# Eigendecomposition and Power Iteration

July 09, 2025

**Giap Bui-Huy**

# Table of contents



# 1. Introduction

This article briefly explains matrix eigen(value) decomposition, which will be abbreviated as EVD. The EVD of a matrix is a factorization of an $n \times n$ matrix $\boldsymbol{A}$ into $\boldsymbol{PDP}^{-1}$, where $\boldsymbol{P} = (\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n)$ are called the eigenvectors of $\boldsymbol{A}$, and $\boldsymbol{D}$ is a diagonal matrix:

$$\boldsymbol{D} = \begin{pmatrix} \lambda_1 & 0 & ... & 0 \\ 0 & \lambda_2 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \lambda_n \end{pmatrix} \tag{1}$$

The diagonal entries $\lambda_1, \lambda_2, ..., \lambda_n$ of $\boldsymbol{D}$ are called the eigenvalues of $\boldsymbol{D}$. An eigenvector $\boldsymbol{v}_i$ and its corresponding eigenvalue $\lambda_i$ is connected through the following formula:

$$\boldsymbol{A}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i \tag{2}$$

Beause of this relationship, the eigenvectors and eigenvalues of a matrix provides many insights into the matrix, such as the general scale and direction of linear transformations and the spread of multivariate datasets. Because of this, EVD and its derivatives — spectral decomposition and singular value decomposition — are valuable tools in the age of data science and machine learning.

This article is an informal exploration of the connection between EVD and matrix power. While using EVD for matrix power is a popular application, what's interesting is going the other way around. This gives us the Power Iteration method, a simple and effective way to find the domiance eigenvector. This method can be extended to find the equivalent eigenvalue, increasing the precision of a rough eigenvector/eigenvalue approximation, and is the basis of more advanced EVD algorithms such as the QR algorithm.

# 2. Use the EVD to compute matrix power

Raising a matrix to an integer power, or computing $\boldsymbol{A}^k$ with $k \in \mathbb{Z}$ is difficult for human, but EVD is *relatively* easy. So it's natural to turn to EVD to compute matrix powers. The reason for this is that $\boldsymbol{D}$ is diagonal, and it can be trivially proven with induction that:

$$\boldsymbol{D}^k = \begin{pmatrix} \lambda_1^k & 0 & \dots & 0 \\ 0 & \lambda_2^k & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n^k \end{pmatrix} \tag{3}$$

This also allows us to extend the concept of matrix power to real numbers, but that's outside the scope of this article. For $k \in \mathbb{Z}$, we can write $A^k$ as repeated multiplications:

$$\boldsymbol{A}^k = \underbrace{\boldsymbol{A}\boldsymbol{A}\dots\boldsymbol{A}}_{k \text{ times}} \tag{4}$$

Then substitue $\boldsymbol{A}$ with its EVD $\boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{-1}$, and cancel out every $\boldsymbol{P}$ and $\boldsymbol{P}^{-1}$ pairs.

$$\begin{aligned} \boldsymbol{A}^k &= \boldsymbol{P}\boldsymbol{D}\underbrace{\boldsymbol{P}^{-1}\boldsymbol{P}}_{\boldsymbol{I}_n}\boldsymbol{D}\dots\boldsymbol{D}\boldsymbol{P}^{-1} \\ &= \boldsymbol{P}\boldsymbol{D}^k\boldsymbol{P}^{-1} \\ &= \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \lambda_1^k & 0 & \dots & 0 \\ 0 & \lambda_2^k & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n^k \end{pmatrix} \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix}^{-1} \end{aligned} \tag{5}$$

Then we can multiply these matrices together to get the closed form expression for $A^k$. Or at least, we can take the EVD of a matrix $\boldsymbol{A}$ to inspect the growth rate of $\boldsymbol{A}^k$.

## 2.1. Example: Fibonacci sequence

To better illustrate this concept, let's consider the problem of finding the $n$-th Fibonacci number. The Fibonacci sequence is defined by a recurrence relation as follow:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned} \tag{6}$$

We can rewrite the case of $n \geq 2$ as a linear system of equation to match the number of unknowns and the number of linear coefficients:

$$\begin{cases} F_{n+1} = \mathbf{1}F_n + \mathbf{1}F_{n-1} \\ F_n \quad\;\; = \mathbf{1}F_n + \mathbf{0}F_{n-1} \end{cases} \tag{7}$$

