MATH96023/MATH97032/MATH97140 - Computational Linear Algebra

Edition 2020.0

Colin J. Cotter

Jul 13, 2020

CONTENTS

CHAPTER

ONE

PRELIMINARIES

In this preliminary section we revise a few key linear algebra concepts that will be used in the rest of the course, emphasising the column space of matrices. We will quote some standard results that should be found in an undergraduate linear algebra course.

1.1 Matrices, vectors and matrix-vector multiplication

We will consider the multiplication of a vector

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad x_i \in \mathbb{C},\ i = 1, 2, \ldots, n, \text{ i.e. } x \in \mathbb{C}^n,$$

by a matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix},$$

i.e. $A \in \mathbb{C}^{m \times n}$. A has m rows and n columns so that the product

$$b = Ax$$

produces $b \in \mathbb{C}^m$, defined by

$$b_i = \sum_{j=1}^{n} a_{ij} x_j,\ i = 1, 2, \ldots, m.$$

In this course it is important to consider the general case where $m \neq n$, which has many applications in data analysis, curve fitting etc. We will usually state generalities in this course for vectors over the field $\mathbb{C}$, noting where things specialise to $\mathbb{R}$.

We can quickly check that the map $x \to Ax$ given by matrix multiplication is a linear map from $\mathbb{C}^n \to \mathbb{C}^m$, since it is straightforward to check from the definition that

$$A(\alpha x + y) = \alpha Ax + Ay,$$

for all $x, y \in \mathbb{C}^n$ and $\alpha \in \mathbb{C}$. (Exercise: show this for yourself.)

It is very useful to interpret matrix-vector multiplication as a linear combination of the columns of A with coefficients taken from the entries of x. If we write A in terms of the columns,

$$A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix},$$

where

$$a_i \in \mathbb{C}^m,\ i = 1, 2, \dots, n,$$

then

$$b = \sum_{j=1}^{n} x_j a_j,$$

i.e. a linear combination of the columns of A as described above.

We can extend this idea to matrix-matrix multiplication. Taking $A \in \mathbb{C}^{m\times l}$, $C \in \mathbb{C}^{l\times n}$, $B \in \mathbb{C}^{m\times n}$, with $B = AC$, then the components of B are given by

$$b_{ij} = \sum_{k=1}^{l} a_{ik} c_{kj}, \quad 1 \le i \le m,\ 1 \le j \le n.$$

Writing $b_j \in \mathbb{C}^m$ as the jth column of B, for $1 \le j \le n$, and c_j as the jth column of C, we see that

$$b_j = Ac_j.$$

This means that the jth column of B is the matrix-vector product of A with the jth column of C. This kind of "column thinking" is very useful in understanding computational linear algebra algorithms.

An important example is the outer product of two vectors, $u \in \mathbb{C}^m$ and $v \in \mathbb{C}^n$. Here it is useful to see these vectors as matrices with one column, i.e. $u \in \mathbb{C}^{m\times 1}$ and $v \in \mathbb{C}^{n\times 1}$. The outer product is $uv^T \in \mathbb{C}^{m\times n}$. The columns of v^T are just single numbers (i.e. vectors of length 1), so viewing this as a matrix multiplication we see

$$uv^T = \begin{pmatrix} uv_1 & uv_2 & \dots & uv_n \end{pmatrix},$$

which means that all the columns of uv^T are multiples of u. We will see in the next section that this matrix has rank 1.

1.2 Range, nullspace and rank

In this section we'll quickly rattle through some definitions and results.

Definition 1 (Range) *The range of A, range(A), is the set of vectors that can be expressed as Ax for some x.*

The next theorem follows as a result of the column space interpretation of matrix-vector multiplication.

Theorem 2 *range(A) is the vector space spanned by the columns of A.*

Definition 3 (Nullspace) *The nullspace null(A) of A (or kernel) is the set of vectors x satisfying $Ax = 0$, i.e.*

$$\mathit{null}(A) = \{x \in \mathbb{C}^n : Ax = 0\}.$$

Definition 4 (Rank) *The rank rank(A) of A is the dimension of the column space of A.*

If

$$A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix},$$

the column space of A is span$(a_1, a_2, \dots, a_n)$.

Definition 5 *An $m \times n$ matrix A is full rank if it has maximum possible rank i.e. rank equal to $\min(m, n)$.*

If $m \geq n$ then A must have n linearly independent columns to be full rank. The next theorem is then a consequence of the column space interpretation of matrix-vector multiplication.

Theorem 6 *An $m \times n$ matrix A is full rank if and only if it maps no two distinct vectors to the same vector.*

Definition 7 *A matrix A is called nonsingular, or invertible, if it is a square matrix ($m = n$) of full rank.*

1.3 Invertibility and inverses

This means that an invertible matrix has columns that form a basis for $\mathbb{C}^m$. Given the canonical basis vectors defined by

$$e_j = \begin{pmatrix} 0 \\ \dots \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{pmatrix},$$

i.e. e_j has all entries zero except for the jth entry which is 1, we can write

$$e_j = \sum_{k=1}^{m} z_{ik} a_k, \quad 1 \leq j \leq m.$$

In other words,

$$\begin{aligned} I = \begin{pmatrix} e_1 & e_2 & \dots & e_m \end{pmatrix} \\ = ZA. \end{aligned}$$

We call Z a (left) inverse of A. (Exercises: show that Z is the unique left inverse of A, and show that Z is also the unique right inverse of A, satisfying $I = AZ$.) We write $Z = A^{-1}$.

The first four parts of the next theorem are a consequence of what we have so far, and we shall quote the rest (see a linear algebra course).

Theorem 8 *Let $A \in \mathbb{C}^{m\times m}$. Then the following are equivalent.*

1. *A has an inverse.*
2. *rank$(A) = m$.*
3. *range$(A) = \mathbb{C}^m$.*
4. *null$(A) = \{0\}$.*
5. *0 is not an eigenvalue of A.*
6. *0 is not a singular value of A.*
7. *The determinant $\det(A) \neq 0$.*

Finding the inverse of a matrix can be seen as a change of basis. Considering the equation $Ax = b$, we have $x = A^{-1}b$ for invertible A. We have seen already that b can be written as

$$b = \sum_{j=1}^{m} x_j a_j.$$

Since the columns of A span $\mathbb{C}^m$, the entries of x thus provide the unique expansion of b in the columns of A which form a basis. Hence, whilst the entries of b give basis coefficients for b in the canonical basis $(e_1, e_2, \ldots, e_m)$, the entries of x give basis coefficients for b in the basis given by the columns of A.

1.4 Orthogonal vectors and orthogonal matrices

Definition 9 (Adjoint) *The adjoint (or Hermitian conjugate) of $A \in \mathbb{C}^{m\times n}$ is a matrix $A^* \in \mathbb{C}^{n\times m}$ (sometimes written $A^\dagger$ or A'), with*

$$a^*_{ij} = \bar{a_{ji}},$$

where the bar denotes the complex conjugate of a complex number. If $A^* = A$ then we say that A is Hermitian.

For real matrices, $A^* = A^T$. If $A = A^T$, then we say that the matrix is symmetric.

The following identity is very important when dealing with adjoints.

Theorem 10 *For matrices A, B with compatible dimensions (so that they can be multiplied),*

$$(AB)^* = B^* A^*.$$

1.5 Inner products and orthogonality

The inner product is a critical tool in computational linear algebra.

Definition 11 (Inner product) *Let $x, y \in \mathbb{C}^m$. Then the inner product of x and y is*

$$x^* y = \sum_{i=1}^{m} \bar{x}_i y_i.$$

(Exercise: check that the inner product is bilinear, i.e. linear in both of the arguments.)

We will frequently use the natural norm derived from the inner product to define size of vectors.

Definition 12 (2-Norm) *Let $x \in \mathbb{C}^m$. Then the 2-norm of x is*

$$\|x\| = \sqrt{\sum_{i=1}^{m} x_i^2} = \sqrt{x^* x}.$$

Orthogonality will emerge as an early key concept in this course.

Definition 13 (Orthogonal vectors) *Let $x, y \in \mathbb{C}^m$. The two vectors are orthogonal if $x^* y = 0$.*

Similarly, let X, Y be two sets of vectors. The two sets are orthogonal if

$$x^* y = 0 \, \forall x \in X, \, y \in Y.$$

A set S of vectors is itself orthogonal if

$$x^* y = 0 \, \forall x, y \in S.$$

We say that S is orthonormal if we also have $\|x\| = 1$ for all $x \in S$.

1.6 Orthogonal components of a vector

Let $S = \{q_1, q_2, \ldots, q_n\}$ be an orthonormal set of vectors in $\mathbb{C}^m$, and take another arbitrary vector $v \in \mathbb{C}^m$. Now take

$$r = v - (q_1^* v) q_1 - (q_2^* v) q_2 - \ldots (q_n^* v) q_n.$$

Then, we can check that r is orthogonal to S, by calculating for each $1 \leq i \leq n$,

$$q_i^* r = q_i^* v - (q_1^* v)(q_i^* q_1) - \ldots (q_n^* v)(q_i^* q_n)$$

= q_i^*v - q_i^*v = 0,

since $q_i^* q_j = 0$ if $i \neq j$, and 1 if $i = j$. Thus,

$$v = r + \sum_{i=1}^{n} (q_i^* v) q_i = r + \sum_{i=1}^{n} \underbrace{(q_i q_i^*)}_{\text{rank-1 matrix}} v.$$

If S is a basis for $\mathbb{C}^m$, then $n = m$ and $r = 0$, and we have

$$v = \sum_{i=1}^{m} (q_i q_i^*) v.$$

1.7 Unitary matrices

Definition 14 (Unitary matrices) *A matrix $Q \in \mathbb{C}^{m \times m}$ is unitary if $Q^* = Q^{-1}$.*

For real matrices, a matrix Q is orthogonal if $Q^T = Q^{-1}$.

