MATH96023/MATH97032/MATH97140 - Computational Linear Algebra

Edition 2020.0

Colin J. Cotter

Aug 21, 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

Hint: A video recording of this section is available here.

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

$$b_i = \sum_{j=1}^{n} a_{ij} x_j,\ i = 1, 2, \ldots, m. \tag{1.1}$$

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 $\mathbb{R}$.

Hint: A video recording of this section is available here.

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.)

Hint: A video recording of this section is available here.

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.

Hint: A video recording of this section is available here.

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.

Hint: A video recording of this section is available here.

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.

Exercise 1 *The* `cla_utils.exercises1.basic_matvec()` *function has been left unimplemented. To finish the function, add code so that it computes the matrix-vector product $b = Ax$ from inputs A and x. In this first implementation, you should simply implement (1.1) with a double nested for loop (one for the sum over j, and one for the i elements of b). Run this script to test your code (and all the exercises from this exercise set):*

```
py.test test/test_exercises1.py
```

from the Bash command line. Make sure you commit your modifications and push them to your fork of the course repository.

Exercise 2 *The* `cla_utils.exercises1.column_matvec()` *function has been left unimplemented. To finish the function, add code so that it computes the matrix-vector product $b = Ax$ from inputs A and x. This second implementation should use the column-space formulation of matrix-vector multiplication, i.e., b is a weighted sum of the columns of A with coefficients given by the entries in x. This should be implemented with a single for loop over the entries of x. The test script* `test_exercises1.py` *will also test this function.*

Hint: It will be useful to use the Python "slice" notation, for example:

```
A[:, 3]
```

will return the 4th (since Python numbers from zero) column of A. For more information, see the Numpy documentation on slicing.

Exercise 3 *The* `cla_utils.exercises1.time_matvecs()` *function computes the execution time for these two implementations for some example matrices and compares them with the built-in Numpy matrix-vector product. Run this function and examine the output. You should observe that the basic implementation is much slower than the built-in implementation. This is because built-in Numpy operations use compiled C code that is wrapped in Python, which avoids the overheads of run-time interpretation of the Python code and manipulation of Python objects. Numpy is really useful for computational linear algebra programming because it preserves the readability and flexibility of Python (writing code that looks much more like maths, access to object-oriented programming models) whilst giving near-C speed if used appropriately. You can read more about the advantages of using Numpy here. You should also observe that the column implementation is somewhere between the speed of the basic implementation and the built-in implementation. This is because (if you did it correctly), each iteration of the for loop involves adding an entire array (a scaling of one of the columns of A) to another array (where b is being calculated). This will also use compiled C code through Numpy, removing some (but not all) of the Python overheads in the basic implementation.*

In this course, we will present algorithms in the notes that generally do not express the way that Numpy should be used to implement them. In these exercises you should consider the best way to make use of Numpy built-in operations (which will often make the code more maths-like and readable, as well as potentially faster).

1.2 Range, nullspace and rank

Hint: A video recording of this section is available here.

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

Definition 4 (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 5 *range(A) is the vector space spanned by the columns of A.*

Definition 6 (Nullspace) *The nullspace null(A) of A (or kernel) is the set of vectors x satisfying $Ax = 0$, i.e.*

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

Hint: A video recording of this section is available here.

Definition 7 (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 8 *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 9 *An $m \times n$ matrix A is full rank if and only if it maps no two distinct vectors to the same vector.*

Definition 10 *A matrix A is called nonsingular, or invertible, if it is a square matrix ($m = n$) of full rank.*

Exercise 11 *The `cla_utils.exercises1.rank2()` function has been left unimplemented. To finish the function, add code so that it computes the rank-2 matrix $A = u_1 v_1^* + u_2 v_2^*$ from $u_1, u_2 \in \mathbb{C}^m$ and $v_1, v_2 \in \mathbb{C}^n$. As you can see, the function needs to implement this rank-2 matrix by first forming two matrices B and C from the inputs, matrix-vector product $b = Ax$ from inputs A and x. The test script `test_exercises1.py` in the `test` directory will also test this function.*

To measure the rank of A, we first need to cast it from a numpy array class to a numpy matrix class, and then use the built-in rank function:

```
r = numpy.linalg.rank(numpy.matrix(A))
```

and we should find that the rank is equal to 2. Can you explain why this should be the case (use the column space interpretation of matrix-matrix multiplication)?

1.3 Invertibility and inverses

Hint: A video recording of this section is available here.

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 fifth and sixth (see a linear algebra course).

Theorem 12 *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. *The determinant $\det(A) \neq 0$.*

Hint: A video recording of this section is available here.

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.

Exercise 13 *For matrices of the form, $A = I + uv^*$, where I is the $m \times m$ identity matrix, and $u, v \in \mathbb{C}^m$, show that whenever A is invertible, the inverse is of the form $A = I + \alpha uv^*$ where $\alpha \in \mathbb{C}$, and calculate the form of α.*

The `cla_utils.exercises1.rank1pert_inv()` *function has been left unimplemented. To finish the function, add code so that it computes A^{-1} using your formula (and not any built-in matrix inversion routines). The test script* `test_exercises1.py` *in the* `test` *directory will also test this function.*

Add a function to `cla_utils.exercises1` *that measures the time to compute the inverse of A for an input matrix of size 400, and compare with the time to compute the inverse of A using the built-in inverse:*

```
numpy.linalg.inv(A)
```

What do you observe? Why do you think this is? We will examine the cost of general purpose matrix inversion algorithms later.

1.4 Adjoints and Hermitian matrices

Hint: A video recording of this section is available here.

Definition 14 (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 15 *For matrices A, B with compatible dimensions (so that they can be multiplied),*

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

Exercise 16 *Consider a matrix $A = B + iC$ where $B, C \in \mathbb{R}^{m\times m}$ and A is Hermitian. Show that $B = B^T$ and $C = -C^T$. To save memory, instead of storing values of A ($m \times m$ complex numbers to store), consider equivalently storing a real-valued $m \times m$ array $\hat{A}$ with $\hat{A}_{ij} = B_{ij}$ for $i \leq j$ and $\hat{A}_{ij} = C_{ij}$ for $i > j$.*

The `cla_utils.exercises1.ABiC()` *function has been left unimplemented. It should implement matrix vector multiplication $z = Ax$, returning the real and imaginary parts of z, given the real and imaginary parts of x as inputs, and given the real array $\hat{A}$ as above. You should implement the multiplication using real arithmetic only, with just one loop over the entries of x, using the column space interpretation of matrix-vector multiplication. The test script* `test_exercises1.py` *in the* `test` *directory will also test this function.*

Hint: You can use the Python "slice" notation, to assign into a slice of an array, for example:

```
x[3:5] = y[3:5]
```

will copy the 4th and 5th entries of y (Python numbers from zero, and the upper limit of the slice is the first index value not to use. For more information, see the Numpy documentation on slicing.

1.5 Inner products and orthogonality

Hint: A video recording of this section is available here.

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

Definition 17 (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 18 (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 19 (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

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

A set S of vectors is itself orthogonal if

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

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 $1 \leq i \leq n$,

$$\begin{aligned} q_i^* r &= q_i^* v - (q_1^* v)(q_i^* q_1) - \ldots - (q_n^* v)(q_i^* q_n) \\ &= q_i^* v - q_i^* v = 0, \end{aligned}$$

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

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

Exercise 20 *The* `cla_utils.exercises2.orthog_cpts()` *function has been left unimplemented. It should implement the above computation, returning* r *and the coefficients of the component of* v *in each orthonormal direction. The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

1.7 Unitary matrices

Definition 21 (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 22 *The columns of a unitary matrix* Q *are orthonormal.*

Proof 23 *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.

Exercise 24 *The* `cla_utils.exercises2.solveQ()` *function has been left unimplemented. Given a square unitary matrix* Q *and a vector* b *it should solve* $Qx = b$ *using information above (it is not expected to work when* Q *is not unitary or square). The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

Add a function to `cla_utils.exercises2` *that measures the time to solve* $Qx = b$ *using* `solveQ` *for an input matrix of sizes 100, 200, 400, and compare with the times to solve the equation using the general purpose solve (which uses LU factorisation, which we will discuss later):*

```
x = numpy.linalg.solve(Q, b)
```

What did you expect and was it observed?

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 25 (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 26 (Projector) *A projector P is a square matrix that satisfies $P^2 = P$.*

If $v \in \text{range}(P)$, then there exists x such that $Px = v$. 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 $v \notin \text{range}(P)$). 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 27 (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 28 (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

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

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 29 *The orthogonal projector P takes the form*

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

Proof 30 *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 31 *$P = P^*$ if and only if Q is orthogonal.*

Exercise 32 *The* `cla_utils.exercises2.orthog_proj()` *function has been left unimplemented. Given an orthonormal set $q_1, q_2, \ldots, q_n$, it should provide the orthogonal projector P. The test script* `test_exercises2.py` *in the* `test` *directory will also test this function.*

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 33 (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 34 (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, $\hat{Q}\hat{R}$.

Exercise 35 *The* `cla_utils.exercises2.orthog_space()` *function has been left unimplemented. Given a set of vectors $v_1, v_2, \ldots, v_n$ that span the subspace $U \subset \mathbb{C}^m$, the function should find an orthonormal basis for the orthogonal complement $U^\perp$ given by*

$$U^\perp = \{x \in \mathbb{C}^m : x^* v = 0, \forall v \in U\}.$$

It is expected that it will only compute this up to a tolerance. You should make use of the built in QR factorisation routine `numpy.linalg.qr()`*. The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

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.)

Exercise 36 *The* `cla_utils.exercises2.GS_classical()` *function has been left unimplemented. It should implement the classical Gram-Schmidt algorithm above, using Numpy slice notation so that only one Python for loop is used. The function should work "in place" by and then changing the values in A, without introducing additional intermediate arrays. The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

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.

