MATH60024 Computational Linear Algebra
Commentary Notes

Feifan Fan

based on lecture contents
by Professor Colin Cotter

Autumn 2021-2022

Declaration

This so-called "commentary notes" is based on lecture notes of MATH60024, Computational Linear Algebra, which is led and lectured by Professor Colin Cotter. The original notes could be found here. I would call it "master notes" most of the time in the following comentary.

I wrote this set of notes to explain several vague points in the master notes, make detailed proofs about theorems which is not being explained clearly, and give step-by-step instructions for doing the coding exercise.

I also gave out something everyone always want to have - the sample solutions of coding exercises. Actually it is not typical to distribute sample solutions of exercises of a coding-based module. But my sample solutions would be aimed at helping you have a deeper understaning of the problem description and knowledge itself, and also let you to understand the implementations from scratch (I hope so :-D).

There is no need to worry if you think you didn't do the Priciples of Programming module in year 2. The module is beneficial but not necessary. What you need to do is just giving yourself enough coding practice via attempting the exercises in this module (and read my explanation if you get stucked!). Programming is the art of craftmenship, and dedicated practice would lead you to perfection.

You might find that I would spend several of time in proving theorems, although this module is totally "computational". The reason I did this is just to make sure everything could be demonstrated clearly. If you are not happy with reading proofs, feel free to skip them!

This module would be a great adventure, hope you enjoy and have lots of fun in this journey!

Acknowledgements

I want to thank Professor Colin Cotter for providing such a brilliant set of open-sourced master notes first. Without his extraodinary work, I cannot digest the knowlege in such a quick and enjoyable way, and come up with the idea of writing these "commentary notes". I hope my work would help him and his students studying this module in the following years.

Also, thanks a lot to my friends Kexin, Tiansheng and Di (pronounced as **Dee**) being the first group of readers of the notes. Their participation of reading notes and using this as additional studying material gives me extra motivation in fixing errors in sample solutions of the code, and keeping me dedicated in updating these commentary notes on track.

SPACES LEFT FOR FAMILY AND OTHER FRIENDS

Contents

List of Figures

Chapter 1

Preliminaries

1.1 Matrix-Vector Multiplication

This part might be too easy for most of you - but it would still give you intuition in the contents we would discuss later. From the master notes we are given that:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \text{ and } x \in \mathbb{C}^n.$$

and

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

And we are asked to consider the matrix multiplication $b = Ax$:

- The first thing we could confirm is $b \in \mathbb{C}^m$ by dimensions of the given matrices.
- Then speaking about the ith component of b, b_i (**obviously,** $1 \leq i \leq m$), how do we determine the value of b_i in general?
- Normally we get the ith row from our matrix A, and multiply with the column vector x using matrix multiplication rules:

$$b_i = a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = \sum_{j=1}^{n} a_{ij}x_j.$$

where $i \in \{1, 2, 3 \dots m\}$.

- This is what we called Matrix-Vector multiplication. And as the master notes saying, we could check the map $x \to Ax$ given by matrix multiplication is a linear map from $\mathbb{C}^n \to \mathbb{C}^m$. We need to show that:

$$\forall x, y \in \mathbb{C}^n, \alpha \in \mathbb{C}, A(\alpha x + y) = \alpha Ax + Ay.$$

- This should be really straightforward.

Proof

- Suppose $b = A(\alpha x + y)$, we compute the ith entry of b, $b_i = [A(\alpha x + y)]_i$ via the formula above:

$$b_i = \sum_{j=1}^{n} a_{ij}(\alpha x + y)_j = \sum_{j=1}^{n} a_{ij}(\alpha x_j + y_j)$$

$$= \alpha(\sum_{j=1}^{n} a_{ij}x_j) + (\sum_{j=1}^{n} a_{ij}y_j).$$

- Can you see that how could we simplify this expression further? Use the Matrix-Vector formula we derived before:

$$b_i = \alpha(Ax)_i + (Ay)_i$$

- Therefore, sub in $b_i = [A(\alpha x + y)]_i$ into the equation we have:

$$[A(\alpha x + y)]_i = \alpha(Ax)_i + (Ay)_i$$

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

- Therefore, we could now confirm that the map $x \to Ax$ by multiplication is a linear map.

- You have now seen the derivation of Matrix-Vector Multiplication formula and its property, but how do we implement this schema as a python function? Try Exercise 1.1!

Exercise 1.1: Basic M-V Multiplication

Problem Description

In this exercise, we are given matrix A and vector x as defined above. What we need to is implementing a function which returns the matrix-vector product $b = Ax$. We are asked to implement this with a **double-nested loop**.
How do we implement this function?

- Since this is the very first exercise, I would show you the mathematics behind the implementation, the steps you need to do in the implementation, and how to translate steps into code, in details.
- **Note: Detailed explanation/analysis about how to write code would only be in this exercise, but would not be provided in any future exercises.**

What to do

1. Think about the formula we derived in the previous section:

$$b_i = \sum_{j=1}^{n} a_{ij}x_j, i \in \{1, 2, 3 \dots m\}.$$

2. There is a summation sign in the caclulation of b_i, and it should indicate a **for** loop in the implementation of calculating b_i.
3. **Note: We always relate the summation in mathematics and for loop in programming together, try to convince yourself that you really understand why we were allowed to do so.**
4. Also the basic 'algorithm' indicates that we would compute the value of b_i for m times, with similar methods. Therefore, there should be another **for** loop to compute all values of b_i in m iterations. Now you should understand why a **double-nested for loop** is necessary for our implementation.

- Therefore, the structure of our implementation should be:

```python
import numpy as np
def basic_matvec(A, x):
    """
    Elementary matrix-vector multiplication.
    ...
    :return b: m-dimensional numpy array
    """
    b = []
    for i in range(m):
        # Compute the value of b_i.
        for j in range(n):
            # Apply the basic formula in summation.
        # Add the computed b_i value to b.
    return np.array(b)
```

Analysis

- Then the implementation has been divided into two parts: compute the values of b_i through the inner loop iterations, and add these values to the placeholder, a list b as each inner loop ends.

- The second part is very easy to implement, just append the computed b_i to b via `b.append(bi)` at the end of each iteration of the outer loop.

- The first part is still straight-forward, but would take more effort in the implementation. To compute the value of b_i, we should initialize the value as 0 at first, i.e. `bi = 0`. Then we could go into the inner loop with counter j to do the summation.

- Then the 'elements' we need in the inner loop is kind of self-intuitive, since it is mentioned in the basic formula:

$$\sum_{i=1}^{n} \rightarrow \texttt{for j in range(n)}$$

$$a_{ij} \rightarrow \texttt{A[i, j]}$$

$$x_j \rightarrow \texttt{x[j]}.$$

- Since we are doing summation, what we are going to do in the inner loop is just to keep adding computed results during iterations of j, i.e. `bi += A[i][j] * x[j]` in each iteration of j.

- The function requires an n-dimentional `ndarray`, so we need to wrap the list `b` with `np.array()` when function returns the result.

Code Implementation

- Therefore, we have all things we need from analysis above, and fill the template we would get the final implementation:

```
def basic_matvec(A, x):
    """
    Elementary matrix-vector multiplication.
    ...
    :return b: m-dimensional numpy array
    """
    b = []
    m, n = A.shape
    for i in range(m):
        # Compute the value of b_i.
        bi = 0
        for j in range(n):
            # Apply the basic formula in summation.
            bi += A[i, j] * x[j]
        # Add the computed b_i value to b.
        b.append(bi)
    return np.array(b)
```

- **(Important!)** To test your code, do not forget to activate your virtual environment first, and then using either `pytest` or `py.test` to do the testing.