Theorem 15 *The columns of a unitary matrix Q are orthonormal.*

Proof 16 *We have $I = Q^*Q$. Then using the column space interpretation of matrix-matrix multiplication,*

$$e_j = Q^* q_j,$$

where q_j is the jth column of Q. Taking row i of e_j, we have

$$\delta_{ij} = q_i^* q_j, \text{ where } \delta_{ij} = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{otherwise} \end{cases}.$$

Extending a theme from earlier, we can interpret $Q^* = Q^{-1}$ as representing a change of orthogonal basis. If $Qx = b$, then $x = Q^*b$ contains the coefficients of b expanded in the basis given by the orthonormal columns of Q.

1.8 Vector norms

Various vector norms are useful to measure the size of a vector. In computational linear algebra we need them for quantifying errors etc.

Definition 17 (Norms) *A norm is a function $\|\cdot\| : \mathbb{C}^m \to \mathbb{R}$, such that*

1. $\|x\| \geq 0$, *and* $\|x\| = 0 \implies x = 0$.
2. $\|x + y\| \leq \|x\| + \|y\|$ *(triangle inequality).*
3. $\|\alpha x\| = |\alpha|\|x\|$ *for all $x \in \mathbb{C}^m$ and $\alpha \in \mathbb{C}$.*

We have already seen the 2-norm, or Euclidean norm, which is part of a larger class of norms called p-norms, with

$$\|x\|_p = \left(\sum_{i=1}^{m} |x_i|^p \right)^{1/p},$$

for real 'p>0'. We will also consider weighted norms

$$\|x\|_{W,p} = \|Wx\|_p,$$

where W is a matrix.

1.9 Projectors and projections

Definition 18 (Projector) *A projector P is a square matrix that satisfies $P^2 = P$.*

If $v \in$ range(P), then there exists x such that $Pv = x$. Then,

$$Pv = P(Px) = P^2x = Px = v,$$

and hence multiplying by P does not change v.

Now suppose that $Pv \neq v$ (so that `vnotin mbox{range}(P)). Then,

$$P(Pv - v) = P^2v - Pv = Pv - Pv = 0,$$

which means that $Pv - v$ is the nullspace of P. We have

$$Pv - v = -(I - P)v.$$

Definition 19 (Complementary projector) *Let P be a projector. Then we call $I - P$ the complementary projector.*

To see that $I - P$ is also a projector, we just calculate,

$$(I - P)^2 = I^2 - 2P + P^2 = I - 2P + P = I - P.$$

If $Pu = 0$, then $(I - P)u = u$.

In other words, the nullspace of P is contained in the range of $I - P$.

On the other hand, if v is in the range of $I - P$, then

there exists some w such that

$$v = (I - P)w = w - Pw.$$

We have

$$Pv = P(w - Pw) = Pw - P^2w = Pw - Pw = 0.$$

Hence, the range of $I - P$ is contained in the nullspace of P. Combining these two results we see that the range of $I - P$ is equal to the nullspace of P. Since P is the complementary projector to $I - P$, we can repeat the same argument to show that the range of P is equal to the nullspace of $I - P$.

We see that a projector P separates $\mathbb{C}^m$ into two subspaces, the nullspace of P and the range of P. In fact the converse is also true: given two subspaces S_1 and S_2 of $\mathbb{C}^m$ with $S_1 \cap S_2 = \{0\}$, then there exists a projector P whose range is S_1 and whose nullspace is S_2.

Now we introduce orthogonality into the concept of projectors.

Definition 20 (Orthogonal projector) *P is an orthogonal projector if*

$$(Pv)^*(Pv - v) = 0, \forall v \in \mathbb{C}^m.$$

In this case, P separates the space into two orthogonal subspaces.

1.10 Constructing orthogonal projectors from sets of orthonormal vectors

Let $\{q_1, \ldots, q_n\}$ be an orthonormal set of vectors in $\mathbb{C}^m$. We write

$$\hat{Q} = \begin{pmatrix} q_1 & q_2 & \cdots & q_n \end{pmatrix}.$$

Previously we showed that for any $v \in \mathbb{C}^m$, we have

$$v = \underbrace{r}_{\text{Orthogonal to column space of } \hat{Q}} + \underbrace{\sum_{i=1}^{n} (q_i q_i^*) v}_{\text{in the column space of } \hat{Q}}.$$

Hence, the map

$$v \mapsto Pv = \underbrace{\sum_{i=1}^{n} (q_i q_i^*)}_{=P} v,$$

is an orthogonal projector. In fact, P has very simple form.

Theorem 21 *The orthogonal projector P takes the form*

$$P = \hat{Q}\hat{Q}^*.$$

Proof 22 *From the change of basis interpretation of multiplication by $\hat{Q}^*$, the entries in $\hat{Q}^* v$ gives coefficients of the projection of v onto the column space of $\hat{Q}$ when expanded using the columns as a basis. Then, multiplication by $\hat{Q}$ gives the projection of v expanded again in the canonical basis. Hence, multiplication by $\hat{Q}\hat{Q}^*$ gives exactly the same result as multiplication by the formula for P above.*

This means that $\hat{Q}\hat{Q}^*$ is an orthogonal projection onto the range of $\hat{Q}$. The complementary projector is $P_\perp = I - \hat{Q}\hat{Q}^*$ is an orthogonal projection onto the nullspace of $\hat{Q}$.

An important special case is when $\hat{Q}$ has just one column, and then

$$P = q_1 q_1^*,\ P_\perp = I - q_1 q_1^*.$$

We notice that $P^* = (\hat{Q}\hat{Q}^*) = \hat{Q}\hat{Q}^* = P$. In fact the following is true.

Theorem 23 *$P = P^*$ if and only if Q is an orthogonal projector.*

CHAPTER

TWO

QR FACTORISATION

A common theme in computational linear algebra is transformations of matrices and algorithms to implement them. A transformation is only useful if it can be computed efficiently and sufficiently free of pollution from truncation errors (either due to finishing an iterative algorithm early, or due to round-off errors). A particularly powerful and insightful transformation is the QR factorisation. In this section we will introduce the QR factorisation and some good and bad algorithms to compute it.

2.1 What is the QR factorisation?

We start with another definition.

Definition 24 (Upper triangular matrix) *An $m \times n$ upper triangular matrix R has coefficients satisfying $r_{ij} = 0$ when $i \geq j$.*

It is called upper triangular because the nonzero rows form a triangle on and above the main diagonal of R.

Now we can describe the QR factorisation.

Definition 25 (QR factorisation) *A QR factorisation of an $m \times n$ matrix A consists of an $m \times m$ unitary matrix Q and an $m \times n$ upper triangular matrix R such that $A = QR$.*

The QR factorisation is a key tool in analysis of datasets, and polynomial fitting. It is also at the core of one of the most widely used algorithms for finding eigenvalues of matrices. We shall discuss all of this later during this course.

When $m > n$, R must have all zero rows after the n block of R consisting of the first n rows, which we call $\hat{R}$. Similarly, in the matrix vector product QR, all columns of Q beyond the n, so it makes sense to only work with the first n columns of Q, which we call $\hat{Q}$. We then have the reduced QR factorisation, $\hat{Q}\hat{R}$.

In the rest of this section we will examine some algorithms for computing the QR factorisation, before discussing the application to least squares problems. We will start with a bad algorithm, before moving on to some better ones.

2.2 QR factorisation by classical Gram-Schmidt algorithm

The classical Gram-Schmidt algorithm for QR factorisation is motivated by the column space interpretation of the matrix-matrix multiplication $A = QR$, namely that the jth column a_j of A is a linear combination of the orthonormal columns of Q, with the coefficients given by the jth column r_j of R.

The first column of R only has a non-zero entry in the first row, so the first column of Q must be proportional to A, but normalised (i.e. rescaled to have length 1). The scaling factor is this first row of the first column of R. The second column of R has only non-zero entries in the first two rows, so the second column of A must be writeable as a linear combination of the first two columns of Q. Hence, the second column of Q must by the second column of A with the first column of Q projected out, and then normalised. The first row of the second column of R is then the coefficient for this projection, and the second row is the normalisation scaling factor. The third row of Q is then the third row of A with the first two columns of Q projected out, and so on.

Hence, finding a QR factorisation is equivalent to finding an orthonormal spanning set for the columns of A, where the span of the first j elements of the spanning set and of the first j columns of A is the same, for $j = 1, \ldots, n$.

Hence we have to find R coefficients such that

$$\begin{aligned} q_1 &= \frac{a_1}{r_{11}}, \\ q_2 &= \frac{a_2 - r_{12}q_1}{r_{22}} \\ &\vdots \\ q_n &= \frac{q_n - \sum_{i=1}^{n-1} r_{in}q_i}{r_{nn}}, \end{aligned}$$

with $(q_1, q_2, \ldots, q_n)$ an orthonormal set. The non-diagonal entries of R are found by inner products, i.e.,

$$r_{ij} = q_i^* a_j, \; i > j,$$

and the diagonal entries are chosen so that $\|q_i\| = 1$, for $i = 1, 2, \ldots, n$, i.e.

$$|r_{jj}| = \left\| a_j - \sum_{i=1}^{j-1} r_{ij}q_i \right\|.$$

Note that this absolute value does leave a degree of nonuniqueness in the definition of R. It is standard to choose the diagonal entries to be real and non-negative.

We now present the classical Gram-Schmidt algorithm as pseudo-code.

- FOR $j = 1$ TO n
  - $v_j \leftarrow a_j$
  - FOR $i = 1$ TO $j - 1$
    - $r_{ij} \leftarrow q_i^* a_j$
    - $v_j \leftarrow v_j - r_{ij}q_i$
  - END FOR
  - $r_{jj} \leftarrow \|v_j\|_2$
  - $q_j \leftarrow v_j / r_{jj}$
- END FOR

(Remember that Python doesn't have END FOR statements, but instead uses indentation to terminate code blocks. We'll write END statements for code blocks in pseudo-code in these notes.)

