Lab 1

Krylov Subspaces

Lab Objective: *Discuss simple Krylov Subspace Methods for finding eigenvalues and show some interesting applications.*

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.

More specifically, we can construct a Krylov subspace just by knowing how a linear transformation acts on vectors, and with these subspaces we can closely approximate eigenvalues of the transformation and solutions to associated linear systems.

The *Arnoldi iteration* is an algorithm for finding an orthonormal basis of a Krylov subspace. Its outputs can also be used to approximate the eigenvalues of the original matrix.

The Arnoldi Iteration

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

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

If the vectors $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \ldots, A^{n-1}\mathbf{x}\}$ 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.

One way to find an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$ would be to use the modified Gram-Schmidt algorithm from Lab TODO on the set $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \ldots, A^{n-1}\mathbf{x}\}$. More efficiently, the Arnold iteration integrates the creation of $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \ldots, A^{n-1}\mathbf{x}\}$ with the modified Gram-Schmidt algorithm, returning 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 operates on a vector b of length n and an $n \times n$ matrix A. It iterates k times or until the norm of the next vector in the iteration is less than tol.

```
1:  procedure ARNOLDI($b, A, k, tol = 1E-8$)
2:      $Q \leftarrow$ empty $(b.size, k+1)$                    ▷ Some initialization steps
3:      $H \leftarrow$ zeros $(k+1, k)$
4:      $Q_{[:,0]} = b$
5:      $Q_{[:,0]} /= \|Q_{[:,0]}\|_2$
6:      for $j = 0,\ j < k$ do                                  ▷ Perform the actual iteration.
7:          $Q_{[:,j+1]} = AQ_{[:,j]}$
8:          for $i = 0,\ i < j+1$ do                            ▷ Modified Gram-Schmidt.
9:              $H_{[i,j]} = \langle Q_{[:,i]}, Q_{[:,j+1]} \rangle$
10:             $Q_{[:,j+1]} -= H_{[i,j]} Q_{[:,i]}$
11:         $H_{[j+1,j]} = \|Q_{[:,j+1]}\|_2$                   ▷ Set subdiagonal element of $H$.
12:         if $|H_{[j+1,j]}| < tol$ then                       ▷ Stop if $\|Q_{[:,j+1]}\|_2$ is too small.
13:             return $H_{[:j+1,:j+1]}$, $Q_{[:,:j+1]}$
14:         $Q_{[:,j+1]} /= H_{[j+1,j]}$                        ▷ Normalize $q_{j+1}$.
15:     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 $\mathcal{K}_k(A, \mathbf{x}) = \text{span}\{\mathbf{x}, \lambda x, \lambda^2 \mathbf{x}, \ldots, \lambda^k \mathbf{x}\}$, which is equal to the span of $\mathbf{x}$. Let us trace through Algorithm **??** in this case. We will use q_i to denote the i^{th} column of Q.

In line 5 we normalize $\mathbf{x}$, setting $q_0 = \mathbf{x}/\|\mathbf{x}\|$. In line 7 we set $q_1 = Aq_0 = \lambda q_0$. Then in line 9

$$H_{0,0} = \langle q_0, q_1 \rangle = \langle q_0, \lambda q_0 \rangle = \lambda \langle q_0, q_0 \rangle = \lambda,$$

so in line 10 we subtract λq_0 from q_1, ending with $q_1 = 0$.

The vector q_1 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 q_0. In fact, q_0 already spans $\mathcal{K}_k(A, \mathbf{x})$. Hence, when in line 12 we find that the norm of q_1 is zero (or close to it, allowing for numerical error), we terminate the algorithm early, returning the 1×1 matrix $H = H_{0,0} = \lambda$ and the $n \times 1$ matrix $Q = q_0$.

A similar phenomenon may occur if the starting vector $\mathbf{x}$ is contained in a proper invariant subspace of A.

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.

```python
def arnoldi(b, Amul, k, tol=1E-8):
    '''Perform `k' steps of the Arnoldi iteration on the linear operator
    defined by `Amul', staring 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=complex128)`).
2. Remember to use complex inner products.
3. TODO: does it matter if you return H or a copy of H?

Finding Eigenvalues Using Arnoldi iteration

Let A be an $n \times n$ matrix. The matrices H_k and Q_k created by k iterations of the Arnoldi algorithm satisfy

$$H_k = Q_k^* A Q_k.$$

Since H_k is a $k \times k$ matrix, 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 the `k' Ritz values with largest real part 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.
numvals - The number of Ritz values to return.
iters   - The number of times to perform the Arnoldi iteration. Must
          be between `numvals' and `dim'.

RETURN:
Return the `k' Ritz values with largest real part 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, claimed by SIAM to be one of the top algorithms of the century [TODO: cite!]. The Fast Fourier Transform is used in many applications, including oil hunting and mp3 compression.