1.2 Column-Space Formulation

In this section, we would give you an intuition to interpret matrix-vector multiplication as a linear combination of columns of A, with coefficients taken from the entries of x.

- If we write the result of $b = Ax$ explicitly, we would get:

$$b = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{pmatrix}$$

$$= x_1 \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} + x_2 \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} + \cdots + x_n \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix}.$$

which is a linear combination with columns of A and entries of x.

- To simplify this more, we could rewrite matrix A in terms of all of its columns as follows:

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

where

$$a_i = \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{mi} \end{pmatrix} \in \mathbb{C}^m, i \in \{1, 2, \dots n\}.$$

- Therefore, the linear combination should be:

$$b = x_1 a_1 + x_2 a_2 + \cdots + x_n a_n = \sum_{j=1}^{n} x_j a_j.$$

Exercise 1.2: M-V Multiplication with C-S Formulation

Problem Description

The question says the given conditions are unchanged. However, in this implementation we were asked to **use a single loop** over entries of A (and x).

So considering the column-space formulation we mentioned above, we could easily get the implementation by using:

$$b = Ax = \sum_{i=1}^{n} x_i a_i.$$

Code Implemenation

```
def column_matvec(A, x):
    """
    Matrix-vector multiplication using the representation of
   the product.
    ...
    :return b: an m-dimensional numpy array which is the
   product of A with x
    """
    b = np.zeros(len(A))
    for i, elem in enumerate(x):
        # Add the products up as a vector sum to the result.
        b += A[:, i] * elem
    return b
```

Analysis

Here are several things also worth mentioning:

- Since we are asked to use a **single loop** only, we could easily see that using the formula with one Σ sign would work. i.e. the formula with column-space formulation.
- Then according to the formula derivation above, we only need to iterate through the column vectors in matrix A, i.e. `A[:, i]` and multiply with the ith entry of x, the final result would be calculated as the sum of products in each iterations.
- The placeholder of the result, `b`, is initialized as a `numpy` array with zeros, rather than an empty array.
- That is because in the addition of numpy, two arrays should be equi-dimentioned. Using an empty array would make the dimension zero, which is obviously different from the dimension of product obtained in each iteration.
- I used `for i, elem in enumerate(x)` in my implementation, rather than `for i in range(len(x))` and `elem = x[i]` for code simplicity and elegance.

1.3 Matrix-Matrix Multiplication

This part of derivation is similar to the Matrix-Vector case. Now our matrix A is in dimension $\mathbb{C}^{m\times l}$, and consider another matrix $C \in \mathbb{C}^{l\times n}$ and we have matrix $B = AC$:

- Then we think about the entry (i, j) of matrix B. To get the value of b_{ij}, we should get the ith row from A and the jth column from C, and the calculation goes like this:

  $$b_{ij} = a_{i1}c_{1j} + a_{i2}c_{2j} + a_{i3}c_{3j} + \ldots + a_{il}c_{lj} = \sum_{k=1}^{l} a_{ik}c_{kj}.$$

  where

  $$(i, j) \in \{1, 2, 3 \ldots m\} \times \{1, 2, 3 \ldots n\}.$$

- Now we have seen the derivation of formula of matrix multiplication, in different cases.

- Then try to apply the column-space formulation to the matrix-matrix multiplication, considering the ith column of the result matrix, b_i, the column vector is obtained via:

$$b_i = Ac_i.$$

  which is another product of matrix-vector multiplication.

- Therefore, we expand the matrix-vector multiplication and we could get:

$$b_i = \sum_{j=1}^{l} A_j c_{ij}.$$

  for a matrix-matrix multiplication with column-space formulation.

Exercise 1.3: Execution Time Comparison

Problem Description

In this section we are not asked to implement anything, but we could use the defined function `time_matvecs()` in package to see the different execution time, for different matrix multiplication algorithm implementations:

```
> basic_matvec(A0, x0) # double-nested loop implementation

> column_matvec(A0, x0) # single loop implementation

> A0.dot(x0) # numpy implementation
```

Efficiency Measuring

Here we compare three implementations mentioned above, and the timing function gives us the following results:

```
>>> import cla_utils
>>> cla_utils.time_matvecs()
Timing for basic_matvec
0.09208189499999975
Timing for column_matvec
0.002095079000000055
Timing for numpy matvec
0.00014622799999841618
```

We would clearly see the execution time:

$$\texttt{nested loop} \gg \texttt{single loop} \gg \texttt{numpy implementation}.$$

As the master notes saying, 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).

1.4 Range of a Matrix

The master notes have the definition of `range`, here I would give the definition in set notation **(alternative version of Definition 1.4 master notes)** :

$$\text{range}(A) = \{a : \exists x \in \mathbb{C}^n, Ax = a\}, \text{ where } A \in \mathbb{C}^{m\times n}.$$

Here we would see any $a \in A$ would be expressed as a product in form of M-V multiplication. In other words, we could manipulate a as a product with column-space formulation, with the following interpretation:

$$a = Ax = \sum_{i=1}^{n} A_i x_i.$$

Therefore, any element in range(A) could be demonstrated as a **linear combination** of **column vectors** in A. In other words, **any element** $a \in$ **range**(A) **would be in the vector space spanned by column vectors of** A**.** So we have the following statement:

$$\forall a \in \text{range}(A), a \in \text{span}(a_1, \cdots, a_n) \implies \text{range}(A) \subseteq \text{span}(a_1, \cdots, a_n).$$

Also we could see that, **any linear combination** of columns of A is an element of range(A), which means:

$$\forall v \in \text{span}(a_1, \cdots, a_n), v \in \text{range}(A) \implies \text{span}(a_1, \cdots, a_n) \subseteq \text{range}(A).$$

Therefore, we finally have:

$$\text{range}(A) = \text{span}(a_1, \cdots, a_n).$$

That is to say, range(A) is the vector space spanned by the columns of A, as mentioned in the **Theorem 1.5** of master notes.

1.5 Null Space and Rank of a Matrix

The definition of the null space of an Matrix is very simple as stated by the master notes:

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

Then we would talk about the rank of a matrix. Actually we have two definitions, **row rank** and **column rank**, which refer to the dimension of the row/column space. It could be shown that these two values are equal for all matrices, here is the proof:

Proof

- Suppose we have a matrix $A \in \mathbb{C}^{m\times n}$, and we could put this matrix into its RRE form, A', by elementary row operations (an example is given below, where $a'_{in} \neq 1$):

$$A' = \begin{pmatrix} 1 & a'_{11} & 0 & a'_{14} & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & a'_{23} & \cdots & 0 & a'_{2n} & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 & a'_{3n} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

- Similarly, we could put A' into its column reduced form, A''. Since the elementary row and column operations are both invertible. That is to say, there exists two invertible matrices P and Q, such that:

$$QPA = A''.$$

  and since P and Q are invertible, the number of linearly independent row (**also for column**) vectors in A'' should be equal to the number in A.

