**LINEAR INDEPENDENCE**

**REPORT**



**School or Department:** Wu Han University

**Grade and Specialty:** Excellent engineer 2016

**Name:** Guo Yang, Kong Chuishun, Tang Rui

**Advisor:** Fangling Pu

April 17 2018

【**Abstract**】In this lab, we pay attention to the Linear independence. It is a very important property of matrix and vector. To strengthen the understanding of that. We use a python code to decompose matrix by the Gram–Schmidt algorithm.

**【Key words】**Python; linear independence; Gram–Schmidt algorithm

**Table of Contents**

[Abstract 1](#_Toc324279571)

[Chapter 1 Introduction 1](#_Toc324279574)

[1.1 linear independence 1](#_Toc324279574)

[1.2 Orthonormal vectors 1](#_Toc324279574)

[1.3 Orthonormal matrix 1](#_Toc324279574)

1.4 [Gram–Schmidt algorithm 2](#_Toc324279574)

[Chapter 2 lab 3](#_Toc324279575)

[2.1 Gram–Schmidt process 3](#_Toc324279575)

[2.2 Numerical stability 4](#_Toc324279575)

[2.3 Determinant formula 5](#_Toc324279575)

[Chapter 3 Result 6](#_Toc324279576)

[Appendix 7](#_Toc324279591)

**Chapter 1 Introduction**

* 1. **Linear independence**

A collection or list of n-vectors *a1,...,ak* (with k ≥ 1) is called linearly dependent if *β1a1 +···+ βkak* = 0 holds for some *β1,...,βk* that are not all zero. In other words, we can form the zero vector as a linear combination of the vectors, with coeﬃcients that are not all zero. Linear dependence of a list of vectors does not depend on the ordering of the vectors in the list. When a collection of vectors is linearly dependent, at least one of the vectors can be expressed as a linear combination of the other vectors: If *βi* 6= 0 in the equation above (and by deﬁnition, this must be true for at least one i), we can move the term *βiai* to the other side of the equation and divide by βi to get ai = (−*β1*/*βi)a1* +···+ (−*βi*−1/*βi*)*ai*−1 + (−*βi*+1/*βi*)*ai*+1 +···+ (−*βk*/*βi*)*ak*. The converse is also true: If any vector in a collection of vectors is a linear combination of the other vectors, then the collection of vectors is linearly dependent.

* 1. **Orthonormal vectors**

A collection of vectors *a1,...,ak* is orthogonal or mutually orthogonal if *ai* ⊥ *aj* for any *i, j* with *i 6= j, i, j = 1,...,k*. A collection of vectors *a1,...,ak* is orthonormal if it is orthogonal and *kaik = 1* for *i = 1,...,k.* (A vector of norm one is called normalized; dividing a vector by its norm is called normalizing it.) Thus, each vector in an orthonormal collection of vectors is normalized, and two diﬀerent vectors from the collection are orthogonal. These two conditions can be combined into one statement about the inner products of pairs of vectors in the collection: *a1,...,ak* is orthonormal means that

Orthonormality, like linear dependence and independence, is an attribute of a collection of vectors, and not an attribute of vectors individually. By convention, though, we say “The vectors *a1,...,ak* are orthonormal” to mean “The collection of vectors *a1,...,ak* is orthonormal”.

* 1. **Orthonormal matrix**

For matrices with orthogonality over the [complex number](https://en.wikipedia.org/wiki/Complex_number) field, see [unitary matrix](https://en.wikipedia.org/wiki/Unitary_matrix).

In [linear algebra](https://en.wikipedia.org/wiki/Linear_algebra), an orthogonal matrix or real orthogonal matrix is a [square matrix](https://en.wikipedia.org/wiki/Square_matrix) with [real](https://en.wikipedia.org/wiki/Real_number) entries whose columns and rows are [orthogonal](https://en.wikipedia.org/wiki/Orthogonal) [unit vectors](https://en.wikipedia.org/wiki/Unit_vector) (i.e., [orthonormal](https://en.wikipedia.org/wiki/Orthonormality) vectors), i.e.

*QTQ = QQT = I*

where I is the [identity matrix](https://en.wikipedia.org/wiki/Identity_matrix).