Problem 3. The eigenvalues of the Fast Fourier Transform are known to be $\{-1, 1, -i, i\}$. 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 ])
```

Convergence

The Arnoldi method for finding eigenvalues quickly converges to eigenvalues whose magnitude is distinctly larger than the rest.

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

```
def plot_ritz(Amul, dim, numvals, iters):
    ''' Plot the 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.
    numvals - The number of Ritz values to plot.
    iters   - The number of times to perform the Arnoldi iteration. Must
              be between `numvals' and `dim'.

    Creates a plot with the number of Arnoldi iterations k on the x-axis, ↩
        and the Ritz values of the Hessenberg approximation H_k on the y-↩
        axis.
    '''
```

Hints:

1. Before saving the Ritz values of H_k, use `np.sort()` to ensure that they are in a canonical order.
2. It is not efficient to call your function from Problem 2 for increasing values of `iters`. If your code takes too long to run, consider integrating your solutions to Problems 1 and 2 with the body of this function.

Run your function on these examples. See the plots below. TODO: output real and complex plots separately

Lanczos Iteration(Optional)

Depending on the symmetry of the problem we may be able to make the Arnoldi iteration more efficient. Consider the case that A is symmetric. This means that the matrx H is both Upper Hessenberg and symmetric, so it is tridiagonal. Since H is tridiagonal, we *should* have that each Aq_n is orthogonal to $q_0, \dots, q_{n-2}$. This means that storage of all the columns of Q_k is no longer necessary. We can run the entire algorithm while storing only the previous two columns of Q that we have computed. We can also represent H as two vectors: a vector α storing the values along the main diagonal of H, and a vector β storing the values in the first subdiagonal (the values in the first superdiagonal are the same). This change in the way things are stored allows Algorithm 1.1 to be simplified to Algorithm 1.2. Algorithm 1.2 is known as the Lanczos iteration.

Problem 5. Write a Python function that performs the Lanczos iteration. Have it accept a starting vector b, a function $Amul$ that computes Ax for

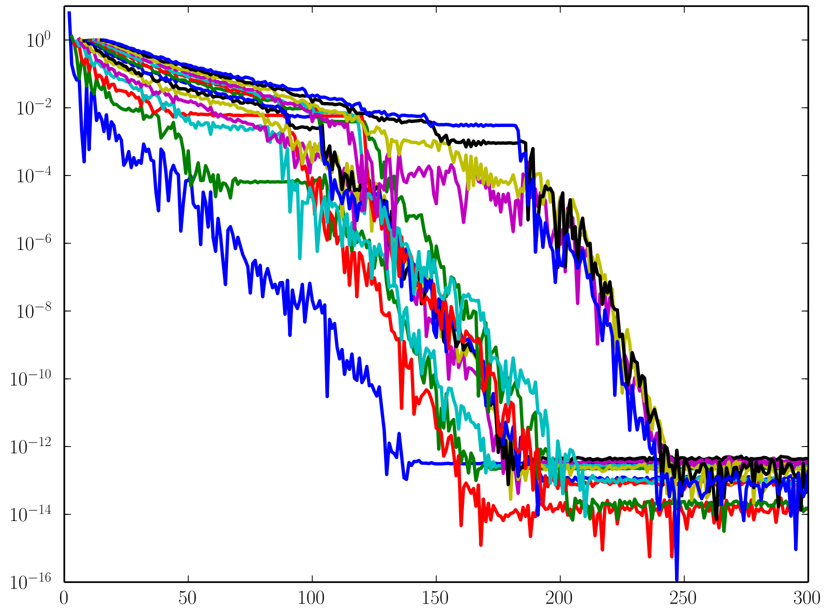

Figure 1.1: The convergence of the Ritz values to the largest eigenvalues of a matrix with random eigenvalues between 0 and 1.

any vector x, a number k of iterations to perform, and an optional argument *tol* that defaults to `1E-8`.

In its most basic form the Lanczos iteration is not stable. In exact arithmetic the vectors q_i are exactly orthogonal, but in the presence of roundoff error this may be absolutely false. In imprecise arithmetic, it is possible for the q_i to suffer from so much roundoff error that *they may no longer even be linearly independent*. There are a variety of modifications to the Lanczos iteration that address this instability. The library used for Lanczos iteration in Scipy uses an algorithm called the Implicitly Restarted Lanczos Method. We will not discuss these algorithms in detail here.

Problem 6. The following code performs matrix multiplication by a tridiagonal symmetric matrix. It accepts vectors a and b and u. a stores the entries in the main diagonal of the matrix. b stores the entries in the first sub/superdiagonal. The function returns the image of u under the matrix represented by a and b.

