EPFL - Master
Computational Science and Engineering
Ivan Bioli

Programming Concepts in Scientific Computing
Fall 2021

1 Compiling the program and the tests

Read our `README` to have instructions on how to build our program and on how to run the program for the provided input file examples.

2 Program execution and flow

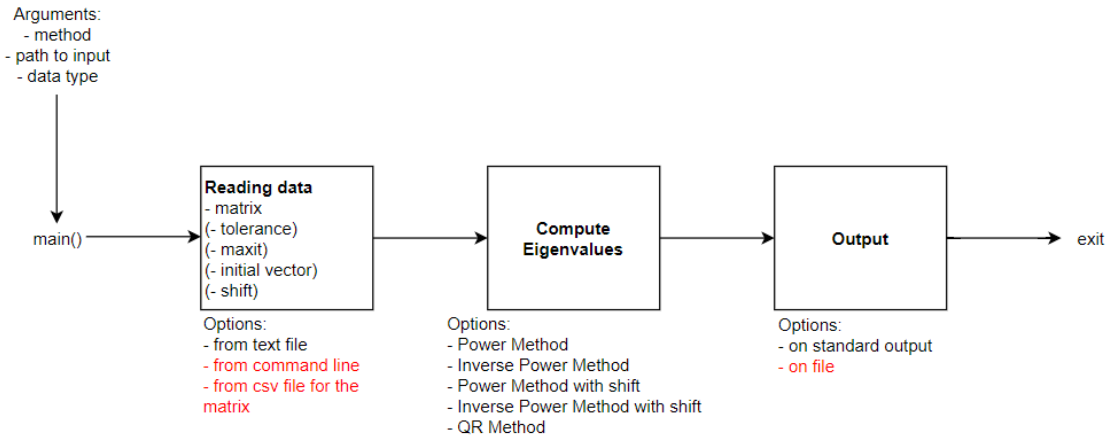

Figure 1: Flow diagram. In red some of the TODOs of the project.

Our central executable, called `main`, requires three arguments:

- the numerical method the user wants to apply: it can be `power` for the Power Method, `invpower` for the Inverse Power Method, `shiftpower` for the Power Method with shift, `shiftinvpower` for the Inverse Power Method with shift or `qr` for the QR Method;
- the path to the file in which the parameters of the method are indicated. Two examples are already provided in `input_files/real_input.txt` and `input_files/complex_input.txt`;
- the type of the input, `real` or `complex`;

Since it is not known at compilation time if the user will use a real or complex input, two (smart) pointers are created: one for the complex input and one for the real input. Only one of them will be effectively used. The same is done for the reader. The program reads the input from the file, instantiates and object of the class correspondent to the method given in input and performs the eigenvalues computation, writing the result on standard output preceded by the matrix and the eventual shift applied.

3 Implementation

3.1 Implementation of the methods

The abstract class `AbstractEigs` is the mother class of the classes in which the methods are effectively implemented. It is a template class, where the template parameter `T` is the scalar type of the matrix. Since we want `T` to be only `double` or `std::complex<double>`, we use explicit template instantiation for the `AbstractEigs` class and its derived classes. The QR method has been implemented for real matrices only, therefore we explicitly instantiate only `QRMethod<double>`.

The matrix and all the parameters of the methods are stored as members of the classes, and the eigenvalues computation is executed using the `ComputeEigs()` method.

The method `AbstractEigs<T>::ComputeEigs()` is a pure virtual method that is overridden in the derived classes to execute the eigenvalues computation according to the correspondent numerical method. Its return type is `Eigen::Vector<std::complex<double>, -1>` independently of the template parameter, because even if the matrix is real its eigenvalues can be complex. See the documentation to explore the class diagram.

3.1.1 Power Methods

The main cost of one step of the Power Method comes from executing the multiplication $Ax^{(k)}$. Therefore, in order to reduce the number of multiplication $Ax^{(k)}$ to one per step, the Power Method can be implemented as in Algorithm 1. Observe that we need to store the vector $x^{(k)}$ and the approximation of the eigenvalue $\lambda^{(k)}$ just for the current and the previous iteration.

Algorithm 1 Power Method

Starting from an initial nonzero vector $x^{(0)}$, divide it by its norm: $x^{(0)} = x^{(0)}/\|x^{(0)}\|$
$x^{(1)} = Ax^{(0)}$
$\lambda^{(0)} = x^{(1)} \cdot x^{(0)}$
Set $k = 1$
while stopping criterion **do**
    $x^{(k)} = x^{(k)}/\|x^{(k)}\|$
    $x^{(k+1)} = Ax^{(k)}$
    $\lambda^{(k)} = x^{(k)} \cdot x^{(k+1)}$
    $k = k + 1$
end while
if converged **then**
    return $\lambda^{(k-1)}$
end if

The Power Method, the Inverse Power Method and their shifted versions actually differ just in the multiplication step and in the returned approximation of the eigenvalue. Indeed:

- for the Inverse Power Method, $x^{(k+1)} = A^{-1}x^{(k)}$ and $1/\lambda^{(k-1)}$ is returned;
- for the Power Method with shift σ, $x^{(k+1)} = (A - \sigma I)x^{(k)}$ and $\lambda^{(k-1)} + \sigma$ is returned;
- for the Inverse Power Method with shift σ, $x^{(k+1)} = (A - \sigma I)^{-1}x^{(k)}$ and $1/\lambda^{(k-1)} + \sigma$ is returned;

Therefore we implemented and abstract class `AbstractPowerMethod`, derived from `AbstractEigs`, with two pure virtual methods:

- `T _return(T &lambda)`, which is supposed to return the eigenvalue transforming $\lambda^{(k-1)}$ according to what described above.
- `Eigen::Vector<T,-1> Multiply(const Eigen::Vector<T,-1> &x)`, which is supposed to execute the multiplication step.

