Math 361S Lecture notes
Direct methods for linear systems: Gaussian elimination

Jeffrey Wong

February 19, 2018

Topics covered

- Linear systems: general (see also K&C, Section 4.1)
  - Motivation
  - Review: determinants, invertibility, outer products
- Gaussian elimination (see also K&C, Section 4.3)
  - Row operations and matrix form
  - LU Factorization via Gaussian elimination
  - Algorithm details
  - Theoretical description
- Pivoting
  - Permutation matrices
  - Partial pivoting
  - Scaled partial pivoting, complete pivoting
  - Implementation
  - Stability of Gaussian elimination (briefly)
- Operation counts
  - Big-O notation
  - Operation count for Gaussian elimination
  - How to interpret operation counts
  - Other considerations for efficiency

1 Introduction

In numerical linear algebra, a fundamental problem is to solve the linear system

$$Ax = b \text{ where } A \text{ is an } n \times n \text{ matrix }, b \in \mathbb{R}^n$$

for $x \in \mathbb{R}^n$. When A is invertible, the exact solution is

$$x = A^{-1}b.$$

Obtaining x numerically turns out to be more challenging - leading to a variety of methods with different properties. Because linear systems are so ubiquitous in scientific computing, it is essential to have efficient algorithms and methods that can be tailored to take advantage of special advantage.

For now, we will focus on solving systems where A is $n \times n$ and invertible.

2 Triangular systems

An *upper triangular matrix* is a matrix A whose entries below the main diagonal (the 'sub-diagonal entries') are zero:

$$a_{ij} = 0 \qquad \text{for } i > j.$$

Written out, the matrix looks like

$$\begin{bmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ 0 & a_{22} & \dots & \dots & a_{2n} \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn} \end{bmatrix}$$

Similarly, a matrix is *lower triangular* if the entries above the main diagonal are zero. A matrix is *unit upper/lower triangular* if it is upper/lower triangular and the diagonal entries are all ones, e.g. a unit lower triangular matrix has the form

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ a_{21} & 1 & \dots & \dots & 0 \\ a_{31} & a_{32} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ a_{n1} & a_{n2} & \dots & a_{n,n-1} & 1 \end{bmatrix}$$

If U is upper triangular then the linear system

$$Ux = b$$

it is easy to solve for x using *back-substitution*. First use the last equation (last row) to solve for x_n:

$$u_{nn}x_n = b_n \implies x_n = b_n/u_{nn}.$$

Then use that in the second-to-last row to solve for x_{n-1}:

$$u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = b_{n-1} \implies x_{n-1} = \frac{1}{u_{n-1,n-1}}\left(b_{n-1} - u_{n-1,n}x_n\right).$$

and in general, for the k-th row,

$$x_k = \frac{1}{u_{k,k}}\left(b_k - \sum_{i=k+1}^{n} u_{k,i}x_i\right)$$

which involves only the values $x_{k+1}, \cdots x_n$ we have already computed (see Algorithm 2 for pseudocode). Note that in the inner sum, we start at the $i+1$-th column since the matrix

---

Input: $U \in \mathbb{R}^{n\times n}$ upper tri. and $b \in \mathbb{R}^n$
Output: $x = U^{-1}b \in \mathbb{R}^n$
Require: $u_{ii} \neq 0$ for $1 \leq i \leq n$

```
x_n ← b_n/u_nn
for i = n − 1, n − 2, ··· 1 do
    x_i ← b_i
    for j = i + 1, ··· n do
        x_i ← x_i − u_ij x_j
    end for
    x_i ← x_i/u_ii.
end for
return x
```

---

is upper triangular (all the entries in $1, \cdots i-1$ in this row are zero).

If L is lower triangular, then the system

$$Lx = b$$

can be solved using *forward substitution*, the same as back substitution but starting with the first row.

3 LU Factorization

To solve the system $Ax = b$, a good approach is the following:

- Find a 'factorization' of A into simpler parts
- Solve simpler systems involving these parts

Here we consider one such approach: We look for an **LU factorization**

$$A = LU$$

where L is lower triangular and U is upper triangular. Then the system $Ax = b$ can be solved via the following:

1) Find the LU factorization $A = LU$

2) Solve $Ly = b$ using forward substitution

3) Solve $Ux = y$ using backward substitution

One benefit of this structure is that step (1) can be completed independent of the other two. Thus , once (1) is done, we can solve $Ax = b$ for a variety of b's without having to re-compute L and U, saving some work.

MATLAB: LU factorization (with pivoting, which we will discuss later) is done using `[L,U] = lu(A)`. Then the system can be solved using `x = U \ (L \ b)`. The backslash command will automatically know to use the correct method for the forward/backward solves.

4 Gaussian elimination

4.1 Row operations

Gaussian elimination, in turns out, is a good approach. You are (hopefully) familiar with the method, but we will develop the algorithm in excruiating detail in order to illustrate some key concepts and lay a foundation for later analysis of the algorithm from a numerical perspective.

Recall that a linear system

$$Ax = b$$

can be reduced by applying *elementary row operations* on the rows $R_1, \cdots R_n$ of the system. Each operation is equivalent to left multiplying A by a certain matrix E. This gives us an 'algorithmic' description (do x to rows, etc.) and a 'matrix' description ($A \to EA$).

There are three RO's, listed below with an example for a 3×3 matrix. Notice that in general, the matrix form E is obtained by applying the RO to the identity matrix I.

- Scaling: $R_i \to \lambda R_i$, e.g.

$$R_1 \to \lambda R_1 \implies E = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

  We will use this matrix later when improving some algorithms.

