Sandra Babyale hwrk5

February 19, 2022

[1]:
```
import numpy as np
from numpy import *

from numpy.random import rand
```

1 Homework #5

1.1 Problem #1

Each matrix $A \in \mathbf{C}^{m \times n}$ with rank r can be categorized as one of four types:

Type I

A is non-singular

$m = n = r$

Type II

A has full column rank but not full row rank

$m > n = r$

Type III

A has full row rank but not full column rank

$m = r < n$

Type IV

A has neither full row rank or full column rank

$m > r$ and $n > r$

(a) For each type matrix, determine the dimensions of Q and R for the **reduced** QR factorization. Describe the structure of the matrix R in terms of a square, upper triangular block and a non-zero rectangular block. Provide the dimensions of each of the sub-blocks that appear in R for each matrix type.

(b) Repeat the above for the **full** QR factorization. Describe the structure of the matrix R in terms of a square, upper triangular block, a non-zero rectangular block and one or more zero sub-blocks. Provide the dimensions of each of the sub-blocks that appear in R for each matrix type.

(c) For the full factorization, from what space are any additional vectors needed to extend Q taken? **Hint:** Consider what $Q^T A$ has to be to properly recover R.

Solution (a)

Type I

Since $m = n = r$, then Q is $m \times m$ and R is also $m \times m$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} \end{bmatrix}$ where $\bar{R}$ is a $m \times m$ square upper triangular block.

Type II

Since $m > n = r$ Q is $m \times r$ and R is $r \times r$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} \end{bmatrix}$ where $\bar{R}$ is a $r \times r$ square upper triangular block.

Type III

Since $m = r < n$, then Q is $m \times m$ and R is $m \times n$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} & M \end{bmatrix}$ where $\bar{R}$ is a $r \times r$ square upper triangular block and M is a $r \times (n - r)$ non-zero rectangular block.

Type IV

Since $m > r$ and $n > r$, then Q is $m \times r$ and R is $r \times n$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} & M \end{bmatrix}$ where $\bar{R}$ is a $r \times r$ square upper triangular block and M is a $r \times (n-r)$ non-zero rectangular block.

(b)

Type I

Since $m = n = r$, then Q is $m \times m$ and R is also $m \times m$.

The structure of matrix R is given by; $R = \begin{bmatrix} \bar{R} \end{bmatrix}$ where $\bar{R}$ is a $m \times m$ square upper triangular block.

Type II

Since $m > n = r$ Q is $m \times m$ and R is $m \times r$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} \\ O \end{bmatrix}$ where $\bar{R}$ is a $r \times r$ square upper triangular block and \$ O \$ is a zero $(m-r) \times r$ block .

Type III

Since $m = r < n$, then Q is $m \times m$ and R is $m \times n$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} & M \end{bmatrix}$ where $\bar{R}$ is a $r \times r$ square upper triangular block and M is a $r \times (n-r)$ non-zero rectangular block.

Type IV

Since $m > r$ and $n > r$, then Q is $m \times m$ and R is $m \times n$.

The structure of matrix R is given by;

$R = \begin{bmatrix} \bar{R} & M \\ O_1 & O_2 \end{bmatrix}$ where $\bar{R}$ is a $r \times r$ square upper triangular block , M is a $r \times (n-r)$ non-zero rectangular block, O_1 is a $(m-r) \times r$ zero sub-block and O_2 is a $(m-r) \times (n-r)$ zero sub-block .

(c)

For the full factorization, if we are to properly recover R that is; $Q^T A = R$, then any additional vectors needed to extend Q are taken from the normalized basis of the null space of Q^T.

1.2 Problem #2 : Reduced factorizations

The Gram-Schmidt code demonstrated in class was limited to matrices A of Type I or Type II only. If the $n > m$ or the columns were linearly dependent, the code would return with an error. The goal of this problem is to fix the Modified Gram-Schmidt code so that it can return **reduced** factorizations for all four types of matrices.