2.3 Projector interpretation of Gram-Schmidt

At each step of the Gram-Schmidt algorithm, a projector is applied to a column of A. We have

$$\begin{aligned} q_1 &= \frac{P_1 a_1}{\|P_1 a_1\|}, \\ q_2 &= \frac{P_2 a_2}{\|P_2 a_2\|}, \\ &\vdots \\ q_n &= \frac{P_n a_n}{\|P_n a_n\|}, \end{aligned}$$

where P_j are orthogonal projectors that project out the first $j-1$ columns $(q_1, \ldots, q_{j-1})$ (P_1 is the identity as this set is empty when $j = 1$). The orthogonal projector onto the first $j-1$ columns is $\hat{Q}_{j-1}\hat{Q}_{j-1}^*$, where

$$\hat{Q}_{j-1} = \begin{pmatrix} q_1 & q_2 & \ldots & q_{j-1} \end{pmatrix}.$$

Hence, P_j is the complementary projector, $P_j = I - \hat{Q}_{j-1}\hat{Q}_{j-1}^*$.

2.4 Modified Gram-Schmidt

There is a big problem with the classical Gram-Schmidt algorithm. It is unstable, which means that when it is implemented in inexact arithmetic on a computer, round-off error unacceptably pollutes the entries of Q and R, and the algorithm is not useable in practice. What happens is that the columns of Q are not quite orthogonal, and this loss of orthogonality spoils everything. We will discuss stability later in the course, but right now we will just discuss the fix for the classical Gram-Schmidt algorithm, which is based upon the projector interpretation which we just discussed.

To reorganise Gram-Schmidt to avoid instability, we decompose P_j into a sequence of $j-1$ projectors of rank $m-1$, that each project out one column of Q, i.e.

$$P_j = P_{\perp q_{j-1}} \ldots P_{\perp q_2} P_{\perp q_1},$$

where

$$P_{\perp q_j} = I - q_j q_j^*.$$

Then,

$$v_j = P_j a_j = P_{\perp q_{j-1}} \ldots P_{\perp q_2} P_{\perp q_1} a_j.$$

Here we notice that we must apply $P_{\perp q_1}$ to all but one columns of A, and $P_{\perp q_2}$ to all but two columns of A, $P_{\perp q_3}$ to all but three columns of A, and so on.

By doing this, we gradually transform A to a unitary matrix, as follows.

$$\begin{aligned} A = &\begin{pmatrix} a_1 & a_2 & a_3 & \ldots & a_n \end{pmatrix} \\ &\begin{pmatrix} q_1 & v_2^1 & v_3^1 & \ldots & v_n^1 \end{pmatrix} \\ \to &\begin{pmatrix} q_1 & q_2 & v_3^2 & \ldots & v_n^2 \end{pmatrix} \\ \ldots \to &\begin{pmatrix} q_1 & q_2 & q_3 & \ldots & q_n \end{pmatrix}. \end{aligned}$$

Then it is just a matter of keeping a record of the coefficients of the projections and normalisation scaling factors and storing them in R.

This process is mathematically equivalent to the classical Gram-Schmidt algorithm, but the arithmetic operations happen in a different order, in a way that turns out to reduce accumulation of round-off errors.

We now present this modified Gram-Schmidt algorithm as pseudo-code.

- FOR $i = 1$ TO n
  - $v_i \leftarrow a_i$
- END FOR
- FOR $i = 1$ TO n
  - $r_{ii} \leftarrow \|v_i\|_2$
  - $q_i = v_i / r_{ii}$
    - FOR $j = i + 1$ TO n

- $r_{ij} \leftarrow q_i^* a_j$
- $v_j \leftarrow v_j - r_{ij} q_i$

* END FOR

- END FOR

This algorithm can be applied "in place", overwriting the entries in A with the v s and eventually the q s.

2.5 Modified Gram-Schmidt as triangular orthogonalisation

This iterative transformation process can be written as right-multiplication by an upper triangular matrix. For example, at the first iteration,

$$\underbrace{\begin{pmatrix} v_1^0 & v_2^0 & \dots & v_n^0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \frac{1}{r_{11}} & -\frac{r_{12}}{r_{11}} & \dots & \dots & -\frac{r_{11}}{r_{11}} \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \dots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}}_{A_1} = \underbrace{\begin{pmatrix} q_1 & v_2^1 & \dots & v_n^1 \end{pmatrix}}_{A_1}.$$

To understand this equation, we can use the column space interpretation of matrix-matrix multiplication. The columns of A_1 are linear combinations of the columns of A with coefficients given by the columns of R_1. Hence, q_1 only depends on v_1^0, scaled to have length 1, and v_i^1 is a linear combination of (v_1^0, v_i^0) such that v_i^1 is orthogonal to q_1, for $1 < i \leq n$.

Similarly, the second iteration may be written as

$$\underbrace{\begin{pmatrix} v_1^1 & v_2^1 & \dots & v_n^1 \end{pmatrix}}_{A_1} \underbrace{\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & r_{22} & -\frac{r_{23}}{r_{22}} & \dots & -\frac{r_{2n}}{r_{nn}} \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \dots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}}_{R_2} = \underbrace{\begin{pmatrix} q_1 & q_2 & v_3^2 \dots & v_n^2 \end{pmatrix}}_{A_2}.$$

It should become clear that each transformation from A_i to A_{i+1} takes place by right multiplication by an upper triangular matrix R_{i+1}, which is an identity matrix plus entries in row i. By combining these transformations together, we obtain

$$A \underbrace{R_1 R_2 \dots R_n}_{\hat{R}^{-1}} = \hat{Q}.$$

Since upper triangular matrices form a group, the product of the R_i matrices is upper triangular. Further, all the R_i matrices have non-zero determinant, so the product is invertible, and we can write this as $\hat{R}^{-1}$. Right multiplication by $\hat{R}$ produces the usual reduced QR factorisation. We say that modified Gram-Schmidt implements triangular orthogonalisation: the transformation of A to an orthogonal matrix by right multiplication of upper triangular matrices.

This is a powerful way to view the modified Gram-Schmidt process from the point of view of understanding and analysis, but of course we do not form the matrices R_i explicitly (we just follow the pseudo-code given above).

2.6 Householder triangulation

This view of the modified Gram-Schmidt process as triangular orthogonalisation gives an idea to build an alternative algorithm. Instead of right multiplying by upper triangular matrices to transform A to $\hat{Q}$, we can consider left multiplying by unitary matrices to transform A to R,

$$\underbrace{Q_n \dots Q_2 Q_1}_{=Q^*} A = R.$$

Multiplying unitary matrices produces unitary matrices, so we obtain $A = QR$ as a full factorisation of A.

To do this, we need to work on the columns of A, from left to right, transforming them so that each column has zeros below the diagonal. These unitary transformations need to be designed so that they don't spoil the structure created in previous columns. The easiest way to ensure this is construct a unitary matrix Q_k with an identity matrix as the $(k-1) \times (k-1)$ submatrix,

$$Q_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & F \end{pmatrix}.$$

This means that multiplication by Q_k won't change the first $k-1$ rows, leaving the previous work to remove zeros below the diagonal undisturbed. For Q_k to be unitary and to transform all below diagonal entries in column k to zero, we need the $(n-k+1) \times (n-k+1)$ submatrix F to also be unitary, since

$$Q_k^* = \begin{pmatrix} I_{k-1} & 0 \\ 0 & F^* \end{pmatrix}, Q_k^{-1} = \begin{pmatrix} I_{k-1} & 0 \\ 0 & F^{-1} \end{pmatrix}.$$

We write the k of A_k as

$$v_k^k = \begin{pmatrix} \hat{v}_k^k \\ x \end{pmatrix},$$

where $\hat{v}_k^k$ contains the first $k-1$ entries of v_k^k. The column gets transformed according to

$$Q_k v_k^k = \begin{pmatrix} \hat{v}_k^k \\ Fx \end{pmatrix}.$$

and our goal is that Fx is zero, except for the first entry (which becomes the diagonal entry of $Q_k v_k^k$). Since F is unitary, we must have $\|Fx\| = \|x\|$, so we choose to have

$$Fx = \pm \|x\| e_1,$$

where we shall consider the sign later. Here we have chosen for Fx to be real. We could alternatively pick another value of the same magnitude but different complex argument (but that would lead to a slightly different QR factorisation).

We can achieve this by using a Householder reflector for F, which is a unitary transformation that does precisely what we need. Geometrically, the idea is that we consider a line joining x and $Fx = \|x\|e_1$, which points in the direction $v = \|x\|e_1 - x$. We can transform x to Fx by a reflection in the hyperplane H that is orthogonal to v. Since reflections are norm preserving, F must be unitary. Applying the projector P given by

$$Px = \left(I - \frac{vv^*}{v^*v}\right) x,$$

does half the job, producing a vector in H. To do a reflection we need to go twice as far,

$$Fx = \left(I - 2\frac{vv^*}{v^*v}\right) x.$$

We can check that this does what we want,

$$\begin{aligned} Fx &= \left(I - 2\frac{vv^*}{v^*v}\right) x, \\ &= x - 2\frac{(\pm\|x\|e_1 - x)}{\|\pm\|x\|e_1 - x\|^2}(\pm\|x\|e_1 - x)^* x, \\ &= x - 2\frac{(\pm\|x\|e_1 - x)}{\|\pm\|x\|e_1 - x\|^2}\|x\|(\pm x_1 - \|x\|), \\ &= x + (\pm\|x\|e_1 - x) = \pm\|x\|e_1, \end{aligned}$$

as required, having checked that

$$\|\|x\| \pm e_1 - x\|^2 = -2\|x\|(\pm x_1 - \|x\|).$$

We can also check that F is unitary. First we check that F is Hermitian,

$$\begin{aligned} \left(I - 2\frac{vv^*}{v^*v}\right)^* &= I - 2\frac{(vv^*)^*}{v^*v}, \\ &= I - 2\frac{(v^*)^* v^*}{v^*v}, \\ &= I - 2\frac{vv^*}{v^*v} = F. \end{aligned}$$

Now we use this to show that F is unitary,

$$\begin{aligned} F^*F &= \left(I - 2\frac{vv^*}{v^*v}\right)\left(I - 2\frac{vv^*}{v^*v}\right) \\ &= I - 4\frac{vv^*}{v^*v}\frac{vv^*}{v^*v} + 4\frac{vv^*}{v^*v}\frac{vv^*}{v^*v} = I, \end{aligned}$$

so $F^* = F^{-1}$. In summary, we have constructed a unitary matrix Q_k that transforms the entries below the diagonal of the kth column of A_k to zero, and leaves the previous $k-1$ columns alone.

