Lab 1

GMRES

Lab Objective: *In this lab we will learn how to use the GMRES algorithm.*

The GMRES ("Generalized Minimal Residuals") algorithm is an effecient way to solve large linear systems. It is an iterative method that uses Krylov subspaces to reduce a high-dimensional problem to a sequence of smaller dimensional problems.

The GMRES Algorithm

Let A be an $m \times m$ matrix and let $\mathbf{b}$ be an m-vector. Let $\mathcal{K}_n(A, \mathbf{b})$ be the order-n Krylov subspace generated by A and $\mathbf{b}$. The idea of the GMRES algorithm is that instead of solving $A\mathbf{x} = \mathbf{b}$ directly, we solve the least squares problem

$$\underset{\mathbf{x}\in\mathcal{K}_n}{\text{minimize}} \qquad \|\mathbf{b} - A\mathbf{x}\|_2 \tag{1.1}$$

for increasing values of n. At each iteration, we compute the *residual*, or error between the least squares solution and a true solution of $A\mathbf{x} = \mathbf{b}$. The algorithm returns when this residual is sufficiently small. In good circumstances, this will happen when n is still much less than m.

The GMRES algorithm is integrated with the Arnoldi iteration for numerical stabiility, so that instead of solving (1.1), at the n^{th} iteration we solve

$$\underset{\mathbf{y}\in\mathbb{F}^n}{\text{minimize}} \qquad \|H_n\mathbf{y} - \|\mathbf{b}\|_2\mathbf{e}_1\|_2. \tag{1.2}$$

Here, H_n is the $(n+1) \times n$ upper Hessenberg matrix generated by the Arnoldi iteration and $\mathbf{e}_1$ is the vector $(1, 0, \ldots, 0)$ of length $n+1$. If $\mathbf{y}$ is the minimizer for the n^{th} iteration of (1.2), then the residual is

$$\frac{\|H_n\mathbf{y} - \|\mathbf{b}\|_2\mathbf{e}_1\|_2}{\|\mathbf{b}\|_2}, \tag{1.3}$$

and the corresponding minimizer for (1.1) is $Q_n\mathbf{y}$, where Q_n is the matrix whose columns are $q_1, \ldots, q_n$ as defined by the Arnoldi iteration. This algorithm is outlined in Algorithm 1.1. For a complete derivation see [TODO: ref textbook].

Algorithm 1.1 The GMRES algorithm. This algorithm operates on a vector b of length m and an $m \times m$ matrix A. It iterates k times or until the residual is less than tol.

procedure GMRES($A, b, k = 100, tol = 1E-8$)
  $Q \leftarrow$ empty $(b.size, k+1)$ ▷ Initialize
  $H \leftarrow$ zeros $(k+1, k)$
  $Q_{[:,0]} = b/\|b\|_2$
  for $n = 1, 2, \ldots, k$ **do**
    Set entries of Q and H as in Arnoldi iteration
    Calculate least squares solution y of 1.2.
    Calculate the residual r given by Equation 1.3.
    if $r < tol$ **then**
      return $Q_{[:,n+1]}y,\ r$
  return $Q_{[:,n+1]}y,\ r$

Problem 1. Use Algorithm 1.1 to complete the following Python function implementing the GMRES algorithm.

```
def gmres(b, Amul, k=100, tol=1E-8):
    '''Use the GMRES algorithm to approximate the solution to Ax=b, where A is the linear operator defined by `Amul'.

    INPUTS:
    b    - A NumPy array.
    Amul - A function handle. Should describe a linear operator.
    k    - Maximum number of iterations of the GMRES algorithm. Defaults to 100.
    tol  - Stop iterating if the residual is less than `tol'. Defaults to 1E-8.

    RETURN:
    Return (y, res) where `y' is an approximate solution to Ax=b and `res' is the residual.

    Examples:
    >>> A = np.array([[1,0,0],[0,2,0],[0,0,3]])
    >>> Amul = lambda x: A.dot(x)
    >>> b = np.array([1, 4, 6])
    >>> gmres(Amul, b)
    (array([ 1.,  2.,  2.]), 1.5083413465299765e-17)
    '''
```

You make make the following assumptions:

1. The input `b` is a real array and the function `Amul()` always outputs real arrays.
2. The vectors found by the Arnoldi iteration will never be zero. In other words, **b** will never be in a proper invariant subspace of A.

Hint: Use `scipy.linalg.lstsq()` to solve the least squares problem.