- Now A'' is in both row reduced and column reduced form, so A'' should be in the following form:

$$A'' = \begin{pmatrix} I_k & 0_{k\times(n-k)} \\ 0_{(m-k)\times k} & 0_{(m-k)\times(n-k)} \end{pmatrix}.$$

- To be more specific, we could see the structure of A'' after the elementary row and column operations through the following figure:

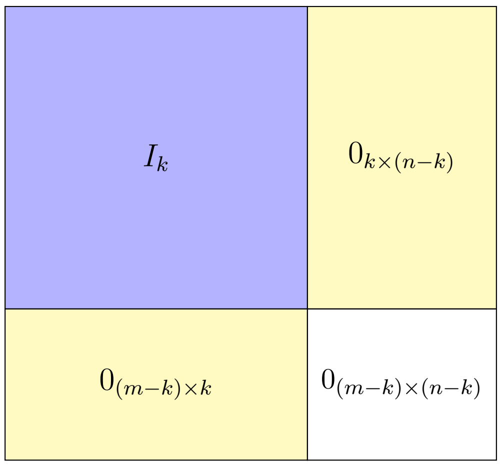


Figure 1.1: Structure of matrix after elementary row and column operations

- Then it is not hard to see that, since apart from the identity matrix block I_k, the other blocks in A'' are just 0s, so the dimension of row space and column space are both equal to the 'size' of the identity matrix block k. i.e. the row rank and the column rank are equal for all matrices.

Here we have seen the equality of row and column ranks, so as the mater notes saying, we just refer to the rank in this course. By using the definition of range and column space of matrix, for all matrix $A \in \mathbb{C}^{m\times n}$, we have:

$$\operatorname{rank}(A) = \dim(\operatorname{span}(a_1, a_2, \cdots, a_n)) = \dim(\operatorname{range}(A)).$$

And I would say the definition of **full rank** is still straightforward though. However, to understand it in a more direct (or say 'geometrical') way using the similar intuition from the proof of equality of ranks. So here are two figures to demonstrate of the idea:

Demonstration

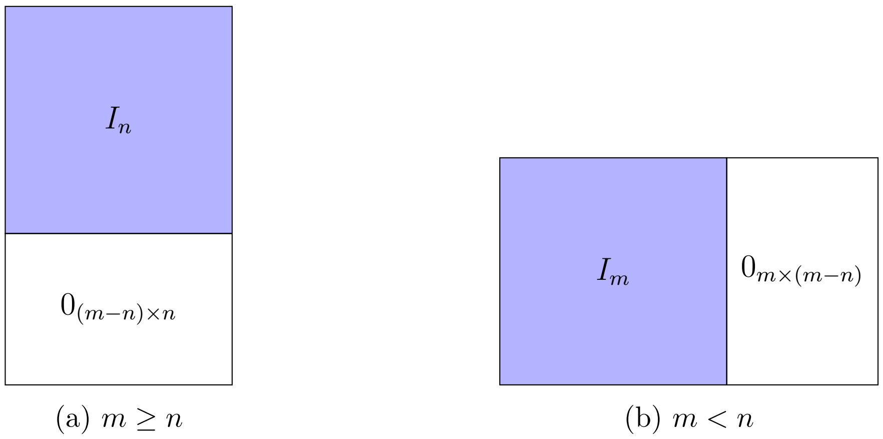


Figure 1.2: Full rank demonstration

- Firstly we could think the case $m \geq n$, here n is the 'lesser' in dimension $\mathbb{C}^{m\times n}$, and for the given matrix A, the maximum number of linearly independent vectors in column vectors the matrix should be n.
- And if we try to continuously do elementary row and column operations like we do in the previous proof, we would get the transformed matrix A'. If all column vectors are linearly independent, unlike any general matrix after transformation (see **Figure 1**), the size of identity matrix block in A' would just fit the 'width' of A'(A as well), as shown in **Figure 2(a)**.
- So the rank of matrix A, i.e. the size of identity matrix block in A', should be n. And the rank cannot be any bigger for all matrix $M \in \mathbb{C}^{m\times n}$. Therefore, we call matrix A has **full** rank in this case.
- In the other case where $m < n$, we could use same intuition to demonstrate the idea of **full rank** where the 'lesser' here is the 'height' of A, i.e. m in this case. You could see it from **Figure 2(b)**.

1.6 Full Rank and Matrix Properties

In this part, we are going to talk about the proof of **Theorem 1.9** in the master notes:

$$\forall A \in \mathbb{C}^{m\times n}, A \text{ full rank} \iff \neg(\exists x, y \in \mathbb{C}^n, x \neq y \implies Ax = Ay)$$

$$\iff (\forall x, y \in \mathbb{C}^n, x \neq y \implies Ax \neq Ay) \iff A \text{ is an injection}$$
$$\iff \forall x \in \mathbb{C}^n Ax \text{ is unique in } \mathbb{C}^m.$$

Proof

- Without loss of generality, we consider the case that $m \geq n$.
- **To prove the forward proposition**, we could use the column-space formulation in multiplication Ax and have:

$$Ax = \sum_{i=1}^{n} x_i a_i$$

  with a_i represents the ith column of matrix A.
- Since A is full rank, so all column vectors are linearly independent. Therefore, Ax gives out a unique linear combination in $\mathbb{C}^m$ formed by columns of A. i.e. no two distinct vectors would be mapped to same vector, so the forward direction has been proved.
- **To prove the backward proposition**, we use the Rank-Nullity theorem, for any given linear map $T : V \to W$, where V, W are vector spaces and V is finite dimensional:

$$\text{rank}(T) + \text{nullity}(T) = \dim(V).$$

- Since $A : \mathbb{C}^n \to \mathbb{C}^m$, we have $\dim(A) = n$.
- Considering $\text{nullity}(A) = \dim(\text{null}(A))$, since for $\text{null}(A)$:

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

- Since A cannot be a zero matrix, so the only vector could satisfy $Ax = 0_m$ should be $x = 0_n$. i.e.

$$\text{null}(A) = \{0_n\} \text{ and } \text{nullity}(A) = \dim(\{0_n\}) = 0.$$

- Therefore, according to Rank-Nullity theorem, we have

$$\text{rank}(A) = n - 0 = n = \min(m, n).$$

- So we could prove A is full rank, and we have done proofs of both forward and backward propositions.

- We could do the same thing for the case $m < n$, and just as the master notes saying, the theorem is a consequence of the column space interpretation of matrix-vector multiplication.

And also it is clear to see the relations between the full rank and invertibility of an matrix, nothing added for explaining **Theorem 1.10**.

Exercise 1.11: Construct a Rank-2 Matrix

Problem Description

Given that we have $u_1, u_2 \in \mathbb{C}^m$ and $v_1, v_2 \in \mathbb{C}^n$, we want to construct a rank-2 matrix $A = u_1 v_1^* + u_2 v_2^*$. However, from the given code and the specification, we need to somehow return matrix A as a product of two other matrices B and C.

What to do

We need to consider the following two questions:

- How to write A in terms of a product of matrices?
- Why A is rank-2 under this construction?