Which we can factor out as a matrix-vector multiplication, and remove the recurrence by converting it into a matrix power expression:

$$\begin{aligned}\begin{pmatrix}F_{n+1}\\F_n\end{pmatrix} &= \begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}F_n\\F_{n-1}\end{pmatrix} = \begin{pmatrix}1&1\\1&0\end{pmatrix}^2\begin{pmatrix}F_{n-1}\\F_{n-2}\end{pmatrix}\\ &= \begin{pmatrix}1&1\\1&0\end{pmatrix}^k\begin{pmatrix}F_{n-k+1}\\F_{n-k}\end{pmatrix} = \begin{pmatrix}1&1\\1&0\end{pmatrix}^n\begin{pmatrix}F_1\\F_0\end{pmatrix}\\ &= \begin{pmatrix}1&1\\1&0\end{pmatrix}^n\begin{pmatrix}1\\0\end{pmatrix}\end{aligned} \tag{8}$$

Technically we already have a closed form solution, but that's for $(F_{n+1}, F_n)^T$, and contains matrices which people don't like for some reasons, so we can solve for the $F_n$ entry of the matrix using EVD method as described above:

$$\begin{aligned}&\begin{pmatrix}1&1\\1&0\end{pmatrix} = \begin{pmatrix}\varphi&1-\varphi\\1&1\end{pmatrix}\begin{pmatrix}\varphi&0\\0&1-\varphi\end{pmatrix}\begin{pmatrix}\varphi&1-\varphi\\1&1\end{pmatrix}^{-1}\\ &\text{where } \varphi = \frac{1+\sqrt{5}}{2} \text{ is the golden ratio}\\ &\begin{pmatrix}1&1\\1&0\end{pmatrix}^n = \begin{pmatrix}\varphi&1-\varphi\\1&1\end{pmatrix}\begin{pmatrix}\varphi^n&0\\0&(1-\varphi)^n\end{pmatrix}\begin{pmatrix}\varphi&1-\varphi\\1&1\end{pmatrix}^{-1}\\ &\begin{pmatrix}F_{n+1}\\F_n\end{pmatrix} = \begin{pmatrix}\varphi&1-\varphi\\1&1\end{pmatrix}\begin{pmatrix}\varphi^n&0\\0&(1-\varphi)^n\end{pmatrix}\begin{pmatrix}\varphi&1-\varphi\\1&1\end{pmatrix}^{-1}\begin{pmatrix}1\\0\end{pmatrix}\end{aligned} \tag{9}$$

If you carry out the multiplication, extract the $F_n$ component and simplify, you'll get:

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{2\varphi - 1} \tag{10}$$

Which is essentially the same the classical Binet's formula for the Fibonacci sequence.

## 2.2. Practical considerations

Of course, if you want to actually know the numerical value of matrix power, taking the EVD and raising the power of the eigenvalues is far from the best way of doing it. EVD is numerically expensive and does not guarantee to exist for non-symmetric matrices. This method is only useful for finding closed-form solution, which I'd argue that it's only useful for proofs, symbolic manipulations, and extension to real exponents. The Fibonacci example also shown how to convert linear recurrences into matrix power which is useful for later examples.

In practice, for computing $A^k$, we can just use repeated multiplication, or even better, exponentation by squaring. If you use numpy, there's the `np.linalg.matrix_power` function which is an efficient implementation for the latter method. It can also make use of the fact that the matrix entries are integers and avoid the overhead and rounding error of floating point arithmetic. This prompts a question: If the EVD can be used to compute matrix power, can we do the opposite and use matrix power to compute the EVD of a matrix?

# 3. Power iteration

The Power Iteration is a simple algorithm for finding the eigenvector with the largest corresponding eigenvalue of a matrix. The algorithm can be extended for more eigenvectors, but even the largest eigenvector alone is already useful. We can use it to find the direction of maximum variance of a dataset.

The method of Power Iteration is usually described and implemented as follow:

- Start with an initial random vector $\boldsymbol{b}_0$
- At step $k$, update the vector to get $\boldsymbol{b}_{k+1}$ using the following recurrence relation:

$$\boldsymbol{b}_{k+1} = \frac{\boldsymbol{A}\boldsymbol{b}_k}{\|\boldsymbol{A}\boldsymbol{b}_k\|_2} \tag{11}$$

Here's an example implementation in Python with NumPy:

```
import numpy as np

type NPMat = np.ndarray[tuple[int, int], np.dtype[np.float64]]
type NPVec = np.ndarray[tuple[int], np.dtype[np.float64]]
type NPRng = np.random.Generator

def power_iteration(A: NPMat, n_iter: int, rng: NPRng) -> NPVec:
  n, m = A.shape
  assert m == n, f"A ({m}×{n}) is non-square"

  b: NPVec = rng.standard_normal((n,))

  for _ in range(n_iter):
    Ab = A @ b
    b = Ab / np.linalg.norm(Ab)

  return b
```

The algorithm is relatively straightforward to implement and execute with no edge-cases to handle. It also work with black-box matrices, where you pass in $f : \mathbb{R}^n \to \mathbb{R}^n$ instead of the explicit $\mathbb{R}^{n\times n}$ matrix which enhance extensibility.

