Lab 1

Krylov Subspaces

Lab Objective: *Use Krylov subspaces to find eigenvalues of extremely large matrices.*

One of the biggest difficulties in computational 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 it can run on any linear operator without knowing the operator's underlying matrix representation. The outputs of the Arnoldi algorithm can be used to approximate the eigenvalues of the matrix of the linear operator.

Krylov Subspaces

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, this basis is usually far from orthogonal, and hence computations using this basis will likely be ill-conditioned.

The Arnoldi Iteration Algorithm

One way to find an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$ is to use the modified Gram-Schmidt algorithm from Lab **??** on the set $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}$. The Arnold iteration does this more efficiently 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. It returns an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$. This algorithm is described in Algorithm 1.1.

In Algorithm 1.1, k is the number of times we multiply by A. This will result in an order-$k+1$ Krylov subspace.

Algorithm 1.1 The Arnoldi Iteration. This algorithm accepts a square matrix A and 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 upper Hessenberg H and orthonormal Q such that $H = Q^{\mathsf{H}}AQ$.

1: **procedure** ARNOLDI($\mathbf{b}, A, k, tol$)
2: $Q \leftarrow \text{empty}(\text{size}(\mathbf{b}), k+1)$ ▷ Some initialization steps
3: $H \leftarrow \text{zeros}(k+1, k)$
4: $Q[:, 0] \leftarrow \mathbf{b} / \|\mathbf{b}\|_2$
5: **for** $j = 0 \dots k-1$ **do** ▷ Perform the actual iteration.
6: $Q[:, j+1] \leftarrow AQ[:, j]$
7: **for** $i = 0 \dots j$ **do** ▷ Modified Gram-Schmidt.
8: $H[i, j] \leftarrow Q[:, i]^{\mathsf{T}} Q[:, j+1]$
9: $Q[:, j+1] \leftarrow Q[:, j+1] - H[i, j]Q[:, i]$
10: $H[j+1, j] \leftarrow \|Q[:, j+1]\|_2$ ▷ Set subdiagonal element of H.
11: **if** $|H[j+1, j]| < tol$ **then** ▷ Stop if $\|Q[:, j+1]\|_2$ is too small.
12: **return** $H[: j+1, : j+1]$, $Q[:, : j+1]$
13: $Q[:, j+1] \leftarrow Q[:, j+1] / H[j+1, j]$ ▷ Normalize $\mathbf{q}_{j+1}$.
14: **return** $H[: -1, :]$, Q ▷ Return H_k.

Something perhaps unexpected happens in the Arnoldi iteration if the starting vector $\mathbf{x}$ is an eigenvector of A. If the corresponding eigenvalue is 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 $\mathbf{x}$. Let us trace through Algorithm 1.1 in this case. We will use $\mathbf{q}_i$ to denote the i^{th} column of Q.

In line 4 we normalize $\mathbf{x}$, setting $\mathbf{q}_1 = \mathbf{x}/\|\mathbf{x}\|$. In line 6 we set $\mathbf{q}_2 = A\mathbf{q}_1 = \lambda\mathbf{q}_1$. Then in line 8

$$H_{1,1} = \langle \mathbf{q}_1, \mathbf{q}_2 \rangle = \langle \mathbf{q}_1, \lambda\mathbf{q}_1 \rangle = \lambda \langle \mathbf{q}_1, \mathbf{q}_1 \rangle = \lambda,$$

so in line 9 we subtract $\lambda\mathbf{q}_1$ from $\mathbf{q}_2$, ending with $\mathbf{q}_2 = 0$.

The vector $\mathbf{q}_2$ is supposed to be the next vector in the orthonormal basis for $\mathcal{K}_k(A, \mathbf{x})$, but since it is 0, it is not linearly independent of $\mathbf{q}_1$. In fact, $\mathbf{q}_1$ already spans $\mathcal{K}_k(A, \mathbf{x})$. Hence, when in line 11 we find that the norm of $\mathbf{q}_2$ is zero (or close to it, allowing for numerical error), we terminate the algorithm early, returning the 1×1 matrix $H = H_{1,1} = \lambda$ and the $n \times 1$ matrix $Q = \mathbf{q}_1$.

