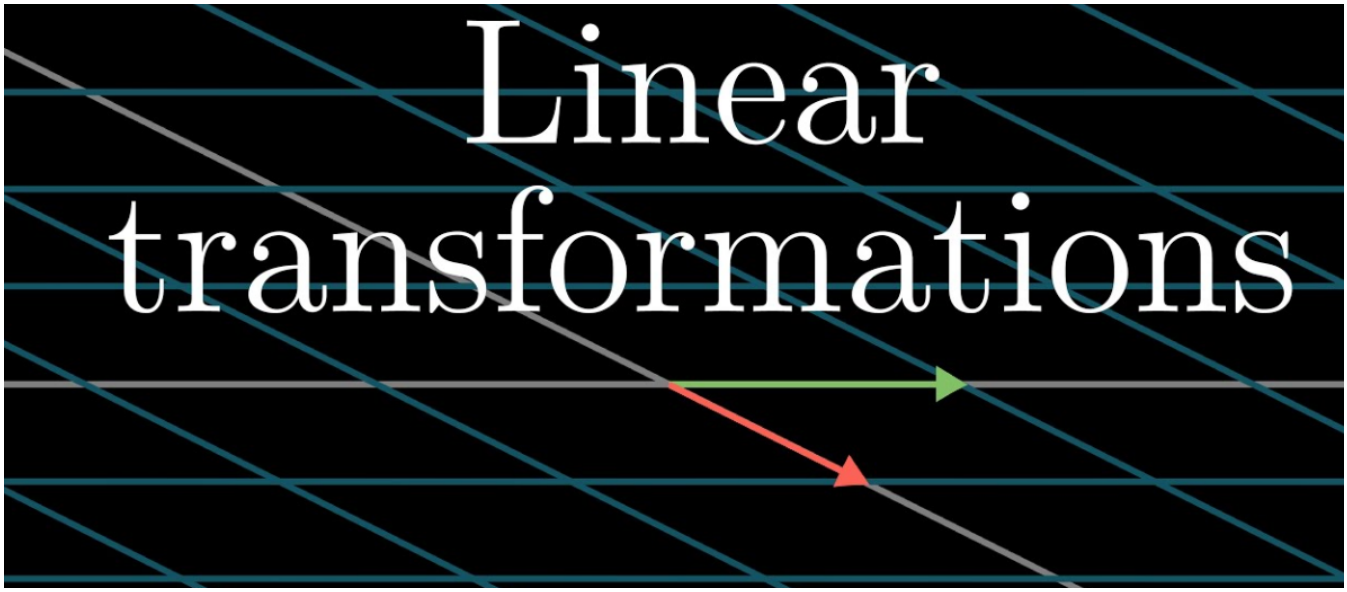


A linear transformation is a function that takes a vector in one vector space and maps it to a vector in another vector space. The transformation must preserve the following two properties:

- **Additivity:** If we add two vectors in the first vector space, the transformation must map the sum to the sum of the images of the two vectors in the second vector space.

Additivity: T(x + y) = T(x) + T(y)

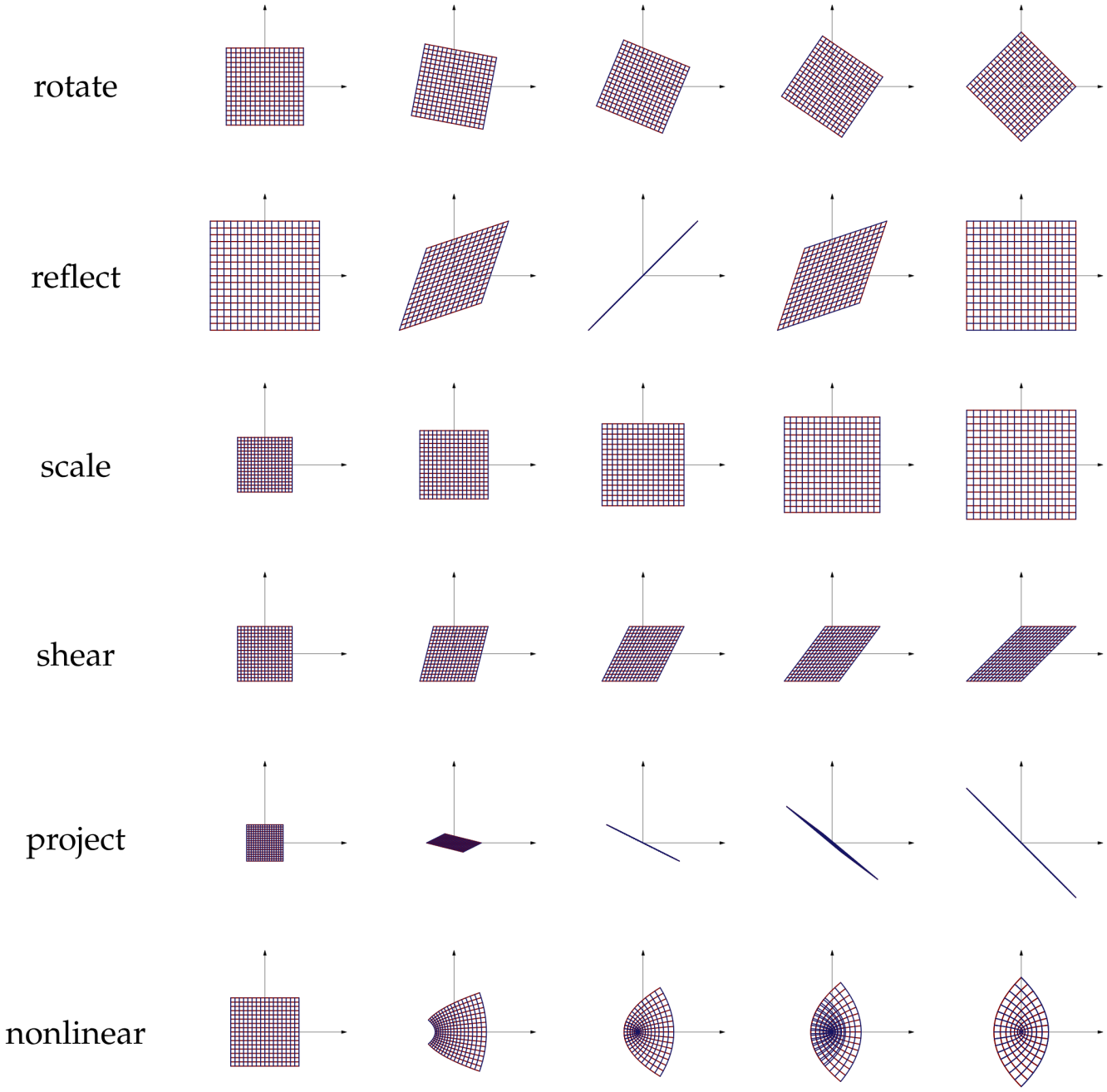
- **Scalar multiplication:** If we multiply a vector in the first vector space by a scalar, the transformation must map the product to the product of the image of the vector and the scalar.

Scalar multiplication: T(ax) = aT(x)

Intuitively, a linear transformation can be thought of as a way of distorting or transforming a geometric object. For example, a linear transformation could stretch an object, shrink it, rotate it, or reflect it. However, a linear transformation cannot change the overall shape of an object.

Here are some examples of linear transformations:

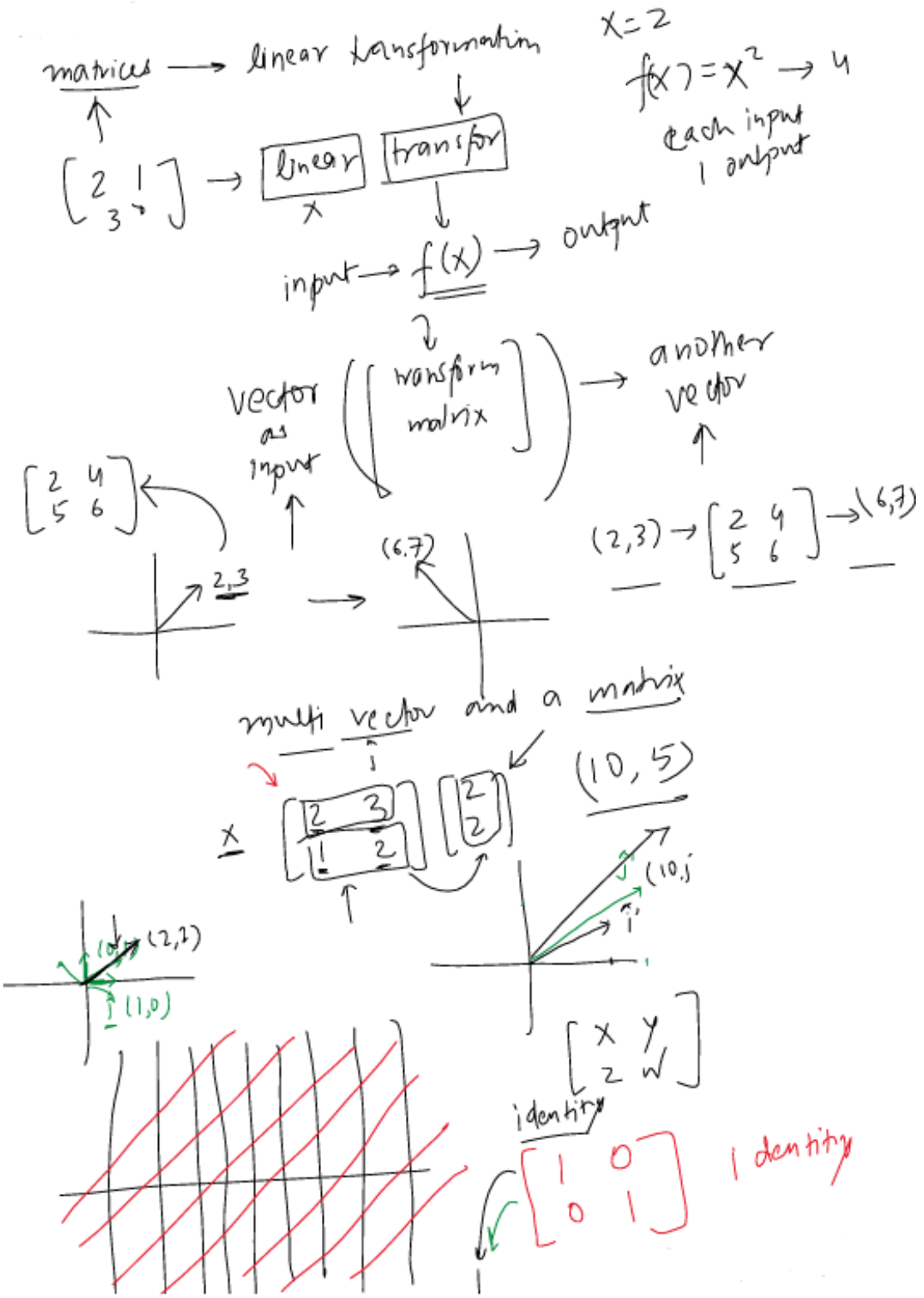
- **Scaling:** A scaling transformation multiplies all the coordinates of a vector by a constant factor. For example, a scaling transformation with a factor of 2 would double the size of an object.
- **Rotation:** A rotation transformation rotates an object by a certain angle. For example, a rotation transformation with an angle of 90 degrees would rotate an object 90 degrees clockwise.
- **Reflection:** A reflection transformation reflects an object across a line. For example, a reflection transformation across the x-axis would reflect an object across the x-axis.



Linear transformations are a powerful tool in mathematics and computer science. They are used in a wide variety of applications, including image processing, machine learning, and cryptography.

Here is some mathematical intuition for linear transformations:

- **Linear transformations are functions.** This means that they can be represented by formulas, just like any other function.
- **Linear transformations are linear.** This means that they satisfy the two properties of additivity and scalar multiplication.
- **Linear transformations can be represented by matrices.** This is a very powerful property, as it allows us to use matrices to perform linear transformations.



Why its Called Linear Transformation?

A linear transformation is called linear because it preserves linear combinations. A linear combination of vectors is a sum of scalar multiples of them.

- For example, the linear combination of the vectors $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$ is the vector $\begin{pmatrix}1\\1\end{pmatrix}$. If we apply a linear transformation to a linear combination of vectors, the result will also be a linear combination

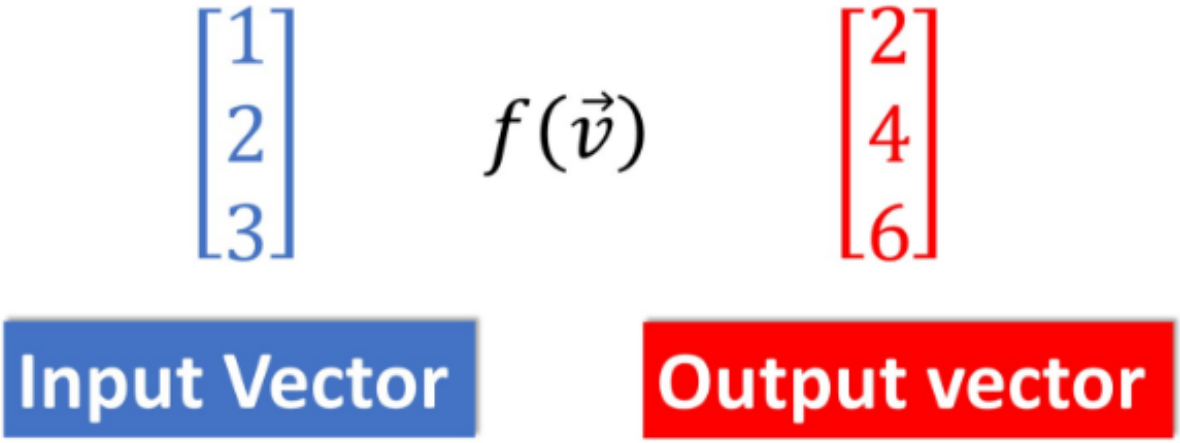
of the transformed vectors.

The term "linear" is also used in other contexts to mean "having to do with lines". This is because the prototypical example of a linear function is the equation y = mx, whose graph is a straight line. However, the term "linear transformation" is not referring to the shape of the graph of the transformation, but rather to the fact that it preserves linear combinations.

In summary, a linear transformation is called linear because **it preserves linear combinations. This is the defining property of linear transformations, and it is what makes them so useful in mathematics and computer science.**

Linear Transformation In detailed ?

a linear transformation is actually a function that maps an input vector into an output vector.



we forget on complex numbers and we observe again our 2D coordinate system. We recall that a single vector in a 2D plane is represented with a pair (x, y). If we map this vector to another one, we say that this is actually a transformation. Recall that sometimes we refer to a vector as a movement. Then, with a linear transformation we are moving that vector again in our plane to get the output vector. Therefore, vectors can be seen as a displacement vectors and by transforming them we are actually moving them in some particular way.