Earlier, we mentioned that there is a choice of sign in v. This choice gives us the opportunity to improve the numerical stability of the algorithm. To avoid unnecessary numerical round off, we choose the sign that makes v furthest from x, i.e.

$$v = \text{sign}(x_1)\|x\|e_1 + x.$$

(Exercise, show that this choice of sign achieves this.)

We are now in a position to describe the algorithm in pseudo-code. Here it is described an "in-place" algorithm, where the successive transformations to the columns of A are implemented as replacements of the values in A. This means that we can allocate memory on the computer for A which is eventually replaced with the values for R. To present the algorithm, we will use the "slice" notation to describe submatrices of A, with $A_{k:l,r:s}$ being the submatrix of A consisting of the rows from k to l and columns from r to s. Be careful with Python implementations, where the numbering of rows and columns is from 0, and not 1.

- FOR $k = 1$ TO n
  - $x = A_{k:m,k}$
  - $v_k \leftarrow \text{sign}(x_1)\|x\|_2 e_1 + x$
  - $v_k \leftarrow v_k/\|v_k\|$
  - $A_{k:m,k:n} \leftarrow A_{k:m,k:m} - 2v_k(v_k^* A_{k:m,k:n})$.
- END FOR

Note that we have not explicitly formed the matrix Q or the product matrices Q_i. In some applications, such as solving least squares problems, we don't explicitly need Q, just the matrix-vector product Q^*b with some vector b. To compute this product, we can just apply the same operations to b that are applied to the columns of A. This can be expressed in the following pseudo-code, working "in place" in the storage of b.

- FOR $k = 1$ TO n
  - $b_{k:m} \leftarrow b_{k:m} - 2v_k(v_k^* b_{k:m})$
- END FOR

We call this procedure "implicit multiplication".

If we really need Q, we can get it by matrix-vector products with each element of the canonical basis $(e_1, e_2, \ldots, e_n)$. This means that first we need to compute a matrix-vector product Qx with a vector x. This is just done by applying the Householder reflections in reverse, since

$$Q = (Q_n \ldots Q_2 Q_1)^* = Q_1 Q_2 \ldots Q_n,$$

having made use of the fact that the Householder reflections are Hermitian. This can be expressed in the following pseudo-code.

- FOR $k = n$ TO 1 (DOWNWARDS)
  - $x_{k:m} \leftarrow x_{k:m} - 2v_k(v_k^* x_{k:m})$
- END FOR

Note that this requires to record all of the history of the v vectors, whilst the Q^* application algorithm above can be interlaced with the steps of the Householder algorithm, using the v values as they are needed and throwing them away. Then we can compute Q via

$$Q = \begin{pmatrix} Qe_1 & Qe_2 & \ldots & Qe_n \end{pmatrix},$$

with each column using the Q application algorithm described above.

2.7 Application: Least squares problems

Least square problems are relevant in data fitting problems, optimisation and control, and are also a crucial ingredient of modern massively parallel linear system solver algorithms. They are a way of solving "long thin" matrix vector problems $Ax = b$ where we want to obtain $x \in \mathbb{C}^m$ from $b \in \mathbb{C}^n$ with A an $n \times m$ matrix. Often the problem does not have a solution as it is overdetermined for $n > m$. Instead we just seek x that minimises the 2-norm of the residual $r = b - Ax$, i.e. x is the minimiser of

$$min_x \|Ax - b\|^2.$$

This residual will not be zero in general, when b is not in the range of A. The nearest point in the range of A to b is Pb, where P is the orthogonal projector onto the range of A. From Theorem 21, we know that $P = \hat{Q}\hat{Q}^*$, where $\hat{Q}$ from the reduced QR factorisation has the same column space as A (but with orthogonal columns).

Then, we just have to solve

$$Ax = Pb,$$

which is now solveable since Pb is in the column space of A (and hence can be written as a linear combination of the columns of A i.e. as a matrix-vector product Ax for some unknown x).

Now we have the reduced QR factorisation of A, and we can write

$$\hat{Q}\hat{R}x = \hat{Q}\hat{Q}^*b.$$

Left multiplication by $\hat{Q}^*$ then gives

$$\hat{R}x = \hat{Q}^*b.$$

This is an upper triangular system that can be solved efficiently using back-substitution (which we shall come to later.)

CHAPTER

THREE

ANALYSING ALGORITHMS

In the previous section we saw three algorithms to compute the QR factorisation of a matrix. They have a beautiful mathematical structure based on orthogonal projectors. But are they useful? To answer this we need to know:

1. Is one faster than others?
2. Is one more sensitive than others to small perturbations due to early truncation of the algorithm or due to round-off errors?

In this course we will characterise answers to the first question by operation count (acknowledging that this is an incomplete evaluation of speed), and answers to the second question by analysing stability.

In this section we will discuss both of these questions by introducing some general concepts but also looking at the examples of the QR algorithms that we have seen so far.

3.1 Operation count

Operation count is one aspect of evaluating how long algorithms take. Here we just note that this is not the only aspect, since transferring data between different levels of memory on chips can be a serious (and often dominant) consideration, even more so when we consider algorithms that make use of large numbers of processors running in parallel. However, operation count is what we shall focus on here.

In this course, a floating point operation (FLOP) will be any arithmetic unary or binary operation acting on single numbers (such as $+$, $-$, $\times$, $\div$, $\sqrt{}$). Of course, in reality, these different operations have different relative costs, and codes can be made more efficient by blending multiplications and additions (fused multiply-adds) for example. Here we shall simply apologise to computer scientists in the class, and proceed with this interpretation, since we are just making relative comparisons between schemes. We shall also concentrate on asymptotic results in the limit of large n and/or m.

3.2 Operation count for modified Gram-Schmidt

We shall discuss operation counts through the example of the modified Gram-Schmidt algorithm. We shall find that the operation count is $\sim mn^2$ to compute the QR factorisation, where the $\sim$ symbol means

$$\lim_{m,n\to\infty} \frac{N_{\text{FLOPS}}}{2mn^2} = 1.$$

To get this result, we return to the pseudocode for the modified Gram-Schmidt algorithm, and concentrate on the operations that are happening inside the inner j loop. Inside that loop there are two operations,

1. $r_{ij} \leftarrow q_i^* v_i$. This is the inner product of two vectors in $\mathbb{R}^m$, which requires m multiplications and $m-1$ additions, so we count $2m-1$ FLOPS per inner iteration.
2. $v_j \leftarrow v_j - r_{ij} q_i$. This requires m multiplications and m subtractions, so we count $2m$ FLOPS per inner iteration.

At each iteration we require a combined operation count of $\sim 4m$ FLOPS. There are n outer iterations over i, and $n-i-1$ inner iterations over j, which we can estimate by approximating the sum as an integral,

$$N_{\text{FLOPS}} \sim \sum_{i=1}^{n} \sum_{j=i+1}^{n} 4m \sim 4m \sum_{i=1}^{n} i \int_{1}^{n} x\,dx \sim 4m\frac{n^2}{2} = 2mn^2,$$

as suggested above.

3.3 Operation count for Householder

In the Householder algorithm, the computation is dominated by the transformation

$$A_{k:m,k:n} \leftarrow A_{k:m,k:n} - \underbrace{2v_k \underbrace{(v_k^* A_{k:m,k:n})}_{1}}_{2},$$

which must be done for each 'k' iteration. To evaluate the part marked 1 requires $n-k$ inner products of vectors in $\mathbb{C}^{m-k}$, at a total cost of $\sim 2(n-k)(m-k)$ (we already examined inner products in the previous example). To evaluate the part marked 2 then requires the outer product of two vectors in $\mathbb{C}^{m-k}$ and $\mathbb{C}^{n-k}$ respectively, at a total cost of $(m-k)(n-k)$ FLOPs. Finally two $(k-m)\times(n-k)$ matrices are substracted, at cost $(k-m)(n-k)$. Putting all this together gives $\sim 4(n-k)(m-k)$ FLOPs per k iteration.

Now we have to sum this over k, so the total operation count is

$$\begin{aligned} 4\sum_{k=1}^{n}(n-k)(m-k) &= 4\sum_{k=1}^{n}(nm - k(n+m) + k^2) \\ &\sim 4n^2m - 4(n+m)\frac{n^2}{2} + 4\frac{n^3}{3} = 2mn^2 - \frac{2n^3}{3}. \end{aligned}$$

3.4 Matrix norms for discussing stability

In the rest of this section we will discuss another important aspect of analysing computational linear algebra algorithms, stability. To do this we need to introduce some norms for matrices in addition to the norms for vectors that we discussed in Section 1.

If we ignore their multiplication properties, matrices in $\mathbb{C}^{m\times n}$ can be added and scalar multiplied, hence we can view them as a vector space, in which we can define norms, just as we did for vectors.

One type of norm arises from simply treating the matrix entries as entries of a vector and evaluating the 2-norm.

Definition 26 (Frobenius norm) *The Frobenius norm is the matrix version of the 2-norm, defined as*

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} A_{ij}^2}.$$

(Exercise: show that $\|AB\|_F \le \|A\|_F \|B\|_F$.)

Another type of norm measures the maximum amount of stretching the matrix can cause when multiplying a vector.

Definition 27 (Induced matrix norm) *Given an $m \times n$ matrix A and any chosen vector norms $\|\cdot\|_{(n)}$ and $\|\cdot\|_{(m)}$ on $\mathbb{C}^n$ and $\mathbb{C}^m$, respectively, the induced norm on A is*

$$\|A\|_{(m,n)} = \sup_{x \in \mathbb{C}^n, x \neq 0} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}}.$$

Directly from the definition we can show

$$\frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} \leq \sup_{x \in \mathbb{C}^n, x \neq 0} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)} = \|A\|_{(m,n)}},$$

and hence $\|Ax\| \leq \|A\|\|x\|$ whenever we use an induced matrix norm.