Exercise 37 *The* `cla_utils.exercises2.GS_modified()` *function has been left unimplemented. It should implement the modified Gram-Schmidt algorithm above, using Numpy slice notation so that only one Python for loop is used. The function should work "in place" by making a copy of A and then changing those values, without introducing additional intermediate arrays. The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

Exercise 38 *Investigate the mutual orthogonality of the Q matrices that are produced by your classical and modified Gram-Schmidt implementations. Is there a way to test mutual orthogonality without writing a loop? Round-off typically causes problems for matrices with large condition numbers and large off-diagonal values. You could also try the opposite of what was done in* `test_GS_classical`*: instead of ensuring that all of the entries in the diagonal matrix D are $\mathcal{O}(1)$, try making some of the values small and some large. See if you can find a matrix that illustrates the differences in orthogonality between the two algorithms.*

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

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

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).

Exercise 39 *In a break from the format so far, the* `cla_utils.exercises2.GS_modified_R()` *function has been implemented. It implements the modified Gram-Schmidt algorithm in the form describe above using upper triangular matrices. This is not a good way to implement the algorithm, because of the inversion of R at the end, and the repeated multiplication by zeros in multiplying entries of the R_k matrices, which is a waste. However it is important as a conceptual tool for understanding the modified Gram-Schmidt algorithm as a triangular orthogonalisation process, and so it is good to see this in a code implementation. Study this function to check that you understand what is happening.*

However, the `cla_utils.exercises2.GS_modified_get_R()` *function has not been implemented. This function computes the R_k matrices at each step of the process. Complete this code. The test script* `test_exercises2.py` *in the* `test` *directory will also test this function.*

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 th column v_k^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 $\|Fx\| = \|x\|$. For now we shall specialise to real matrices, so we choose to have

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

where we shall consider the sign later. Complex matrices have a more general formula for Householder transformations which we shall not discuss here.

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 $Fx = \pm\|x\|e_1$, which points in the direction $v = \pm\|x\|e_1 - x$. 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 (assuming x is real)

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

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. In the case of real matrices, 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.

- 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

Exercise 40 *The* `cla_utils.exercises3.householder()` *function has been left unimplemented. It should implement the algorithm above, using only one loop over k. It should return the resulting R matrix. The test script* `test_exercises3.py` *in the* `test` *directory will test this function.*

Hint: Don't forget that Python numbers from zero, which will be important when implementing the submatrices using Numpy slice notation.

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".

Exercise 41 *Show that the implicit midpoint procedure is equivalent to computing an extended array*

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

and performing Householder on the first n rows. Transform the equation $Ax = b$ into $Rx = \hat{b}$ where $QR = A$, and find the form of $\hat{b}$, explaining how to get $\hat{b}$ from Householder applied to $\hat{A}$ above. Solving systems with upper triangular matrices is much cheaper than solving general matrix systems as we shall discuss later.

Now, say that we want to solve multiple equations

$$Ax_i = b_i, i = 1, 2, \dots, k,$$

which have the same matrix A but different right hand sides $b = b_i$, $i = 1, 2, \dots, k$. Extend this idea above to the case $k > 1$, by describing an extended $\hat{A}$ containing all the b_i vectors.

The `cla_utils.exercises3.householder_solve()` *function has been left unimplemented. It takes in a set of right hand side vectors $b_1, b_2, \dots, b_k$ and returns a set of solutions $x_1, x_2, \dots, x_k$. It should construct an extended array $\hat{A}$, and then pass it to* `cla_utils.exercises3.householder()`*. If you have not already done so, you will need to modified* `cla_utils.exercises3.householder()` *to use the* `kmax` *argument. You may make use of the built-in tridiagonal solve algorithm* `numpy.linalg.solve_triangular()` *(we shall consider tridiagonal matrix algorithms briefly later). The test script* `test_exercises3.py` *in the* `test` *directory will also test this function.*

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

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

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

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

with each column using the Q application algorithm described above.

Exercise 42 *Show that the implicit multiplication procedure applied to the columns of I produces Q^*, from which we can easily obtain Q, explaining how. Show how to implement this by applying Householder to an augmented matrix $\hat{A}$ of some appropriate form.*

The `cla_utils.exercises3.householder_qr()` *function has been left unimplemented. It takes in the $m \times n$ array A and returns Q and R. It should use the method of this exercise to compute them by forming an appropriate $\hat{A}$, calling* `cla_utils.exercises3.householder()` *and then extracting appropriate subarrays using slice notation. The test script* `test_exercises3.py` *in the* `test` *directory will also test this function.*

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 such as GMRES, which we shall encounter later in the course. They are a way of solving "long thin" matrix vector problems $Ax = b$ where we want to obtain $x \in \mathbb{C}^n$ from $b \in \mathbb{C}^m$ with A an $m \times n$ matrix. Often the problem does not have a solution as it is overdetermined for $m > n$. 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 29, 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.)

Exercise 43 *The* `cla_utils.exercises3.householder_ls()` *function has been left unimplemented. It takes in the $m \times n$ array A and a right-hand side vector b and solves the least squares problem minimising $\|Ax-b\|$ over x. It should do this by forming an appropriate augmented matrix $\hat{A}$, calling* `cla_utils.exercises3.householder()` *and extracting appropriate subarrays using slice notation, before using* `numpy.linalg.solve_triangular()` *to solve the resulting upper triangular system, before returning the solution x. The test script* `test_exercises3.py` *in the* `test` *directory will also test this function.*

Hint: You will need to do extract the appropriate submatrix to obtain the square (and invertible) reduced matrix $\hat{R}$.

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

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

Exercise 44 *Compute FLOP counts for the following operations.*

1. $\alpha = x^*y$ *for* $x, y \in \mathbb{C}^m$.
2. $y = y + ax$ *for* $x, y \in \mathbb{C}^m$, $a \in \mathbb{C}$.
3. $y = y + Ax$ *for* $x \in \mathbb{C}^n$, $y \in \mathbb{C}^m$, $A \in \mathbb{C}^{m\times n}$.
4. $C = C + AB$ *for* $A \in \mathbb{C}^{m\times r}$, $B \in \mathbb{C}^{r\times n}$, $C \in \mathbb{C}^{m\times n}$.

Exercise 45 *Suppose* $D = ABC$ *where* $A \in \mathbb{C}^{m\times n}$, $B \in \mathbb{C}^{n\times p}$, $C \in \mathbb{C}^{p\times q}$. *This can either be computed as* $D = (AB)C$ *(multiply* A *and* B *first, then* C*), or* $D = A(BC)$ *(multiply* B *and* C *first, then* A*). Compute the FLOP count for both approaches. For which values of* m, n, p, q *would the first approach be more efficient?*

Exercise 46 *Suppose* $W \in \mathbb{C}^{n\times n}$ *is defined by*

$$w_{ij} = \sum_{q=1}^{n}\sum_{p=1}^{n} x_{ip}y_{pq}z_{qj},$$

where $X, Y, Z \in \mathbb{C}^{n\times n}$. *What is the FLOP count for computing the entries of* W*?*

The equivalent formula

$$w_{ij} = \sum_{p=1}^{n}\left(\sum_{q=1}^{n} x_{ip}y_{pq}z_{qj}\right),$$

computes the bracket contents first for all p, j, before doing the sum over p. What is the FLOP count for this alternative method of computing the entries of W?

Using what you have learned, propose an $\mathcal{O}(n^3)$ procedure for computing $A \in \mathbb{C}^{n\times n}$ with entries

$$a_{ij} = \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} E_{ki}F_{ki}G_{lk}H_{lm}F_{lm}G_{mj}.$$

Exercise 47 *Let $L_1, L_2 \in \mathbb{C}^{m\times m}$ be lower triangular matrices. If we apply the usual formula for multiplying matrices, we will waste computation time by multiplying numbers by zero and then adding the result to other numbers. Describe a more efficient algorithm as pseudo-code and compute the FLOP count, comparing with the FLOP count for the standard algorithm.*

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 48 (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 $\|AB\|_F \leq \|A\|_F\|B\|_F$.)

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

Definition 49 (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.

Exercise 50 *We can reformulate the induced definition as a constrained optimisation problem*

$$\|A\|_{(m,n)} = \sqrt{\sup_{x\in\mathbb{C}^n, \|x\|^2=1} \|Ax\|_{(m)}^2}.$$

Introduce a Lagrange multiplier $\lambda \in \mathbb{C}$ to enforce the constraint $\|x\|^2 = 1$. Consider the case above where the norms on $\mathbb{C}^m$ and $\mathbb{C}^n$ are both 2-norms. Show that λ must be an eigenvalue of some matrix (which you should compute). Hence, given those eigenvalues, provide an expression for the operator norm of A.

The `cla_utils.exercises4.operator_2_norm()` *function has been left unimplemented. It takes in an $m \times n$ matrix A and returns the operator norm using the procedure in this exercise. You may use the built in function* `numpy.linalg.eig()` *to compute the eigenvalues of any matrices that you need. (We will discuss algorithms to compute eigenvalues later in the course.) The test script* `test_exercises4.py` *in the* `test` *directory will test this function.*

Exercise 51 *Add a function to* `cla_utils.exercises4` *to verify the inequality $\|Ax\| \leq \|A\|\|x\|$ using* `cla_utils.exercises4.operator_2_norm()`*, considering various m and n.*

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

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

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

Definition 53 (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 54 *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 55

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

so

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

as required.

Exercise 56 *Add a function to* `cla_utils.exercises4` *to verify this theorem for various l, m and n.*

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 57 (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 58 (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 59 (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

$$\kappa = \frac{\|J\|/\|f\|}{1/\|x\|} = \frac{\|J\|\|x\|}{\|f\|}.$$

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.

Exercise 60 *The* `cla_utils.exercises4.cond()` *function has been left unimplemented. It takes in an* $m \times m$ *matrix* A *and returns the condition number. You should use a method similar to that in Exercise 50, using the* `numpy.linalg.eig()` *to compute the eigenvalues of any matrices that you need. Try to think about minimising the number of eigenvalue calculations you need to do. The test script* `test_exercises4.py` *in the* `test` *directory will test this function.*

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 largest 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 61 (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 62 (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 63 (Floating point axiom II)

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

Exercise 64 *The formula for the roots of a quadratic equation $x^2 - 2px - q = 0$ is well-known,*

$$x = p \pm \sqrt{p^2 + q}.$$

Show that the smallest root (with the minus sign above) also satisfies

$$x = \frac{q}{p + \sqrt{p^2 + q}}.$$

In the case $p = 12345678$ and $q = 1$, compare the result of these two methods for computing the smallest root when using double floating point arithmetic (the default floating point numbers in Python/NumPy). Which is more accurate? Why is this?