And here we consider the A as a summation of two matrices first:

$$A = u_1 v_1^* + u_2 v_2^*$$

where

$$u_1 v_1^* = \begin{pmatrix} v_{11}^* u_1 & v_{12}^* u_1 & \cdots & v_{1n}^* u_1 \end{pmatrix}$$

and

$$u_2 v_2^* = \begin{pmatrix} v_{21}^* u_2 & v_{22}^* u_2 & \cdots & v_{2n}^* u_2 \end{pmatrix}.$$

Therefore, A could be written in the sum:

$$A = u_1 v_1^* + u_2 v_2^* = \begin{pmatrix} v_{11}^* u_1 + v_{21}^* u_2 & v_{12}^* u_1 + v_{22}^* u_2 & \cdots & v_{1n}^* u_1 + v_{2n}^* u_2 \end{pmatrix}$$

with

$$a_i = v_{1i}^* u_1 + v_{2i}^* u_2 = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \begin{pmatrix} v_{1i}^* \\ v_{2i}^* \end{pmatrix}.$$

To generalize this, we could now write A in terms of a product of two matrices:

$$A = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \begin{pmatrix} v_1^* \\ v_2^* \end{pmatrix}.$$

So our first job has been done, now here is the implementation:

Code Implementation

```python
def rank2(u1, u2, v1, v2):
    """
    Return the rank2 matrix A = u1*v1^* + u2*v2^*.
    ...
    """
    # Construct the matrix from column vectors u1 and u2.
    # Using transpose is because matrix is constructed from
   rows by default.
    B = np.array([u1, u2]).T
    # Construct the conjugate matrix formed by v1 and v2.
    C = np.conj(np.array([v1, v2]))

    A = B.dot(C)

    return A
```

Now let us talk about the rank of A, why it is necessarily 2? We recall the result of A we have found with column-space formulation:

$$A = u_1v_1^* + u_2v_2^* = \begin{pmatrix} v_{11}^*u_1 + v_{21}^*u_2 & v_{12}^*u_1 + v_{22}^*u_2 & \cdots & v_{1n}^*u_1 + v_{2n}^*u_2 \end{pmatrix}.$$

We could see that, every column of A is a linear combination of u_1 and u_2, assume that they are linearly independent. Therefore, when we are talking about the range of A, it should be a vector space spanned by u_1 and u_2 and the dimension of the space should be 2. **i.e. the rank is 2 obviously.**

1.7 Invertibility and inverses

The idea and properties related to invertibility and inverses should be quite familiar, I would not talk about this here. What I want to focus is the proof exercise mentioned in the master notes, and the exercise in this part.

Problem Description

The exercise is asked us to prove that, there exists a unique **left** inverse Z of a given square matrix $A \in \mathbb{C}^{n\times n}$ such that $ZA = I_n$. And here is the proof:

Proof

- Suppose there exists another left inverse Z' of A, such that $Z'A = I_n$ but $Z' \neq Z$.
- Consider the (i, j) entry of the product $Z'A$, if $i \neq j$:

  $$(Z'A)_{ij} = \sum_{k=1}^{n} z'_{ik} a_{kj} = 0$$

  and similarly

  $$(ZA)_{ij} = \sum_{k=1}^{n} z_{ik} a_{kj} = 0.$$

- Therefore, we subtract one equation from another, we would have:

  $$(ZA)_{ij} - (Z'A)_{ij} = \sum_{k=1}^{b} (z_{ik} - z'_{ik}) a_{kj} = 0 - 0 = 0.$$

- The sum equals zero indicates that, when $i \neq j$, for all k, the non-trivial case would give us:

  $$z_{ik} - z'_{ik} = 0 \implies z_{ik} = z'_{ik}.$$

- Similarly, when $i = j$, we would know that $(Z'A)_{ij} = (ZA)_{ij} = 1$. So after subtraction like we did above, we would get:

  $$(ZA)_{ij} - (Z'A)_{ij} = \sum_{k=1}^{b} (z_{ik} - z'_{ik}) a_{kj} = 1 - 1 = 0.$$

  and again

  $$z_{ik} - z'_{ik} = 0 \implies z_{ik} = z'_{ik}.$$

- Therefore, we have seen $z_{ik} = z'_{ik}$ in each scenario, therefore, our suggested new left inverse Z' is just original Z itself, which gives a contradiction.
- So we have now proved the left inverse is unique.
- We could use the same strategy to prove the uniqueness of right inverse Z such that $AZ = I_n$.
- Therefore, we could say the inverse Z, of a square matrix A, such that $ZA = AZ = I_n$, is unique if the matrix itself is invertible.

Exercise 1.13: Find the inverse of a given matrix

Problem Description

Given that a square matrix $A = I_m + uv^* \in \mathbb{C}^{m\times m}$, where $u, v \in \mathbb{C}^m$. Suppose we could find the inverse A' written in $A' = I_m + \alpha uv^*$, determine the value of α and implement the function `rank1pert_inv()` to return the inverse matrix A'. So we try to determine the value of α first:

Analysis

- Given that $AA^{-1} = I_m$:

$$AA^{-1} = (I_m + uv^*)(I_m + \alpha uv^*) = I_m + uv^* + \alpha uv^* + \alpha uv^* uv^* = I_m.$$

- From the numerical analysis course we know the inner product, v^*u is a scalar, then we could have:

$$uv^* + \alpha uv^* + \alpha(v^*u)uv^* = 0_{m\times m} \implies (1 + \alpha + \alpha v^*u)uv^* = 0_{m\times m}.$$

- Therefore, we have:

$$\alpha(1 + v^*u) = -1 \implies \alpha = -\frac{1}{1 + v^*u}$$

  and

$$A^{-1} = I_m - \frac{1}{1 + v^*u}uv^*$$

  whenever A is invertible.

So the code implementation shoule be very simple.

Code Implementation

```python
def rank1pert_inv(u, v):
    """
    Return the inverse of the matrix A = I + uv^*, where ...
    """
    # A^-1 = I - (uv^*)/(1 + v^*u)
    return np.eye(len(u)) - (np.outer(u, np.conj(v)) /
                             (1 + np.inner(np.conj(v), u)))
```

Efficiency Measuring

To measure the time taken for computing inverse of this implementation, and compare it with the built-in numpy implementation (just like what we did for timing the matrix-vector multiplication), I add the following code:

```python
import numpy.random as random
u0 = random.randn(400)
v0 = random.randn(400)


def timeable_basic_matinv():
    a_inv = rank1pert_inv(u0, v0)  # noqa


def timeable_numpy_matinv():
    a_inv = np.linalg.inv(np.eye(len(u0)) + np.outer(u0, np.
   conj(v0)))  # noqa


def time_matinvs():
    print("Timing for basic_matinv")
    print(timeit.Timer(timeable_basic_matinv).timeit(number
   =1))
    print("Timing for numpy matinv")
    print(timeit.Timer(timeable_numpy_matinv).timeit(number
   =1))
```

and the output shows that:

```
>>> import cla_utils
>>> cla_utils.time_matinvs()
Timing for basic_matinv
0.0013024520000044504
Timing for numpy matinv
0.005589449000012792
```

We could see that the numpy implementation is slower. The reason of this output I suppose is that, the complexity of the numpy implementation, to be more specific, the number of operations in the numpy implementation is more than that in our implementation above. Therefore, it takes longer time to compute the inverse.

1.8 Adjoints and Hermitian Matrices

We firstly skip the definition of adjoints and Hermitian matrices since they are straightforward enough. Then we see the proof of **theorem 1.15**, which states:

$$(AB)^* = B^*A^*, A \in \mathbb{C}^{m\times n}, B \in \mathbb{C}^{n\times l}.$$

And the proof should be simple.