The code below includes the Modified Gram-Schmidt code (called `MGS` here) demonstrated in class, in addition to a "wrapper" function which we will use to actually call the MGS routine. This wrapper function, called `QR` provides the user interface to the `MGS` code. For this problem, you will only need to modify the `MGS` to handle Type III and IV matrices.

Also below are two routines which can be used to select matrices of types I, II, III or IV, and to display results.

Task **Fix** the routine MGS so that we can get rid of the two `assert` statements in the code. The resulting code should be able to produce reduced factorizations for all four matrix types.

[2]:
```
fstr = {'float' : "{:>12.8f}".format}
set_printoptions(formatter=fstr)

def display_mat(msg,A):
    print(msg)
    display(A)
    print("")
```

The following routine is used to select sample matrices of four different types.

[3]:
```
def matrix_example(mat_type):

    # Type I : m == n == r
    A1 = np.array(np.mat('1,2,-1; 3,7,4; 5,6,4'),dtype='float')

    # Type II : m > n=r   (full column rank)
    A2 = np.array(np.mat('1,2; 3,4; 5,-1'),dtype='float')

    # Type III : m = r < n (full row rank)
    A3 = np.array(np.mat('1,2,-1,7; 3,7,4,9'),dtype='float')
```

```
    # Type IV :  m > r and n > r
    A4 = np.array(np.mat('1, 3, 5; 1,1,1; 2,4,6'),dtype='float')

    A_choice = (A1, A2, A3, A4)

    assert mat_type > 0 and mat_type < 5, "mat_type must be in [1,4]"

    return A_choice[mat_type-1]
```

This the Modified Gram-Schmidt (MGS) routine demonstrated in class. It only handles reduced factorizations for Type I and Type II matrices. Matrices of Type III and IV will return an error. In Problem #2, you will modify the Gram-Schmidt routine below to return reduced QR factorizations for all four matrix types.

[4]:
```
def MGS(A,tol=1e-12):
    m,n = A.shape
    R = np.zeros((n,n))
    Q = np.zeros((m,n))
    V = A.copy()
    P = array([True]*n, dtype=bool)  # Hint!
    # Loop over all columns of V.
    for i in range(n):
        # Assign qi to unit vector in direction vi
        v = V[:,i:i+1]
        R[i,i] = np.linalg.norm(v,2)
        if R[i,i]<tol:
            P[i] = False

        elif R[i,i]==0:
            P[i] = False

        else :
            #P[i] = 1
            qi = v/R[i,i]
            Q[:,i:i+1] = qi
            for j in range(i+1,n):
                vj = V[:,j:j+1]
                R[i,j] = vj.T@qi
                V[:,j:j+1] = vj - R[i,j]*qi

    return Q,R,P
```

In this assigment, you will not call Gram-Schmidt directly, but rather call the `QR` wrapper routine below. By default, this routine will return a reduced factorization. To get the full factorization set the mode to `full`. In Problem #3, you will modify this code to return a full factorization for all four matrix types.

[5]:
```
# "Wrapper" function for Gram-Schmidt code, above.
def QR(A,mode='reduced'):

    Q1,R1,P = MGS(A)
    Q = Q1[:,P]
    R = R1[P,:]
    #m,r = Q.shape
    #r,n = R.shape
    m,n = A.shape
    r=(P==True).sum()

    B = rand(m,abs(m-r))

    ## Extract columns and rows of Q and R using Boolean vector.
    if mode == 'full':
        Q2 = np.column_stack([Q,B])
        Q3,R3,P = MGS(Q2)
        R0 = np.zeros((m-r,n))
        R1 = np.row_stack((R,R0))
        R=R1
        Q=Q3
    return Q,R
```

You can test out your code with the following test cases. Your code should return the correct reduced QR factorizatoin for all four matrix types.