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 A_{mul} is some linear function, then we can modify the pseudocode above by replacing $AQ[:, j]$ with $A_{mul}(Q[:, j])$. This will make it possible to find the eigenvalues of an arbitrary linear transformation. We will use this method in the problem below.

Problem 1. Using Algorithm 1.1, complete the following Python function that performs the Arnoldi iteration. Write this function so that it can run on complex arrays.

```
def arnoldi(b, Amul, k, tol=1E-8):
    '''Perform `k' steps of the Arnoldi iteration on the linear operator
    defined by `Amul', starting with the vector 'b'.

    INPUTS:
    b    - A NumPy array. The starting vector for the Arnoldi iteration.
    Amul - A function handle. Should describe a linear operator.
    k    - Number of times to perform the Arnoldi iteration.
    tol  - Stop iterating if the next vector in the Arnoldi iteration has
           norm less than `tol'. Defaults to 1E-8.

    RETURN:
    Return the matrices H_n and Q_n defined by the Arnoldi iteration. The
    number n will equal k, unless the algorithm terminated early, in which
    case n will be less than k.

    Examples:
    >>> A = np.array([[1,0,0],[0,2,0],[0,0,3]])
    >>> Amul = lambda x: A.dot(x)
    >>> H, Q = arnoldi(np.array([1,1,1]), Amul, 3)
    >>> np.allclose(H, np.conjugate(Q.T).dot(A).dot(Q) )
    True

    >>> H, Q = arnoldi(np.array([1,0,0]), Amul, 3)
    >>> H
    array([[ 1.+0.j]])
    >>> np.conjugate(Q.T).dot(A).dot(Q)
    array([[ 1.+0.j]])
    '''
```

Hints:

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. Remember to use complex inner products.
3. This function can be tested on a matrix A by passing in `A.dot` for `Amul`.

Finding Eigenvalues Using 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 k^{th} stage of the algorithm. Then these matrices satisfy

$$H_k = Q_k^{\mathsf{H}} A Q_k. \tag{1.1}$$

If $k < n$, then H_k is a low-rank approximation to A. We may use its eigenvalues as approximations to the eigenvalues of A. The eigenvalues of H_k are called *Ritz*

values, and in fact they converge quickly to the largest eigenvalues of A.

Problem 2. Finish the following function that computes the Ritz values of a matrix.

```
def ritz(Amul, dim, k, iters):
    ''' Find `k' Ritz values of the linear operator defined by `Amul'.

    INPUTS:
    Amul    - A function handle. Should describe a linear operator on
              R^(dim).
    dim     - The dimension of the space on which `Amul' acts.
    k       - The number of Ritz values to return.
    iters   - The number of times to perform the Arnoldi iteration. Must
              be between `k' and `dim'.

    RETURN:
    Return `k' Ritz values of the operator defined by `Amul.'
    '''
```

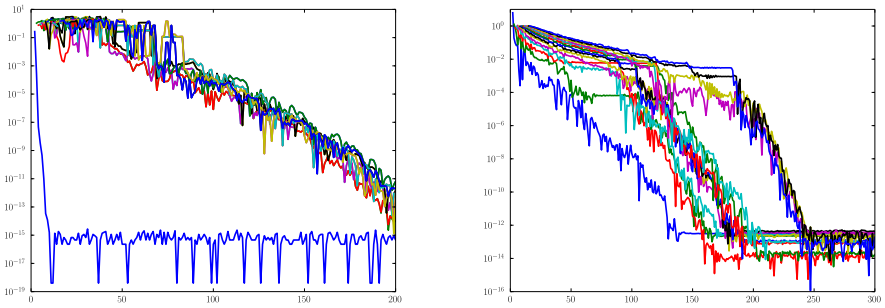
One application of the Arnoldi iteration is to find the eigenvalues of linear operators that are too large to store in memory. For example, if an operator acts on $\mathbb{C}^{2^{20}}$, then its matrix representation contains 2^{40} complex values. Storing such a matrix would require 64 terabytes of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

