Deep Dive

Minerva University

CS113: Linear Algebra

Prof. Levitt

April 14, 2023

Deep Dive

Part 1:

a) **Additive identity and additive inverse**

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Figure 1: Operations in $\mathbb{F}_2$

Additive identity is the value that when added to the element does not change its value

According to figure 1:

For 0: 0+0 = 0

For 1: 1+0 = 1

Therefore the additive identity is 0 because in any case when we add 0, the value of the original element does not change.

Additive inverse is the value that when added to the elements, the element changes its value.

According to figure 1:

For 0: 0+1 = 1;

For 1: 1+1 = 0

So in both cases when we add 1 to anything, the value changes. Therefore the additive inverse is 1.

b) Matrix multiplication

i) A =

[1 0 1]

[1 1 1]

B =

[1]

[1]

[1]

AB=

$[1x1 + 0X1 + 1x1]$

$[1x1 + 1x1 + 1x1]$

When adding the elements we get the following

$[1 + 0 + 1] = 0$

$[1 + 1 + 1] = 1$

Therefore AB =

[0]

[1]

ii)

A =

[1 1]

[1 0]

[0 1]

B =

[0 1]

[1 1]

AB =

$[1x0 + 1x1,\ 1x1 + 1x1]$

$[0x1 + 0x1,\ 1x1 + 0x1]$

$[0x0 + 1x1,\ 0x1 + 1x1]$

$[0 + 1 \quad 1 + 1] = [1,\ 0]$

$[0 + 0 \quad 1 + 0] = [0,\ 1]$

$[0 + 1 \quad 1 + 0] = [1,\ 1]$

AB =

[1, 0]

[0, 1]

[1, 1]

iii) C =

$[0\ 1]^{-1}$

[1 1]

We augment the matrix with an inverse and reduce the first part to RREF to get an inverse on the right hand side.

$[0,\ 1|\ 1,\ 0\ ]$

$[1,\ 1|\ 0,\ 1\ ]$

$R2\ =\ R1\ +\ R2$

$[0,\ 1|\ 1,\ 0\ ]$

$[1,\ 0|\ 1,\ 1\ ]$

Now we need to switch rows

$R1\ =\ R2,\ R2\ =\ R1$

$[1,\ 0|\ 1,\ 1\ ]$

$[0,\ 1|\ 1,\ 0\ ]$

Therefore

$[0,\ 1]^{-1}\ =\ [1,\ 1]$

$[1,\ 1]\qquad [1,\ 0]$

Sage check: the same results are obtained

```
A = matrix(GF(2), [[1,0,1],[1,1,1]])
B = matrix(GF(2), [[1], [1], [1]])
print('AxB  ')
print(A*B)

A2 = matrix(GF(2), [[1,1],[1,0],[0,1]])
B2 = matrix(GF(2), [[0,1],[1,1]])
print('A2xB2  ')
print(A2*B2)

C = matrix(GF(2), [[0, 1], [1,1]])
Ci = C.inverse()
print('C inverse is')
print(Ci)
```

```
AxB
[0]
[1]
A2xB2
[1 0]
[0 1]
[1 1]
C inverse is
[1 1]
[1 0]
```

Figure 2: *code to find the inverse of the matrix*

c) To solve the linear system of equations we need to put the coefficients in the matrix and augment it with the solution vector.. After that we reduce the matrix to RREF, if we get an identity matrix on the right hand side we will get the solution vector on the left hand side .

[1, 1, 1| 0]

[1, 1, 0, | 1]

[1, 0, 1|1]

R3 = R3+ R2

[1, 1, 1| 0]

[1, 1, 0, | 1]

[0, 1, 1|0]

R1 = R1+ R3

[1, 0, 0| 0]

[1, 1, 0, | 1]

[0, 1, 1|0]

R2 = R2+R1

[1, 0, 0| 0]

[0, 1, 0, | 1]

[0, 1, 1|0]

R3 = R3+ R2

[1, 0, 0| 0]

[0, 1, 0, | 1]

[0, 0, 1|1]

Result:

X1 = 0

X2 = 1

X3 = 1

Sage check: the same result is obtained

```
M = matrix(GF(2), [[1,1,1],[1,1,0],[1, 0, 1]])
b = vector([1, 1, 1])
M = M.augment(b,subdivide = True)
M.rref()
```

```
[1 0 0|1]
[0 1 0|0]
[0 0 1|0]
```

Figure 3: *code to find the solution to RREF*

d) Correcting codes.

i. If the sender sends the vector (x, x, x) instead of x, it means that the value x is repeated three times.

Case 1:

x x x

[0 0 0]

X = 0 based on the majority vote.

Case 2:

x x x

[1 0 1]

X = 1 based on the majority vote (1 of the bits got flipped from 1 to 0)

Case 3:

x x x

[0 0 1]

X = 0 based on the majority vote (1 of the bits got flipped from 0 to 1)

Correct message is [0, 1, 0]

To find the correct value of x in each of the 3 cases, we analyzed each vector and made the individual decision based on the majority vote.

What is the maximum number of flips it could detect?

In the given case we can detect only 1 bit per vector. We had 3 vectors so the total possible number of bits detected is $3x1 = 3$

From which we can infer that the maximum number of bits detected per vector is

$(n - 1)/2$

Where n is the length of the vector

The total number of the bits flipped is the number of bits flipped per vector multiplied by the number of vectors:

$m(n - 1)/2$

Where m is the number of vectors

Let's consider matrix A

[1, 0, 0, 1]

[1, 1, 0, 1]

[0, 0, 1, 1]

This is the nxm matrix where n is equal to the number of rows which is equal to the length of each vector and m is the number of columns which is equal to the number of vectors

$4(3 - 1)/2 = 4$

Therefore the total number of errors that we can detect in matrix A is 4.

If we make a correction, can we be certain that the corrected message is the original message? Give examples to support your conclusions.

No we cannot be certain that the corrected message is the right one because we do not know the probability of flip and we just assume that it is a) the same one for each bit b) it is about 0.5 so that we can make conclusion based on the majority vote

For example, the original message is x=1.

The vector representation of this message is (1, 1, 1). Due to errors during transmission, the received message is (0, 0, 0). By taking the majority vote of the repeated bits, we can make the decision that x=0. In this case, the correction is incorrect, and we cannot be certain that the corrected message is the original message.

ii)

$x1, x1, x2, x2, \ x1 + x2$

[1 1 1 1 | 0]

[0 1 0 0 | 0]

[1 0 0 0 | 1]

[0 1 1 1 | 1]

[0 1 0 0 | 1]

10

[0 1 0 1 | 0]

How should the receiver interpret the following vectors?

In this case we consider not only 1 column but 2 columns

We will again make the decision based on the majority vote:

X1 = 1

X2 = 0

X1 + X2 = 1

X1+x2 = max(v1,v2) + max(v3,v4)

How many flips could this method accurately detect?

The formula for the previous case will be modified by adding multiplication by 2 since we now have 2 columns representing the same bit.

$2n/2 * 2$

Where n is the length of the column (vector)

If we make a correction, can we be certain that the corrected message is the original message? Give examples to support your conclusions.

No, we cannot be certain what it is the original message for the same reasons as in the previous case. We can consider an extreme example when the probability of flip is 100%. In that case, the original message [1, 0] that should have been decoded as [1, 1, 0, 0, 1] will be decoded as [0,0,1,1,1] and it will be interpreted as [0, 1] which is not the original message. Basically if the number of flips is larger than the number of flips we can detect we will not be able to detect all the flips which will lead to the wrong result. Because of this limit of the error detection, there is always a chance that the message will be misinterpreted. However, in the case of parity codes

there is a higher chance that the original message if the corrected message because of the increased accuracy of detection, since we not only repeat the element more than once to decrease the probability of misinterpreting but we also introduce a new column for the sum of the 2 bits of the original message that gives us an additional insight into the nature of the original bits. For example, if in the case of x1 = 1, x2 = 1 one of the bits got flipped which resulted in x1 = 0, x2 = 1, we will have the sum to x1+x2 = 0 which is based on the original message, and it will give a hint that either both bits should be 1 or 0. Since we send not one vector but multiple, we can use this information to make an educated guess that the original message would be [1, 1].

e)

Repetition:

The repetition code method involves repeating each element of the message several times to create a longer codeword. The generator matrix for this method is constructed by creating a row vector of ones whose length is equal to the length of the message. This row vector is then repeated vertically to form a matrix with the same number of rows as the length of the message. If the message vector is

$x = [x1\ x2\ x3]$

then the generator matrix is constructed as G = [1 1 1],[1 1 1],[1 1 1]. We fill it with ones because anything multiplied by 1 will change its value to the value it was multiplied by. So we multiply G by [x1 x2 x3] to form the codeword matrix C.

For example, if the message vector is x = [0 1 1], then the generator matrix would be G = [1 1 1],[1 1 1],[1 1 1] times x = [0 1 1]. It will result in C = [0 0 0],[1 1 1],[1 1 1].

Parity codes:

For a parity code, we can use the same approach as we did for the repetition code. We need to repeat each data bit twice and add them together modulo 2 to get the parity bit. This can be represented by a generation matrix

$D = [1, 1][1, 1][0]$

where each row corresponds to a bit position in the codeword.

To encode a message using this approach, we can multiply the data bits by the generation matrix D to get the encoded message. For example, if our codeword is $x1 = 1$ and $x2 = 0$, the matrix G representing the message would be [1][0][1]. We can then multiply G by D to get the encoded message C [1, 1][0, 0][1].

f) Generation matrix H

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

i. What is the encoding of the message x = (0, 1, 0, 1)?

