Two Classic Monographs


Linear Systems of Equations

- We start with a square linear system. Suppose \(A \in \mathbb{R}^{n \times n}\) is a given matrix, \(b \in \mathbb{R}^n\) a given vector, and \(x \in \mathbb{R}^n\) a vector that is determined by the linear system

\[
Ax = b. \quad (1)
\]

How to solve \(Ax = b\)

- Of course we know (from College Algebra) how to go about solving this system by Gaussian Elimination and Backward Substitution. Our first task is to look at this process more incisively.

- We start simply, and refine the procedure in steps.

\(^{-1}\) Horn calls it matrix analysis rather than linear algebra because in his view linear algebra is about statements that are independent of the choice of basis. Most of us would refer to the topic of this book less accurately as linear algebra.
In Gaussian Elimination we proceed through the columns, starting in the first. In the \( k \)-th column we subtract multiples of the \( k \)-th row from the rows underneath the \( k \)-th row where the coefficients (called the multipliers in this context) are chose so as to zero the entries in the \( k \)-th column below the \( k \)-th row:

\[
\begin{bmatrix}
  a_{11} & a_{1n} & a_{1,n+1} \\
  a_{21} & a_{2n} & a_{2,n+1} \\
  \vdots & \vdots & \vdots \\
  a_{n1} & a_{nn} & a_{n,n+1}
\end{bmatrix}
\]

\[
b_i = a_{i,n+1}
\]

\[
X_n = \frac{a_{n,n+1}}{a_{nn}}
\]

\[
a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = a_{n-1,n+1}
\]
• This leads to the **Gaussian Elimination** algorithm:

For \( k = 1, \ldots, n-1 \)

For \( i = k+1, \ldots, n \)

\[
\begin{align*}
m_{ik} &= \frac{a_{ik}}{a_{kk}} \quad \text{(multiplier)} \\
a_{ij} &= a_{ij} - m_{ik}a_{kj}
\end{align*}
\]

• To find \( x \) we use **Backward Substitution**:

For \( i = n, n - 1, n - 2, \ldots 1 \)

\[
x_i = \frac{\sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}}
\]
• What can go wrong?
• We must not divide by zero.

Moreover, we must not divide by numbers close to zero!

**Pivoting**

• Use row (and perhaps column) interchanges.
• The pivot is the entry in the \((k, k)\) position.

**Partial Pivoting**

• Let \(\mu\) be defined by

\[
|a_{\mu k}| = \max_{k \leq i \leq n} |a_{ik}|
\]

Pick the first (or the last) such \(\mu\) if there are several.

• Then interchange the \(\mu\)-th and \(k\)-th rows before proceeding further.

**Total Pivoting**

• Let \(\mu\) and \(\nu\) be defined by

\[
|a_{\mu \nu}| = \max_{k \leq i, j \leq n} |a_{ij}|
\]

Pick the first (or the last, or any) such \((\mu, \nu)\) if there are several.

• Then interchange the \(\mu\)-th and \(k\)-th rows, and the \(\nu\)-th and \(k\)-th columns before proceeding further.

• Interchanging columns corresponds to reordering the variables.

• As a practical matter we don’t interchange rows and columns physically, we keep track of their location using index vectors.

• **Scaling** is possible.

• Matlab uses unscaled partial pivoting.
Several Right Hand Sides

- Example: computing the inverse, the right hand sides are the standard unit vectors.
- We could include all right hand sides in the working array.
- But we might know a new right hand side only after solving the system for the previous right hand side.
- Example: a version of Newton’s Method where we keep the Jacobian constant.
- We could handle this by first processing the coefficient matrix, and then processing any right hand sides.
- The latter approach is by far the more common.
- We need to keep track of the multipliers.

\[
\text{For } k = 1, \ldots, n-1 \\
\text{Pivot} \\
\text{For } i = k+1, \ldots, n \\
\alpha_{ik} = \frac{a_{ik}}{a_{kk}} \\
\text{For } j = k+1, \ldots, n \\
\alpha_{ij} = \alpha_{ij} - \alpha_{ik} \alpha_{kj} \\
\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} 1 + n-k 
\]

Math 5600 Spring 2021 Notes of 3/16/21 page 5
Matrix Factorization

- Nowadays most analysis in linear algebra is conducted in terms of matrices rather than in terms of nested loops as above.

- Ignoring the issue of pivoting for the moment we write $A$ as

$$A = LU$$  \hspace{1cm} (2)

where

$L$ is **unit lower triangular**, i.e.,

$$l_{ii} = 1, \quad i = 1, 2, \ldots n \quad \text{and} \quad j > i \quad \Rightarrow \quad a_{ij} = 0$$

$U$ is **upper triangular**, i.e.,

$$l_{ii} = 1, \quad i = 1, 2, \ldots n \quad \text{and} \quad j > i \quad \Rightarrow \quad a_{ij} = 0$$

- Given the matrix factorization (2) we solve the system $Ax = LUx = b$ in two steps:

  1. solve: $Lz = b$ \hspace{1cm} Forward Substitution
  2. solve: $Ux = z$ \hspace{1cm} Backward Substitution

Gaussian Elimination computes the $LU$ factorization

- the two procedures are equivalent!

- The standard argument to show this is a mess that involves “elementary matrices” and their inverses, and generous use of groups of three dots . . .

- However, Gil Strang of MIT came up with a beautifully simple argument.
• Consider the evolution of the working array during Gaussian Elimination, for a $4 \times 4$ system.

