MATH3511/6111: Scientific Computing

10. Gaussian Elimination

Lindon Roberts

Semester 1, 2022

Office: Hanna Neumann Building #145, Room 4.87
Email: lindon.roberts@anu.edu.au
Based on lecture notes written by S. Roberts, L. Stals, Q. Jin, M. Hegland, K. Duru.

Linear Systems

We will now look at algorithms for solving linear systems: given $A \in \mathbb{R}^{n\times n}$ invertible and $\boldsymbol{b} \in \mathbb{R}^n$, solve

$$\boxed{A\boldsymbol{x} = \boldsymbol{b}}$$

This is possibly the most important task in scientific computing, used for:

- Solving nonlinear equations (multidimensional Newton's method): $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{s}_k$ where $\boldsymbol{F}'(\boldsymbol{x}_k)\boldsymbol{s}_k = F(\boldsymbol{x}_k)$.
- Interpolation (Vandermonde matrix) and spline construction
- Solving differential equations
- Least-squares data fitting
- ...

Linear Systems

From first-year, you know two ways to solve $A\boldsymbol{x} = \boldsymbol{b}$:

- Gaussian elimination (row reduction)
- Matrix inversion (calculate A^{-1}, then $\boldsymbol{x} = A^{-1}\boldsymbol{b}$)

You can probably do both methods by hand, if n is not too big (e.g. $n \leq 5$).

In most scientific computing settings, n is much larger (frequently hundreds or thousands, potentially even millions).

Linear Systems

There are two basic categories of algorithm for solving linear systems:

- Direct methods: algorithms which produce an exact solution (assuming no rounding errors) in a finite number of steps.
- Iterative methods: algorithms which generate a sequence of vectors converging to the solution, $\lim_{k\to\infty} \boldsymbol{x}_k = \boldsymbol{x}$.

Both categories are useful in practice, but usually in different settings.

We will first talk about direct methods. They are a common 'default' choice for solving linear systems: e.g. when n is not too large (10s or maybe 100s) and A is dense (most entries are nonzero).

The most common direct method is Gaussian elimination* (aka LU factorisations). We will pay particular attention to what happens in the presence of rounding errors.

* *Named after Carl Friedrich Gauss (early 1800s), but known since at least c. 200BC in China.*

Elementary Matrices

Elementary matrices are used to perform row reduction. They have the form:

$$E_k\boldsymbol{a} = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

We choose $m_j = a_j/a_k$ to introuce zeros in the $k+1, \ldots, n$ entries of $\boldsymbol{a}$.

In Gaussian elimination, we use elementary matrices to introduce zeros below a pivot entry.

Elementary Matrices

Elementary matrices have several useful properties which make them very convenient to work with:

- E_k is lower triangular with unit main diagonal.
- We can write $E_k = I - \boldsymbol{m}_k \boldsymbol{e}_k^T$, where $\boldsymbol{m}_k = \begin{bmatrix} 0 & \cdots & 0 & m_{k+1} & \cdots & m_k \end{bmatrix}^T$ and $\boldsymbol{e}_k$ is the k-th coordinate vector.
- E_k is invertible, with $E_k^{-1} = I + \boldsymbol{m}_k \boldsymbol{e}_k^T$ (also unit lower triangular). We will write $L_k = E_k^{-1}$.
- If $k < j$, then $E_k E_j = I - \boldsymbol{m}_k \boldsymbol{e}_k^T - \boldsymbol{m}_j \boldsymbol{e}_j^T$.
  - More generally, $E_1 E_1 \cdots E_{n-1} = I - \sum_{k=1}^{n-1} \boldsymbol{m}_k \boldsymbol{e}_k^T$ (still unit lower triangular).
  - Can be proved by using $\boldsymbol{e}_k^T \boldsymbol{m}_j = 0$ for $k < j$.

Gaussian Elimination

In Gaussian elimination, row operations can be achieved by multiplying by an elementary matrix.