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. We shall see below that we have to lower our aspirations depending on the condition number of A.

Definition 65 (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 66 (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 67 (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 68 *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,*

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

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 69 *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 70 *See the textbook by Trefethen and Bau, Lecture 16.*

Exercise 71 *The* `cla_utils.exercises5.backward_stability_householder()` *function has been left unimplemented. It generates random Q_1 and R_1 matrices of dimension m provided, and forms $A = QR$. It is very important that the two matrices Q_1 and R_1 are uncorrelated (in particular, computing them as the QR factorisation of the same matrix would spoil the experiment). To complete the function, pass A to the built-in QR factorisation function* `numpy.linalg.qr()` *(which uses Householder transformations) to get Q_2 and R_2. Print out the value of $\|Q_2 - Q_1\|$, $\|R_2 - R_1\|$, $\|A - Q_2R_2\|$. Explain what you see using what you know about the stability of the Householder algorithm.*

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 the m dimensional extension of the formula for the inverse of 2×2 matrices that you learned at school. Kramer's rule has an operation count scaling like $\mathcal{O}(m!)$ and is numerically unstable. Hence it is so disasterous that we won't even show the formula for Kramer's rule here.

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.

Exercise 72 *The* `cla_utils.exercises5.solve_R()` *function has been left unimplemented. It should implement the $\mathcal{O}(m^2)$ back-substitution algorithm to solve $Rx = b$, with a single loop over the columns. The test script* `test_exercises5.py` *in the* `test` *directory will test this function.*

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 $\|\delta\tilde{R}\|/\|\tilde{R}\| = \mathcal{O}(\varepsilon)$.

Exercise 73 *Complete the function* `cla_utils.exercises5.back_stab_solve_R()` *so that it verifies backward stability for back substitution, using* `cla_utils.exercises5.solve_R()`.

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

Theorem 74 *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 75 *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 76 *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 77 *From Theorem 67, 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 $\kappa(A)$.

Exercise 78 *Complete the function* `cla_utils.exercises5.back_stab_householder_solve()` *so that it verifies backward stability for solving $m \times m$ dimensional square systems $Ax = b$ using* `cla_utils.exercises3.householder_solve()`.

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

$$A_n = L_n \dots L_2 L_1 A = U.$$

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 k-th 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. Note that the matrices $L_1, L_2, \dots$ should not be explicitly formed during the elimination process, they are just a mathematical concept to translate from the row operations into the final L matrix.

Exercise 79 *Having said that, let's take a moment to compute some examples using the $L_1, L_2, \dots$ matrices (to help with understanding). The* `cla_utils.exercises6.get_Lk()` *function has been left unimplemented. It should return one of these matrices given the l_k entries. The test script* `test_exercises6.py` *in the* `test` *directory will test this function.*

Once it passes the tests, experiment with the inverse and multiplication properties above, to verify that they work.

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

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

Exercise 80 *Since the diagonal entries of L are all ones, the total amount of combined memory required to store L and U is the same as the amount of memory required to store A. Further, each iteration of the LU factorisation algorithm computes one column of L and one rows of U, and the corresponding column an row of A are not needed for the rest of the algorithm. This creates the opportunity for a memory-efficient 'in-place' algorithm in which the matrix A is modified until it contains the values for L and U.*

The `cla_utils.exercises6.LU_inplace()` *function has been left unimplemented. It should implement this in-place low-storage procedure, applying the changes to the provided matrix A. The test script* `test_exercises6.py` *in the* `test` *directory will test this function.*

Exercise 81 *The LU factorisation requires 3 loops (this is why it has a cubic FLOP count). In the algorithm above, there are two explicit loops and one explicit one (in the slice notation). It is possible to rewrite this in*

a single loop, using an outer product. Identify this outer product, and update `cla_utils.exercises6.LU_inplace()` *to make use of this reformulation (using* `numpy.outer()`*). Do you notice any improvement in speed?*

Exercise 82 *The functions* `cla_utils.exercises6.solve_L()` *and* `cla_utils.exercises6.solve_U()` *have been left unimplemented. They should use forward and backward substitution to solve lower and upper triangular systems respectively. The interfaces are set so that multiple right hand sides can be provided and solved at the same time. The functions should only use one loop over the columns of L (or U), to efficiently solve the multiple problems. The test script* `test_exercises6.py` *in the* `test` *directory will test these functions.*

Exercise 83 *Propose an algorithm to use the LU factorisation to compute the inverse of a matrix. The functions* `cla_utils.exercises6.inverse_LU()` *has been left unimplemented. Complete it using your algorithm, using functions developed in the previous exercises where possible. The test script* `test_exercises6.py` *in the* `test` *directory will test these functions.*

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 84 (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, $L_1 P_2$ 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, $P_2 L_1 P_2$ exchanges row 2 of $L_1 P_2$ with row j of $L_1 P_2$. This just exchanges l_{12} with l_{1j}, and swaps the 1s in columns 2 and j back to the diagonal. In summary, $P_2 L_1 P_2$ is the same as L_1 but with l_{12} exchanged with l_{1j}.

Moving on to the next stage, and we have

$$A_3 = L_3 P_3 L_2 L_1 P_2 P_1 = L_3 \underbrace{P_3 L_2 P_3}_{=L_2^{(3)}} \underbrace{P_3 L_1 P_3}_{=L_1^{(3)}} P_3 P_2 P_1.$$

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

$$\underbrace{L_{m-1}^{(m-1)} \dots L_2^{(m-1)} L_1^{(m-1)}}_{L^{-1}} \underbrace{P_{m-1} \dots P_2 P_1}_{P} = U,$$

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

$$L^{-1} P A = U \implies P A = L U.$$

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.

We call this strategy *partial pivoting*. In contrast, *complete pivoting* additionally employs permutations Q_k on the right that swap columns of A_k as well as the rows swapped by the permutations P_k. By similar arguments, one can obtain the LU factorisation with complete pivoting, $PAQ = LU$.

Exercise 85 *The function* `cla_utils.exercises7.perm()` *has been left unimplemented. It should take an $m \times m$ permutation matrix P, stored as a vector of indices $p \in \mathbb{N}^m$ so that $(Px)_i = x_{p_i}$, $i = 1, 2, \ldots, m$, and replace it with the matrix $P_{i,j}P$ (also stored as a vector of indices) where $P_{i,j}$ is the permutation matrix that exchanges the entries i and j. The test script* `test_exercises7.py` *in the* `test` *directory will test this function.*

Exercise 86 *The function* `cla_utils.exercises7.LUP()` *has been left unimplemented. It should extend the in-place algorithm for LU factorisation (with the outer-product formulation, if you managed it) to the LUP factorisation. As well as computing L and U "in place" in the array where the input A is stored, it will compute a permutation matrix, which can should be constructed using* `cla_utils.exercises7.perm()`*.The test script* `test_exercises7.py` *in the* `test` *directory will test this function.*

Exercise 87 *The function* `cla_utils.exercises7.solve_LUP()` *has been left unimplemented. It should use the LUP code that you have written to solve the equation $Ax = b$ for x given inputs A and b. The test script* `test_exercises7.py` *in the* `test` *directory will test this function.*

Exercise 88 *Show how to compute the determinant of A from the LUP factorisation in $\mathcal{O}(m)$ time (having already constructed the LUP factorisation which costs $\mathcal{O}(m^3)$). Complete the function* `cla_utils.exercises7.det_LUP()` *to implement this computation. The test script* `test_exercises7.py` *in the* `test` *directory will test this function.*

4.3 Stability of LU factorisation

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

Theorem 89 *Let $\tilde{L}$ and $\tilde{U}$ be the result of the Gaussian elimination algorithm implemented in a floating point number system satisfying axioms I and II. If no zero pivots are encountered, then*

$$\tilde{L}\tilde{U} = A + \delta A$$

where

$$\frac{\|\delta A\|}{\|L\|\|U\|} = \mathcal{O}(\varepsilon),$$

for some perturbation δA.

The algorithm is backward stable if $\|L\|\|U\| = \mathcal{O}(\|A\|)$, but there will be problems if $|L\|\|U\| \gg \|A\|$. For a proof of this result, see the textbook by Golub and van Loan.

A similar result exists for pivoted LU. The main extra issue is that small changes could potentially lead to a different pivoting matrix $\tilde{P}$ which is then $O(1)$ different from P. This is characterised in the following result (which we also do not prove).

Theorem 90 *Let $\tilde{P}$, $\tilde{L}$ and $\tilde{U}$ be the result of the partial pivoted Gaussian elimination algorithm implemented in a floating point number system satisfying axioms I and II. If no zero pivots are encountered, then*

$$\tilde{L}\tilde{U} = A + \delta A$$

where

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

for some perturbation δA, and where ρ is the growth factor,

$$\rho = \frac{\max_{ij} |u_{ij}|}{|a_{ij}|}.$$

Thus, partial pivoting (and complete pivoting turns out not to help much extra) can keep the entries in L under control, but there can still be pathological cases where entries in U can get large, leading to large ρ and unstable computations.

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 $i = 1, 2, \ldots, m$.

Similarly, if $A \in \mathcal{C}^{dm \times dm}$ is block diagonal, i.e.

$$A = \begin{pmatrix} B_1 & 0 & \ldots & 0 \\ 0 & B_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \ldots & B_m \end{pmatrix},$$

where $B_i \in \mathcal{C}^{d \times d}$ for $i = 1, 2, \ldots, m$. The inverse of A is

$$A = \begin{pmatrix} B_1^{-1} & 0 & \ldots & 0 \\ 0 & B_2^{-1} & \ldots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \ldots & B_m^{-1} \end{pmatrix}.$$

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.

Exercise 91 *Using your own LU factorisation, compute the LU factorisation of the 10×10 matrix $A = I + e_3 e_9^*$. What do you observe about the number of non-zero entries in L and U? Explain this using what you have just learned about banded matrices. Can the situation be improved by pivoting?*

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.

Another example that we have already encountered is unitary matrices Q. Since $Q^{-1} = Q^*$, solving the system $Qx = b$ is just the cost of applying Q^*, with operation count $\mathcal{O}(m^2)$.