- Swapping (permutation of rows): $R_i \leftrightarrow R_j$, e. g.

$$R_1 \to \lambda R_2 \implies E = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

- Adding a mult. of a row to another: $R_i \to R_i - \lambda R_j$, e.g.

$$R_1 \to \lambda R_1 \implies E = \begin{bmatrix} 1 & 0 & 0 \\ -\lambda & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

4.2 Algorithm

The process for reducing A using Gaussian elimination is straightforward, if a little tedious to write out. Let $A^{(k)}$ be the matrix after the first $k-1$ columns are reduced. We reduce the matrix column-by-column, ending up with an upper triangular matrix:

$$A^{(1)} \xrightarrow[\text{col. 1}]{\text{reduce}} A^{(2)} \xrightarrow[\text{col. 2}]{\text{reduce}} \cdots \xrightarrow[\text{col. } n-1]{\text{reduce}} A^{(n)} = U$$

Let R_i be short for 'i-th row' and let $a_{ij}^{(k)}$ be the (i,j) entry of $A^{(k)}$. The algorithm (in mathematical notation) is as follows:

- Set L to be the $n \times n$ identity matrix
- For columns $k = 1, \cdots, n-1$, do the following to reduce the k-th column of $A^{(k)}$:
  - For $i = i+1, \cdots n$, replace the row R_i with $R_i - \tau_{ik} R_k$ where

    $$\tau_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}.$$

    is the **multiplier** that zeros the (i,j) entry.
  - Record τ_{ij} in the (i,j) entry of L.

The result is that A is reduced to an upper triangular matrix U. It turns out that $A = LU$ where L is the unit lower triangular matrix of multipliers, which we will prove shortly.

Theorem. *If $a_{kk}^{(k)} \neq 0$ at each step $k = 1, \cdots n-1$ then $A = LU$ where U is the upper triangular matrix obtained by Gaussian elimination and L is the unit lower triangular matrix of multipliers:*

$$\begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ \tau_{21} & 1 & \dots & \dots & 0 \\ \tau_{31} & \tau_{32} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ \tau_{n1} & \tau_{n2} & \dots & \tau_{n,n-1} & 1 \end{bmatrix}$$

The first part is easy to see; the condition guarantees all the steps of the algorithm work, and it is clear that the reduced matrix is upper triangular. The last part requires a little more work, and will be proven shortly.

Algorithm 1 does Gaussian elimination as described above. It is important to be careful and do only the operations that are needed. Note that:

- For the k-th column, we do not have to do anything to the first $k-1$ columns or the first $k-1$ rows.
- Furthermore, the sub-diagonal entries in the k-th will end up zero, so we do not need to compute $a_{ik} - (a_{ik}/a_{kk})a_{kk} = 0$.

Thus the inner loops only iterate over entries (i, j) with $k+1 \leq i, j \leq n$ (the entries below the k-th row and to the right of the k-th column). [1]

Algorithm 1 Gaussian Elimination: basic algorithm

Input: $A \in \mathbb{R}^{n \times n}$ and space for L
Output: U stored in A and L

```
L ← n × n identity matrix
for k = 1, 2, ··· n − 1 do          ▷ Reduce k-th column
    for i = k + 1, ··· n do          ▷ Update i-th row (below diagonal)
        m ← a_ik / a_kk
        for j = k + 1, ··· n do          ▷ move along columns of the row to update
            a_ij ← a_ij − m a_kj
        end for
        ℓ_ik ← m
    end for
end for
```

4.3 Example

Consider the linear system

$$Ax = b$$

where

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 3 & 3 & 9 \\ 3 & 3 & 5 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

[1] There are a few other considerations at the implementation level, since the efficiency arithmetic over vectors and rows/columns of matrices depends on the way the values are stored in memory. Certain ordering of loops can have advantages. In MATLAB/python, such issues are hidden from the user. We may discuss it in class if time permits. See Golub & Van Loan,x *Matrix Computations* for a good explanation.

Entries that are changed in the given step are shown in red.

$$A: \quad \begin{bmatrix} 2 & -1 & 1 \\ 3 & 3 & 9 \\ 3 & 3 & 5 \end{bmatrix} \xrightarrow[R_3 \to R_3 - 3/2 R_2]{R_2 \to R_2 - 3/2 R_1} \begin{bmatrix} 2 & -1 & 1 \\ 0 & 9/2 & 15/2 \\ 0 & 9/2 & 7/2 \end{bmatrix} \xrightarrow{R_3 \to R_3 - R_2} \begin{bmatrix} 2 & -1 & 1 \\ 0 & 9/2 & 15/2 \\ 0 & 0 & -4 \end{bmatrix}$$

$$L: \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix}$$

The result is that $A = LU$ where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 9/2 & 15/2 \\ 0 & 0 & -4 \end{bmatrix}.$$

Now we solve $Ax = b$ by first solving

$$Ly = b \iff \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

using forward substitution to obtain $y_1 = 1$, then

$$3y_1/2 + y_2 = 0 \implies y_2 = -3/2$$

and $y_3 = 1$. Then, finally, solve $Ux = y$ using back substitution.

If we do not need the LU factorization itself, the more familiar augmented matrix method where we instead apply the row operations also to b and ignore constructing L:

$$\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \xrightarrow[R_3 \to R_3 - 3/2 R_2]{R_2 \to R_2 - 3/2 R_1} \begin{bmatrix} 1 \\ -3/2 \\ -1/2 \end{bmatrix} \xrightarrow{R_3 \to R_3 - R_2} \begin{bmatrix} 1 \\ -3/2 \\ 1 \end{bmatrix}$$

This gives us y, and then we solve $Ux = y$ using back substitution. Thus this method is equivalent to doing the forward substitution $Ly = b$ in parallel to the reduction of A.

4.4 Economizing storage

One more improvement can be made by observing that once an entry is zeroed out in A, that space is free to use. The multipliers can be stored there instead of allocating a separate matrix. The diagonal entries of L are all ones, so they do not need to be stored at all.