```
def tri_mul(a, b, u):
    v = a * u
    v[:-1] += b * u[1:]
```

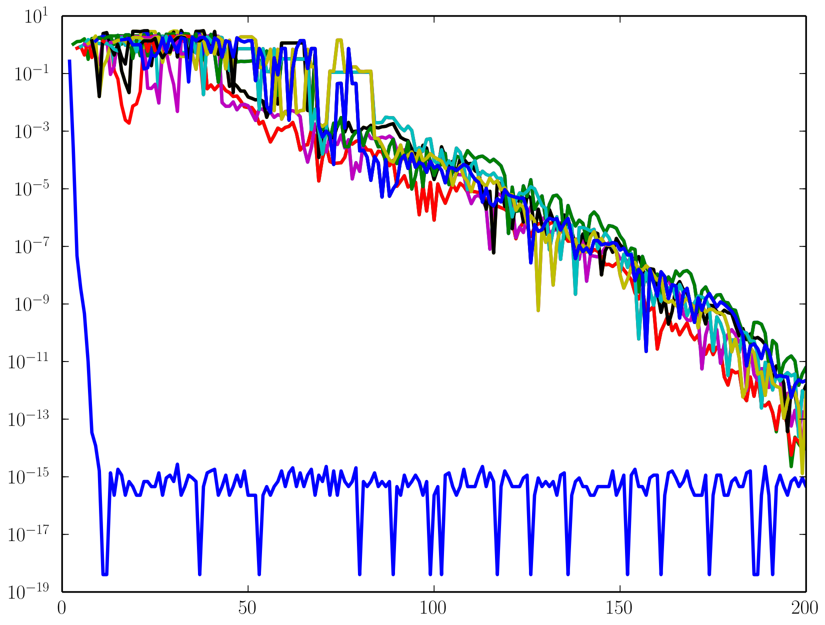

Figure 1.2: The convergence of the Ritz values to the largest eigenvalues of a matrix with random entries between 0 and 1. Matrices of this form generally have a single isolated eigenvalue that is much larger than the rest. It can be seen here that the Ritz values converge to this eigenvalue much more quickly.

Algorithm 1.2 The Lanczos Iteration

```
1: procedure LANCZOS($b, Amul, k, tol = 1E-8$)
2:     $q_0 \leftarrow 0$                                   ▷ Some initialization
3:     $q_1 \leftarrow \frac{b}{\|b\|_2}$
4:     $\alpha \leftarrow \text{empty}\,(k)$
5:     $\beta \leftarrow \text{empty}\,(k)$
6:     $\beta_{-1} = 0$
7:     for $i = 0$, $i < k$ do                           ▷ Perform the iteration.
8:         $z \leftarrow Amul\,(q_1)$                    ▷ $z$ is a temporary vector to store $q_{i+1}$.
9:         $\alpha_i = \langle q_1, z\rangle$            ▷ $q_1$ is used to store the previous $q_i$.
10:        $z - = \alpha_i q_1 + \beta_{i-1} q_0$        ▷ $q_0$ is used to store $q_{i-1}$.
11:        $\beta_i = \|z\|_2$                           ▷ Initialize $\beta_i$.
12:        if $\beta_i < tol$ then                       ▷ Stop if $\|q_{i+1}\|_2$ is too small.
13:            return $\alpha[: i+1]$, $\beta[: i]$
14:        $z / = \beta_i$
15:        $q_0, q_1 = q_1, z$                           ▷ Store new $q_{i+1}$ and $q_i$ on top of $q_1$ and $q_0$.
16:    return $\alpha$, $\beta[: -1]$
```

```
    v[1:] += b * u[:-1]
    return v
```

Use the Lanczos iteration function you wrote for Problem 5 to estimate the 5 largest eigenvalues of a symmetric tridiagonal matrix A with random values in its nonzero diagonals (i.e. make a and b random). For demonstration purposes, let A be 1000×1000. Perform 100 iterations. Compare the 5 eigenvalues of largest absolute value with the 5 Ritz values of largest norm. How do they compare?

Try running your simulation a few times for different vectors a and b. You may notice that, occasionally, the largest eigenvalue is repeated in the Ritz values. This happens because of the lack of orthogonality between the vectors used in the Lanczos iteration. These erroneous eigenvaleus are called "ghost eigenvalues." They generally converge to actual eigenvalues of the matrix and can make the multiplicity of an eigenvalue look higher than it really is.

Arnoldi Iteration in SciPy

SciPy interfaces with a Fortran library called ARPACK that has good implementations of the Arnoldi and Lanczos algorithms. The Arnoldi iteration is found in `scipy.sparse.linalg.eigs`. The Implicitly Restarted Lanczos iteration is found in `scipy.sparse.linalg.eigsh` These functions allow you to find either the largest or smallest eigenvalues of a sparse or dense matrix. The function `scipy.sparse.linalg.svds` uses the Implicitly Restarted Lanczos iteration on A^*A to find singular values.