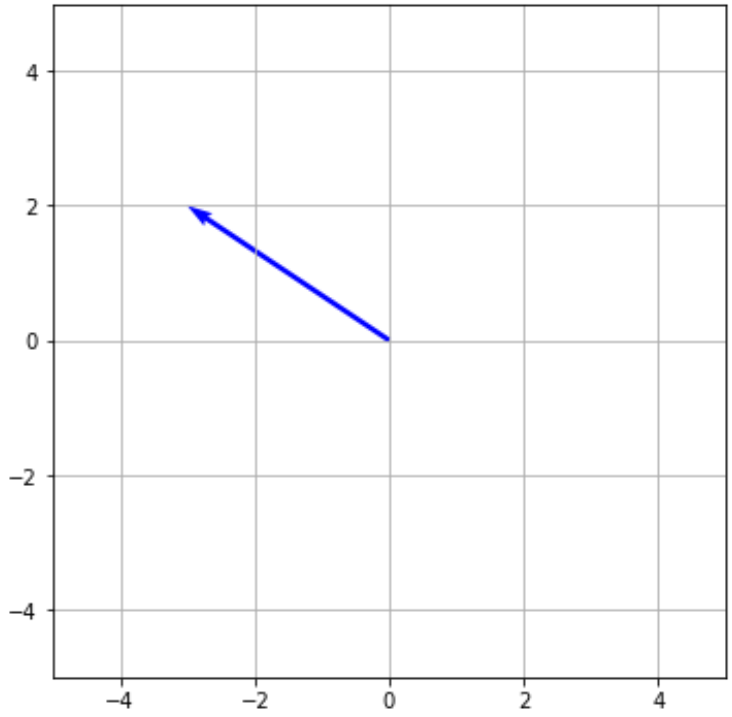
The word “transformation” suggests an association with the movement.

Example:

we will have an input vector that’s transformed to the output vector. To be more precise we will rotate our input vector for 90 degrees clockwise.

In [3]:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the vector
vec = np.array([[-3], [2]])

# Define the origin point
origin = np.zeros(vec.shape)

# Create the figure
plt.figure(figsize=(6, 6))

# Plot the vector
plt.quiver(*origin, *vec, color=['b'], scale=1, units='xy')

# Set grid
plt.grid()

# Set the x and y limits
plt.xlim(-5, 5)
plt.ylim(-5, 5)

# Set aspect ratio
plt.gca().set_aspect('equal')

# Display the plot
plt.show()
```



In [4]:

```
# Create a rotation matrix
theta = np.radians(-90)
R = np.zeros((2, 2))
R[0, 0] = np.cos(theta)
R[0, 1] = -np.sin(theta)
R[1, 0] = np.sin(theta)
R[1, 1] = np.cos(theta)

# Print the rotation matrix
print(R)
```

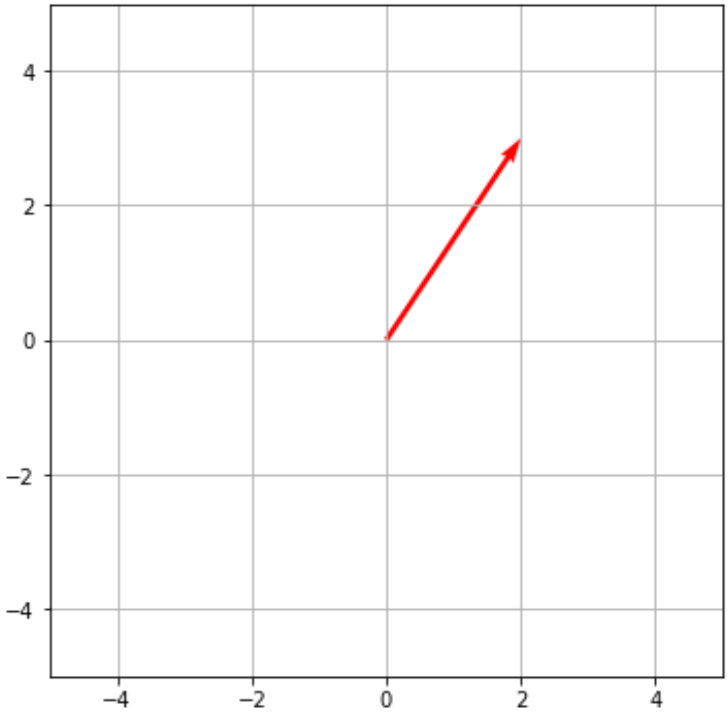
```
[[ 6.123234e-17  1.000000e+00]
 [-1.000000e+00  6.123234e-17]]
```

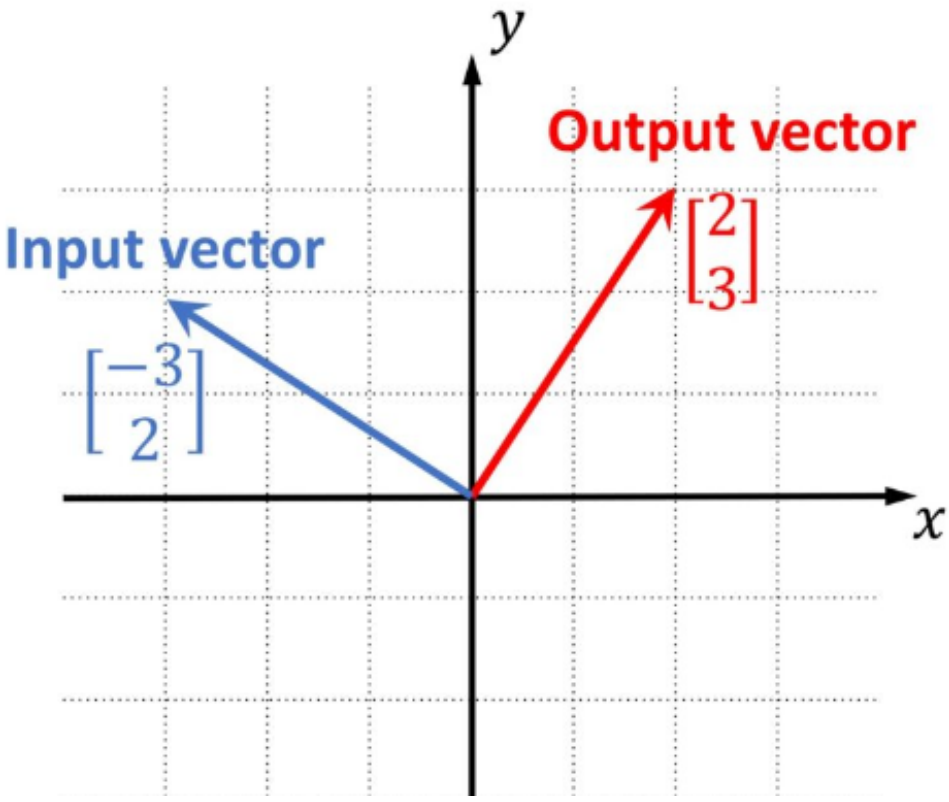
In [5]:

```
new = np.zeros(vec.shape)

new[0] = R[1,1]*vec[0] + R[0,1]*vec[1]
new[1] = R[1,0]*vec[0] + R[0,0]*vec[1]


# Numpy allows us to do vector-matrix multiplication much faster
new = R.dot(vec)
```

In [6]:

```
origin = np.zeros(new.shape) # origin point

plt.figure(figsize=(6,6))
plt.quiver(*origin, *new, color=['r'], scale=1, units='xy')

plt.grid()
plt.xlim(-5,5)
plt.ylim(-5,5)
plt.gca().set_aspect('equal')
plt.show()
```





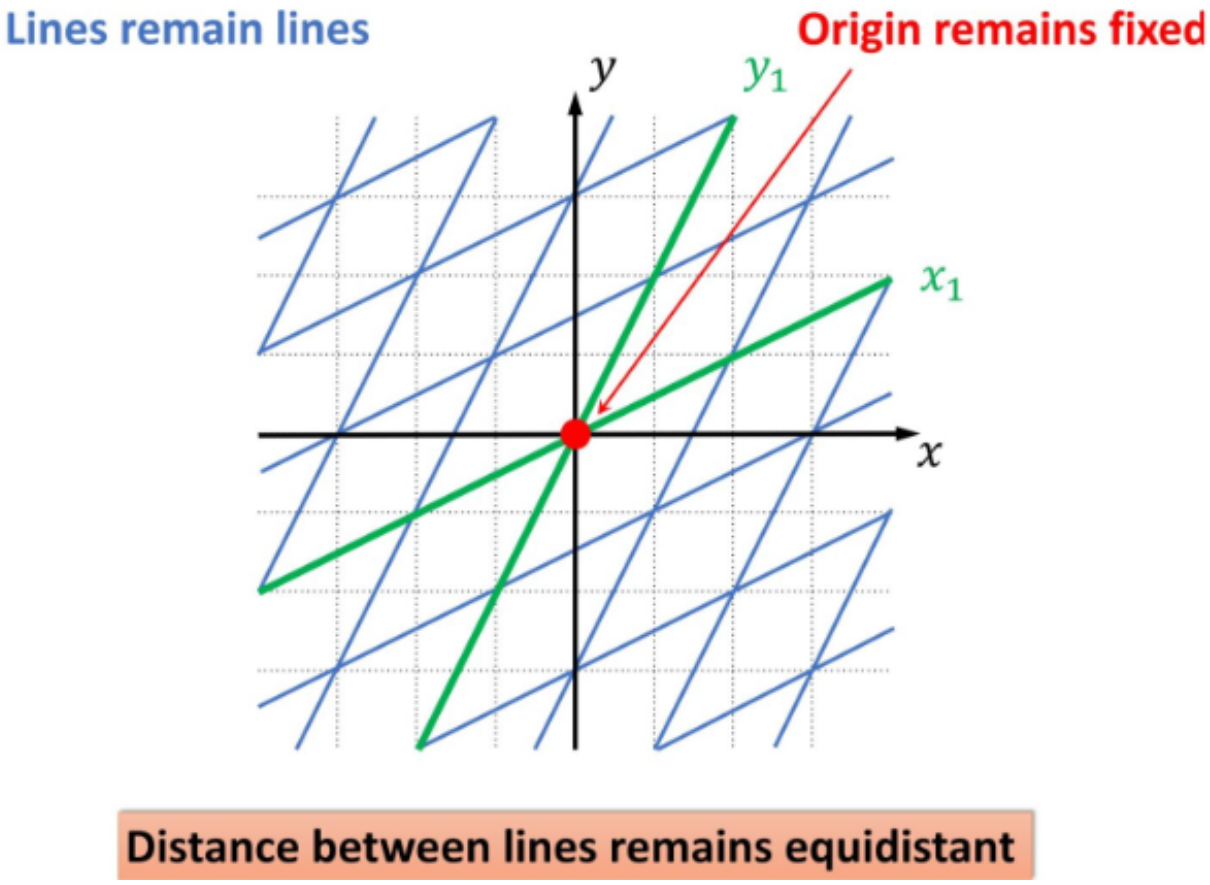
Moreover, this same transform can be applied not only to a single vector, but can be actually applied on the whole set of vectors. So, basically, let's say that we want to transform the whole plane and to see where majority of the vectors from that plane will be mapped. One way to visualize this is to represent vectors not as displacement arrows, but as points (positions). Then, we can map each of these points and observe where they will land after the transformation. This will give us an idea how our transformation actually looks like.

2. Linear transformation and basis vectors

There are three properties that a linear transformation has to obey so that we are allowed to call it a linear transformation.

- **First, a line should remain a line once we transform our coordinate system**.
- **Second, an origin should remain at the fixed place**
- **Third ,the distance between the grid lines should remain equidistant and parallel.**

If we apply something more sophisticated like a Neural network, then we should obtain a nonlinear transformation and actually the grid lines will be pretty much curved and deformed.

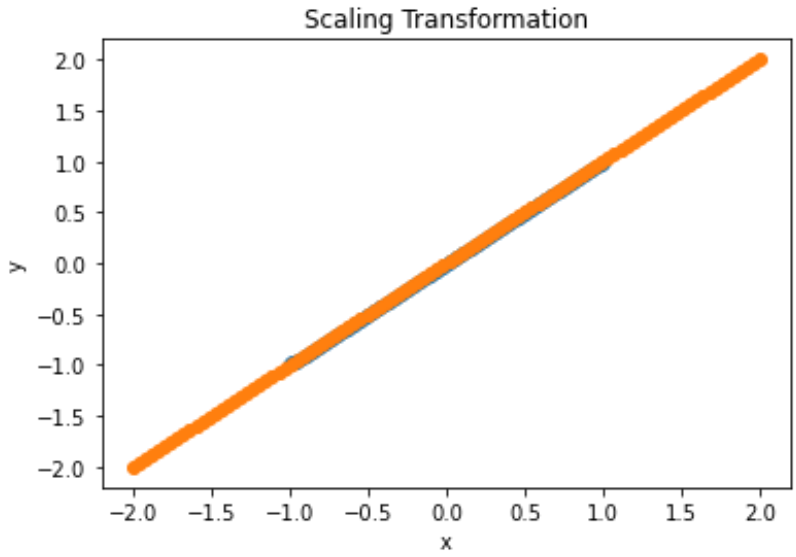


Some common geometric transformations expressed as matrix-vector products in 2D space. In each example, we'll consider the mapping from the vector v to w, where v is a 2D vector representing a position, and A is a 2x2 matrix.

Scaling : Scaling uniformly changes the size of a vector by multiplying each component by a constant scale factor. The scaling transformation can be expressed as: w = [[s_1, 0], [0, s_2]] v This scales v by factors s_1 along the x-axis and s_2 along the y-axis.

In [7]:

```
import matplotlib.pyplot as plt
import numpy as np

# Create a scaling matrix
A = np.array([[2, 0], [0, 2]])

# Create a set of points
x = np.linspace(-1, 1, 100)
y = np.linspace(-1, 1, 100)

# Plot the points
plt.plot(x, y, 'o')

# Apply the scaling transformation to the points
scaled_x = A[0, 0] * x + A[0, 1] * y
scaled_y = A[1, 0] * x + A[1, 1] * y

# Plot the scaled points
plt.plot(scaled_x, scaled_y, 'o')

# Add a title and labels to the axes
plt.title('Scaling Transformation')
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



Dilation:

Dilation is a linear transformation that stretches a vector by different factors along the two different axes. The diagonal matrix D has the scale factors on the diagonal.

