-world applications, such as when dealing with dynamic processes.

2 The Power Iteration method

Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be a square, diagonalizable matrix, and let

- $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ be the eigenvalues of $\mathbf{A}$ and assume that the following holds:

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

- $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ be the corresponding eigenvectors of $\mathbf{A}$ such that

$$||\mathbf{x}_i|| = 1 \ \forall i \in \{1, 2, \ldots, n\}.$$

- $\mathbf{x}^{(0)} \in \mathbb{R}^n$, such that $||\mathbf{x}^{(0)}|| = 1$.

The *Power Iteration* method finds the *dominant eigenvalue* building the following successions:

$$\mathbf{z}^{(k+1)} = \mathbf{A}\mathbf{x}^{(k)} \tag{2}$$

$$\mathbf{x}^{(k+1)} = \frac{\mathbf{z}^{(k+1)}}{||\mathbf{z}^{(k+1)}||} \tag{3}$$

$$\nu^{(k+1)} = \langle \mathbf{x}^{(k+1)}, \mathbf{A}\mathbf{x}^{(k+1)} \rangle \tag{4}$$

The notation $\langle \cdot, \cdot \rangle$ denotes the scalar product between two quantities.

It can be proven that:

- $\{\mathbf{x}^{(k)}\}_{k \geq 0} \xrightarrow[k \to \infty]{} e^{i\theta}\mathbf{x}_1$.
- $\{\nu^{(k)}\}_{k \geq 0} \xrightarrow[k \to \infty]{} \lambda_1$.
- The error at the k^{th} iteration is bounded by $\left|\frac{\lambda_2}{\lambda_1}\right|^k$.

To define convergence criteria, we introduce the following quantities:

$$residual^{(k)} = ||\mathbf{A}\mathbf{x}^{(k)} - \nu^{(k)}\mathbf{x}^{(k)}||, \tag{5}$$

$$increment^{(k)} = \frac{|\nu^{(k)} - \nu^{(k-1)}|}{|\nu^{(k)}|}. \tag{6}$$

The convergence criterion of the algorithm is: *Stop at the k^{th} iteration if both the following conditions are met*

$$\begin{cases} residual^{(k)} < tolerance \\ increment^{(k)} < tolerance \end{cases} \tag{7}$$

Algorithm 1 presents the complete algorithm for the Power Iteration method. Note that in Algorithm 1 the value of *tolerance* is set to 10^{-6} and that of T_{max} is set to 10000. The algorithm takes as input a matrix $\mathbf{A}$ and a vector $\mathbf{x}^{(0)}$, where the latter must satisfy the condition $||\mathbf{x}^{(0)}|| = 1$.

Algorithm 1 Power Iteration method.

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{x}^{(0)} \in \mathbf{R}^n$ such that $||\mathbf{x}^{(0)}|| = 1$
Initialize: *tolerance* $\leftarrow 10^{-6}$,
  $T_{max} \leftarrow 10000$,
  residual $\leftarrow$ *increment* $\leftarrow$ *tolerance*$+1$,
  $t \leftarrow 0$,
  converged $\leftarrow$ *False*
while not *converged* and $t < T_{max}$ **do**
  Compute $\mathbf{x}^{(k+1)}$ and $\nu^{(k+1)}$ as in (2)-(4)
  converged $\leftarrow$ Check convergence as defined in (5)
  $t \leftarrow t + 1$
end while

3 Variants of the Power Iteration method

3.1 Inverse Power Iteration method

The *Inverse Power Iteration* method allows us to approximate the eigenvalue of minimum modulus (i.e., $|\lambda_n|$) of $\mathbf{A}$. The key idea is that the eigenvalues of $\mathbf{A}^{-1}$ are the reciprocals of the eigenvalues of $\mathbf{A}$. Therefore, the minimum modulus eigenvalue of $\mathbf{A}$ can be computed through the following process:

- Compute the *dominant eigenvalue* $\tilde{\lambda}_1$ of $\mathbf{A}^{-1}$ using the *Power Iteration* method.
- Compute the minimum modulus eigenvalue as $\lambda_n = \tilde{\lambda}_1^{-1}$.

Note that, for this method, the error at the k^{th} iteration is bounded by $\left|\frac{\lambda_n}{\lambda_{n-1}}\right|^k$.

3.2 Inverse Power Iteration method with shift

Let $\mu \in \mathbb{R}$. The *Inverse Power Iteration method with shift* allows us to approximate the eigenvalue of $\mathbf{A}$ that is closest to μ. The core idea is the following: if λ is an eigenvalue of $\mathbf{A}$, then $\tilde{\lambda} = \lambda - \mu$ is an eigenvalue of $\mathbf{A} - \mu\mathbf{I}$. Therefore, the eigenvalue of $\mathbf{A}$ closest to μ can be computed as follows:

- Compute the minimum modulus eigenvalue $\tilde{\lambda}_\mu$ of $\mathbf{A} - \mu\mathbf{I}$ using the *Inverse Power Iteration* method.
- Compute the eigenvalue of $\mathbf{A}$ closest to μ as $\lambda_\mu = \mu + \tilde{\lambda}_\mu$.

Note that if μ is an eigenvalue of $\mathbf{A}$, the method fails (in our case, we defined the test matrices in such a way that the algorithm never fails).