3.5 Norm inequalities

Often it is difficult to find exact values for norms, so we compute upper bounds using inequalities instead. Here are a few useful inequalities.

Definition 28 (H"older inequality) *Let $x, y \in \mathbb{C}^m$, and $p, q \in \mathbb{R}+$ such that $\frac{1}{p} + 1q = 1$. Then*

$$|x^* y| \leq \|x\|_p \|y\|_q.$$

In the case $p = q = 2$ this becomes the Cauchy-Schwartz inequality.

Definition 29 (Cauchy-Schwartz inequality) *Let $x, y \in \mathbb{C}^m$. Then*

$$|x^* y| \leq \|x\|_2 \|y\|_2.$$

For example, we can use this to bound the operator norm of the outer product $A = uv^*$ of two vectors.

$$\|Ax\|_2 = \|uv^* x\|_2 = \|u(v^* x)\|_2 = |v^* x| \|u\|_2 \leq \|u\|_2 \|v\|_2 \|x\|_2,$$

so $\|A\|_2 \leq \|u\|_2 \|v\|_2$.

We can also compute bounds for $\|AB\|_2$.

Theorem 30 *Let $A \in \mathbb{C}^{l \times m}$, $B \in \mathbb{C}^{m \times n}$. Then*

$$\|AB|_{(l,n)} \leq \|A\|_{(l,m)} \|B\|_{(m,n)}.$$

Proof 31

$$\|ABx\|_{(l)} \leq \|A\|_{(l,m)} \|Bx\|_{(m)} \leq \|A\|_{(l,m)} \|B\|_{(m,n)} \|x\|_{(n)},$$

so

$$\|AB\|_{(l,n)} = \sup_{x \neq 0} \frac{\|ABx\|_{(l)}}{\|x\|_{(n)}} \leq \|A\|_{(l,m)} \|B\|_{(m,n)},$$

as required.

3.6 Condition number

The key tool to understanding numerical stability of computational linear algebra algorithms is the condition number. The condition number is a very general concept that measures the behaviour of a mathematical problem under perturbations. Here we think of a mathematical problem as a function $f : X \to Y$, where X and Y are normed vector spaces (further generalisations are possible). It is often the case that f has different properties under perturbation for different values of $x \in X$.

Definition 32 (Well conditioned and ill conditioned.) *We say that a problem is well conditioned (at x) if small changes in x lead to small changes in $f(x)$. We say that a problem is ill conditioned if small changes in x lead to large changes in $f(x)$.*

These changes are measured by the condition number.

Definition 33 (Absolute condition number.) *Let δx be a perturbation so that $x \mapsto x + \delta x$. The corresponding change in $f(x)$ is $\delta f(x)$,*

$$\delta f(x) = f(x + \delta x) - f(x).$$

The absolute condition number of f at x is

$$\hat{\kappa} = \sup_{\delta x \neq 0} \frac{\|\delta f\|}{\|\delta x\|},$$

i.e. the maximum that f can change relative to the size of the perturbation δx.

It is easier to consider linearised perturbations, defining a Jacobian matrix $J(x)$ such that

$$J(x)\delta x = \lim_{\epsilon \to 0} \frac{f(x + \epsilon \delta x) - f(x)}{\epsilon}$$

and then the linear absolute condition number is

$$\hat{\kappa} = \sup_{\delta x \neq 0} \frac{\|J(x)\delta x\|}{\|\delta x\|} = \|J(x)\|,$$

which is the operator norm of $J(x)$.

This definition could be improved by measuring this change relative to the size of f itself.

Definition 34 (Relative condition number.) *The relative condition number of a problem f measures the changes δx and δf relative to the sizes of x and f.*

$$\kappa = \sup_{\delta \neq 0} \frac{\|\delta f\|/\|f\|}{\|\delta x\|/\|x\|}.$$

The linear relative condition number is

$$\kappa = \frac{\|J\|/\|f\|}{\|x\|} = \frac{\|J\|\|x\|}{\|f\|}.$$

Since we use floating point numbers on computers, it makes more sense to consider relative condition numbers in computational linear algebra, and from here on we will always use them whenever we mention condition numbers. If κ is small ($1 - 100$, say) then we say that a problem is well conditioned. If κ is large ($> 10^6$, say), then we say that a problem is ill conditioned.

As a first example, consider the problem of finding the square root, $f : x \mapsto \sqrt{x}$, a one dimensional problem. In this case, $J = x^{1/2}/2$. The (linear) condition number is

$$\kappa = \frac{|x^{-1/2}/2||x|}{|x^{1/2}|} = 1/2.$$

Hence, the problem is well-conditioned.

As a second example, consider the problem of finding the roots of a polynomial, given its coefficients. Specifically, we consider the polynomial $x^2 - 2x + 1 = (x-1)^2$, which has two roots equal to 1. Here we consider the change in roots relative to the coefficient of x^0 (which is 1). Making a small perturbation to the polynomial, $x^2 - 2x + 0.9999 = (x - 0.99)(x - 1.01)$, so a relative change of 10^{-4} gives a relative change of 10^{-2} in the roots. Using the general formula

$$r = 1 \pm \sqrt{1-c} = 1 \pm \sqrt{\delta c} \implies \delta r = \pm\sqrt{\delta c},$$

where r returns the two roots with perturbations δr and c is the coefficient of x^0 with perturbatino δc. is the perturbation to the coefficient of x^0 (so 1 becomes $1 + \delta c$). The (nonlinear) condition number is then the sup over $\delta c \neq 0$ of

$$\frac{|\delta r|/|r|}{|\delta c|/|c|} = \frac{|\delta r|}{|\delta c|} = \frac{|\delta c|^{1/2}}{|\delta c|} = |\delta c|^{-1/2} \to \infty \text{ as } \delta c \to 0,$$

so the condition number is unbounded and the problem is catastrophically ill conditioned. For an even more vivid example, see the conditioning of the roots of the Wilkinson polynomial.

3.7 Conditioning of linear algebra computations

We now look at the condition number of problems from linear algebra. The first problem we examine is the problem of matrix-vector multiplication, i.e. for a fixed matrix $A \in \mathbb{C}^{m\times n}$, the problem is to find Ax given x. The problem is linear, with $J = A$, so the condition number is

$$\kappa = \frac{\|A\|\|x\|}{\|Ax\|}.$$

When A is non singular, we can write $x = A^{-1}Ax$, and

$$\|x\| = \|A^{-1}Ax\| \leq \|A^{-1}\|\|Ax\|,$$

so

$$\kappa \leq \frac{\|A\|\|A^{-1}\|\|Ax\|}{\|Ax\|} = \|A\|\|A^{-1}\|.$$

We call this upper bound the condition number $\kappa(A)$ of the matrix A.

The next problem we consider is the condition number of solving $Ax = b$, with b fixed but considering perturbations to A. So, we have $f : A \mapsto x$. The condition number of this problem measures how small changes δA to A translate to changes δx to x. The perturbed problem is

$$(A + \delta A)(x + \delta x) = b,$$

which simplifies (using $Ax = b$) to

$$\delta A(x + \delta x) + A\delta x = 0,$$

which is independent of b. If we are considering the linear condition number, we can drop the nonlinear term, and we get

$$\delta Ax + A\delta x = 0, \implies \delta x = -A^{-1}\delta Ax,$$

from which we may compute the bound

$$\|\delta x\| \leq \|A^{-1}\|\|\delta A\|\|x\|.$$

Then, we can compute the condition number

$$\kappa = \sup_{\|\delta A\|\neq 0} \frac{\|\delta x\|/\|x\|}{\|\delta A\|/\|A\|}. \leq \sup_{\|\delta A\|\neq 0} \frac{\|A^{-1}\|\|\delta A\|\|x\|/\|x\|}{\|\delta A\|/\|A\|}. = \|A^{-1}\|\|A\| = \kappa(A),$$

having used the bound for δx. Hence the bound on the condition number for this problem is the condition number of A.

3.8 Floating point numbers and arithmetic

Floating point number systems on computers use a discrete and finite representation of the real numbers. One of the first things we can deduce from this fact is that there exists a largest and a smallest positive number. In "double precision", the standard floating point number format for scientific computing these days, the largest number is $N_{\max} \approx 1.79 \times 10^{308}$, and the smallest number is $N_{\min} \approx 2.23 \times 10^{-308}$. The second thing that we can deduce is that there must be gaps between adjacent numbers in the number system. In the double precision format, the interval $[1, 2]$ is subdivided as $(1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, 1 + 3 \times 2^{-52}, \ldots, 2)$. The next interval $[2, 4]$ is subdivided as $(2, 2 + 2^{-51}, 2 + 2 \times 2^{-51}, \ldots, 4)$. In general, the interval $[2^j, 2^{j+1}]$ is subdivided by multiplying the set subdividing $[1, 2]$ by 2^j. In this representation, the gaps between numbers scale with the number size. We call this set of numbers the (double precision) floating point numbers $\mathbb{F} \subset \mathbb{R}$.

A key aspect of a floating point number system is "machine epsilon" (ε), which measures the larges relative distance between two numbers. Considering the description above, we see that ε is the the distance between 1 and the adjacent number, i.e.

$$\varepsilon = 2^{-53} \approx 1.11 \times 10^{-16}.$$

ε defines the accuracy with which arbitrary real numbers (within the range of the maximum magnitude above) can be approximated in $\mathbb{F}$.

$$\forall x \in \mathbb{R}, \exists x' \in \mathbb{F} \text{ such that } |x - x'| \leq \varepsilon|x|.$$

Definition 35 (Floating point rounding function) *We define $f_L : \mathbb{R} \to \mathbb{F}$ as the function that rounds $x \in \mathbb{R}$ to the nearest floating point number.*

The following axiom is just a formal presentation of the properties of floating point numbers that we discussed below.