## 3.1. Proof of convergence

We will prove that the algorithm will converge to a largest eigenvector, if we order the eigenvalue so that their absolute value is in ascending order $|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_n|$, we'll have to show that if $\lim_{k\to\infty} \boldsymbol{b}_k = \boldsymbol{v}$, then $\boldsymbol{A}\boldsymbol{v} = \lambda_1\boldsymbol{v}$. For that, let's take a better look at the recurrence. Similar with the Fibonacci example, we can try to factor it into a matrix power expression like we did in Equation 8.

$$\boldsymbol{b}_{k+1} = \frac{\boldsymbol{A}\boldsymbol{b}_k}{\|\boldsymbol{A}\boldsymbol{b}_k\|_2} = \frac{\boldsymbol{A}\frac{\boldsymbol{A}\boldsymbol{b}_{k-1}}{\|\boldsymbol{A}\boldsymbol{b}_{k-1}\|_2}}{\left\|\boldsymbol{A}\frac{\boldsymbol{A}\boldsymbol{b}_{k-1}}{\|\boldsymbol{A}\boldsymbol{b}_{k-1}\|_2}\right\|_2} = \frac{\frac{1}{\|\boldsymbol{A}\boldsymbol{b}_{k-1}\|_2}}{\frac{1}{\|\boldsymbol{A}\boldsymbol{b}_{k-1}\|_2}}\frac{\boldsymbol{A}^2\boldsymbol{b}_{k-1}}{\|\boldsymbol{A}^2\boldsymbol{b}_{k-1}\|_2} = \frac{\boldsymbol{A}^{k+1}\boldsymbol{b}_0}{\|\boldsymbol{A}^{k+1}\boldsymbol{b}_0\|_2} \tag{12}$$

If we substitute $k+1$ with $k$, we get the following formula for $\boldsymbol{b}_k$, which is the matrix power of $\boldsymbol{A}$ applied to the initial vector $\boldsymbol{b}_0$.

$$\boldsymbol{b}_k = \frac{\boldsymbol{A}^k \boldsymbol{b}_0}{\left\| \boldsymbol{A}^k \boldsymbol{b}_0 \right\|_2} \tag{13}$$

This is much easier to analyze the asymptotic as $k$ approaches $\infty$, we can compute the EVD of $A$ (assuming that it exists), and use the following substitutions:

$$\begin{aligned} \boldsymbol{A}^k = \boldsymbol{P}\boldsymbol{D}^k\boldsymbol{P}^{-1} &= \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & ... & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \lambda_1^k & 0 & ... & 0 \\ 0 & \lambda_2^k & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \lambda_n^k \end{pmatrix} \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & ... & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix}^{-1} \\ \tilde{\boldsymbol{b}} = \begin{pmatrix} \tilde{\boldsymbol{b}}_1 & \tilde{\boldsymbol{b}}_2 & ... & \tilde{\boldsymbol{b}}_n \end{pmatrix}^T &= \boldsymbol{P}^{-1}\boldsymbol{b}_0 \Rightarrow \boldsymbol{b}_0 = \boldsymbol{P}\tilde{\boldsymbol{b}_0} \end{aligned} \tag{14}$$

Then, expanding $\boldsymbol{A}^k\boldsymbol{b}_0$ gives us:

$$\boldsymbol{A}^k\boldsymbol{b}_0 = \boldsymbol{P}\boldsymbol{D}^k\boldsymbol{P}^{-1}\boldsymbol{P}\tilde{\boldsymbol{b}} = \boldsymbol{P}\boldsymbol{D}^k\tilde{\boldsymbol{b}} = \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & ... & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \lambda_1^k & 0 & ... & 0 \\ 0 & \lambda_2^k & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \lambda_n^k \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{b}}_1 \\ \tilde{\boldsymbol{b}}_2 \\ \vdots \\ \tilde{\boldsymbol{b}}_n \end{pmatrix} \tag{15}$$