The output is then the matrix A where the entries on/above the diagonal form U, and the ones below the diagonal form L. In compact form, the previous example goes like this:

$$\begin{bmatrix} 2 & -1 & 1 \\ 3 & 3 & 9 \\ 3 & 3 & 5 \end{bmatrix} \xrightarrow[R_3 \to R_3 - 3/2 R_2]{R_2 \to R_2 - 3/2 R_1} \begin{bmatrix} 2 & -1 & 1 \\ 3/2 & 9/2 & 15/2 \\ 3/2 & 9/2 & 7/2 \end{bmatrix} \xrightarrow{R_3 \to R_3 - R_2} \begin{bmatrix} 2 & -1 & 1 \\ 3/2 & 9/2 & 15/2 \\ 3/2 & 1 & -4 \end{bmatrix}$$

and the output would be that A stores the data

$$\begin{bmatrix} 2 & -1 & 1 \\ 3/2 & 9/2 & 15/2 \\ 3/2 & 1 & -4 \end{bmatrix}.$$

The key thing is to remember that the variable A now houses L and U and not the original matrix. Doing so is okay as long as we only want to solve linear systems $Ax = b$ and don't need the entries of A for anything else.

Algorithm 2 Gaussian Elimination: compact storage

Input: $A \in \mathbb{R}^{n \times n}$
Output: L and U both stored in A

```
for k = 1, 2, ··· n − 1 do
    for i = k + 1, ··· n do
        m ← a_ik / a_kk
        for j = k + 1, ··· n do
            a_ij ← a_ij − m a_kj
        end for
        a_ik ← m
    end for
end for
```

4.5 Theory

The theoretical treatment here illustrates a common theme in numerical linear algebra: a description of the algorithm in terms of matrices is often more useful in proofs than the 'algorithmic' one (like the pseudocode).

4.5.1 Some preliminaries:

The *outer product* of an $1 \times m$ column vector v and $1 \times n$ column vector w is

$$vw^T$$

which is an $m \times n$ **matrix** whose (i, j)-th entry is $v_i w_j$. The outer product contains all possible pairwise products of elements in v and w. Note that if $v, w \in \mathbb{R}^n$ and the order is reversed, we instead get the dot product

$$v^T w = w^T v$$

which is a *scalar*.

The *standard basis vectors* $\vec{e}_1, \cdots \vec{e}_n$ for $\mathbb{R}^n$ are defined to be column vectors where $\vec{e}_k$ has 1 in its k-th entry and zeros in all other entries. These vectors have nice properties, for instance:

$$e_i e_j^T = \text{matrix with } (i, j) \text{ entry 1, all others 0.}$$

Taking the dot product of e_i with $x \in \mathbb{R}^n$ 'selects' the i-th element:

$$e_i^T x = x_i.$$

Similarly, for a square matrix A,

$$e_i^T A = i\text{-th row of } A, \qquad Ae_i = i\text{-th column of } A.$$

Lastly, note that if M, A are matrices and A has columns $\vec{x}_1, \vec{x}_2, \cdots \vec{x}_n$ then

$$MA = \begin{bmatrix} M\vec{x}_1 & M\vec{x}_2 & \cdots & M\vec{x}_n \end{bmatrix}$$

i.e. MA is obtained by applying M to each of the columns of A.

MATLAB note: Vectors are actually matrices in Matlab, either row vectors ($1 \times n$) or column vectors ($n \times 1$). If you have two *row* vectors `v` and `w` then `v*w'` will give the inner product, `v'*w` will give the $n \times n$ outer product and `v*w` will give an error (in Matlab, `'` is transpose). This is a common source of error.

Further remark (valid only for r2017 and up): If `v,w` are both row or column vectors then he elementwise `v.*w` will return the vector whose i-th entry is $v_i w_i$, with the same shape as `v` and `w`.

However, if `v` is a row vector and `w` is a column vector, then `v.*w` will (unexpectedly) return the outer product.

4.5.2 Row reduction as a matrix

First we construct the matrix that reduces a column in general. Given a vector $x \in \mathbb{R}^n$, we seek a matrix that zeros out elements $k+1, \cdots, n$ of x and leaves the rest untouched:

$$M \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

This matrix is the composition of row operations to zero out the $k+1$ to n elements of $\vec{x}$. By applying these row operations to the identity matrix, we find that

$$M = \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & -x_{k+1}/x_k & 1 & \ddots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \ddots & 0 \\ 0 & \dots & \underbrace{-x_n/x_k}_{\text{k-th col.}} & \dots & 0 & 1 \end{bmatrix}$$

The matrix can be compactly written in terms of an outer product:

$$M = I - \vec{v}e_k^T, \qquad \vec{v} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_{k+1}/x_k \\ \vdots \\ x_n/x_k \end{bmatrix}.$$

i.e. $\vec{v}_i = 0$ for $i \leq k$ and $\vec{v}_i = x_i/x_k$ for $i > k$. This fact is easily verified by computing

$$(Mx)_i = x_i - (e_k^T x)\vec{v}_i = x_i - x_k\vec{v}_i = \begin{cases} x_i & i \leq k \\ 0 & i > k. \end{cases}$$

It is also easy to check that

$$M^{-1} = I + \vec{v}e_k^T. \tag{1}$$

This is because M is inverted by doing the same row operations with a minus sign in reverse order (i.e. $R_i \to R_i - \lambda R_k$ is inverted by doing $R_i \to R_i + \lambda R_k$). One could also check by verifying $MM^{-1} = I$ using (1).