To encode the message x using the given generation matrix H, we simply multiply x by H to get the encoded message y:

$y = (1, 1, 1, 0, 0, 1, 1)$

```
In [4]  1 H = matrix(GF(2),[[1, 0, 0, 1],[0, 1 ,1, 0],[0, 0, 1, 1],[0, 1, 1, 1],[1, 0, 0, 0],[0, 1, 0, 0],[0, 0, 0, 1]])
        2 x = vector(GF(2),[0,1,0,1])
        3 y = H*x
        4 y

Run Code

Out [4]
          (1, 1, 1, 0, 0, 1, 1)
```

Figure 4: *code to find y vector*

ii. Write pseudocode for a procedure that decodes a codeword y. Use your procedure to decode y1 = (1, 1, 1, 0, 0, 1, 1).

To decode y1 we have to basically go backwards and find the solution for the system of equations that represent encoding each of the bits. To do so we append the y1 to the matrix H and reduce it to rref and get the x1 as the output

This is the pseudocode:

H = matrix[]

y = vector[]

H2 = H augment with y

R = H2 reduced to rref

print(r) #Check is the system is consistent

Result = (last bit of 1st row, last bit of 2nd row, last bit of 3rd row,last bit of 4th row)

Print result

```
In [41]
1 H = matrix(GF(2),[[1, 0, 0, 1],[0, 1 ,1, 0],[0, 0, 1, 1],[0, 1, 1, 1],[1, 0, 0, 0],[0, 1, 0, 0],[0, 0, 0, 1]])
2 y = vector(GF(2),[1, 1, 1, 0, 0, 1, 1])
3 H2 = H.augment(y,subdivide = True)
4 r = H2.rref()
5 result = r[0][4],r[1][4],r[2][4],r[3][4]
6 result
7
```

Run Code

```
Out [41]
(0, 1, 0, 1)
```

Figure 5: *code to find the original vector given y1 vector and matrix H*

As we can see we got the same output vector as our initial vector x1 which means that there were no errors in the encoding process.

iii. Suppose that you receive the message y2 = (0, 1, 1, 0, 0, 1, 1) via a noisy channel. Can you decode y2? Why or why not?

We will do the same manipulation as we did for the previous part.

```
In [46]
1 H = matrix(GF(2),[[1, 0, 0, 1],[0, 1 ,1, 0],[0, 0, 1, 1],[0, 1, 1, 1],[1, 0, 0, 0],[0, 1, 0, 0],[0, 0, 0, 1]])
2 y2 = vector(GF(2),[0, 1, 1, 0, 0, 1, 1])
3 H2 = H.augment(y2,subdivide = True)
4 r = H2.rref()
5 r
6
```

Run Code

```
Out [46]
[1 0 0 0|0]
[0 1 0 0|0]
[0 0 1 0|0]
[0 0 0 1|0]
[0 0 0 0|1]
[0 0 0 0|0]
[0 0 0 0|0]
```

Figure 6: *code to find the original vector given y2 vector given the matrix H*

We can see that this system has 0 solutions because we have a contradiction of 0 =1 in the 5th row. Therefore it is impossible to find the original x2 message using this method.

iv. To find the error let's assume at most one bit was flipped. Then $y2 = y1 + ei$ where ei is the ith standard basis vector in F^7. Why?

ei represents the error vector here and this is the vector added to y1 that changes its value. Since we have the information that this is the basis vector we can assume that it has one entry of 1 and the rest are zeros. We also know that zero is the additive identity thus when we add a zero to any value, that value will not get changed. At the same time we also know that 1 works in the opposite way, thus when we add 1 to any value it will change (which is equivalent to a bit flip). To understand how exactly ei looks we will compare the vectors y1 and y2 and based on the location of their different bit, we will know where should be the 1 in the error vector.

$y1 = (1, 1, 1, 0, 0, 1, 1)$

$y2 = (0, 1, 1, 0, 0, 1, 1)$

Based on the observation we can see that there is a difference in the first bit, therefore the error vector will be the following:

$ei = (1, 0, 0\,0, 0, 0, 0, 0)$

From this we can see that indeed $y2 = y1 + ei$

$(0, 1, 1, 0, 0, 1, 1) = (1, 1, 1, 0, 0, 1, 1) + (1, 0, 0\,0, 0, 0, 0, 0)$

We can do the same using computational tools

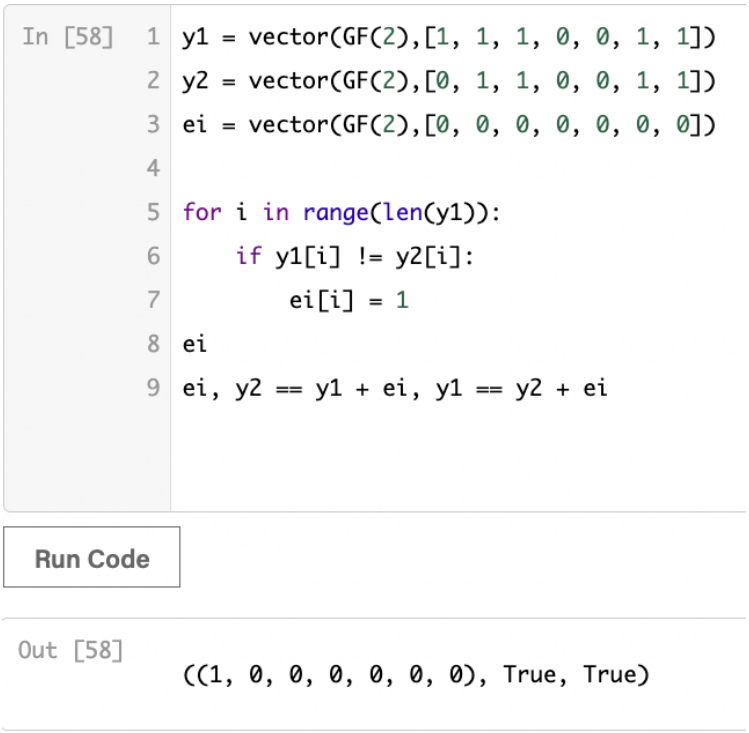


Figure 7: *code to find an error vector and prove the statements that y1+ei = y2 and y2+ei = 1*

v. This implies that $y1 = y2 + ei$ **Why?**

It is because the difference between y1 and y2 is the flipped first element. Thus when we switch the first element in y1 by adding the error vector to it, we will get the vector identical to y2. Similarly we can do the same thing in the reverse order. We can use the error vector ei to change the first element of y2 and make it identical to y1. We did the code for the previous part and saw that it is indeed true. To illustrate this even better we can think about the following example: $5 = 4 + 1$ is the same as $4 = 5 - 1$. When we operate with numbers in the regular system we have to switch the sign before 1 but when we operate with the number in the binary system the change of sign is not needed because we know that 1 is the additive inverse.

vi.Compute y2 + ej for j = 1 . . . m to find j so that y2 +ej is a codeword. Then the message is x such that Hx = y2 + ej . Use this method to decode y2.

```
In [125]
H = matrix(GF(2),[[1, 0, 0, 1],[0, 1 ,1, 0],[0, 0, 1, 1],[0, 1, 1, 1],[1, 0, 0, 0],[0, 1, 0, 0],[0, 0, 0, 1]])
y2 = vector(GF(2),[0, 1, 1, 0, 0, 1, 1])
e = matrix(GF(2),[[1, 0, 0, 0, 0, 0, 0],
     [0, 1, 0, 0, 0, 0, 0],
     [0, 0, 1, 0, 0, 0, 0],
     [0, 0, 0, 1, 0, 0, 0],
     [0, 0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 0, 0, 1]])

#loop throught all the basis vectors to find the possible vectors y1 given y2 and ej
possible_y1 = []
for i in e:
    possible_y1.append(y2+i)

rrefs = []
#check what of the cases are plausible
for i in range(7):
    H2 = H.augment(possible_y1[i],subdivide = True)
    rrefs.append('for flip in {}th bit'.format(i+1))
    rrefs.append(H2.rref())

rrefs
```

Figure 8: *code to find possible original vectors*

```
Out [125]
[
                        [1 0 0 0|0]                         [1 0 0 0|0]
                        [0 1 0 0|1]                         [0 1 0 0|0]
                        [0 0 1 0|0]                         [0 0 1 0|0]
                        [0 0 0 1|1]                         [0 0 0 1|0]
                        [0 0 0 0|0]                         [0 0 0 0|1]
                        [0 0 0 0|0]                         [0 0 0 0|0]
'for flip in 1th bit', [0 0 0 0|0], 'for flip in 2th bit', [0 0 0 0|0],

                        [1 0 0 0|0]                         [1 0 0 0|0]
                        [0 1 0 0|0]                         [0 1 0 0|0]
                        [0 0 1 0|0]                         [0 0 1 0|0]
                        [0 0 0 1|0]                         [0 0 0 1|0]
                        [0 0 0 0|1]                         [0 0 0 0|1]
                        [0 0 0 0|0]                         [0 0 0 0|0]
'for flip in 3th bit', [0 0 0 0|0], 'for flip in 4th bit', [0 0 0 0|0],

                        [1 0 0 0|1]                         [1 0 0 0|0]
                        [0 1 0 0|1]                         [0 1 0 0|0]
                        [0 0 1 0|0]                         [0 0 1 0|0]
                        [0 0 0 1|1]                         [0 0 0 1|0]
                        [0 0 0 0|0]                         [0 0 0 0|1]
                        [0 0 0 0|0]                         [0 0 0 0|0]
'for flip in 5th bit', [0 0 0 0|0], 'for flip in 6th bit', [0 0 0 0|0],

                        [1 0 0 0|0]
                        [0 1 0 0|0]
                        [0 0 1 0|0]
                        [0 0 0 1|0]
                        [0 0 0 0|1]
                        [0 0 0 0|0]
'for flip in 7th bit', [0 0 0 0|0]
]
```

Figure 9: *output of the code form the figure 8*

As we can see, the only consistent systems are those that represent the flip in the 1st bit and the 5th bit. Therefore we have 2 possible cases for the original message

$x2 = (0, 1, 0, 1)$ or $x2 = (1, 1, 0, 1)$

Therefore we cannot know for sure the true value for the first bit.

vii. Adjust your pseudocode to decode a noisy signal.

