

6.2: Numerical Eigensolvers

As discussed above, Abel's impossibility theory tells us that there is no general algebraic formula for calculating the eigenvalues of an $N \times N$ matrix, for $N \geq 5$. In practice, however, there exist numerical methods, called **eigensolvers**, which can compute eigenvalues (and eigenvectors) even for very large matrices, with hundreds of rows/columns, or larger. How could this be?

The answer is that numerical eigensolvers are *approximate, not exact*. But even though their results are not exact, they are very precise—they can approach the exact eigenvalues to within the fundamental precision limits of floating-point arithmetic. Since we're limited by those floating-point precision limits anyway, that's good enough!

6.2.1 Sketch of the Eigensolver Method

We will not go into detail about how numerical eigensolvers work, as that would involve a rather long digression. For those interested, the following paper provides a good pedagogical presentation of the subject:

- Bruno Lang, *Direct Solvers for Symmetric Eigenvalue Problems*. Modern Methods and Algorithms of Quantum Chemistry, Proceedings, Second Ed. (2000). [PDF download link](#)

Here is a very brief sketch of the basic method. Similar to Gaussian elimination, the algorithm contains two phases, a relatively costly/slow initial phase and a relatively fast second phase. The first phase, which is called [Householder reduction](#), applies a carefully-chosen set of similarity transformations to the input matrix \mathbf{A}_0 :

$$\mathbf{A}_0 \rightarrow \mathbf{A}_1 = \mathbf{X}_1^{-1} \mathbf{A}_0 \mathbf{X}_1 \rightarrow \mathbf{A}_2 = \mathbf{X}_2^{-1} \mathbf{A}_1 \mathbf{X}_2, \text{ etc.} \quad (6.2.1)$$

The end result is a matrix \mathbf{A}_k which is in Hessenberg form: the elements below the first subdiagonal are all zero (the elements immediately below the main diagonal, i.e. along the first subdiagonal, are allowed to be nonzero). The entire Householder reduction phase requires $O(N^3)$ arithmetic operations, where N is the size of the matrix.

The second phase of the algorithm is called QR iteration. Using a different type of similarity transformation, the elements along the subdiagonal of the Hessenberg matrix are reduced in magnitude. When these elements become negligible, the matrix becomes upper-triangular; in that case, the eigenvalues are simply the elements along the diagonal.

The QR process is *iterative*, in that it progressively reduces the magnitude of the matrix elements along the subdiagonal. Formally, an infinite number of iterations would be required to reduce these elements to zero—that's why Abel's impossibility theorem isn't violated! In practice, however, QR iteration converges extremely quickly, so this phase ends up taking only $O(N^2)$ time.

Hence, the overall runtime for finding the eigenvalues of a matrix scales as $O(N^3)$. The eigenvectors can also be computed as a side-effect of the algorithm, with no extra increase in the runtime scaling.

6.2.2 Python Implementation

There are four main numerical eigensolvers implemented in Scipy, which are all found in the `scipy.linalg` package:

- `scipy.linalg.eig` returns the eigenvalues and eigenvectors of a matrix.
- `scipy.linalg.eigvals` returns the eigenvalues (only) of a matrix.
- `scipy.linalg.eigh` returns the eigenvalues and eigenvectors of a Hermitian matrix.
- `scipy.linalg.eigvalsh` returns the eigenvalues (only) of a Hermitian matrix.

The reason for having four separate functions is efficiency. The runtimes of all four functions *scale* as $O(N^3)$, but for each N the actual runtimes of `eigvals` and `eigvalsh` will be shorter than `eig` and `eigh`, because the eigensolver is only asked to find the eigenvalues and need not construct the eigenvectors. Furthermore, `eigvalsh` is faster than `eigvals`, and `eigh` is faster than `eig`, because the eigensolver algorithm can make use of certain numerical shortcuts which are valid only for Hermitian matrices.

If you pass `eigvalsh` or `eigh` a matrix that is not actually Hermitian, the results are unpredictable; the function may return a wrong value without signaling any error. Therefore, you should only use these functions if you are sure that the input matrix is definitely Hermitian (which is usually because you constructed the matrix that way); if the matrix is Hermitian, `eigvalsh` or `eigh` are certainly preferable to use, because they run faster than their non-Hermitian counterparts.