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & -1 & -1 \\ -3 & 3 & 2 \end{bmatrix} \quad \begin{array}{l} (\text{R2} - 2\ \text{R1} \to \text{R2}) \\ (\text{R3} + 3\ \text{R1} \to \text{R3}) \\ \longrightarrow \end{array} \quad \begin{bmatrix} 1 & 2 & 4 \\ 0 & -5 & -9 \\ 0 & 9 & 14 \end{bmatrix}$$

We pick the elementary matrix E_1 which will add zeros the first column (except for the first element):

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix}}_{E_1} \underbrace{\begin{bmatrix} 1 & 2 & 4 \\ 2 & -1 & -1 \\ -3 & 3 & 2 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & 2 & 4 \\ 0 & -5 & -9 \\ 0 & 9 & 14 \end{bmatrix}}_{=E_1 A}$$

Gaussian Elimination

The basic Gaussian elimination algorithm is:

1. Apply an elementary matrix E_1 to introduce zeros in the first column (below $a_{1,1}$).
2. Apply an elementary matrix E_2 to introduce zeros in the second column (below $a_{2,2}$).
3. Continue until $(n-1)$-th column; nothing needed for n-th column as no entries below $a_{n,n}$.

If we call the resulting matrix U, we have found E_k such that $E_{n-1}E_{n-2}\cdots E_2E_1A = U$. That is,

$$A = E_1^{-1}E_2^{-1}\cdots E_{n-2}^{-1}E_{n-1}^{-1}U = L_1L_2\cdots L_{n-2}L_{n-1}U,$$

If we define $L = L_1L_2\cdots L_{n-2}L_{n-1}$, we have

$$\boxed{A = LU}$$

This is called an LU factorisation of A.

Gaussian Elimination

Question

What has this really achieved? Aren't we just rewriting A?

No! If we calculate L and U this way, there are two important results:

- U is upper triangular (since we introduced zeros below all diagonal entries)
- L is unit lower triangular — this is surprising, and is because of the nice properties of elementary matrices!

(hence using the letters L and U for lower and upper triangular matrices)

We will see that the effort required to calculate L and U will help us to solve $A\boldsymbol{x} = \boldsymbol{b}$.

LU Factorisation: Example

Example

Calculate the LU factorisation of the matrix associated with the linear system

$$\begin{aligned} 2x_1 + 4x_2 - 2x_3 &= 2 \\ 4x_1 + 9x_2 - 3x_3 &= 8 \\ -2x_1 - 3x_2 + 7x_3 &= 10 \end{aligned}$$

In matrix notation:

$$\underbrace{\begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix}}_{\boldsymbol{b}}.$$

LU Factorisation: Example

Row operations: add zeros to first column (below $a_{1,1}$):

$$E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix}.$$

$$E_1 \boldsymbol{b} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 12 \end{bmatrix}.$$

Next, add zeros to second column of $E_1 A$ (below $(2,2)$ entry):

$$E_2 E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix} = \underbrace{\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix}}_{U}, \quad E_2 E_1 \boldsymbol{b} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 12 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}.$$

LU Factorisation: Example

We have reached the upper triangular system

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}.$$

The corresponding L is

$$L = L_1 L_2 = E_1^{-1} E_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix}.$$

Check our factorisation:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix}}_{A}.$$

LU Factorisation

Once we have computed the factorisation $A = LU$, it is easy to solve $A\boldsymbol{x} = \boldsymbol{b}$. Our procedure is:

- Calculate the factorisation $A = LU$.
- Solve $L\boldsymbol{y} = \boldsymbol{b}$ to get $\boldsymbol{y}$
  - This can be done at the same time as calculating $A = LU$, since $\boldsymbol{y} = E_{n-1}E_{n-2}\cdots E_2E_1\boldsymbol{b}$.
  - Or, if $\boldsymbol{b}$ is not known until later, just solve $L\boldsymbol{y} = \boldsymbol{b}$ directly.
- Solve $U\boldsymbol{x} = \boldsymbol{y}$ to get $\boldsymbol{x}$.

So, Gaussian elimination transformation the problem of solving one linear system into the problem of solving two triangular linear systems. Fortunately, solving triangular linear systems is easy.