Carrying out the matrix multiplications, we get a linear combination of eigenvectors:

$$\begin{aligned} \boldsymbol{A}^k\boldsymbol{b}_0 &= \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & ... & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \lambda_1^k\tilde{\boldsymbol{b}}_1 \\ \lambda_2^k\tilde{\boldsymbol{b}}_2 \\ \vdots \\ \lambda_n^k\tilde{\boldsymbol{b}}_n \end{pmatrix} \\ &= \lambda_1^k\tilde{\boldsymbol{b}}_0\boldsymbol{v}_1 + \lambda_2^k\tilde{\boldsymbol{b}}_2\boldsymbol{v}_2 + ... + \lambda_n^k\tilde{\boldsymbol{b}}_n\boldsymbol{v}_n \\ &= \lambda_1^k\left(\tilde{\boldsymbol{b}}_0\boldsymbol{v}_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k\tilde{\boldsymbol{b}}_2\boldsymbol{v}_2 + ... + \left(\frac{\lambda_n}{\lambda_1}\right)^k\tilde{\boldsymbol{b}}_n\boldsymbol{v}_n\right) \end{aligned} \tag{16}$$

Let $m$ be the number of repeated top eigenvalues, we have:

$$|\lambda_1| = ... = |\lambda_m| > \left|\lambda_{m+1}\right| \geq ... \geq |\lambda_n| \tag{17}$$

Assuming $\exists i (1 \leq i \leq m)$ such that $\tilde{\boldsymbol{b}}_i \neq 0$, we have the following limit:

$$\begin{aligned} \lim_{k\to\infty}\left(\frac{\lambda_i}{\lambda_1}\right)^k &= \begin{cases} 1 \text{ if } i \leq m \\ 0 \text{ if } i > m \end{cases} \\ \lim_{k\to\infty}\boldsymbol{A}^k\boldsymbol{b}_0 &= \sum_{i=1}^m \tilde{\boldsymbol{b}}_i\lambda_1^k\boldsymbol{v}_i = \lambda_1^k\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i \\ \lim_{k\to\infty}\frac{\boldsymbol{A}^k\boldsymbol{b}}{\left\|\boldsymbol{A}^k\boldsymbol{b}\right\|_2} &= \lim_{k\to\infty}\frac{\lambda_1^k\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i}{\left\|\lambda_1^k\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i\right\|_2} = \lim_{k\to\infty}\frac{\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i}{\left\|\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i\right\|_2} \end{aligned} \tag{18}$$

All that's left to do is to check for the eigenvector invariance:

$$\boldsymbol{A}\left(\lim_{k\to\infty}\frac{\boldsymbol{A}^k\boldsymbol{b}_0}{\left\|\boldsymbol{A}^k\boldsymbol{b}_0\right\|_2}\right)=\lim_{k\to\infty}\frac{\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{A}\boldsymbol{v}_i}{\left\|\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i\right\|_2} \tag{19}$$

Remember that $v_i(1\leq i\leq m)$ are the eigenvectors corresponding to $\lambda_1$, so $\boldsymbol{A}\boldsymbol{v}_i=\lambda_1\boldsymbol{v}_i$, therefore:

$$\begin{aligned}\boldsymbol{A}\left(\lim_{k\to\infty}\frac{\boldsymbol{A}^k\boldsymbol{b}}{\left\|\boldsymbol{A}^k\boldsymbol{b}\right\|_2}\right)&=\lim_{k\to\infty}\frac{\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\lambda_1\boldsymbol{v}_i}{\left\|\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i\right\|_2}=\lambda_1\lim_{k\to\infty}\frac{\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i}{\left\|\sum_{i=1}^m\tilde{\boldsymbol{b}}_i\boldsymbol{v}_i\right\|_2}\\&=\lambda_1\left(\lim_{k\to\infty}\frac{\boldsymbol{A}^k\boldsymbol{b}}{\left\|\boldsymbol{A}^k\boldsymbol{b}\right\|_2}\right)\end{aligned} \tag{20}$$

Which proves that the limit converges to an eigenvector corresponding the top eigenvalue $\lambda_1$. In the case of a unique top eigenvalue, the limit is just the normalized top eigenvector:

$$\lim_{k\to\infty}\frac{\boldsymbol{A}^k\boldsymbol{b}}{\left\|\boldsymbol{A}^k\boldsymbol{b}\right\|_2}=\frac{\tilde{\boldsymbol{b}}_1\boldsymbol{v}_1}{\left\|\tilde{\boldsymbol{b}}_1\boldsymbol{v}_1\right\|_2}=\frac{\boldsymbol{v}_1}{\left\|\boldsymbol{v}_1\right\|_2} \tag{21}$$

Which is exactly the top eigenvector, and the algorithm doesn't depend on the initial choice $\boldsymbol{b}_0$. Otherwise, the direction of $\boldsymbol{b}_0$ will affect both the speed and result. If $\boldsymbol{b}_0$ is selected such that $\tilde{\boldsymbol{b}}_i=0$ for all $i$ such that $|\lambda_i|=|\lambda_1|$, the Equation 18 is no longer true and the algorithm can only converge to a smaller eigenvector. We also shown via Equation 13 that this algorithm is just computing a really large matrix power, and the recurrence relation described in Equation 11 is just a method for doing it while avoiding overflow.

