16 The Arnoldi Iteration

Lab Objective: *The Arnoldi Iteration is an efficient method for finding the eigenvalues of extremely large matrices. Instead of using standard methods, the iteration uses Krylov subspaces to approximate how a linear operator acts on vectors. With this approach, the Arnoldi Iteration facilitates the computation of eigenvalues for enormous matrices without needing to physically create the matrix in memory. We will explore this subject by implementing the Arnoldi iteration algorithm, using our implementation for eigenvalue computation, and then graphically representing the accuracy of our approximated eigenvalues.*

Krylov Subspaces

One of the biggest difficulties in numerical linear algebra is the amount of memory needed to store a large matrix and the amount of time needed to read its entries. Methods using Krylov subspaces avoid this difficulty by studying how a matrix acts on vectors, making it unnecessary in many cases to create the matrix itself.

The *Arnoldi Iteration* is an algorithm for finding an orthonormal basis of a Krylov subspace. One of its strengths is that it can run on any linear operator without knowing the operator's underlying matrix representation. The outputs of the Arnoldi algorithm can then be used to approximate the eigenvalues of the matrix of the linear operator.

The order-n Krylov subspace of A generated by $\mathbf{x}$ is

$$\mathcal{K}_n(A, \mathbf{x}) = \text{span}\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}.$$

If the vectors $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}$ are linearly independent, then they form a basis for $\mathcal{K}_n(A, \mathbf{x})$. However, $A^n x$ frequently converges to a dominant eigenvector of A as n gets large, which fills the basis with many almost parallel vectors. This yields a basis prone to ill-conditioned computations and numerical instability.

The Arnoldi Iteration Algorithm

The Arnoldi iteration focuses on efficiently creating an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$ by integrating the creation of $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}$ with the modified Gram-Schmidt algorithm. This process yields an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$ that can be used for further computations.

The algorithm begins by initializing a matrix H which will be an upper Hessenberg matrix and a matrix Q which will be filled with the basis vectors of our Krylov subspace. It also requires an initial vector $\mathbf{b} \neq 0$ which is normalized to get $\mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|$. This represents the basis for the initial Krylov subspace, $\mathcal{K}_1(A, \mathbf{b})$.

For the kth iteration, compute the next basis vector $\mathbf{q}_{k+1}$ by using the modified Gram-Schmidt process to make $A\mathbf{q}_k$ orthonormal to $\mathbf{q}_k$. This entails making each column of Q orthogonal to $\mathbf{q}_k$ before proceeding to the next iteration. The vectors $\{\mathbf{q}_i\}_{i=1}^{k}$ are then a basis for $\mathcal{K}_k(A, \mathbf{b})$. If $\|\mathbf{q}_{k+1}\|$ is below a certain tolerance, stop and return H and Q. Otherwise, normalize the new basis vector new $\mathbf{q}_{k+1}$ and continue to the next iteration.

Algorithm 16.1 The Arnoldi iteration. This algorithm accepts a square matrix A and a starting vector $\mathbf{b}$. It iterates k times or until the norm of the next vector in the iteration is less than `tol`. The algorithm returns an upper Hessenberg H and an orthonormal Q such that $H = Q^{\mathsf{H}}AQ$.

procedure ARNOLDI($\mathbf{b}$, A, k, `tol`)
  $Q \leftarrow$ empty(size($\mathbf{b}$), $k+1$) ▷ Some initialization steps
  $H \leftarrow$ zeros($k+1, k$)
  $Q_{:,0} \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$
  for $j = 0 \dots k-1$ **do** ▷ Perform the actual iteration.
    $Q_{:,j+1} \leftarrow A(Q_{:,j})$
    for $i = 0 \dots j$ **do** ▷ Modified Gram-Schmidt.
      $H_{i,j} \leftarrow Q_{:,i}^{\mathsf{H}} Q_{:,j+1}$
      $Q_{:,j+1} \leftarrow Q_{:,j+1} - H_{i,j} Q_{:,i}$
    $H_{j+1,j} \leftarrow \|Q_{:,j+1}\|_2$ ▷ Set subdiagonal element of H.
    if $|H_{j+1,j}| <$ `tol` **then** ▷ Stop if $\|Q_{:,j+1}\|_2$ is small enough.
      return $H_{:j+1,:j+1}$, $Q_{:,:j+1}$
    $Q_{:,j+1} \leftarrow Q_{:,j+1}/H_{j+1,j}$ ▷ Normalize $\mathbf{q}_{j+1}$.
  return $H_{:-1,:}$, Q ▷ Return H_k and Q.