In [9]:

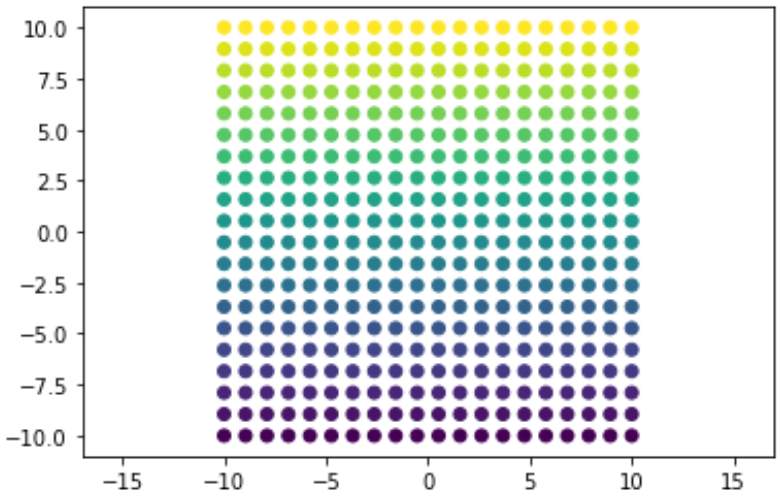
```
x_grid = np.linspace(-10,10,20)
y_grid = np.linspace(-10,10,20)
x_mesh, y_mesh = np.meshgrid(x_grid, y_grid)
```

In [10]:

```
plt.scatter(x_mesh,y_mesh, c=y_mesh)
plt.axis('equal')
```

Out[10]:

```
(-11.0, 11.0, -11.0, 11.0)
```

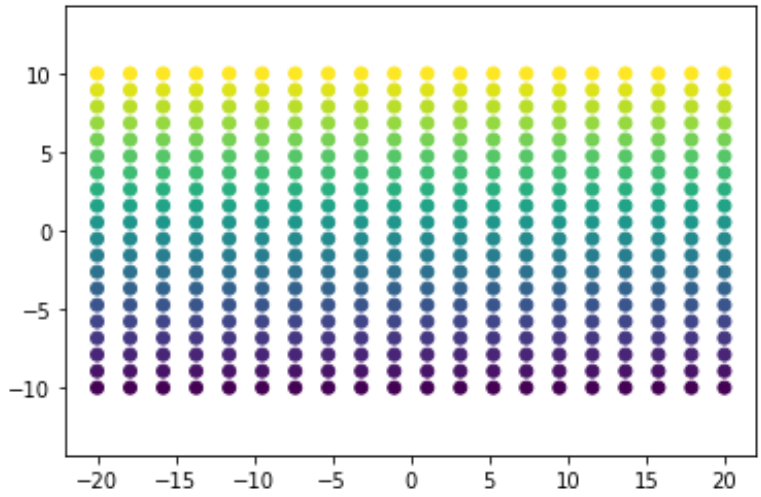


In [11]:

```
#streching along the x axis
A2 = np.zeros((2,2))
A2[0,0] = 2
A2[1,1] = 1

X_new_mesh_2 = A2[0,0] * x_mesh + A2[0,1] * y_mesh
Y_new_mesh_2 = A2[1,0] * x_mesh + A2[1,1] * y_mesh

plt.scatter(X_new_mesh_2,Y_new_mesh_2, c=y_mesh)
plt.axis('equal')
```

Out[11]:

```
(-22.0, 22.0, -11.0, 11.0)
```



Rotation

a rotation matrix is a matrix that can be used to rotate a vector in 2D space. The matrix is made up of the cosine and sine of the angle of rotation, and it is orthogonal, which means that its inverse is its transpose.

To rotate a vector by an angle of θ radians counterclockwise, we can multiply the vector by the rotation matrix. The result will be the rotated vector.

In [12]:

```
A = np.zeros((2,2))
theta = np.pi / 4;
A[0,0] = np.cos(theta)
A[0,1] = - np.sin(theta)
A[1,0] = np.sin(theta)
A[1,1] = np.cos(theta)

print(A)
```

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

In [13]:

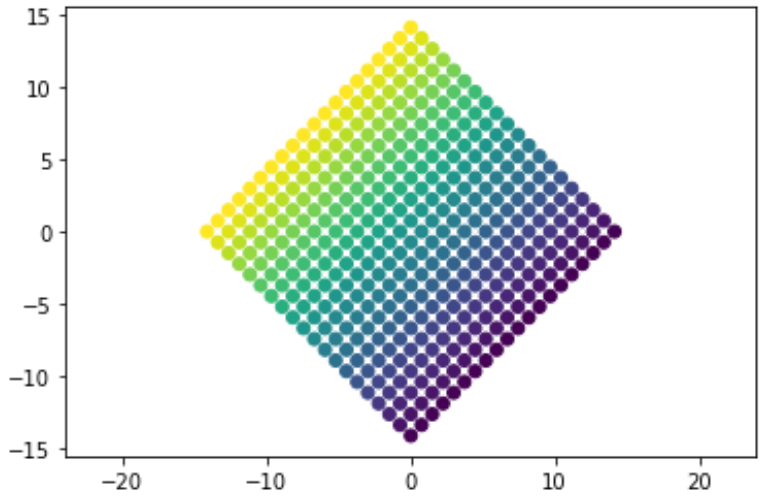
```
X_new_mesh = A[0,0] * x_mesh + A[0,1] * y_mesh
Y_new_mesh = A[1,0] * x_mesh + A[1,1] * y_mesh
```

In [14]:

```
plt.scatter(X_new_mesh,Y_new_mesh, c=y_mesh)
plt.axis('equal')
```

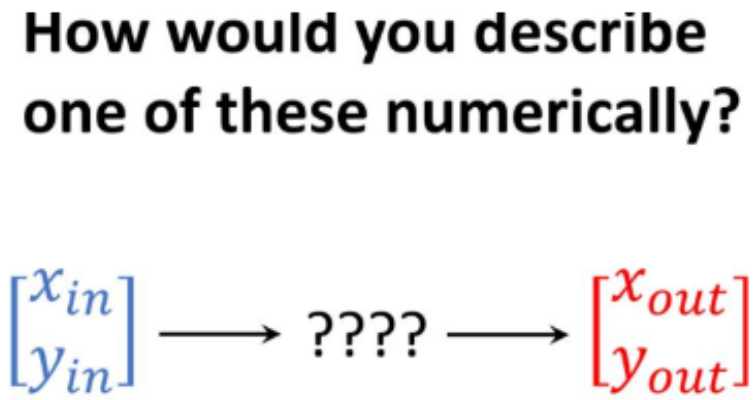
Out[14]:

```
(-15.556349186104047,
 15.556349186104047,
 -15.556349186104047,
 15.556349186104047)
```



Reflection

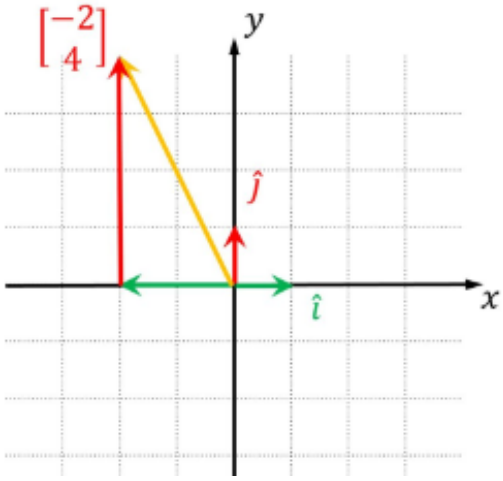
A reflection is a linear transformation that flips a vector across a line. The line that the vector is flipped across is called the mirror line. The matrix for a reflection is a diagonal matrix with 1s on the diagonal and -1s in the off-diagonal elements.



- **Basis vectors** are two vectors that form a coordinate system in a plane. They are usually denoted by $\hat{\imath}$ and $\hat{\jmath}$, and they have a length of 1.

- **Linear combinations** are sums of vectors multiplied by scalars. For example, the vector $\begin{bmatrix} -2 \\ 4 \end{bmatrix}$ is a linear combination of the basis vectors $\hat{\imath}$ and $\hat{\jmath}$, where the scalars are -2 and 4.

$$\vec{v} = -2\hat{\imath} + 4\hat{\jmath}$$



In the case of reflection, the linear transformation takes a vector and reflects it across a line. The line that the vector is reflected across is called the mirror line.

The matrix for a reflection across a line is a diagonal matrix with 1s on the diagonal and -1s in the off-diagonal elements. The line that the vector is reflected across is the line that passes through the origin and is perpendicular to the matrix.

In the case of the reflection described in the question, the mirror line is the line that passes through the origin and is inclined θ radians with respect to the horizontal. The matrix for this reflection is:

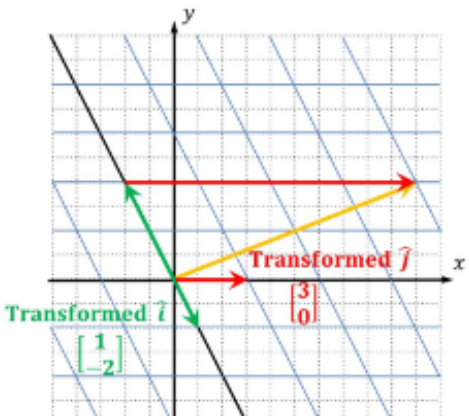
```
R = [[\cos (2\theta), \sin (2\theta)], [\sin (2\theta), -\cos (2\theta)]]
```

This matrix can be used to reflect any vector across the mirror line.

$$\vec{v} = -2\hat{\imath} + 4\hat{\jmath}$$

$$\text{Transformed } \vec{v} = -2\,(\text{Transformed } \hat{\imath}) + 4\,(\text{Transformed } \hat{\jmath})$$

$$= -2\begin{bmatrix} 1 \\ -2 \end{bmatrix} + 4\begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \end{bmatrix}$$



So, we will proceed further as follows. If we want to map our arbitrary input vector (x, y) with a linear transformation, the output vector will preserve this pair of numbers, but they will multiply the transformed basis vectors.

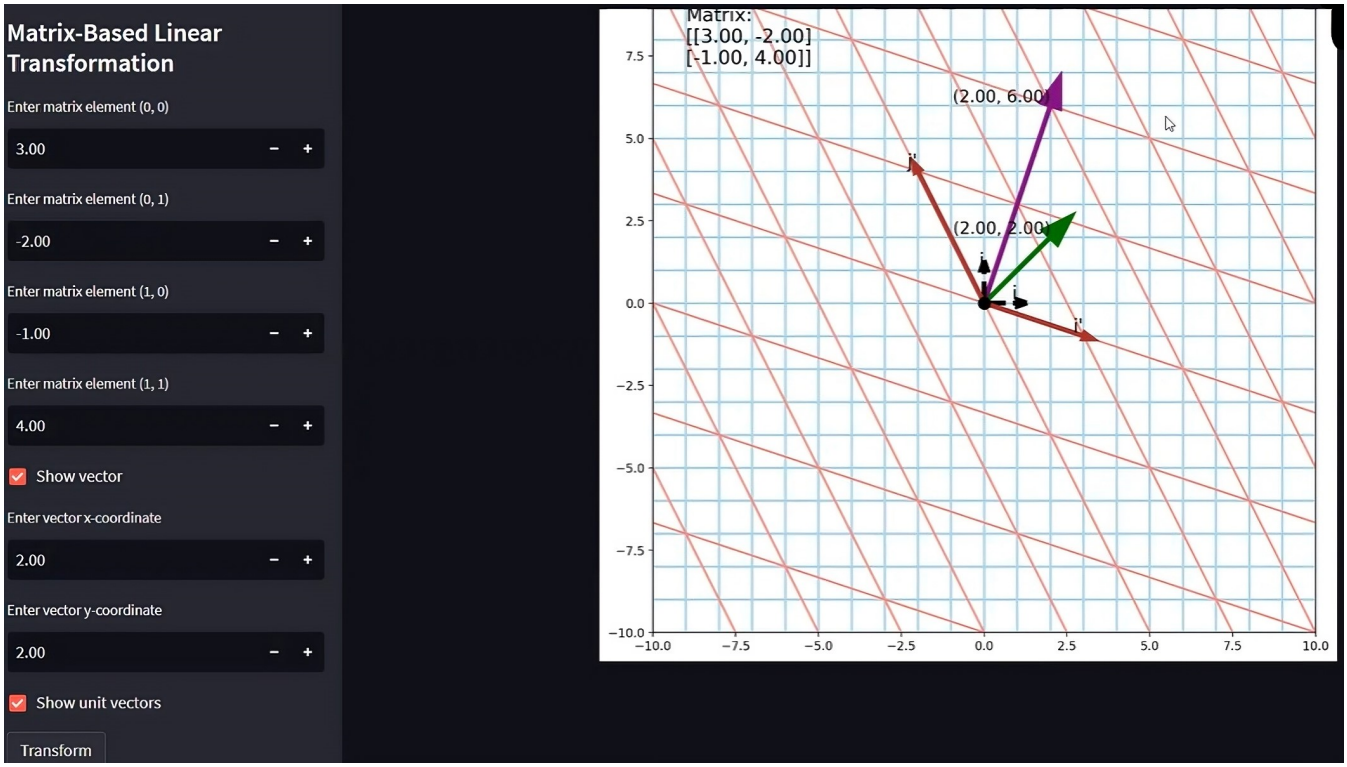
This can also be explained with the following equations and formulas:

$$\hat{\imath} \to \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad \hat{\jmath} \to \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \to x \begin{bmatrix} 1 \\ -2 \end{bmatrix} + y \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 1x + 3y \\ -2x + 0y \end{bmatrix}$$

Matrix-Based Linear Transformation

Matrix-based linear transformations are a convenient and efficient way to represent and work with linear transformations. In this approach, a linear transformation is defined by a matrix that encodes the transformation's effects on the vectors in a given vector space.



A 2×2 Matrix as a linear transformation

The whole transformation can be defined with two transformed basis vectors. These vectors can be stacked along the columns to form a 2x2 matrix. This matrix can then be used to transform any vector in the plane.

In the case of the reflection described in the question, the transformed basis vectors are:

\(\begin{bmatrix}1 & 3\\-2 & 0\end{bmatrix}\)

These vectors can be stacked along the columns to form the matrix:

```
R = [[1, 3], [-2, 0]]
```

This matrix can then be used to reflect any vector in the plane.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

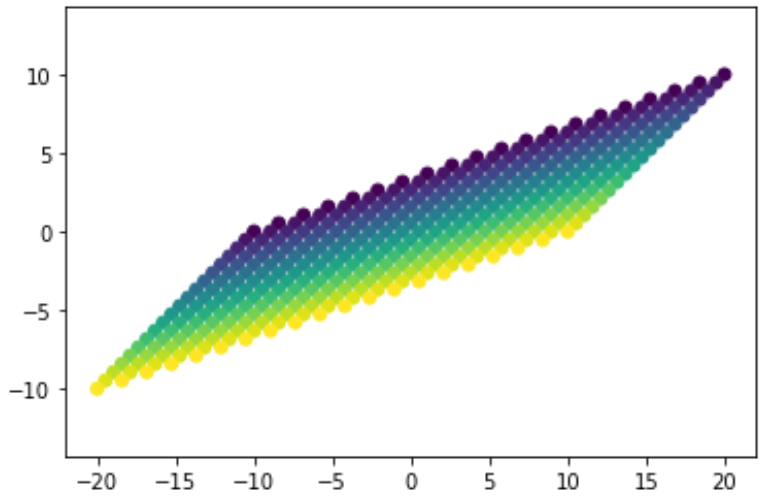
it's just the scaling two column vectors and then summing them and this is what we get as the resulting output. This can be more intuitive way to think about the matrix-vector multiplication.