Proof

- Considering the (i, j) entry of $(AB)^*$ and we have:

$$(AB)^*_{ij} = \overline{(AB)_{ji}} = \overline{\sum_{k=1}^{n} a_{jk}b_{ki}} = \sum_{k=1}^{n} \overline{a_{jk}b_{ki}} = \sum_{k=1}^{n} \overline{a_{jk}}\,\overline{b_{ki}}$$

$$= \sum_{k=1}^{n} (a^*)_{kj}(b^*)_{ik} = \sum_{k=1}^{n} (b^*)_{ik}(a^*)_{kj} = (B^*A^*)_{ij}.$$

- Therefore, we finally have $(AB)^* = B^*A^*$.

And then we could look at the following exercise implementing the required function `ABiC()`.

Exercise 1.16: Hermitian Matrices and Multiplication

This exercises have several mini-questions for us to solve. The first part is mainly about proof with hermitian matrices, and the function implementation part is about the application. Let's see the proof exercise first.

Task 1

Given that $A = B + iC \in \mathbb{C}^{m\times m}$ where B, C are both real matrices with compatible dimensions, and A is hermitian, show that:

$$B = B^T \text{ and } C = -C^T.$$

Proof

- Given that A is hermitian then the property tells us:
$$A = A^* \text{ with } a_{ij} = (a^*)_{ij} = \overline{a_{ji}}.$$
- Considering the (i, j) and (j, i) entries of A, we have:
$$a_{ij} = b_{ij} + ic_{ij} \text{ and } a_{ji} = b_{ji} + ic_{ji}.$$
- Therefore, since A is hermitian, we have:
$$a_{ij} = \overline{a_{ji}} \implies b_{ij} + ic_{ij} = \overline{b_{ji} + ic_{ji}} = b_{ji} - ic_{ji}.$$
- Clearly
$$b_{ij} = b_{ji} \text{ and } c_{ij} = -c_{ji}.$$
- Therefore,
$$B = B^T \text{ and } C = -C^T.$$

Task 2

We required to compute the real and imaginary part of the matrix multiplication product $z = Ax$, where A is defined above. To save memory the function would accept the contents of A as an argument, but a modified matrix $\hat{A}$, where
$$\hat{A}_{ij} = \begin{cases} B_{ij} \text{ where } i \geq j \\ C_{ij} \text{ where } i < j. \end{cases}.$$
Therefore, given a matrix $\hat{A}$ and x_r, x_i where $x = x_r + ix_i$, we need to implement our function `ABiC()`.

What to do

1. Since we don't have contents of $A = B + iC$ as required, we could firstly try to get contents in B and C from $\hat{A}$.
2. The contents in $\hat{A}$ should look like this:
$$\hat{A} = \begin{pmatrix} b_{11} & c_{12} & c_{13} & \dots & c_{1m} \\ b_{21} & b_{22} & c_{23} & \dots & c_{2m} \\ b_{31} & b_{32} & b_{33} & \dots & c_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \dots & b_{mm} \end{pmatrix}.$$

So what we could do is to extract the lower-triangular part of B and upper-triangular part of C from $\hat{A}$, by using `np.triu()` and `np.tril()`, and then get the full form of B and C by transposition.

3. After having B and C, we could simply calculate the value of $z = Ax$:

$$z = Ax = (B + iC)(x_r + ix_i) = (Bx_r - Cx_i) + i(Bx_i + Cx_r)$$

where z_r and z_i could be easily seen.

Code Implementation

Now the implementation should be clear, additional details are stated in comments.

```python
def ABiC(Ahat, xr, xi):
    """Return the real and imaginary parts of z = A*x,
    where A = B + iC.
    """
    # Find B, C.
    # Get the part of B, C in Ahat without leading diagonal.
    B, C = np.tril(Ahat, -1), np.triu(Ahat, 1)

    B = B + B.T + np.diag(np.diag(Ahat))
    C = C - C.T

    # Find zr, zi using column space formulation.
    m, _ = Ahat.shape
    zr = np.zeros(m)
    zi = np.zeros(m)

    for j, (er, ei) in enumerate(zip(xr, xi)):
        zr += B[:, j] * er - C[:, j] * ei
        zi += B[:, j] * ei + C[:, j] * er

    return zr, zi
```

1.9 Inner Products and Orthogonality

The idea of these two concepts are very simple but they are extremely important in computational linear algebra. We would leave the proof of showing inner map is bilinear as an exercise here.

Proof

- Considering $f(x,y) = x^*y$, we firstly show the inner map is linear respect to x by deriving $f(\alpha x + x', y)$:

$$f(\alpha x + x', y) = (\alpha x + x')^* y = \sum_{i=1}^{n} \overline{(\alpha x + x')_i} y_i$$

$$= \sum_{i=1}^{n} (\alpha \overline{x}_i + \overline{x'}_i) y_i = \alpha \sum_{i=1}^{n} \overline{x}_i y_i + \sum_{i=1}^{n} \overline{x'_i} y_i$$
$$= \alpha x^* y + x'^* y = f(\alpha x, y) + f(x', y).$$

- Clearly the inner map is linear with respect to x, and similarly the part about y could be shown in a similar way.

- Therefore we could see the inner map is bilinear.

1.10 Orthogonal Components of a Vector

Given that we have a orthonormal set of vectors $S = \{q_1, q_2, \ldots, q_n\}$ where $q_i \in \mathbb{C}^m$, and for any vector $v \in \mathbb{C}^m$, we have:

$$v = r + \sum_{i=1}^{n} (q_i q_i^*) v.$$

More specifically, it also shows that:

- Given that we have an orthonormal set(basis) S, we could always split this vector to a linear combination of vectors in the orthonormal set, and a secret vector r.
- This secret vector r orthogonal to any element in S.
- So if we add r to S and gives out a new set S', the newly formed set is still necessarily an orthonormal basis.
- Can we use this idea in finding an orthonormal basis from any basis?

We would discuss this later in next chapter for QR decomposition.

Exercise 1.20: Orthonormal Decomposition of Vectors

Problem Description

Given a vector v and an orthonormal set Q, we were asked to write v as a sum of linear combination of elements in Q **and** the residual vector r.

The function `orthog_cpts()` would do this work. It returns the coefficients u in the linear combination, and the residual vector r. Our work is to implement this function.

What to do

1. Recall that
$$v = r + \sum_{i=1}^{n} (q_i^* v) q_i.$$
We know that the coefficients u_i comes from the inner product between q_i and v where $u_i = q_i^* v$.

2. And what we need to do is just keep substracting compoents of q_i from v (should be $u_i q_i$), until all orthogonal components are removed from v, then we would get our r.

Code Implementation

```python
def orthog_cpts(v, Q):
    """
    Given a vector v and an orthonormal set of vectors
    q_1,...q_n, compute v = r + u_1q_1 +  ... + u_nq_n
    for scalar coefficients u_1, u_2, ..., u_n and
    residual vector r.
    """
    r, u = v.copy(), np.array([])

    for i in range(len(Q[0])):
        qi = Q[:, i]
        # Find the scale factor of q_i in v,
        # and remove the orthogonal component.
        ui = np.conj(qi).dot(v)
        r -= ui * qi
        u = np.append(u, [ui])

    return r, u
```

The code should be self-intuitive enough for you to understand.