Forward Substitution

We can solve a lower triangular linear system $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b}$ using forward substitution.

$$\begin{array}{ccccccc}
\ell_{1,1}y_1 & & & & & = & b_1 \\
\ell_{2,1}y_1 & + & \ell_{2,2}y_2 & & & = & b_2 \\
\vdots & & \vdots & \ddots & & = & \vdots \\
\ell_{n,1}y_1 & + & \ell_{n,2}y_2 & + \cdots + & \ell_{n,n}y_n & = & b_n
\end{array}$$

We first find $y_1 = b_1/\ell_{1,1}$.

Substituting into the second equation, we get $y_2 = (b_2 - \ell_{2,1}y_1)/\ell_{2,2}$, etc.

In general, we have

$$y_i = \frac{b_i - \sum_{k=1}^{i-1} \ell_{i,k}y_k}{\ell_{i,i}}, \qquad i = 1, 2, \ldots, n.$$

Note that for LU factorisations, $\ell_{i,i} = 1$ always.

Backward Substitution

We can solve an upper triangular linear system $U\boldsymbol{x} = \boldsymbol{y}$ using backward substitution.

$$\begin{array}{ccccccc} u_{1,1}x_1 & + & u_{1,2}x_2 & +\cdots+ & u_{1,n}x_n & = & y_1 \\ & & u_{2,2}x_2 & +\cdots+ & u_{2,n}x_n & = & y_2 \\ & & & \ddots & \vdots & \vdots & \vdots \\ & & & & u_{n,n}x_n & = & y_n \end{array}$$

Working backwards, we first get $x_n = y_n/u_{n,n}$.

Substituting into the second-last equation, we get $x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}$, etc.

In general,

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} u_{i,k}x_k}{u_{i,i}}, \qquad i = n, n-1, \ldots, 1.$$

For LU factorisations, why do we not worry about $u_{i,i} = 0$?

Backward Substitution: Example

Example

(continued from before) Solve the upper-triangular system

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}.$$

Should get $\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T = \begin{bmatrix} -1 & 2 & 2 \end{bmatrix}^T$.

Gaussian Elimination: Implementation

The below code computes the LU factorisation of a matrix A. Note it only computes L and U, it doesn't solve $A\boldsymbol{x} = \boldsymbol{b}$.

```python
import numpy as np

def lu_factorisation(A):
    n = A.shape[0] # dimension of A
    L = np.eye(n)  # L starts as the identity matrix
    U = A.copy()   # U starts as A (we will put in zeros as we go)
    for k in range(n-1):          # column of U where we are adding zeros
        for i in range(k+1, n):   # row of U to add zero
            L[i, k] = U[i,k] / U[k,k]  # entry of L to zero out U[i,k]
            for j in range(k, n):    # apply row operation to nonzero part of row i
                U[i, j] = U[i, j] - L[i, k]*U[k, j]
    return L, U
```

The last loop (over j) can be rewritten using NumPy slicing.

Gaussian Elimination: Flops

How long will it take to compute an LU factorisation?

We can answer this by counting the number of flops (floating-point operations: $+$, $-$, $\times$, $\div$).

```
for k in range(n-1):                                    # k = 0, ..., n-2
    for i in range(k+1, n):                             # i = k+1, ..., n-1
        L[i, k] = U[i,k] / U[k,k]                       # 1 flop
        for j in range(k, n):                           # j = k, ..., n-1
            U[i, j] = U[i, j] - L[i, k]*U[k, j]         # 2 flops
```

$$\text{flops} = \sum_{k=0}^{n-2}\sum_{i=k+1}^{n-1}\left[1+\sum_{j=k}^{n-1}2\right] = \sum_{k=0}^{n-2}(n-1-k)\left[1+2(n-k)\right].$$

We can simplify this expression by changing variables, $\ell = n-1-k$:

$$\text{flops} = \sum_{\ell=1}^{n-1}\ell[1+2(\ell+1)] = \sum_{\ell=1}^{n-1}(2\ell^2+3\ell) = 2\left(\sum_{\ell=1}^{n-1}\ell^2\right)+3\left(\sum_{\ell=1}^{n-1}\ell\right).$$

Gaussian Elimination: Flops

$$\boxed{\text{flops} = 2\left(\sum_{\ell=1}^{n-1} \ell^2\right) + 3\left(\sum_{\ell=1}^{n-1} \ell\right)}$$

The second term is a formula you probably know: $\sum_{\ell=1}^{n-1} \ell = (n-1)n/2 = \mathcal{O}(n^2)$.