In [18]:

```
# General linear mapping
# A matrix A has two linearly independent columns.

A3 = np.zeros( (2,2))

A3[0,0] = 3./2
A3[0,1] = -1/2
A3[1,0] = 1./2
A3[1,1] = -1./2

X_new_mesh_3 = A3[0,0] * x_mesh + A3[0,1] * y_mesh
Y_new_mesh_3 = A3[1,0] * x_mesh + A3[1,1] * y_mesh

plt.scatter(X_new_mesh_3, Y_new_mesh_3, c=y_mesh)
plt.axis('equal')
```

Out[18]:

```
(-22.0, 22.0, -11.0, 11.0)
```



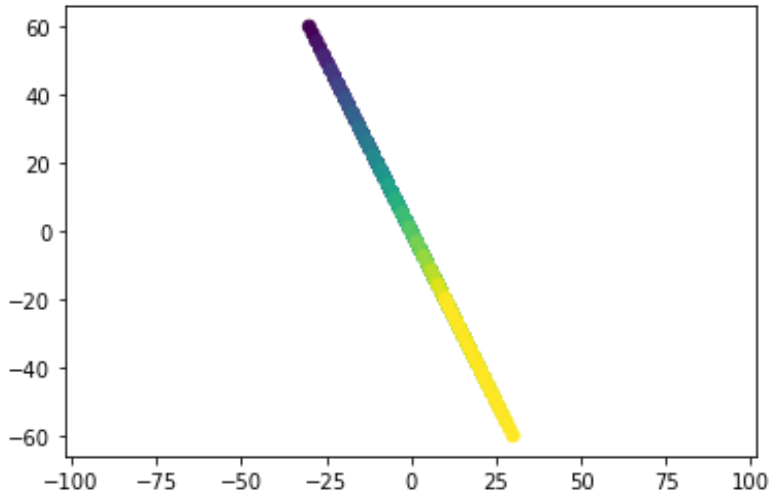
If we have a matrix whose vectors are linearly dependent, then, a 2-D plane will be mapped to a single line. It will not be possible to go back and perform the reconstruction.

In [19]:

```
# General linear mapping
# A matrix A has two linearly dependent columns.

A4 = np.zeros((2,2))

A4[0,0] = 1
A4[0,1] = 2
A4[1,0] = -2
A4[1,1] = -4

X_new_mesh_4 = A4[0,0] * x_mesh + A4[0,1] * y_mesh
Y_new_mesh_4 = A4[1,0] * x_mesh + A4[1,1] * y_mesh

plt.scatter(X_new_mesh_4, Y_new_mesh_4, c=y_mesh)
plt.axis('equal')
```

Out[19]:

```
(-33.0, 33.0, -66.0, 66.0)
```



Matrix Multiplication as Composition

it means More than one linear transformations applies to a graph one by one.

In linear algebra, a linear transformation is a mapping that takes a vector in one space and maps it to a vector in another space. It can be represented by a matrix.

Matrix multiplication is the operation of multiplying two matrices together. It can be seen as composing two linear transformations.

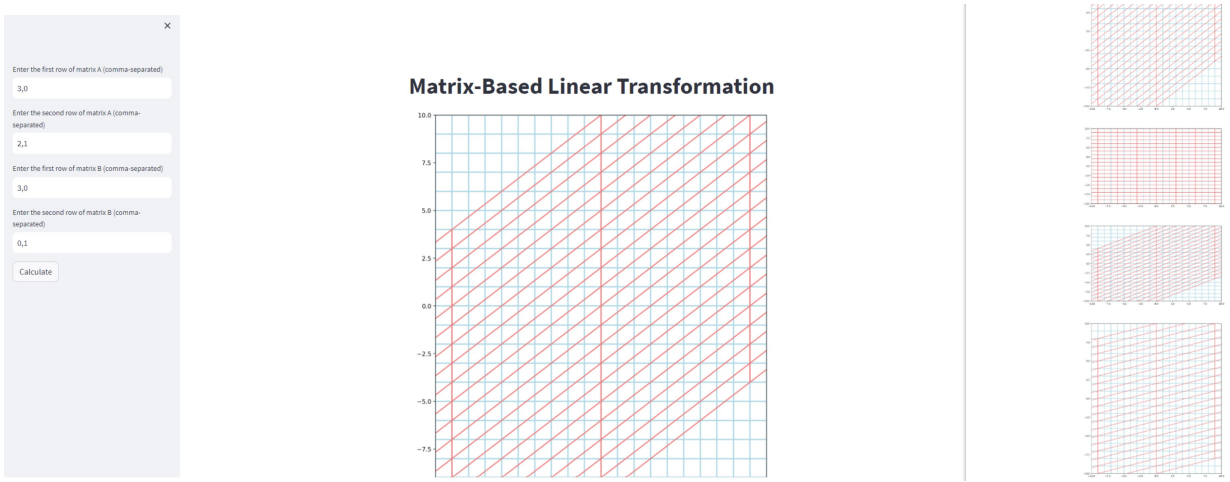
For example, let's say we have two linear transformations T and S. We can represent them as matrices A and B, respectively. Then, the matrix multiplication AB can be seen as composing the two transformations T and S.

In other words, if we have a vector v, then ABv is the vector that results from applying the transformation S to the vector that results from applying the transformation T to v.

Here is a diagram that illustrates this:

```
v → T(v) → S(T(v)) → ABv
```

As you can see, matrix multiplication can be seen as composing two linear transformations. This is a powerful property that makes matrix multiplication a very useful tool in mathematics and computer science.

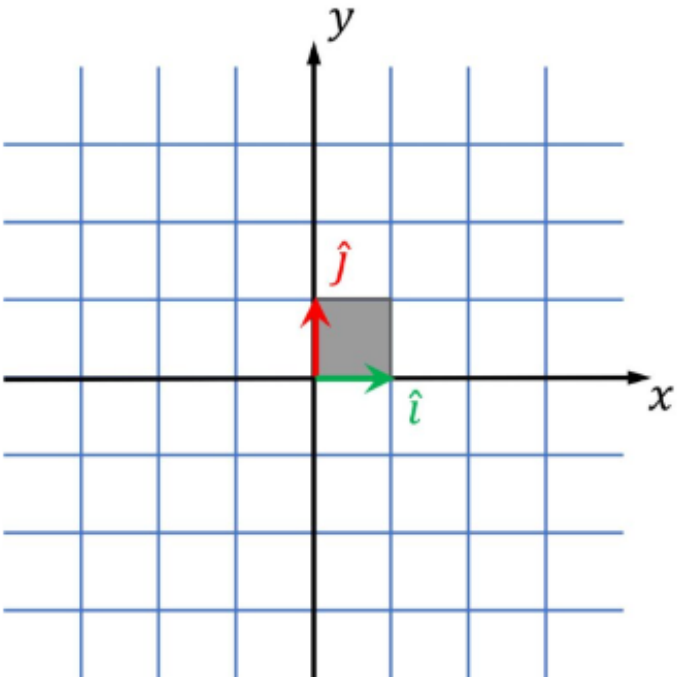


Determinant in a 2-D coordinate system

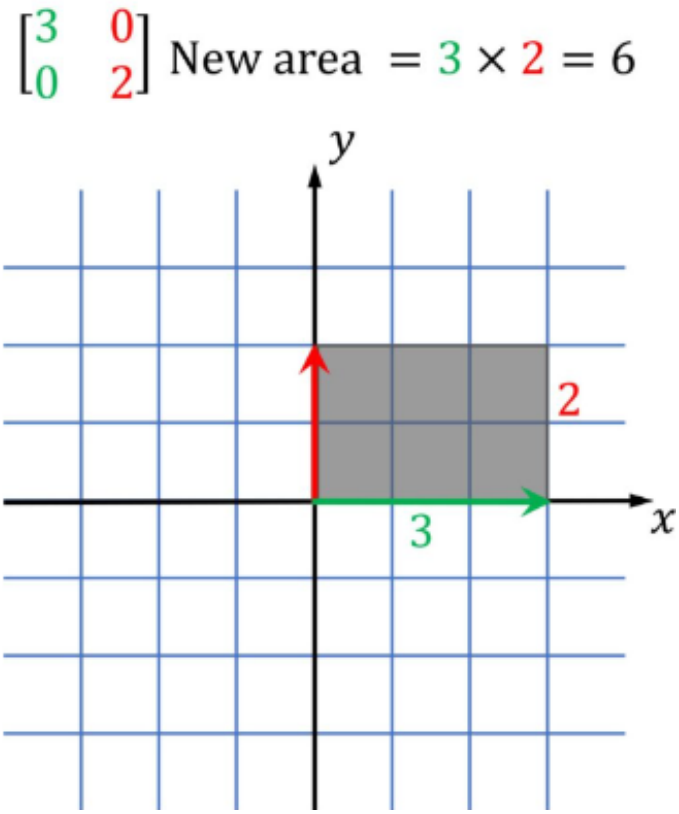
The determinant of a matrix can be used to measure how much the area of a parallelogram is changed under a linear transformation. In the case of a dilation, the determinant of the matrix will be equal to the scale factor of the dilation. This means that the determinant of the matrix will tell us how much the area of a unit square is changed under the dilation.

For example, if we have a dilation with a scale factor of 2, then the determinant of the matrix will be 2. This means that the area of a unit square will be doubled under the dilation.

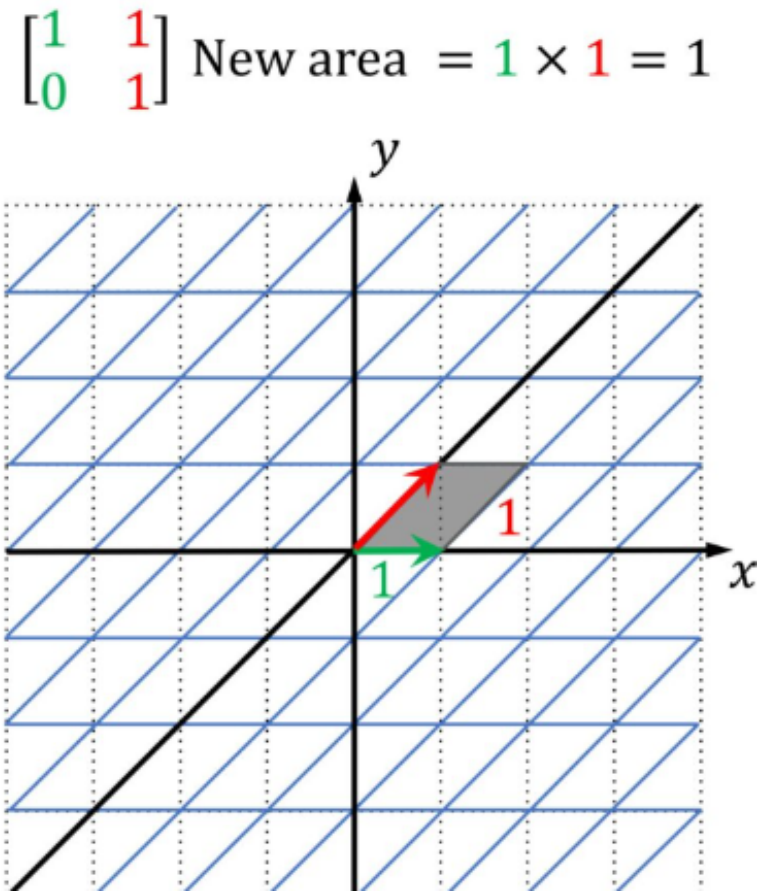
The determinant of a matrix can also be used to determine whether a linear transformation is invertible. A linear transformation is invertible if and only if the determinant of the matrix is not equal to 0. This means that the determinant of a matrix can be used to tell us whether a linear transformation is reversible.



In this case, a unit square will be mapped into a rectangle of size 6. Hence, we can say that the area has increased 6 times.



In the case of a shear, a unit square is mapped into a unit parallelogram. This is due to the fact that the area of the parallelogram is a “length x height”. So, in this case the area will remain the same, but the square will be stretched.



Determinant of Matrix

The determinant of a matrix can be used to determine the type of transformation that the matrix represents.

For example,

- if the determinant of a matrix is equal to 1, then the matrix represents an identity transformation. An identity transformation is a transformation that does not change the coordinates of a vector.
- If the determinant of a matrix is equal to 0, then the matrix represents a singular transformation. A singular transformation is a transformation that does not have an inverse.
- If the determinant of a matrix is greater than 1, then the matrix represents a dilation transformation. A dilation transformation is a transformation that changes the size of a vector, but does not change its direction.
- If the determinant of a matrix is less than 1, then the matrix represents a contraction transformation. A contraction transformation is a transformation that changes the size of a vector, but in the opposite direction of a dilation transformation.

Here is a table that summarizes the different types of transformations and their determinants:

Type of transformation	Determinant
Identity transformation	1
Singular transformation	0
Dilation transformation	Greater than 1
Contraction transformation	Less than 1

In [21]:

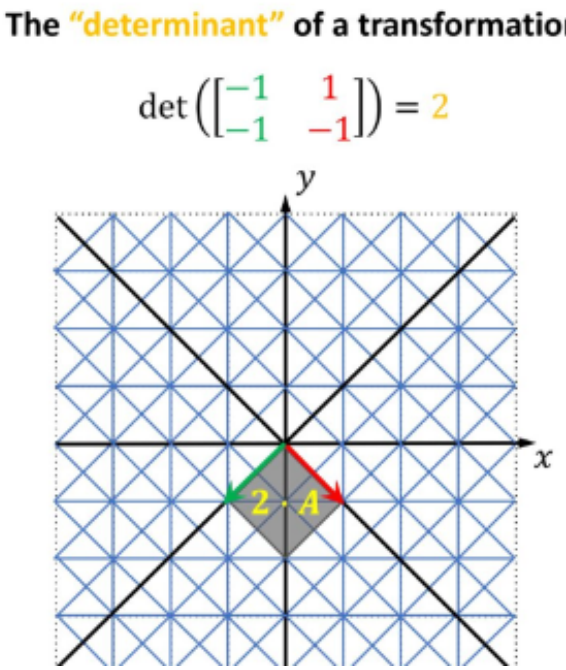
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
a1 = np.array([[-1, 1], [-1, -1]])

det = a1[0][0] * a1[1][1] - a1[1][0] * a1[0][1]
print(det)
```

```
2
```

In [22]:

```
det = np.linalg.det(a1)
print(det)
```

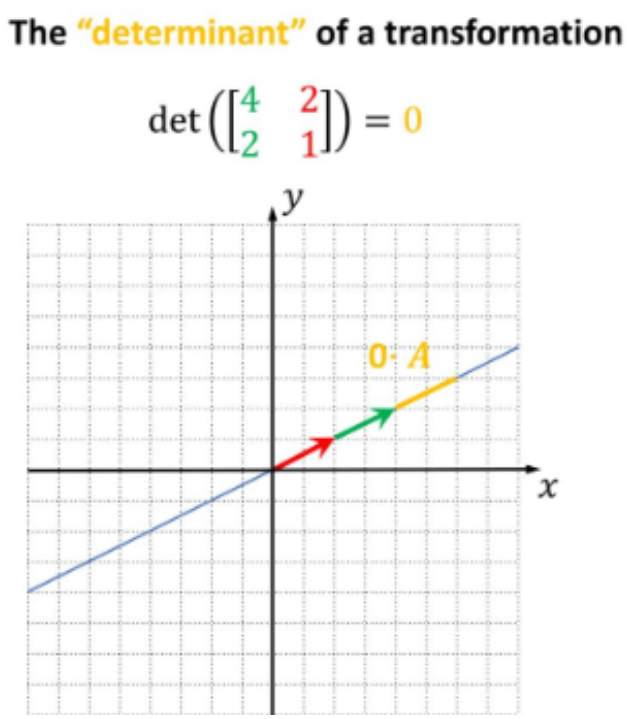
```
2.0
```



The determinant of a single line is 0

The determinant of a single line is 0. This is because the area of a single line is 0. The determinant of a matrix can be interpreted as the area of the parallelogram formed by the two column vectors of the matrix. If the two column vectors are dependent, then they will lie on the same line, and the area of the parallelogram will be 0.

The transformation matrix that maps a 2-D plane to a line is a 2x2 matrix with one of the column vectors equal to the zero vector. The determinant of this matrix is 0, as expected.



In [3]:

```
import numpy as np

a2 = np.array([[4, 2], [2, 1]])
det = np.linalg.det(a2)

print(det)
```

```
0.0
```

The determinant of a matrix can be positive or negative

A positive determinant means that the transformation preserves the orientation of the space, while a negative determinant means that the transformation reverses the orientation of the space.

For example,

if we have a 2-D plane with the vectors $\hat{i}$ and $\hat{j}$ pointing in the positive x and y directions, respectively, and we apply a transformation that preserves the orientation of the space, then $\hat{j}$ will still be to the left of $\hat{i}$ after the transformation.

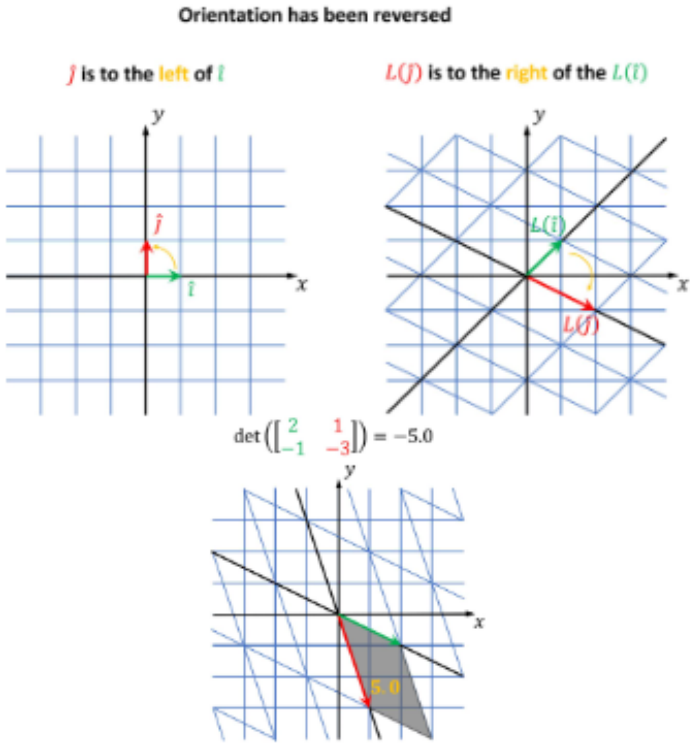
However, if we apply a transformation that reverses the orientation of the space, then $\hat{j}$ will be to the right of $\hat{i}$ after the transformation.

The determinant of a matrix that reverses the orientation of the space is negative. This is because the determinant of a matrix can be interpreted as the signed area of the parallelogram formed by the two column vectors of the matrix. If the transformation reverses the orientation of the space, then the parallelogram will be flipped, and the area will be negative.

For example, the matrix `[[-1, 0], [0, 1]]` represents a reflection across the x-axis. If we apply this matrix to the vectors $\hat{i}$ and $\hat{j}$, then $\hat{j}$ will be flipped to the right of $\hat{i}$. The determinant of this matrix is -1, which confirms that it reverses the orientation of the space.

The sign of the determinant can be used to determine whether a transformation preserves or reverses the orientation of the space. This is a useful property for many applications, such as computer graphics and robotics.

In simpler terms, **the determinant of a matrix can be used to determine whether a transformation flips the orientation of a space. A positive determinant means that the transformation does not flip the orientation, while a negative determinant means that the transformation does flip the orientation.**

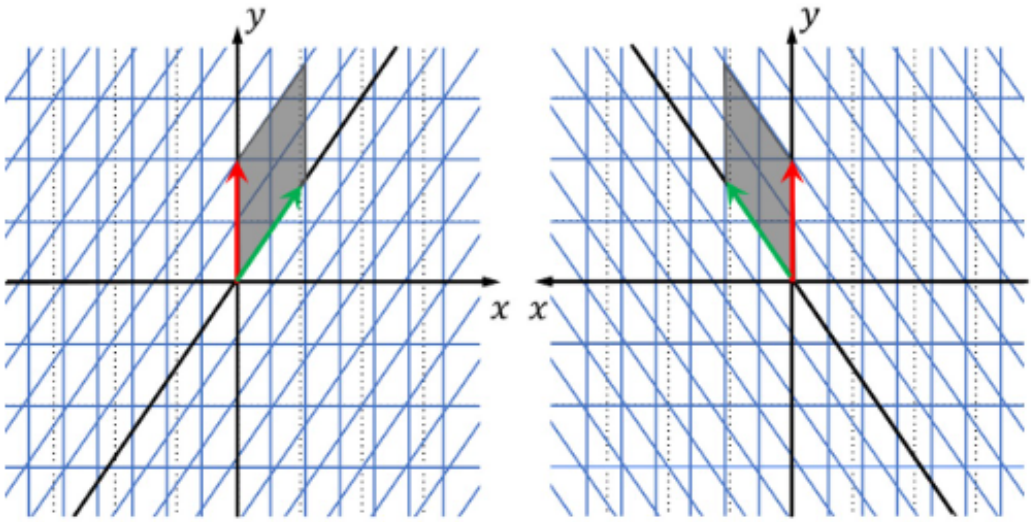


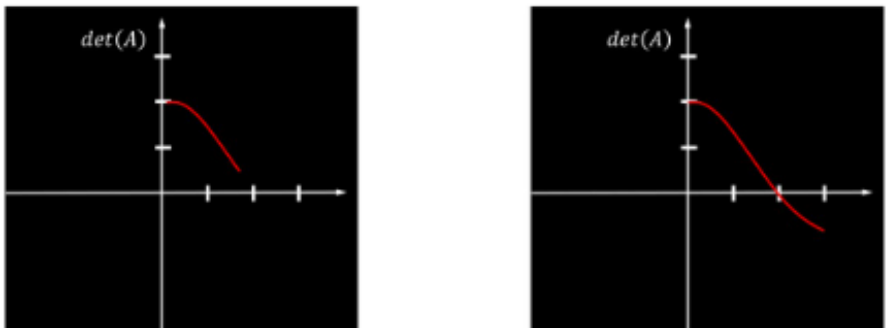
In [4]:

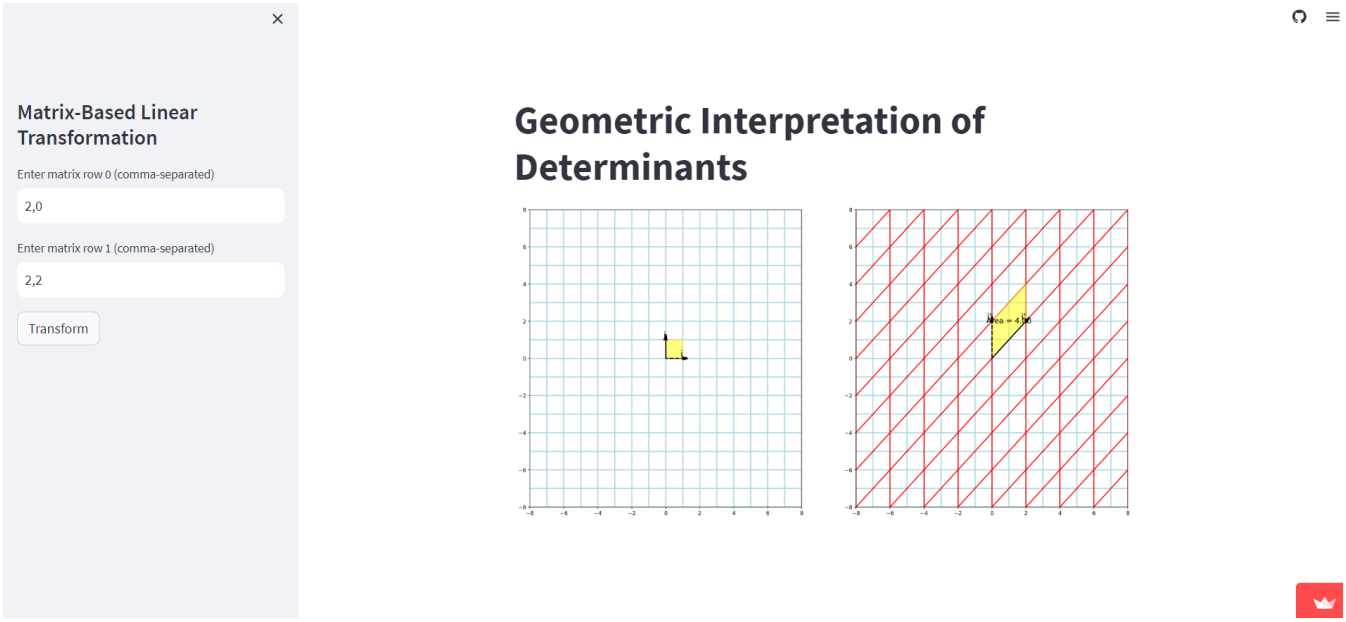
```
a3 = np.array([[2, 1], [-1, -3]])
det = np.linalg.det(a3)

print(det)
```

```
-5.000000000000001
```

A nice intuition about the determinant and its sign we can get with the following experiment. Imagine that we start stretching a unit square to some position defined with the left image below. The more we stretch the point of a parallelogram (opposite of the coordinate origin) and move it closer to the y axis, the value of the determinant is getting smaller and smaller. Once we “cross” on the other side of the y axis, we actually obtain a reversed order of axes and the determinant becomes negative. If we continue this process, our parallelogram is increasing its size, but the sign of the determinant remains negative. This is illustrated in the right image below.

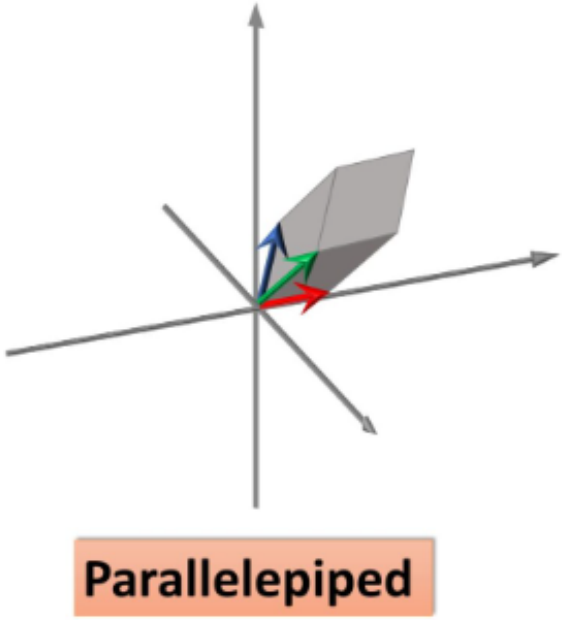






Determinant in a 3-D coordinate system

what is actually happening in a 3-D space? We have volumes there, right? Well, the idea is quite similar. Instead of an area that we used to calculate determinant for the plane, in a 3-D space we actually use determinant to calculate a volume. Now, instead of a unit square we have a unit cube. Thus, our determinant will tell us how much the volume of that unit cube will change when we apply a linear transformation. So, we will get something called **parallelepiped**. This can be one illustrated with the image below, and how we can change and transform our 3-D space.



In [5]:

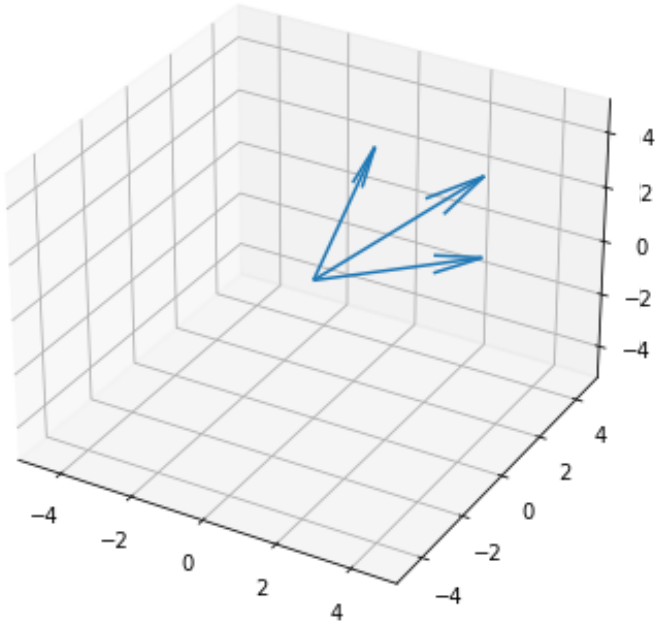
```
a4 = np.array([[3,0,3],[3,3,3],[0,3,3]])

det = np.linalg.det(a4)
print(det)
```

```
27.0
```

In [8]:

```
from mpl_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6,6))
ax = fig.gca(projection='3d')

origin = np.zeros(a4.shape) # origin point

ax.quiver(*origin, *a4)

plt.grid()
ax.set_zlim3d(-5, 5)
ax.set_ylim3d(-5, 5)
ax.set_xlim3d(-5, 5)
plt.show()
```

```
C:\Users\user\AppData\Local\Temp/ipykernel_4988/3259830683.py:6: Matplo
tlibDeprecationWarning: Calling gca() with keyword arguments was deprec
ated in Matplotlib 3.4. Starting two minor releases later, gca() will t
ake no keyword arguments. The gca() function should only be used to get
the current axes, or if no axes exist, create new axes with default key
word arguments. To create a new axes with non-default arguments, use pl
t.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

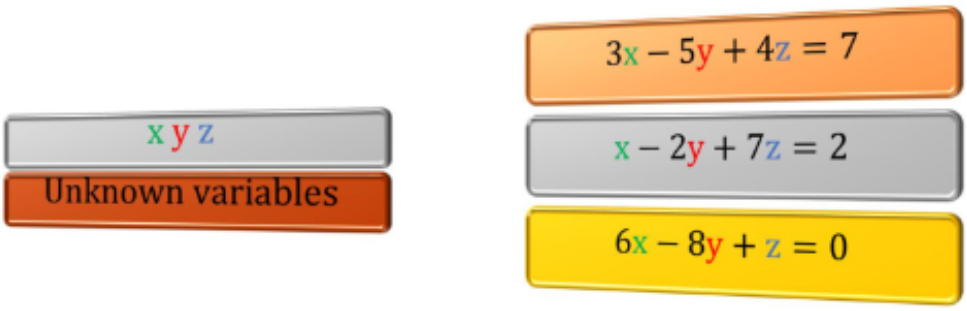


a determinant can be negative. Again, this can happen with the inverse order of the axes. In addition, the difference is that now we also have three vectors of size 3. One way to calculate it is to decompose as a linear combination of three determinants of 2×2 matrices.

$$\det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) = a\det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b\det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c\det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right)$$

Inverse matrices

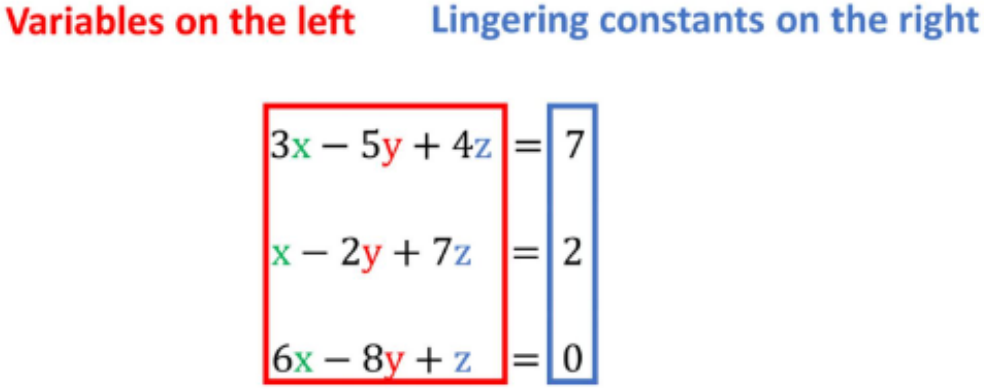
Usually, when you first hear about linear algebra, the first thing that pops into your mind are equations. In particular, solving equations of linear systems.