1.11 Unitary Matrices

The definition, theorem and proof stated in the master notes in this section is clear enough, and I would like to emphasize one useful trick dealing with unitary matrices:

$$Q \text{ is unitary } \iff Q^* = Q^{-1} \iff I = Q^*Q.$$

and $Q^* = Q^{-1}$ could be interpreted as a change of orthogonal basis.

Exercise 1.23: Solve the System $Qx = b$

Problem Description

Given a square **unitary** matrix Q and a vector b, find the vector x satisfies $Qx = b$ without using inverses of Q in the implementation of `solveQ()`.

What to do

1. Since we are not allowed to use inverses, and Q is unitary, as we stated above, we could replace Q^{-1} by Q^* then $x = Q^{-1}b = Q^*b$.

2. Then Q^* could be easily represented as the combination of transposing the conjugate of matrix Q via `numpy` operations.

   (Hint: try to consider `numpy.conj()` and `numpy.ndarray.T`).

3. Then we could simply get $x = Q^*b$ and implement the function.

Code Implementation

```
def solveQ(Q, b):
    """
    Given a unitary mxm matrix Q and a vector b,
    solve Qx=b for x.
    """
    return np.conj(Q).T.dot(b)
```

Efficiency Measuring

To test the performance of function efficiency for random matrices with different sizes (compared with the general purpose `numpy.linalg.solve()`), we have the following code:

```
import timeit
from numpy import random, linalg
def time_solveQ():
    """
    Get some timings for solveQ.
    """
    for size in [100, 200, 400]:
        Q = random.randn(size, size)
        b = random.randn(size)
        print("--- Input matrix size n = {} ---".format(size)
   )
        print("Timing for solveQ")
        print(timeit.Timer(lambda: solveQ(Q, b)).timeit(
   number=1))
        print("Timing for general purpose solve")
        print(timeit.Timer(lambda: linalg.solve(Q, b)).timeit
   (number=1))
        print("--- End for testing matrix with n = {} ---".
   format(size))
```

Here I pass an `lambda` to the `timeit.Timer` constructor rather than an predefined helper (timeable) function for code simplicity, and I can also pass variables to either `solveQ()` and `numpy.linalg.solve()` as I want.

Since we use adjoint matrices rather than inverses, which reduces a certain amount of operations in finding inverses, so I would expect `solveQ()` would take less time than general purpose `solve()`. And the python console gives me the following result:

```
>>> import cla_utils
>>> cla_utils.time_solveQ()
--- Input matrix size n = 100 ---
Timing for solveQ
0.00011220100000031152
Timing for general purpose solve
0.0003564039999996993
--- End for testing matrix with n = 100 ---
--- Input matrix size n = 200 ---
Timing for solveQ
0.0002462439999950697
Timing for general purpose solve
0.0006074579999992835
--- End for testing matrix with n = 200 ---
--- Input matrix size n = 400 ---
Timing for solveQ
0.0007872969999986879
Timing for general purpose solve
0.002527909000001216
--- End for testing matrix with n = 400 ---
```

We could easily see that our implementation is more efficient than the general purpose `solve()`, given that the matrix is unitary.

Note: The explanation of vector norms in **master notes 1.8** is clear enough so I would not cover this in my commentary.

1.12 Projectors and Projections

The concept of projector is self-intuitive, but the relations between projector P and its complementary projector are still worth mentioning.

Let's consider a projector P first:

- Suppose $\exists x : Px = v$, i.e. $v \in \text{range}(P)$, we could see that:

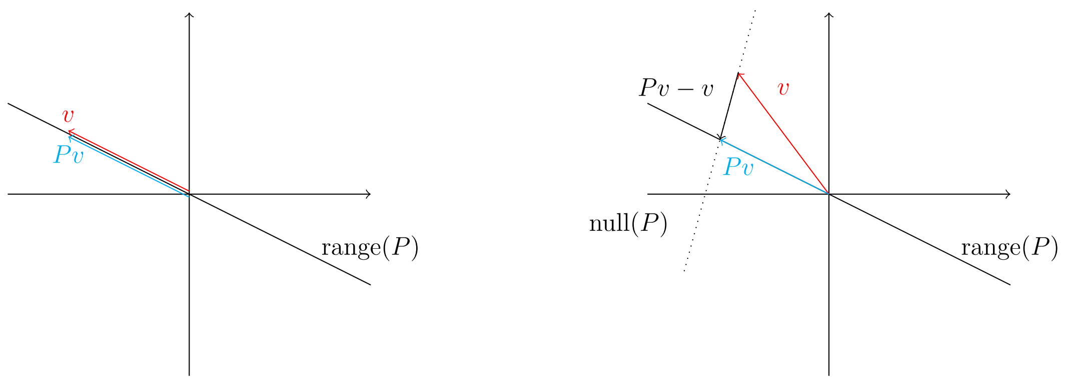
  $$Pv = P(Px) = P^2x = Px = v.$$

  we could see projector P doesn't change vectors in its range.(see **Figure 1.3(a)**).

- Suppose $v \notin \text{range}(P)$, considering the vector $Pv - v$:

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

Here we could say $Pv - v$ is the nullspace of P, shown in **Figure 1.3(b)**.



(a) $v \in \text{range}(P)$ (b) $v \notin \text{range}(P)$

Figure 1.3: Projecting v with projector P

Then consider the **complementary projector** $I - P$:

- Obviously it's a projector as well. i.e. $(I-P)^2 = I - P$, and simply
$$Pu = 0 \implies (I-P)u = u - Pu = u.$$
- The proposition above shows that, if a vector $u \in \text{null}(P)$, it is in range$(I-P)$. To generalize this, we have:
$$[\forall u \in \text{null}(P) \implies \text{range}(I-P)] \implies$$
$$\text{null}(P) \subseteq \text{range}(I-P).$$
In other words, the nullspace of P is contained in the range of $I - P$.
- Suppose there exists an vector $v \in \text{range}(I-P)$, in other words
$$\exists w \in \mathbb{C}^m : v = (I-P)w = w - Pw.$$
- Then it shows that
$$Pv = P(w - Pw) = Pw - P^2w = Pw - Pw = 0.$$
- So indeed $v \in \text{null}(P)$, in this direction we could have:
$$[v \in \text{range}(I-P) \implies v \in \text{null}(P)] \implies$$
$$\text{range}(I-P) \subseteq \text{null}(P).$$

- And this leads to the result

  $$\text{null}(P) = \text{range}(I - P).$$

  similarly

  $$\text{null}(I - P) = \text{range}(P).$$

Important things for projector P:

- It separates $\mathbb{C}^m$ into two spaces, range(P) and null(P), where

  $$\text{range}(P) \cap \text{null}(P) = \{\mathbf{0}\}.$$

  The reason of why the intersection contains $\mathbf{0}$ only is that, considering the two sets null(P) and null($I - P$), the only common element is $\mathbf{0}$. That is because any vector v in both sets satisfies $v = v - Pv = (I - P)v = 0$. And since we know null$(I - P) =$ range(P), the results could be shown as stated above.

- Conversely, if there exists two subspaces S_1 and S_2 of $\mathbb{C}^m$, where $S_1 \cap S_2 = \{\mathbf{0}\}$, there exists a projector P whose range is S_1 and nullspace is S_2.

Proof

- The proposition above states that, any vector in $\mathbb{C}^m$ could be split into two components, in subspaces S_1 and S_2 respectively.