While the Power Iteration method is a simple algorithm with an interesting connection to matrix power, its efficiency and utility are leave much to be desired. The next sections go over some enhancements of the algorithm and ways to generalize them to get more eigenvectors or even the full decomposition.

## 3.2. Stopping condition and eigenvalue

Since the algorithm is just evaluating a limit as $k$ grows to indefinitely, to implement we just have to pick a large enough $k$, which is the `n_iter` parameter of the example implementation. But how large is "large enough"? The convergence rate depends on the converge rate of $\left|\frac{\lambda_i}{\lambda_1}\right|^k$, which is not constant among all matrices. So if we use a fixed number of iterations, we'll get unreliable precision. To determine when to safely stop the algorithm, we can use the eigenvalue equation $\boldsymbol{A}\boldsymbol{v}=\lambda\boldsymbol{v}$, but replace $\boldsymbol{v}$ and $\lambda$ with the approximation $\boldsymbol{b}$ and $\mu$. Rearranging, we get:

$$\boldsymbol{b}\mu=\boldsymbol{A}\boldsymbol{b} \tag{22}$$

If $\boldsymbol{b}$ haven't converged to $\boldsymbol{v}$, Equation 22 has no solution, but we convert it into a least squares problem and minimize the residual:

$$\left(\hat{\boldsymbol{b}}, \hat{\mu}\right) = \arg\min_{\boldsymbol{b},\mu} \|\boldsymbol{b}\mu - \boldsymbol{A}\boldsymbol{b}\|_2^2 \tag{23}$$

$\boldsymbol{b}$ is already optimized by Power Iteration, so if we fix $\boldsymbol{b}$ and try to optimize for $\mu$, we have:

$$\hat{\mu}(\boldsymbol{b}) = \arg\min_{\mu} \|\boldsymbol{b}\mu - \boldsymbol{A}\boldsymbol{b}\|_2^2 = \frac{\boldsymbol{b}^T\boldsymbol{A}\boldsymbol{b}}{\boldsymbol{b}^T\boldsymbol{b}} \tag{24}$$

This is call the Rayleigh quotient of the vector $\boldsymbol{b}$, or the best approximation of the eigenvalue given an eigenvector approximation $\boldsymbol{b}$. We will come back to this later, but currently our objective is to find a stopping condition. For this, we can use the error of Equation 22.

$$\|\boldsymbol{b}\mu - \boldsymbol{A}\boldsymbol{b}\|_2^2 = \left\|\boldsymbol{b}\frac{\boldsymbol{b}^T\boldsymbol{A}\boldsymbol{b}}{\boldsymbol{b}^T\boldsymbol{b}} - \boldsymbol{A}\boldsymbol{b}\right\|_2^2 = \left\|\frac{\boldsymbol{b}\boldsymbol{b}^T\boldsymbol{A}\boldsymbol{b}}{\boldsymbol{b}^T\boldsymbol{b}} - \boldsymbol{A}\boldsymbol{b}\right\|_2^2 = \left\|\left(\frac{\boldsymbol{b}\boldsymbol{b}^T}{\boldsymbol{b}^T\boldsymbol{b}} - I_n\right)\boldsymbol{A}\boldsymbol{b}\right\|_2^2 \tag{25}$$

Since $b$ is normalized every iteration, you can drop the $\boldsymbol{b}^T\boldsymbol{b}$ term for simplification. Instead of running for a fixed number of iterations, we can run until the error is below a fixed tolerance.

```
def power_iteration(A: NPMat, rng: NPRng, tol=1e-12) -> NPVec:
  n, m = A.shape
  assert m == n, f"A ({m}×{n}) is non-square"

  I = np.eye(n)
  b: NPVec = rng.standard_normal((n,))
  b /= np.linalg.norm(b)

  while True:
    Ab = A @ b
    r = (np.outer(b) - I) @ Ab
    if r @ r < tol:
      return b

    b = Ab / np.linalg.norm(Ab)
```

Most of the time parametrizing by tolerance is preferable because getting accurate results is more important. In practice, you might want to throw an error after a fixed number of iterations if it haven't converged. The choice of the tolerance is also interesting I wont go into details here. More importantly, we can also extend the algorithm to get the largest eigenvalue using the Rayleigh quotient described in Equation 24.

## 4. Rayleigh quotient Iteration

But we can do more that just returning the eigenvalue. If we incorporate the eigenvalue into the iteration process, and if we don't need the top eigenvector, there's an algorithm with a much better convergence rate. It's called Rayleigh quotient Iteration (RQI). And as its name suggest, it extend power iteration with the Rayleigh quotient to find an eigenvector close to the initial approximation. The convergence behavior of RQI is much more complicated than plain Power Iteration, so the method won't get analyzed in detail. But I'll show how to get from power iteration to RQI.