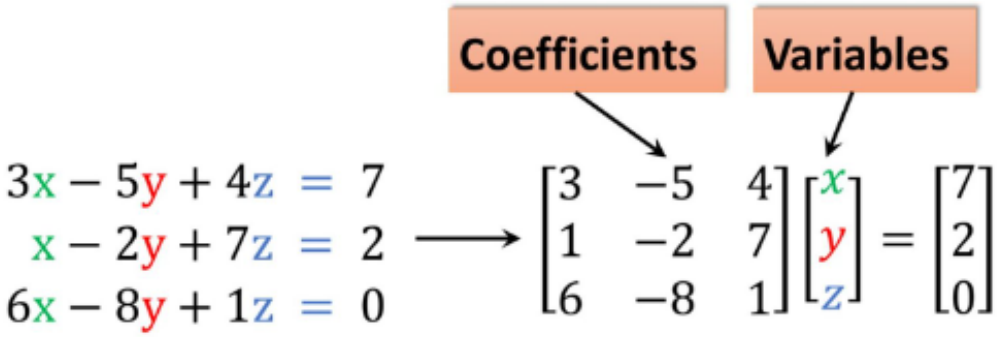
Above, we have an example of linear equations. The reason that we call them linear is because all variables that we need to find (x), (y) and (z) are actually multiplied with scalars. So, there are no any complex relation like (\sin(x)), (x^{2}) or (y^{3}).

In other words, if you observe these variables (x, y) and (z) they are only scaled by a scalar (a number) and they are just summed. We don't allow any fancy mathematical operations for instance (x\cdot y) when we speak about linear equations.

Inverse matrices, linear equations So, how we usually proceed with solving these equations is that we first have the variables and we align them on the left. Then, we also have, the so called, **lingering constraint** on the right and we will put them into another matrix.



So, how we usually proceed with solving these equations is that we first have the variables and we align them on the left. Then, we also have, the so called, lingering constraint on the right and we will put them into another matrix.



In [1]:

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

In [2]:

```
a = np.array([[3,-5,4],[1,-2,7],[6,-8,1]])
b = np.array([[7],[2],[0]])
```

In [3]:

```
x = np.linalg.solve(a, b)
print(x)
```

```
[[-12.        ]
 [ -9.07407407]
 [ -0.59259259]]
```

The system of equations can be written as a matrix-vector multiplication, where the matrix represents the coefficients of the equations and the vector represents the variables. This is a very convenient way of representing linear equations, and it makes it easy to solve them using matrices.

The matrix-vector multiplication for the system of equations you have shown is:

```
A * x = b
```

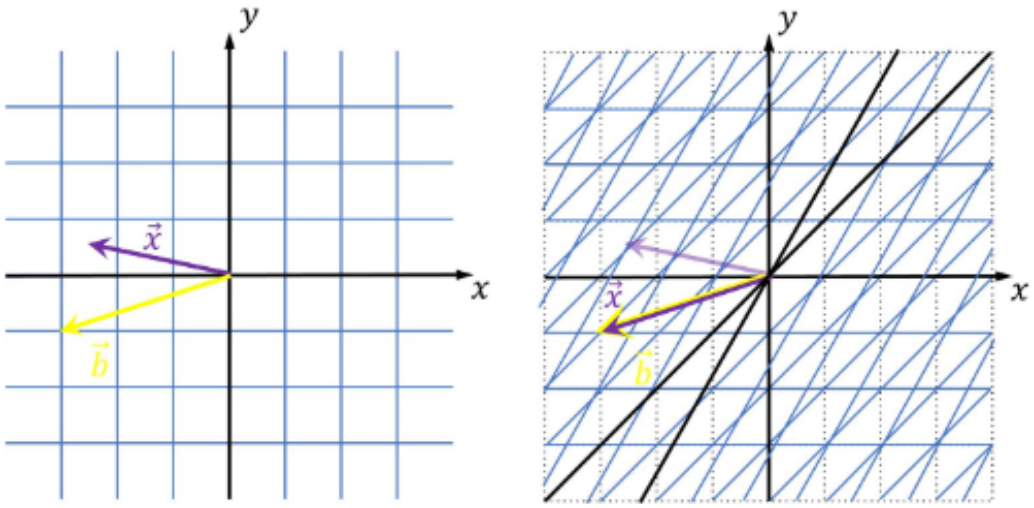
where:

- A is the coefficient matrix, which is a 3x3 matrix in this case.
- x is the variable vector, which is a 3x1 vector in this case.
- b is the constant vector, which is a 3x1 vector in this case.

The matrix-vector multiplication can be solved to find the values of x, which are the solutions to the system of equations.

The fact that the system of equations can be written as a matrix-vector multiplication shows that linear equations are closely related to matrices. This is why matrices are such a powerful tool for solving linear

$$A\vec{x} = \vec{b}$$



example Here, we have an example how a 2-D input vector can be transformed into (\vec{b}).

$$2x + 2y = -4$$

$$1x + 3y = -1$$

$$\begin{bmatrix} 2 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -4 \\ -1 \end{bmatrix}$$

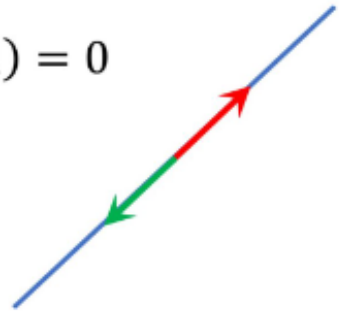
$$A \quad \vec{x} \quad \vec{b}$$

Our goal is to solve our system of equations in 2-D space where we are going to have our vector (\vec{x}) with the coordinates (x) and (y). We would like to see how this system can be transformed into this so-called interpretation where vector (\vec{x}) is mapped into (\vec{v}) using a matrix (A). So, in other words, we can say that (\vec{x}) is our input vector and it is transformed with a matrix (A). So, it moved somewhere in the 2-D space. Hence, we have obtained our resulting vector (\vec{v}). However, in this situation we don't know where the (x) was, we just know the result, so actually we somehow have to go back from (\vec{v}) to (x) in order to reconstruct what the (x) was.

Here, we should recall that actually two things can happen. One is that we have a matrix (A) that actually transforms the 2-D plane into a line. Of course, that will happen when we have linearly dependent columns in (A).

$$A\vec{x} = \vec{b}$$

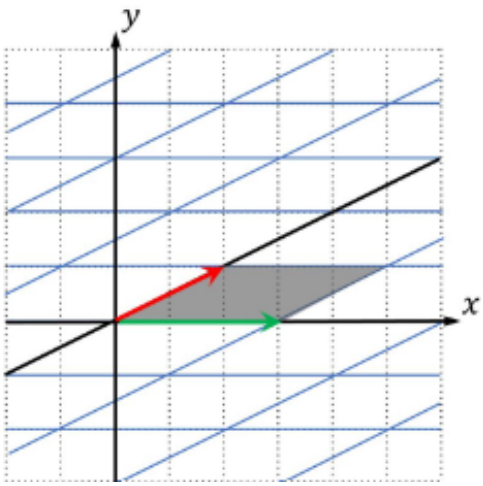
$$det(A) = 0$$



As an alternative, we have an example where our linear transformation just a little bit modifies our unit square. This transformation gives us something that we commonly expect. Hence, will have to examine what of these two cases we have to observe our determinant of a matrix (A).

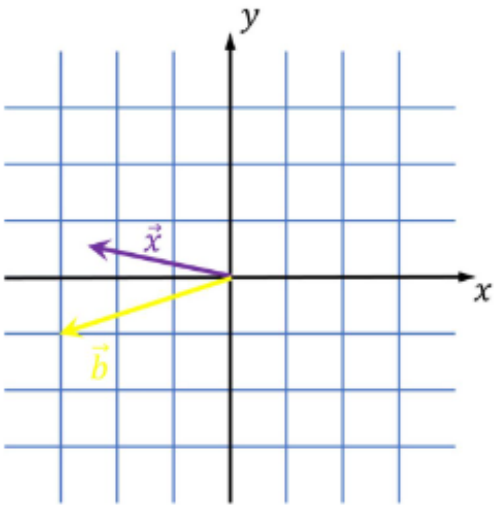
$$A\vec{x} = \vec{b}$$

$$det(A) \neq 0$$



Moreover, if the determinant of (A) is different from zero that means that we will probably have something that we call a unique solution.

So, imagine that we can start with something that's more intuitive and that we actually have a regular linear transformation. In such case, our (\vec{x}) will be mapped to (\vec{b}). We can go back from (\vec{b}) to (\vec{x}) in order to search for (\vec{x}).



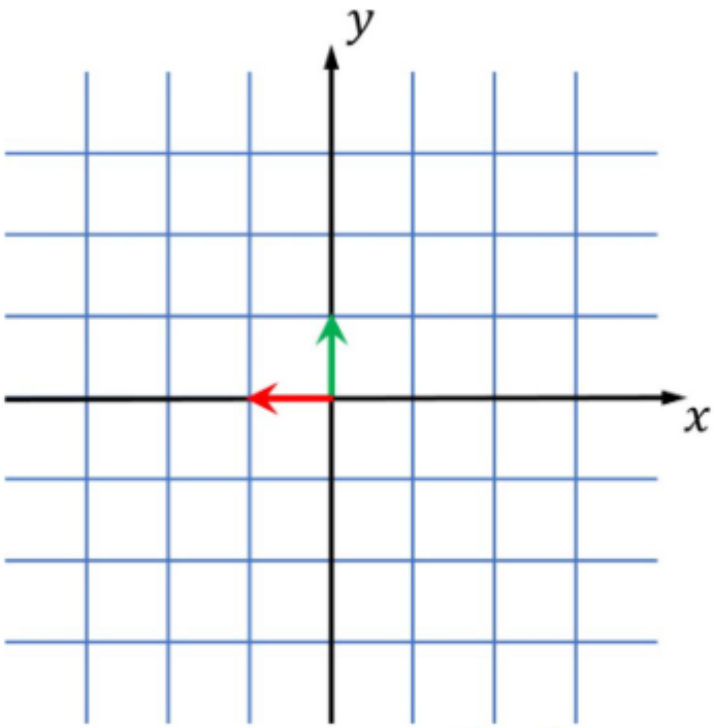
So, **what does it mean to go backward?** We can illustrate this for some simple transformation where for instance we have a counter clockwise (90^{\circ}) rotation. Then, our basis vector ended up so they are rotated in this way.

What an inverse transformation should do? Obviously, it should give us back our original basis vectors (\hat{i}) and (\hat{j}). That means that we will now rotate those vectors back in a clockwise direction for (90^{\circ}).

Now the question is what matrix will achieve this backward mapping? Such a transformation, we call an inverse transformation or an inverse matrix (A). This will be a matrix that will give us back original basis vectors by applying this transformation. This is illustrated in the following two images.

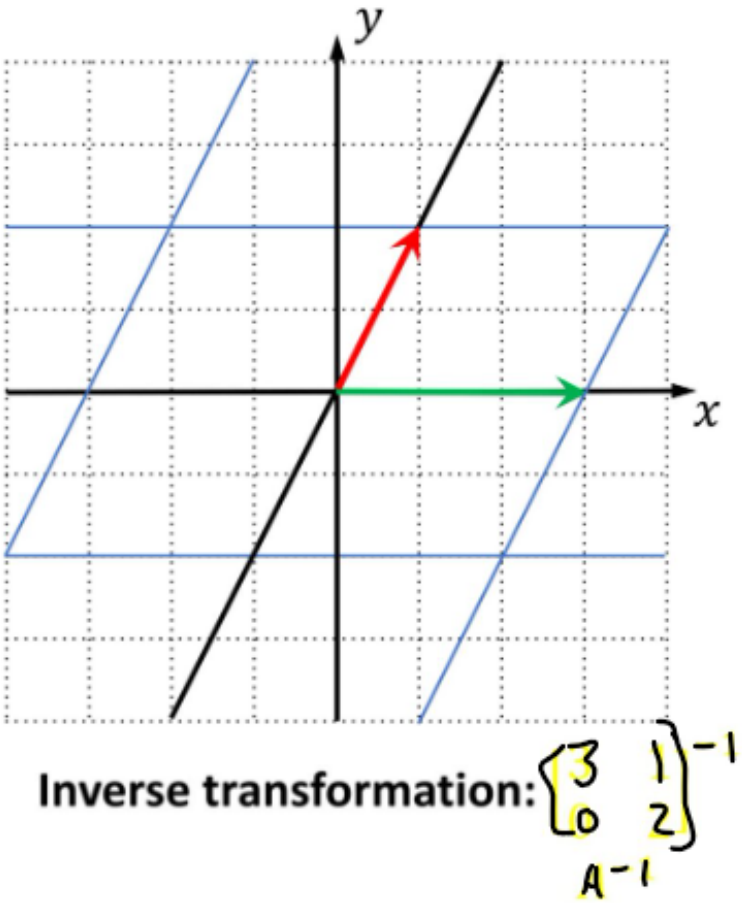
Transformation: $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

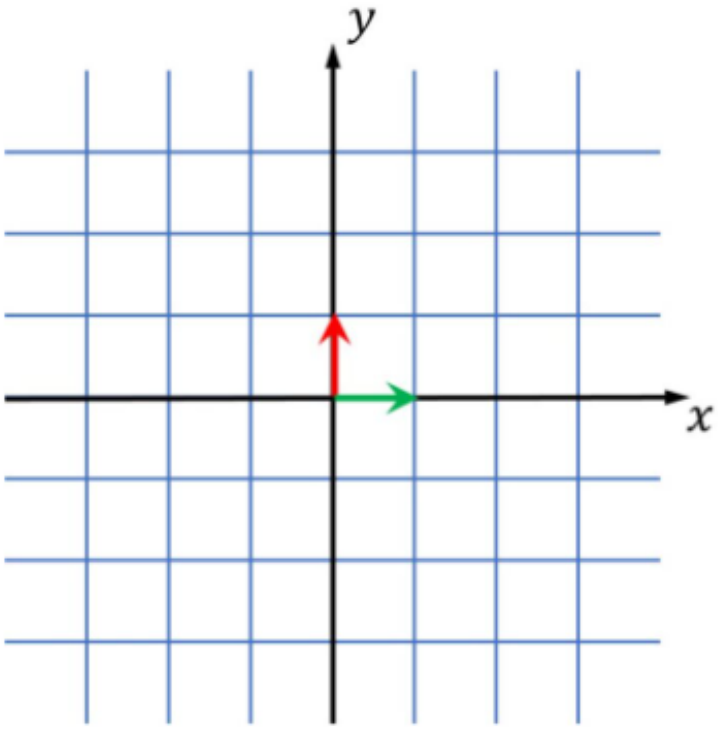
90° Couterclokwise



Transformation: $\begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$

A





But, what is the most important is that if we apply consecutively (A) and then (A^{-1}) as a transformation we should go back to our original basis vectors.