An important matrix that we shall encounter later is an upper Hessenberg matrix, that has a lower bandwidth of 1, but no particular zero structure above the diagonal. In this case, the L matrix is still banded (with lower bandwidth 1) but the U matrix is not. This means that there are still savings due to the zeros in L, but work has to be done on the entire column of U above the diagonal, and so solving an upper Hessenberg system has operation count $\mathcal{O}(m^2)$.

4.5 Cholesky factorisation

An example of extra structure which we shall discuss in a bit more detail is the case of Hermitian positive definite matrices. Recall that a Hermitian matrix satisfies $A^* = A$, whilst positive definite means that

$$x^* A x > 0, \, \forall \|x\| > 0.$$

When A is Hermitian positive definite, it is possible to find an upper triangular matrix R such that $A = R^* R$, which is called the Cholesky factorisation. To show that it is possible to compute the Cholesky factorisation, we start by assuming that A has a 1 in the top-left hand corner, so that

$$A = \begin{pmatrix} 1 & w^* \\ w & K \end{pmatrix}$$

where w is a $m-1$ vector containing the rest of the first column of A, and K is an $(m-1) \times (m-1)$ Hermitian positive definite matrix. (Exercise: show that K is Hermitian positive definite.)

After one stage of Gaussian elimination, we have

$$\underbrace{\begin{pmatrix} 1 & 0 \\ -w & I \end{pmatrix}}_{L_1^{-1}} \underbrace{\begin{pmatrix} 1 & w^* \\ w & K \end{pmatrix}}_{A} = \begin{pmatrix} 1 & w^* \\ 0 & K - ww^* \end{pmatrix}.$$

Further,

$$\begin{pmatrix} 1 & w^* \\ 0 & K - ww^* \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & K - ww^* \end{pmatrix}}_{A_1} \underbrace{\begin{pmatrix} 1 & w^T \\ 0 & I \end{pmatrix}}_{(L_1^{-1})^* = L_1^{-*}},$$

so that $A = L_1^{-1} A_1 L_1^{-*}$. If $a_{11} \neq 1$, we at least know that $a_{11} = e_1^* A e_1 > 0$, and the factorisation becomes

$$A = \underbrace{\begin{pmatrix} \alpha & 0 \\ w/\alpha & I \end{pmatrix}}_{R_1^T} \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & K - \frac{ww^*}{a_{11}} \end{pmatrix}}_{A_1} \underbrace{\begin{pmatrix} \alpha & w/\alpha \\ 0 & I \end{pmatrix}}_{R_1},$$

where $\alpha = \sqrt{a_{11}}$. We can check that A_1 is positive definite, since

$$x^* A_1 x = x^* R_1^{-*} A R_1 x = (R_1^{-1} x)^* A R_1 x = y^* A y > 0, \text{ where } y = R_1 x.$$

Hence, $K - ww^*/a_{11}$ is positive definite, since

$$r^* \left(K - \frac{ww^*}{a_{11}} \right) r = \begin{pmatrix} 0 \\ r \end{pmatrix}^* A_1 \begin{pmatrix} 0 \\ r \end{pmatrix} > 0,$$

and hence we can now perform the same procedure all over again to $K - ww^*/a_{11}$. By induction we can always continue until we have the required Cholesky factorisation, which is unique (since there were no choices to be made at any step).

We can then present the Cholesky factorisation as pseudo-code.

- $R \leftarrow A$
- FOR $k = 1$ TO m
  - FOR $j = k + 1$ to m
    - $R_{j,j:m} \leftarrow R_{j,j:m} - R_{k,j:m} \bar{R}_{kj} / R_{kk}$
  - $R_{k,k:m} \leftarrow R_{k,k:m} / \sqrt{R_{k:k}}$

The operation count of the Cholesky factorisation is dominated by the operation inside the j loop, which has one division, $m - j + 1$ multiplications, and $m - j + 1$ subtractions, giving $\sim 2(m - j)$ FLOPs. The total operation count is then

$$N_{\text{FLOPs}} = \sum_{k=1}^{m} \sum_{j=k+1}^{m} \sim \frac{1}{3} m^3.$$

CHAPTER

FIVE

FINDING EIGENVALUES OF MATRICES

We start with some preliminary terminology. A vector $x \in \mathbb{C}^m$ is an *eigenvector* of a square matrix $A \in \mathbb{C}^{m\times m}$ with *eigenvalue* λ if $Ax = \lambda x$. An eigenspace is the subspace $E_\lambda \subset \mathbb{C}^m$ containing all eigenvectors of A with eigenvalue λ.

There are a few reasons why we are interested in computing eigenvectors and eigenvalues of a matrix A.

1. Eigenvalues and eigenvectors encode information about A.
2. Eigenvalues play an important role in stability calculations in physics and engineering.
3. We can use eigenvectors to underpin the solution of linear systems involving A.
4. ...

5.1 How to find eigenvalues?

The method that we first encounter in our mathematical education is to find solutions of $(A - \lambda I)x = 0$, which implies that $\det(A - \lambda I) = 0$. This gives a degree m polynomial to solve for λ, called the *characteristic polynomial*. Unfortunately, there is no general solution for polynomials of degree 5 or greater (from Galois theory). Further, the problem of finding roots of polynomials is numerically unstable. All of this means that we should avoid using polynomials finding eigenvalues. Instead, we should try to apply transformations to the matrix A to a form that means that the eigenvalues can be directly extracted.

Example 92 *Consider the $m \times m$ diagonal matrix*

$$A = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & 2 & \dots & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \dots & \ddots & \vdots \\ 0 & 0 & \dots & \dots & m \end{pmatrix}.$$

The characteristic polynomial of A is

$$(\lambda - 1)(\lambda - 2)\dots(\lambda - m),$$

and the eigenvalues are clearly $1, 2, 3, \dots, m$. This is called the Wilkinson Polynomial. Numpy has some tools for manipulating polynomials which are useful here. When an $m \times m$ array is passed in to `numpy.poly()`, it returns an array of coefficients of the polynomial. For $m = 20$, obtain this array and then perturb the coefficients $a_k \to \tilde{a}_k = a_k(1 + 10^{-10}r_k)$ where r_k are randomly sampled normally distributed numbers with mean 0 and variance 1. `numpy.roots()` will compute the roots of this perturbed polynomial. Plot these roots as points in the complex plane. Repeat this 100 times, superposing the root plots on the same graph. What do you observe? What does it tell you about this problem and what should we conclude about the wisdom of finding eigenvalues using characteristic polynomials?

The eigenvalue decomposition of a matrix A finds a nonsingular matrix X and a diagonal matrix Λ such that

$$A = X\Lambda X^{-1}.$$

The diagonal entries of Λ are the eigenvalues of A. Hence, if we could find the eigenvalue decomposition of A, we could just read off the eigenvalues of A; the eigenvalue decomposition is "eigenvalue revealing". Unfortunately, it is not always easy or even possible to transform to an eigenvalue decomposition. Hence we shall look into some other eigenvalue revealing decompositions of A.

We quote the following result that explains when an eigenvalue decomposition can be found.

Theorem 93 *An $m \times m$ matrix A has an eigenvalue decomposition if and only if it is non-defective, meaning that the geometric multiplicity of each eigenvalue (the dimension of the eigenspace for that eigenvalue) is equal to the algebraic multiplicity (the number of times that the eigenvalue is repeated as a root in the characteristic polynomial $\det(I\lambda - A) = 0$.*

If the algebraic multiplicity is greater than the geometric multiplicity for any eigenvalue of A, then the matrix is defective, the eigenspaces do not span $\mathbb{C}^m$, and an eigenvalue decomposition is not possible.

This all motivates the search for other eigenvalue revealing decompositions of A.

Definition 94 (Similarity transformations) *For $X \in \mathbb{C}^{m\times m}$ a nonsingular matrix, the map $A \mapsto X^{-1}AX$ is called a similarity transformation of A. Two matrices A and B are* similar *if $B = X^{-1}AX$.*

The eigenvalue decomposition shows that (when it exists), A is similar to Λ. The following result shows that it may be useful to examine other similarity transformations.

Theorem 95 *Two similar matrices A and B have the same characteristic polynomial, eigenvalues, and geometric multiplicities.*

Proof 96 *See a linear algebra textbook.*

The goal is to find a similarity transformation such that A is transformed to a matrix B that has some simpler structure where the eigenvalues can be easily computed (with the diagonal matrix of the eigenvalue decomposition being one example).

One such transformation comes from the Schur factorisation.

Definition 97 (Schur factorisation) *A Schur factorisation of a square matrix A takes the form $A = QTQ^*$, where Q is unitary (and hence $Q^* = Q^{-1}$) and T is upper triangular.*

It turns out that, unlike the situation for the eigenvalue decomposition, the following is true.

Theorem 98 *Every square matrix has a Schur factorisation.*

This is useful, because the characteristic polynomial of an upper triangular matrix is just $\prod_{i=1}^{m}(\lambda - T_{ii})$, i.e. the eigenvalues of T are the diagonal entries $(T_{11}, T_{22}, \ldots, T_{mm})$. So, if we can compute the Schur factorisation of A, we can just read the eigenvalues from the diagonal matrices of A.

There is a special case of the Schur factorisation, called the unitary diagonalisation

Definition 99 (Unitary diagonalisation) *A unitary diagonalisation of a square matrix A takes the form $A = Q\Lambda Q^*$, where Q is unitary (and hence $Q^* = Q^{-1}$) and Λ is diagonal.*

A unitary diagonalisation is a Schur factorisation *and* an eigenvalue decomposition.

Theorem 100 *A Hermitian matrix is unitary diagonalisable, with real Λ.*

Hence, if we have a Hermitian matrix, we can follow a Schur factorisation strategy (such as we shall develop in this section), and obtain an eigenvalue decomposition as a bonus.

5.2 Transformations to Schur factorisation

Just as for the QR factorisations, we will compute the Schur factorisation successively, with multiplication by a sequence of unitary matrices $Q_1, Q_2, \ldots$. There are two differences for the Schur factorisation. First, the matrices must be multiplied not just on the left but also on the right with the inverse, i.e.

$$A \mapsto \underbrace{Q_1^* A Q_1}_{A_1} \mapsto \underbrace{Q_2^* Q_1^* A Q_2 Q_1}_{A_2}, \ldots$$

At each stage, we have a similarity transformation,

$$A = \underbrace{Q_1 Q_2 \ldots Q_k}_{=Q} A_k \underbrace{Q_k^* \ldots Q_2^* Q_1^*}_{=Q^*},$$

i.e. A is similar to A_k. Second, the successive sequence is infinite, i.e. we will develop an iterative method that converges in the limit $k \to \infty$. We should terminate the iterative method when A_k is sufficiently close to being upper triangular (which can be measured by checking some norm on the lower triangular part of A and stopping when it is below a tolerance).