H = matrix[]

y = vector[]

```
E = matrix[] #matrix of the basis vector sin F^n
#loop throught all the basis vectors to find the possible vectors y1 given y2 and ej
possible_y1 = []
for i in e:
    y1 = y2 + e[i] # where e[i] represents the basis vector
   Append y1 to the list possible_y1
rrefs = []
#check what of the cases are plausible
for i in range(len(H)):
    H2 = augment H with each of the vectors from the list of possible y1
    rref = H2.rref()
    add given rref to list of all refs

print(rrefs)
```

Part 2:

2. Animation Studios(#Transformations, #ComputationalTools)

a) Why linear transformations cannot perform translations

Linear transformations are those that preserve the line grid as linear, parallel, and evenly spaced, while also keeping the origin fixed at its original point. Linear transformations can not perform translations because translations do not preserve the origin fixed at its original point.

Translations cannot be preserved by linear transformations because they do not preserve the zero vector. In linear transformations, T(0) must always be equal to 0. However, if we consider a translation T(v) = A*v + b, then T(0) would be equal to A*0 + b = b. This means that the zero vector is not preserved and becomes b, which violates the requirement of a linear transformation to keep the zero vector unchanged.

b) Find the matrix A and the vector b that translates a vector one unit to the right. Test your affine transformation on the vector v = ⟨2, −1⟩.

To translate a vector one unit to the right, we add 1 to the x-coordinate of the vector. Therefore, the transformation can be represented by the following system of equations:

$$x' = x + 1$$

$$y' = y$$

where (x,y) are the coordinates of the original vector, and (x',y') are the coordinates of the translated vector.

We can represent this system of equations as a matrix-vector equation:

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \end{vmatrix} = \begin{vmatrix} x + 1 \\ y \end{vmatrix}$$

Thus, the matrix A is:

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

And the vector b is:

[1,0]

Look at the figures below to see this translation on Sage:

```
#Question 2.b

A = matrix.identity(2) #Identity Matrix

v = vector([2,-1])

b = vector([1,0])

print('A*v+b: ', A*v+b) #Result: (3,-1)

a = arrow((0,0), (2,-1), color="blue") #Vector before transformation: v
z = arrow((0,0), (3,-1), color="purple") #A*v+b: (3,-1) Vector transformed one unit to the right.

show(a+z, aspect_ratio=2)
```

Figure 10: *Code to create the matrix A, the vector v and the vector b. It is also calculating the translation by multiplying A*v+b and then plotting using arrows.*

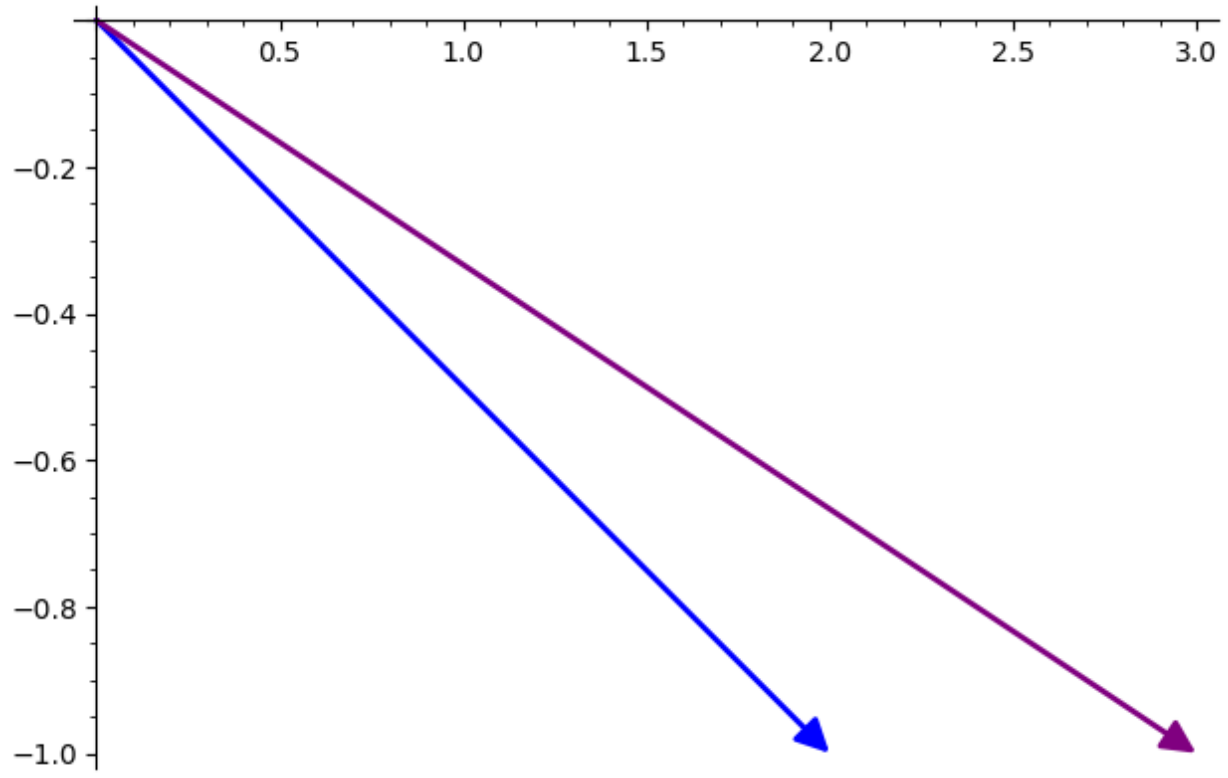


Figure 11: *Visualization of the vector v (blue), the vector v translated (purple).*

In conclusion, when we multiply A * v and A is an identity matrix, the v does not change.

So when we add the vector b -> v + b, we add one value to the x in v -> v = [x+1,y+0], moving one unit to the right. v translated = [2+1,-1].

c) **We may combine transformations, for example, by rotating a vector by 45 degrees counterclockwise, and then translating it to the right by one unit as a single affine transformation. What should A and b be to perform these two transformations? Again, test it with v = ⟨2, −1⟩.**

We already know that the vector b = [1,0] translates v one unit to the right. So we do not need to change anything on vector b because we already control the x-axis and y-axis, which is already giving us the transformation we want.

To rotate 45 degrees, we need to find a matrix that multiplies v and rotates the plan. Then we just add b to translate one unit to the right.

Thus the matrix A is:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos 45 & -\sin 45 \\ \sin 45 & \cos 45 \end{bmatrix} = \begin{bmatrix} 0.5253 & -0.8509 \\ 0.8509 & 0.5253 \end{bmatrix}$$

And the vector b is still:

[1,0]

For the calculations in Sage:

```
A:
 [ 1/2*sqrt(2) -1/2*sqrt(2)]
[ 1/2*sqrt(2)  1/2*sqrt(2)]
```

```
#Question 2.c

A = matrix([[1/sqrt(2), -1/sqrt(2)],[1/sqrt(2), 1/sqrt(2)]])

v = vector([2,-1])

b = vector([1,0])

print('A*v+b: ', A*v+b) #Result: (3/2*sqrt(2) + 1, 1/2*sqrt(2))

a = arrow((0,0),(2,-1),color="blue")
y=arrow((0,0),(3/2*sqrt(2)+1, 1/2*sqrt(2)),color='purple') #Rotating 45 degrees and moving 1 unit to the right.

show(a+y, aspect_ratio=1)
```

```
A*v+b:  (3/2*sqrt(2) + 1, 1/2*sqrt(2))
```

Figure 12: *Code to create the matrix A, the vector v and the vector b. It is also calculating the translation by multiplying A*v+b and then plotting using arrows.*

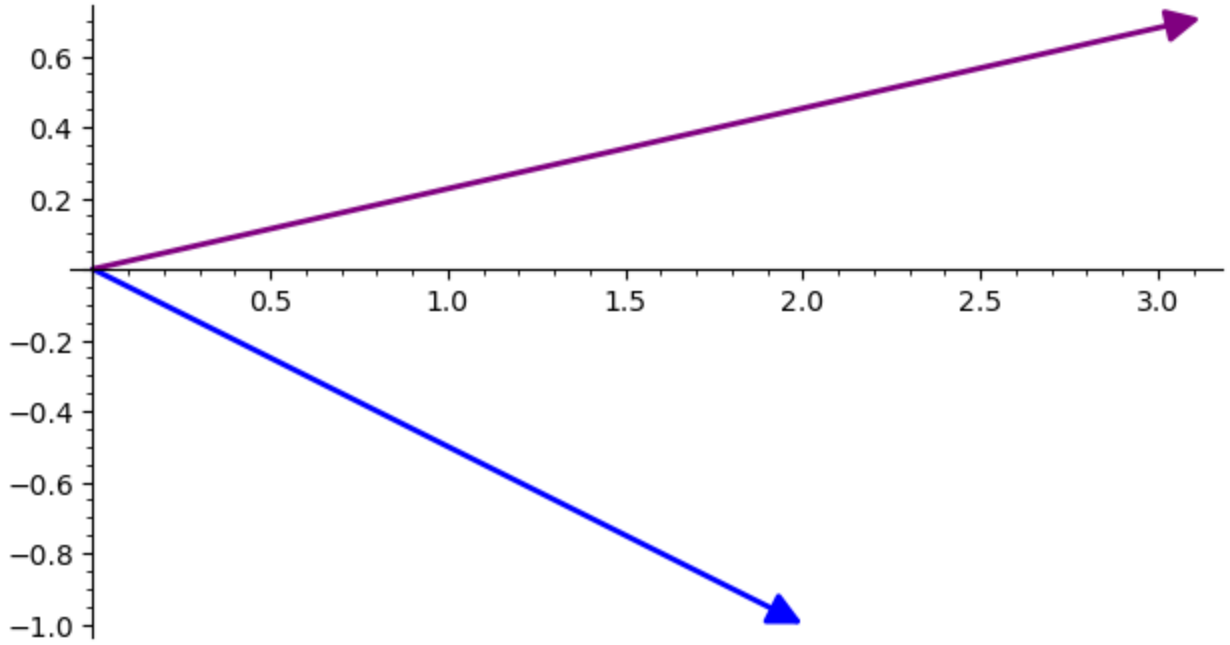


Figure 13: *Visualization of the vector v (blue), the vector v translated (rotated 45 degrees counterclockwise) (purple).*