Here is a short program that uses `eigvals` to find the eigenvalues of a 3×3 matrix:

```
from scipy import *
import scipy.linalg as lin

A = array([[1,3,1],[1, 3, 4],[2, 4, 2]])
lambd = lin.eigvals(A)

print(lambd)
```

Running the program outputs:

```
[ 7.45031849+0.j          -0.72515925+0.52865751j -0.72515925-0.52865751j]
```

The return value of `eigvals` is a 1D array of complex numbers, storing the eigenvalues of the input. The `eigvalsh` function behaves similarly, except that a real array is returned (since Hermitian matrices have real eigenvalues). Note: we cannot use `lambda` as the name of a variable, because `lambda` is reserved as a special keyword in Python.

Here is an example of using `eig` :

```
>>> A = array([[1,3,1],[1, 3, 4],[2, 4, 2]])
>>> lambd, Q = lin.eig(A)
>>> lambd
array([ 7.45031849+0.j          , -0.72515925+0.52865751j,   -0.72515925-0.52865751j])
>>> Q
array([[ 0.40501343+0.j          ,  0.73795979+0.j          ,  0.73795979-0.j          ],
       [ 0.65985810+0.j          , -0.51208724+0.22130102j, -0.51208724-0.22130102j],
       [ 0.63289132+0.j          ,  0.26316357-0.27377508j,  0.26316357+0.27377508j]])
```

The `eig` function returns a pair of values; the first is a 1D array of eigenvalues (which we name `lambd` in the above example), and the second is a 2D array containing the corresponding eigenvectors in each column (which we name `Q`). For example, the first eigenvector can be accessed with `Q[:,0]`. We can verify that this is indeed an eigenvector:

```
>>> dot(A, Q[:,0])
array([ 3.01747903+0.j,  4.91615298+0.j,  4.71524187+0.j])
>>> lambd[0] * Q[:,0]
array([ 3.01747903+0.j,  4.91615298+0.j,  4.71524187+0.j])
```

The `eigh` function behaves similarly, except that the 1D array of eigenvalues is real.

6.2.3 Generalized Eigenvalue Problem

Sometimes, you might also come across **generalized eigenvalue problems**, which have the form

$$\mathbf{A} \vec{x} = \lambda \mathbf{B} \vec{x}, \quad (6.2.2)$$

for known equal-sized square matrices \mathbf{A} and \mathbf{B} . We call λ a "generalized eigenvalue", and \vec{x} a "generalized eigenvector", for the pair of matrices (\mathbf{A}, \mathbf{B}) . The generalized eigenvalue problem reduces to the ordinary eigenvalue problem when \mathbf{B} is the identity matrix.

The naive way to solve the generalized eigenvalue problem would be to compute the inverse of \mathbf{B}^{-1} , and then solve the eigenvalue problem for $\mathbf{B}^{-1}\mathbf{A}$. However, it turns out that the generalized eigenvalue problem can be solved directly with only slight modifications to the usual numerical eigensolver algorithm. In fact, the Scipy eigensolvers described in the previous section will solve the generalized eigenvalue problem if you pass a 2D array \mathbf{B} as the second input (if that second input is omitted, the eigensolvers solve the ordinary eigenvalue problem, as described above).

Here is an example program for solving a generalized eigenvalue problem:

```
from scipy import *
import scipy.linalg as lin

A = array([[1,3,1],[1, 3, 4],[2, 4, 2]])
B = array([[0,2,1], [0, 1, 1], [2, 0, 1]])

lambda, Q = lin.eig(A, B)

## Verify the solution for first generalized eigenvector:
lhs = dot(A,Q[:,0])           # A . x
rhs = lambda[0] * dot(B, Q[:,0]) # lambda B . x

print(lhs)
print(rhs)
```

Running the above program prints:

```
[-0.16078694+0.j -0.07726949+0.j  0.42268561+0.j]
[-0.16078694+0.j -0.07726949+0.j  0.42268561+0.j]
```

The Hermitian eigensolvers, `eigh` and `eigvalsh`, can be used to solve the generalized eigenvalue problem only if *both* the **A** and **B** matrices are Hermitian.

This page titled [6.2: Numerical Eigensolvers](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.