Definition 36 (Floating point axiom I)

$$\forall x \in \mathbb{R},\ \exists \epsilon' \text{ with } |\epsilon'| \leq \varepsilon,$$
$$\text{such that } f_L(x) = x(1+\epsilon').$$

The arithmetic operations $+, -, \times, \div$ on $\mathbb{R}$ have analogous operations $\oplus, \ominus, \otimes$, etc. In general, binary operators $\odot$ (as a general symbol representing the floating point version of a real arithmetic operator $\cdot$ which could be any of the above) are constructed such that

$$x \odot y = f_L(x \cdot y),$$

for $x, y \in \mathbb{F}$, with $\cdot$ being one of $+, -, \times, \div$.

Definition 37 (Floating point axiom II)

$$\forall x, y \in \mathbb{F}, \exists \epsilon' \text{ with } |\epsilon'| \leq \varepsilon, \text{ such that}$$
$$x \odot y = (x \cdot y)(1+\epsilon').$$

3.9 Stability

Stability describes the perturbation behaviour of a numerical algorithm when used to solve a problem on a computer. Now we have two problems $f : X \to Y$ (the original problem implemented in the real numbers), and $\tilde{f} : X \to Y$ (the modified problem where floating point numbers are used at each step).

Given a problem f (such as computing the QR factorisation), we are given:

1. A floating point system $\mathbb{F}$,
2. An algorithm for computing f,
3. A floating point implementation $\tilde{f}$ for f.

Then the chosen $x \in X$ is rounded to $x' = f_L(x)$, and supplied to the floating point implementation of the algorithm to obtain $\tilde{f}(x) \in Y$.

Now we want to compare $f(x)$ with $\tilde{f}(x)$. We can measure the absolute error

$$\|\tilde{f}(x) - f(x)\|,$$

or the relative error (taking into account the size of f),

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|}.$$

An aspiration (but an unrealistic one) would be to aim for an algorithm to accurate to machine precision, i.e.

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \mathcal{O}(\varepsilon),$$

by which we mean that $\exists C > 0$ such that

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \leq C\varepsilon,$$

for sufficiently small ε.

Definition 38 (Stability) *An algorithm $\tilde{f}$ for f is stable if for each $x \in X$,*

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = \mathcal{O}(\varepsilon),$$

for all $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\varepsilon).$$

We say that a stable algorithm gives nearly the right answer to nearly the right question.

Definition 39 (Backward stability) *An algorithm $\tilde{f}$ for f is backward stable if for each $x \in X$, $\exists \tilde{x}$ such that*

$$\tilde{f}(x) = f(\tilde{x}), \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\varepsilon).$$

A backward stable algorithm gives exactly the right answer to nearly the right answer. The following result shows what accuracy we can expect from a backward stable algorithm, which involves the condition number of f.

Theorem 40 (Accuracy of a backward stable algorithm) *Suppose that a backward stable algorithm is applied to solve problem $f : X \to Y$ with condition number κ using a floating point number system satisfying the floating point axioms I and II. Then the relative error satisfies*

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \mathcal{O}(\kappa(x)\epsilon).$$

Proof 41 *Since $\tilde{f}$ is backward stable, we have $\tilde{x}$ with $\tilde{f}(x) = f(\tilde{x})$ and $\|\tilde{x} - x\|/\|x\| = \mathcal{O}(\varepsilon)$ as above. Then,*

$$\begin{aligned} \frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} &= \frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|}, \\ &= \underbrace{\frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|}\frac{\|x\|}{\|\tilde{x} - x\|}}_{=\kappa}\underbrace{\frac{\|\tilde{x} - x\|}{\|x\|}}_{=\mathcal{O}(\epsilon)}, \end{aligned}$$

as required.

This type of calculation is known as backward error analysis, originally introduced by Jim Wilkinson to analyse the accuracy of eigenvalue calculations using the PILOT ACE, one of the early computers build at the National Physical Laboratory in the late 1940s and early 1950s. In backward error analysis we investigate the accuracy via conditioning and stability. This is usually much easier than forward analysis, where one would simply try to keep a running tally of errors committed during each step of the algorithm.

3.10 Backward stability of the Householder algorithm

We now consider the example of the problem of finding the QR factorisation of a matrix A, implemented in floating point arithmetic using the Householder method. The input is A, and the exact output is Q, R, whilst the floating point algorithm output is $\tilde{Q}, \tilde{R}$. Here, we consider $\tilde{Q}$ as the exact unitary matrix produced by composing Householder rotations made by the floating point vectors $\tilde{v}_k$ that approximate the v_k vectors in the exact arithmetic Householder algorithm.

For this problem, backwards stability means that there exists a perturbed input $A + \delta A$, with $\|\delta A\|/\|A\| = \mathcal{O}(\varepsilon)$, such that $\tilde{Q}, \tilde{R}$ are exact solutions to the problem, i.e. $\tilde{Q}\tilde{R} = A + \delta A$. This means that there is very small backward error,

$$\frac{\|A - \tilde{Q}\tilde{R}\|}{\|A\|} = \mathcal{O}(\varepsilon).$$

It turns out that the Householder method is backwards stable.

Theorem 42 *Let the QR factorisation be computed for A using a floating point implementation of the Householder algorithm. This factorisation is backwards stable, i.e. the result $\tilde{Q}\tilde{R}$ satisfy*

$$\tilde{Q}\tilde{R} = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\varepsilon).$$

Proof 43 *See the textbook by Trefethen and Bau, Lecture 16.*

3.11 Backward stability for solving a linear system using QR

The QR factorisation provides a method for solving systems of equations $Ax = b$ for x given b, where A is an invertible matrix. Substituting $A = QR$ and then left-multiplying by Q^* gives

$$Rx = Q^*b = y.$$

The solution of this equation is $x = R^{-1}y$, but if there is one message to take home from this course, it is that you should *never* form the inverse of a matrix. It is especially disasterous to use Kramer's rule, which has an operation count scaling like $\mathcal{O}(m!)$ and is numerically unstable. There are some better algorithms for finding the inverse of a matrix if you really need it, but in almost every situation it is better to *solve* a matrix system rather than forming the inverse of the matrix and multiplying it. It is particularly easy to solve an equation formed from an upper triangular matrix. Written in components, this equation is

$$\begin{aligned}
R_{11}x_1 + R_{12}x_2 + \ldots + R_{1(m-1)}x_{m-1} + R_{1m}x_m &= y_1, \\
0x_1 + R_{22}x_2 + \ldots + R_{2(m-1)}x_{m-1} + R_{2m}x_m &= y_2, \\
&\vdots \\
0x_1 + 0x_2 + \ldots + R_{(m-1)(m-1)}x_{m-1} + R_{(m-1)m}x_m &= y_{m-1}, \\
0x_1 + 0x_2 + \ldots + 0x_{m-1} + R_{mm}x_m &= y_m.
\end{aligned}$$

The last equation yields x_m directly by dividing by R_{mm}, then we can use this value to directly compute x_{m-1}. This is repeated for all of the entries of x from m down to 1. This procedure is called back substitution, which we summarise in the following pseudo-code.

- $x_m \leftarrow y_m / R_{mm}$

- FOR $i = m - 1$ TO 1 (BACKWARDS)
  - $x_i \leftarrow (y_i - \sum_{k=i+1}^{m} R_{ik}x_k)/R_{ii}$

In each iteration, there are $m - i - 1$ multiplications and subtractions plus a division, so the total operation count is $\sim m^2$ FLOPs.

In comparison, the least bad way to form the inverse Z of R is to write $RZ = I$. Then, the `k`th column of this equation is

$$Rz_k = e_k,$$

where z_k is the kth column of Z. Solving for each column independently using back substitution leads to an operation count of $\sim m^3$ FLOPs, much slower than applying back substitution directly to b. Hopefully this should convince you to always seek an alternative to forming the inverse of a matrix.

There are then three steps to solving $Ax = b$ using QR factorisation.

1. Find the QR factorisation of A (here we shall use the Householder algorithm).
2. Set $y = Q^*b$ (using the implicit multiplication algorithm).
3. Solve $Rx = y$ (using back substitution).

So our f here is the solution of $Ax = b$ given b and A, and our $\tilde{f}$ is the composition of the three algorithms above. Now we ask: "Is this composition of algorithms stable?"

We already know that the Householder algorithm is stable, and a floating point implementation produces $\tilde{Q}, \tilde{R}$ such that $\tilde{Q}\tilde{R} = A + \delta A$ with $\|\delta A\|/\|A\| = \mathcal{O}(\varepsilon)$. It turns out that the implicit multiplication algorithm is also backwards stable, for similar reasons (as it is applying the same Householder reflections). This means that given $\tilde{Q}$ (we have already perturbed Q when forming it using Householder) and b, the floating point implementation gives $\tilde{y}$ which is not exactly equal to $\tilde{Q}^*b$, but instead satisfies

$$\tilde{y} = (\tilde{Q} + \delta Q)^* b \implies (\tilde{Q} + \delta Q)\tilde{y} = b,$$

for some perturbation δQ with $\|\delta Q\| = \mathcal{O}(\varepsilon)$ (note that $\|Q\| = 1$ because it is unitary). Note that here, we are treating b as fixed and considering the backwards stability under perturbations to $\tilde{Q}$.

Finally, it can be shown (see Lecture 17 of Trefethen and Bau for a proof) that the backward substitution algorithm is backward stable. This means that given $\tilde{y}$ and $\tilde{R}$, the floating point implementation of backward substitution produces $\tilde{x}$ such that

$$(\tilde{R} + \delta\tilde{R})\tilde{x} = \tilde{y},$$

for some upper triangular perturbation such that $\|\delta\tilde{R}\|/\|\tilde{R}\| = \mathcal{O}(\varepsilon)$.

Using the individual backward stability of these three algorithms, we show the following result.

Theorem 44 *The QR algorithm to solve $Ax = b$ is backward stable, producing a solution $\tilde{x}$ such that*

$$(A + \Delta A)\tilde{x} = b,$$

for some $\|\Delta A\|/\|A\| = \mathcal{O}(\varepsilon)$.