d) Order matters! What if we had instead translated the vector v one unit to the right first, and then rotated it by 45 degrees counterclockwise? What would be the matrix A and vector b that perform these operations, written in the standard affine form A*v +b?

The matrix A and the vector b would still be the same, but, the v+b would be calculated first. So, instead of A*v+b. We would have A*(v+b), which performs the translation first (moving the vector one unit to the right) and then multiplies by A (rotating 45 degrees).

```
#Question 2.d

A = matrix([[1/sqrt(2), -1/sqrt(2)],[1/sqrt(2), 1/sqrt(2)]])

v = vector([2,-1])

b = vector([1,0])

print('A*v+b: ', A*v+b) #Result: (3/2*sqrt(2) + 1, 1/2*sqrt(2))
print('A*(v+b): ', A*(v+b)) #Result: (2*sqrt(2), sqrt(2))

a = arrow((0,0),(2,-1),color="blue")
y=arrow((0,0),(3/2*sqrt(2)+1, 1/2*sqrt(2)),color='purple') #Rotating 45 degrees and moving 1 unit to the right.
s=arrow((0,0),(2*sqrt(2), sqrt(2)),color='red') #Moving 1 unit to the right and Rotating 45 degrees.

show(a+y+s, aspect_ratio=1)
```

```
A*v+b:  (3/2*sqrt(2) + 1, 1/2*sqrt(2))
A*(v+b):  (2*sqrt(2), sqrt(2))
```

Figure 14: *Code to create the matrix A, the vector v and the vector b. It is also calculating the translation by multiplying A*(v+b) and then plotting using arrows.*

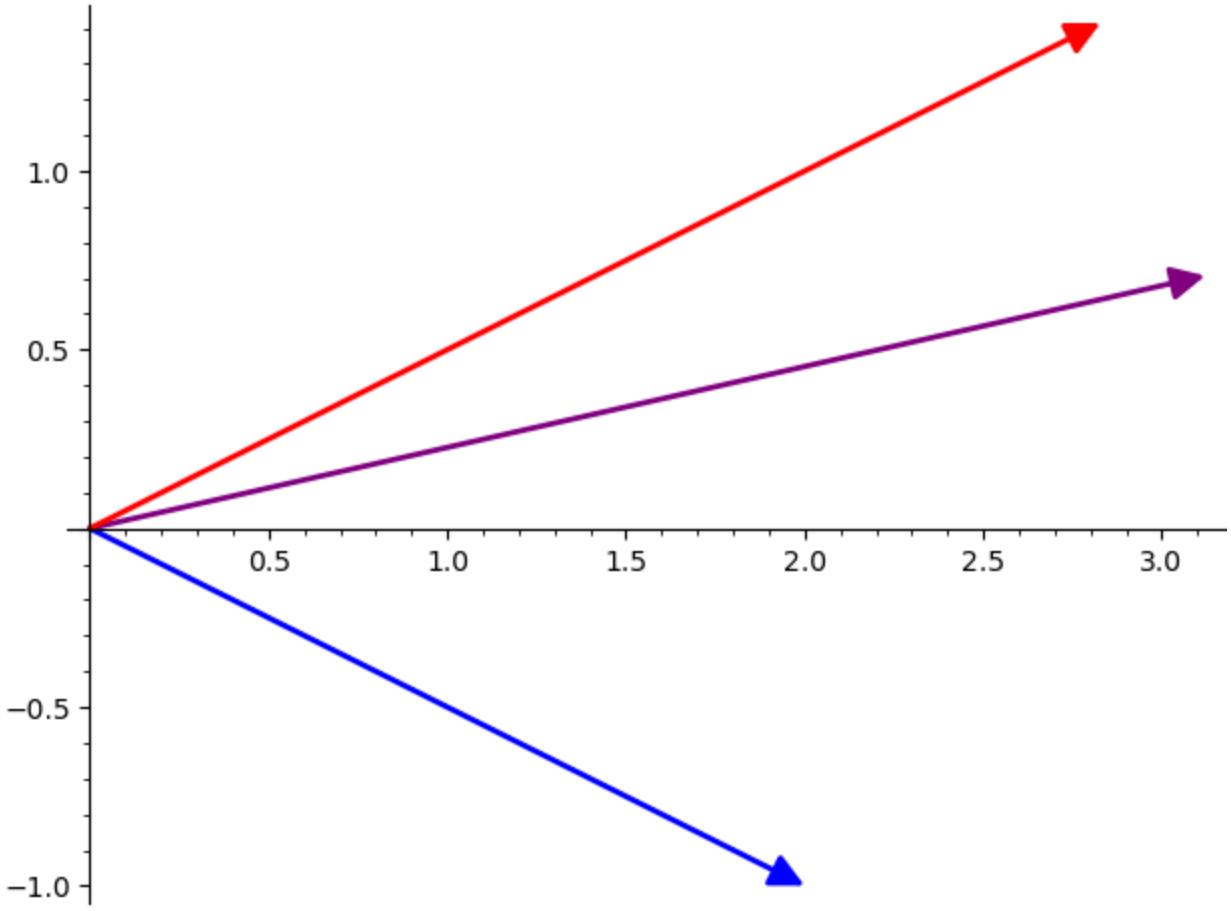


Figure 15: *Visualization of the vector v (blue), the vector v translated (rotated 45 degrees counterclockwise) (purple), the vector v translated following the different order (A*(v+b)) (rotated 45 degrees counterclockwise) (red).*

As we can see by looking at the green and red arrows, the final vector is different when we change the order.

e) Compare your matrices A and b, and your final transformed vector A*v +b from parts (c) and (d).

When A is a rotation matrix and we compare the two combined transformations, we notice that they both involve multiplying the input vector by A, but they differ in the addition of a vector b. The first transformation requires us to add b, which is responsible for the translation,

as we need to rotate first and then translate. On the other hand, in the second transformation, we add a new vector x = A*b, indicating that we should translate first and then apply the rotation. This disparity arises due to the compound function, where f(g(x)) is not equal to g(f(x)).

f) Finally, we can "cheat" to encode everything as a single matrix multiplication. We construct a combined vector V given as:

$$V = \begin{bmatrix} v \\ 1 \end{bmatrix}$$

and a combined matrix M:

$$M = \begin{bmatrix} A & \vec{b} \\ 0 \cdots 0 & 1 \end{bmatrix}$$

Describe the result of multiplying the matrix M by the vector V.

When performing the calculations, the difference is that the result will be a vector in which the upper value is the same vector obtained in exercises b and c.

When using Sage, when multiplying M*V we get a vector with 3 dimensions, in which x and y (the first two dimensions) form the same vectors obtained in exercises b and c.

g) Construct the combined vector V and matrix M that perform each of the affine transformations in parts (b) - (e) and verify your results.

Exercise b:

```
#2.g.b
V = vector([2, -1, 1])
M = matrix([[1, 0, 1],[0, 1, 0],[0, 0, 1]])

print('V: ', V)
print('M:\n',M)

M*V #Result: (3, -1, 1) / Result in b: (3, -1)
```

```
V:  (2, -1, 1)
M:
 [1 0 1]
[0 1 0]
[0 0 1]

(3, -1, 1)
```

Figure 16: *Code creating the vector V, and the matrix M. Then calculating M*V. Then comparing the result of the calculation with the results obtained before during exercises, showing that the first elements in the vector resultant form the same vector obtained in the exercise.*

Exercise c:

```
#2.g.c
V = vector([2,-1,1])
M = matrix([[1/sqrt(2), -1/sqrt(2),1],[1/sqrt(2), 1/sqrt(2),0],[0,0,1]])

print('V: ', V)
print('M:\n',M)

M*V #Result: (3/2*sqrt(2) + 1, 1/2*sqrt(2), 1) / Result in c: (3/2*sqrt(2) + 1, 1/2*sqrt(2))
```

```
V:  (2, -1, 1)
M:
 [ 1/2*sqrt(2) -1/2*sqrt(2)           1]
[ 1/2*sqrt(2)  1/2*sqrt(2)           0]
[           0           0           1]

(3/2*sqrt(2) + 1, 1/2*sqrt(2), 1)
```

Figure 17: *Code creating the vector V, and the matrix M. Then calculating M*V. Then comparing the result of the calculation with the results obtained before during exercises, showing that the first elements in the vector resultant form the same vector obtained in the exercise.*

