

1 ADVANCED VARIATIONS OF THE STANDARD PROBLEMS

The standard problems in linear algebra are

\[
\begin{align*}
\min|Ax - b| & \quad \text{least squares} \\
Ax &= b & \quad \text{linear systems} \\
Ax &= \lambda x & \quad \text{eigenvalues} \\
Ax &= \sigma y & \quad \text{singular values}
\end{align*}
\]

In this note, we look at variations on these problems that we call *advanced* not because they are hard, but because they would occur in the context of a different type of application. We will also see one new problem, the matrix function!

For a linear system of least squares problem, an extremely common variation is that we have a set of right hand sides to solve. This occurs in two forms, one where all of the vectors are available at once, and a second where each solution gives rise to a new problem.

1.1 MULTIPLE RIGHT HAND SIDES ALL KNOWN AHEAD OF TIME

The problem here is that we need to solve \( Ax = b \) for many vectors \( b_1, \ldots, b_k \). In the simplest case, we will know \( b_1, \ldots, b_k \) ahead of time. In this case, we really have the matrix problem

\[
AX = B \quad B = [b_1 \quad \ldots \quad b_k]
\]

where \( X \) is the \( n \) by \( k \) matrix of all \( k \) solutions.

Such a scenario arises in a number of places. First, consider actually computing the inverse of a matrix \( A \). Then we would set \( B = I \), and there are \( k = n \) vectors \( b \) all known.

Another, more realistic scenario, arises in block Gaussian elimination. Suppose we are solving

\[
Ab = b \quad \text{where we have the partition} \quad \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.
\]

Then note that if \( A_1 \) is non-singular, then \( x_1, x_2 \) must satisfy \( A_1 x_1 + A_2 x_1 = b_1 \) or

\[
x_1 = A_1^{-1} b_1 + A_2^{-1} A_2 x_2.
\]

The matrix \( A_1^{-1} A_2 \) is exactly this type of system.

The simplest way to solve these is just to call \( \backslash \) in Julia or Matlab. This will look at the structure of \( A \) and choose an appropriate method to solve for all right hand sides simultaneously. It will use multiple threads and processors as appropriate.

In practice, what this will do is compute a factorization of the matrix \( A \) and then apply this to all the vectors \( b_1, \ldots, b_k \) at the same time.

In general, for a dense system of linear equations, it takes \( O(n^3) \) work to compute a factorization and then \( O(n^3) \) work to solve a system with the factors. This gives an overall runtime of \( O(n^3 k + n^3) \), which is \( O(n^3) \) if \( k \leq O(n) \) and *more interesting* if \( k \geq O(n) \).

As an example where the latter scenario arises, consider the partition above where \( A_1 \) is \( 16 \times 16 \) and \( n = 1024 \).
1.2 MULTIPLE RIGHT HAND SIDES DETERMINED SEQUENTIALLY

The second setting for multiple right hand sides is that we have

\[ Ax_1 = b_1 \]

which determines \( b_2 \), so \( b_2 \) is unknown until we have solved \( x_1 \). Then we must solve

\[ Ax_2 = b_2 \]

\[ Ax_3 = b_3 = \text{function of } x_2. \]

and so on...

The key is that the matrix \( A \) is fixed, which is a scenario that arises in

\[ \begin{align*}
& \cdot \text{ the inverse power method for eigenvalues} \\
& \cdot \text{backward Euler for linear ODEs.}
\end{align*} \]

**Inverse power method**

Recall the power method for dominant eigenvalue, eigenvector pair of a matrix

\[ x^{(0)} = \text{arbitrary} \quad x^{(k+1)} = \rho_k A x^{(k)} \quad \rho_k = \frac{1}{\| A x^{(k)} \|}. \]

If the largest magnitude eigenvalue of \( A \) is unique, then \( x^{(k)} \) will converge towards the associated eigenvector. The inverse power method simply runs this iteration on \( A^{-1} \) instead (assuming \( A \) is non-singular). For instance, if \( A \) is symmetric positive definite, then the inverse power method will converge to the smallest eigenvalue of \( A \). Here, we have exactly this type of setting where \( b^{(k+1)} = \rho_k x^{(k)} \).

\[ ^* \text{Again, note that the way to do this isn't to compute } A^{-1} \text{ and use that instead of } A! \]

That's another good way to fail the class.

**Backward Euler for a linear ODE**

XXX-TODO-XXX

**The factorization solution**

The best way to approach these problems is to factorize your linear system \( A \) via Cholesky, LU, or QR. These are be done on \( O(n^3) \) work and then each solve is \( O(n^2) \) time. This approach also allows us to exploit structure in the matrix \( A \) that may not exist in the inverse \( A^{-1} \) to make things go faster.

**The Julia code**

Julia includes a number of awesome routines to work with matrix factorizations *like* the original matrix. For instance,

```julia
1  F = lufact(A) # produces a factorization object F
2  F \ y # solves Ax = y using the LU factorizations without recomputing it.
```

So we could implement the inverse power method as follows

```julia
1  function invpower(A::Matrix)
2    x = normalize!(randn(size(A,1)))
3    F = lufact(A)
4    for iter = 1:maxiter
5      x = normalize!(F\x)
6    end
7    return x
8  end
```