We should not be surprised by the news that the method needs to be iterative, since if the successive sequence were finite, we would have derived an explicit formula for computing the eigenvalues of the characteristic polynomial of A which is explicit in general.

In fact, there are two stages to this process. The first stage, which is finite (takes $m-1$ steps) is to use similarity transformations to upper Hessenberg form ($H_{ij} = 0$ for $i > j+1$). If A is Hermitian, then H will be tridiagonal. This stage is not essential but it makes the second, iterative, stage much faster.

5.3 Similarity transformation to upper Hessenberg form

We already know how to use a unitary matrix to set all entries to zero below the diagonal in the first column of A by left multiplication $Q_1^* A$, because this is the Householder algorithm. The problem is that we then have to right multiply by Q_1 to make it a similarity transformation, and this puts non-zero entries back in the column again. The easiest way to see this is to write $Q_1^* A Q_1 = (Q_1^*(Q_1^* A)^*)^*$. $(Q_1^* A)^*$ has zeros in the first row to the right of the first entry. Then, $Q_1^*(Q_1^* A)$ creates linear combinations of the first column with the other columns, filling the zeros in with non-zero values again. Then finally taking the adjoint doesn't help with these non-zero values. Again, we shouldn't be surprised that this is impossible, because if it was, then we would be able to build a finite procedure for computing eigenvalues of the characteristic polynomial, which is impossible in general.

Exercise 101 *The* `cla_utils.exercises8.Q1AQ1s()` *function has been left uncompleted. It should apply the Householder transformation Q_1 to the input A (without forming Q_1 of course) that transforms the first column of A to have zeros below the diagonal, and then apply a transformation equivalent to right multiplication by Q_1^* (again without forming Q_1). The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

Experiment with the output of this function. What happens to the first column?

A slight modification of this idea (and the reason that we can transform to upper Hessenberg form) is to use a Householder rotation Q_1^* to set all entries to zero below the *second* entry in the first column. This matrix leaves the first row unchanged, and hence right multiplication by Q_1 leaves the first column unchanged. We can create zeros using Q_1^* and Q_1 will not destroy them. This procedure is then repeated with multiplication by Q_2^*, which leaves the first two rows unchanged and puts zeros below the third entry in the second column, which are not spoiled by right multiplication by Q_2. Hence, we can transform A to a similar upper Hessenberg matrix H in $m-2$ iterations.

This reduction to Hessenberg form can be expressed in the following pseudo-code.

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

Note the similarities and differences with the Householder algorithm for computing the QR factorisation.

Exercise 102 *The* `cla_utils.exercises8.hessenberg()` *function has been left unimplemented. It should implement the algorithm above, using only one loop over k. It should return the resulting Hessenberg matrix. At the left multiplication, your implementation should exploit the fact that zeros do not need to be re-computed where there are already expected to be zeros. The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

To calculate the operation count, we see that the algorithm is dominated by the two updates to A, the first of which applies a Householder reflection to rows from the left, and the second applies the same reflections to columns from the right.

The left multiplication applies a Householder reflection to the last $m - k$ rows, requiring 4 FLOPs per entry. However, these rows are zero in the first $k - 1$ columns, so we can skip these and just work on the last $m - k + 1$ entries of each of these rows.

Then, the total operation count for the left multiplication is

$$4 \times \sum_{k=1}^{m-1} (m-k)(m-k+1) \sim \frac{4}{3}m^3.$$

The right multiplication does the same operations but now there are no zeros to take advantage of, so all m entries in the each of the last $m - k$ columns need to be manipulated. With 4 FLOPs per entry, this becomes

$$4 \times \sum_{k=1}^{m-1} m(m-k) \sim \frac{10}{3}m^3 FLOPs.$$

In the Hermitian case, the Hessenberg matrix becomes tridiagonal, and these extra zeros can be exploited, leading to an operation count $\sim 4m^3/3$.

It can be shown that this transformation to a Hessenberg matrix is backwards stable, i.e. in a floating point implementation, it gives $\tilde{Q}, \tilde{H}$ such that

$$\tilde{Q}\tilde{H}\tilde{Q}^* = A + \delta A, \ \frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\varepsilon),$$

for some δA.

Exercise 103 *The* `cla_utils.exercises8.hessenbergQ()` *function has been left unimplemented. It should implement the Hessenberg algorithm again (you can just copy paste the code from the previous exercise) but it should also return the matrix Q such that $QHQ^* = A$. You need to work out how to alter the algorithm to construct this. The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

Exercise 104 *The* `cla_utils.exercises8.ev()` *function has been left unimplemented. It should return the eigenvalues and eigenvectors of A by first reducing to Hessenberg form, using the functions you have already created, and then calling* `cla_utils.exercises8.hessenberg_ev()`*, which computes the eigenvectors and eigenvalues of upper Hessenberg matrices (do not edit this function!). The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

In the next few sections we develop the iterative part of the transformation to the upper triangular matrix T. This algorithm works for a broad class of matrices, but the explanation is much easier for the case of real symmetric matrices, which have real eigenvalues and orthogonals eigenvectors (which we shall normalise to $\|q_i\| = 1$, $i = 1, 2, \ldots, m$). The idea is that we will have already transformed to Hessenberg form, which will be tridiagonal in this case. Before describing the iterative transformation, we will discuss a few key tools in explaining how it works.

5.4 Rayleigh quotient

The first tool that we shall consider is the Rayleigh quotient. If $A \in \mathbb{C}^{m\times m}$ is a real symmetric matrix, then the Rayleigh quotient of a vector $x \in \mathbb{C}^m$ is defined as

$$r(x) = \frac{x^T A x}{x^T x}.$$

If x is an eigenvector of A, then

$$r(x) = \frac{x^T \lambda x}{x^T x} = \lambda,$$

i.e. the Rayleigh quotient gives the corresponding eigenvalue. If x is not exactly an eigenvector of A, but is just close to one, we might hope that $r(x)$ is close to being an eigenvalue. To investigate this we will consider the Taylor series expansion of $r(x)$ about an eigenvector q_J of A. We have

$$\nabla r(x) = \frac{2}{x^T x}\left(Ax - r(x)x\right),$$

which is zero when $x = q_J$, because then $r(q_J) = \lambda_J$: eigenvectors of A are stationary points of $r(x)$! Hence, the Taylor series has vanishing first order term,

$$r(x) = r(q_J) + (x - q_J)^T \underbrace{\nabla r(q_J)}_{=0} + \mathcal{O}(\|x - q_J\|^2),$$

i.e.

$$r(x) - r(q_J) = \mathcal{O}(\|x - q_J\|^2), \quad \text{as } x \to q_J.$$

The Rayleigh quotient gives a quadratically accurate estimate to the eigenvalues of A.

Exercise 105 *Add a function to* `cla_utils.exercises8` *that investigates this property by:*

1. *Forming a Hermitian matrix A,*
2. *Finding an eigenvector v of A with eigenvalue λ (you can use* `numpy.linalg.eig()` *for this),*
3. *Choosing a perturbation vector r, and perturbation parameter $\epsilon > 0$,*
4. *Comparing the Rayleigh quotient of $v + \epsilon r$ with λ,*
5. *Plotting (on a log-log graph, use* `matplotlib.pyplot.loglog()`*) the error in estimating the eigenvalue as a function of ϵ.*

The best way to do this is to plot the computed data values as points, and then superpose a line plot of $a\epsilon^k$ for appropriate value of k and a chosen so that the line appears not to far away from the points on the same scale. This means that we can check by eye if the error is scaling with ϵ at the expected rate.

5.5 Power iteration

Power iteration is a simple method for finding the eigenvalue of A with largest eigenvalue (in magnitude). It is based on the following idea. We expand a vector v in eigenvectors of A,

$$v = a_1 q_1 + a_2 q_2 + \dots a_m q_m,$$

where we have ordered the eigenvalues so that $|\lambda_1| \geq |\lambda_2| \geq |\lambda 3| \geq \dots \geq |\lambda_m$.

Then,

$$Av = a_1 \lambda_1 q_1 + a_2 \lambda_2 q_2 + \dots a_m \lambda_m q_m,$$

and hence, repeated applications of A gives

$$\begin{aligned} A^k v &= \underbrace{AA\dots A}_{k \text{ times}} v \\ &= a_1 \lambda_1^k q_1 + a_2 \lambda_2^k q_2 + \dots a_m \lambda_m^k q_m. \end{aligned}$$

If $|\lambda_1| > |\lambda_2|$, then provided that $a_1 = q_1^T v \neq 0$, the first term $a_1 \lambda_1^k q_1$ rapidly becomes larger than all of the others, and so $A^k v \approx a_1 \lambda_1^k q_1$, and we can normalise to get $q_1 \approx A^k v / \|A^k v\|$. To keep the magnitude of the estimate from getting too large or small (depending on the size of λ_1 relative to 1), we can alternately apply A and normalise, which gives the power iteration. Along the way, we can use the Rayleigh quotient to see how our approximation of the eigenvalue is going.

- Set v_0 to some initial vector (hoping that $\|q_1^T v_0\| > 0$).
- FOR $k = 1, 2, \dots$
  - $w \leftarrow A v^{k-1}$,
  - $v^k \leftarrow w / \|w\|$,
  - $\lambda^{(k)} \leftarrow (v^k)^T A v^k$.

Here we have used the fact that $\|v^k\| = 1$, so there is no need to divide by it in the Rayleigh quotient. We terminate the power iteration when we decide that the changes in λ indicate that the error is small. This is guided by the following result.

Theorem 106 *If $|\lambda_1| > |\lambda_2|$ and $\|q_1^T v_0\| > 0$, then after k iterations of power iteration, we have*

$$\|v^k - \pm q_1\| = \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right), \quad |\lambda^{(k)} - \lambda_1| = \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}\right),$$

as $k \to \infty$. At each step $\pm$ we mean that the result holds for either $+$ or $-$.