Exercise d:

$$M = \begin{bmatrix} R & R(\langle 1,0 \rangle) \\ 0 \cdots 0 & 1 \end{bmatrix}$$

$$V = \begin{bmatrix} \langle 2,-1 \rangle \\ 1 \end{bmatrix}$$

$$MV = \begin{bmatrix} R(\langle 2,-1 \rangle) + R(\langle 1,0 \rangle) \\ 1 \end{bmatrix} = \begin{bmatrix} 3\cos(45) + \sin(45), -\cos(45) + 3\sin(45) \\ 1 \end{bmatrix}$$

(3cos(45) + sin(45), -cos(45) + 3sin(45)) = A*(v+b) = (2*sqrt(2), sqrt(2))

h) Create a figure using vectors (like the "house" we drew in class). Apply at least three affine transformations to your figure. Report your findings.

```
def transform_house(A,b):
    vb1 = vector([0, 0])
    vb2 = vector([2, 0])
    vb3 = vector([0, 0])
    vb4 = vector([0, 2])
    vb5 = vector([0, 2])
    vb6 = vector([2, 2])

    v1 = vector([0, 2])
    v2 = vector([2, 2])
    v3 = vector([2, 0])
    v4 = vector([2, 2])
    v5 = vector([1, 3])
    v6 = vector([1, 3])

    a1 = arrow(A*vb1+b,A*v1+b,color="blue")
    a2 = arrow(A*vb2+b,A*v2+b,color="blue")
    a3 = arrow(A*vb3+b,A*v3+b,color="blue")
    a4 = arrow(A*vb4+b,A*v4+b,color="blue")
    a5 = arrow(A*vb5+b,A*v5+b,color="blue")
    a6 = arrow(A*vb6+b,A*v6+b,color="blue")

    show(a1+a2+a3+a4+a5+a6, aspect_ratio=1, xmax=6, xmin=-6, ymax=6, ymin=-6)
```

Figure 18: *Function that receives as an input a matrix and a vector to perform the translation and the rotation in a figure of a home constructed before. Then plot the figure in a 2d plan.*

```
#Original Position (No transformation)
A = matrix.identity(2)
b = vector([0,0])

transform_house(A,b)
```

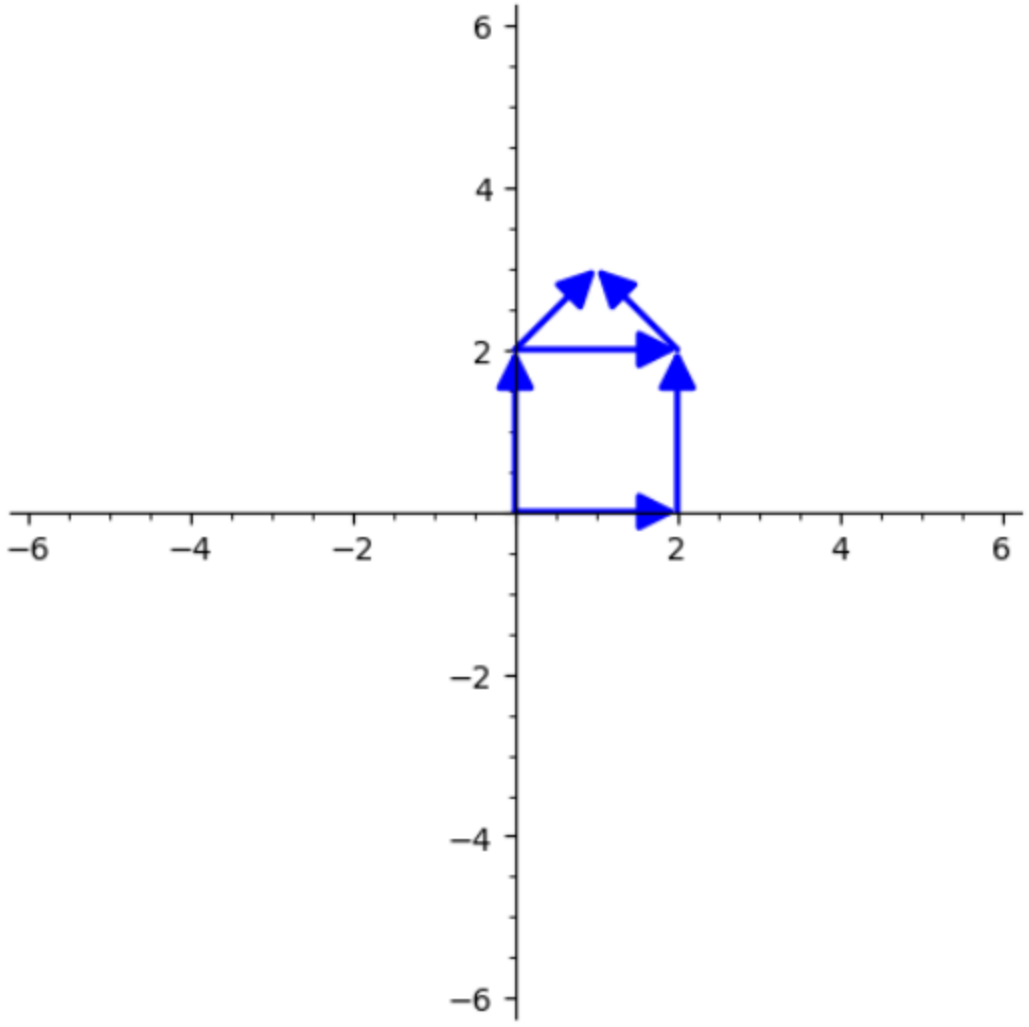


Figure 19: *This is the home in its original position (how it was constructed). The Inputs are created on the top (A as a matrix and b as a vector).*

```
A = matrix.identity(2) #Identity Matrix -> No Rotation
b = vector([1,0]) #Translating one unit to the right

transform_house(A,b)
```

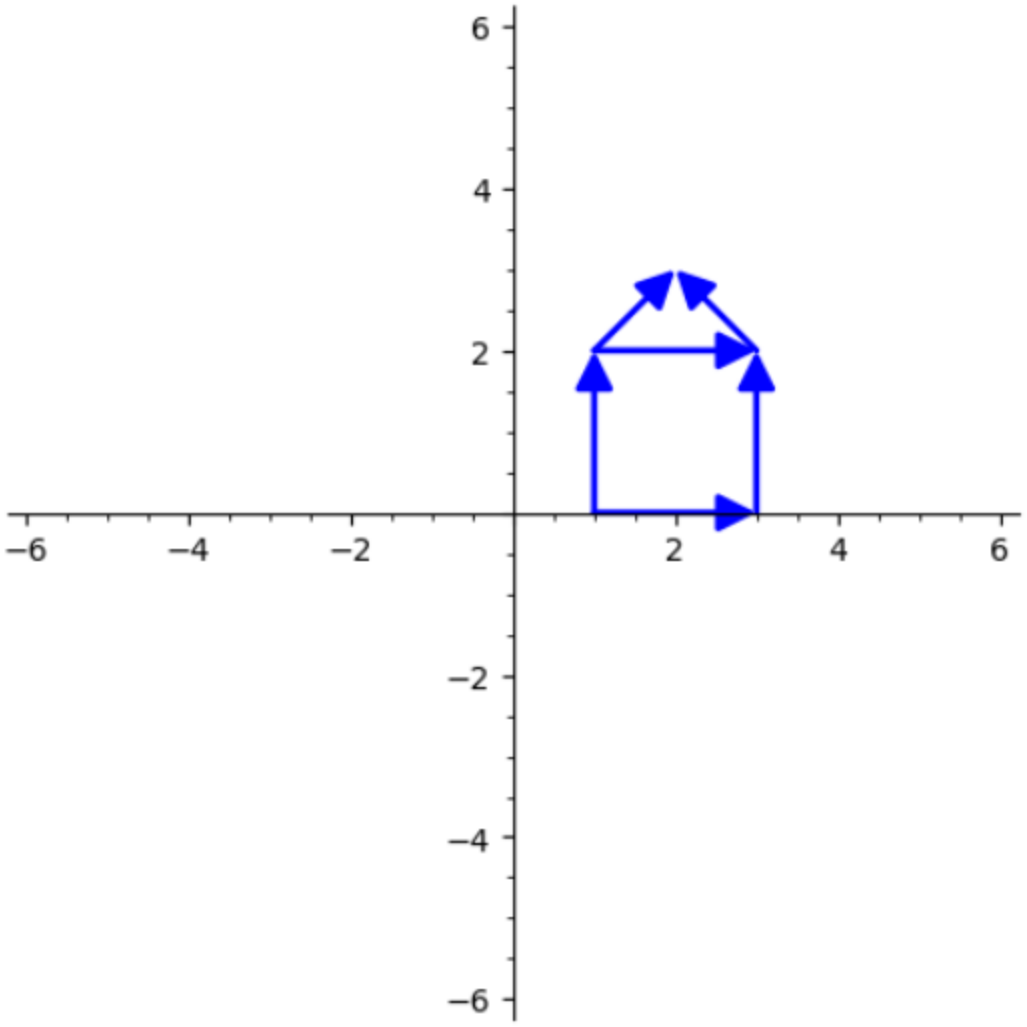


Figure 20: *This is the home being translated one unit to the right. The Inputs are created on the top (A as a matrix and b as a vector).*

```
A = A = matrix([[1/sqrt(2), -1/sqrt(2)],[1/sqrt(2), 1/sqrt(2)]]) #Rotating 45 degrees
b = vector([1,0]) #Translating one unit to the right

transform_house(A,b)
```