Proof 45 *From backward stability for the calculation of Q^*b, we have*

$$
\begin{aligned}
b &= (\tilde{Q} + \delta Q)\tilde{y}, \\
&= (\tilde{Q} + \delta Q)(\tilde{R} + \delta R)x,
\end{aligned}
$$

having substituted the backward stability formula for back substitution in the second line. Multiplying out the brackets and using backward stability for the Householder method gives

$$
\begin{aligned}
b &= (\tilde{Q}\tilde{R} + (\delta Q)\tilde{R} + \tilde{Q}\delta R + (\delta Q)\delta R)\tilde{x}, \\
&= \underbrace{(A + \delta A + (\delta Q)\tilde{R} + \tilde{Q}\delta R + (\delta Q)\delta R)}_{=\Delta A}\tilde{x}.
\end{aligned}
$$

This defines ΔA and it remains to estimate each of these terms. We immediately have $\|\delta A\| = \mathcal{O}(\varepsilon)$ from backward stability of the Householder method.

Next we estimate the second term. Using $A + \delta A = \tilde{Q}\tilde{R}$, we have

$$
\tilde{R} = \tilde{Q}^*(A + \delta A),
$$

we have

$$
\frac{\|\tilde{R}\|}{\|A\|} \leq \|\tilde{Q}^*\| \frac{\|A + \delta A\|}{\|A\|} = \mathcal{O}(1), \text{ as } \varepsilon \to 0.
$$

Then we have

$$
\frac{\|(\delta Q)\tilde{R}\|}{\|A\|} \leq \|\delta Q\| \frac{\|\tilde{R}\|}{\|A\|} = \mathcal{O}(\varepsilon).
$$

To estimate the third term, we have

$$
\frac{\|\tilde{Q}\delta R\|}{\|A\|} \leq \frac{\|\delta R\|}{\|A\|} \underbrace{\|\tilde{Q}\|}_{=1} = \underbrace{\frac{\|\delta R\|}{\|\tilde{R}\|}}_{\mathcal{O}(\varepsilon)} \underbrace{\frac{\|\tilde{R}\|}{\|A\|}}_{\mathcal{O}(1)} = \mathcal{O}(\varepsilon).
$$

Finally, the fourth term has size

$$
\frac{\|\delta Q \delta R\|}{\|A\|} \leq \underbrace{\|\delta Q\|}_{\mathcal{O}(\varepsilon)} \underbrace{\frac{\|\delta R\|}{\|\tilde{R}\|}}_{\mathcal{O}(\varepsilon)} \underbrace{\frac{\|\tilde{R}\|}{\|A\|\}}}_{\mathcal{O}(1)} = \mathcal{O}(\epsilon^2),
$$

hence $\|\delta A\|/\|A\| = \mathcal{O}(\varepsilon)$.

Corollary 46 *When solving $Ax = b$ using the QR factorisation procedure above, the floating point implementation produces an approximate solution $\tilde{x}$ with*

$$
\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\kappa(A)\varepsilon).
$$

Proof 47 *From Theorem 40, using the backward stability that we just derived, we know that*

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\kappa\varepsilon),$$

where κ is the condition number of the problem of solving $Ax = b$, which we have shown is bounded from above by $\kappa(A)$.

CHAPTER

FOUR

LU DECOMPOSITION

In this section we look at the some other algorithms for solving the equation $Ax = b$ when A is invertible. On the one hand the QR factorisation has great stability properties. On the other, it can be beaten by other methods for speed when there is particular structure to exploit (such as lots of zeros in the matrix). In this section, we explore the the family of methods that go right back to the technique of Gaussian elimination, that you will have been familiar with since secondary school.

4.1 An algorithm for LU decomposition

The computational way to view Gaussian elimination is through the LU decomposition of an invertible matrix, $A = LU$, where L is lower triangular ($l_{ij} = 0$ for $j < i$) and U is upper triangular ($u_{ij} = 0$ for $j > i$). Here we use the symbol U instead of R to emphasise that we are looking as square matrices. The process of obtaining the LU decomposition is very similar to the Householder algorithm, in that we repeatedly left multiply A by matrices to transform below-diagonal entries in each column to zero, working from the first to the last column. The difference is that whilst the Householder algorithm left multiplies with unitary matrices, here, we left multiply with lower triangular matrices.

The first step puts zeros below the first entry in the first column.

$$A_1 = L_1 A = \begin{pmatrix} u_1 & v_2^1 & v_2^1 & \dots & v_n^1 \end{pmatrix},$$
$$u_1 = \begin{pmatrix} u_{11} \\ 0 \\ \dots \\ 0 \end{pmatrix}.$$

Then, the next step puts zeros below the second entry in the second column.

$$A_2 = L_2 L_1 A = \begin{pmatrix} u_1 & u_2 & v_2^2 & \dots & v_n^2 \end{pmatrix},$$
$$u_2 = \begin{pmatrix} u_{12} \\ u_{22} \\ 0 \\ \dots \\ 0 \end{pmatrix}.$$

After repeated left multiplications we have

$$A_n = \underbrace{L_n \dots L_2 L_1} A = U.$$

This process of transforming A to U is called Gaussian elimination.

If we assume (we will show this later) that all these lower triangular matrices are invertible, we can define

$$L = (L_n \dots L_2 L_1)^{-1} = L_1^{-1} L_2^{-1} \dots L_n^{-1},$$
so that
$$L^{-1} = L_n \dots L_2 L_1.$$

Then we have $L^{-1}A = U$, i.e. $A = LU$.

So, we need to find lower triangular matrices L_k that do not change the first $k-1$ rows, and transforms the kth column 'x_k' of A_k as follows.

$$Lx_k = L \begin{pmatrix} x_{1k} \\ \vdots \\ x_{kk} \\ x_{k+1,k} \\ \vdots \\ x_{m,k} \end{pmatrix} = \begin{pmatrix} x_{1k} \\ \vdots \\ x_{kk} \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

As before with the Householder method, we see that we need the top-left $k \times k$ submatrix of L to be the identity (so that it doesn't change the first k rows). We claim that the following matrix transforms x_k to the required form.

$$L_k = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & \dots & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 & \dots & \dots & \vdots & 0 \\ 0 & 0 & 1 & \dots & 0 & \dots & \dots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 & \dots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & -l_{k+1,k} & 1 & \dots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & -l_{k+2,k} & 0 & \ddots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & 0 & \dots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -l_{m,k} & 0 & \dots & \dots & 1 \end{pmatrix},$$

$$l_k = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ l_{k+1,k} = x_{k+1,k}/x_{kk} \\ l_{k+2,k} = x_{k+2,k}/x_{kk} \\ \vdots \\ l_{m,k} = x_{m,k}/x_{kk} \end{pmatrix}.$$

This has the identity block as required, and we can verify that L_k puts zeros in the entries of x_k below the diagonal by first writing $L_k = I - l_k e_k^*$. Then,

$$L_k x_k = I - l_k e_k^* = x_k - l_k \underbrace{(e_k^* x_k)}_{=x_{kk}},$$

which subtracts off the below diagonal entries as required. Indeed, multiplication by L_k implements the row operations that are performed to transform below diagonal elements of A_k to zero during Gaussian elimination.

The determinant of a lower triangular matrix is equal to the trace (product of diagonal entries), so $\det(L_k) = 1$, and consequently L_k is invertible, enabling us to define L^{-1} as above. To form L we need to multiply the inverses of all the L_k matrices together, also as above. To do this, we first note that $l_k^* e_k = 0$ (because l_k is zero in the only entry that e_k is nonzero). Then we claim that $L_k^{-1} = I + l_k e_k^*$, which we verify as follows.

$$\begin{aligned}(I + l_k e_k^*)L_k = (I + l_k e_k^*)(I - l_k e_k^*) = I + l_k e_k^* - l_k e_k^* + (l_k e_k^*)(l_k e_k *) \\ = I + \underbrace{l_k (e_k^* l_k) e_k *}_{=0} = I,\end{aligned}$$

as required. Similarly if we multiply the inverse lower triangular matrices from two consecutive iterations, we get

$$\begin{aligned}L_k^{-1} L_{k+1}^{-1} = (I + l_k e_k^*)(I + l_{k+1} e_{k+1}^*) = I + l_k e_k^* + l_{k+1} e_{k+1}^* + l_k \underbrace{(e_k^* l_{k+1})}_{=0} e_{k+1}^* \\ = I + l_k e_k^* + l_{k+1} e_{k+1}^*,\end{aligned}$$

since $e_k^* l_{k+1} = 0$ too, as l_{k+1} is zero in the only place where e_k is nonzero. If we iterate this argument, we get

$$L = I + \sum_{i=1}^{m-1} l_i e_i^*.$$

Hence, the k is the same as the k, i.e.,

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & \dots & \dots & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 & \dots & \dots & \vdots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 & \dots & \dots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 & \dots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & l_{k+1,k} & 1 & \dots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & l_{k+2,k} & l_{k+2,k+1} & \ddots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & l_{m-1,k+1} & \dots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & l_{m,k} & l_{m,k+1} & \dots & \dots & 1 \end{pmatrix}.$$

In summary, we can compute entries of L during the Gaussian elimination process of transforming A to U.

So, what's the advantage of writing $A = LU$? Well, we can define $y = Ux$. Then, we can solve $Ax = b$ in two steps, first solving $Ly = b$ for y, and then solving $Ux = y$ for x. The latter equation is an upper triangular system that can be solved by the back substitution algorithm we introduced for QR factorisation. The former equation can be solved by forward substitution, derived in an analogous way, written in pseudo-code as follows.