Proof 107 *We have already shown the first equation using the Taylor series, and the second equation comes by combining the Taylor series error with the Rayleigh quotient error.*

The $\pm$ feature is a bit annoying, and relates to the fact that the normalisation does not select v^k to have the direction as q_1.

Exercise 108 *The* `cla_utils.exercises9.pow_it()` *function has been left unimplemented. It should apply power iteration to a given matrix and initial vector, according to the docstring. Note that the docstring provides a different truncation criteria, not based on the Rayleigh quotient, so that it can be used for non-Hermitian matrices too. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 109 *The functions* `cla_utils.exercises9.A3()` *and* `cla_utils.exercises9.B3()` *each return a 3x3 matrix, A_3 and B_3 respectively. Apply* `cla_utils.exercises9.pow_it()` *to each of these functions. What differences in behaviour do you observe? What is it about A_3 and B_3 that causes this?*

5.6 Inverse iteration

Inverse iteration is a modification of power iteration so that we can find eigenvalues other than λ_1. To do this, we use the fact that eigenvectors q_j of A are also eigenvectors of $(A-\mu I)^{-1}$ for any $\mu \in \mathbb{R}$ not an eigenvalue of A (otherwise $A-\mu I$ is singular). To show this, we write

$$(A-\mu I)q_j = (\lambda_j - \mu)q_j \implies (A-\mu I)^{-1}q_j = \frac{1}{\lambda_j - \mu}q_j.$$

Thus q_j is an eigenvalue of $(A-\mu I)^{-1}$ with eigenvalue $1/(\lambda_j - \mu)$. We can then apply power iteration to $(A-\mu I)^{-1}$ (which requires a matrix solve per iteration), which converges to an eigenvector q for which $1/|\lambda - \mu|$ is smallest, where λ is the corresponding eigenvalue. In other words, we will find the eigenvector of A whose eigenvalue is closest to μ.

This algorithm is called inverse iteration, which we express in pseudo-code below.

- $v^0 \leftarrow$ some initial vector with $\|v^0\| = 1$.
- FOR $k = 1, 2, \ldots$
  - SOLVE $(A-\mu I)w = v^{k-1}$ for w
  - $v^k \leftarrow w/\|w\|$
  - $\lambda^{(k)} \leftarrow (v^k)^T A v^k$

We can then directly extend Theorem 106 to the inverse iteration algorithm. We conclude that the convergence rate is not improved relative to power iteration, but now we can "dial in" to different eigenvalues by choosing μ.

Exercise 110 *The* `cla_utils.exercises9.inverse_it()` *function has been left unimplemented. It should apply inverse iteration to a given matrix and initial vector, according to the docstring. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 111 *Using the A_3 and B_3 matrices, explore the inverse iteration using different values of μ. What do you observe?*

5.7 Rayleigh quotient iteration

Since we can use the Rayleigh quotient to find an approximation of an eigenvalue, and we can use an approximation of an eigenvalue to find the nearest eigenvalue using inverse iteration, we can combine them together. The idea is to start with a vector, compute the Rayleigh quotient, use the Rayleigh quotient for μ, then do one step of inverse iteration to give an updated vector which should now be closer to an eigenvector. Then we iterate this whole process. This is called the Rayleigh quotient iteration, which we express in pseudo-code below.

- v^0 some initial vector with $\|v^0\| = 1$.
- $\lambda^{(0)} \leftarrow (v^0)^T A v^0$
- FOR $k = 1, 2, \ldots$
  - SOLVE $(A - \lambda^{(k-1)} I)w = v^{k-1}$ for w
  - $v^k \leftarrow w/\|w\|$
  - $\lambda^{(k)} \leftarrow (v^k)^T A v^k$

This dramatically improves the convergence since if $\|v^k - q_J\| = \mathcal{O}(\delta)$ for some small δ, then the Rayleigh quotient gives $|\lambda^{(k)} - q_J| = \mathcal{O}(\delta^2)$. Then, inverse iteration gives an estimate

$$\|v^{k+1} - \pm q_J\| = \mathcal{O}(|\lambda^{(k)} - \lambda_J| \|v^k - q_J\|) = \mathcal{O}(\delta^3).$$

Thus we have cubic convergence, which is super fast!

Exercise 112 *The* `cla_utils.exercises9.rq_it()` *function has been left unimplemented. It should apply inverse iteration to a given matrix and initial vector, according to the docstring. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 113 *The interfaces to* `cla_utils.exercises9.inverse_it()` *and* `cla_utils.exercises9.rq_it()` *have been designed to optionally provide the iterated values of the eigenvector and eigenvalue. For a given initial condition (and choice of μ in the case of inverse iteration), compare the convergence speeds of the eigenvectors and eigenvalues, using some example matrices of different sizes (don't forget to make them Hermitian).*

5.8 The pure QR algorithm

We now describe the QR algorithm, which will turn out to be an iterative algorithm that converges to the diagonal matrix (upper triangular matrix for the general nonsymmetric case) that A is similar to. Why this works is not at all obvious at first, and we shall explain this later. For now, here is the algorithm written as pseudo-code.

- $A^{(0)} \leftarrow A$
- FOR $k = 1, 2, \ldots$
  - FIND $Q^{(k)}, R^{(k)}$ such that $Q^{(k)} R^{(k)} = A^{(k-1)}$ (USING QR FACTORISATION)
  - $A^{(k)} = R^{(k)} Q^{(k)}$

Exercise 114 *The* `cla_utils.exercises9.pure_QR()` *function has been left unimplemented. It should implement the pure QR algorithm as above, using your previous code for finding the QR factorisation using Householder transformations. You should think about avoiding unecessary allocation of new numpy arrays inside the loop. The method of testing for convergence has been left as well. Have a think about how to do this and document your implementation. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 115 *Investigate the behaviour of the pure QR algorithm applied to the functions provided by* `cla_utils.exercises9.get_A100()`, `cla_utils.exercises9.get_B100()`, `cla_utils.exercises9.get_C100()`, *and* `cla_utils.exercises9.get_D100()`. *You can use* `matplotlib.pyplot.pcolor()` *to visualise the entries, or compute norms of the components of the matrices below the diagonal, for example. What do you observe? How does this relate to the structure of the four matrices?*

The algorithm simply finds the QR factorisation of A, swaps Q and R, and repeats. We call this algorithm the "pure" QR algorithm, since it can be accelerated with some modifications that comprise the "practical" QR algorithm that is used in practice.

We can at least see that this is computing similarity transformations since

$$A^{(k)} = R^{(k)} Q^{(k)} = (Q^{(k)})^* Q^{(k)} R^{(k)} Q^{(k)} = (Q^{(k)})^* A^{(k-1)} Q^{(k)},$$

so that $A^{(k)}$ is similar to $A^{(k-1)}$ and hence to $A^{(k-2)}$ and all the way back to A. But why does $A^{(k)}$ converge to a diagonal matrix? To see this, we have to show that the QR algorithm is equivalent to another algorithm called simultaneous iteration.

5.9 Simultaneous iteration

One problem with power iteration is that it only finds one eigenvector/eigenvalue pair at a time. Simultaneous iteration is a solution to this. The starting idea is simple: instead of working on just one vector v, we pick a set of linearly independent vectors $v_1^0, v_2^0, \ldots, v_n^0$ and repeatedly apply A to each of these vectors. After a large number applications and normalisations in the manner of the power iteration, we end up with a linear independent set $v_1^k, v_2^k, \ldots, v_n^k$, $n \leq m$. All of the vectors in this set will be very close to q_1, the eigenvector with largest magnitude of corresponding eigenvalue. We can choose v_1^k as our approximation of q_1, and project this approximation of q_1 from the rest of the vectors $v_2^k, v_3^k, \ldots v_m^k$. All the remaining vectors will be close to q_2, the eigenvector with the next largest magnitude of corresponding eigenvalue. Similarly we can choose the first one of the remaining projected vectors as an approximation of q_2 and project it again from the rest.

We can translate this idea to matrices by defining $V^{(0)}$ to be the matrix with columns given by the set of initial v applications of A, we have $V^{(k)} = A^k V^{(0)}$. By the column space interpretation of matrix-matrix multiplication, each column of $V^{(k)}$ is A^k multiplied by the corresponding column of $V^{(0)}$. To make the normalisation and projection process above, we could just apply the Gram-Schmidt algorithm, sequentially forming an orthonormal spanning set for the columns of $V^{(k)}$ working from left to right. However, we know that an equivalent way to do this is to form the (reduced) QR factorisation of $V^{(k)}$, $\hat{Q}^{(k)}\hat{R}^{(k)} = V^{(k)}$; the columns of $\hat{Q}^{(k)}$ give the same orthonormal spanning set. Hence, the columns of $\hat{Q}^{(k)}$ will converge to eigenvectors of A, provided that:

1. The first n eigenvalues of A are distinct in absolute value: $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$. If we want to find all of the eigenvalues $n = m$, then all the absolute values of the eigenvalues must be distinct.
2. The v vectors can be expressed as a linear sum of the first n eigenvectors $q_1, \ldots, q_n$ in a non-degenerate way. This turns out to be equivalent (we won't show it here) to the condition that $\hat{Q}^T V^{(0)}$ has an LU factorisation (where $\hat{Q}$ is the matrix whose columns are the first n eigenvectors of A).

One problem with this idea is that it is not numerically stable. The columns of $V^{(k)}$ rapidly become a very ill-conditioned basis for the spanning space of the original independent set, and the values of eigenvectors will be quickly engulfed in rounding errors. There is a simple solution to this though, which is to orthogonalise after each application of A. This is the simultaneous iteration algorithm, which we express in the following pseudo-code.

- TAKE A UNITARY MATRIX $\hat{Q}^{(0)}$
- FOR $k = 1, 2, \ldots$
  - $Z \leftarrow A\hat{Q}^{(k-1)}$
  - FIND $Q^{(k)}, R^{(k)}$ such that $Q^{(k)}R^{(k)} = Z$ (USING QR FACTORISATION)

This is mathematically equivalent to the process we described above, and so it converges under the same two conditions listed above.

We can already see that this looks rather close to the QR algorithm. The following section confirms that they are in fact equivalent.