This leads to the equivalent characterization: a matrix Q is orthogonal if its [transpose](https://en.wikipedia.org/wiki/Transpose) is equal to its [inverse](https://en.wikipedia.org/wiki/Inverse_matrix):

An orthogonal matrix *Q* is necessarily [invertible](https://en.wikipedia.org/wiki/Invertible_matrix) (with inverse *Q−1 = QT*), [unitary](https://en.wikipedia.org/wiki/Unitary_matrix) (*Q−1 = Q∗*) and therefore [normal](https://en.wikipedia.org/wiki/Normal_matrix) (*Q∗Q = QQ∗*) in the reals. The [determinant](https://en.wikipedia.org/wiki/Determinant) of any orthogonal matrix is either +1 or −1. As a [linear transformation](https://en.wikipedia.org/wiki/Linear_transformation), an orthogonal matrix preserves the [dot product](https://en.wikipedia.org/wiki/Dot_product) of vectors, and therefore acts as an [isometry](https://en.wikipedia.org/wiki/Isometry) of [Euclidean space](https://en.wikipedia.org/wiki/Euclidean_space), such as a [rotation](https://en.wikipedia.org/wiki/Rotation_(mathematics)) or [reflection](https://en.wikipedia.org/wiki/Reflection_(mathematics)). In other words, it is a [unitary transformation](https://en.wikipedia.org/wiki/Unitary_transformation).

The set of n × n orthogonal matrices forms a [group](https://en.wikipedia.org/wiki/Group_(mathematics)) O(n), known as the [orthogonal group](https://en.wikipedia.org/wiki/Orthogonal_group). The [subgroup](https://en.wikipedia.org/wiki/Subgroup) SO(n) consisting of orthogonal matrices with [determinant](https://en.wikipedia.org/wiki/Determinant) +1 is called the [special orthogonal group](https://en.wikipedia.org/wiki/Special_orthogonal_group), and each of its elements is a special orthogonal matrix. As a linear transformation, every special orthogonal matrix acts as a rotation.

* 1. **Gram–Schmidt algorithm**

In this section we describe an algorithm that can be used to determine if a list of n-vectors *a1,...,ak* is linearly independent. In later chapters we will see that it has many other uses as well. The algorithm is named after the mathematicians Jorgen Pedersen Gram and Erhard Schmidt, although it was already known before their work. If the vectors are linearly independent, the Gram–Schmidt algorithm produces an orthonormal collection of vectors *q1,...,qk* with the following properties: For each *i = 1,...,k, ai* is a linear combination of q1,...,qi, and qi is a linear combination of a1,...,ai. If the vectors a1,...,aj−1 are linearly independent, but a1,...,aj are linearly dependent, the algorithm detects this and terminates. In other words, the Gram–Schmidt algorithm ﬁnds the ﬁrst vector aj that is a linear combination of previous vectors *a1,...,aj.*

The orthogonalization step, with *i* = 1, reduces to *˜ q1 = a1*. If the algorithm does not quit (in step 2), i.e., ˜ *q1*,..., ˜ qk are all nonzero, we can conclude that the original collection of vectors is linearly independent; if the algorithm does quit early, say, with ˜ *qj* = 0, we can conclude that the original collection of vectors is linearly dependent (and indeed, that aj is a linear combination of *a1,...,aj−1*). Figure 1 illustrates the Gram–Schmidt algorithm for two 2-vectors. The top row shows the original vectors; the middle and bottom rows show the ﬁrst and second iterations of the loop in the Gram–Schmidt algorithm, with the left-hand side showing the orthogonalization step, and the right-hand side showing the normalization step

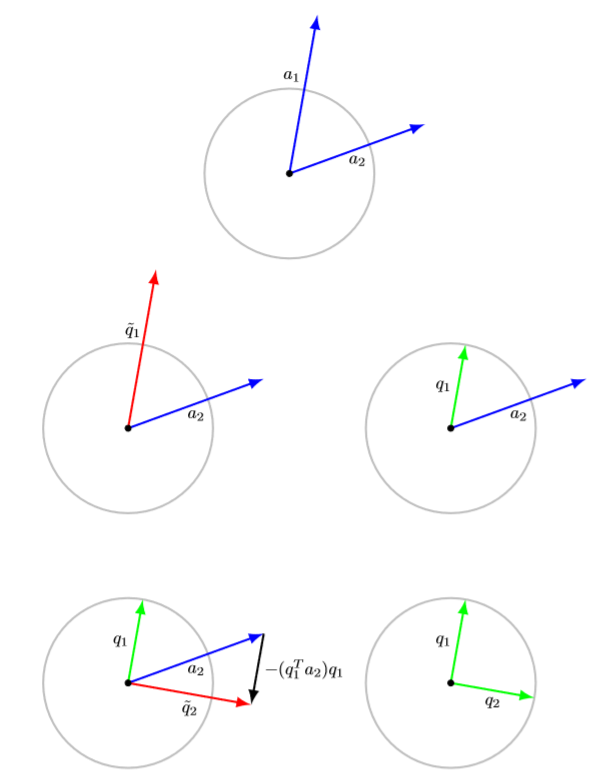


Figure1 Gram–Schmidt algorithm applied to two 2-vectors *a1, a2.* Top. The original vectors *a1* and *a2.* The gray circle shows the points with norm one. Middle left. The orthogonalization step in the ﬁrst iteration yields ˜ *q1* = *a1*. Middle right. The normalization step in the ﬁrst iteration scales ˜ *q1* to have norm one, which yields *q1*. Bottom left. The orthogonalization step in the second iteration subtracts a multiple of *q1* to yield the vector ˜ *q2*, which is orthogonal to q1. Bottom right. The normalization step in the second iteration scales ˜ *q2* to have norm one, which yields *q2*.