4.5.3 Matrix form of the algorithm

Recall that $A^{(1)} = A$ and that $A^{(k)}$ is the the matrix after zeroing out sub-diagonal entries in the first $k-1$ columns. Denote the row operations need to reduce the k-th column by M_k. Then the path to U is

$$A^{(1)} \xrightarrow[M_1]{} A^{(2)} \xrightarrow[M_2]{} \cdots \xrightarrow[M_{n-1}]{} A^{(n)} = U$$

where U (the result) is an upper triangular matrix. Because row operations are matrices, we can define M_k to be the matrix such that

$$A^{(k+1)} = M_k A^{(k)}.$$

The matrix M_k reduces the k-th column of $A^{(k)}$, so by the previous discussion,

$$M_k = I - v^{(k)} e_k^T, \qquad v_i^{(k)} = \begin{cases} 0 & i \le k \\ \tau_{ik} & i > k \end{cases}$$

where $\tau_{ik} = a_{ik}^k / a_{kk}^{(k)}$ are the multipliers as before. Explicitly,

$$M_k = \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & -\tau_{ik} & 1 & \ddots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \ddots & 0 \\ 0 & \dots & \underbrace{-\tau_{nk}}_{\text{k-th col.}} & \dots & 0 & 1 \end{bmatrix}$$

The result of the reduction is

$$U = M_{n-1} M_{n-2} \cdots M_1 A$$

which is equivalent to the pseudocode algorithm from before. That is,

$$A = LU, \qquad L = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1}.$$

To prove the theorem (and verify the algorithm works), we need to check that

i) U is upper triangular

ii) L is lower triangular and the (i,j)-th entry is τ_{ij}.

The details are left to you (and homework); a sketch is provided below. The important point is that different representations (via matrices or via row operations) give different ways to argue the same thing. Sometimes one approach gives a much cleaner proof (here, both are about the same).

4.5.4 Sketch of proof

For (i), it is enough to show that applying $M^{(k)}$ to a matrix leaves the first $k-1$ columns untouched. This is true because the k-th row of $A^{(k)}$ has zero entries up to column $k-1$, so row reduction using this row does not affect the first $k-1$ columns. Alternately,

$$M_k A^{(k)} = A^{(k)} - v^{(k)} e_k^T A^{(k)} = A^{(k)} - v^{(k)} R_k$$

where R_k is the k-th row of $A^{(k)}$. Since the first $k-1$ columns are reduced, the row R_k has zeros in its first $k-1$ entries, so the outer product $v^{(k)} R_k$ has all zeros in its first $k-1$ columns.

For (ii), note that

$$L = M_1^{-1} \cdots M_{n-1}^{-1}$$

is the result of applying the row operations corresponding to these matrices to the identity: 'un-reduce' (apply the inverse of the row reduction) the $n-1$-th column, then the $n-2$th and so on. This process deposits the multipliers τ_{ij} into the desired entries and, as in (i), M_k^{-1} does not touch the columns to the right of column k (so our progress is never undone).

Alternately, one could use the formula (1) and show that

$$L = I + \sum_{k=1}^{n-1} v^{(k)} e_k^T.$$

The k-th term puts the correct multipliers in the k-th column. The outer product notation lets us write a matrix (L) a 'sum' of its columns ($v^{(k)} e_k^T$ is the matrix that is zero except for the k-th column).

5 Pivoting

5.1 The problem

For the initial discussion and motivating example, see K&C Section 4.3.

5.2 Permutation matrices

Recall that swapping rows i and j is an elementary row operation; its associated matrix E is obtained by swapping rows i and j in the identity matrix I. A (row) **permutation matrix** P is a product of row-swap matrices:

$$P = P_n P_{n-1} \cdots P_1.$$

Applying P to a matrix A permutes (re-orders) the rows of A, equivalent to applying the sequence of row-swap operations to A. For example, the matrices for the RO's

$$R_1 \leftrightarrow R_2, \quad R_2 \leftrightarrow R_3 \tag{2}$$

are, respectively,

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

The permutation matrix that does the two swaps (2) in sequence is

$$P = P_2 P_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}. \tag{3}$$

If A is a 3×3 matrix with rows R_1, R_2, R_3 then

$$PA = \begin{bmatrix} - & - & R_2 & - & - \\ - & - & R_3 & - & - \\ - & - & R_1 & - & - \end{bmatrix}.$$

The **permutation vector** p is a more compact representation of P. This vector is constructed by applying the defining rows swaps to $\{1, 2, \cdots, n\}$. In the previous example,

$$\{1, 2, 3\} \to \{2, 1, 3\} \to \{2, 3, 1\}$$

so $p = \{2, 3, 1\}$. This vector tells us how the rows of the permuted and original matrix are related. for our purposes, the key property is that

$$(PA)_{ij} = A_{p(i),j} \tag{4}$$

or in words,

the i-th row of PA is the $p(i)$-th row of A.

Put yet another way,

$$p(i) = j \implies \text{row } j \text{ of } A \text{ gets moved to row } i \text{ of } PA.$$

For the permutation matrix in (3), the 1-st row of PA is the 2-nd row of A since $p(1) = 2$.

A few other properties are worth noting:

- $P^{-1} = P^T$ (see HW)
- $\det(P)$ is $+1$ for an even number of swaps and -1 for an odd number of swaps.
- A (column) permutation matrix Q can be constructed in a similar way, as a product of column swaps E (recall that these are applied on the right: $A \to AE$).

  Multiplying by Q on the right permutes the columns of A. Since $(PA)^T = A^T P^T$, the row/column versions are related by $Q = P^T$.

5.3 Partial pivoting

In matrix form, Gaussian elimination with pivoting yields

$$M_{n-1}P_{n-1}M_{n-2}P_{n-2}\cdots M_1 P_1 A = U$$

where M_k is the row-reduction for column k and P_k is the row-swap applied to column k.

It turns out that the P_k's can be factored out (proof omitted), as if we had done all of the row swaps on A at the start. The result is the following theorem:

Theorem. *If $A \in \mathbb{R}^{n\times n}$ is invertible then there is a permutation matrix P and unit lower triangular L, upper triangular U such that*

$$PA = LU.$$

Gaussian elimination with pivoting gives L and U (in the same way as before) and

$$P = P_{n-1}P_{n-2}\cdots P_1$$

where P_k is the row-swap applied to column k.

Note that the P produced by Gaussian elimination depends on the pivoting scheme. For a given P, the factors L and U are unique.