5.10 The pure QR algorithm and simultaneous iteration are equivalent

To be precise, we will show that the pure QR algorithm is equivalent to simultaneous iteration when the initial independent set is the canonical basis I, i.e. $Q^{(0)} = I$. From the above, we see that that algorithm converges provided that Q^T has an LU decomposition, where Q is the limiting unitary matrix that simultaneous iteration is converging to. To show that the two algorithms are equivalent, we append them with some auxiliary variables, which are not needed for the algorithms but are needed for the comparison.

To simultaneous iteration we append a running similarity transformation of A, and a running product of all of the R matrices.

- $Q'^{(0)} \leftarrow I$
- FOR $k = 1, 2, \ldots$

  - $Z \leftarrow AQ'^{(k-1)}$
  - FIND $Q'^{(k)}, R^{(k)}$ such that $Q'^{(k)}R^{(k)} = Z$ (USING QR FACTORISATION)
  - $A^{(k)} = (Q'^{(k)})^T AQ'^{(k)}$
  - $R'^{(k)} = R^{(k)}R^{(k-1)} \dots R^{(1)}$

To the pure QR factorisation we append a running product of the Q^k matrices, and a running product of all of the R matrices (again).

- $A^{(0)} \leftarrow A$
- FOR $k = 1, 2, \dots$
  - FIND $Q^{(k)}, R^{(k)}$ such that $Q^{(k)}R^{(k)} = A^{(k-1)}$ (USING QR FACTORISATION)
  - $A^{(k)} = R^{(k)}Q^{(k)}$
  - $Q'^{(k)} = Q^{(1)}Q^{(2)} \dots Q^{(k)}$
  - $R'^{(k)} = R^{(k)}R^{(k-1)} \dots R^{(1)}$

Theorem 116 (pure QR and simultaneous iteration with I are equivalent) *The two processes above generate identical sequences of matrices $R'^{(k)}$, $Q'^{(k)}$ and $A^{(k)}$, which are related by $A^k = Q'^{(k)}R'^{(k)}$ (the k-th power of A, not $A^{(k)}$!), and $A^{(k)} = (Q'^{(k)})^T AQ'^{(k)}$.*

Proof 117 *We prove by induction. At $k = 0$, $A_k = R'^{(k)} = Q'^{(k)} = 0$. Now we assume that the inductive hypothesis is true for k, and aim to deduce that it is true for $k + 1$.*

For simultaneous iteration, we immediately have the simularity formula for $A^{(k)}$ by definition, and we just need to verify the QR factorisation of A^k. From the inductive hypothesis,

$$A^k = AA^{k-1} = AQ'^{(k-1)}R'^{(k-1)} = ZR'^{(k-1)} = Q'^{(k)}\underbrace{R^{(k)}R'^{(k-1)}}_{=R'^{(k)}} = Q'^{(k)}R'^{(k)},$$

as required (using the definition of Z and then the definition of $R'^{(k)}$).

For the QR algorithm, we again use the inductive hypothesis on the QR factorization of A^k followed by the inductive hypothesis on the similarity transform to get

$$A^k = AA^{k-1} = AQ'^{(k-1)}R'^{(k-1)}Q'^{(k-1)}A^{(k-1)}R'^{(k-1)} = Q'^{(k-1)}Q^{(k)}R^{(k)}R'^{(k-1)} = Q'^{(k)}R'^{(k)},$$

where we used the algorithm definitions in the third equality and then the definitions of $Q'^{(k)}$ and $R'^{(k)}$. To verify the similarity transform at iteration k we use the algorithm definitions to write

$$A^{(k)} = R^{(k)}Q^{(k)} = (Q^{(k)})^T R^{(k)}Q^{(k)} = (Q'^{(k)})^T A(Q')^{(k)},$$

as required.

This theorem tells us that the QR algorithm will converge under the conditions that simultaneous iteration converges. It also tells us that the QR algorithm finds an orthonormal basis (the columns of $Q'^{(k)}$) from the columns of each power of A^k; this is how it relates to power iteration.

5.11 The practical QR algorithm

The practical QR algorithm for real symmetric matrices has a number of extra elements that make it fast. First, recall that we start by transforming to tridiagonal (symmetric Hessenberg) form. This cuts down the numerical cost of the steps of the QR algorithm. Second, the Rayleigh quotient algorithm idea is incorporated by applying shifts $A^{(k)}-\mu^{(k)}I$, where $\mu^{(k)}$ is some eigenvalue estimate. Third, when an eigenvalue is found (i.e. an eigenvalue appears accurately on the diagonal of $A^{(k)}$) the off-diagonal components are very small, and the matrix decouples into a block diagonal matrix where the QR algorithm can be independently applied to the blocks (which is cheaper than doing them all together). This final idea is called deflation.

A sketch of the practical QR algorithm is as follows.

- $A^{(0)} \leftarrow$ TRIDIAGONAL MATRIX
- FOR $k = 1, 2, \ldots$
  - PICK A SHIFT $\mu^{(k)}$ (discussed below)
  - $Q^{(k)}R^{(k)} = A^{(k-1)} - \mu^{(k)}I$ (from QR factorisation)
  - $A^{(k)} = R^{(k)}Q^{(k)} + \mu^{(k)}I$
  - IF $A^{(k)}_{j,j+1} \approx 0$ FOR SOME j
    - $A_{j,j+1} \leftarrow 0$
    - $A_{j+1,j} \leftarrow 0$
    - continue by applying the practical QR algorithm to the diagonal blocks A_1 and A_2 of

$$A_k = \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}$$

One possible way to select the shift $\mu^{(k)}$ is to calculate a Rayleigh quotient with A using the last column $q^{(k)}_m$ of $Q'^{(k)}$, which then gives cubic convergence for this eigenvector and eigenvalue. In fact, this is just A^k_{mm},

$$A^{(k)}_{mm} = e_m^T A^{(k)} e_m = e_m^T (Q'^{(k)})^T A Q'^{(k)} e_m = (q^{(k)}_m)^T A q_m = \mu^{(k)}.$$

This is very cheap, we just read off the bottom right-hand corner from $A^{(k)}$! This is called the Rayleigh quotient shift.

It turns out that the Rayleigh quotient shift is not guaranteed to work in all cases, so there is an alernative approach called the Wilkinson shift, but we won't discuss that here.

CHAPTER

SIX

ITERATIVE KRYLOV METHODS FOR $AX = B$

In the previous section we saw how iterative methods are necessary (but can also be fast) for eigenvalue problems $Ax = \lambda x$. Iterative methods can also be useful for solving linear systems $Ax = b$, generating a sequence of vectors x^k that converge to the solution. We shall examine Krylov subspace methods, where each iteration mainly involves a matrix-vector multiplication. For dense matrices, matrix-vector multiplication costs $\mathcal{O}(m^2)$, but often (e.g. numerical solution of PDEs, graph problems, etc.) A is sparse (i.e. has zeros almost everywhere) and the matrix-vector multiplication costs $\mathcal{O}(m)$. The goal is then to find a method where only a few iterations are necessary before the error is very small, so that the solver has total cost $\mathcal{O}(mN)$ where N is the total number of iterations, hopefully small.

Since we only need the result of a matrix-vector multiplication, it is even possible to solve a linear system without storing A explicitly. Instead one can just provide a subroutine that implements matrix-vector multiplication in some way; this is called a "matrix-free" implementation.

6.1 Krylov subspace methods

In this section we will introduce Krylov subspace methods for solving $Ax = b$ (we will not specialise to real or symmetric matrices here). The idea is to approximate the solution using the basis

$$(b, Ab, A^2b, A^3b, \ldots, A^kb)$$

whose span is called a Krylov subspace. After each iteration the Krylov subspace grows by one dimension. As we have already seen from studying power iteration, the later elements in this sequence will get very parallel (they will all be approximating the eigenvector with largest magnitude of eigenvalue). Hence, we once again need to resort to orthogonalising the basis. We could simply take the QR factorisation of this basis, but the Arnoldi iteration coming up next also provides a neat way to solve the equation when projected onto the Krylov subspace.

6.2 Arnoldi iteration

The key to Krylov subspace methods turns out to be the transformation of A to an upper Hessenberg matrix by orthogonal similarity transforms, so that $A = QHQ^*$. We have already looked at using Householder transformations to do this in the previous section. The Householder technique is not so suitable for large dimensional problems, so instead we look at a way of proceeding column by column, just like the Gram-Schmidt method does for finding QR factorisations.

We do this by rewriting $AQ = QH$. The idea is that at iteration n we only look at the first n columns of Q, which we call $\hat{Q}_n$. When m is large, this is a significant saving: $mn \ll m^2$. To execute the iteration, it turns out that we should look at the $(n+1) \times n$ upper left-hand section of H, i.e.

$$\tilde{H}_n = \begin{pmatrix} h_{11} & \ldots & h_{1n} \\ h_{21} & \ddots & \vdots \\ 0 & \ddots & h_{nn} \\ 0 & 0 & h_{n+1,n} \end{pmatrix}$$

Then, $A\hat{Q}_n = \hat{Q}_{n-1}\tilde{H}_n$. Using the column space interpretation of matrix-matrix multiplication, we see that the n-th column is

$$Aq_n = h_{1n}q_1 + h_{2n}q_n + \ldots h_{n,n}q_n + h_{n+1,n}q_{n+1}.$$

This formula shows us how to construct the non-zero entries of the nth column of H; this defines the Arnoldi algorithm which we provide as pseudo-code below.

- $q_1 \leftarrow b/\|b\|$
- FOR $n = 1, 2, \ldots$
  - $v \leftarrow Aq_n$
  - FOR $j = 1$ TO n
    - $h_{jn} \leftarrow q_j^* v$
    - $v \leftarrow v - h_{jn}q_j$
  - END FOR
  - $h_{n+1,n} \leftarrow \|v\|$
  - $q_{n+1} \leftarrow v/\|v\|$
- END FOR

Exercise 118 *The* `cla_utils.exercises10.arnoldi()` *function has been left unimplemented. It should implement the Arnoldi algorithm using one loop over the iteration count, and return the Q and H matrices after the requested number of iterations is complete. The test script* `test_exercises10.py` *in the* `test` *directory will test this function.*

If we were to form the QR factorisation of the $m \times n$ Krylov matrix

$$K_n = \begin{pmatrix} b & Ab & \ldots & A^{n_1}b \end{pmatrix}$$

then we would get $Q = Q_n$. Importantly, in the Arnoldi iteration, we never form K_n or R_n explicitly, since these are very ill-conditioned and not useful numerically.