## 4.1. Inverse Iteration

The first step is to instead find the eigenvector with the smallest absolute eigenvalue. We can use the following result:

$$\boldsymbol{A}^{-1} = \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \frac{1}{\lambda_1} & 0 & \dots & 0 \\ 0 & \frac{1}{\lambda_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{\lambda_n} \end{pmatrix} \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix}^{-1} \tag{26}$$

You can easily verify this by multiplying it with $\boldsymbol{A}$ or its EVD, which indeeds result in the identity matrix. This allows us to extend the matrix power formula to negative numbers by inverting and raising to the absolute power. But more importantly, if $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_n|$, then $\left|\frac{1}{\lambda_1}\right| \leq \left|\frac{1}{\lambda_2}\right| \leq \dots \leq \left|\frac{1}{\lambda_n}\right|$, in other words, the inverse of a matrix $\boldsymbol{A}^{-1}$ has the same eigenvectors of the original matrix, but eigenvalues have their order of absolute values reversed. So to find the smallest eigenvector, we can apply Power iteration to $\boldsymbol{A}^{-1}$, or:

$$\boldsymbol{b}_{k+1} = \frac{\boldsymbol{A}^{-1}\boldsymbol{b}_k}{\left\|\boldsymbol{A}^{-1}\boldsymbol{b}_k\right\|_2} \tag{27}$$

Note that if $\boldsymbol{A}$ is singular, then the smallest eigenvalue is $0$, and the eigenvector is in the nullspace of $\boldsymbol{A}$ which needs to be computed with a different method. Also, if the smallest eigenvalue is near $0$, inverting $\boldsymbol{A}$ will be extremely unstable. So instead of inverting, we can instead solve a linaer system of equation:

$$\boldsymbol{A}\boldsymbol{b}_{k+1} = \boldsymbol{b}_k \tag{28}$$

Then we can normalize the result to avoid overflowing. The system can be solved by numerically stable methods, such as LU Decomposition in the general case, Bunch-Kaufman factorization if the matrix is symmetric, and Cholesky decomposition if the matrix is positive-definite. The most expensive part of these method is the matrix decomposition, but it only have to be done once at the start of the algorithm.

## 4.2. The Rayleigh quotient method

Once we can find the smallest eigenvector, we can easily get the eigenvector close to an initial eigenvector. Instead of minimizing $|\lambda|$, we can instead minimize $|\lambda - \mu|$, where $\mu$ is our target eigenvalue. To subtract $\mu$ from each eigenvalue while retain the eigenvectors, we can use this matrix:

$$\begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \lambda_1 - \mu & 0 & \dots & 0 \\ 0 & \lambda_2 - \mu & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n - \mu \end{pmatrix} \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix}^{-1} \tag{29}$$

Which you can work out to be $\boldsymbol{A} - \mu\boldsymbol{I}_n$. To find the smallest eigenvector of this matrix, we can apply Inverse Iteration as described in the previous section.

But what value of $\mu$ should we use, the closer $\mu$ is to an eigenvalue $\lambda$, the faster the algorithm will converge as $\left|\frac{1}{\mu-\lambda}\right|$ quickly dominate. And how to get an eigenvalue approximation from an eigenvector? The Rayleigh quotient. Therefore, the update step of RQI is as follow:

$$\boldsymbol{b}_{k+1} = \frac{\left(\boldsymbol{A} - I_n \boldsymbol{b}_k^T \boldsymbol{A} \boldsymbol{b}_k\right)^{-1} \boldsymbol{b}_k}{\left\|\left(\boldsymbol{A} - I_n \boldsymbol{b}_k^T \boldsymbol{A} \boldsymbol{b}_k\right)^{-1} \boldsymbol{b}_k\right\|_2} \tag{30}$$

Again, use a numerically stable system solver instead of inverting, you can even decompose $A$ once at the start and perform a rank-1 update each iteration. This algorithm is very fast, but since each iteration is expensive, you may want to initialize it with a good approximation of $b_0$. This is why RQI is often used as an "Eigenpair refinement algorithm", where you have a crude approximation for an eigenvector/eigenvalue pair, and want to reduce the approximation error. Here's an implementation that requires at least an initial eigenvector:

```
def rayleigh_quotient_iteration(
  A: NPMat,
  b: NPVec,
  mu: float | None = None,
  tol=1e-12,
) -> tuple[float, NPVec]:
  n, m = A.shape
  assert m == n, f"A ({m}×{n}) is non-square"

  I = np.eye(n)
  b /= np.linalg.norm(b)
  mu = b.T @ A @ b if mu is None else mu

  while True:
    try:
      w = np.linalg.solve(A - I * mu)
    except:
      return mu, b

    b = w / np.linalg.norm()
    Ab = A @ b
    mu = b.T @ Ab
    r = mu * b - Ab

    if r @ r < tol:
      return mu, b
```