Notation: We refer to the factorization $PA = LU$ as the 'LU factorization' of A even when there is a P in it. To be precise, the factorization of A is $P^T LU$.

5.4 Example: Gaussian elimination with partial pivoting

We obtain the factorization $PA = LU$ for

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 0 & 1 \\ -2 & 2 & 4 \end{bmatrix}$$

using partial pivoting. The multipliers are shown in red, stored in the zero-ed out entry as in the algorithm (only for convenience; for the purposes of the row reduction from A to U the entries there are zeros).

Initialization: set the permutation vector to $p = \{1, 2, 3\}$.

First column: The largest (absolute) entry is in the third row, so swap rows 3 and 1:

$$\underbrace{\begin{bmatrix} 1 & 2 & 4 \\ 1 & 0 & 1 \\ -2 & 2 & 4 \end{bmatrix}}_{A} \xrightarrow{R_1 \leftrightarrow R_3} \underbrace{\begin{bmatrix} -2 & 2 & 4 \\ 1 & 0 & 1 \\ 1 & 2 & 4 \end{bmatrix}}_{P_1 A}, \quad \underbrace{p = \{3, 2, 1\}}_{\text{equiv. to } P_1}$$

The matrix for this swap is $P_1 = \left[\begin{smallmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{smallmatrix}\right]$. Now reduce the new second and third rows:

$$\begin{bmatrix} -2 & 2 & 4 \\ 1 & 0 & 1 \\ 1 & 2 & 4 \end{bmatrix} \xrightarrow[R_3 \to R_3 - (-1/2)R_1]{R_2 \to R_2 - (-1/2)R_1} \underbrace{\begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 1 & 3 \\ -1/2 & 3 & 6 \end{bmatrix}}_{M_1 P_1 A}, \quad p = \{3, 2, 1\}$$

Note that the row 2 and row 3 that we actually reduce are row 2 and row 1 of the original matrix A. The permutation vector tells us this fact: $p(2) = 2$ and $p(3) = 1$.

Second column: The largest absolute entry in $\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ is 3, so we swap rows 2 and 3:

$$\begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 1 & 3 \\ -1/2 & 3 & 6 \end{bmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \underbrace{\begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 3 & 6 \\ -1/2 & 1 & 3 \end{bmatrix}}_{P_2 M_1 P_1 A}, \quad \underbrace{p = \{3, 1, 2\}}_{\text{equiv. to } P_2 P_1}$$

The matrix for this swap is $P_2 = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{smallmatrix}\right]$. Now reduce ths column in PA:

$$\begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 3 & 6 \\ -1/2 & 1 & 3 \end{bmatrix} \xrightarrow{R_3 \to R_3 - (1/3)R_2} \underbrace{\begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 3 & 6 \\ -1/2 & 1/3 & 1 \end{bmatrix}}_{M_2 P_2 M_1 P_1 A}, \quad p = \{3, 1, 2\}.$$

The result is that $PA = LU$ where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/2 & 1/3 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} -2 & 2 & 4 \\ 0 & 3 & 6 \\ 0 & 0 & 1 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

noting that P corresponds to the permutation vector $p = \{3, 1, 2\}$.

5.5 Implementation

When implemented, it is standard to **not actually swap rows of** A in memory. Instead, we simply keep track of the rows swaps necessary using p. When reducing column k, we are applying the steps to PA but have A stored in memory instead.

For reducing the k-th column, we do the following:

- Find the row index r to swap with row k in PA.
- Swap $p(r)$ and $p(k)$ (update the permutation vector); this updates P
- Reduce the k-th column in PA via row swaps.

Because A is stored in memory instead of PA, to access row i of PA we must instead access the $p(i)$-th row of A (recall the rule (4) from earlier):

$$\underbrace{(PA)_{ij}}_{\text{theoretical}} = \underbrace{\texttt{A(p(i),j)}}_{\text{in code}}.$$

For the pseudocode algorithm see K&C, section 4.3 (p171-172).

> **Remark:** The cost of computing $p(i)$ is typically less than the cost of swapping the values in the rows of A. Whether this is true depends on details like the way data in A is arranged in memory.

5.6 Example: Partial pivoting, implicit swapping

We apply the algorithm to the previous example, showing A as it is stored in memory.

Initialization: Set $p = \{1, 2, 3\}$. Denote by R_i the i-th row of A.

First column: The largest (absolute) entry is in the third row, so swap 3 and 1 in p:

$$\{1, 2, 3\} \xrightarrow{(R_1 \leftrightarrow R_3)} \{3, 2, 1\}.$$

However, the rows R_i of A are left unswapped!

Now we need to reduce the new second and third rows of PA, which are rows $p(2) = 2$ and $p(3) = 1$ in the stored matrix A. The row operations done in the code on A are:

$$R_{p(2)} \leftarrow R_{p(2)} - (-1/2)R_{p(1)}, \qquad R_{p(3)} \leftarrow R_{p(3)} - (-1/2)R_{p(2)}$$

or, explicitly,

$$R_2 \leftarrow R_2 - (-1/2)R_3, \qquad R_1 \leftarrow R_1 - (-1/2)R_3.$$

The result is

$$\begin{bmatrix} 1 & 2 & 4 \\ 1 & 0 & 1 \\ -2 & 2 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} -1/2 & 3 & 6 \\ -1/2 & 1 & 3 \\ -2 & 2 & 4 \end{bmatrix}, \quad p = \{3, 2, 1\}$$

We have reduced the first column of PA, which is now

$$PA = \begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 3 & 6 \\ -1/2 & 1 & 3 \end{bmatrix} \quad \text{(not stored!)}.$$

Second column: To find the pivot, we look at the second column of PA (rows $i = 2, 3$), which are the first two rows of A.