How big is $\sum_{\ell=1}^{n-1} \ell^2$? One trick is to compare the sum to related integrals.

$$\int_0^{n-1} \ell^2 d\ell \leq \sum_{\ell=1}^{n-1} \ell^2 \leq \int_0^{n-1} (\ell+1)^2 d\ell$$

Both integrals are of size $n^3/3 + \mathcal{O}(n^2)$, so $\sum_{\ell=1}^{n-1} \ell^2 = \frac{1}{3}n^3 + \mathcal{O}(n^2)$.

(exact formula is $\sum_{\ell=1}^{n-1} \ell^2 = \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$, can be proved by induction)

Therefore, Gaussian Elimination requires $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops. The main contribution is from the work inside the inner-most loop.

Forward/Backward Substitution: Flops

How many flops does it take to solve a triangular linear system? For example,

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} u_{i,k} x_k}{u_{i,i}}, \qquad i = n, n-1, \dots, 1.$$

Calculating each x_i requires one subtraction, one division, and $(n-i)$ multiplications and $(n-i-1)$ additions.

$$\text{flops} = \sum_{i=1}^{n} [2 + (n-i) + (n-i-1)] = \sum_{i=1}^{n} [1 + 2(n-i)].$$

Relabelling $\ell = n - i$, we get

$$\text{flops} = \sum_{\ell=0}^{n-1} [1 + 2\ell] = n + 2\left(\sum_{\ell=0}^{n-1} \ell\right) = n^2 + \mathcal{O}(n).$$

So forward/backward substitution takes $\mathcal{O}(n^2)$ flops. This is cheaper than Gaussian Elimination (and even faster than matrix-vector multiplication, $2n^2 + \mathcal{O}(n)$ flops!)

Memory Storage

When we calculate $A = LU$, we need to create two new matrices of size $n \times n$. This means we need $2n^2 \times$ (64 bits) of memory (in double precision).

To save memory, often L and U are both stored in one $n \times n$ matrix, where the upper triangular entries are U and the lower triangular entries are L. Since L has unit diagonal, we don't need to explicitly save those values.

For example,

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix}}_{U} \Longrightarrow \begin{bmatrix} 2 & 4 & -2 \\ 2 & 1 & 1 \\ -1 & 1 & 4 \end{bmatrix}.$$

If n is very large, sometimes the data in A is deleted and replaced with this compressed LU information ("in-place" factorisation).

LU Factorisation: Existence

Since we form E_k by calculating $m_j = a_j / a_k$ (for our given column), we need $a_k \neq 0$. This means, for example, that

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix},$$

does not have an LU factorisation (even though A is invertible).

Definition

The leading principal $r \times r$ minor of A is $\det(A^r)$, where $A^r \in \mathbb{R}^{r \times r}$ is the matrix of the first r rows and columns of A.

Theorem (LU decomposition existence and uniqueness)

If the first $n-1$ leading principal minors of A are all nonzero, then A has a unique LU factorisation.

LU Factorisation: Stability

Actually, the situation is even worse: LU factorisation is very sensitive to rounding errors. So, even if an LU factorisation exists, Gaussian elimination may not be able to find it accurately!

For example, consider a small perturbation of the matrix from the last slide:

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}.$$

A is invertible and well-conditioned ($\kappa(A) \approx 2.62$), so we might not expect any problems.