[6]:
```
#for i in range(1,5):

    #A = matrix_example(i)  # Should work for matrices of Types I, II, III and␣
 ↪IV.

    #Q,R = QR(A,)
    #display_mat("A1 = ",A)
    #display_mat("Q = ",Q)
    #display_mat("R = ",R)
    #display_mat("QR = ",Q@R)
    #display_mat("Q^TQ = ",Q.T@Q)
```

[7]:
```
#for i in range(1,5):
A = matrix_example(1)
Q,R = QR(A,)
display_mat("A1 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)
```

```
A = matrix_example(2)
Q,R = QR(A,)
display_mat("A2 = ",A)
display_mat("Q = ",Q.T@Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)
```

```
A1 =

array([[  1.00000000,   2.00000000,  -1.00000000],
       [  3.00000000,   7.00000000,   4.00000000],
       [  5.00000000,   6.00000000,   4.00000000]])


Q =

array([[  0.16903085,   0.16426846,  -0.97182532],
       [  0.50709255,   0.83100515,   0.22866478],
       [  0.84515425,  -0.53145678,   0.05716620]])


R =

array([[  5.91607978,   8.95863510,   5.23995638],
       [  0.00000000,   2.95683228,   1.03392501],
       [  0.00000000,   0.00000000,   2.11514922]])


QR =

array([[  1.00000000,   2.00000000,  -1.00000000],
       [  3.00000000,   7.00000000,   4.00000000],
       [  5.00000000,   6.00000000,   4.00000000]])


Q^TQ =

array([[  1.00000000,  -0.00000000,  -0.00000000],
       [ -0.00000000,   1.00000000,  -0.00000000],
       [ -0.00000000,  -0.00000000,   1.00000000]])


A2 =

array([[  1.00000000,   2.00000000],
       [  3.00000000,   4.00000000],
       [  5.00000000,  -1.00000000]])


Q =
```

```
array([[  1.00000000,   0.00000000],
       [  0.00000000,   1.00000000]])


R =

array([[  5.91607978,   1.52127766],
       [  0.00000000,   4.32269757]])


QR =

array([[  1.00000000,   2.00000000],
       [  3.00000000,   4.00000000],
       [  5.00000000,  -1.00000000]])


Q^TQ =

array([[  1.00000000,   0.00000000],
       [  0.00000000,   1.00000000]])
```

[8]:
```
A = matrix_example(3)
Q,R = QR(A,)
display_mat("A3 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)

A = matrix_example(4)
Q,R = QR(A,)
display_mat("A4 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)
```

```
A3 =

array([[  1.00000000,   2.00000000,  -1.00000000,   7.00000000],
       [  3.00000000,   7.00000000,   4.00000000,   9.00000000]])


Q =

array([[  0.31622777,  -0.94868330],
       [  0.94868330,   0.31622777]])


R =
```

```
array([[  3.16227766,   7.27323862,   3.47850543,  10.75174404],
       [  0.00000000,   0.31622777,   2.21359436,  -3.79473319]])


QR =

array([[  1.00000000,   2.00000000,  -1.00000000,   7.00000000],
       [  3.00000000,   7.00000000,   4.00000000,   9.00000000]])


Q^TQ =

array([[  1.00000000,   0.00000000],
       [  0.00000000,   1.00000000]])


A4 =

array([[  1.00000000,   3.00000000,   5.00000000],
       [  1.00000000,   1.00000000,   1.00000000],
       [  2.00000000,   4.00000000,   6.00000000]])


Q =

array([[  0.40824829,   0.70710678],
       [  0.40824829,  -0.70710678],
       [  0.81649658,  -0.00000000]])


R =

array([[  2.44948974,   4.89897949,   7.34846923],
       [  0.00000000,   1.41421356,   2.82842712]])


QR =

array([[  1.00000000,   3.00000000,   5.00000000],
       [  1.00000000,   1.00000000,   1.00000000],
       [  2.00000000,   4.00000000,   6.00000000]])


Q^TQ =

array([[  1.00000000,  -0.00000000],
       [ -0.00000000,   1.00000000]])
```

[ ]:

1.3 Problem #3 : Full factorizations

Finally, we want to be able to create full QR factorization. To do this, we will need to be able to extend Q with additional vectors in the space orthogonal to Col(A).