The largest is in row $p(3) = 1$, so we swap entries 2 and 3 to update p:

$$\{3, 2, 1\} \xRightarrow{(R_2 \leftrightarrow R_3)} \{3, 1, 2\}.$$

Again, A is left unchanged; at this point in the algorithm, we have

$$PA = \begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 1 & 3 \\ -1/2 & 3 & 6 \end{bmatrix}.$$

Now reduce the second column of PA using $R_{p(3)} \to R_{p(3)} - (1/3)R_{p(2)}$, which is

$$R_2 \to R_2 - (1/3)R_1.$$

The result is

$$A = \begin{bmatrix} -1/2 & 3 & 6 \\ -1/2 & 1/3 & 1 \\ -2 & 2 & 4 \end{bmatrix}, \qquad p = \{3, 1, 2\}$$

To reconstruct L and U, compute

$$PA = \begin{bmatrix} -2 & 2 & 4 \\ -1/2 & 3 & 6 \\ -1/2 & 1/3 & 1 \end{bmatrix},$$

and then separate this result into L and U in the usual way.

Put another way, the algorithm actually produces

$$P^{-1}\tilde{A} = P^T\tilde{A}$$

where $\tilde{A}$ stores L and U in the lower/upper triangular parts; $\tilde{A}$ is recovered by applying P.

5.7 Using the result to solve linear systems

There are several ways to write the code to solve the linear system

$$Ax = b$$

using LU factorization, depending on the output scheme.

Version 1 (with explicit swapping): Write a function

```
[L,U,p] = lu(A)
```

that returns the matrices L, U and permutation vector p such that $PA = LU$, and a function

```
x = lusolve(L,U,p,b)
```

that solves $Ly = Pb$ using forward substitution, then $Ux = y$ using forward substitution. Note that for a column vector v (e.g. b or a column of A), Pv can be obtained from p via

```
for i=1:n
    swap(v(i),v(p(i)))
end
```

where `swap` denotes swapping the values of the two variables.

With this method, either `lu` needs to explicitly swap rows when reducing A or L and U must be reconstructed at the end, which takes a bit of work in either case.

Version 2 (no explicit swapping): The function `[A,p] = lu(A)` returns p and the LU factorization stored compactly in A as in section 5.5, i.e. PA contains the elements of L below the diagonal and U on/above the diagonal. The solve function

```
x = lusolve(A,p,b)
```

takes as its input A and p from `lu` (not the original matrix!). The entries of L and U are accessed via the formulas

$$\ell_{ij} = \texttt{A(p(i),j)}, \quad i < j, \qquad u_{ij} = \texttt{A(p(i),j)}, \quad i \geq j.$$

The algorithm is shown below. Note that $\ell_{ii} = 1$ so there is no division in the loop for y.

Input: A and p from `lu`, b

$y_1 \leftarrow b_{p_1}/a_{p_1,p_1}$

for $i = 2, \cdots n$ **do** ▷ Forward subs. to solve $Ly = Pb$

$\quad y_i \leftarrow b_{p_i} - \sum_{j=1}^{i-1} a_{p_i,j} y_j$

end for

$x_1 \leftarrow y_1$

for $i = n-1, n-2, \cdots, 1$ **do** ▷ Back subs. to solve $Ux = y$

$\quad x_i \leftarrow \left(y_i - \sum_{j=i+1}^{n} a_{p_i,j} x_j\right) / a_{p_i,p_i}$

end for

return x

Remark: A little space can be saved by overwriting b with y in solving $Ly = Pb$. In the algorithm above, y_i can be replaced by b_{p_i}, e.g. the inside of the first loop becomes

$$b_{p_i} \leftarrow b_{p_i} - \sum_{j=1}^{i-1} a_{p_i,j} b_{p_j}.$$

5.8 Scaled partial pivoting

Again, see K&C for the motivating example.

Scaled partial pivoting is similar to partial pivoting but selects the entry that is largest in magnitude *relative to the 'size' of the row.* In practice, we first compute the **row scales**

$$s_i = \max_{1 \le j \le n} |a_{ij}|,$$

i.e. s_i is the max. size of entries in the i-th row. Then, before reducing the k-th column, find the row r such that

$$\frac{|a_{rk}|}{s_r} = \max_{i \le k \le n} \frac{|a_{ik}|}{s_i} \tag{5}$$

and swap rows r and k and the scales s_r and s_k (think of the scales as being attached to the rows here). The rest of the reduction proceeds as before.

Practical note: Note that the scales are only computed once at the start. The scaled could also be re-computed after each column is reduced. However, doing so typically does not provide much benefit, so the scales are left fixed to save some work.

As with partial pivoting, if rows are implicitly swapped then the row indices in (5) should be replaced with p(index):

$$\frac{|a_{p_r,k}|}{s_{p_r}} = \max_{i \le k \le n} \frac{|a_{p_i,k}|}{s_{p_i}}.$$

5.9 Example: Gaussian elimination with scaled partial pivoting

Here we do scaled partial pivoting (explicitly swapping rows) on the earlier example. The matrix and scales are

$$\begin{bmatrix} 1 & 2 & 4 \\ 1 & 0 & 1 \\ -2 & 2 & 4 \end{bmatrix}, \quad \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 4 \end{bmatrix}.$$

First column: The first column, with elements divided by the scales, is

$$\begin{bmatrix} 1/4 \\ 1 \\ -1/2 \end{bmatrix}.$$

The largest entry is in the second row, so we swap the first/second rows (note that dividing by the scales is just done to find the pivot; the elements are not updated here):

$$\begin{bmatrix} 1 & 2 & 4 \\ 1 & 0 & 1 \\ -2 & 2 & 4 \end{bmatrix} \xrightarrow{R_1 \leftrightarrow R_2} \begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 4 \\ -2 & 2 & 4 \end{bmatrix}, \quad s = \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix}, \quad p = \{2, 1, 3\},$$