An example of such an operator is the Fast Fourier Transform, cited by SIAM as one of the top algorithms of the century [**Cipra2000**]. The Fast Fourier Transform is used ubiquitously in the field of 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 `ritz()` from Problem 2 to approximate the eigenvalues of the Fast Fourier Transform. Set `k` to be 10 and set `dim` to be 2^{20}. For the argument `Amul`, use the `fft` function from `scipy.fftpack`.

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.

```
>>> B = np.random.rand(10000).reshape(100, 100)
>>> sp.sparse.linalg.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 ])
```



(a) The blue line plots the error of the Ritz value of largest magnitude. This eigenvalue converges after fewer than 20 iterations

(b) All Ritz values have roughly equivalent magnitude. They take from 150 to 250 iterations to converge.

Figure 1.1: These plots show the relative error of the Ritz values as approximations to the eigenvalues of a matrix. The figure at left plots the largest 15 Ritz values for a 500×500 matrix with random entries. The figure at right plots the largest 15 Ritz values for a 500×500 matrix with uniformly distributed eigenvalues.

Convergence

The Arnoldi method for finding eigenvalues quickly converges 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 1.1a.

However, Ritz values converge more slowly for matrices with random eigenvalues. Figure 1.1b plots convergence of the Ritz values for a matrix with eigenvalues uniformly distributed in $[0, 1)$.

Problem 4. Finish the following function to visualize the convergence of the Ritz values.

```
def plot_ritz(A, n, iters):
    ''' Plot the relative error of the Ritz values of `A'.

    INPUTS:
    A     - A NumPy array.
    n - The number of Ritz values to plot.
    iters    - The number of times to perform the Arnoldi iteration.

    Create the following plot:
    - The x-axis is the number k of Arnoldi iterations on the x-axis
    - The y-axis is the relative error of the Ritz values of H_k as
    approximations to the eigenvalues of A.
    '''
```

If $\tilde{\mathbf{x}}$ is an an approximation to $\mathbf{x}$, then the *absolute error* in the approximation

is

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|}.$$

Hint: The most difficult part of this problem is to identify which Ritz values correspond to which eigenvalues. After finding the Ritz values (or eigenvalues) of largest magnitude, use `np.sort()` to put them in order. Make sure that this order is preserved throughout your program.

It may help to use the following algorithm.

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 $k \in$ `[1, iters)`,
   (a) Compute H_k with the Arnoldi iteration.
   (b) Find n eigenvalues of A of largest magnitude. Note that for small k, the matrix H_k may not have this many eigenvalues.
   (c) Store the absolute error. Make sure that the errors are stored in the correct order. For small k, some entries in the row or column may not be used.
3. Use array broadcasting to compute the absolute error.
4. Iteratively plot the errors. Lines for distinct eigenvalues should start at different places on the x-axis.

Run your function on these examples. The plots should be fairly similar to Figures 1.1b and 1.1a.

```
>>> # A matrix with random entries
>>> 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.dot( D.dot(la.inv(A)) )
>>> plot_ritz(B, 10, 175)
```

If your code takes too long to run, consider integrating your solutions to Problems 1 and 2 with the body of this function.

Lanczos Iteration(Optional)

The Lanczos iteration is a version of the Arnoldi iteration that is optimized to operate on symmetric matrices. If A is symmetric, then (1.1) 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 $\mathbf{q}_1, \ldots, \mathbf{q}_{n-2}$. 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.)

The Lanczos iteration is found in Algorithm 1.2.