To get these additional vectors, we will use the following fact about random matrices

Random matrices Given a matrix $B \in \mathbb{R}^{m \times n}$ whose entries are taken from a uniformly distributed random sampling, B will have full rank (with probability 1).

We can use the above fact to find vectors in the space orthogonal to Col(A) by orthogonalizing a random matrix B against vectors $\mathbf{q}_i, i = 1, 2, \ldots, r$.

In NumPy, we can construct a random matrix with the following code:

```
from numpy.random import rand
B = rand(m,n)
```

Task Complete the `QR` code above to return a full QR factorization for each of the four types of matrices. **Hint:** use the random matrix idea and two passes of MGS.

[9]:
```
#for i in range(1,5):

    #A = matrix_example(i)  # Should work for matrices of Types I, II, III and IV.

    #Q,R = QR(A,'full')
    #display_mat("A = ",A)
    #display_mat("Q = ",Q)
    #display_mat("R = ",R)
    #display_mat("QR = ",Q@R)
    #display_mat("Q^TQ = ",Q.T@Q)
```

[10]:
```
A = matrix_example(1)
Q,R = QR(A,'full')
display_mat("A1 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)

A = matrix_example(2)
Q,R = QR(A,'full')
display_mat("A2 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)
```

```
A1 =
```

```
array([[  1.00000000,   2.00000000,  -1.00000000],
       [  3.00000000,   7.00000000,   4.00000000],
       [  5.00000000,   6.00000000,   4.00000000]])

Q =

array([[  0.16903085,   0.16426846,  -0.97182532],
       [  0.50709255,   0.83100515,   0.22866478],
       [  0.84515425,  -0.53145678,   0.05716620]])

R =

array([[  5.91607978,   8.95863510,   5.23995638],
       [  0.00000000,   2.95683228,   1.03392501],
       [  0.00000000,   0.00000000,   2.11514922]])

QR =

array([[  1.00000000,   2.00000000,  -1.00000000],
       [  3.00000000,   7.00000000,   4.00000000],
       [  5.00000000,   6.00000000,   4.00000000]])

Q^TQ =

array([[  1.00000000,  -0.00000000,  -0.00000000],
       [ -0.00000000,   1.00000000,  -0.00000000],
       [ -0.00000000,  -0.00000000,   1.00000000]])

A2 =

array([[  1.00000000,   2.00000000],
       [  3.00000000,   4.00000000],
       [  5.00000000,  -1.00000000]])

Q =

array([[  0.16903085,   0.40318739,   0.89937117],
       [  0.50709255,   0.74688811,  -0.43013404],
       [  0.84515425,  -0.52877035,   0.07820619]])

R =

array([[  5.91607978,   1.52127766],
       [  0.00000000,   4.32269757],
       [  0.00000000,   0.00000000]])

QR =
```

```
array([[  1.00000000,   2.00000000],
       [  3.00000000,   4.00000000],
       [  5.00000000,  -1.00000000]])


Q^TQ =

array([[  1.00000000,   0.00000000,  -0.00000000],
       [  0.00000000,   1.00000000,   0.00000000],
       [ -0.00000000,   0.00000000,   1.00000000]])
```