Note that the scale array also has entries 1 and 2 swapped. Reducing, we obtain

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 4 \\ -2 & 2 & 4 \end{bmatrix} \xrightarrow[R_3 \to R_3 - (-2)R_1]{R_2 \to R_2 - R_1} \begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 3 \\ -2 & 2 & 6 \end{bmatrix}, \quad s = \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix}, \quad p = \{2, 1, 3\}$$

Second column: No pivoting is done here. After reducing, the result is

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 2 & 3 \\ -2 & 1 & 3 \end{bmatrix}, \qquad p = \{2, 1, 3\}.$$

5.10 Complete pivoting

The most robust (and expensive) pivoting strategy is to search through the columns as well as rows to find the largest element. That is, we find a row r and column s (in the un-reduced part of the matrix) such that

$$|a_{rs}| = \max_{k \le i,j \le n} |a_{ij}|.$$

This element is swapped with the pivot element using a row swap P_1 (rows r and k) and a column swap Q_1 (columns s and k):

$$A \to P_1 A Q_1.$$

We therefore obtain, in matrix form, the reduction

$$M_{n-1} P_{n-1} \cdots M_1 P_1 A Q_1 Q_2 \cdots Q_{n-1} = U.$$

It can be shown (as before) that this leads to the factorization

$$PAQ = LU$$

where $Q = Q_1 Q_2 \cdots Q_{n-1}$ is the matrix obtained by applying the *column* swaps used in reduction to I. A linear system $Ax = b$ is solved by writing it in the form

$$LUQ^T x = Pb.$$

First, solve $Ly = Pb$, then $Uz = y$ for $z = Q^T x$, and finally compute $x = Qz$.

However, because complete pivoting requires $(n-k)^2$ comparisons per column (see Section 6), it requires a significant amount of work (unlike partial pivoting). Because it is almost never needed, complete pivoting is rarely done in practice. The details are noted in the section on operation counts.

5.11 Stability of Gaussian elimination

When is Gaussian elimination unstable (i.e. rounding errors can propagate and grow large, exponentially with n)? The following[2] turns out to be true:

- With no pivoting: unstable (easy to find disastrous examples, even for small n)
- With partial pivoting: unstable (does much better than with no pivoting)
- With scaled partial pivoting: unstable, but 'practically' stable (disasters are rare)
- Complete pivoting: stable

Thus in practice, scaled partial pivoting is the best compromise between stability and efficiency when solving a 'random' linear system (i.e. one without any particular structure except that A^{-1} exists). The cost is minimal, and it is almost guaranteed to be stable. Complete pivoting is used only when absolutely necessary.

It is worth noting that some matrices have special structure that guarantees no pivoting is needed. Gaussian elimination *without pivoting* may be a good method to use for such systems.

[2] See Golub & Van Loan, *Matrix Computations* for a discussion.

6 Operation counts

6.1 Big-O Notation

A function $f(x)$ (or sequence) is 'Big-O' of $g(x)$ as $x \to a$, written

$$f = O(g(x)) \text{ as } x \to a$$

or more succinctly as

$$f = O(g(x))$$

if there is a constant $C > 0$ such that

$$|f(x)| \leq C|g(x)| \text{ as } x \to a \tag{6}$$

The values of a are always either $a = 0$ or $a = \pm\infty$. The limit is usually implied by context and is left unstated.

To be precise: The meaning of (6) is the following:

If $a = \infty$: there are values x_0 and C such that the bound holds for all $x > x_0$.

If $a = 0$: There are values x_0 and C such that the bound holds for $|x| < x_0$.

The notation expresses that f grows at most as fast as g (up to a constant factor) as $x \to a$. (Alternately, if $g \to 0$ then f decays at least as fast as g).

The same definitions apply for sequence a_n, where the only meaningful limit is $n \to \infty$.

For example (in the limit $x \to \infty$),

$$x^3 + x^2 + x = O(x^3), \qquad e^x + x^{10} = O(e^x).$$

Big-O provides an easy way to state the size of a term without being to specific. One can choose when to stop being exact if there are many terms, e.g. it is also true that

$$x^3 + x^2 + x = x^3 + O(x^2).$$

Since g is only an upper bound, a function that is $O(g)$ is also O(anything larger than g). For example,

$$\begin{aligned} x \to \infty: &\quad x^m = O(x^n) \text{ for } 0 < m < n \\ x \to 0: &\quad x^m = O(x^n) \text{ for } 0 < n < m \end{aligned}$$

Other examples:

- A degree n polynomial $P_n(x)$ is $O(x^n)$ as $x \to \infty$.

- A degree n polynomial with lowest degree term $a_k x^k$ is $O(x^k)$ as $x \to 0$.
- Similarly, a polynomial in $1/x$ is big-O of its smallest degree term as $x \to \infty$, e.g.

$$\frac{1}{x} - \frac{2}{x^3} + \frac{3}{x^5} = O(1/x) \text{ as } x \to \infty.$$

- Every polynomial is $O(e^x)$ as $x \to \infty$.

A Big-O term in an equation eats all smaller terms (convenient when we don't care about terms of that size). If $f_1 = O(g)$ and $f_2 = O(h)$ and $0 < g \leq h$ then

$$f_1 + f_2 = O(h).$$

Similarly, $O(f_1 f_2) = O(gh)$. Take, for instance, calculating a product of power series near $x = 0$ like

$$\sin x \cos x = (x - O(x^3))(1 - x^2/2 + O(x^4)).$$

The $O(x^3)$ term will absorb x^3 and above terms, so we only need to calculate the terms that are x^2 and below:

$$\sin x \cos x = x + O(x^3).$$

Note that $xO(x^4) = O(x^5)$ and the product of $O(x^3)$ and $O(x^4)$ terms is $O(x^7)$. The obvious downside is that we lose some information (what is the x^3 term above?), so it is important to only use $O(\cdots)$ when the details of those terms are not important.