By hand, we get an LU factorisation by subtracting 10^{20} times the first row from the second row:

$$A = \underbrace{\begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}}_{U}.$$

LU Factorisation: Stability

$$\underbrace{\begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}}_{U}.$$

In double precision, $1 - 10^{20}$ is rounded to -10^{20}. So numerically we get

$$LU = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix} \neq A,$$

so our numerical LU factorisation is very wrong: $\|A - LU\|_2 = 1$.

Issue

Gaussian Elimination has two problems: LU factorisations do not exist for all invertible A, and can be fail catastrophically in the presence of rounding errors.

Pivoting

Fortunately there is one solution to both of these problems: row swaps. Remember from first year that this is a valid operation in row reduction.

In Gaussian Elimination, this is called pivoting (not used until late 1940s).

Row swaps can be written as multiplication by a permutation matrix: a square matrix given by applying the same row swap to the identity.

For example, to swap rows 2 and 3 of a 4×4 matrix, premultiply it by

$$P = \begin{bmatrix} \boldsymbol{e}_1^T \\ \boldsymbol{e}_3{}^T \\ \boldsymbol{e}_2{}^T \\ \boldsymbol{e}_4^T \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Permutation matrices are always invertible, and the product of two permutation matrices is also a permutation matrix. The identity matrix is a permutation matrix (corresponding to “no row swaps”).

Pivoting: Example

Let's try to calculate an LU factorisation (with pivoting) for

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 4 & 6 & 8 \end{bmatrix}$$

First, we put zeros in the first column by calculating

$$E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 3 \\ 0 & 2 & 4 \end{bmatrix}.$$

Without pivoting, we would now be stuck: the $(2,2)$ entry is zero, so we can't build E_2. Instead, swap rows 2 and 3 with a permutation:

$$P_2 E_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 3 \\ 0 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 4 \\ 0 & 0 & 3 \end{bmatrix} \qquad \longleftarrow \text{upper triangular!}$$

Pivoting

In general, at each step we first do an optional row swap, then introduce zeros in an elimination step.

$$\underset{\text{Pivot selection}}{\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & a_{i,k} & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix}} \xrightarrow[P]{} \underset{\text{Row swap}}{\begin{bmatrix} \times & \times & \times & \times & \times \\ & a_{i,k} & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix}}$$

$$\xrightarrow[E]{} \underset{\text{Elimination}}{\begin{bmatrix} \times & \times & \times & \times & \times \\ & a_{i,k} & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \end{bmatrix}}$$

Pivoting

Using this process, we eventually get an upper triangular matrix:

$$E_{n-1}P_{n-1}E_{n-2}P_{n-2}\cdots E_1P_1A = U.$$

Is this an LU factorisation of A?

Almost! We are lucky yet again: for any E_k and any permuation P, we have

$$PE_k = \widetilde{E}_k P,$$

where $\widetilde{E}_k$ comes from applying the row swap P to the entries below the diagonal (no change to the diagonal). That is, $\widetilde{E}_k$ is also an elementary matrix!

Rearranging, we get

$$\widetilde{E}_{n-1}\widetilde{E}_{n-2}\cdots\widetilde{E}_1P_{n-1}P_{n-2}\cdots P_1A = U.$$

Like before, $L = \widetilde{E}_1^{-1}\cdots\widetilde{E}_{n-1}^{-1}$ is unit lower triangular, so we have computed

$$\boxed{PA = LU}$$

LU Factorisation with Pivoting

Pivoting addresses our first problem: LU factorisations not existing for some invertible matrices.

Theorem

If A is invertible, then the factorisation $PA = LU$ exists, for a permutation matrix P, unit lower triangular matrix L and upper triangular matrix U.

In fact, all entries of L are ≤ 1 in absolute value.

By choosing our row swaps cleverly, we also solve the second problem: numerical instability.

Partial Pivoting: Example (revisited)

In our earlier example, we got catastrophic rounding errors when L had very large entries:

$$\underbrace{\begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}}_{U}.$$

We can fix this by using partial pivoting: do row swaps so that the pivot (diagonal) entry is the entry to be eliminated with largest absolute value.