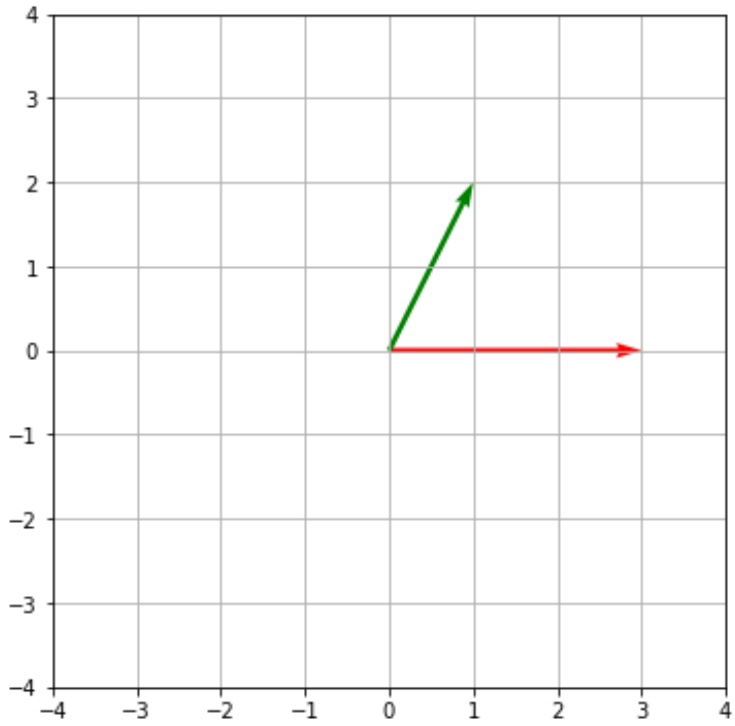
In [4]:

```
A = np.array([[3, 1],
              [0, 2]])

print(np.linalg.det(A))
```

```
6.0
```

In [5]:

```
origin = np.zeros(A.shape)

plt.figure(figsize=(6,6))
plt.quiver(*origin, *A, color=['r','g','b'], scale=1, units='xy')

plt.grid()
plt.xlim(-4,4)
plt.ylim(-4,4)
plt.gca().set_aspect('equal')
plt.show()
```



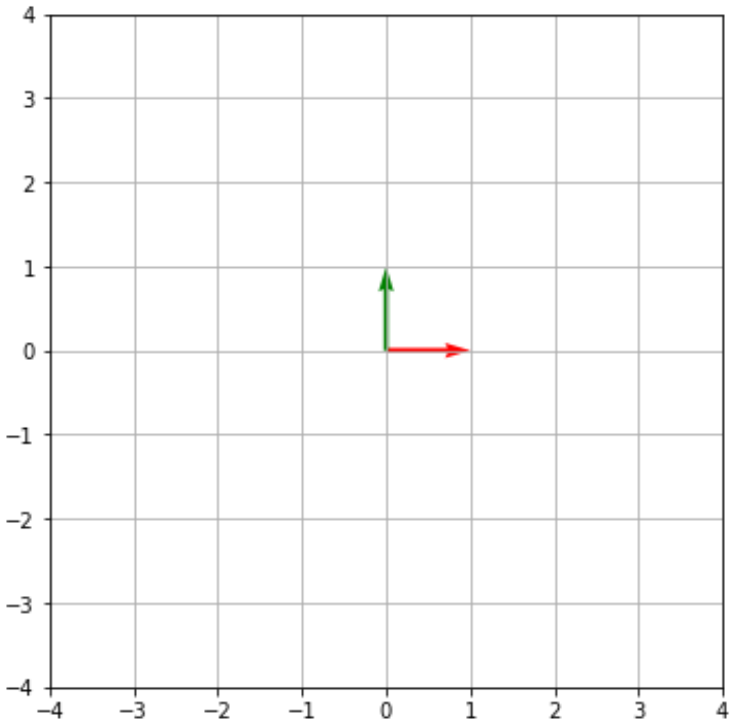
In [6]:

```
A_inv = np.linalg.inv(A)

print(A_inv)
```

```
[[ 0.33333333 -0.16666667]
 [ 0.          0.5       ]]
```

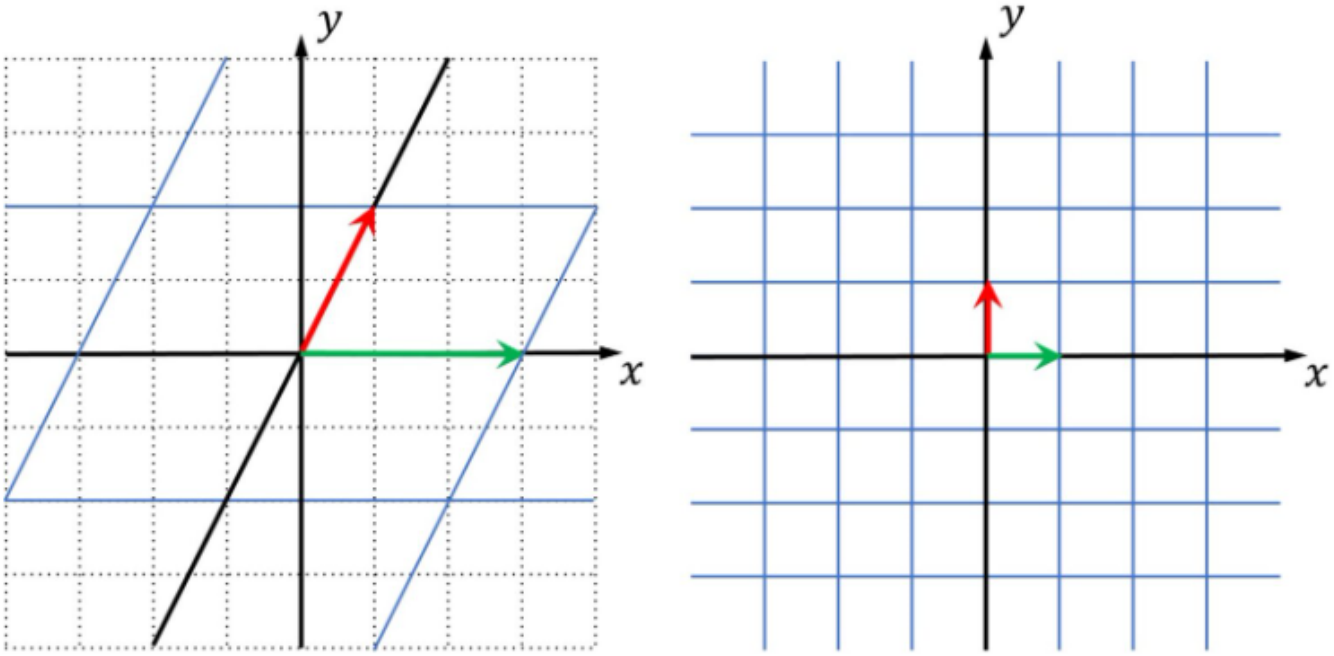
In [7]:

```
b = np.dot(A, A_inv)
origin = np.zeros(b.shape)

plt.figure(figsize=(6,6))
plt.quiver(*origin, *b, color=['r','g','b'], scale=1, units='xy')

plt.grid()
plt.xlim(-4,4)
plt.ylim(-4,4)
plt.gca().set_aspect('equal')
plt.show()
```



$$\underbrace{A^{-1}A}$$

Inverse transformation



Also, here we get one interesting concept and that's the **identity transformation** . So, if we combine (A) and (A^{-1}), with the assumption that (A) determinant is not equal to zero, we will obtain the so-called identity matrix. It is defined in such a way that here we have (\hat{i}) vector and (\hat{j}) vector and they are (

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

) begin{bmatrix}0\1\end{bmatrix}

Now, we can try to solve our equation if we multiply both the left and the right hand side of our equation with (A^{-1}). Then, this term (A^{-1}) times (A) matrix will be an identity matrix. This will give us a vector (\vec{x}) on the left hand side and on the right hand side we will have a matrix-vector multiplication. However, our new matrix will be an inverse matrix of (A).

$$\underbrace{A^{-1}A\vec{x}} = A^{-1}\vec{b}$$

The "do nothing" matrix

On the other hand, when our determinant of a matrix (A) is equal to (0) then any vector will be squashed into the line. This means that the vector mapped into this line, cannot determine its location of origin. That is, from the mapped vector in a 1-D, we cannot uniquely reconstruct the original vector in a 2-D plane.

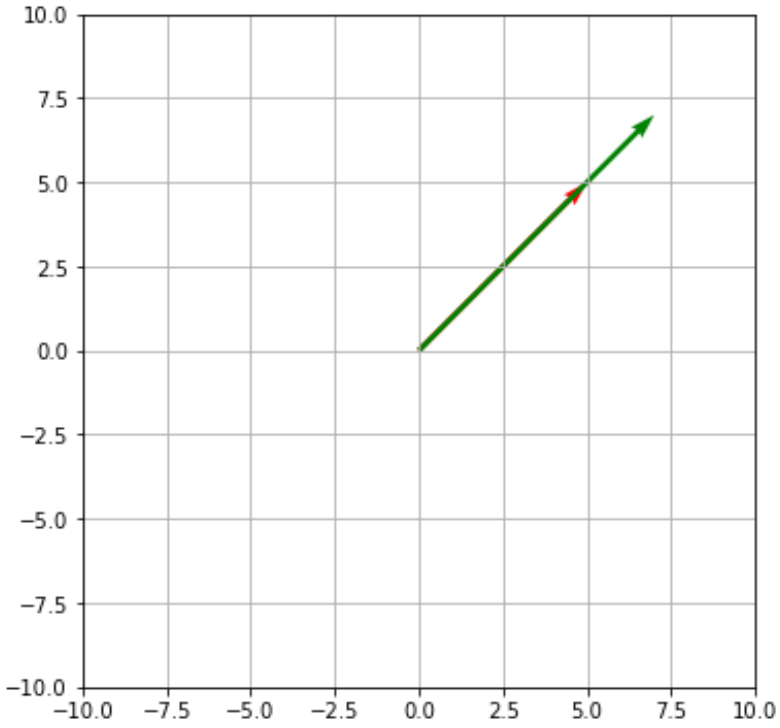
In [9]:

```
A = np.array([[1, 2],
              [1, 2]])

print(np.linalg.det(A))
```

```
0.0
```

In [10]:

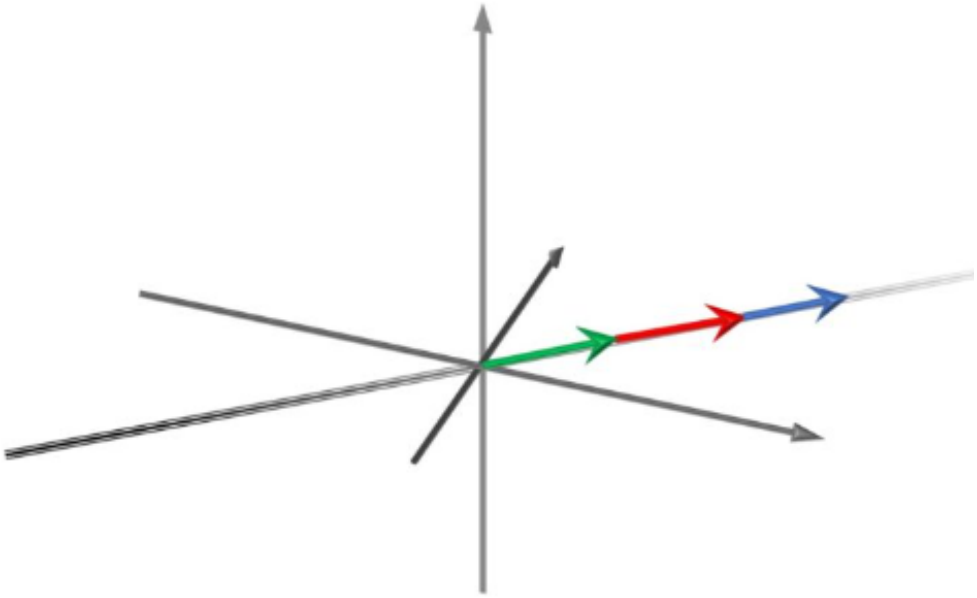
```
B = np.array([[1],[2]])
C = np.array([[3],[2]])

B_new = np.dot(A,B)
C_new = np.dot(A,C)
vectors = np.concatenate((B_new, C_new), axis=1)

origin = np.zeros(vectors.shape)

plt.figure(figsize=(6,6))
plt.quiver(*origin, *vectors, color=['r','g','b'], scale=1, units='xy')
plt.grid()
plt.xlim(-10,10)
plt.ylim(-10,10)
plt.gca().set_aspect('equal')
plt.show()
```



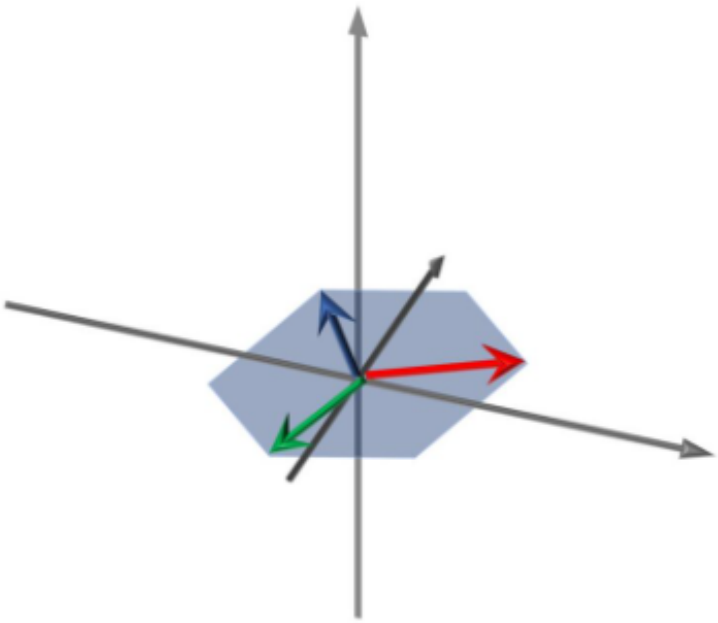
It's also interesting that in a 3-D, if the determinant is zero, then a 3-D vector will be mapped into a 2-D plane (or even a 1-D line). In this case we also cannot uniquely find an inverse transformation that will map vectors from 2-D plane into a 3-D vectors.