Convergence of GMRES

At the n-th iteration, GMRES computes the best approximate solution $\mathbf{x} \in \mathcal{K}_n$ to $A\mathbf{x} = \mathbf{b}$. If A is full rank, then $\mathcal{K}_m = \mathbb{F}^m$, so the m^{th} iteration will always return an exact answer. However, we say the algorithm converges after n steps if the n^{th} residual is sufficiently small.

The rate of convergence of GMRES depends on the eigenvalues of A.

Problem 2. Finish the following Python function by modifying your solution to Problem 1.

```
def plot_gmres(b, A, tol=1E-8):
    '''Use the GMRES algorithm to approximate the solution to Ax=b.

    INPUTS:
    b   - A 1-D NumPy array of length m.
    A   - A 2-D NumPy array of shape mxm.
    tol - Stop iterating and create the desired plots if the residual is
          less than `tol'. Defaults to 1E-8.

    OUTPUT:
    Follow the GMRES algorithm until the residual is less than tol, for a
    maximum of m iterations. Then create the two following plots (subplots
    of a single figure):

    1. Plot the eigenvalues of A in the complex plane

    2. Plot the convergence of the GMRES algorithm by plotting the
    iteration number on the x-axis and the residual on the y-axis.
    Use a log scale on the y-axis.
    '''
```

Use this function to investigate the convergence of GMRES as follows. Define an $m \times m$ matrix

$$A_n = nI + P,$$

where I is the $m \times m$ identity matrix and P is a $m \times m$ matrix of numbers from a random normal distribution with mean 0 and standard deviation $1/(2\sqrt{m})$. Call `plot_gmres` on A_n for $n = -4, -2, 0, 2, 4$. Use $m = 200$ and let `b` be an array of ones. What do you conjecture about the convergence of the GMRES algorithm?

Hints:

1. After creating a plot in matplotlib, change the y-axis to have a log scale with the command `plt.gca().set_yscale('log')`.

2. Create a matrix with entries from a random normal distribution with `np.random.rand()`.

3. Output for $n = 2$, $m = 200$ is in Figure 1.1 below.

Ideas for this problem were taken from Trefethen [TODO: cite better].

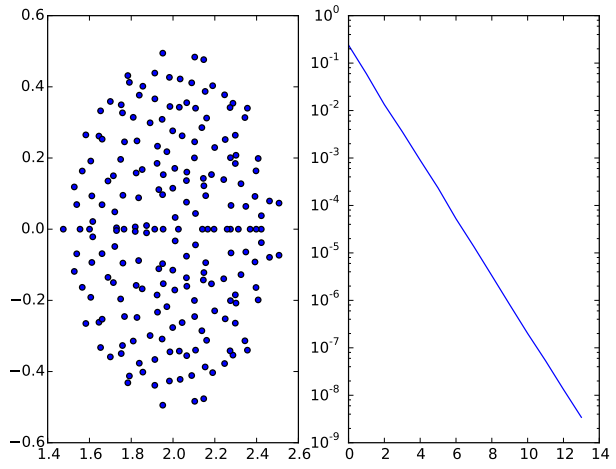

Figure 1.1: The left plot is the eigenvalues of the matrix A_2, which is defined in Problem 2. The right plot is the convergence of the GMRES algorithm on A_2 with starting vector $\mathbf{b} = (1, 1, \ldots, 1)$. This figure is one possible output of the function `plot_gmres()`.

Breakdowns in GMRES

One of the selling points of GMRES is that it can't break down unless it has reached an exact solution. Why is this the case? Breakdowns can occur as we try to expand to larger Krylov subspaces. In each iteration, we have already converted $b, Ab, A^2b, \cdots, A^{n-1}b$ into an orthonormal set $q_0, q_1, \cdots, q_{n-1}$. After computing A^nb, we will try to make it orthogonal to each of the q_i. But what if A^nb is already a linear combination of the q_i? In other words, what if A^nb lies in $\mathcal{K}_{n-1}$? This means that

$$\mathcal{K}_{n-1} = \mathcal{K}_n = \mathcal{K}_{n+1} = \cdots,$$

so we have reached an *invariant subspace*, and the algorithm cannot proceed without modification. It also means that $A^nb = c_0b + c_1Ab + \cdots + c_{n-1}A^{n-1}b$. Rearranging this, we have that $c_0b = c_1Ab + c_2A^2b + \cdots + A^nb$, so that b lies in $\mathcal{K}_{n+1}$. Therefore, b lies in $A\mathcal{K}_n$, so the least-squares problem will actually deliver an exact solution, up to rounding errors, in this case. To summarize: *GMRES only breaks down when it has found an exact solution.*

Problem 3. If necessary, update your solution to 1 to deal with breakdowns. Consider any division by zero cases in your code. Make the necessary adjustments to avoid any such cases.