In this case, the largest entry (in absolute value) to be eliminated in the first column is 1, so we swap to make this the pivot:

$$P_1 A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 10^{-20} & 1 \end{bmatrix},$$

$$E_1 P_1 A = \begin{bmatrix} 1 & 0 \\ -10^{-20} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 10^{-20} & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 - 10^{-20} \end{bmatrix}}_{U}.$$

Partial Pivoting: Example (revisited)

We have

$$\underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}}_{A} = \begin{bmatrix} 1 & 1 \\ 10^{-20} & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ 10^{-20} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 - 10^{-20} \end{bmatrix}}_{U}.$$

With rounding errors in double precision, $1 - 10^{-20}$ is represented as 1, and so we actually get

$$LU = \begin{bmatrix} 1 & 0 \\ 10^{-20} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 10^{-20} & 1 + 10^{-20} \end{bmatrix} \approx PA.$$

Now, our error is very small: $\|PA - LU\|_2 = 10^{-20}$.

(actually, $1 + 10^{-20}$ is represented as 1 in double precision, so computed this way we get zero error)

Partial Pivoting: Algorithm

To solve the system $A\boldsymbol{x} = \boldsymbol{b}$, we do:

- Compute factorisation $PA = LU$.
- Solve $LU\boldsymbol{x} = P\boldsymbol{b}$ using forwards/backwards substitution.
  - Solve $L\boldsymbol{y} = P\boldsymbol{b}$ using forwards substitution
  - Solve $U\boldsymbol{x} = \boldsymbol{y}$ using backwards substitution

As before, we calculate $\boldsymbol{y}$ at the same time as doing Gaussian Elimination (if we want to).

Partial Pivoting: Stability

As we might expect, once we add pivoting we get stability:

Theorem

In floating-point arithmetic, Gaussian Elimination applied to $A\boldsymbol{x} = \boldsymbol{b}$ computes a vector $\widetilde{\boldsymbol{x}}$ which solves $(A + \delta A)\widetilde{\boldsymbol{x}} = \boldsymbol{b}$ for some error matrix δA with

$$\frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\rho\epsilon),$$

where ϵ is the machine precision and $\rho = (\max_{i,j}|u_{i,j}|)(\max_{i,j}|l_{i,j}|)/\max_{i,j}|a_{i,j}|$ is the "growth factor". With partial pivoting this simplifies with $\max_{i,j}|l_{i,j}| = 1$.

Note: the constant inside $\mathcal{O}(\cdot)$ depends on the system size n and choice of matrix norm.

Using partial pivoting, the growth factor can be as large as $\rho = 2^{n-1}$ in theory, but in almost every practical setting ρ is not too large (usually closer to $\sqrt{n}$). So for all intents and purposes, Gaussian Elimination with partial pivoting is backwards stable.

Partial Pivoting: Residuals

Often it is useful to estimate how good our computed solution $\widetilde{\boldsymbol{x}}$ is.

Ideally, we would calculate the error $\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|$, but we can't do this because we don't know the true solution $\boldsymbol{x}$.

Instead, we often calculate the residual $\boxed{\boldsymbol{r} = \boldsymbol{b} - A\widetilde{\boldsymbol{x}}}$.

From our stability result,

$$\boldsymbol{r} = (A + \delta A)\widetilde{\boldsymbol{x}} - A\widetilde{\boldsymbol{x}} = \delta A\widetilde{\boldsymbol{x}}.$$

This means that $\|\boldsymbol{r}\| \leq \|\delta A\| \cdot \|\widetilde{\boldsymbol{x}}\|$, so the relative residual satisfies

$$\frac{\|\boldsymbol{r}\|}{\|A\| \cdot \|\widetilde{\boldsymbol{x}}\|} \leq \frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\rho\epsilon).$$

The relative residual is approx. $\mathcal{O}(\epsilon)$ (if ρ is reasonable) and can be computed from known quantities.

Partial Pivoting: Summary

To summarise, Gaussian Elimination with partial pivoting:

- Is backwards stable (in practice).
- Requires $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops (same as without pivoting).