ACHTUNG!

If the starting vector $\mathbf{x}$ is an eigenvector of A with corresponding eigenvalue λ, then by definition $\mathcal{K}_k(A, \mathbf{x}) = \text{span}\{\mathbf{x}, \lambda\mathbf{x}, \lambda^2\mathbf{x}, \dots, \lambda^k\mathbf{x}\}$, which is equal to the span of $\mathbf{x}$. So, when $\mathbf{x}$ is normalized with $\mathbf{q}_1 = \mathbf{x}/\|\mathbf{x}\|$, $\mathbf{q}_2 = A\mathbf{q}_1 = \lambda\mathbf{q}_1$.

The vector $\mathbf{q}_2$ is supposed to be the next vector in the orthonormal basis for $\mathcal{K}_k(A, \mathbf{x})$, but it is not linearly independent of $\mathbf{q}_1$. In fact, $\mathbf{q}_1$ already spans $\mathcal{K}_k(A, \mathbf{x})$. Hence, the Gram-Schmidt process fails and results in a `ZeroDivisionError` or an extremely early termination of the algorithm. A similar phenomenon may occur if the starting vector $\mathbf{x}$ is contained in a proper invariant subspace of A.

Arnoldi Iteration on Linear Operators

A major strength of the Arnoldi iteration is that it can run on a linear operator, even without knowing the matrix representation of the operator. If L is some linear function, then we can modify the pseudocode above by replacing $AQ_{:,j}$ with $A_{mul}(Q_{:,j})$. This makes it possible to find the eigenvalues of an arbitrary linear transformation.

Problem 1. Write a function that accepts a starting vector $\mathbf{b}$ for the Arnoldi Iteration, a function handle L that describes a linear operator, the number of times n to perform the iteration, and a tolerance `tol` that defaults to 10^{-8}. Use Algorithm 16.1 to implement the Arnoldi Iteration with these parameters. Return the upper Hessenberg matrix H and the orthonormal matrix Q from the iteration.

Consider the following implementation details.

1. Since H and Q will eventually hold complex numbers, initialize them as complex arrays (e.g., `A = np.empty((3,3), dtype=np.complex128)`).
2. This function can be tested on a matrix A by passing in `A.dot` for a linear operator.
3. Remember to use complex inner products. Here is an example of how to evaluate $A^H A$:

```
b = A.conj() @ B
```

Test your function by comparing the resulting H with $Q^{\mathsf{H}}AQ$.

Finding Eigenvalues Using the Arnoldi Iteration

Let A be an $n \times n$ matrix. Let Q_k be the matrix whose columns $\mathbf{q}_1, \ldots, \mathbf{q}_k$ are the orthonormal basis for $\mathcal{K}_m(A, \mathbf{x})$ generated by the Arnoldi algorithm, and let H_k be the $k \times k$ upper Hessenburg matrix defined at the kth stage of the algorithm. Then these matrices satisfy

$$H_k = Q_k^{\mathsf{H}} A Q_k. \tag{16.1}$$

If $k < n$, then H_k is a low-rank approximation to A and the eigenvalues of H_k may be used as approximations for the eigenvalues of A. The eigenvalues of H_k are called *Ritz Values*, and we will later show that they converge quickly to the largest eigenvalues of A.

Problem 2. Write a function that accepts a function handle L that describes a linear operator, the dimension of the space `dim` that the linear operator works on, the number of times k to perform the Arnoldi Iteration, and the number of Ritz values n to return. Use the previous implementation of the Arnoldi Iteration and an eigenvalue function such as `scipy.linalg.eigs()` to compute the largest Ritz values of the given operator. Return the `n` largest Ritz values.

One application of the Arnoldi iteration is to find the eigenvalues of linear operators that are too large to store in memory. For example, if an operator acts on a vector $\mathbf{x} \in \mathbb{C}^{2^{20}}$, then its matrix representation contains 2^{40} complex values. Storing such a matrix would require 64 terabytes of memory!

An example of such an operator is the Fast Fourier Transform, cited by SIAM as one of the top algorithms of the century [Cip00]. The Fast Fourier Transform is used very commonly in signal processing.