- $x_1 \leftarrow b_1 / L_{11}$
- FOR $i = 2$ TO m
  - $x_i \leftarrow (b_i - \sum_{k=1}^{i} L_{ik} x_k) / L_{ii}$

Forward substitution has an operation count that is identical to back substitution, by symmetry, i.e. $\mathcal{O}(m^2)$. In contrast, we shall see shortly that the Gaussian elimination process has an operation count $\mathcal{O}(m^3)$. Hence, it is much cheaper to solve a linear system with a given LU factorisation than it is to form L and U in the first place. We can take advantage of this in the situation where we have to solve a whole sequence of linear systems $Ax = b_i$, $i = 1, 2, \ldots, K$, with the same matrix A but different right hand side vectors. In this case we can pay the cost of forming LU once, and then use forward and back substitution to cheaply solve each system. This is particularly useful when we need to repeatedly solve systems as part of larger iterative algorithms, such as time integration methods or Monte Carlo methods.

The Gaussian elimination algorithm is written in pseudo-code as follows. We start by copying A into U, and setting L to an identity matrix, and then work "in-place" i.e. replacing values of U and L until they are completed. In a computer implementation, this memory should be preallocated and then written to instead of making copies (which carries overheads).

- $U \leftarrow A$
- $L \leftarrow I$
- FOR $k = 1$ TO $m - 1$
  - for $j = k + 1$ TO m
    - $l_{jk} \leftarrow u_{jk}/u_{kk}$
    - $u_{j,k:m} \leftarrow u_{j,k:m} - l_{jk}u_{k,k:m}$
  - END FOR
- END FOR

To do an operation count for this algorithm, we note that the dominating operation is the update of U inside the j loop. This requires $m - k + 1$ multiplications and subtractions, and is iterated $m - k$ times in the j loop, and this whole thing is iterated from $j = k + 1$ to m. Hence the asymptotic operation count is

$$\begin{aligned} N_{\text{FLOPs}} &= \sum_{k=1}^{m-1} \sum_{j=k+1}^{m} 2(m-k+1), \\ &= \sum_{k=1}^{m-1} 2(m-k+1) \underbrace{\sum_{j=k+1}^{m} 1}_{=m-k} \\ &= \sum_{k=1}^{m-1} 2m^2 - 4mk + 2k^2 \\ &\sim 2m^3 - 4\frac{m^3}{2} + \frac{2m^3}{3} = \frac{2m^3}{3}. \end{aligned}$$

4.2 Pivoting

Gaussian elimination will fail if a zero appears on the diagonal, i.e. we get $x_{kk} = 0$ (since then we can't divide by it). Similarly, Gaussian elimination will amplify rounding errors if x_{kk} is very small, because a small error becomes large after dividing by x_{kk}. The solution is to reorder the rows in A_k so that that x_{kk} has maximum magnitude. This would seem to mess up the LU factorisation procedure. However, it is not as bad as it looks, as we will now see.

The main tool is the permutation matrix.

Definition 48 (Permutation matrix) *An $m \times m$ permutation matrix has precisely one entry equal to 1 in every row and column, and zero elsewhere.*

A compact way to store a permutation matrix P as a size m vector p, where p_i is equal to the number of the column containing the 1 entry in row i of P. Multiplying a vector x by a permutation matrix P simply rearranges the entries in x, with $(Px)_i = x_{p_i}$.

During Gaussian elimination, say that we are at stage k, and $(A_k)_{kk}$ is not the largest magnitude entry in the k. We reorder the rows to fix this, and this is what we call *pivoting*. Mathematically this reordering is equivalent to multiplication by a permutation matrix P_k. Then we continue the Gaussian elimination procedure by left multiplying by L_k, placing zeros below the diagonal in column k of $P_k A_k$.

In fact, P_k is a very specific type of permutation matrix, that only swaps two rows. Therefore, $P_k^{-1} = P_k$, even though this is not true for general permutation matrices.

We can pivot at every stage of the procedure, producing a permutation matrix P_k, $k = 1, \ldots, m-1$ (if no pivoting is necessary at a given stage, then we just take the identity matrix as the pivoting matrix for that stage). Then, we end up with the result of Gaussian elimination with pivoting,

$$L_{m-1}P_{m-1}\ldots L_2P_2L_1P_1 = U.$$

This looks like it has totally messed up the LU factorisation, because LP is not lower triangular for general lower triangular matrix L and permutation matrix P. However, we can save the situation, by trying to swap all the permutation matrices to the right of all of the L matrices. This does change the L matrices, because matrix-matrix multiplication is not commutative. However, we shall see that it does preserve the lower triangular matrix structure.

To see how this is done, we focus on how things look after two stages of Gaussian elimination. We have

$$A_2 = L_2P_2L_1P_1 = L_2\underbrace{P_2L_1P_2}_{=L_1^{(2)}}P_2P_1 = L_2L_1^{(2)}P_2P_1,$$

having used $P_2^{-1} = P_2$. Left multiplication with P_2 exchanges row 2 with some other row j with $j > 2$. Hence, right multiplication with P_2 does the same thing but with columns instead of rows. Therefore, L_1P_2 is the same as L_1 but with column 2 exchanged with column j. Column 2 is just e_2 and column j is just e_j, so now column 2 has the 1 in row j and column j has the 1 in row 2. Then, $P_2L_1P_2$ exchanges row 2 of L_1P_2 with row j of L_1P_2. This just exchanges l_{12} with l_{1j}, and swaps the 1s in columns 2 and j back to the diagonal. In summary, $P_2L_1P_2$ is the same as L_1 but with l_{12} exchanged with l_{1j}.

Moving on to the next stage, and we have

$$A_3 = L_3P_3L_2L_1P_2P_1 = L_3\underbrace{P_3L_2P_3}_{=L_2^{(3)}}\underbrace{P_3L_1P_3}_{=L_1^{(3)}}P_3P_2P_1.$$

By similar arguments we see that $L_2^{(3)}$ is the same as L_2 but with l_{23} exchanged with l_{2j} for some (different) j, and $L_2^{(3)}$ is the same as $L_2^{(2)}$ with l_{13} exchanged with l_{1j}. After iterating this argument, we can obtain

$$\underbrace{L_{m-1}^{(m-1)}\ldots L_2^{(m-1)}L_1^{(m-1)}}_{L^{-1}}\underbrace{P_{m-1}\ldots P_2P_1}_{P} = U,$$

where we just need to keep track of the permutations in the L matrices as we go through the Gaussian elimination stages. These L matrices have the same structure as the basic LU factorisation, and hence we obtain

$$L^{-1}PA = U \implies PA = LU.$$

This is equivalent to permuting the rows of A using P and then finding the LU factorisation using the basic algorithm (except we can't implement it like that because we only decide how to build P during the Gaussian elimination process).

The LU factorisation with pivoting can be expressed in the following pseudo-code.

- $U \leftarrow A$
- $L \leftarrow I$
- $P \leftarrow I$
- FOR $k = 1$ TO $m - 1$
  - Choose $i \geq k$ to maximise $|u_{ik}|$
  - $u_{k,k:m} \leftrightarrow u_{i,k:m}$ (column swaps)
  - $l_{k,1:k-1} \leftrightarrow l_{i,1:k-1}$ (column swaps)
  - $p_{k,1:m} \leftrightarrow p_{i,1:m}$ (column swaps)
  - FOR $j = k + 1$ TO m
    - $l_{ik} \leftarrow u_{jk}/u_{kk}$
    - $u_{j,k:m} \leftarrow u_{j,k:m} - l_{jk}u_{k,k:m}$
  - END FOR
- END FOR

To solve a system $Ax = b$ given the a pivoted LU factorisation $PA = LU$, we left multiply the equation by P and use the factorisation get $LUx = Pb$. The procedure is then as before, but b must be permuted to Pb before doing the forwards and back substitutions.

4.3 Stability of LU factorisation

To characterise the stability of LU factorisation, we quote the following result.

Theorem 49 *Let $\tilde{L}$ and $\tilde{U}$ be*

4.4 Taking advantage of matrix structure

The cost of the standard Gaussian elimination algorithm to form L and U is $\mathcal{O}(m^3)$, which grows rather quickly as m increases. If there is structure in the matrix, then we can often exploit this to reduce the cost. Understanding when and how to exploit structure is a central theme in computational linear algebra. Here we will discuss some examples of structure to be exploited.

When A is a lower or upper triangular matrix then we can use forwards or back substitution, with $\mathcal{O}(m^2)$ operation count as previously discussed.

When A is a diagonal matrix, i.e. $A_{ij} = 0$ for $i \neq j$, it only has m nonzero entries, that can be stored as a vector, $(A_{11}, A_{22}, \ldots, A_{mm})$. In this case, $Ax = b$ can be solved in m operations, just by setting $x_i = b_i/A_{ii}$, for $i = 1, 2, \ldots, m$.

A generalisation of a diagonal matrix is a banded matrix, where $A_{ij} = 0$ for $i > j + p$ and for $i < j - q$. We call p the upper bandwidth of A; q is the lower bandwidth. When the matrix is banded, there are already zeros below the diagonal of A, so we know that the corresponding entries in the L_k matrices will be zero. Further, because there are zeros above the diagonal of A, these do not need to be updated when applying the row operations to those zeros.

The Gaussian elimination algorithm (without pivoting) for a banded matrix is given as pseudo-code below.

- $U \leftarrow A$

- $L \leftarrow I$
- FOR $k = 1$ TO $\min(k + p, m)$
  - $l_{jk} \leftarrow u_{jk}/u_{kk}$
  - $n \leftarrow \min(k + q, m)$
    - $u_{j,k:n} \leftarrow u_{j,k:n} - l_{jk}u_{k,k:n}$
  - END FOR
- END FOR

The operation count for this banded matrix algorithm is $\mathcal{O}(mpq)$, which is linear in m instead of cubic! Further, the resulting matrix L has lower bandwidth p and U has upper bandwidth q. This means that we can also exploit this structure in the forward and back substitution algorithms as well. For example, the forward substitution algorithm is given as pseudo-code below.

- $x_1 \leftarrow b_1/L_11$
- FOR $k = 2$ TO m
  - $j \leftarrow \max(1, k - p)$
  - $x_k \leftarrow \frac{b_k - L_{k,j:k-1}x_{j:k-1}}{L_{kk}}$
- END FOR

This has an operation count $\mathcal{O}(mp)$. The story is very similar for the back substitution.