Recall that the information in the matrices L and U can be stored in a single $n \times n$ matrix. The permutation matrix P can be stored efficiently too: if

$$P = \begin{bmatrix} \boldsymbol{e}_{p_1}^T \\ \boldsymbol{e}_{p_2}^T \\ \vdots \\ \boldsymbol{e}_{p_n}^T \end{bmatrix},$$

for some rearrangement $(p_1, \ldots, p_n)$ of $(1, \ldots, n)$, we just store $p = \begin{bmatrix} p_1 & \cdots & p_n \end{bmatrix}^T$. Storing n integers is much simpler than storing n^2 floating point numbers (the entries in P).

Partial Pivoting: Example

Example Problem

$$\begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix}$$

First column: largest entry is 4, so make this the pivot (swap rows 1 and 2).

$$P_1 A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 2 & 4 & -2 \\ -2 & -3 & 7 \end{bmatrix}, \qquad \text{and} \qquad P_1 \boldsymbol{b} = \begin{bmatrix} 8 \\ 2 \\ 10 \end{bmatrix}.$$

New system is $P_1 A\boldsymbol{x} = P_1 \boldsymbol{b}$.

Partial Pivoting: Example

First column: apply elementary matrix to put zeros below diagonal of first column.

$$E_1 P_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 9 & -3 \\ 2 & 4 & -2 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 0 & -1/2 & -1/2 \\ 0 & 3/2 & 11/2 \end{bmatrix}, \quad \text{and} \quad E_1 P_1 \boldsymbol{b} = \begin{bmatrix} 8 \\ -2 \\ 14 \end{bmatrix}.$$

Second column: largest entry on/below diagonal is 3/2, so use this as the pivot (i.e. swap rows 2 and 3).

$$P_2 E_1 P_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 4 & 9 & -3 \\ 0 & -1/2 & -1/2 \\ 0 & 3/2 & 11/2 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & -1/2 & -1/2 \end{bmatrix}, \quad \text{and} \quad P_2 E_1 P_1 \boldsymbol{b} = \begin{bmatrix} 8 \\ 14 \\ -2 \end{bmatrix}.$$

Partial Pivoting: Example

<u>Second column</u>: apply elementary matrix to put zeros below diagonal of second column.

$$E_2 P_2 E_1 P_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & -1/2 & -1/2 \end{bmatrix} = \begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & 0 & 4/3 \end{bmatrix}, \quad E_2 P_1 E_1 P_1 \boldsymbol{b} = \begin{bmatrix} 8 \\ 14 \\ 8/3 \end{bmatrix}.$$

<u>Solve upper triangular system</u>: solve $(E_2 P_2 E_1 P_1) A \boldsymbol{x} = (E_2 P_2 E_1 P_1) \boldsymbol{b}$.

$$\begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & 0 & 4/3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 14 \\ 8/3 \end{bmatrix}$$

Using backwards substitution, we get $\boldsymbol{x} = \begin{bmatrix} -1 & 2 & 2 \end{bmatrix}^T$.

Partial Pivoting: Example

Let's reconstruct our LU factorisation. We got

$$E_2 P_2 E_1 P_1 A = \underbrace{\begin{bmatrix} 4 & 9 & -3 \\ 0 & 3/2 & 11/2 \\ 0 & 0 & 4/3 \end{bmatrix}}_{U}.$$

Remember $P_2 E_1 = \widetilde{E}_1 P_2$, where $\widetilde{E}_1$ comes from applying P_2 (swap rows 2 & 3) to the below-diagonal entries of E_1. That is,

$$E_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix}, \qquad \text{so} \qquad \widetilde{E}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix}.$$

Now we have $E_2 \widetilde{E}_1 P_2 P_1 A = U$.