- Therefore, to prove this proposition, we could simplify this problem to the following proposition:

  $$\text{Given that } S_1 \cap S_2 = \{0\}, \forall v \in \mathbb{C}^m, \text{ find } v_1 \in S_1, v_2 \in S_2 : v_1 + v_2 = v.$$

  where the projection Pv gives v_1 and the complementary projection $(I - P)v$ would gives v_2.

- Obviously we could find the vector pair, since these two subspaces are distinct. But is the vector pair (v_1, v_2) unique?

- Suppose there exists another vector pair $(v_1', v_2') \in S_1 \times S_2$ which satisfies the conditions above, where clearly $v_i \neq v_i'$ and:

  $$\begin{aligned} v_1 + v_2 &= v \\ v_1' + v_2' &= v \end{aligned} .$$

- Subtract the first equation by the second, and we get:

$$(v_1 - v_1') + (v_2 - v_2') = \mathbf{0}.$$

- Say $v_1'' = v_1 - v_1'$ and $v_2'' = v_2 - v_2'$, and we could clearly see $v_i'' \in S_i$ due to the properties of subspaces. And we could get:

$$v_1'' + v_2'' = \mathbf{0} \implies v_1'' = -v_2''.$$

- We know that the subspace preserves scalar multiplication, therefore, we could see v_1'' is in S_2 and v_2'' is in S_1 as well.

- In this case, we know that v_1'', v_2'' are both in S_1, S_2. So we could interpret $v_1'', v_2'' \in S_1 \cap S_2 = \{\mathbf{0}\}$.

- Then obviously $v_1'' = v_2'' = \mathbf{0}$, and

$$\begin{matrix} v_1'' = v_1 - v_1' = \mathbf{0} \\ v_2'' = v_2 - v_2' = \mathbf{0} \end{matrix} \implies \begin{matrix} v_1 = v_1' \\ v_2 = v_2' \end{matrix}.$$

  which contradicts our assumption $v_i \neq v_i'$, so the vector pair (v_1, v_2) should be unique.

- Therefore, we could find a unique decomposition of v into two components v_1, v_2 in S_1 and S_2.

- Since we take v arbitrarily, and the components v_1, v_2 in corresponding subspaces S_1, S_2 could be written as Pv and $(I - P)v$ for all chosen v, (which are both **bijections**). So we could say there exists a projector P such that range$(P) = S_1$ and null$(P) = S_2$.

1.13 Orthogonal Projectors and Construction

We have known the definition of projectors, but how about the orthogonal projectors? The definition states that a projector P is orthogonal if:

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

This definition might still be too abstract, so we put the definition into the following **Figure 1.4** :

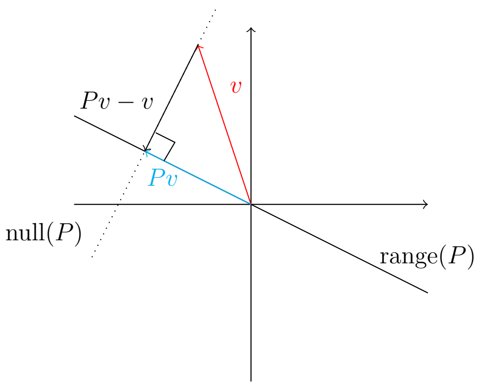


Figure 1.4: Projecting v with Orthogonal Projector P

We could see the projector P splits v into two components, Pv and $Pv - v$. However, in this case these two components are orthogonal to each other, and computationally, their inner product should be 0. That's where the definition of **orthogonal** projector comes from.
Note: More specifically, an orthogonal projector splits $\mathbb{C}^m$ into two orthogonal subspaces, range(P) and null(P).

Now we know the definition of an orthogonal projector, but how do we construct an orthogonal projector? **In this course, we are introduced to construct from a set of orthonormal vectors.**

- From the knowledge we get in **section 1.10, commentary notes**, we know that, given we have a set of orthonormal vectors $\{q_1, q_2, \ldots, q_n\}$, for all vector $v \in \mathbb{C}^m$, we could write v in terms of:
$$v = r + \sum_{i=1}^{n} (q_i q_i^*) v.$$
- And if we write matrix $\hat{Q}$ as:
$$\hat{Q} = \begin{pmatrix} q_1 & q_2 & \ldots & q_n \end{pmatrix}.$$
we could see $\sum_{i=1}^{n} (q_i q_i^*) v$ is in the column space of $\hat{Q}$, and the component r in v should be orthogonal to any vectors in the orthonormal set $\{q_i\}$, i.e. r should be orthogonal to the column space of $\hat{Q}$.
- So if we consider $P = \sum_{i=1}^{n} (q_i q_i^*)$, then we have:
$$v = r + Pv.$$
and r and Pv should be orthogonal to each other, as we stated above. So in this case, we have P as our orthogonal projector.

So now we know how to construct an orthogonal projector, but what the form of an orthogonal projector usually looks like?

$$P \text{ orthogonal } \implies P = \hat{Q}\hat{Q}^*.$$

Proof

- First of all, computationally if we expand the multiplication $\hat{Q}\hat{Q}^*$:

$$\hat{Q}\hat{Q}^* = \begin{pmatrix} q_1 & q_2 & \dots & q_n \end{pmatrix} \begin{pmatrix} q_1^* \\ q_2^* \\ \vdots \\ q_n^* \end{pmatrix} = q_1q_1^* + q_2q_2^* + \dots + q_nq_n^* = \sum_{i=1}^{n}(q_iq_i^*).$$

  which is exactly what we see in the construction of the orthogonal projector P. **Personally I think all of us should understand the proof at this level.**

- Also, in the light of change of basis, the product $\hat{Q}^*v$ gives the coefficients of v after projecting along the columns of $\hat{Q}$, and the multiplication of $\hat{Q}$ on $\hat{Q}^*v$ gives the projections of v expanded in the canonical basis.

- Therefore, the multiplication by $\hat{Q}\hat{Q}^*$ does the same job as the orthogonal projector P does. We could finally confirm that the orthogonal projectors has a simple form $P = \hat{Q}\hat{Q}^*$.

- **Note:** the last two parts of the proof are mentioned under **Theorem 1.28 master notes**, and I think it is not necessary to memory by heart but would give you a deeper understanding if digest them.

Since any orthogonal projector has the form $P = \hat{Q}\hat{Q}^*$, we have the following properties of orthogonal projectors to discuss:

Property 1

- We have known that the orthogonal projector P gives an orthogonal projection to range of P.

- However, from our interpretation in the previous proof, the orthogonal projector $\hat{Q}\hat{Q}^*$ would eventually expands any v by columns of $\hat{Q}$.

- So we could conclude that, $P = \hat{Q}\hat{Q}^*$ is an orthogonal projection to range of $\hat{Q}$.

- Moreover we could say range(P) = range$(\hat{Q})$, since the orthogonal projection on these two planes are equivalent.

Property 2

- Similarly, the complementary projector $P_\perp = I - \hat{Q}\hat{Q}^*$ would give a orthogonal projection to nullspace of $\hat{Q}$.
- And clearly null(P) = null$(\hat{Q})$.