## 5. Deterministic Power Iteration

RQI is a good choice for improving the precision of eigenvector/eigenvalue pairs, but when you need the top eigenvector, such as when finding the direction of maximum variance, you either have to compute all of them or go back to Power Iteration. Which means that you have to give up the fast convergence rate of RQI.

Another problem that plagues both method is the sensivity to initialization. There's nothing wrong with randomized algorithms, but it's interesting to see if we can make Power Iteration work without having to select the initial vector. Just a head up about practicality, what I'm

about to describe here takes away one big advantage of power iteration: its ability to work efficiently with sparse and black-box matrices.

In practice, if the matrix is dense, you're better of using more advanced methods to compute all eigenvectors and pick the one with the largest eigenvalue, since these algorithm converges much faster anyways. But an advantage of this method is that it's reayy simple to implement, which may or may not justify its usage.

## 5.1. The limit of normalized matrix power

The idea is to remove $b_0$ from power iteration formula, and just see what it converges to:

$$\lim_{k\to\infty} \frac{\boldsymbol{A}^k}{\|\boldsymbol{A}^k\|_F} \tag{31}$$

Where $\|\boldsymbol{A}\|_F$ is the Frobenius norm, which can be expressed as:

$$\|\boldsymbol{A}\|_F = \sqrt{\text{tr}(\boldsymbol{A}^T\boldsymbol{A})} = \sqrt{\lambda_1^2 + \lambda_2^2 + ... + \lambda_n^2} \tag{32}$$

Evaluating the limit similar to Section 3.1. gives us:

$$\begin{aligned}
\lim_{k\to\infty} \boldsymbol{A}^k &= \lambda_1^k \sum_{i=1}^{m} \boldsymbol{v}_i \boldsymbol{w}_i \\
\lim_{k\to\infty} \|\boldsymbol{A}^k\|_F &= \lambda_1^k \sqrt{m} \\
\lim_{k\to\infty} \frac{\boldsymbol{A}^k}{\|\boldsymbol{A}^k\|_F} &= \frac{1}{\sqrt{m}} \sum_{i=1}^{m} \boldsymbol{v}_i \boldsymbol{w}_i
\end{aligned} \tag{33}$$

Where $(\boldsymbol{w}_1^T, \boldsymbol{w}_2^T, ..., \boldsymbol{w}_n^T) \in \mathbb{R}^{n\times n}$ are the rows of $P^{-1}$, in other words:

$$\boldsymbol{A} = \boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{-1} = \begin{pmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & ... & \boldsymbol{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 & ... & 0 \\ 0 & \lambda_2 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \lambda_n \end{pmatrix} \begin{pmatrix} — & \boldsymbol{w}_1 & — \\ — & \boldsymbol{w}_2 & — \\ & \vdots & \\ — & \boldsymbol{w}_2 & — \end{pmatrix} \tag{34}$$

We can rewrite the result of the limit in Matrix form:

$$\lim_{k\to\infty} \frac{\boldsymbol{A}^k}{\|\boldsymbol{A}^k\|_F} = \begin{pmatrix} | & & | & & | \\ \boldsymbol{v}_1 & ... & \boldsymbol{v}_m & ... & \boldsymbol{v}_n \\ | & & | & & | \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{m}} & ... & 0 & ... & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & ... & \frac{1}{\sqrt{m}} & ... & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & ... & 0 & ... & 0 \end{pmatrix} \begin{pmatrix} — & \boldsymbol{w}_1 & — \\ & \vdots & \\ — & \boldsymbol{w}_m & — \\ & \vdots & \\ — & \boldsymbol{w}_n & — \end{pmatrix} \tag{35}$$

Then take the matrix-vector product of the limit and a vector $\boldsymbol{b}$, we get:

$$\begin{aligned}\left(\lim_{k\to\infty}\frac{\boldsymbol{A}^k}{\|\boldsymbol{A}^k\|_F}\right)\boldsymbol{b} &= \left(\lim_{k\to\infty}\frac{\boldsymbol{A}^k}{\|\boldsymbol{A}^k\|_F}\right)\boldsymbol{P}\tilde{\boldsymbol{b}} \\ &= \begin{pmatrix} | & & | & & | \\ \boldsymbol{v}_1 & \dots & \boldsymbol{v}_m & \dots & \boldsymbol{v}_n \\ | & & | & & | \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{m}} & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & \frac{1}{\sqrt{m}} & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{b}}_1 \\ \vdots \\ \tilde{\boldsymbol{b}}_m \\ \vdots \\ \tilde{\boldsymbol{b}}_n \end{pmatrix} \\ &= \frac{1}{\sqrt{m}}\sum_{i=0}^{m}\boldsymbol{v}_i\tilde{\boldsymbol{b}}_i \end{aligned} \tag{36}$$