Rank

Here are some new definitions that can help us when we work with linear transformations and matrices. In essence, we can say that if a linear transformation lands on a line which is a 1-D we say that we have a rank-1 transformation. If we have a rank-2 transformation then any of ours 3-D vectors will land into a plane. Somehow we get an intuition that in these cases, these transformations are not fully complete.



Rank 1



Rank 2

In [12]:

```
A = np.array([[3, 1],[0, 2]])

print(np.linalg.matrix_rank(A))
```

```
2
```

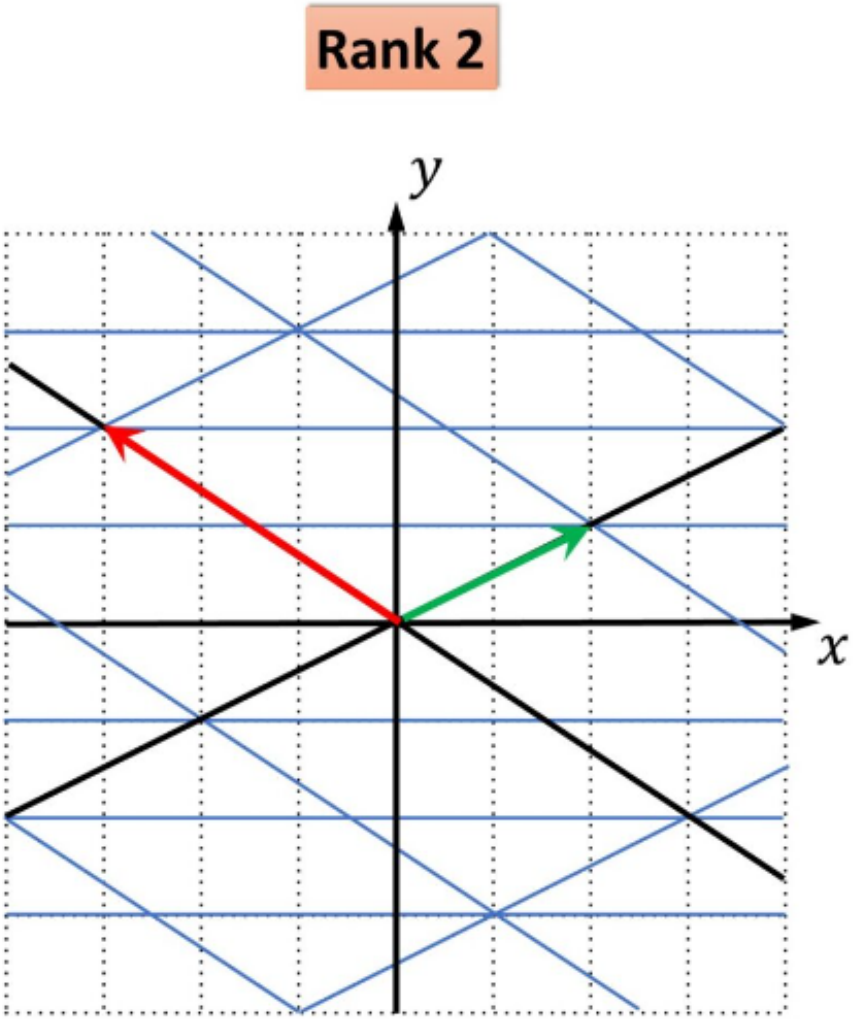
In [13]:

```
A = np.array([[3, 1, 3],[0, 2, 1],[2, 2, 3]])
print(np.linalg.matrix_rank(A))
```

```
3
```

So, one additional idea is that rank can correspond to the number of dimensions in the output. So, if we start with a 3-D vector and we end up on the plane every time, then our rank will be (2), so the number of dimensions in our output vector is (2).

In our 2-D coordinate system if we have a transformation matrix of rank (2) that's as good as we can get. Basically, our 2-D vector will be mapped into another 2-D vector and that somehow preserves our transformation completely.



If on the other hand we have a 3-D vector, this will be a rank (3) vector because there are (3) dimensions. If this is preserved we see that a determinant does not equal to zero.

Column Span

Another thing that we can define is that we would like to know for any input vector (\vec{v}) what is the so called “span”. In other words, what are the all possible solutions that one matrix (A) can provide or where our solution vectors can lie. In this case we call it a “Column space” of (A).

Another interpretation is that if we have a span of columns of two vectors (of size 2) that are linearly dependent than our column space will be just a line. So, we can only get an output vector that lies on this line.

Null space

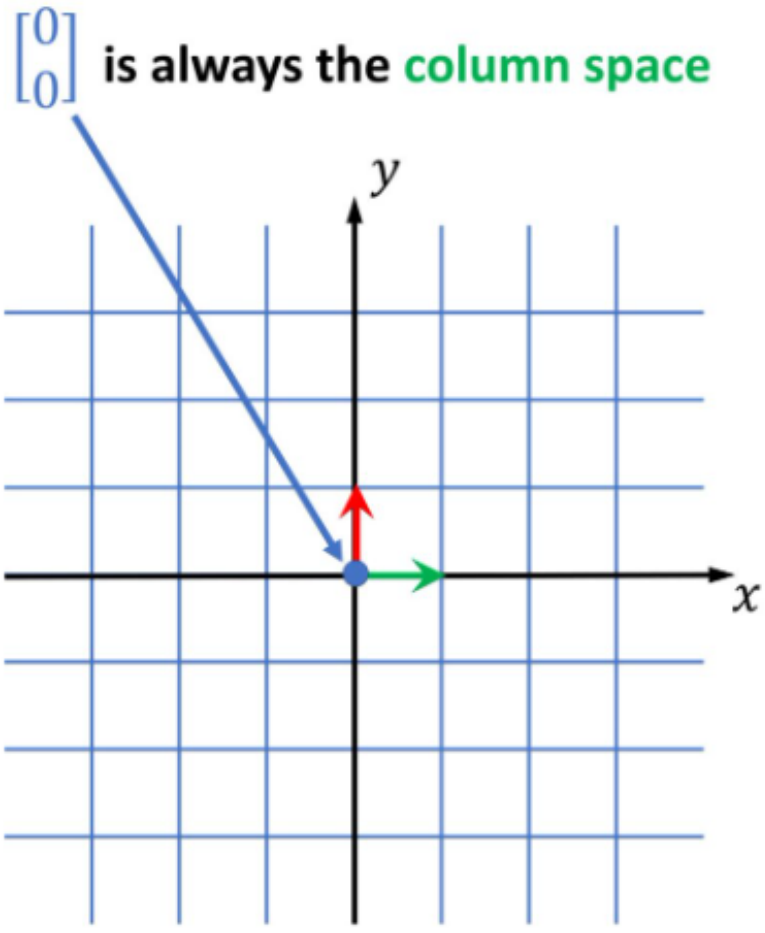
in addition, we have another concept that’s called a Null space of a matrix (A). That will be a space that gives us all vectors for which the solution (A\cdot \vec{b}= 0). One solution that will always be the case is a zero vector. This is due to the fact that linear transformations preserve the origin. So, (

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

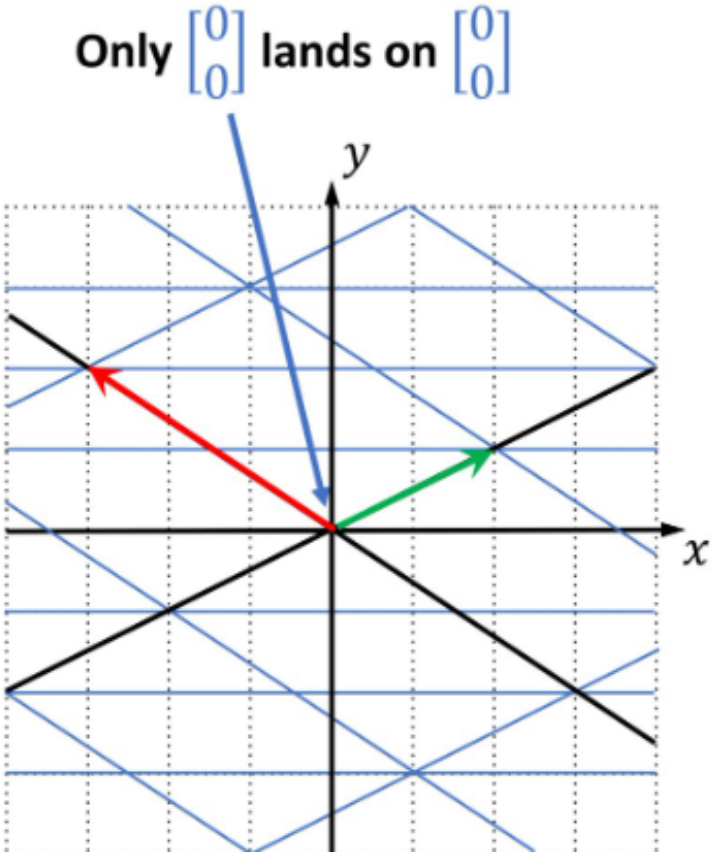
) vector will always be mapped to (

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

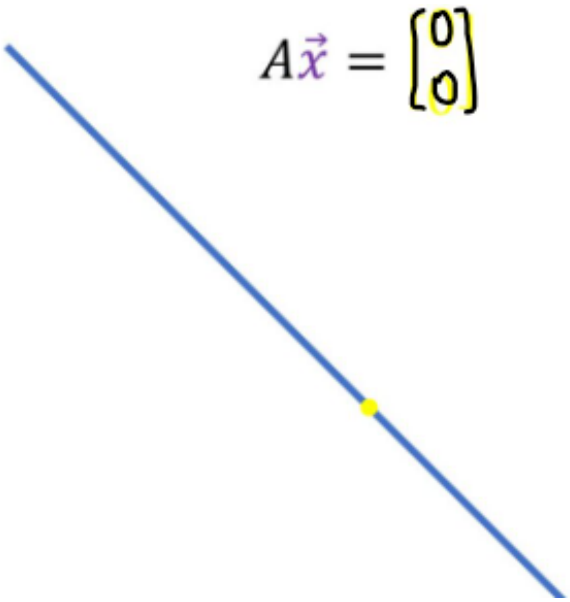
). This means that it is in the Null space, and it is always a possible solution. However, depending on the rank we can have some different interpretations abut this.



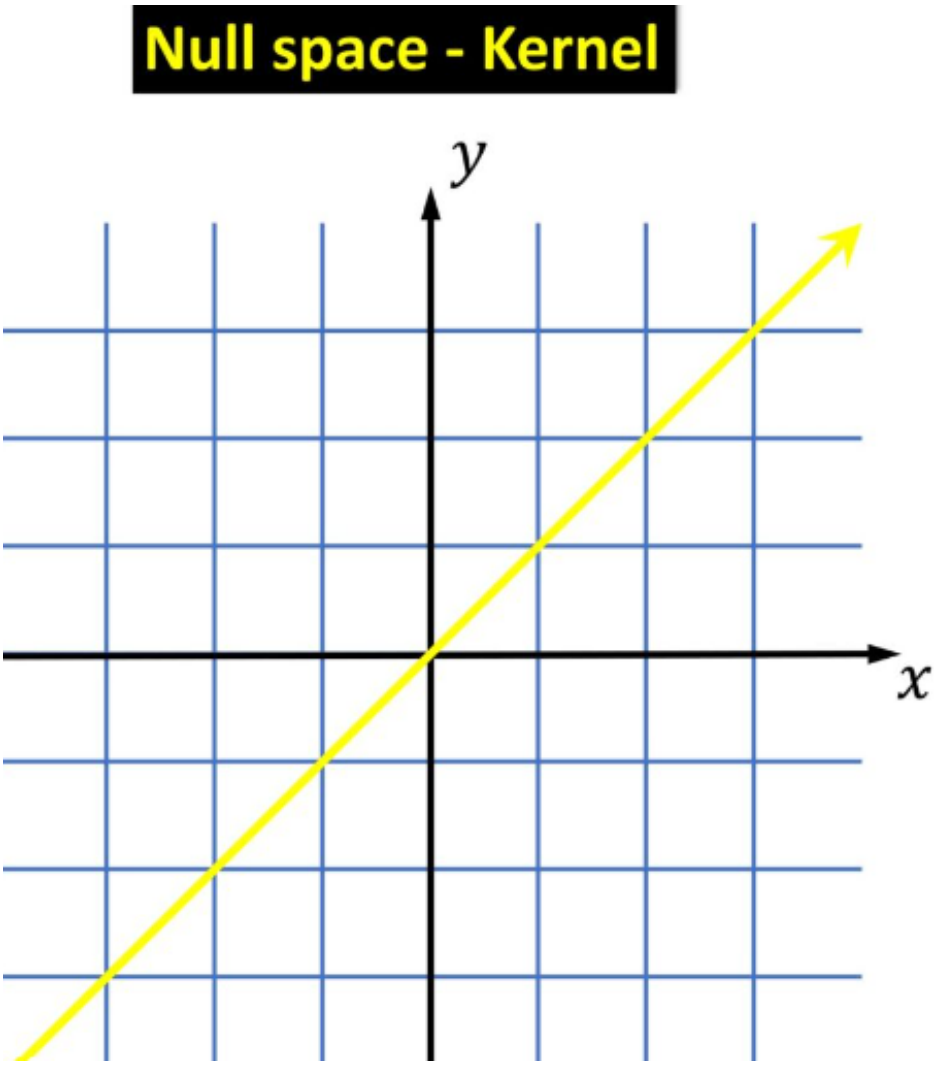
For a full rank transformation only($\begin{bmatrix}0\\0\end{bmatrix}$) will land on ($\begin{bmatrix}0\\0\end{bmatrix}$). Everything else is rotated or scaled, but then, only the center remains here as the point.



One interesting concept is that if we have a transformation that has columns which are dependent, then our 2-D space will be mapped completely into a single line.



And then we have the whole set of vectors that lie on this yellow line here which will be mapped into the ($\begin{bmatrix} 0 \\ 0 \end{bmatrix}$) vector.



In this case we can have a complete set of vectors, so not just one solution, but any vector that sits on this line will be mapped into a ($\begin{bmatrix} 0 \\ 0 \end{bmatrix}$) vector. So in this case where we do not have a full rank matrix transformation than this line will completely go into a ($\begin{bmatrix} 0 \\ 0 \end{bmatrix}$) and we have enough space that's complete line and infinite number of solutions. So, a null space is a set of all vectors that are mapped into a ($\begin{bmatrix} 0 \\ 0 \end{bmatrix}$) vector.

reference from: https://datahacker.rs/inverse-matrices-rank/ (https://datahacker.rs/inverse-matrices-rank/)