Property 3

- Given that any orthogonal projector has the form $P = \hat{Q}\hat{Q}^*$, we could see its adjoint $P^* = (\hat{Q}\hat{Q}^*)^* = (\hat{Q}^*)^*(\hat{Q})^* = \hat{Q}\hat{Q}^* = P$.
- And we have the following theorem:

$$P^* = P \iff P \text{ orthogonal.}$$

Proof

- The proof of the backward proposition has been shown above, and we need to prove the forward one: $P^* = P \implies P$ orthogonal.
- We choose a vector $v \in \mathbb{C}^m$ arbitrarily, and we have:

$$(Pv)^*(Pv - v) = v^*P^*(P - I)v = v^*P(P - I)v$$

$$= v^*(P^2 - P)v = v^*(P - P)v = 0.$$

  Therefore P is orthogonal.

Exercise 1.30 Constructing Orthogonal Projectors

Problem Description

This exercise is a relatively easy one - given an orthonormal set of vectors Q, we need to compute an orthogonal projector P which projects vectors to the subspace spanned by elements of Q.

What to do

We just need to use the construction of orthogonal projector $P = \hat{Q}\hat{Q}^*$, and in this exercise we can easily use our provided Q as the orthonormal matrix. The code implementation goes as follows (should be straightforward enough):

Code Implementation

```python
def orthog_proj(Q):
    """
    Given a vector v and an orthonormal set of vectors
    q_1,...q_n, compute the orthogonal projector P
    that projects vectors onto the subspace
    spanned by those vectors.
    """
    return Q.dot(np.conj(Q).T)
```

After learning all the basic things in this chapter, the exited things finally would come in the following chapters ...! Next chapter would be based on the knowledge of orthogonal projectors and we would talk about **QR factorisation**.

Chapter 2

QR Factorisation

In computational linear algebra, we would spend a lot of time in matrix equations or expressions like $Ax = b$. However, for large matrices A, it is always difficult and inefficient to find the value of x, if we try finding the inverse of A directly. So what we want to do is to "transform" our A into smaller building blocks with specific properties, and use the properties of them to make the equation easier to solve.

The transformations, should preserve the correctness in matrix calculations (be sufficiently free from truncation errors, as said in the **Introduction of Chapter 2, master notes**), and be efficient enough to perform. Here, let us talk about the very first transformation, the **QR Factorisation**.

2.1 QR Factorisation Concept

What does **QR Factorisation** do is to simply decompose an arbitrary matrix A, to a unitary matrix Q and an upper triangular matrix R. i.e. to write

$A = QR$ where $A \in \mathbb{C}^{m\times n}, Q$ unitary $\in \mathbb{C}^{m\times m}, R$ upper triangular $\in \mathbb{C}^{m\times n}$.

And without loss of generality we consider $m > n$, and have:

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} q_{11} & q_{12} & \dots & q_{1n} & \dots & q_{1m} \\ q_{21} & q_{22} & \dots & q_{2n} & \dots & q_{2m} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ q_{m1} & q_{m2} & \dots & q_{mn} & \dots & q_{mm} \end{pmatrix}}_{Q} \underbrace{\begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ 0 & r_{22} & \dots & r_{2n} \\ 0 & 0 & \dots & r_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_{nn} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}}_{R}$$

And we called this as the **complete QR factorisation**.

Note that, the term $q_{ik}r_{kj}$ would always give out 0 as result when $k > n$ and make no contribution to the value of a_{ij}. Therefore, we could only consider the first n columns of Q, $\hat{Q}$ and the first n rows of R, $\hat{R}$ as result of factorisation instead (i.e. Determine $A = \hat{Q}\hat{R}$ as a reduced alternative).

Exercise 2.3: Find an orthonormal basis of the orthogonal complement of a subspace

Problem Description

Given a set of vectors $\{v_1, v_2, \ldots, v_n\}$ spans subspace $U \subset \mathbb{C}^m$, we need to find an orthonormal basis of its orthogonal complement $U^{\perp}$, defined as:

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

(hint: You should make use of the built in QR factorisation routine `numpy.linalg.qr()`.)

What to do

1. Think about what does QR factorisation actually do: what are the properties of Q and R (and similarly $\hat{Q}$ and $\hat{R}$ in the reduced version)?
2. You might notice Q (and $\hat{Q}$) is unitary, so try to think about the relationship between column vectors in matrix by expanding $Q^*Q = I$.
3. And the basis of U could be easily seen by the columns of $\hat{Q}$. So if we find the vectors orthogonal to all column vectors in $\hat{Q}$, we should find the basis of the orthogonal complement of U.
4. Think that how we could find the required orthogonal vectors in Q. Your task is now to use the relationship you found in (2) to find these vectors, and code with Python.

Code Implementation

```
def orthog_space(V):
    """
    Given set of vectors u_1,u_2,..., u_n, compute the
    orthogonal complement to the subspace U spanned by the
   vectors.
    """
    _, n = V.shape
    # Get Q unitary from comoplete qr factorisation.
    Q, _ = np.linalg.qr(V, 'complete')

    # Then the subspace orthogonal to U should be spanned
    # by the remaining m - n col vectors by mutal
   orthogonality.
    return Q[:, n:]
```

Analysis

Here we have some explanation to this implementation:

- Given that we know Q is unitary, we would see all column vectors are orthogonal to each other by expanding

$$[Q^*Q]_{ij} = q_i^* q_j = I_{ij} = \begin{cases} 0 \text{ when } i \neq j \\ 1 \text{ when } i = j \end{cases}.$$

- And since we need to find the vectors that orthogonal to all column vectors in $\hat{Q}$, which is the first n columns of Q. We could actually see the remaining $m-n$ columns in Q are actually orthogonal to all column vectors in $\hat{Q}$, by the mutual orthogonality we observed in column space of Q.

- Therefore, what we need to do is just to find the value of Q via **complete** QR factorisation, via `np.linalg.qr(A, 'complete')`.

- And we need to get the **last** $m - n$ **columns** from Q, via `Q[:, n]`. It should be the basis of the orthogonal complementary $U^{\perp}$.

2.2 QR factorisation by classical Gram-Schmidt

We have already seen the concept of QR factorisation, and one of the application through the previous exercise. Then we are going to explore what exactly happens in this factorisation, via algorithms. The first implementation we would discuss would be the **classical Gram-Schmidt algorithm**.

But before we just show the explicit implementation, we could look backwards to see how vectors $\{a_i\}$ factctorised to orthonormal vectors $\{q_i\}$. Just expand $A = QR$ by arithmetic:

$$A = QR \implies \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} = \begin{pmatrix} q_1 & q_2 & \dots & q_n \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ 0 & r_{22} & \dots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_{nn} \end{pmatrix}.$$

So we could see the value of any column vector a_j via column-space interpretation and get:

$$a_j = \begin{pmatrix} q_1 & q_2 & \dots & q_j & \dots & q_n \end{pmatrix} \begin{pmatrix} r_{1j} \\ r_{2j} \\ \vdots \\ r_{jj} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$= r_{1j}q_1 + r_{2j}q_2 + r_{3j}q_3 + \dots + r_{jj}q_j + 0 \cdot q_{j+1} + \dots + 0 = \sum_{i=1}^{j} r_{ij}q_i.$$

Since we know $\{q_i\}$ is an orthonormal set of vectors, so what we exactly did in QR factorisation is to project a vector a_i orthogonally into every direction in $\{q_i\}$, with approprate scale factors of different base vectors. These scale factors r_{ij} would be stored in R. To find the scale factor of the projection in direction of q_i, we could easily get the value:

$$r_{ij} = q_i^* a_j.$$