Problem 3. The four largest eigenvalues of the Fast Fourier Transform are known to be $\{-\sqrt{n}, \sqrt{n}, -i\sqrt{n}, i\sqrt{n}\}$ where n is the dimension of the space on which the transform acts.

Use your function from Problem 2 to approximate the eigenvalues of the Fast Fourier Transform. Set $k = 10$ and `dim` $= 2^{20}$. For the argument L, use the `scipy.fftpack.fft()`.

The Arnoldi iteration for finding eigenvalues is implemented in a Fortran library called ARPACK. Scipy interfaces with the Arnoldi iteration in this library via the function `scipy.sparse.linalg.eigs()`. This function has many more options than the implementation we wrote in Problem 2. In this example, the keyword argument `k=5` specifies that we want five Ritz values. Note that even though this function comes from the `sparse` library in Scipy, we can still call it on regular Numpy arrays.

```
>>> from scipy.sparse import linalg as spla
>>> B = np.random.random((100,100))
>>> spla.eigs(B, k=5, return_eigenvectors=false)
array([ -1.15577072-2.59438308j,  -2.63675878-1.09571889j,
        -2.63675878+1.09571889j,  -3.00915592+0.j        ,  50.14472893+0.j ])
```

Convergence

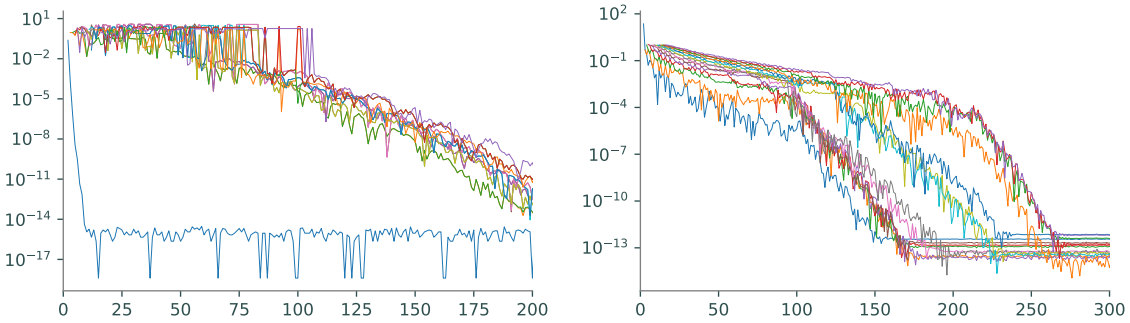
As more iterations of the Arnoldi method are performed, our approximations are of higher rank. Consequently, the Ritz values become more accurate approximations to the eigenvalues of the linear operator.

This technique converges quickly to eigenvalues whose magnitude is distinctly larger than the rest. For example, matrices with random entries tend to have one eigenvalue of distinctly greatest magnitude. Convergence of the Ritz values for such a matrix is plotted in Figure 16.1a.

However, Ritz values converge more slowly for matrices with random eigenvalues. Figure 16.1b plots convergence of the Ritz values for a matrix with eigenvalues uniformly distributed in $[0, 1)$.

Problem 4. Write a function that accepts a linear operator A, the number of Ritz values to plot n, and the the number of times to perform the Arnoldi iteration `iters`. Use these parameters to create a plot of the absolute error between the largest Ritz values of A and the largest eigenvalues of A.

1. Find n eigenvalues of A of largest magnitude. Store these in order.
2. Create an empty array to store the relative errors for every $k = 0, 1, \ldots,$ `iters`.
   (a) Use your Ritz function to find the n largest Ritz values of the operator. Note that for small k, the matrix H_k may not have this many eigenvalues. Due to this, the graphs of some eigenvalues have to begin after a few iterations.
   (b) Store the absolute error between the eigenvalues of A and the Ritz values of H. Make sure that the errors are stored in the correct order.
3. Iteratively plot the errors for each eigenvalue with the range of the iterations.



(a) (b)

Figure 16.1: These plots show the relative error of the ritz values as approximations to the eigenvalues of a matrix. The figure on the left plots the largest 15 Ritz values for a 500×500 matrix with random entries and demonstrates that the largest eigenvalue (the blue line) converges after 20 iterations. The figure at right plots the largest 15 Ritz values for a 500×500 matrix with uniformly distributed eigenvalues in $[0, 1)$ and demonstrates that all the eigenvalues take from 150 to 250 iterations to converge.