**Chapter 2 lab**

**2.1 The Gram–Schmidt process**

The application of the Gram–Schmidt process to the column vectors of a full column [rank](https://en.wikipedia.org/wiki/Rank_(linear_algebra)) [matrix](https://en.wikipedia.org/wiki/Matrix_(mathematics)) yields the [QR decomposition](https://en.wikipedia.org/wiki/QR_decomposition) (it is decomposed into an [orthogonal](https://en.wikipedia.org/wiki/Orthogonal_matrix) and a [triangular matrix](https://en.wikipedia.org/wiki/Triangular_matrix)).

We define the [projection](https://en.wikipedia.org/wiki/Projection_(linear_algebra)) [operator](https://en.wikipedia.org/wiki/Operator_(mathematics)) by

Proju<**v**> = u

Where <u, v>  denotes the [inner product](https://en.wikipedia.org/wiki/Inner_product) of the vectors u and v:v: <u, v> = uTv(Dot product) (or <u, v>=u\*v  for the complex case). This operator projects the vector v orthogonally onto the line spanned by vector u. If u = 0, we define proj0(v):=0 . i.e., the projection map pro0 is the zero map, sending every vector to the zero vector.

The Gram–Schmidt process then works as follows:

*u1 = v1,*  e=

*u1 = v2 – proju1(v2), e=*

*u1 = v3 – proju1(v3) – proju2(v3) e=*

*u1 = v4 – proju1(v4) – proju2(v4) – proju3(v4) e=*

*uk = vk – e=*

The sequence *u1, ... , uk* is the required system of orthogonal vectors, and the normalized vectors *e1, ... , ek* form an [orthonormal](https://en.wikipedia.org/wiki/Orthonormal) set. The calculation of the sequence *u1, ..., uk*is known as Gram–Schmidt [orthogonalization](https://en.wikipedia.org/wiki/Orthogonalization), while the calculation of the sequence *e1,..., ek* is known as Gram-Schmidt [orthonormalization](https://en.wikipedia.org/wiki/Orthonormalization) as the vectors are normalized.

**2.2 Numerical stability**

When this process is implemented on a computer, the vectors **uk** are often not quite orthogonal, due to [rounding errors](https://en.wikipedia.org/wiki/Round-off_error). For the Gram–Schmidt process as described above (sometimes referred to as "classical Gram–Schmidt") this loss of orthogonality is particularly bad; therefore, it is said that the (classical) Gram–Schmidt process is [numerically unstable](https://en.wikipedia.org/wiki/Numerical_stability).

The Gram–Schmidt process can be stabilized by a small modification; this version is sometimes referred to as modified Gram-Schmidt or MGS. This approach gives the same result as the original formula in exact arithmetic and introduces smaller errors in finite-precision arithmetic. Instead of computing the vector **uk** as

***uk*** *=* ***vk*** *– proju1(****vk****) - proju2(****vk****)-···-projuk-1(****vk****)*

it is computed as

***uk****(1) =* ***vk*** *– proju1(****vk****),*

***uk****(2) =* ***uk****(1) – proju1(****uk(1)****),*

*···*

*uk(k-2) = uk(k-3) – projuk-2(uk(k-3)),*

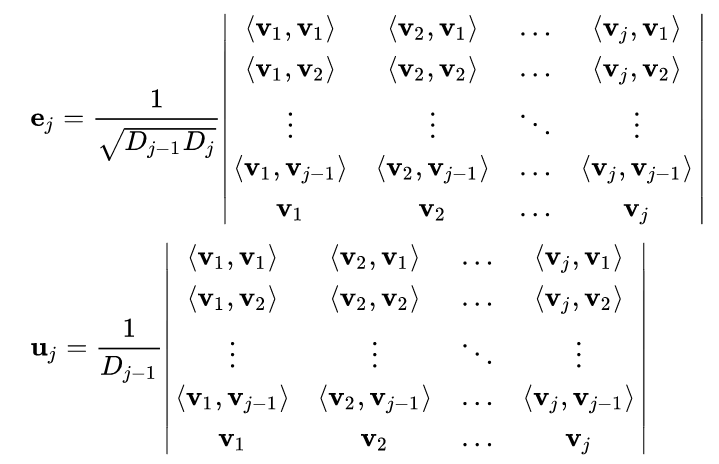
*uk(k-1) = uk(k-3) – projuk-1(uk(k-2)),*