But what is the use of the $\tilde{H}_n$ matrix? Applying $\hat{Q}_n^*$ to $A\hat{Q}_n = \hat{Q}_{n+1}\tilde{H}_n$ gives

$$\begin{aligned} \hat{Q}_n^* A \hat{Q}_n &= \hat{Q}_n^* \hat{Q}_{n+1} \tilde{H}_n, \\ &= \begin{pmatrix} 1 & 0 & \ldots & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \ldots & \ldots & 1 & 0 \end{pmatrix} \tilde{H}_n = H_n, \end{aligned}$$

where H_n is the $n \times n$ top left-hand corner of H.

The intrepretation of this is that H_n is the orthogonal projection of A onto the Krylov subspace K_n. To see this, take any vector v, and project Av onto the the Krylov subspace K_n.

$$PAv = \hat{Q}_n \hat{Q}_n^* v.$$

Then, changing basis to the orthogonal basis gives

$$\hat{Q}_n^*(\hat{Q}_n \hat{Q}_n^* A)\hat{Q}_n = \hat{Q}_n^* A \hat{Q}_n = H_n.$$

6.3 GMRES

The Generalised Minimum Residual method (GMRES), due to Saad (1986), exploits these properties of the Arnoldi iteration. The idea is that we build up the orthogonal basis for the Krylov subspaces one by one, and at each iteration we solve the projection of $Ax = b$ onto the Krylov basis as a least squares problem, until the residual $\|Ax - b\|$ is below some desired tolerance.

To avoid the numerical instabilities that would come from using the basis $(b, Ab, A^2b, \ldots)$, we use the Arnoldi iteration to build an orthonormal basis, and seek approximate solutions of the form $x_n = \hat{Q}_n y$ for $y \in \mathbb{C}^n$. We then seek the value of y that minimises the residual

mathcal{R}_n = |AQ_ny - b|.

This explains the Minimum Residual part of the name. We also see from this definition that the residual cannot increase with iterations, because it only increases the subspace where we seek a solution.

This problem can be simplified further by using $AQ_n = Q_{n+1}\tilde{H}_n$, so

$$\mathcal{R}_n = \|\hat{Q}_{n+1}\tilde{H}_n y - b\|.$$

Remembering that $b = \|b\|q_1$, we see that the entire residual is in the column space of $\hat{Q}_{n+1}$, and hence we can invert on the column space using $\hat{Q}^*_{n+1}$ which does not change the norm of the residual due to the orthonormality.

$$\mathcal{R}_n = \|\tilde{H}_n y - \hat{Q}^*_{n+1} b\| = \|\tilde{H}_n y - e_1\|b\|\|.$$

Finding y to minimise $\mathcal{R}_n$ requires the solution of a least squares problem, which can be computed via QR factorisation as we saw much earlier in the course.

We are now in position to present the GMRES algorithm as pseudo-code.

- $q_1 \leftarrow b/\|b\|$
- FOR $n = 1, 2, \ldots$
  - APPLY STEP n OF ARNOLDI
  - FIND y TO MINIMIZE $\|\tilde{H}_n y - \|b\|e_1\|$
  - $x_n \leftarrow \hat{Q}_n y$
  - CHECK IF $\mathcal{R}_n <$ TOL
- END FOR

Exercise 119 *The* `cla_utils.exercises10.GMRES()` *function has been left unimplemented. It should implement the basic GMRES algorithm above, using one loop over the iteration count. You can paste code from your* `cla_utils.exercises10.arnoldi()` *implementation, and you should use your least squares code to solve the least squares problem. The test script* `test_exercises10.py` *in the* `test` *directory will test this function.*

Exercise 120 *The least squares problem in GMRES requires the QR factorisation of H_k. It is wasteful to rebuild this from scratch given that we just computed the QR factorisation of H_{k-1}. Modify your code so that it recycles the QR factorisation, applying just one extra Householder rotation per GMRES iteration. Don't forget to check that it still passes the test.*

Hint: Don't get confused by the two Q matrices involved in GMRES! There is the Q matrix for the Arnoldi iteration, and the Q matrix for the least squares problem. These are not the same.

Exercise 121 *We can also make GMRES more efficient by exploiting the upper Hessenberg structure, since we only have one non-zero value below the diagonal in each column. Instead of using a Householder transformation, we can use a Givens rotation, which only alters two rows (the row corresponding to the diagonal, and the row below). The Givens rotation simply replaces these two rows (call them r_k and r_{k+1}) by*

$$\begin{aligned}\hat{r}_k &= \cos(\theta) r_k + \sin(\theta) r_{k+1},\\ \hat{r}_{k+1} &= -\sin(\theta) r_k + \cos(\theta) r_{k+1}.\end{aligned}$$

where the angle θ is chosen so that $\hat{r}_{k+1,k} = 0$. This is cheaper, because we only update two rows, but still corresponds to a unitary transformation. (Note that a slightly more general formula is required for complex matrices, but the tests have been set up for real matrices only.) Modify your code so it uses Givens rotations to make it more efficient. Don't forget to check that it still passes the test.

6.4 Convergence of GMRES

The algorithm presented as pseudocode is the way to implement GMRES efficiently. However, we can make an alternative formulation of GMRES using matrix polynomials.

We know that $x_n \in K_n$, so we can write

$$x_n = c_0 b + c_1 A b + c_2 A^2 b + \ldots + c_{n-1} A^{n-1} b = p'(A) b,$$

where

$$p'(z) = c_0 + c_1 z + c_2 z^{@} + \ldots + \ldots c_{n-1} z^{n-1} \implies p'(A) = c_0 I + c_1 A + c_2 A^2 + \ldots + c_{n-1} A^{n-1}.$$

Here we have introduced the idea of a matrix polynomial, where the kth power of z is replaced by the kth power of A.

The residual becomes

$$r_n = b - A x_n = b - A p'(A) b = (I - A p'(A)) b = p(A) b,$$

where $p(z) = 1 - z p'(z)$. Thus, the residual is a matrix polynomial p of A applied to b, where $p \in \mathcal{P}_n$, and

$$\mathcal{P}_n = \{\text{degree } \leq n \text{ polynomials with } p(0) = 1\}.$$

Hence, we can recast iteration n of GMRES as a polynomial optimisation problem: find $p_n \in \mathcal{P}_n$ such that $\|p_n(A) b\|$ is minimised. We have

$$\|r_n\| = \|p_n(A) b\| \leq \|p_n(A)\| \|b\| \implies \frac{\|r_n\|}{\|b\|} \leq \|p_n(A)\|.$$

Assuming that A is diagonalisable, $A = V \Lambda V^{-1}$, then $A^s = V \Lambda^s V^{-1}$, and

$$\|p_n(A)\| = \|V p_n(\Lambda^s) V^{-1}\| \leq \underbrace{\|V\| \|V^{-1}\|}_{=\kappa(V)} \|p_n\|_{\Lambda(A)},$$

where $\Lambda(A)$ is the set of eigenvalues of A, and

$$\|p\|_{\Lambda(A)} = \sup_{x \in \Lambda(A)} \|p(x)\|.$$

Hence we see that GMRES will converge quickly if V is well-conditioned, and $p(x)$ is small for all $x \in \Lambda(A)$. This latter condition is not trivial due to the $p(0) = 1$ requirement. One way it can happen is if A has all eigenvalues clustered in a small number of groups. Then we can find a low degree polynomial that passes through 1 at $x = 0$, and 0 near each of the clusters. Then GMRES will essentially converge in a small number of iterations (equal to the degree of the polynomial). There are problems if the eigenvalues are scattered over a wide region of the complex plane: we need a very high degree polynomial to make $p(x)$ small at all the eigenvalues and hence we need a very large number of iterations.

Exercise 122 *Investigate the convergence of the matrices provided by the functions* `cla_utils.exercises10.get_AA100()`, `cla_utils.exercises10.get_BB100()`, *and* `cla_utils.exercises10.get_CC100()`, *by looking at the residual after each iteration (which should be provided by* `cla_utils.exercises10.GMRES()` *as specified in the docstring). What do you observe? What is it about the three matrices that causes this different behaviour?*

6.5 Preconditioning

This final topic has been a real focus of computational linear algebra over the last 30 years. Typically, the matrices that we want to solve do not have eigenvalues clustered in a small number of groups, and so GMRES is slow. The solution (and the challenge) is to find a matrix M such that $Mx = y$ is cheap to solve (diagonal, or triangular, or something else) and such that $M^{-1}A$ *does* have eigenvalues clustered in a small number of groups (e.g. M is a good approximation of A, so that $M^{-1}A \approx I$ which has eigenvalues all equal to 1). We call M the preconditioning matrix, and the idea is to apply GMRES to the (left) preconditioned system

$$M^{-1}Ax = M^{-1}b.$$

GMRES on this preconditioned system is equivalent to the following algorithm, called preconditioned GMRES.

- SOLVE $M\tilde{b} = b$.
- $q_1 \leftarrow \tilde{b}/\|\tilde{b}\|$
- FOR $n = 1, 2, \ldots$
  - SOLVE $Mv = Aq_n$
  - FOR $j = 1$ TO n
    - $h_{jn} = q_j^* v$
    - $v = v - h_{jn} q_j$
  - END FOR
  - $h_{n+1,n} \leftarrow \|v\|$
  - q_{n+1}gets v/|v|`
  - FIND y TO MINIMIZE $\|\tilde{H}_n y - \|\tilde{b}\| e_1\|$
  - $x_n \leftarrow \hat{Q}_n y$
  - CHECK IF $\mathcal{R}_n <$ TOL
- END FOR

Exercise 123 *Show that this algorithm is equivalent to GMRES applied to the preconditioned system.*

The art and science of finding preconditioning matrices M (or matrix-free procedures for solving $Mx = y$) for specific problems arising in data science, engineering, physics, biology etc. can involve ideas from linear algebra, functional analysis, asymptotics, physics, etc., and represents a major activity in scientific computing today.

PYTHON MODULE INDEX

C