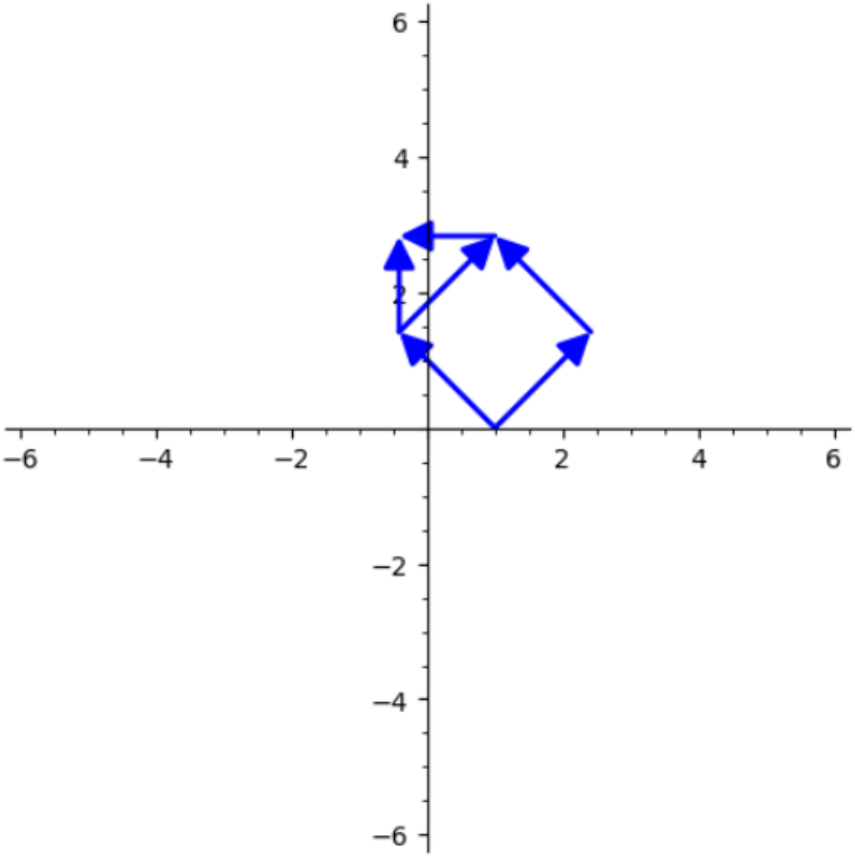


Figure 21: *This is the home being rotated 45 degrees counterclockwise and then translated one unit to the right. The Inputs are created on the top (A as a matrix and b as a vector).*

```
A = matrix([[1, 0],[0, -1]]) #Iverting x-axis
b = vector([1,0]) #Translating one unit to the right

transform_house(A,b)
```

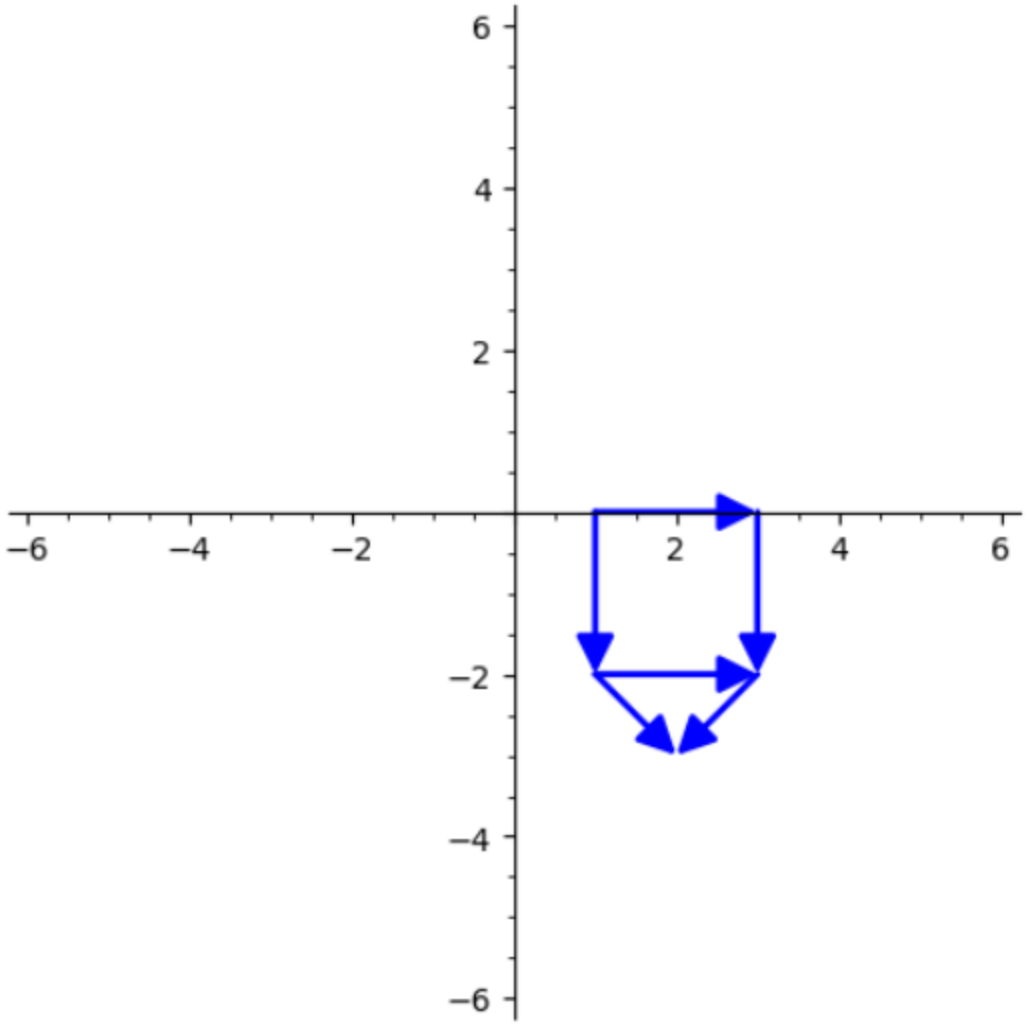


Figure 22: *This is the home being inverted in the base x-axis and y-axis negative, and then translated one unit to the right. The Inputs are created on the top (A as a matrix and b as a vector).*

```
A = matrix([[-1, 0],[0, -1]]) #Iverting x-axis and y-axis
b = vector([-1,-1]) #Translating one unit to the left and one unite down

transform_house(A,b)
```

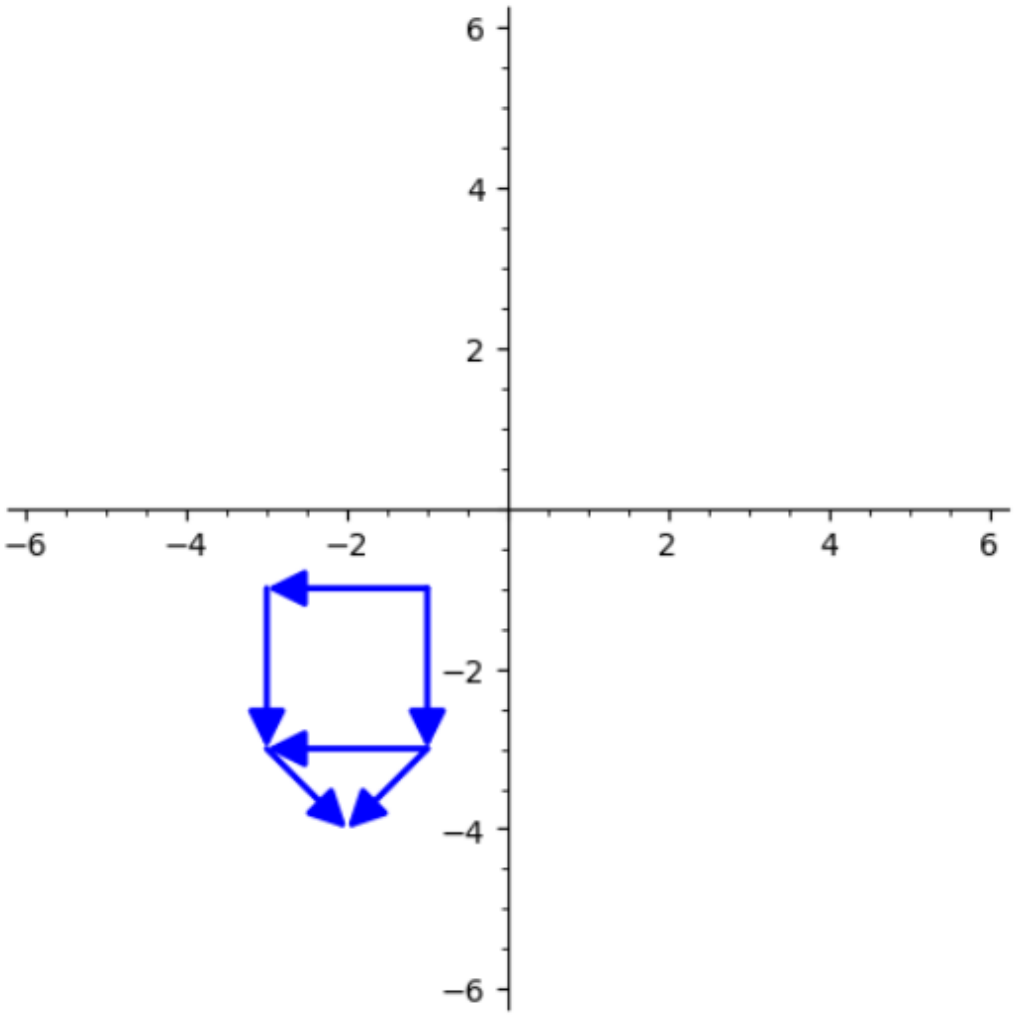
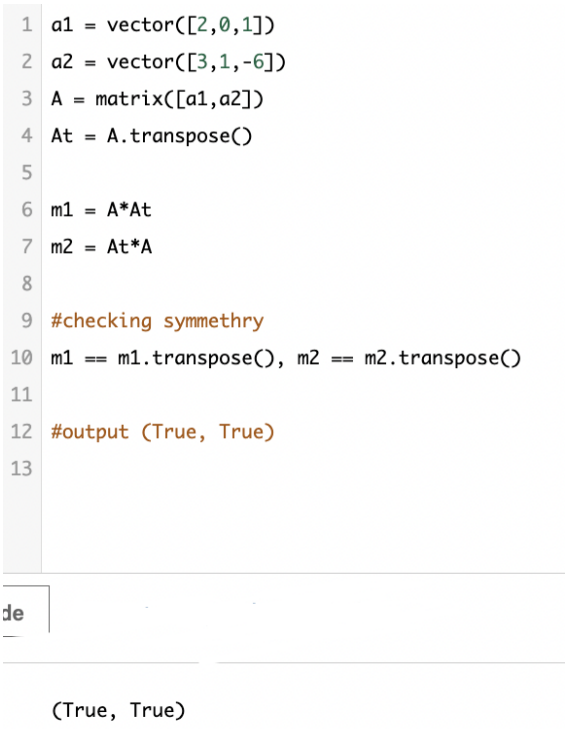