So this means that we get the same result as plain Power Iteration, but instead of selecting a vector at the start and hope that it's not orthogonal to the largest eigenvector, we can evaluate the limit **then** compute the vector product. If we encounter an orthogonal vector then we can just discard it and try another vector. We can't do this with plain Power Iteration because everytime we try another vector, we have to restart the entire iteration process.

Moreover, we can try trivial vectors, such as the standard basis vectors $(e_1, e_2, ..., e_n)$ and just pick any non-zero product. If we exhaust all standard basis vector and can't find a non-zero product then we can safely confirm that $\boldsymbol{A}$ is the zero matrix and any vector is the largest eigenvector. Multiplying a matrix with a standard basis vector $e_i$ is essentially extracting the $i$th column of the matrix, so we can return the column with the largest norm as a robust non-zero eigenvector. This gives us a deterministic implementation as follow:

```python
def matrix_power_iteration(A: NPMat, n_iter: int) -> NPVec:
  n, m = A.shape
  assert m == n, f"A ({m}×{n}) is non-square"

  norm2 = (A * A).sum()
  if norm2 < np.finfo(A.dtype).eps:
    v = np.zeros_like(A[0])
    v[0] = 1
    return v

  B = A.copy()
  for _ in range(n_iter):
    B /= np.linalg.norm(B)
    B = A @ B

  col_norm = np.linalg.norm(B, axis=0)
  max_col = col_norm.argmax()
  return B[:, max_col] / col_norm[max_col]
```

Again, there's the concern about stopping condition, and for that you can use the squared norm of the difference between consecutive iterations. But instead of that I'm going to propose a simple modification to the algorithm that make it so fast that you don't need a stopping condition.

## 5.2. Exponentiation by squaring

Back in Section 2.2., I mentioned how to compute $A^k$, we can use a method called exponentiation by squaring, and `np.linalg.matrix_power` implements it. It turns out that you can easily use it decrease the number of iterations. We can rewrite the iteration step:

$$\begin{aligned} B_k &= \frac{A^k}{\|A^k\|_F} \\ B_{2k} &= \frac{A^k A^k}{\|A^k A^k\|_F} = \frac{\frac{A^k}{\|A^k\|_F}\frac{A^k}{\|A^k\|_F}}{\left\|\frac{A^k}{\|A^k\|_F}\frac{A^k}{\|A^k\|_F}\right\|_F} \\ &= \frac{B_k B_k}{\|B_k B_k\|_F} = \frac{B_k^2}{\|B_k^2\|_F} \end{aligned} \tag{37}$$

Therefore instead of multiplying by $A$ each step, we can just square and renormalize. This basically reduce the number of iterations exponentially, so if previously we need 1 million iterations, now we only need about 20. And the cost per iteration is the same as the previous method as we just change what matrix to multiply by each step.

```
def squaring_power_iteration(A: NPMat) -> NPVec:
  n, m = A.shape
  assert m == n, f"A ({m}×{n}) is non-square"

  norm2 = (A * A).sum()
  if norm2 < np.finfo(A.dtype).eps:
    v = np.zeros_like(A[0])
    v[0] = 1
    return v

  for _ in range(24):
    A /= np.linalg.norm(A)
    A = A @ A

  col_norm = np.linalg.norm(A, axis=0)
  max_col = col_norm.argmax()
  return A[:, max_col] / col_norm[max_col]
```

In the code, I fixed the number of iterations to 24, because that's more than enough for practical purposes. This gives us a simple, fast, and deterministic algorithm for finding the top eigenvector, but since matrix-matrix multiplication is $\mathcal{O}(n^3)$, this should only be used for small matrices, and when you only care about the top eigenvector. Otherwise, switch back to plain power iteration or find all eigenvectors with a different method.

Another important thing to note is that it's not necessary to use the Frobenius norm for each iteration step. You can use other norm that's easier to compute such as the maximum component norm $\max_{i,j}|A_{i,j}|$ and it will still converge as long as the submultiplicativity property is preserved. In case you really want the algorithm to fully converge regardless of the input matrix, set the number of iteration to the number of bits of the machine precision.