ADVANCED VARIATIONS OF THE STANDARD PROBLEMS

David F. Gleich

August 21, 2023

1 ADVANCED VARIATIONS OF THE STANDARD PROBLEMS

The standard problems in linear algebra are

$$\min \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\| \qquad \text{least squares}$$

$$\boldsymbol{A}\mathbf{x} = \mathbf{b} \qquad \text{linear systems}$$

$$\boldsymbol{A}\mathbf{x} = \lambda \mathbf{x} \qquad \text{eigenvalues}$$

$$\boldsymbol{A}\mathbf{x} = \sigma \mathbf{y} \qquad \text{singular values}$$

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 $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ for many vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k$. In the simplest case, we will know $\mathbf{b}_1, \ldots, \mathbf{b}_k$ ahead of time. In this case, we really have the matrix problem

$$\boldsymbol{A}\boldsymbol{X} = \boldsymbol{B} \qquad \boldsymbol{B} = \begin{bmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_k \end{bmatrix}$$

where $\boldsymbol{X}$ 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 $\boldsymbol{A}$. [1] Then we would set $\boldsymbol{B} = \boldsymbol{I}$, and there are $k = n$ vectors $\mathbf{b}$ all known.

[1] Aside, you shouldn't generally do this! It's a good way to fail the class if you do this without careful consideration of the alternatives.

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

$$\boldsymbol{A}\mathbf{b} = \mathbf{b} \text{ where we have the partition } \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_2 \\ \boldsymbol{A}_3 & \boldsymbol{A}_4 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}.$$

Then note that if $\boldsymbol{A}_1$ is non-singular, then $\mathbf{x}_1$ must satisfy $\boldsymbol{A}_1\mathbf{x}_1 + \boldsymbol{A}_2\mathbf{x}_1 = \mathbf{b}_1$ or

$$\mathbf{x}_1 = \boldsymbol{A}_1^{-1}\mathbf{b}_1 + \boldsymbol{A}_1^{-1}\boldsymbol{A}_2\mathbf{x}_2.$$

The matrix $\boldsymbol{A}_1^{-1}\boldsymbol{A}_2$ is exactly this type of system.

The simplest way to solve these is just to call \ in Julia or Matlab. This will look at the structure of $\boldsymbol{A}_1$ 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 $\boldsymbol{A}$ and then apply this to all the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k$ 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 $O(n^2)$ work to solve a system with the factors. This gives an overall runtime of $O(n^2k + n^3)$, which is $O(n^3)$ if $k \leq O(n)$ and *more interseting* if $k \geq O(n)$.

As an example where the latter scenario arises, consider the partiton above where $\boldsymbol{A}_1$ is 16×16 and n is 1024.

1.2 MULTIPLE RIGHT HAND SIDES DETERMINED SEQUENTIALLY

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

$$\boldsymbol{A}\mathbf{x}_1 = \mathbf{b}_1$$

which determines $\mathbf{b}_2$, so $\mathbf{b}_2$ is unknown until we have solved $\mathbf{x}_1$. Then we must solve

$$\boldsymbol{A}\mathbf{x}_2 = \mathbf{b}_2$$

$$\boldsymbol{A}\mathbf{x}_3 = \mathbf{b}_3 = \text{function of } \mathbf{x}_2.$$

$$\text{and so on...}$$

The key is that the matrix $\boldsymbol{A}$ is fixed, which is a scenario that arises in

- the inverse power method for eigenvalues
- backward Euler for linear ODEs.

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

$$\mathbf{x}^{(0)} = \text{ arbitrary} \qquad \mathbf{x}^{(k+1)} = \rho_k \boldsymbol{A}\mathbf{x}^{(k)} \quad \rho_k = \frac{1}{\|\boldsymbol{A}\mathbf{x}^{(k)}\|}.$$

If the largest magnitude eigenvalue of $\boldsymbol{A}$ is unique, then $\mathbf{x}^{(k)}$ will converge towards the associated eigenvector. The inverse power method simply runs this iteration on $\boldsymbol{A}^{-1}$ instead (assuming $\boldsymbol{A}$ is non-singular). For instance, if $\boldsymbol{A}$ is symmetric positive definite, then the inverse power method will converge to the smallest eigenvalue of $\boldsymbol{A}$. Here, we have exactly this type of setting where $\mathbf{b}^{(k+1)} = \rho_k \mathbf{x}^{(k)}$.[2]

[2] Again, note that the way to do this isn't to compute $\boldsymbol{A}^{-1}$ and use that instead of $\boldsymbol{A}$! 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 $\boldsymbol{A}$ 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 $\boldsymbol{A}$ that may not exist in the inverse $\boldsymbol{A}^{-1}$ 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,

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

So we could implement the inverse power method as follows

```
function invpower(A::Matrix)
  x = normalize!(randn(size(A,1)))
  F = lufact(A)
  for iter = 1:maxiter
    x = normalize!(F\x)
  end
  return x
end
```