6.2 Asymptotic-to notation

To be more precise, it is useful to have notation that includes the leading constant in the bound as well. We say that $f(x)$ is *asymptotic to* $g(x)$ in a limit $x \to a$, written

$$f \sim g \qquad \text{as } x \to a$$

if

$$\lim_{x \to a} \frac{f(x)}{g(x)} = 1.$$

Informally, $f \sim g$ means that

$$f(x) = g(x) + \text{smaller terms}.$$

Thus the expression tells us how fast f grows/decays. For example,

$$a_n = 2n^3 + n^2 + \cdots = O(n^3)$$

could be more precisely written as $a_n \sim 2n^3$ or $a_n = 2n^3 + O(n^2)$.

Notation: Sometimes, the optimal constant in the bound is included in the $O(\cdots)$, e.g. $a_n = 2n^3 + n^2 + \cdots = O(2n^3)$. Of course, it is true that $a_n = O(Cn^3)$ for *any* positive constant C, so this is just a convention.

6.3 Operation counting

Our interest here is in seeing how the work required by an algorithm scales with the problem size n (e.g. the dimension of the linear system). One crude measure of efficiency is to count up the number of arithmetic operations (floating point operations, or 'flops') done by an algorithm.

For simplicity, we will count **multiplication/division** as one operation, and ignore everything else unless otherwise noted. For counting flops, additions should be counted too, but it is often tedious. Sometimes additions/mults. are counted separately because they have different costs (addition is faster).

A word of caution: It is important to emphasize that this number is only a rough estimate of how the amount of work scales up; it may not tell us which of two algorithms is faster, particularly for small values of n. We are ignoring all sorts of essential considerations, including:

- Comparisons (to be discussed soon)
- Writing to memory
- Reading memory/arrays
- Loading chunks of data into the cache

all of which matter. Every type of operation has a different cost (e.g. division is more expensive than multiplication; comparisons are cheaper than arithmetic operations), so the efficiency of the algorithm depends in a complicated way on the number of each type of operation. Moroever, many low-level optimizations depend on context like memory layout, the way the loops are ordered, etc. so the speed of operations may depend on the code context as well.

The one trick to counting operations is this: We are mostly interested in an estimate

$$\text{leading term} + O(\text{smaller terms})$$

e.g. $\frac{1}{3}n^3 + O(n^2)$. The form permits us to skip over some operations that are negligible, like single multiplications done at the start to initialize some variable.

Otherwise, it is just a matter of going through the algorithm and counting them up. Operations in a for loop should be summed, of course. Below, we count the operations for Gaussian elimination (without pivoting).

6.3.1 Gaussian elimination

For the LU factorization step: given k and i, in the innermost (j) loop the updates

$$a_{ij} = a_{ij} - ma_{kj}$$

take 1 mult. each. In the i loop, we do one mult. to obtain m. Thus the number of multiplications required is

$$\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} \left(1 + \sum_{j=k+1}^{n} 1 \right) = \sum_{k=1}^{n-1} (n-k)(n-k+1) = \sum_{k=1}^{n-1} k(k+1) = \frac{1}{3}n^3 - \frac{1}{3}n = \frac{1}{3}n^3.$$

It is easy to check (see HW) that both the back and forward substitution steps take $O(n^2)$ mults. Since the n^3 term is much larger, most of the work is done in computing the LU factorization; solving $Ax = b$ requires

$$\frac{1}{3}n^3 + O(n^2)$$

multiplications in total. (Note: if counting additions too, the result is $\frac{2}{3}n^3 + O(n^2)$).

On calculating the inverse: Gauss-Jordan elimination (which reduces A to the identity instead of stopping at U) requires $n^3/2 + O(n^2)$ multiplications. Calculating the inverse of A by this method takes n^3 mults; while the constant can be improved a bit by more clever means, finding A^{-1} and then computing $A^{-1}b$ is still much more inefficient than the $n^3/3$ mults. required for Gaussian elimination.

6.3.2 With pivoting:

Pivoting adds some work:

- Partial pivoting: $\sum_{k=1}^{n-1}(n-k) \sim n^2/2$ comparisons, no extra arithmetic. Some $O(n^2)$ overhead involved in having to compute $p(i)$ for the index (or swapping rows).
- Scaled partial pivoting: as above plus $n^2/2$ divisions (from dividing by the scales).
- Complete pivoting: $\sum_{k=1}^{n-1}(n-k)^2 \sim n^3/3$ comparisons.

Only complete pivoting adds a substantial amount of work compared to the $n^3/3$ multiplications of the LU factorization process. However, scaled partial pivoting does have at least a small cost (comparable to the $Ux = y$ and $Ly = b$ solves).

6.3.3 Special cases

When matrices with special structure, it can be exploited to improve the $n^3/3$ amount of work required. For example, a **tridiagonal** matrix T has one diagonal above/below the main one, with the rest zeros:

$$T = \begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 \\ a_1 & b_2 & c_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & 0 & a_{n-1} & b_n \end{bmatrix}$$

This is an example of a **sparse matrix**, which is a matrix that has mostly zero entries.[3]

To solve the system $Tx = b$, we can use Gaussian elimination. Note that only one entry (a_k) needs to be zeroed for the k-th column, and only two entries in that row have to be adjusted when reducing that row. This means that reducing the k-th column takes a constant amount of work.

The same is true of the forward/backward solves, and we end up with $O(n)$ operations - much faster than $O(n^3)$! Finding a numerical solution of, say, a system of 10 million equations would not be so bad if the system were tridiagonal. On the other hand, a full system (with arbitrary elements) would take an astronomical amount of time.

Note: It is best to write a function dedicated to solving tridiagonal systems instead of using a general-purpose Gaussian elimination code. Such a function can take better advatange of the structure - for instance, we can store the entries in a $3 \times n$ array instead of an $n \times n$ matrix, and avoid having to set up the inner for loops.

[3] Efficient solution of sparse linear systems is crucial in many applications (e.g. partial differential equations, machine learning). Much of the modern research in numerical linear algebra is devoted to dealing with such systems.