Figure 23: *This is the home being inverted in which x-axis and y-axis negative, and then translated one unit to the left, and one unit down. The Inputs are created on the top (A as a matrix and b as a vector).*

3. A valuable factor #Transformations, #TheoreticalTools

a) i. Show that AAT and AT A are both symmetric matrices.

Symmetric matrix is the matrix that is equal to its transpose. They have mirrored values on both sides of the main diagonal.



```
a1 = vector([2,0,1])
a2 = vector([3,1,-6])
A = matrix([a1,a2])
At = A.transpose()

m1 = A*At
m2 = At*A

#checking symmethry
m1 == m1.transpose(), m2 == m2.transpose()

#output (True, True)
```

de

(True, True)

Figure 24: *figure to prove that the matrices m1 and m1 are equal to their transposes.*

As we can see both matrices are equal to their transposes, therefore they are symmetrical.

ii. Let M be an m × n matrix. Show that MTM and MMT are both symmetric matrices

To show that a matrix is symmetric, we need to show that it is equal to its transpose.

Therefore we will consider 2 cases for each M^TM and MM^T

1. To show that M^TM is symmetric, we need to show that:

$$(M^TM)^T = M^TM$$

Since the transpose of a product of matrices is the product of the transposes in reverse order, we have:

$$(M^TM)^T = M\ (M^T)^T$$

Since $(M^T)^T = M$,

$(M^T M)^T = M^T M$

Therefore, we have shown that $M^T M$ is equal to its transpose, and is therefore a symmetric matrix.

2. For MM^T we basically repeat the same steps as we did above

$MM^T = (MM^T)^T$

Since the transpose of a product of matrices is the product of the transposes in reverse order, we have:

$MM^T = (M^T)^T M^T$

Since $(M^T)^T = M$,

$MM^T = MM^T$

Therefore MM^T is a symmetric matrix because it is equal to its transpose.

iii. By the spectral theorem we can find orthogonal matrices U and V such that $AA^T = UD_1U^T$ and $AA^T = VD_2V^T$. Find U, V, D_1 and D_2. How are D_1 and D_2 related?

We created 2 matrices and checked their symmetry.

$m1 = AA^T$

$m2 = A^TA$

We know that

Columns of U are eigenvectors of AA^T

Columns of V are eigenvectors of A^TA

Using this information we got the following:

```
Check symmethry by comparing matrix to its transpose True True
U
[  1   1   1]
[1/3   0 -15]
[ -2 1/2  -2]
Ut
[  1 1/3  -2]
[  1   0 1/2]
[  1 -15  -2]
V
[0 1]
[1 0]
Vt
[0 1]
[1 0]
```

Figure 25: the figure shows the output of the code that generated the matrices, U, U^T, V, V^T. The code can be found in Appendix A

Based on the spectral theorem we know that matrices D_1 and D_2 should have the eigenvalues of matrices m1 and m2 on their diagonals.

```
print('eigenstuff for m1',m1.right_eigenvectors())
print('-----')
print('eigenstuff for m2',m2.right_eigenvectors())
```

Run Code

```
Out [14]    eigenstuff for m1 [(46, [
            (0, 1)
            ], 1), (5, [
            (1, 0)
            ], 1)]
            -----
            eigenstuff for m2 [(46, [
            (1, 1/3, -2)
            ], 1), (5, [
            (1, 0, 1/2)
            ], 1), (0, [
            (1, -15, -2)
            ], 1)]
```

Figure 26: *Eigenstuff for m1 and m2*

We know the eigenvalues, so we can construct the matrices:

D1 =

[46, 0]

[0, 5]

D2 =

[46, 0, 0]

[0, 5, 0]

[0, 0, 0]

By comparing D1 and D2 we can see that they have the same non zero eigenvalues.

Now let's plug in the matrices in the original equations $AA^T = UD_1U^T$ and $AA^T = VD_2V^T$ and check if they hold.

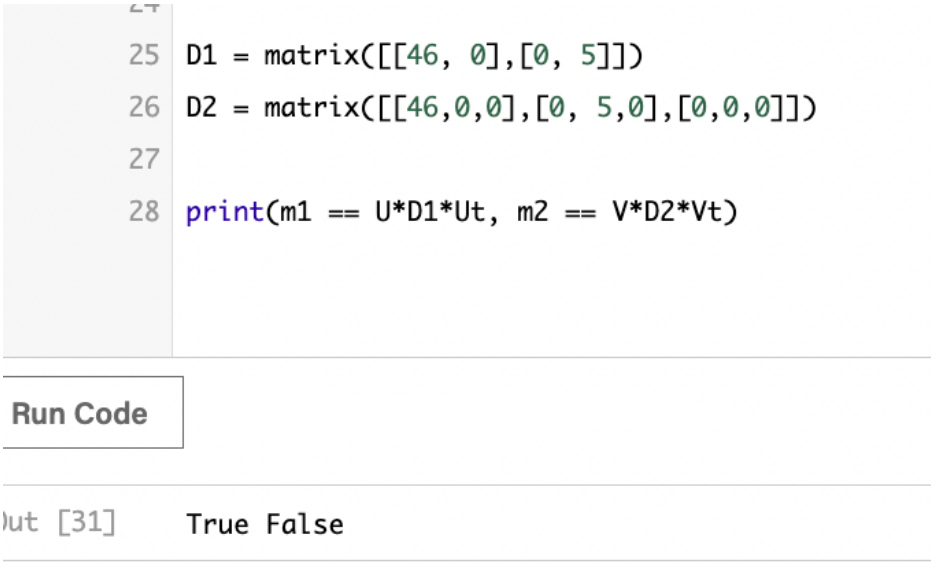


Figure 27: *figure represents checking the original statement.*

$A^TA = VD_2V^T$ is false because the columns of V do not necessarily diagonalize A^TA, while $AA^T = UD_1U^T$ is true because the columns of U do diagonalize AA^T.

iv. (Optional) Prove that M^TM and MM^T have the same non-zero eigenvalues, and these eigenvalues are positive.

To prove that M^TM and MM^T have the same non-zero eigenvalues, we need to show that if λ is a non-zero eigenvalue of M^TM, then it is also an eigenvalue of MM^T, and vice versa.

Let v be a non-zero eigenvector of M^TM with corresponding eigenvalue λ.

$M^TM\,V = \lambda v$

Multiplying both sides of this equation by M^T, we get:

$M^TM\,(M^Tv) = \lambda(M^Tv)$

This shows that M^Tv is an eigenvector of M^TM with eigenvalue λ. Let's do the same for MM^T.

Let w be a non-zero eigenvector of MM^T with corresponding eigenvalue μ. Then we have:

$MM^T w = \mu w$

Multiplying both sides of this equation by M, we get:

$M^T(MM^T w) = \mu(M^T w)$

This shows that $M^T w$ is an eigenvector of MM^T with an eigenvalue μ.

Thus, we have shown that the non-zero eigenvalues of $M^T M$ and MM^T are the same. Specifically, the eigenvalues are the squared singular values of M, which are non-negative by definition.

v. Try factoring with U and V instead. Compute $\Sigma = U^T AV$. What form does Σ take? How is Σ related to D1 and D2 above?

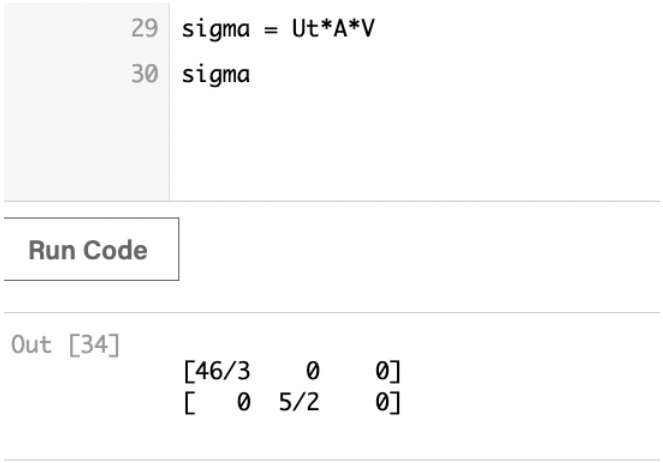


Figure 28: *This represents the sigma matrix*

Sigma (Σ) is a 2x3 matrix obtained from the SVD of matrix A. We can see that the values on the diagonal of Σ changed, but they are still related to the singular values we had for D1 and D2. The singular values of a matrix A are the square roots of the eigenvalues of $A^T A$.