Each step finds a vector *uk(i)*orthogonal to *uk(i-1)*. Thus *uk(i)*is also orthogonalized against any errors introduced in computation of *uk(i-1)*

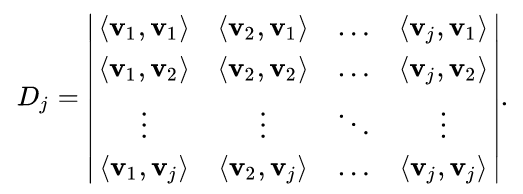
This method is used in the previous animation, when the intermediate v'3 vector is used when orthogonalizing the blue vector v3.

**2.3 Determinant formula**

The result of the Gram–Schmidt process may be expressed in a non-recursive formula using [determinants](https://en.wikipedia.org/wiki/Determinant).



where D 0=1 and, for j ≥ 1, D j is the [Gram determinant](https://en.wikipedia.org/wiki/Gram_determinant)

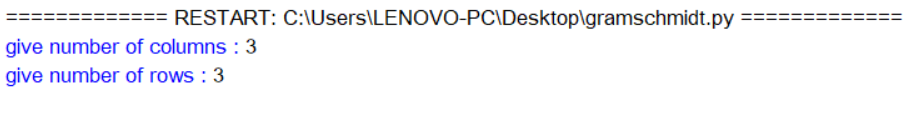


Note that the expression for *uk* is a "formal" determinant, i.e. the matrix contains both scalars and vectors; the meaning of this expression is defined to be the result of a [cofactor expansion](https://en.wikipedia.org/wiki/Laplace_expansion) along the row of vectors.

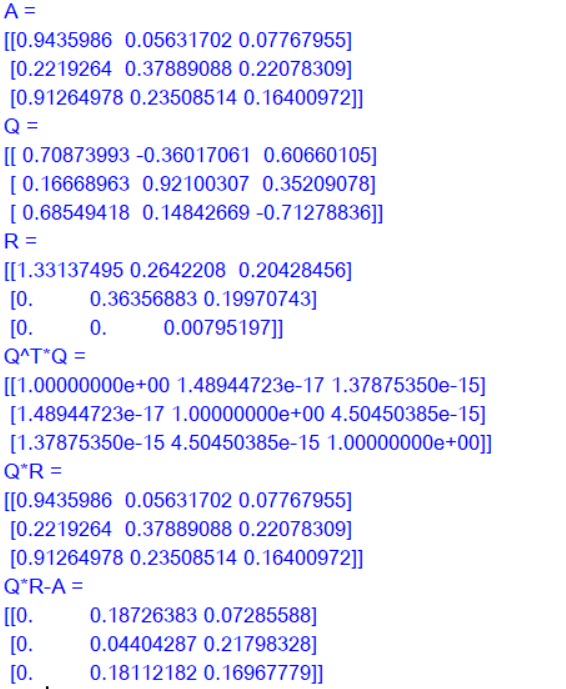
The determinant formula for the Gram-Schmidt is computationally slower (exponentially slower) than the recursive algorithms described above; it is mainly of theoretical interest.

**Chapter 3 Result**

When we run the program, we need to input the columns and rows:



And we can get the result:



**Appendix**

import numpy as np

def gramschmidt(A):

"""

Applies the Gram-Schmidt method to A

and returns Q and R, so Q\*R = A.

"""

R = np.zeros((A.shape[1], A.shape[1]))

Q = np.zeros(A.shape)

for k in range(0, A.shape[1]):

R[k, k] = np.sqrt(np.dot(A[:, k], A[:, k]))

Q[:, k] = A[:, k]/R[k, k]

for j in range(k+1, A.shape[1]):

R[k, j] = np.dot(Q[:, k], A[:, j])

A[:, j] = A[:, j] - R[k, j]\*Q[:, k]

return Q, R

def main():

"""

Prompts for n and generates a random matrix.

"""

cols = input('give number of columns : ')

rows = input('give number of rows : ')

A = np.random.rand(int(rows), int(cols))

print ('A = ')

print (A)

Q, R = gramschmidt(A)

print ('Q = ')

print (Q)

print ('R = ')

print (R)

print ('Q^T\*Q = ')

print (np.dot(Q.transpose(), Q))

print ('Q\*R =')

print (np.dot(Q, R))

print ('Q\*R-A = ')

print (np.dot(Q, R)-A)

main()