Hints: If $\tilde{\mathbf{x}}$ is an an approximation to $\mathbf{x}$, then the *absolute error* in the approximation is $\|\mathbf{x} - \tilde{\mathbf{x}}\|$.

Sort your eigenvalues from greatest to least. An example of how to do this is included:

```
# Evaluate the eigenvalues
eigvalues = la.eig(A)[0]
# Sort them from greatest to least (use np.abs to account for complex ↩
    parts)
eigvalues = eigvalues[np.sort(np.abs(eigvalues))[::-1]]
```

In addition, remember that certain eigenvalues of H will not appear until we are computing enough iterations in the Arnoldi algorithm. As a result, we will have to begin the graphs of several eigenvalues after we are computing sufficient iterations of the algorithm.

Run your function on these examples. The plots should be fairly similar to Figures 16.1b and 16.1a.

```
>>> A = np.random.rand(300, 300)
>>> plot_ritz(a, 10, 175)

>>> # A matrix with uniformly distributed eigenvalues
>>> d = np.diag(np.random.rand(300))
>>> B = A @ d @ la.inv(A)
>>> plot_ritz(B, 10, 175)
```

Additional Material

The Lanczos Iteration

The Lanczos iteration is a version of the Arnoldi iteration that is optimized to operate on symmetric matrices. If A is symmetric, then (16.1) shows that H_k is symmetric and hence tridiagonal. This leads to two simplifications of the Arnoldi algorithm.

First, we have $0 = H_{k,n} = \langle \mathbf{q}_k, A\mathbf{q}_n \rangle$ for $k \leq n-2$; i.e., $A\mathbf{q}_n$ is orthogonal to $\mathbf{q}_1, \dots, \mathbf{q}_{n-2}$. Thus, if the goal is only to compute H_k (say to find the Ritz values), then we only need to store the two most recently computed columns of Q. Second, the data of H_k can also be stored in two vectors, one containing the main diagonal and one containing the first subdiagonal of H_k (by symmetry, the first superdiagonal equals the first subdiagonal of H_k).

Algorithm 16.2 The Lanczos Iteration. This algorithm operates on a vector $\mathbf{b}$ of length n and an $n \times n$ symmetric matrix A. It iterates k times or until the norm of the next vector in the iteration is less than *tol*. It returns two vectors $\mathbf{x}$ and $\mathbf{y}$ that respectively contain the main diagonal and first subdiagonal of the current Hessenberg approximation.

```
1:  procedure LANCZOS(b, A, k, tol)
2:      q_0 ← zeros(size(b))                          ▷ Some initialization
3:      q_1 ← b/‖b‖_2
4:      x ← empty(k)
5:      y ← empty(k)
6:      for i = 0...k−1 do                            ▷ Perform the iteration.
7:          z ← Aq_1                                  ▷ z is a temporary vector to store q_{i+1}.
8:          x[i] ← q_1^T z                            ▷ q_1 is used to store the previous q_i.
9:          z ← z − x[i]q_1 + y[i−1]q_0               ▷ q_0 is used to store q_{i−1}.
10:         y[i] = ‖z‖_2                              ▷ Initialize y[i].
11:         if y[i] < tol then                        ▷ Stop if ‖q_{i+1}‖_2 is too small.
12:             return x[: i+1], y[: i]
13:         z = z/y[i]
14:         q_0, q_1 = q_1, z                         ▷ Store new q_{i+1} and q_i on top of q_1 and q_0.
15:     return x, y[: −1]
```

As it is described in Algorithm 16.2, the Lanczos iteration is not stable. Roundoff error may cause the $\mathbf{q}_i$ to be far from orthogonal. In fact, it is possible for the $\mathbf{q}_i$ to be so adulterated by roundoff error that they are no longer linearly independent.

There are modified versions of the Lanczos iteration that are numerically stable. One of these, the Implicitly Restarted Lanczos Method, is found in SciPy as `scipy.sparse.linalg.eigsh()`.

Bibliography

[Cip00] Barry A. Cipra. The best of the 20th century: editors name top 10 algorithms. *Siam news*, 33(4), 16 May 2000. [3]