Partial Pivoting: Example

Given $E_2 \widetilde{E}_1 P_2 P_1 A = U$, we have $PA = LU$ for $P = P_2 P_1$ and $L = \widetilde{E}_1^{-1} E_2^{-1}$. That is,

$$P = P_2 P_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

and

$$L = \widetilde{E}_1^{-1} E_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/3 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & -1/3 & 1 \end{bmatrix}.$$

(check for yourself that $PA = LU$)

Partial Pivoting in Python

In Python, Gaussian Elimination with partial pivoting is the standard method for solving linear systems:

- `numpy.linalg.solve` uses it to solve $A\boldsymbol{x} = \boldsymbol{b}$.
- `scipy.linalg.lu_factor` calculates the factorisation $A = PLU$, stored efficiently (L and U in one $n \times n$ matrix, P as a vector of integers).
  - Note two conventions: $A = PLU$ vs. $PA = LU$. What is the relationship between the two Ps?
- `scipy.linalg.lu_solve` uses a pre-calculated factorisation $A = PLU$ to solve $A\boldsymbol{x} = \boldsymbol{b}$.
- `scipy.linalg.lu` calculates $A = PLU$ but where P, L and U are all stored as $n \times n$ matrices (don't use for large systems!)

Complete Pivoting

Another approach is to swap rows and columns in each step, to make the pivot the largest (absolute) element of the whole reduced matrix.

This is called complete pivoting.

It is slower than partial pivoting, and in practice the improvement in stability is not worthwhile.

Why not A^{-1}?

Question

Why can't we just compute A^{-1} and then $\boldsymbol{x} = A^{-1}\boldsymbol{b}$?

You can do this if you want (`np.linalg.inv`), but it is almost never a good idea:

- Slower: you need to use GE with partial pivoting to calculate each column of A^{-1}, then do the multiplication. Done carefully, total flops is $\frac{8}{3}n^3 + \mathcal{O}(n^2)$, or about $4\times$ slower than solving $A\boldsymbol{x} = \boldsymbol{b}$ directly.
- Worse solution: usually more sensitive to rounding errors.

"In the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute A^{-1}." — Forsythe, Malcolm & Moler (1977).

Only compute A^{-1} when you have a specific reason to. Do not use it to solve linear systems.

Sparse LU Factorisations

If A is very large and has many zero entries, it is often better to store A as a sparse matrix (`scipy.sparse` module).

Instead of storing all n^2 entries, we just store the locations and values of the nonzero entries.

Sparse matrices are very common in scientific applications (e.g. when solving partial differential equations).

Directly solving sparse linear systems is tricky, because often A^{-1} will be dense (not sparse), and usually n is large enough that we don't want to store such a large matrix.

Sparse LU factorisation routines (e.g. `scipy.sparse.linalg.lu`) use pivoting methods that try to balance the growth factor ρ (i.e. numerical stability) vs. sparsity of L and U (speed).

Other Factorisations

There are many other common matrix factorisations that are useful for solving linear systems (and other important tasks). Some of the most important are:

- Cholesky decomposition: if A is square and symmetric positive definite, write $A = LL^T$ for lower triangular matrix L.
  - Useful for solving linear systems and generating correlated random numbers.
  - Takes $\frac{1}{3}n^3 + \mathcal{O}(n^2)$ flops ($2\times$ faster than GE).
  - If A symmetric but not positive definite, get $A = LDL^T$ where D is almost diagonal.
- QR factorisation: if A any $m \times n$ matrix with $m \geq n$ (not just square), write $A = QR$, where $Q^T Q = I$ and R is upper triangular.
  - Used for least-squares problems (overdetermined linear systems) or finding a basis for the column space of A.
  - Takes $\sim 2mn^2$ flops for an $m \times n$ matrix.

Other Factorisations

- Singular Value Decomposition (SVD): if A is any matrix (not just square), write $A = U\Sigma V^T$ where $U^T U = I$, $V^T V = I$ and Σ is diagonal.
  - Used for least-squares problems, finding a basis for column or null space of A, approximation of A by low-rank matrices.
  - Takes $\mathcal{O}(mn^2)$ flops for an $m \times n$ matrix (depends on exact algorithm used), but much slower than QR.

NumPy/SciPy can compute all of these factorisations.

For more details, see MATH3512.