4 Code implementation

The provided code, zipped in `Assignment1_initial.zip`, contains:

- the `vectorhelpers` header and source files;
- the `squarematrix` header and source files;
- the `matrixhelpers` header and source files;
- the `power_iteration` class implementation;
- the `inverse_power_iteration` class implementation;
- the `shift_inverse_power_iteration` class implementation;
- the `eig_finder_helpers.h` file;
- the `inputs` folder;

- the `main.cpp` file.

After carefully reading the code, you have to implement the following functions:

```
double power_iteration::solve(
                        const linear_algebra::square_matrix& A,
                        const std::vector<double>& x0)  const;

double inverse_power_iteration::solve(
                        const linear_algebra::square_matrix& A,
                        const std::vector<double>& x0)  const;

double shift_inverse_power_iteration::solve(
                        const linear_algebra::square_matrix& A,
                        const double& mu,
                        const std::vector<double>& x0)  const;
```

These functions implement, respectively, the *Power Iteration* method, the *Inverse Power Iteration* method, and the *Inverse Power Iteration method with shift*. Each function takes the square matrix $\mathbf{A}$ and the initial vector $\mathbf{x}^{(0)}$ as inputs and returns an approximation of the desired eigenvalue. The last function also takes μ as input, as defined in the previous section.

Remark: for the *Inverse Power Iteration* method(s), you do not need to explicitly invert $\mathbf{A}$. Instead, you should rely on the *LU* factorization, solving the following linear system:

$$\mathbf{Ax} = \mathbf{b}.$$

By factorizing $\mathbf{A}$ as $\mathbf{A} = \mathbf{LU}$—where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular matrices, respectively—the linear system becomes:

$$\mathbf{LUx} = \mathbf{b}.$$

This system can be solved by breaking it into the following two triangular systems:

$$\mathbf{Ly} = \mathbf{b}$$

$$\mathbf{Ux} = \mathbf{y}.$$

The first step is a *forward substitution*, and the second step is a *backward solution*.

You can rely on the following functions implemented in the `matrixhelpers` files:

```
void lu(const square_matrix& A, square_matrix& L, square_matrix& U);

std::vector<double> backsolve(const square_matrix& U,
                              const std::vector<double>& b);
```

```
std::vector<double> forwardsolve(const square_matrix& L,
                                 const std::vector<double>& b);
```

Note that these three functions are defined in the `linear_algebra` namespace.

Important: the only files to be modified are

- `power_iteration.cpp`,
- `inverse_power_iteration.cpp` and
- `shift_inverse_power_iteration.cpp`.

The implementations must be consistent with the declarations.

You can rely on the already implemented functions for your own implementation.

The following is an extract of the `main` function in the `main.cpp` file.

```
int main() {

    std::string filename = "../inputs/input_10.txt";

    linear_algebra::square_matrix A(filename);
    std::vector<double> x0(A.size());
    x0[0] = 1.;    //starting point

    eigenvalue::power_iteration pi(10000, 1e-6, BOTH);

    double max_expected;

    if (filename == "../inputs/input_10.txt"){
        max_expected = 5.10274;
    }

    double max_obtained = pi.solve(A, x0);
    std::string result;
    if (std::abs(max_obtained - max_expected) < 1e-3)
        result = "CONVERGED";
    else
        result = "NOT CONVERGED";

    return 0;
}
```

Note that you can retrieve the input files from the `inputs` folder, and you need to modify the string value of the `filename` variable to match the desired file in order to test different matrices. Additionally, in the `if` block(s), you will

find the expected outputs of the three algorithms for the three different input matrices we provide. From these results, we will consider all solutions with a predetermined tolerance to be acceptable. In order to compare the results with those in the `main.cpp` file, it is important that you do not change the algorithm parameters, otherwise the algorithm may work but not show the same results.

Important: During the evaluation, we will test your code not only on the provided tests, but also on some matrices that are not available to you.

5 Delivery Instructions

The assignment is not mandatory. If the solution is implemented correctly, it can lead to a +1 point in the final grade.

Please, follow these instructions for the delivery:

- Download the zipped folder `Assignment1_initial.zip` from WeBeep.
- Unzip the `Assignment1_initial.zip` folder.
- Change the name of the **unzipped** folder in "`YourCodicePersona`" (e.g., "`10699999`"). **It is important to do this before re-zipping the project with your solution.**
- Implement your code within the provided files.
- Test the code on the provided three test cases (changing the input file in `main.cpp`).
- Zip the folder containing the entire project, making sure that the resulting file name is "`YourCodicePersona.zip`" (e.g., "`10699999.zip`").
- Upload on WeBeep $\longrightarrow$ Assignments $\longrightarrow$ Assignment1.

Attention: The assignment is personal; no group nor team work is allowed. In case of plagiarism, you will be assessed 2 penalty points (i.e., a -2 on the total score), and you will not be able to participate in the next assignments.

If you have questions, post them on the WeBeep Assignments forum. **Note that we will not provide feedback in the last 24 hours before the deadline.**

Code submission opens on 21/10/2024 at 08:00 and closes on 25/10/2024 at 18:00 (Rome time).

If something changes in the source code we provided, an announcement will notify you. So, keep an eye on WeBeep these days.