• The letter $x$ denotes an entry in the working array. The symbol $\bigotimes$ denotes an entry that is final and no longer changed in the process. The letter $m$ denotes the multipliers, stored in the lower part of the working array.

• We get

\[
\begin{bmatrix}
\bigotimes & \bigotimes & \bigotimes & \bigotimes \\
x & x & x & x \\
x & x & x & x \\
x & x & x & x \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\bigotimes & \bigotimes & \bigotimes & \bigotimes \\
m_{21} & \bigotimes & \bigotimes & \bigotimes \\
m_{31} & x & x & x \\
m_{41} & x & x & x \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\bigotimes & \bigotimes & \bigotimes & \bigotimes \\
m_{21} & \bigotimes & \bigotimes & \bigotimes \\
m_{31} & m_{32} & \bigotimes & \bigotimes \\
m_{41} & m_{42} & x & x \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\bigotimes & \bigotimes & \bigotimes & \bigotimes \\
m_{21} & \bigotimes & \bigotimes & \bigotimes \\
m_{31} & m_{32} & \bigotimes & \bigotimes \\
m_{41} & m_{42} & m_{43} & \bigotimes \\
\end{bmatrix}
\]

• Let’s denote row $i$ of a matrix $A$ by $r_i(A)$ and consider in particular the third row of the working array. We get

\[
r_3(U) = r_3(A) - m_{31}r_1(U) - m_{32}r_2(U)
\]

• This can be rewritten as

\[
r_3(A) = m_{31}r_1(U) + m_{32}r_2(U) + r_3(U)
\]

That last equation is exactly what we get in the matrix multiplication $A = LU$!
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
m_{21} & 1 & 0 & 0 \\
m_{31} & m_{32} & 1 & 0 \\
m_{41} & m_{42} & m_{43} & 1 \\
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix} r_1(U) \\ r_2(U) \\ r_3(U) \\ r_4(U) \end{bmatrix} \\
\end{bmatrix} = U
\]
\[
\begin{bmatrix}
\begin{bmatrix} r_1(U) \\ r_2(U) \\ r_3(U) \\ r_4(U) \end{bmatrix} \\
\end{bmatrix} = A
\]
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
m_{21} & 1 & 0 & 0 \\
m_{31} & m_{32} & 1 & 0 \\
m_{41} & m_{42} & m_{43} & 1 \\
\end{bmatrix}
= L
\]

- Clearly, this applies in general, not just to the third row of a $4 \times 4$ matrix!
- $A = LU$, $L$ is unit lower triangular, $U$ is upper triangular.
The Inverse Matrix

• If we knew the inverse matrix we could solve the linear system by multiplying the right hand side with the inverse:

\[ Ax = b \iff x = A^{-1}b \]

• The inverse is defined by

\[ AA^{-1} = I \]

• Thus the \( j \)-th column of the inverse is the solution of

\[
Ax = e_j = \begin{bmatrix}
0 \\
\vdots \\
0 \\
1 \\
0 \\
\vdots \\
0 
\end{bmatrix}
\]

• So computing the inverse amounts to solving \( n \) linear systems all of which have the coefficient matrix \( A \).
Measuring Numerical Effort

• “numerical effort” could mean the time a computer needs to execute an algorithm.

• That time depends on the computer (and also the particular implementation and the computer language being used).

• We want to compare algorithms, independently of computer specifics.

• The standard way to compare linear algebra algorithms is to count the number of flops which usually means the number of multiplications and divisions.

• The reason not to count additions and subtractions, or other operations like assignments and data lookups, is that in typical implementations these operations require an amount of time that is roughly proportional to the number of flops.

• The following Table shows the number of flops required for some common operations:

\[
\begin{align*}
A &= LU & \frac{n^3}{3} + O(n^2) \\
Ly &= b & \frac{n^2}{2} - \frac{n}{2} \\
Ux &= y & \frac{n^2}{2} + \frac{n}{2} \\
x &= A^{-1}b & n^2 \\
A^{-1} &= \quad n^3 + O(n^2)
\end{align*}
\]

note that forward and backward substitution together take exactly the same effort as multiplying with the inverse!

• However, computing the inverse is three times as expensive as computing the $LU$ factorization.
• Moreover, as the first step in computing $A^{-1}$ you compute $LU$ anyway. Thus computing $A^{-1}$ after computing the $LU$ factorization just introduces additional round-off errors.

However, the biggest problem with matrix inversion is that it destroys sparsity. For example (see problem 10 of hw 2) the inverse of a tridiagonal matrix in general is full, whereas $L$ and $U$ in the LU factorization are still tridiagonal.

• In general, an $n \times n$ tridiagonal system can be solved in $5n$ flops.

Simplifying things somewhat we arrive at the conclusion

Never Invert a Matrix

• Can you get below $O(n^3)$? Yes, but it’s tricky, and beyond our scope.
• bet with Nick Trefethen:

25 June 1985

N. Trefethen hereby bets Peter Alfeld

that by 31 December, 1994, a method
will have been found to solve $Ax = b$
(non linear system of eqs.) in $O(n^{2+E})$
operations for any $E>0$. Numerical stability
is not required.

The winner gets $100 from

the loser

Peter Alfeld
Nick M. Trefethen

Witnesses:

For Erik Koch
S.P. Nørsett (This is a crude problem)

Figure 1. A Bet.