Optimizing Least-Squares for GMRES (Optional)

The Hessenberg structure and the Krylov subspace relations enable us to save time on the least-squares part of the problem if we use QR factorization. Observe that if H_n can be factored as $Q_n R_n$, where Q_n is not the same matrix as above and R_n is invertible upper triangular, we may solve the least squares problem by simply solving $R_n x_n = \|b\| Q_n^H e_1$ via back substitution. There are two ways in which we can speed up this process. First, we take advantage of the Hessenberg structure by using the techniques from Problem **??** in Lab **??**. Recall that the technique in this situation was to use Givens rotations to eliminate the subdiagonal elements one at a time. This process, which was part of a previous lab, reduces the operation count from $O(n^3)$ to $O(n^2)$. The second speedup comes from the fact that we already know the QR factorization for H_{n-1} from the previous step of the algorithm. This means that we can simply update the QR factorization from the previous step rather than computing it all over again. Since H_n has only one more column and row than H_{n-1}, all we need to do is update the last column by performing all previous Givens rotations on just the last column of H_n, which requires only $O(n)$ work. Then we perform one final Givens rotation on H_n to eliminate the new subdiagonal entry which was not present in H_{n-1}. Thus, the QR factorization of H_n can be reduced from an $O(n^3)$ process to only $O(n)$ using these special techniques.

The back substitution necessary to solve the least squares problem can also be reduced to an operation of $O(n)$.

The speedup from $O(n^3)$ to $O(n)$ is very good, but it can only partially alleviate the problems that come with a problem that is ill-suited for GMRES. It may still be useful because it allows us to perform more iterations in a reasonable amount of time. In many situations, the simple technique of the next section will keep n low enough that the optimizations from this section are not critical.

GMRES(k)

One of the serious drawbacks of GMRES is that it requires a lot of storage. At step n, we need to store H_n and Q_n, so the storage is $O(n^2)$. It is also true that the complexity of the each iteration increases as n increases, a fact that can substantially slow down the process. If we want the iteration to proceed over many steps, these challenges can become prohibitive, so there is a modification called GMRES(k), or GMRES with restarts, that seeks to alleviate these difficulties by restarting the algorithm, but with an improved initial guess. It then builds the Krylov subspaces generated by this improved initial guess and repeats the process as many times as needed. Here's the outline of the algorithm:

1. *Set* k, the maximum number of iterations before memory requirements or complexity are too large
2. *Set* b, the initial guess
3. *while* $i < k$ and convergence is not yet reached and the number of times through this while loop is less than a set maximum

   Run GMRES for up to k iterations, starting with initial guess

   If convergence has not yet occurred, *set* initial guess to equal the most recent estimate of the solution

This process keeps storage under control and ensures that each iteration stays fast, but at the cost of reliability. Thus, when GMRES(k) does converge, it may be much quicker and require less storage than GMRES without restarts, but in situations where the true solution, x is nearly orthogonal to the first k Krylov subspaces, not much improvement is made at each iteration, so convergence is slow or may fail entirely. This is, however, an important variation of GMRES and is frequently used.

Problem 4. Adapt your prior implementation of GMRES to include restarts. It should also take the additional argument k, the number of iterations before restarting. Test its speed on the special 1000×1000 matrices we constructed to test GMRES and compare its speed to GMRES without restarts. No general statement can be made about the comparative speed of these two algorithms. How do they compare in the case of these matrices with $k = 10$ and A has dimension 1000×1000?

Industrial-grade GMRES package

In practice, of course, you will most frequently use GMRES algorithms written by specialists rather than writing your own version. In python, one option is the function `scipy.sparse.linalg.gmres`. It also includes an option to specify whether restarts are needed.

Problem 5. Test your GMRES algorithm against SciPy's GMRES algorithm on random 500×500 matrices, and report any difference on how quickly convergence is reached.

Summary

In this lab, we developed the mathematical basis for GMRES, practiced implementing and working with the algorithm, and explored several variations and optimizations which, when used in conjunction and in the right situations, make GMRES the

go-to algorithm for many, many applications. Avoid falling into the trap of thinking that GMRES is a universal solution to all systems of equations dilemmas, since we have already seen that even for simple random matrices, GMRES has horrendous performance. Learning and knowing when to apply a variety of algorithms is key to effective computing with large matrices.