The Power Method scheme is implemented in the method `AbstractPowerMethod<T>::ComputeEigs()` making use of the two pure vitual methods `_return` and `Multiply`. In the derived classes these two pure virtual methods are overridden in order to implement the four versions of the Power Method. This allows us to avoid the repetition of code for the steps that are shared by the four methods.

A few additional remarks:

- In the Inverse Power Method and in the Inverse Power Method with shift, the multiplication step is actually executed solving a linear system. Since the matrix of the linear system is the same across all the iterations, the LU factorization is computed at the beginning of the execution of the method and stored in a member of the class.
- The `ShiftInvPowerMethod` class (i.e. the class for the Inverse Power Method with shift) is a derived class of `ShiftPowerMethod` (i.e. the class for the Power Method with shift). This is done to avoid repetition of code in the costructors, the `SetShift` and the `GetShift` methods.

3.2 Implementation of the reader

The abstract class `Reader` is the mother class of the classes that read matrices, vector and scalars, storing a map that associates each of these object with a "name" that the user provides. It is a template class, where the template parameter `T` is the scalar type of which the input matrices and vectors are assumed to be. For our purposes, `T` can only be `double` or `std::complex<double>`, therefore we use explicit template instantiation for the `Reader` class and its derived classes. The reading is executed using the `Read()` method. The main idea behind this class is to have a general reader that could read from various type of input and return them in a "standardized format", i.e. a map. This standardized format can be passed to any of the implemented eigenvalues solver, which "knows who he is" and can extract from the map the needed parameters.

Due to time constraints only one derived class is implemented, the `FileReader` class, that reads the input from a text file. We read the input line by line and distinguish in four cases. If we encounter a line containing the word `Matrix`, we assume that we will read in the following lines: the "name" of the matrix, the dimensions, the entries of the matrix entered by line. A similar pattern is used for reading vectors, which are announced by the word `Vector`, and scalars of type `T`, which are announced by the word `Type`. If none of the mentioned keywords is encountered, we assume the input to be of type `double`. See the documentation to have a more precise description of the assumed file format.

To read the next numeric input from the file, we use the template private method `NumericInput`.

3.3 Tests

The numerical methods for the eigenvalues computation are tested using two matrices whose eigenvalues are known, one for the real case and one for the complex case. The matrices, their eigenvalues and the expected output of the methods are the ones reported in the `README`.

We use a typed test suite to test both the real and the complex case. The `Initialization()` method is template specialized to set the right matrix and the shift according to the type for which we want to execute the tests. The first block of tests validates the constructors and methods for initializing the classes, before the effective computation of the eigenvalues. These tests are executed for the `PowerMethod` and `ShiftPowerMethod` classes, since they cover all the possible constructors and initializing methods implemented. The second block of tests validates the eigenvalues computation. For the test matrices defined in the `Initialization` method, the eigenvalues are computed using the implemented tests and compared to the exact ones.

The `FileReader` class is tested using two different test input files, `test_input_real.txt` and `test_input_complex.txt`, for the real and the complex case respectively. We use a typed test suite and we define an `Initialization()` method similarly to the tests for the methods. The first test checks that the `FileNotOpen` exception is thrown when the provided path does not correspond to any file. The second block of tests reads from the test input files and checks that the stored objects match with the ones provided in the file.

4 Limitations, TODO's and perspectives

- The tests for the numerical methods could have been templated also over the type of the eigenvalues solver, but we did not manage to do it before the deadline of the project.
- The QR Method has been implemented only for the real case. This is a limitation of our implementation, but since the `AbstractEigs` class and its derived classes are already template classes, the QR Method can be extended to complex matrices without modifying the structure of the classes and their inheritance relationships.
- If the matrix is real but with complex conjugate eigenvalues, the sequence of matrices generated by the QR mathod converges to an almost upper triangular matrix, where the main subdiagonal has nonzero entries only when there is a complex conjugate pair of eigenvalues. This second case is not handled by our implementation, that only works if the matrix is real with real eigenvalues. This case should be handled in a future extension of the code.
- Our `Reader` class has only one derived class and it could seem useless to have this abstract class. However, the initial intention was to allow more types of inputs (command line for instance), which would have been added as additional derived classes. This is still a TODO of this project, but the main idea was to have a general reader that could read from various type of input and return them in a "standardized format", i.e. a map. This standardized format can be passed to any of the developed eigenvalues solver, which "knows who he is" and can extract from the map the needed parameters.
- The `FileReader` class has strong restrictions on the expected file format to read and assumes that the input file format is correct, without checking this. Some examples of checks that could be added in future extensions are checks on the entries of the matrix effectively given in the file (check if the number of rows given corresponds to what indicated in the line after the matrix key. Then check if, for each row, the number of entries given effectively corresponds to the number of columns indicated) and same checks for vectors. In a successive stage, automatically extrapolate the number of columns and rows from the input file. Throw specific errors if the numeric input does not correspond to the one expected. Throw errors if a key is present without any value or if, viceversa, a value is present without any key. More in general, allow a less restricted input style.
- Similarly to what has been done for the reader, a class for a writer should be developed, allowing the user to have the output stored in a file instead than printed on screen.

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. English. Johns Hopkins University Press Baltimore, 1983, xvi, 476 p. :