[11]:
```
A = matrix_example(3)
Q,R = QR(A,'full')
display_mat("A3 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)

A = matrix_example(4)
Q,R = QR(A,'full')
display_mat("A4 = ",A)
display_mat("Q = ",Q)
display_mat("R = ",R)
display_mat("QR = ",Q@R)
display_mat("Q^TQ = ",Q.T@Q)
```

```
A3 =

array([[  1.00000000,   2.00000000,  -1.00000000,   7.00000000],
       [  3.00000000,   7.00000000,   4.00000000,   9.00000000]])


Q =

array([[  0.31622777,  -0.94868330],
       [  0.94868330,   0.31622777]])


R =

array([[  3.16227766,   7.27323862,   3.47850543,  10.75174404],
       [  0.00000000,   0.31622777,   2.21359436,  -3.79473319]])


QR =

array([[  1.00000000,   2.00000000,  -1.00000000,   7.00000000],
       [  3.00000000,   7.00000000,   4.00000000,   9.00000000]])


Q^TQ =
```

```
array([[  1.00000000,   0.00000000],
       [  0.00000000,   1.00000000]])


A4 =

array([[  1.00000000,   3.00000000,   5.00000000],
       [  1.00000000,   1.00000000,   1.00000000],
       [  2.00000000,   4.00000000,   6.00000000]])


Q =

array([[  0.40824829,   0.70710678,   0.57735027],
       [  0.40824829,  -0.70710678,   0.57735027],
       [  0.81649658,   0.00000000,  -0.57735027]])


R =

array([[  2.44948974,   4.89897949,   7.34846923],
       [  0.00000000,   1.41421356,   2.82842712],
       [  0.00000000,   0.00000000,   0.00000000]])


QR =

array([[  1.00000000,   3.00000000,   5.00000000],
       [  1.00000000,   1.00000000,   1.00000000],
       [  2.00000000,   4.00000000,   6.00000000]])


Q^TQ =

array([[  1.00000000,   0.00000000,  -0.00000000],
       [  0.00000000,   1.00000000,  -0.00000000],
       [ -0.00000000,  -0.00000000,   1.00000000]])
```

1.4 Problem #4 : Uniqueness of the QR factorization

Compare your results above to the QR factorization routine in NumPy. What can you say about the uniqueness of the reduced and full QR factorization? Can you comment on the NumPy routine?

[12]:
```
A = matrix_example(2)

q,r = np.linalg.qr(A)

display_mat("q = ",q)
display_mat("r = ",r)
A = matrix_example(4)
```

```
q,r = np.linalg.qr(A)

display_mat("q = ",q)
display_mat("r = ",r)
```

```
q =
```

```
array([[ -0.16903085,  -0.40318739],
       [ -0.50709255,  -0.74688811],
       [ -0.84515425,   0.52877035]])
```

```
r =
```

```
array([[ -5.91607978,  -1.52127766],
       [  0.00000000,  -4.32269757]])
```

```
q =
```

```
array([[ -0.40824829,   0.70710678,  -0.57735027],
       [ -0.40824829,  -0.70710678,  -0.57735027],
       [ -0.81649658,  -0.00000000,   0.57735027]])
```

```
r =
```

```
array([[ -2.44948974,  -4.89897949,  -7.34846923],
       [  0.00000000,   1.41421356,   2.82842712],
       [  0.00000000,   0.00000000,   0.00000000]])
```

Except for the difference in the signs on the entries of Q and R, the QR factorization routine in NumPy gives the same results as the corresponding reduced or full QR factorization for all the types.

For types I and II that is matrices with $m \geq n$ and full column rank, both the reduced and full QR factorization will always be unique. However, for types III and IV the reduced QR factorization is unique but the full QR factorization is not unique since we get other columns of Q from the null space of Q^T and also because the columns in the original matrices are not linearly independent.

1.5 Problem #5

Suppose we have planes $P^{(1)}$ and $P^{(2)}$ in $\mathbb{R}^3$ given by

$$P^{(1)} = \langle \mathbf{x}^{(1)}, \mathbf{y}^{(1)} \rangle, \qquad P^{(2)} \langle \mathbf{x}^{(2)}, \mathbf{y}^{(2)} \rangle \tag{1}$$

where $\mathbf{x}^{(1)}, \mathbf{y}^{(1)} \in \mathbb{R}^3$ are linearly independent and $\mathbf{x}^{(2)}, \mathbf{y}^{(2)} \in \mathbb{R}^3$ are linearly independent.

(a) How can you find $P^{(1)} \cap P^{(2)}$ using a QR decomposition? **Hint:** Use the full QR decomposition.