Algorithm 1.2 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.

procedure LANCZOS($\mathbf{b}, A, k, tol$)
  $\mathbf{q}_0 \leftarrow \text{zeros}(\text{size}(\mathbf{b}))$ ▷ Some initialization
  $\mathbf{q}_1 \leftarrow \mathbf{b}/\left\|\mathbf{b}\right\|_2$
  $\mathbf{x} \leftarrow \text{empty}(k)$
  $\mathbf{y} \leftarrow \text{empty}(k)$
  for $i = 0 \ldots k-1$ **do** ▷ Perform the iteration.
    $\mathbf{z} \leftarrow A\mathbf{q}_1$ ▷ $\mathbf{z}$ is a temporary vector to store $\mathbf{q}_{i+1}$.
    $\mathbf{x}[i] \leftarrow \mathbf{q}_1^{\mathsf{T}}\mathbf{z}$ ▷ $\mathbf{q}_1$ is used to store the previous $\mathbf{q}_i$.
    $\mathbf{z} \leftarrow \mathbf{z} - \mathbf{x}[i]\mathbf{q}_1 + \mathbf{y}[i-1]\mathbf{q}_0$ ▷ $\mathbf{q}_0$ is used to store $\mathbf{q}_{i-1}$.
    $\mathbf{y}[i] = \left\|\mathbf{z}\right\|_2$ ▷ Initialize $\mathbf{y}[i]$.
    if $\mathbf{y}[i] < tol$ **then** ▷ Stop if $\left\|\mathbf{q}_{i+1}\right\|_2$ is too small.
      return $\mathbf{x}[:i+1]$, $\mathbf{y}[:i]$
    $\mathbf{z} = \mathbf{z}/\mathbf{y}[i]$
    $\mathbf{q}_0, \mathbf{q}_1 = \mathbf{q}_1, \mathbf{z}$ ▷ Store new $\mathbf{q}_{i+1}$ and $\mathbf{q}_i$ on top of $\mathbf{q}_1$ and $\mathbf{q}_0$.
  return $\mathbf{x}$, $\mathbf{y}[:-1]$

Problem 5. Implement Algorithm 1.2 by completing the following function. Write it so that it can operate on complex arrays.

```
def lanczos(b, Amul, k, tol=1E-8):
    '''Perform `k' steps of the Lanczos iteration on the symmetric linear
    operator defined by `Amul', starting with the vector 'b'.

    INPUTS:
    b    - A NumPy array. The starting vector for the Lanczos iteration.
    Amul - A function handle. Should describe a symmetric linear operator.
    k    - Number of times to perform the Lanczos iteration.
    tol  - Stop iterating if the next vector in the Lanczos iteration has
           norm less than `tol'. Defaults to 1E-8.

    RETURN:
    Return (alpha, beta) where alpha and beta are the main diagonal and
    first subdiagonal of the tridiagonal matrix computed by the Lanczos
    iteration.
```

```
    '''
```

As it is described in Algorithm 1.2, 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.

Problem 6. The following code performs multiplication by a tridiagonal symmetric matrix.

```
def tri_mul(a, b, u):
    ''' Return Au where A is the tridiagonal symmetric matrix with main
    diagonal a and subdiagonal b.
    '''
     v = a * u
     v[:-1] += b * u[1:]
     v[1:] += b * u[:-1]
     return v
```

Let A be a 1000×1000 symmetric tridiagonal matrix with random values in its nonzero diagonals. Use the function `lanczos()` from Problem 5 with 100 iterations to estimate the 5 eigenvalues of A of largest norm. Compare these to the 5 largest true eigenvalues of A

If you do this problem for different vectors a and b, you may notice that occasionally the largest Ritz value is repeated. This happens because the vectors used in the Lanczos iteration may not be orthogonal. These erroneous eigenvalues are called "ghost eigenvalues."

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 the function `scipy.sparse.linalg.eigsh()`.