In the case of $U^T AV$, we have:

$$det(U^T AV - \lambda I) = det(A^T A - \lambda I) = det(V(D^2 - \lambda I)V\text{^}^T) = det(D^2 - \lambda I)$$

where D is the diagonal matrix of singular values of A and V is the orthogonal matrix of eigenvectors of $A^T A$.

Therefore, the eigenvalues of $A^T A$ are the entries on the diagonal of D^2. The singular values of A are 46 and 5, so the eigenvalues of $A^T A$ are 46^2 and 5^2. When we write $\Sigma = U^T AV$, we are taking the square root of these eigenvalues and placing them on the diagonal of Σ. That is why we have the fractions 46/3 and 5/2 instead of the integers 46 and 5. These fractions are the square roots of the eigenvalues of $A^T A$, and they are positive real numbers that satisfy the equation $\Sigma = U^T AV$.

(b) Find singular value decompositions

i.B = [1 −4 2]

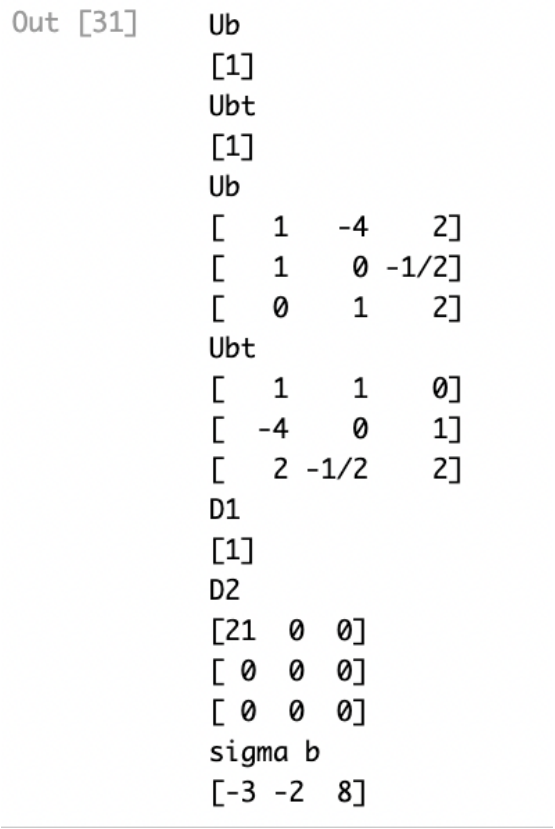


Figure 29: *This represents the sigma matrix for matrix b*

Code in appendix A

ii.C = [0 2]

[−1 0]

```
Out [39]    Uc
            [1 0]
            [0 1]
            Uct
            [1 0]
            [0 1]
            Vc
            [   1   -4    2]
            [   1    0 -1/2]
            [   0    1    2]
            Vbt
            [   1    1    0]
            [  -4    0    1]
            [   2 -1/2    2]
            D1
            [4 0]
            [0 1]
            D2
            [4 0]
            [0 1]
            sigma c
            [ 0  2]
            [-1  0]
```

Figure 30: *This This represents the sigma matrix for matrix c*

iii. D = [−4 1]

[−2 −4]

[0 −2]

[2 −2]

```
Out [44]    Ud
            [   1   -4   -2   -2]
            [   1  1/2    0 -1/2]
            [   1    0 -3/2    2]
            [   0    1   -3    1]
            Udt
            [   1    1    1    0]
            [  -4  1/2    0    1]
            [  -2    0 -3/2   -3]
            [  -2 -1/2    2    1]
            Vd
            [1 0]
            [0 1]
            Vdt
            [1 0]
            [0 1]
            D1
            [25  0  0  0]
            [ 0 24  0  0]
            [ 0  0  0  0]
            [ 0  0  0  0]
            D2
            [25  0]
            [ 0 24]
            sigma d
            [-6 -5]
            [17 -8]
            [ 2  7]
            [11 -6]
```

Figure 31: *This This represents the sigma matrix for matrix d*

Appendix A

Calculation of the singular value decomposition for matrix A

In [10]:

```
a1 = vector([2,0,1])
a2 = vector([3,1,-6])
A = matrix([a1,a2])
At = A.transpose()
m1 = A*At
m2 = At*A
#checking symmethry
print('Check symmethry by comparing matrix to its transpose', m1 == m1.transpose(), m2 == m2.transpose())
#output (True, True)
#---------------------- we find V, Vt
m1.right_eigenvectors()
v1 = vector([0,1])
v2 = vector([1,0])
V = matrix([v1,v2])
Vt = V.transpose()
#----------------------we find U, Ut
m2.right_eigenvectors()
u1 = vector([1, 1/3, -2])
u2 = vector([1, 0, 1/2])
u3 = vector([1, -15, -2])
Ut = matrix([u1,u2,u3])
U = Ut.transpose()

print('U')
print(U)
print('Ut')
print(Ut)
print('V')
print(V)
print('Vt')
print(Vt)
```

Calculation of the singular value decomposition for matrix B

In [31]:

```
B =  matrix([[1, -4, 2]])
Bt = B.transpose()
b1 = B*Bt
b2 = Bt*B
b1.right_eigenvectors()
Ub = matrix([[1]])
Ubt = Ub.transpose()
print('Ub')
print(Ub)
print('Ubt')
print(Ubt)
b2.right_eigenvectors()
Vb = matrix([[1, -4, 2],[1, 0, -1/2],[0, 1, 2]])
Vbt = Vb.transpose()
print('Ub')
print(Vb)
print('Ubt')
print(Vbt)
D1 = matrix([[1]])
D2 = matrix([[21,0,0],[0,0,0],[0,0,0]])
print('D1')
print(D1)
print('D2')
print(D2)
sigma_b = Ubt*B*Vb
print('sigma b')
print(sigma_b)
```

Calculation of the singular value decomposition for matrix C

In [39]:

```
C =  matrix([[0,2],[-1,0]])

Ct = C.transpose()
c1 = C*Ct
c2 = Ct*C
c1.right_eigenvectors()
Uc = matrix([[1,0],[0,1]])
Uct = Uc.transpose()
print('Uc')
print(Uc)
print('Uct')
print(Uct)

Vc = matrix([[1,0],[0,1]])
Vct = Vc.transpose()
print('Vc')
print(Vb)
print('Vbt')
print(Vbt)
D1 = matrix([[4, 0],[0,1]])
D2 = matrix([[4, 0],[0,1]])
print('D1')
print(D1)
print('D2')
print(D2)
sigma_c = Uct*C*Vc
print('sigma c')
print(sigma_c)

```

Calculation of the singular value decomposition for matrix D

In [44]:

```
D =  matrix([[-4,1],[-2,-4],[0,-2],[2,-2]])
Dt = D.transpose()
d1 = D*Dt
d2 = Dt*D
d1.right_eigenvectors()
Ud = matrix([[1, -4, -2, -2],[1, 1/2, 0, -1/2],[1, 0, -3/2, 2],[0, 1, -3, 1]])
Udt = Ud.transpose()
print('Ud')
print(Ud)
print('Udt')
print(Udt)
Vd = matrix([[1,0],[0,1]])
Vdt = Vd.transpose()
print('Vd')
print(Vd)
print('Vdt')
print(Vdt)
D1 = matrix([[25, 0,0,0],[0,24,0,0],[0,0,0,0],[0,0,0,0]])
D2 = matrix([[25, 0],[0,24]])
print('D1')
print(D1)
print('D2')
print(D2)
sigma_d = Udt*D*Vd
print('sigma d')
print(sigma_d)

```

Reflection:

#EmergentProperties:

The members of this group worked on all the questions together, as 2 different minds thinking about the same problems meant that they had a greater number of insights on the same question, which consequently made the answers more complete. The team was working together almost all the time by connecting on Zoom. It allowed constant communication and created an emergent property of efficiency at work. A clear example of this is question 2. a, in which each group member explained 1 of the reasons why linear transformations cannot perform translations. They discussed their thoughts and reasons and came up with an explanation that is much better than just an explanation one team member could give and multiplied by 2. If this dynamic of working together had not happened, the answer would only have the perspective of one of the members and would not be complete.

#Differences:

The team was allocated to the abilities of each group member, so that they could exploit their best abilities. One of the members has more affinity with code, so they had the responsibility of making the sage codes to have the figures in the assignment confirming results and showing graphs. The other member, on the other hand, is very good at explaining the theory, so they did the written parts. One team member is more comfortable working

#Strategize: We knew that we needed a strategic approach to achieve our goals. So, we started by diagnosing the problem at hand and identifying the key leverage points that we could exploit.

We analyzed the assignment requirements, determined our strengths and weaknesses, and came up with a plan to work based on each member's expertise. After developing a guiding policy to shape our approach, we started with coherent actions, such as researching relevant concepts, doing problem-solving, and discussing any difficulties or questions that arose during the process. We also collaborated on each step to ensure the accuracy and efficiency of the solutions, and constantly provided feedback to each other to improve our performance. By employing this strategic approach, we were able to complete the assignment successfully, enhance our learning outcomes, and develop our teamwork and communication skills. We had identified the leverage points and exploited our relative strengths to achieve our goals and overcome any challenges that we faced. We learned that a strategic approach is essential for effective collaboration and can significantly improve the effectiveness and efficiency of writing this team assignment.

AI tools:

We used chat gpt for checking for grammar and improving long explanations to make them more coherent and sound more native. Chat gpt was also used in question 2.a to explore all the reasons to prove that linear transformations cannot perform translations.