(b) Find the intersection of the planes described by the following two sets of vectors.

$$\mathbf{x}^{(1)} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}, \qquad \mathbf{y}^{(1)} = \begin{bmatrix} 3 \\ 0 \\ 5 \end{bmatrix}, \qquad \mathbf{x}^{(2)} = \begin{bmatrix} 7 \\ -3 \\ 1 \end{bmatrix}, \qquad \mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 4 \\ 8 \end{bmatrix} \tag{2}$$

(c) Construct projections P_1 onto $P^{(1)}$ and P_2 onto $P^{(2)}$ and show that for some $\mathbf{v} \in P^{(1)} \cap P^{(2)}$, we have $P_1\mathbf{v} = P_2\mathbf{v} = \mathbf{v}$. Verify the result by displaying

$$\|P_1\mathbf{v} - \mathbf{v}\|, \qquad \|P_2\mathbf{v} - \mathbf{v}\| \tag{3}$$

Hint: You only need to verify this numerically for one $\mathbf{v}$. Use the QR decompositions to easily compute the projections.

1.5.1 Solution

(a)

Let $P = P^{(1)} \cap P^{(2)}$. This means that P is a straight line which is orthogonal to the cross products $x^{(1)} \times y^{(1)}$ and $x^{(2)} \times y^{(2)}$.

From QR decomposition, we notice that the columnns q_j for $j > n$(number of columns) of matrix Q are in the range(A). This means that the third column of Q with respect to plane $P^{(1)}$ should be equivalent to $x^{(1)} \times y^{(1)}$, similarly the third column of Q with respect to plane $P^{(2)}$ should be equivalent to $x^{(2)} \times y^{(2)}$. The QR decompositon of the two third columns from the $Q's$ with respect to two planes above results into a vector $v \in P$ and this occupies the last column of this new Q and v is orthogonal to the first two columns of new Q.

(b)

[13]:
```
#
P_1 = np.array(np.mat('1,3;2,0; -1,5'),dtype='float') #plane 1
P_2 = np.array(np.mat('7,0;-3,4;1,8'),dtype='float') #plane 2

q1,r1 = QR(P_1,'full')
q2,r2 = QR(P_2,'full')

m,n =q1.shape

q13 = q1[:,n-1:n]
q23 = q2[:,n-1:n]

qo = np.column_stack([q13,q23])
q,r = QR(qo,'full')
 # The third column of q gives the vector in the intersection of the two planes
display_mat("The intersection P = ",q[:,n-1:n])
```

```
The intersection P =

array([[  0.57735027],
       [  0.11547005],
       [  0.80829038]])
```

(c)

[14]:
```
v = q[:,n-1:n]
P1 = q1[:,0:2]@q1[:,0:2].T #Projection P1
P2 = q2[:,0:2]@q2[:,0:2].T #Projection P2
display_mat("Projection P_1 onto P^(1) = ",P1)
display_mat("Projection P_2 onto P^(2) = ",P2)
display_mat("v = ",v)
display_mat("P1v = ",P1@v)
display_mat("P2v = ",P2@v)
display_mat(" ||P1v-v|| = ",np.linalg.norm(P1@v-v))
display_mat("||P2v-v|| = ",np.linalg.norm(P2@v-v))
```

```
Projection P_1 onto P^(1) =

array([[  0.50000000,   0.40000000,   0.30000000],
       [  0.40000000,   0.68000000,  -0.24000000],
       [  0.30000000,  -0.24000000,   0.82000000]])


Projection P_2 onto P^(2) =

array([[  0.83333333,  -0.33333333,   0.16666667],
       [ -0.33333333,   0.33333333,   0.33333333],
       [  0.16666667,   0.33333333,   0.83333333]])


v =

array([[  0.57735027],
       [  0.11547005],
       [  0.80829038]])


P1v =

array([[  0.57735027],
       [  0.11547005],
       [  0.80829038]])


P2v =

array([[  0.57735027],
       [  0.11547005],
       [  0.80829038]])


 ||P1v-v|| =

1.0295784775289034e-15
```

```
||P2v-v|| =
2.8609792490763985e-16
```

From above, since $\|P_1 v - v\| = 0$ and $\|P_2 v - v\| = 0$, then $P_1 v = P_2 v = v$.

[ ]: