A. Introduction to Linear Algebra

A1. What is a vector?

- Vectors
  - We can see that vectors don't have to be just geometric objects in the physical order of space. They can describe ***directions*** along any sorts of axes. So we can think of vectors as just being ***lists***.
  - A Vector is:
    - A list of numbers or attributes of an object.
      - Vectors are usually viewed by computers as an ordered list of numbers which they can perform "operations" on - some operations are very natural and, as we will see, very useful!
    - Position in three dimensions of space and in one dimension of time.
      - A vector in space-time can be described using 3 dimensions of space and 1 dimension of time according to some co-ordinate system.
    - Something which moves in a space of fitting parameters.
      - As we will see, vectors can be viewed as a list of numbers which describes some optimisation problem.
      - Vectors can be thought of in a variety of different ways - some geometrically, some algebraically, some numerically. In this way, there are a lot of techniques one can use to deal with vectors.
  - The performance of a model can be quantified in a single number. One measure we can use is the Sum of Squared Residuals, SSR. Here we take all of the residuals (the difference between the measured and predicted data), square them and add them together.

$$SSR(p) = |f - g_p|^2$$

A2. Vector addition

- Vector addition
  - Let's say we have 2 vectors, $\vec{r}$ and $\vec{s}$.

$$\vec{r} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\vec{s} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

- If we want to add these two vectors together, we can simply just ***add up the components***.

$$\vec{r} + \vec{s} = \vec{s} + \vec{r} = \begin{bmatrix} 3-1 \\ 2+2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

- Vectors addition follows ***associative***:

$$\vec{r} + (\vec{s} + \vec{t}) = (\vec{r} + \vec{s}) + \vec{t}$$

A3. Vector multiplication

- Vector multiplication
  - Let's say we have a vector $\vec{r}$, and then we multiply by a scalar 2, which means 2 times the components of $\vec{r}$:

$$\vec{r} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$2\vec{r} = \begin{bmatrix} 2*3 \\ 2*2 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$$

B. Vectors and Dot Products

B1. The legnth of vectors

- The length of vectors
  - It's also called ***size***.
  - We define the size of a vector through the sums of the squares of its components (and then take the square root).

$$\vec{r} = a_i + b_j = \begin{bmatrix} a \\ b \end{bmatrix}$$

$$|r| = \sqrt{a^2 + b^2}$$

B2. The dot products

- The dot products
  - It means multiplying some vectors together.

$$\vec{r} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} r_i \\ r_j \end{bmatrix}$$

$$\vec{s} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} s_i \\ s_j \end{bmatrix}$$

- We define $\vec{r}$ dotted with $\vec{s}$ to be given by multiplying the i components together
- The dot product is just a number, a scalar number, given by multiplying the components of the vector together in turn, and adding those up

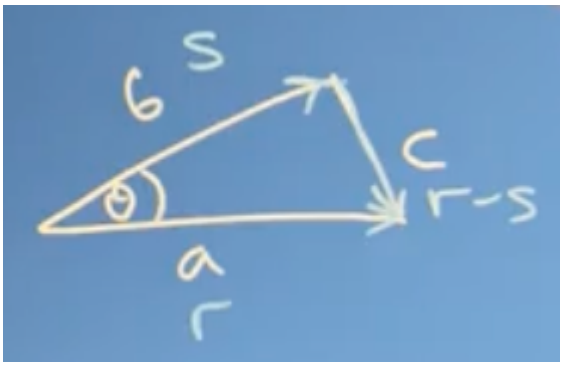
$$\vec{r} \cdot \vec{s} = r_i s_i + r_j s_j = 3 * -1 + 2 * 2 = 1 = \vec{s} \cdot \vec{r}$$

- Properties of dot produtcs:
  - ***Commutative*** means $\vec{r} \cdot \vec{s}$ is equal to $\vec{s} \cdot \vec{r}$.
  - ***Distributive*** over addition means $\vec{r} \cdot (\vec{s} + \vec{t}) = \vec{r} \cdot \vec{s} + \vec{r} \cdot \vec{t}$
  - ***Associative*** over scalar multiplication means $\vec{r} \cdot (a\vec{s}) = a(\vec{r} \cdot \vec{s})$
- If we take a vector and dot it with itself, what we get is just the sums of the squares of its components. So if we want to get the size of a vector, we can do that just by dotting the vector with itself and taking the square root.

$$\vec{r} \cdot \vec{r} = r_1^2 + r_2^2 = (\sqrt{r_1^2 + r_2^2})^2 = |r|^2$$

B3. The cosine rule and the dot products

- The cosine rule and the dot product
  - Cosine rule: $c^2 = a^2 + b^2 - 2ab\cos\theta$



$$|r - s|^2 = |r|^2 + |s|^2 - 2|r||s|\cos\theta$$

$$= (r - s) \cdot (r - s) = r \cdot r - r \cdot s - s \cdot r - s \cdot -s$$

$$= |r|^2 - 2s \cdot r + |s|^2$$

If we cancel the squares $|r|^2$ and $|s|^2$, we will get:

$$-2s \cdot r = -2|s||r|\cos\theta$$

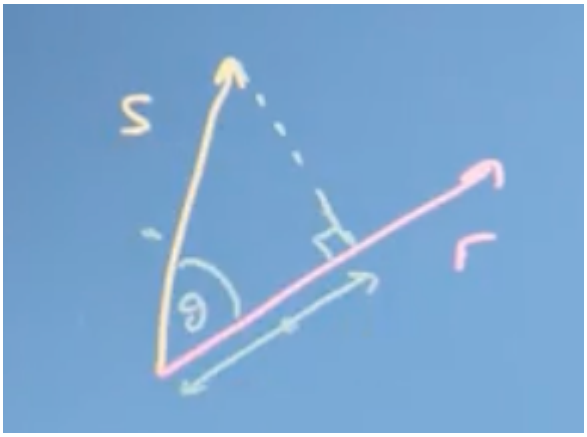
$$s \cdot r = |s||r|\cos\theta$$

- The angle between two vectors
  - If the two vectors on the other hand we're at 90 degrees to each other, also called orthogonal (or perpendicular) to each other, which means that

$\cos 90^\circ = 0$. So the dot product would also be 0.
- If the two vectors both pointed in the same direction, that means $\cos 0^\circ = 1$, then $r \cdot s = |r||s|$
- If r and s, and the angle between them is a 180 degrees $\cos 180^\circ = -1$. So $r \cdot s = -|r||s|$.

B4. Vector projection

- Vector projection
  - If we have two vectors $\vec{r}$ and $\vec{s}$, the the shadow of $\vec{s}$ on $\vec{r}$ is called the projection of $\vec{s}$ on to $\vec{r}$.



$$\cos\theta = \frac{adj}{hyp} = \frac{adj}{|s|}$$

$$r \cdot s = |r||s|\cos\theta = |r| \cdot adj = |r| \cdot projection$$

- Scalar projection
  - The dot product is also called the projection product, because it takes the projection of one vector onto another.
  - We just have to divide by the length of $\vec{r}$, and if $\vec{r}$ happened to be a unit vector or one of the vectors we used to find the space of length one, then that would be of length one and our dot $\vec{r}$ would just be the scalar projection of $\vec{r}$ onto that all that vector defining the axes.

$$\frac{r \cdot s}{|r|} = |s|\cos\theta$$

- Vector projection
  - Scalar projection gives the length of vector s along direction r, while vector projection multiplies this scalar by the unit vector of r to give the actual projected vector:

$$\text{proj}_{\mathbf{r}}(\mathbf{s}) = \frac{r \cdot s}{|r|} \cdot \frac{r}{|r|} = \left(\frac{\mathbf{r} \cdot \mathbf{s}}{|\mathbf{r}|^2}\right)\mathbf{r}$$

- For example:

$$r = \begin{bmatrix} 3 \\ 0 \end{bmatrix}, s = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$$

$$r \cdot s = 3 * 4 + 0 * 4 = 12$$

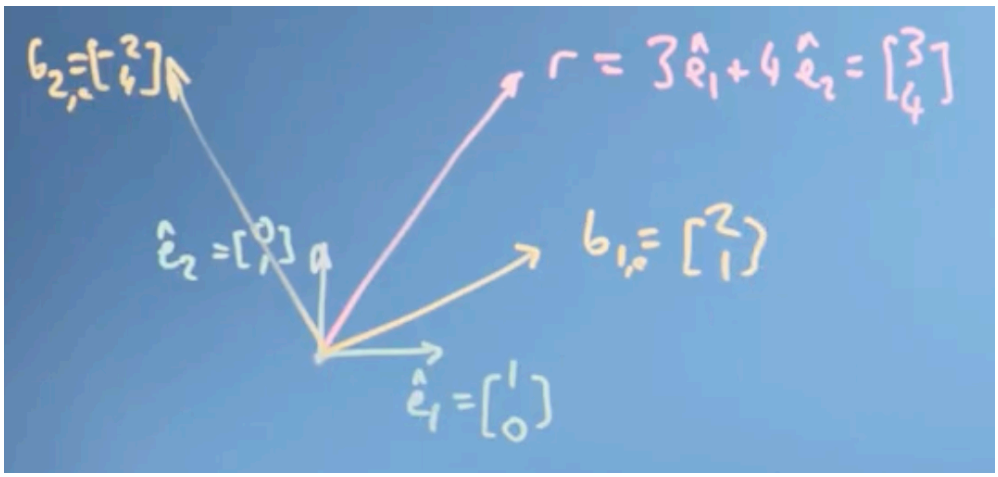
$$|r| = \sqrt{3^2 + 0^2} = 3$$

$$ScalarProjection = \frac{12}{3} = 4$$

$$VectorProjection = \frac{12}{3^2} \cdot \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

B5. Changing basis (co-ordinate systems)

- Coordinate system
  - How do we change from one coordinate system to another?



  - Let's say we have 3 vectors $\vec{r}$, $\hat{e_1}$, $\hat{e_2}$
    $\hat{e_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \hat{e_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, r_e = 3\hat{e_1} + 4\hat{e_2} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$
  - And we have another set of vectors $\vec{b_1}$, $\vec{b_2}$ $b_{1,e} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, b_{2,e} = \begin{bmatrix} -2 \\ 4 \end{bmatrix}$
  - $\vec{r}$ referred to the basis vectors $\vec{b}$ also exists, and we can use projection or dot product to find out the numbers for $\vec{r}$ in the new basis $\vec{b}$.
    - Orthogonal: if the new basis vectors $\vec{b}$, are at 90 degrees to each other, and we know what the $\vec{b}$ s are in terms of $\vec{e}$, then we can use the projection product to find $\vec{r}$ described in terms of the $\vec{b}$ s.
      - If we project $\vec{r}$ down onto $\vec{b_1}$, and project down at 90 degrees, we get a length for scalar product, and that's scalar projection is the shadow of $\vec{r}$ to $\vec{b_1}$.

    - If we take the vector projection of $\vec{r}$ onto $\vec{b_2}$ going this way, we are going to get a vector in the direction of $\vec{b_2}$ of length equal to that projection.
  - Not orthogonal: then we need ***matrices*** to do what's called a transformation of axis from the $\vec{e}$ to the $\vec{b}$ based on basis vectors.
  - How do we check the two new basis vectors are 90 degress to each other? We just need to take the dot products.

$$\cos\theta = \frac{b_1 \cdot b_2}{|b_1||b_2|}$$

$$b_1 \cdot b_2 = 2 * -2 + 1 * 4 = 0$$

The detailed calculation of changing basis:

$$\left(\frac{\mathbf{r_e} \cdot \mathbf{b_1}}{|\mathbf{b_1}|^2}\right)\mathbf{b_1} = \frac{3*2+4*1}{1+4} * b_1 = 2 * b_1 = 2 * \begin{bmatrix}2\\1\end{bmatrix} = \begin{bmatrix}4\\2\end{bmatrix}$$

$$\left(\frac{\mathbf{r_e} \cdot \mathbf{b_2}}{|\mathbf{b_2}|^2}\right)\mathbf{b_2} = \frac{3*-2+4*4}{4+16} * b_2 = \frac{1}{2} * b_1 = \frac{1}{2} * \begin{bmatrix}-2\\4\end{bmatrix} = \begin{bmatrix}-1\\2\end{bmatrix}$$

$$r_b = \begin{bmatrix}2\\\frac{1}{2}\end{bmatrix}$$

$$r_e = \begin{bmatrix}4\\2\end{bmatrix} + \begin{bmatrix}-1\\2\end{bmatrix} = \begin{bmatrix}3\\4\end{bmatrix}$$

B6. Basis, vector space, and linear independence

- Basis is a set of n vectors that:
  1. are not linear combinations of each other (linearly independent)
  2. span the space
  3. the space is then n-dimensional
- For example, let's say we have $\vec{b_1}$ and $\vec{b_2}$, and $\vec{b_3}$ as a valid third basis, it has to be:

$$\vec{b_1} \neq a_1 b_1 + a_2 b_2$$

- Which means $\vec{b_3}$ has some component out of the board so we can use $\vec{b_3}$ to give us a three-dimensional space.
- Any mapping we do from one set of basis vectors, from one coordinate system to another, keeps the vector space being a regularly spaced grid. Things might be stretched or rotated or inverted, but everything remains evenly spaced and linear combinations still work.
- (!) If the new basis vectors aren't orthogonal, then to do the change from one basis to another, we won't just be able to use the dot product, we'll need to use

matrices instead

C. Matrices

C1. Introduction to matrices

- Matrices
  - Matrices are objects that rotate and stretch vectors and they can also solve the simultaneous equation problems.
  - Let's say we have simultaneous equations like below

$$2a + 3b = 8$$

$$10a + 1b = 13$$

  - We can write them down in another way with matrices

$$\begin{pmatrix} 2 & 3 \\ 10 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 8 \\ 13 \end{pmatrix}$$

$$\begin{pmatrix} 2a + 3b \\ 10a + 1b \end{pmatrix} = \begin{pmatrix} 8 \\ 13 \end{pmatrix}$$

  - So what a set of simultaneous equations here really is? in effect, it means what vector we need in order to get a transformed product at the position 8 13 --- In order to get an output of 8 13.

- Linear algebra
- Linear algebra is linear, because it just takes input values, for example, a and b, and multiplies them by constants.
- So everything is linear. And it's algebra, it's a notation describing mathematical objects and a system of manipulating those notations.
- Linear algebra is a mathematical system for manipulating vectors in the spaces described by vectors.

C2. Matrices in linear algebra: operating on vectors

- Matrices and Basis Vectors
  - A matrix represents a linear transformation of space. The columns of a matrix show where the basis vectors ($\hat{e_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$) land after the transformation.
- Properties of Linear Transformations

- A linear transformation ensures that the geometry of the space is preserved in specific ways:
  1. Grid lines remain parallel and evenly spaced: The transformation may stretch or shear the grid, but lines stay parallel.
  2. The origin remains fixed: The vector

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

     is always mapped to

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

  3. No warping or curving: The space does not get distorted.

- The Rules of Linearity

  1. Scalar Multiplication: applying a transformation A to a scaled vector nr is equivalent to scaling the transformed vector A_r.

     $$A(nr) = n(A_r) = nr'$$

  2. Vector Addition: applying a transformation A to the sum of two vectors $(r + s)$ is equivalent to transforming each vector individually and then summing the results.

     $$A(r + s) = A_r + A_s$$

- The Key Insight: transformation by basis vectors

  - Any vector v in a space can be represented as a scalar sum of the basis vectors $(v = n\hat{e_1} + m\hat{e_2})$. Due to the rules of linearity, the transformation of v is simply the sum of the transformed basis vectors:

    $$A(n\hat{e_1} + m\hat{e_2}) = n(A\hat{e_1}) + m(A\hat{e_2}) = n\hat{e_1'} + m\hat{e_2'}$$

- A linear transformation is entirely determined by where it maps the basis vectors.

- Example:

  - Given a matrix $A = \begin{bmatrix} 2 & 3 \\ 10 & 1 \end{bmatrix}$ and a vector $V = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$.

    1. Standard matrix multiplication:
       $A_v = \begin{bmatrix} 2 & 3 \\ 10 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} (2 \times 3) + (3 \times 2) \\ (10 \times 3) + (1 \times 2) \end{bmatrix} = \begin{bmatrix} 12 \\ 32 \end{bmatrix}$
    2. Transformation via basis vectors:
    - the vector v can be written as $3\hat{e_1} + 2\hat{e_2}$.
    - The columns of A are the transformed basis vectors:
      $\hat{e_1'} = A\hat{e_1} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$ $\hat{e_2'} = A\hat{e_2} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$

    - By linearity, $A_v = 3(A\hat{e_1}) + 2(A\hat{e_2})$:

$$3\begin{bmatrix} 2 \\ 10 \end{bmatrix} + 2\begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 30 \end{bmatrix} + \begin{bmatrix} 6 \\ 2 \end{bmatrix} = \begin{bmatrix} 12 \\ 32 \end{bmatrix}$$

- This confirms that the transformation result is consistent, demonstrating that we only need to understand what the matrix does to the basis vectors.

- Types of matrix transformation
    1. Idendity matrix ($\mathbb{I}$ or 1 or I): a matrix that doesn't change anything: a matrix is just composed of the basis vectors of the space. For example: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$
    2. Scaling matrix: a matrix that transforms vectors by changing their size (either stretching or shrinking) along one or more axes.
        - For example, the matrix below will scale the x axis here by a factor of 3, and the y axis is going to scale by a multiple of 2 (when we multiply the matrix with the vectors). $\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x \\ 2y \end{bmatrix}$
    3. Reflection matrix: a matrix that flips (or "reflects") vectors or coordinate axes across a line or a plane, acting like a mirror. The columns of the matrix show where the basis vectors land after the reflection.
        - Reflection across the line y=x (diagonal mirror): this matrix swaps the x and y components of a vector. It's like placing a mirror along the y=x line (a 45-degree angle). $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x \end{bmatrix}$
        - Reflection across the line y=−x (opposite diagonal mirror): this matrix swaps the x and y components and negates both. It's like placing a mirror along the y=−x line. $\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -y \\ -x \end{bmatrix}$
        - Reflection across the y-axis (vertical mirror): this matrix negates the x-component while keeping the y-component the same. It's like placing a mirror along the vertical y-axis. $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix}$
        - Reflection across the x-axis (horizontal mirror): this matrix negates the y-compoent while keeping the x-component the same. It's basically placing a mirror alonge the horizontal x-axis. $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ -y \end{bmatrix}$
    4. Inversion matrix: an inversion matrix flips a vector or object across the origin (the point (0,0) in 2D or (0,0,0) in 3D). This effectively reflects every point through the origin, transforming both axes. $\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -x \\ -y \end{bmatrix}$
        - This transformation essentially rotates the object 180 degrees around the origin, which is equivalent to reflecting it across both the X and Y axes simultaneously.
    5. Shear matrix: a shear matrix skews or "pushes over" vectors or shapes along a particular axis. It transforms a square into a parallelogram without

changing its area (in 2D) or volume (in 3D).

- Example: shearing the y-axis along the x-axis. Suppose we want to keep the $\hat{e_1}$ vector (x-axis unit vector, $\begin{bmatrix}1\\0\end{bmatrix}$) in place, but move the $\hat{e_2}$ vector (y-axis unit vector, $\begin{bmatrix}0\\1\end{bmatrix}$) to ($\begin{bmatrix}1\\1\end{bmatrix}$)
- So the shear matrix for this transformation is:
  $\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}x+y\\y\end{bmatrix}$

6. Rotation matrix: a rotation matrix rotates vectors or objects around a point (in 2D, usually the origin) or an axis (in 3D) by a specific angle.
   - Example: 90-degree counter-clockwise rotation. The original $\hat{e_1}$ vector (x-axis unit vector, $\begin{bmatrix}1\\0\end{bmatrix}$) after a 90-degree counter-clockwise rotation becomes ($\begin{bmatrix}0\\1\end{bmatrix}$). The original $\hat{e_2}$ vector (x-axis unit vector, $\begin{bmatrix}0\\1\end{bmatrix}$) after a 90-degree counter-clockwise rotation becomes ($\begin{bmatrix}-1\\0\end{bmatrix}$).
   - Therefore, the 90-degree counter-clockwise rotation matrix is:
     $\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}-y\\x\end{bmatrix}$
   - General 2D rotation matrix (for angle θ counter-clockwise): For a general counter-clockwise rotation by an angle θ in 2D, the rotation matrix is: $\begin{bmatrix}\cos\theta & -\sin\theta\\\sin\theta & \cos\theta\end{bmatrix}$
   - Note that a positive θ denotes a counter-clockwise rotation, and a negative θ denotes a clockwise rotation. For example, 90-degree clockwise rotation (e.g., $\theta = -90^\circ$ for a 90-degree clockwise rotation), the sine and cosine functions will correctly handle the direction. For example, for -90°(90-degree clockwise):
     $\begin{bmatrix}\cos(-90^\circ) & -\sin(-90^\circ)\\\sin(-90^\circ) & \cos(-90^\circ)\end{bmatrix} = \begin{bmatrix}0 & -(-1)\\-1 & 0\end{bmatrix} = \begin{bmatrix}0 & 1\\-1 & 0\end{bmatrix}$

C3. Matrix composition (combining transformations)

- Importance of transformation composition
  - Any complex shape alteration (e.g., transforming a face in an image) can be achieved by a sequence of basic linear transformations (like rotations, shears, inversions, reflections, and scaling). When we apply multiple transformations to a vector r sequentially (e.g., first $A1$ then $A2$ to the result), this is called a composition of transformations.
    - In matrix operations, this composition is achieved through matrix multiplication:

$$A_2(A_1 r) = (A_2 A_1) r$$

- This means the vector r is first acted upon by the rightmost matrix A_1,and then the result is acted upon by the leftmost matrix A_2. Thus, the order of application is from right to left.

- Rotation followed by reflection VS reflection followed by rotation (comparing orders)

  - Basic vectors:

$$\hat{e_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\hat{e_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

- Transformation A1: 90-degree counter-clockwise rotation

$$\hat{e_1} \to \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \hat{e_2} \to \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

- Transformation A2: reflection across the y-axis (vertical mirror)

$$\hat{e_1} \to \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \hat{e_2} \to \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Case 1: apply A1 (rotate) first, then A2 (reflect)—this corresponds to calculating $A_2 A_1$.

$$A_2 A_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

  Performing the matrix multiplication:

$$= \begin{bmatrix} (-1)(0) + (0)(1) & (-1)(-1) + (0)(0) \\ (0)(0) + (1)(1) & (0)(-1) + (1)(0) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

  Result: This composite matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ is a reflection across the line ***y=x***.

- Case 2: apply A2 (reflect) first, then A1 (rotate)—this corresponds to calculating $A_1 A_2$.

$$A_1 A_2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

  Performing the matrix multiplication:

$$= \begin{bmatrix} (0)(-1)+(-1)(0) & (0)(0)+(-1)(1) \\ (1)(-1)+(0)(0) & (1)(0)+(0)(1) \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$

Result: This composite matrix $\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$ is a reflection across the line ***y=−x***.

- Important Properties of Matrix Multiplication From the examples above, we can derive two key properties of matrix multiplication:
  - ***"Not" commutative***: the order of matrix multiplication matters; generally, $A_2A_1 \neq A_1A_2$. This means the ***sequence in which transformations are applied affects the final outcome***. Geometrically, rotating then reflecting is usually different from reflecting then rotating.
  - Associative: matrix multiplication is associative, $(A_3A_2)A_1 = A_3(A_2A_1)$. This means if we have three or more matrices multiplied together, we can group them in different ways for calculation, and the final result will be the same, as long as their relative order remains unchanged.

C4. Matrix inverses

Solving systems of linear equations & the inverse matrix

- This section introduces a method to solve systems of linear equations, like the "apples and bananas" problem, and connects it to the concept of a matrix inverse.

1. The Apples and bananas problem revisited: a system of linear equations can be represented in matrix form:
   - 2 apples + 3 bananas = 8 euros
   - 10 apples + 1 banana = 13 euros

This can be written as:

$$A_r = s$$

Where:

$$A = \begin{bmatrix} 2 & 3 \\ 10 & 1 \end{bmatrix}$$

(the coefficient matrix)

$$r = \begin{bmatrix} a \\ b \end{bmatrix}$$

(the vector of unknowns / apple and banana prices)

$$s = \begin{bmatrix} 8 \\ 13 \end{bmatrix}$$

(the output vector / total costs) The goal is to find the values in vector r.

2. Introducing the inverse matrix (A^{-1}): the inverse of a matrix A, denoted as (A^{-1}), is a special matrix that, when multiplied by A, yields the identity matrix (I).

$$A^{-1}A = I$$

How the inverse solves the problem: if we have the equation $A_r = s$, we can multiply both sides by A^{-1} from the left:

$$A^{-1}(A\mathbf{r}) = A^{-1}\mathbf{s}$$

Since $A^{-1}A = \mathbf{I}$, the equation becomes:

$$\mathbf{Ir} = A^{-1}\mathbf{s}$$

Which simplifies to:

$$\mathbf{r} = A^{-1}\mathbf{s}$$

Therefore, if we can find the inverse of matrix A, we can directly solve for the unknown vector r.

3. Solving linear equations without explicitly computing the inverse (Gaussian elimination): while the inverse matrix offers a general solution, specific systems of equations can often be solved efficiently using elimination and back substitution, a process similar to Gaussian Elimination. This method transforms the system's matrix into a simpler form.

- Example: apples, bananas, and carrots problem: consider a 3x3 system:

  1 apple + 1 banana + 3 carrots = 15 euros

  1 apple + 2 bananas + 4 carrots = 21 euros

  1 apple + 1 banana + 2 carrots = 13 euros

- Matrix form:

$$\begin{bmatrix} 1 & 1 & 3 \\ 1 & 2 & 4 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 15 \\ 21 \\ 13 \end{bmatrix}$$

- Step-by-step solution:

  1. Elimination (forward elimination): the goal is to make all entries below the main diagonal zero, creating an upper triangular matrix (also known as row echelon form). This involves subtracting multiples of one row from another.
  - Row 2 - row 1

$$([1 \quad 2 \quad 4] - [1 \quad 1 \quad 3]) = [0 \quad 1 \quad 1]$$

$$(21 - 15) = 6$$

- row 3 - row 1

$$([1 \quad 1 \quad 2] - [1 \quad 1 \quad 3]) = [0 \quad 0 \quad -1]$$

$$(13 - 15) = -2$$

- The system transforms to

$$\begin{bmatrix} 1 & 1 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 15 \\ 6 \\ -2 \end{bmatrix}$$

- From the third row, we directly get $-c = -2$, so $c = 2$.

2. Back substitution: now, use the known value of c to find b, and then use b and c to find a.

- Multiply row 3 by -1: (to make leading diagonal 1)

$$[0 \quad 0 \quad 1], 2$$

- Eliminate c from row 1 and row 2 using the new row3
- Row 1 - 3* row 3:

$$([1 \quad 1 \quad 3] - 3[0 \quad 0 \quad 1]) = [1 \quad 1 \quad 0]$$

$$(15 - 6) = 9$$

- Row 2 - 1* row 3:

$$([0 \quad 1 \quad 1] - [0 \quad 0 \quad 1]) = [0 \quad 1 \quad 0]$$

$$(6 - 2) = 4$$

- The system now is:

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \\ 2 \end{bmatrix}$$

- From the second row, we get $b = 4$.
- Eliminate b from row 1 using row 2:
  - row 1 - row 2:

$$([1 \quad 1 \quad 0] - [0 \quad 1 \quad 0]) = [1 \quad 0 \quad 0]$$

$$(9 - 4) = 5$$

- The final system becomes:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 2 \end{bmatrix}$$

3. Solution: $a = 5, b = 4, c = 2$. This method (Gaussian elimination) is computationally efficient for solving specific systems. However, it only provides the solution for a given output vector s.

- Finding the inverse matrix using Gaussian elimination
  - This section explains how to find the inverse of a matrix (A^{-1}), which provides a general solution to linear equation systems, regardless of the output vector. The method involves extending the elimination and back-substitution process (Gaussian Elimination) to solve for all inverse components simultaneously.

    1. The inverse matrix revisited: recall that for a matrix A, its inverse A^{-1} is defined such that when they are multiplied together, they yield the identity matrix (I):

       $$A^{-1}A = I$$

       The inverse matrix is special because this relationship holds true regardless of the multiplication order:

       $$AA^{-1} = A^{-1}A = I$$

    If we have a system of linear equations in the form $Ar = s$, where r is the vector of unknowns, we can solve for r by multiplying both sides by A^{-1} from the left: $A^{-1}(A\mathbf{r}) = A^{-1}\mathbf{s}\,\mathbf{Ir} = A^{-1}\mathbf{s}\,\mathbf{r} = A^{-1}\mathbf{s}$

    This means finding A^{-1} allows us to solve for r for any output vector s, providing a general solution.

    2. Finding the inverse using simultaneous elimination and back-substitution: instead of solving $Ar = s$ for each different s (which would involve solving the system multiple times), we can find the inverse matrix A^{-1} by extending the elimination process.

Consider a 3×3 matrix A and its inverse $B = A^{-1}$:

$$AB = I$$

$$\text{Let } A = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 2 & 4 \\ 1 & 1 & 2 \end{bmatrix} \text{ and } B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}. \text{ The indentity matrix } I =$$

We can think of $AB = I$ as three separate systems of equations:

- $A \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ (Solve for the first column of B)
- $A \begin{bmatrix} b_{12} \\ b_{22} \\ b_{32} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ (Solve for the second column of B)

- $A \begin{bmatrix} b_{13} \\ b_{23} \\ b_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ (Solve for the third column of B)

The key insight is that we can solve all three systems simultaneously by applying the row operations to an augmented matrix formed by appending the identity matrix to A: $[A|\mathbf{I}]$ The goal is to transform the left side (A) into the identity matrix ($\mathbf{I}$) using row operations. The same row operations applied to the right side ($\mathbf{I}$) will transform it into A^{-1}.

$$[A|\mathbf{I}] \xrightarrow{\text{Row Operations}} [\mathbf{I}|A^{-1}]$$

Step-by-step example (using the given matrix A):

Start with the augmented matrix:

$$\left[\begin{array}{ccc|ccc} 1 & 1 & 3 & 1 & 0 & 0 \\ 1 & 2 & 4 & 0 & 1 & 0 \\ 1 & 1 & 2 & 0 & 0 & 1 \end{array}\right]$$

- Forward elimination (to create an upper triangular form on the left):
  - $R_2 \leftarrow R_2 - R_1$
  - $R_3 \leftarrow R_3 - R_1$

$$\left[\begin{array}{ccc|ccc} 1 & 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & -1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 0 & 1 \end{array}\right]$$

- Make leading diagonal entries 1 (by multiplying rows):
  - $R_3 \leftarrow R_3 \times -1$

$$\left[\begin{array}{ccc|ccc} 1 & 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & -1 \end{array}\right]$$

- Back substitution (to create zeros above the leading diagonal on the left):
  - $R_2 \leftarrow R_2 - R_3$
  - $R_1 \leftarrow R_1 - 3R_3$

$$\left[\begin{array}{ccc|ccc} 1 & 1 & 0 & -2 & 0 & 3 \\ 0 & 1 & 0 & -2 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & -1 \end{array}\right]$$

- Complete the identity matrix on the left:
  - $R_1 \leftarrow R_1 - R_2$

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & -1 & 2 \\ 0 & 1 & 0 & -2 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & -1 \end{array}\right]$$

- The left side is now the identity matrix. The matrix on the right side is the inverse of A. So, the inverse matrix $A^{-1}$is:

$$A^{-1} = \begin{bmatrix} 0 & -1 & 2 \\ -2 & 1 & 1 \\ 1 & 0 & -1 \end{bmatrix}$$

This process of applying row operations to the augmented matrix $[A \mid I]$ to transform A into I (and I into A^{-1}) is a computationally efficient way to find the inverse matrix. While simple systems can be solved by direct substitution, finding the inverse allows for a general solution for any output vector s. This method, a form of Gaussian Elimination, is fundamental to how computers solve linear equation problems and is often implemented in numerical libraries (e.g., inv(A) in software) that automatically select the most efficient decomposition methods.

C5. Special matrices (determinants and inverses)

- This section introduces the determinant of a matrix, a fundamental property that reveals how a linear transformation scales space. We'll explore its connection to linear independence of basis vectors and the existence of a matrix inverse.
- Determinant: the determinant of a transformation matrix quantifies the ***scaling factor*** of the space it transforms.
  - Simple case: diagonal matrix

    $$A = \begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix}$$

    This matrix scales the original basis vectors $\hat{e_1}$ to ($[\,a, 0\,]^T$) and $\hat{e_2}$ to ($[\,0, d\,]^T$). If the original space was a 1x1 unit square, this transformation scales it by a factor of a in one direction and d in the other. The new area of the transformed unit square is $a \times d = ad$. This value, ad, is the determinant of this transformation matrix. All areas in the space are scaled by this factor.
  - Sheared transformation (parallelogram area)

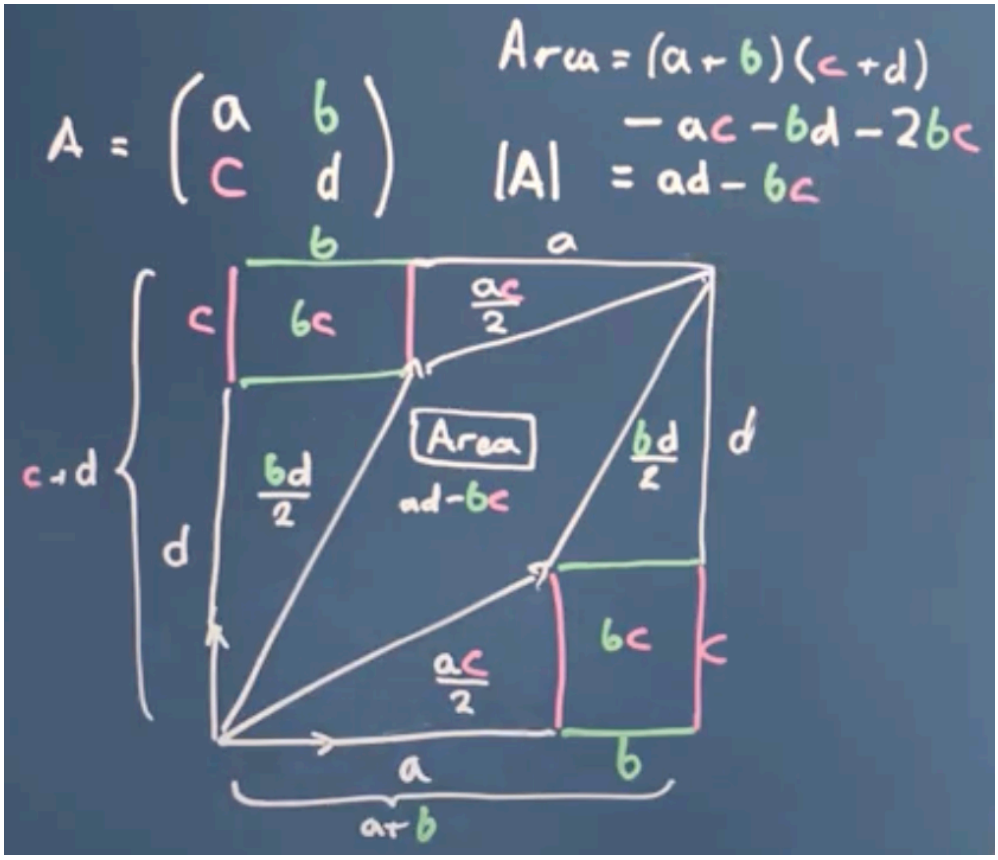
    $$A = \begin{bmatrix} a & b \\ 0 & d \end{bmatrix}$$

    Here, $\hat{e_1}$ is scaled to ($[\,a, 0\,]^T$), but $\hat{e_2}$ is transformed to ($[\,b, d\,]^T$). This transforms the original unit square into a ***parallelogram***. Despite the shear, the area of this parallelogram is still its base times its perpendicular height. The base remains a (along the x-axis) and the perpendicular height is d (the

y-component of the transformed $\hat{e_2}$). Thus, the area (and the determinant) is still ad.

- General 2x2 matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$



This transforms the original unit square into a parallelogram. The area of this parallelogram is found to be $ad - bc$. We denote the operation of finding this area with vertical lines, like $|A|$, and call it the determinant of A.
$|A| = det(A) = ad - bc$.

- Determinants and the inverse matrix
  - Recall that the inverse of a 2x2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is given by:

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

  - Notice that the term $ad - bc$ appears in the denominator. This is the determinant! This means:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

  - When we multiply A by this inverse, we get the identity matrix:

$$A^{-1}A = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \frac{1}{\det(A)} \begin{bmatrix} da - bc & db - bd \\ -ca + ac & -cb + ad \end{bmatrix}$$

    - The determinant essentially represents the scaling factor that needs to be "undone" by the inverse to return the space to its original scale (identity).
- The special case: determinant is zero
    - A crucial aspect of the determinant is what happens when it is zero ($\det(A) = 0$).
        - Linear dependence of basis vectors
            - If $\det(A) = 0$, it implies that the transformed basis vectors are linearly dependent. For example, consider the matrix:

$$A = \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}$$

            - Here,$\det(A) = (1)(2) - (2)(1) = 2 - 2 = 0$.
            - This matrix transforms $\hat{e_1}$ to $[1, 1]^T$ and $\hat{e_2}$ to $[2, 2]^T$.
            - Notice that the transformed $\hat{e_2}$ is simply a multiple of the transformed $\hat{e_1}$ ($2 \times [1, 1]^T = [2, 2]$). They lie on the same line.
            - What this matrix does is collapse the entire 2D space onto a 1D line. The original area (the unit square) is squashed into a line, which has zero area. This geometrically explains why the determinant is zero.
- Implications for hhigher dimensions
    - In 3D space, if a 3x3 matrix has a determinant of zero, it means the transformation collapses the 3D space into a 2D plane or even a 1D line. In either case, the enclosed volume becomes zero, hence the determinant is zero.
- Inability to solve systems of equations / no inverse
    - If $\det(A) = 0$, then:
        1. The matrix has no inverse. We cannot calculate $\frac{1}{\det(A)}$ because we cannot divide by zero.
        2. We cannot uniquely solve the system of simultaneous linear equations.

           Represented by $Ar = s$. This is because the transformation has lost information by collapsing dimensions. If we transform a 2D space into a line, we cannot "undo" that transformation to get back to a unique 2D point because the information about the second dimension has been lost.
- Determinants and echelon form
    - Consider a system of simultaneous equations whose coefficient matrix has linearly dependent columns (or rows).

      For example:

        - Row 3 = Row 1 + Row 2
        - Column 3 = 2 * Column 1 + Column 2

This means the transformation matrix does not describe independent basis vectors; it collapses 3D space into a 2D space. When we try to reduce such a matrix to row echelon form using Gaussian elimination, we will encounter a row of all zeros.

If we have a matrix that leads to: $\begin{bmatrix} 1 & 1 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 15 \\ 6 \\ 0 \end{bmatrix}$

The last row implies $0c = 0$. This equation is true for any value of c, meaning there are infinitely many solutions for c, and thus for the system. This indicates that we don't have enough independent equations (information) to find a unique solution. This directly correlates with the determinant being zero.

Python code: Echelon form transformation and Singularity

In [173…

```
# Echelon form funciton.
import numpy as np

# Our function will go through the matrix replacing each row in order tur
# If at any point it fails because it can't put a 1 in the leading diagon
# we will return the value True, otherwise, we will return False.
# There is no need to edit this function.
def isSingular(A) :
    B = np.array(A, dtype=np.float64) # Make B as a copy of A, since we'r
    try:
        fixRowZero(B)
        fixRowOne(B)
        fixRowTwo(B)
        fixRowThree(B)
    except MatrixIsSingular:
        return True
    return False

# This next line defines our error flag. For when things go wrong if the
# There is no need to edit this line.
class MatrixIsSingular(Exception): pass

# For Row Zero, all we require is the first element is equal to 1.
# We'll divide the row by the value of A[0, 0].
# This will get us in trouble though if A[0, 0] equals 0, so first we'll
# and if this is true, we'll add one of the lower rows to the first one b
# We'll repeat the test going down each lower row until we can do the div
# There is no need to edit this function.
def fixRowZero(A) :
    if A[0,0] == 0 :
        A[0] = A[0] + A[1]
    if A[0,0] == 0 :
        A[0] = A[0] + A[2]
    if A[0,0] == 0 :
        A[0] = A[0] + A[3]
    if A[0,0] == 0 :
```

```
        raise MatrixIsSingular()
    A[0] = A[0] / A[0,0]
    return A

# First we'll set the sub-diagonal elements to zero, i.e. A[1,0].
# Next we want the diagonal element to be equal to one.
# We'll divide the row by the value of A[1, 1].
# Again, we need to test if this is zero.
# If so, we'll add a lower row and repeat setting the sub-diagonal elemen
# There is no need to edit this function.
def fixRowOne(A) :
    A[1] = A[1] - A[1,0] * A[0]
    if A[1,1] == 0 :
        A[1] = A[1] + A[2]
        A[1] = A[1] - A[1,0] * A[0]
    if A[1,1] == 0 :
        A[1] = A[1] + A[3]
        A[1] = A[1] - A[1,0] * A[0]
    if A[1,1] == 0 :
        raise MatrixIsSingular()
    A[1] = A[1] / A[1,1]
    return A

# This is the first function that you should complete.
# Follow the instructions inside the function at each comment.
def fixRowTwo(A) :
    # Insert code below to set the sub-diagonal elements of row two to ze
    A[2] = A[2] - A[2,0] * A[0]
    A[2] = A[2] - A[2,1] * A[1]
    # Next we'll test that the diagonal element is not zero.
    if A[2,2] == 0 :
        # Insert code below that adds a lower row to row 2.
        A[2] = A[2]+A[3]
        # Now repeat your code which sets the sub-diagonal elements to ze
        A[2] = A[2] - A[2,0] * A[0]
        A[2] = A[2] - A[2,1] * A[1]
    if A[2,2] == 0 :
        raise MatrixIsSingular()
    # Finally set the diagonal element to one by dividing the whole row b
    A[2] = A[2] / A[2,2]
    return A

# You should also complete this function
# Follow the instructions inside the function at each comment.
def fixRowThree(A) :
    # Insert code below to set the sub-diagonal elements of row three to
    A[3] = A[3] - A[3,0] * A[0]
    A[3] = A[3] - A[3,1] * A[1]
    A[3] = A[3] - A[3,2] * A[2]
    # Complete the if statement to test if the diagonal element is zero.
    if A[3,3] == 0:
        raise MatrixIsSingular()
    # Transform the row to set the diagonal element to one.
    A[3] = A[3] / A[3,3]
    return A

# Testing dataset 1
A = np.array([
        [2, 0, 0, 0],
        [0, 3, 0, 0],
```

```
        [0, 0, 4, 4],
        [0, 0, 5, 5]
    ], dtype=np.float64)
print('Testing dataset:',isSingular(A))

# Testing dataset 2
B = np.array([
        [0, 7, -5, 3],
        [2, 8, 0, 4],
        [3, 12, 0, 5],
        [1, 3, 1, 3]
    ], dtype=np.float64)

# Echelon form
print('\nFixed row zero\n', fixRowZero(B))
print('\nFixed row one\n', fixRowOne(B))
print('\nFixed row two\n', fixRowTwo(B))
print('\nFixed row three\n', fixRowThree(B))

print('\nFinal result\n', B)
```

```
Testing dataset: True

Fixed row zero
 [[ 1.   7.5 -2.5  3.5]
 [ 2.   8.   0.   4. ]
 [ 3.  12.   0.   5. ]
 [ 1.   3.   1.   3. ]]

Fixed row one
 [[ 1.          7.         -2.          3.        ]
 [-0.          1.         -0.71428571  0.42857143]
 [ 3.         12.          0.          5.        ]
 [ 1.          3.          1.          3.        ]]

Fixed row two
 [[ 1.          7.         -2.          3.        ]
 [-0.          1.         -0.71428571  0.42857143]
 [ 0.          0.          1.          1.5       ]
 [ 1.          3.          1.          3.        ]]

Fixed row three
 [[ 1.          7.         -2.          3.        ]
 [-0.          1.         -0.71428571  0.42857143]
 [ 0.          0.          1.          1.5       ]
 [ 0.          0.          0.          1.        ]]

Final result
 [[ 1.          7.         -2.          3.        ]
 [-0.          1.         -0.71428571  0.42857143]
 [ 0.          0.          1.          1.5       ]
 [ 0.          0.          0.          1.        ]]
```

D. Matrices make linear mappings

D1. Einstein summation convention and the symmetry of the dot product

- Einstein's summation convention
  - Einstein's summation convention is a shorthand notation for matrix and vector operations, particularly useful for programming and theoretical proofs. It simplifies the way we write sums by ***dropping*** the summation symbol $\sum$.
  - The standard way to define matrix multiplication for a product matrix $C = AB$ is to write each element as:

$$C_{ik} = \sum A_{ij}B_{jk}$$

  - This formula states that to find the element in row i and column k of matrix C, we sum the ***products of elements*** from row i of matrix A and column k of matrix B.
  - The summation convention simplifies this by removing the summation symbol. If an index (like j in this case) appears twice in a term, it implies a summation over all possible values of that index.

$$C_{ik} = A_{ij}B_{jk}$$

  - This compact notation makes it clear what operations are needed for programming: we can use nested loops to iterate through all i, j, and k indices to compute the product matrix.
- Non-square matrices
  - The summation convention makes it clear that we can multiply matrices of different shapes, as long as the "inner" dimensions match. For a matrix A with dimensions $m \times n$ and a matrix B with dimensions $n \times p$, the product matrix $C = AB$ will have dimensions $m \times p$. The repeated index j in $A_{ij}B_{jk}$ confirms that the number of columns in A must equal the number of rows in B.

Python code: Using nested loops to compute the product matrix

In [174…

```python
# Python example of using nested loops to compute the product matrix
import numpy as np

# define 2 matrices
A = np.array([[1, 2],
              [3, 4]])

B = np.array([[5, 6],
              [7, 8]])

# init the matrix C, 2x2, and fill with 0s
C = np.zeros((2, 2), dtype=int)

# Einstein summation convention nested 3 loops
# i: Iterate the elements of row A
for i in range(A.shape[0]):
```

```
    # Iterate the elements of column B
    for k in range(B.shape[1]):
        # Get the product of summation
        for j in range(A.shape[1]):
            C[i, k] += A[i, j] * B[j, k]

print("Matrix A:\n", A)
print("\nMatrix B:\n", B)
print("\nProduct matrix C (nested loops):\n", C)

# Using Numpy to verify the calculation
C_numpy = A @ B
print("\nProduct matrix C (using NumPy ):\n", C_numpy)
```

```
Matrix A:
 [[1 2]
 [3 4]]

Matrix B:
 [[5 6]
 [7 8]]

Product matrix C (nested loops):
 [[19 22]
 [43 50]]

Product matrix C (using NumPy ):
 [[19 22]
 [43 50]]
```

D2. Dot product revisited with the summation convention

- The summation convention also provides a new perspective on the dot product.
  - For example, dot product of two vectors u and v can be written as:

$$u \cdot v = \sum_i u_i v_i$$

  - Using the summation convention, this becomes even simpler:

$$u \cdot v = u_i v_i$$

  - This notation reveals that the dot product is essentially a form of matrix multiplication. If we treat a column vector u as a $1 \times n$ row matrix and a column vector v as an $n \times 1$ column matrix, their product is:

$$u^T v = \begin{bmatrix} u_1, u_2, \ldots, u_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ v_n \end{bmatrix} = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$$

  - This shows that the dot product is equivalent to multiplying a row vector by a column vector. This elegant connection highlights the underlying relationship between these two fundamental operations.

- Dot product as projection
  - The dot product is also geometrically linked to vector projection. The dot product $u \cdot v$ gives us a scalar that is the length of the projection of vector u onto vector v, scaled by the length of v.
    - This relationship is symmetric: the projection of u onto v has the same length as the projection of v onto u.
    - This is why the dot product is commutative ($u \cdot v = v \cdot u$).

D3. Revisit: Transforming Vectors Between Different Bases

- A vector's components depend entirely on the basis vectors of the coordinate system it's defined in. A transformation matrix acts as a bridge, allowing us to convert a vector from one basis to another.
- (1) From 'Panda's World' to 'Our World':
  - Let's say we have two coordinate systems: our standard orthonormal basis (our world) and Panda's non-standard basis (Panda's world).
    - **Panda's basis vectors** in our coordinate system are the vectors that define Panda's axes.
    - **The transformation matrix**, let's call it B, is constructed by making Panda's basis vectors the columns of the matrix.
  - To convert a vector v_{Panda} from Panda's world to its representation in our world, v_{me}, we perform a simple matrix multiplication:

$$v_{me} = Bv_{Panda}$$

    - This means to find a vector in our world, we need to know Panda's basis vectors in our coordinates.
        - For example, Panda's basis vectors are $\begin{bmatrix}3\\1\end{bmatrix}$ and $\begin{bmatrix}1\\1\end{bmatrix}$ in our world.
        - The transformation matrix is $B = \begin{bmatrix}3 & 1\\1 & 1\end{bmatrix}$
        - A vector in Panda's world is $v_{Panda} = \begin{bmatrix}3/2\\1/2\end{bmatrix}$
        - So the corresponding vector in our world is:

$$v_{me} = \begin{bmatrix}3 & 1\\1 & 1\end{bmatrix}\begin{bmatrix}3/2\\1/2\end{bmatrix} = \begin{bmatrix}3(3/2)+1(1/2)\\1(3/2)+1(1/2)\end{bmatrix} = \begin{bmatrix}10/2\\4/2\end{bmatrix} = \begin{bmatrix}5\\2\end{bmatrix}$$

- (2) From 'Our World' to 'Panda's World':
    - To perform the reverse transformation—converting a vector from our world to Panda's—we need the inverse of the transformation matrix, B^{-1}.

$$v_{Panda} = B^{-1}v_{me}$$

    - This inverse matrix, B^{-1}, essentially contains **our basis vectors expressed in Panda's coordinates.**
        - For example, the inverse of the matrix of $B = \begin{bmatrix}3 & 1\\1 & 1\end{bmatrix}$ is $B^{-1} = \frac{1}{\det(B)}\begin{bmatrix}1 & -1\\-1 & 3\end{bmatrix}$
        - $\det(B) = (3)(1) - (1)(1) = 2$. So, $B^{-1} = \frac{1}{2}\begin{bmatrix}1 & -1\\-1 & 3\end{bmatrix}$
        - Using the vector $v_{me} = \begin{bmatrix}5\\2\end{bmatrix}$

$$v_{me} = \frac{1}{2}\begin{bmatrix}1 & -1\\-1 & 3\end{bmatrix}\begin{bmatrix}5\\2\end{bmatrix} = \frac{1}{2}\begin{bmatrix}3\\1\end{bmatrix} = \begin{bmatrix}2/3\\1/2\end{bmatrix}$$

    - This returns us to our original vector in Panda's world.
- Special case: orthonormal bases
    - When the basis vectors are orthonormal (orthogonal and of unit length), the inverse matrix B^{-1} is simply the **transpose** of the matrix B.

$$B^{-1} = B^T$$

    - This makes the reverse transformation much simpler.
- Transformation using projections
    - For an **orthonormal basis**, we can convert a vector from one basis to another without complex matrix inversion. The new components of a vector are simply its projections onto the new basis vectors.
    - To convert our vector v_{me} into Panda's world components, we simply take the dot product of our vector with each of Panda's basis vectors.
    - If Panda's orthonormal basis vectors are $\hat{b_1}$ and $\hat{b_2}$, then:
        - The first component of v_{Panda} is $v_{me} \cdot \hat{b_1}$.
        - The second component of v_{Panda} is $v_{me} \cdot \hat{b_2}$.

- This projection method only works for ***orthonormal bases***. If the basis vectors are not orthogonal, we must use the more general matrix inversion method.

D4. Transformation in a changed basis

- Sometimes we have a vector in a non-standard basis and want to apply a linear transformation, but the transformation is only defined in the standard basis. To do this, we must convert the vector to the standard basis, apply the transformation, and then convert the result back to the original non-standard basis.
- This process is a fundamental concept in linear algebra, often referred to as similarity transformation or change of basis transformation.
- The 3-Step transformation process:
  - Let's use our familiar "Bear's world" and "my world" analogy.
  - My world uses the standard orthonormal basis $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$.
  - Bear's world uses a non-standard basis, defined in my world by the columns of matrix B.
  - R is a transformation matrix (e.g., a rotation) defined in my world.
  - We want to find the transformation matrix R_B that performs the same operation in Bear's world.
  - The process to apply a transformation R (defined in my world) to a vector v (defined in Bear's world) and get a new vector v'(also in Bear's world) is a sequence of three steps:
    1. **Translate the vector to my world**: Convert the vector v from Bear's world to my world by multiplying it by the transformation matrix B.

       $$B_v$$

    2. **Apply the transformation in my world**: Apply the transformation R to the now-translated vector.

       $$R(B_v)$$

    3. **Translate the result back to Bear's world**: Convert the resulting vector back into Bear's basis by multiplying it by the inverse matrix B^{-1}

       $$B^{-1}(R(B_v)) = (B^{-1}RB)v$$

  - The resulting matrix $(B^{-1}RB)$ is the transformation matrix that performs the same operation as R but in Bear's coordinate system.
  - For example, a 45-degree rotation in Bear's World
    - **Bear's basis matrix (in my world)**:

      $$B = \begin{bmatrix}3 & 1\\1 & 1\end{bmatrix}$$

    - **45-degree rotation matrix (in my world)**:

$$R = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$$

    - **Bear's inverse basis matrix (my basis in Bear's world)**:

$$B^{-1} = \frac{1}{2} \begin{bmatrix} 1 & -1 \\ -1 & 3 \end{bmatrix}$$

- To find the matrix R_B that performs a 45-degree rotation in Bear's world, we compute the product $B^{-1}RB$:

$$R_B = \frac{1}{2} \begin{bmatrix} 1 & -1 \\ -1 & 3 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 \\ 1 & 1 \end{bmatrix} = \frac{1}{2\sqrt{2}} \begin{bmatrix} -2 & -2 \\ 10 & 6 \end{bmatrix}$$

D5. The transpose and orthogonal matrices

- In linear algebra, the ***transpose*** is a new operation on a matrix where we interchange its rows and columns. The transpose of matrix A is denoted as A^T. The element at position (i, j) in A^T is the element at position (j, i) in the original matrix A.
    - For example, $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, it's transpose is $A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$. The diagonal elements remain the same.
- Orthogonal matrices and orthonormal bases
    - A transformation matrix A is called an orthogonal matrix if its columns form an orthonormal basis. An orthonormal basis is a set of vectors that are:
        - Orthogonal: The dot product of any two distinct vectors is zero ($a_i \cdot a_j = 0 \text{ for } i \neq j$).
        - Unit Length: The dot product of any vector with itself is one ($a_i \cdot a_j = 1$).
    - This special property leads to a powerful relationship between a matrix and its transpose. When we multiply an orthogonal matrix A by its transpose A^T , the result is the **identity matrix** (I).

$$A^T A = I$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (0)(0) + (1)(1) & (0)(1) + (1)(0) \\ (1)(0) + (0)(1) & (1)(1) + (0)(0) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

    - This relationship is crucial because the identity matrix is the result we get when we multiply a matrix by its inverse ($A^{-1}A = I$). Therefore, for an ***orthogonal matrix***, its transpose is also its inverse.

$$A^T = A^{-1}$$

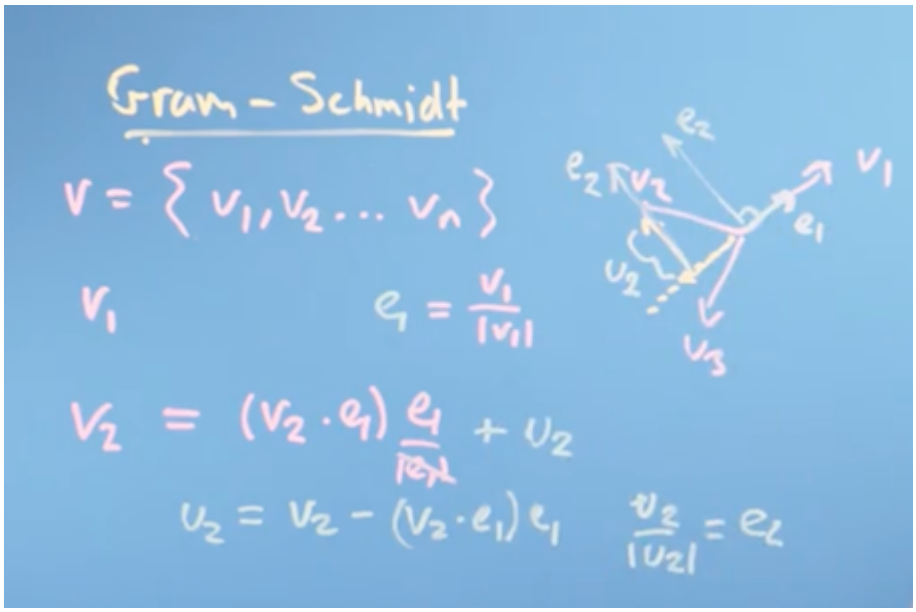
    - This means that for an orthogonal matrix, calculating the inverse is as simple as finding its transpose, which is a significant computational shortcut. This

property is also symmetric: the rows of an orthogonal matrix are also orthonormal.

- Properties of orthogonal matrices
  - Because the columns and rows of an orthogonal matrix have unit length, the transformation it represents must preserve lengths and angles.
    - Determinant: an orthogonal matrix scales space by a factor of one. Its determinant must therefore be either $+1$ (a standard rotation or reflection) or -1 (a rotation or reflection followed by an inversion, which makes the new basis "left-handed").
    - Ease of calculation: orthogonal matrices are highly desirable in data science and other fields because they make calculations simple. Since the basis vectors are orthogonal, a vector's components in this new basis can be found with a simple **dot product** (projection) instead of a complex matrix multiplication.

D6. The Gram-Schmidt process (how to construct an orthonormal basis)

- Constructing an orthonormal basis: the Gram-Schmidt process
  - We've established that working with an **orthonormal basis** (where all basis vectors are perpendicular and have unit length) makes many linear algebra operations, like finding a matrix inverse, much easier. However, most vector sets aren't orthonormal. The **Gram-Schmidt process** is a systematic method for converting any set of linearly independent vectors into an orthonormal set.
  - The process works by taking a set of initial vectors, v_1, v_2,..., v_n and constructing a new set of orthonormal vectors, e_1, e_2,..., e_n.



  - Step 1: Normalize the first vector
    - The first vector of our new basis, e_1, is simply the first original vector, v_1 , normalized to have a length of one. We do this by dividing the vector by its own length.

$$e_1 = \frac{v1}{||v1||}$$

- This gives us the first unit vector in our new, orthonormal basis.

- Step 2: Construct the second orthogonal vector
  - To find the second basis vector, e_2, we need to find a vector that is perpendicular to e_1. We start with the second original vector, v_2, and subtract its **projection** onto e_1. The projection is the component of v_2 that lies in the direction of e_1.
    1. **Find the orthogonal component**: The un-normalized vector u_2 that is perpendicular to e_1 is given by:

       $$u_2 = v_2 - (v_2 \cdot e_1)e_1$$

       Here, $(v_2 \cdot e_1)$ gives us the scalar projection of v_2 onto e_1, and multiplying it by e_1 gives us the vector projection.
    2. **Normalize the component**: Once we have v_2, we normalize it to get our second orthonormal basis vector, e_2.

       $$e_2 = \frac{u_2}{||u_2||}$$

- Step 3: Extend to higher dimensions
  - The process continues in the same way for all subsequent vectors. For a third vector, v_3, we would subtract its projections onto both e_1 and e_2 to find the component that is perpendicular to the plane spanned by e_1 and e_2.
    1. **Find the orthogonal component**: The un-normalized vector u_2 that is perpendicular to e_1 is given by:

       $$u_3 = v_3 - (v_3 \cdot e_1)e_1 - (v_3 \cdot e_2)e_2$$

    2. **Normalize the component**: We then normalize u_3 to get our third orthonormal basis vector, e_3.

       $$e_3 = \frac{u_3}{||u_3||}$$

       This process can be repeated for all n vectors in the set, giving you a full set of orthonormal basis vectors e_1, e_2,..., e_n, that span the same space as our original vectors.

Python code: The Gram Schmidt process

In [175…

```
import numpy as np
import numpy.linalg as la

verySmallNumber = 1e-14 # That's 1×10⁻¹⁴ = 0.00000000000001

# Our first function will perform the Gram-Schmidt procedure for 4 basis
```

```
# We'll take this list of vectors as the columns of a matrix, A.
# We'll then go through the vectors one at a time and set them to be orth
# to all the vectors that came before it. Before normalising.
# Follow the instructions inside the function at each comment.
# You will be told where to add code to complete the function.
def gsBasis4(A) :
    B = np.array(A, dtype=np.float64) # Make B as a copy of A, since we'r
    # The zeroth column is easy, since it has no other vectors to make it
    # All that needs to be done is to normalise it. I.e. divide by its mo
    B[:, 0] = B[:, 0] / la.norm(B[:, 0])
    # For the first column, we need to subtract any overlap with our new
    B[:, 1] = B[:, 1] - B[:, 1] @ B[:, 0] * B[:, 0]
    # If there's anything left after that subtraction, then B[:, 1] is li
    # If this is the case, we can normalise it. Otherwise we'll set that
    if la.norm(B[:, 1]) > verySmallNumber :
        B[:, 1] = B[:, 1] / la.norm(B[:, 1])
    else :
        B[:, 1] = np.zeros_like(B[:, 1])
    # Now we need to repeat the process for column 2.
    # Insert two lines of code, the first to subtract the overlap with th
    # and the second to subtract the overlap with the first.
    B[:, 2] = B[:, 2] - B[:, 2] @ B[:, 0] * B[:, 0]
    B[:, 2] = B[:, 2] - B[:, 2] @ B[:, 1] * B[:, 1]
    # Again we'll need to normalise our new vector.
    # Copy and adapt the normalisation fragment from above to column 2.
    if la.norm(B[:, 2]) > verySmallNumber :
        B[:, 2] = B[:, 2] / la.norm(B[:, 2])
    else :
        B[:, 2] = np.zeros_like(B[:, 2])
    # Finally, column three:
    # Insert code to subtract the overlap with the first three vectors.
    B[:, 3] = B[:, 3] - B[:, 3] @ B[:, 0] * B[:, 0]
    B[:, 3] = B[:, 3] - B[:, 3] @ B[:, 1] * B[:, 1]
    B[:, 3] = B[:, 3] - B[:, 3] @ B[:, 2] * B[:, 2]
    # Now normalise if possible
    if la.norm(B[:, 3]) > verySmallNumber :
        B[:, 3] = B[:, 3] / la.norm(B[:, 3])
    else :
        B[:, 3] = np.zeros_like(B[:, 3])

    # Finally, we return the result:
    return B

# The second part of this exercise will generalise the procedure.
# Previously, we could only have four vectors, and there was a lot of rep
# We'll use a for-loop here to iterate the process for each vector.
def gsBasis(A) :
    B = np.array(A, dtype=np.float64) # Make B as a copy of A, since we'r
    # Loop over all vectors, starting with zero, label them with i
    for i in range(B.shape[1]) :
        # Inside that loop, loop over all previous vectors, j, to subtrac
        for j in range(i) :
            # Complete the code to subtract the overlap with previous vec
            # you'll need the current vector B[:, i] and a previous vecto
            B[:, i] = B[:, i] - B[:, i] @ B[:, j] * B[:, j]
        # Next insert code to do the normalisation test for B[:, i]
        if la.norm(B[:, i]) > verySmallNumber :
            B[:, i] = B[:, i] / la.norm(B[:, i])
        else:
            B[:, i] = np.zeros_like(B[:, i])
```

```
    # Finally, we return the result:
    return B

# This function uses the Gram-schmidt process to calculate the dimension
# spanned by a list of vectors.
# Since each vector is normalised to one, or is zero,
# the sum of all the norms will be the dimension.
def dimensions(A) :
    return np.sum(la.norm(gsBasis(A), axis=0))
```

In [176…
```
V = np.array([[1,0,2,6],
              [0,1,8,2],
              [2,8,3,1],
              [1,-6,2,3]], dtype=np.float64)
gsBasis4(V)
```

Out[176…
```
array([[ 0.40824829, -0.1814885 ,  0.04982278,  0.89325973],
       [ 0.        ,  0.1088931 ,  0.99349591, -0.03328918],
       [ 0.81649658,  0.50816781, -0.06462163, -0.26631346],
       [ 0.40824829, -0.83484711,  0.07942048, -0.36063281]])
```

In [177…
```
# Once you've done Gram-Schmidt once,
# doing it again should give you the same result. Test this:
U = gsBasis4(V)
gsBasis4(U)
```

Out[177…
```
array([[ 0.40824829, -0.1814885 ,  0.04982278,  0.89325973],
       [ 0.        ,  0.1088931 ,  0.99349591, -0.03328918],
       [ 0.81649658,  0.50816781, -0.06462163, -0.26631346],
       [ 0.40824829, -0.83484711,  0.07942048, -0.36063281]])
```

In [178…
```
# Try the general function too.
gsBasis(V)
```

Out[178…
```
array([[ 0.40824829, -0.1814885 ,  0.04982278,  0.89325973],
       [ 0.        ,  0.1088931 ,  0.99349591, -0.03328918],
       [ 0.81649658,  0.50816781, -0.06462163, -0.26631346],
       [ 0.40824829, -0.83484711,  0.07942048, -0.36063281]])
```

In [179…
```
# See what happens for non-square matrices
A = np.array([[3,2,3],
              [2,5,-1],
              [2,4,8],
              [12,2,1]], dtype=np.float64)
gsBasis(A)
```

Out[179…
```
array([[ 0.23643312,  0.18771349,  0.22132104],
       [ 0.15762208,  0.74769023, -0.64395812],
       [ 0.15762208,  0.57790444,  0.72904263],
       [ 0.94573249, -0.26786082, -0.06951101]])
```

In [180…
```
dimensions(A)
```

Out[180…
```
np.float64(3.0)
```

In [181…
```
B = np.array([[6,2,1,7,5],
              [2,8,5,-4,1],
```

```
                [1,-6,3,2,8]], dtype=np.float64)
gsBasis(B)
```

Out[181…
```
array([[ 0.93704257, -0.12700832, -0.32530002,  0.        ,  0.
],
       [ 0.31234752,  0.72140727,  0.61807005,  0.        ,  0.
],
       [ 0.15617376, -0.6807646 ,  0.71566005,  0.        ,  0.
]])
```

In [182…
```
dimensions(B)
```

Out[182…
```
np.float64(3.0)
```

In [183…
```
# Now let's see what happens when we have one vector that is a linear com
C = np.array([[1,0,2],
              [0,1,-3],
              [1,0,2]], dtype=np.float64)
gsBasis(C)
```

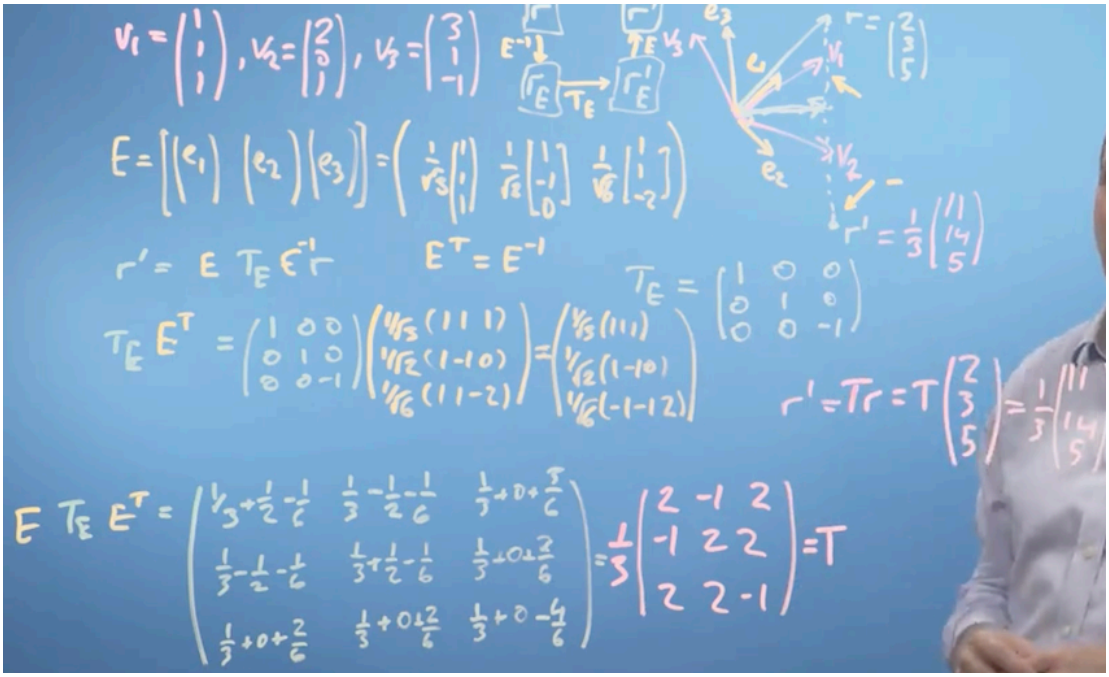
Out[183…
```
array([[0.70710678, 0.        , 0.        ],
       [0.        , 1.        , 0.        ],
       [0.70710678, 0.        , 0.        ]])
```

In [184…
```
dimensions(C)
```

Out[184…
```
np.float64(2.0)
```

D7. Example of transformation - Reflecting in a plane

- Solving complex linear transformations in arbitrary spaces can be a daunting task using traditional geometry and trigonometry. However, by leveraging the power of **basis transformations**, we can simplify the problem dramatically. The strategy is to move the problem from our complex "world" into a simple, new coordinate system, perform the transformation, and then move back.



- Our problem: find the reflection of a vector $r = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}$ across a mirror plane defined by two vectors: $v_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ and $v_2 = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$. A third vector,$v_3 = \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix}$, is provided, which is out of the plane.
  - Step 1: Constructing an Orthonormal Basis

    The first step is to construct an orthonormal basis, $E = [e_1, e_2, e_3]$, from our initial vectors v_1,v_2,v_3 using the **Gram-Schmidt process**. An orthonormal basis is a set of mutually perpendicular vectors, each with a length of one. This is crucial because, in an orthonormal basis, the inverse of the matrix E is simply its transpose, $E^{-1} = E^T$.

    1. Find e_1 (from v_1)

       We normalize v_1 to get the first basis vector.

       $$||v_1|| = \sqrt{1^2 + 1^2 + 1^2} = \sqrt{3}$$

       $$e_1 = \frac{v_1}{||v_1||} = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

    2. Find e_2 (from v_2)

       We subtract the projection of v_2 onto e_1 from v_2 to get an orthogonal vector u_2. Then we normalize u_2.

       $$\mathbf{u}_2 = \mathbf{v}_2 - (\mathbf{v}_2 \cdot \mathbf{e}_1)\mathbf{e}_1$$

$$(\mathbf{v}_2 \cdot \mathbf{e}_1) = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} \cdot \left(\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \frac{1}{\sqrt{3}}(2+0+1) = \frac{3}{\sqrt{3}} = \sqrt{3}$$

$$\mathbf{u}_2 = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} - \sqrt{3} \left(\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

$$\|\mathbf{u}_2\| = \sqrt{1^2 + (-1)^2 + 0^2} = \sqrt{2}$$

$$\mathbf{e}_2 = \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

3. Find e_3 (from v_3)

   We subtract the projections of v_3 onto both e_1 and e_2 to get u_3, which will be orthogonal to the plane. Then we normalize it.

$$\mathbf{u}_3 = \mathbf{v}_3 - (\mathbf{v}_3 \cdot \mathbf{e}_1)\mathbf{e}_1 - (\mathbf{v}_3 \cdot \mathbf{e}_2)\mathbf{e}_2$$

$$(\mathbf{v}_3 \cdot \mathbf{e}_1) = \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix} \cdot \left(\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \frac{1}{\sqrt{3}}(3+1-1) = \frac{3}{\sqrt{3}} = \sqrt{3}$$

$$(\mathbf{v}_3 \cdot \mathbf{e}_2) = \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix} \cdot \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \right) = \frac{1}{\sqrt{2}}(3-1+0) = \frac{2}{\sqrt{2}} = \sqrt{2}$$

$$\mathbf{u}_3 = \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix} - \sqrt{3} \left(\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) - \sqrt{2} \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \right)$$

$$\mathbf{u}_3 = \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix}$$

$$\|\mathbf{u}_3\| = \sqrt{1^2 + 1^2 + (-2)^2} = \sqrt{6}$$

$$\mathbf{e}_3 = \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} = \frac{1}{\sqrt{6}} \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix}$$

   Our new orthonormal basis matrix is $E = [e_1, e_2, e_3]$.

- Step 2: Defining the Reflection Transformation (T_E)

  In our new coordinate system, defined by the basis vectors e_1, e_2, and e_3, the mirror plane lies in the e_1-e_2 plane. The vector e_3 is perpendicular to it. Reflecting a vector across this plane is simple: its e_1 and e_2 components remain unchanged, while its e_3 component is flipped (multiplied by -1).

The transformation matrix in this new basis, T_E, is a diagonal matrix:

$$T_E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

- Step 3: Performing the Basis Change & Transformation

  Now we can combine everything. We need to:

  1. Transform our vector r from its original basis to the new E basis.
  2. Apply the simple reflection transformation T_E.
  3. Transform the resulting vector back to the original basis.

  The formula for this is:

  $$T = E \cdot T_E \cdot E^{-1}$$

  Since our basis E is orthonormal, we can simplify E^{-1} to E^T.

  $$T = E \cdot T_E \cdot E^T$$

  This matrix multiplication yields the final transformation matrix T in our original basis.

- Step 4: Applying the Final Transformation

  We apply the final transformation matrix T to our vector r to get the reflected vector r'.

$$\mathbf{r}' = T\mathbf{r}$$

$$T = E \cdot T_E \cdot E^T = \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & 0 & -\frac{2}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & -\frac{} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & 0 & \frac{2}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & -\frac{2}{\sqrt{6}} \end{bmatrix} = \begin{bmatrix} \frac{1}{3} + \frac{1}{2} - \frac{1}{6} & \frac{1}{3} - \frac{1}{2} - \\ \frac{1}{3} - \frac{1}{2} - \frac{1}{6} & \frac{1}{3} + \frac{1}{2} - \\ \frac{1}{3} + \frac{2}{6} & \frac{1}{3} + \frac{2}{6} \end{bmatrix}$$

$$\mathbf{r}' = \frac{1}{3} \begin{bmatrix} 2 & -1 & 2 \\ -1 & 2 & 2 \\ 2 & 2 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} (2)(2) + (-1)(3) + (2)(5) \\ (-1)(2) + (2)(3) + (2)(5) \\ (2)(2) + (2)(3) + (-1)(5) \end{bmatrix} = \frac{1}{3} \begin{bmatrix}$$

Python code: Transformation-Relfection (Reflecting Bear)

In [185…

```
# PACKAGE
# Run this cell first once to load the dependancies.
import numpy as np
```

```
from numpy.linalg import norm, inv
from numpy import transpose
```

In [186…

```
# In this function, we will return the transformation matrix T,
# having built it out of an orthonormal basis set E that we create from B
# and a transformation matrix in the mirror's coordinates TE.
def build_reflection_matrix(bearBasis) : # The parameter bearBasis is a 2
    # Use the gsBasis function on bearBasis to get the mirror's orthonorm
    E = gsBasis(bearBasis)
    # Write a matrix in component form that performs the mirror's reflect
    # Recall, the mirror operates by negating the last component of a vec
    # Replace a,b,c,d with appropriate values
    TE = np.array([[1, 0],
                   [0, -1]])
    # Combine the matrices E and TE to produce our transformation matrix.
    T = E @ TE @ inv(E)
    # Finally, we return the result. There is no need to change this line
    return T

def draw_mirror(bearBasis, ax):
    # The mirror is defined by the first vector in the basis
    mirror_vector = bearBasis[:, 0]
    # To find a line through the origin, we need two points.
    # We can use the mirror_vector and its negative.
    x = [-mirror_vector[0] * 5, mirror_vector[0] * 5]
    y = [-mirror_vector[1] * 5, mirror_vector[1] * 5]
    ax.plot(x, y, color='lightgreen', linewidth=2, zorder=0)

bear_white = 'whitesmoke'
bear_black = 'k'
```

In [187…

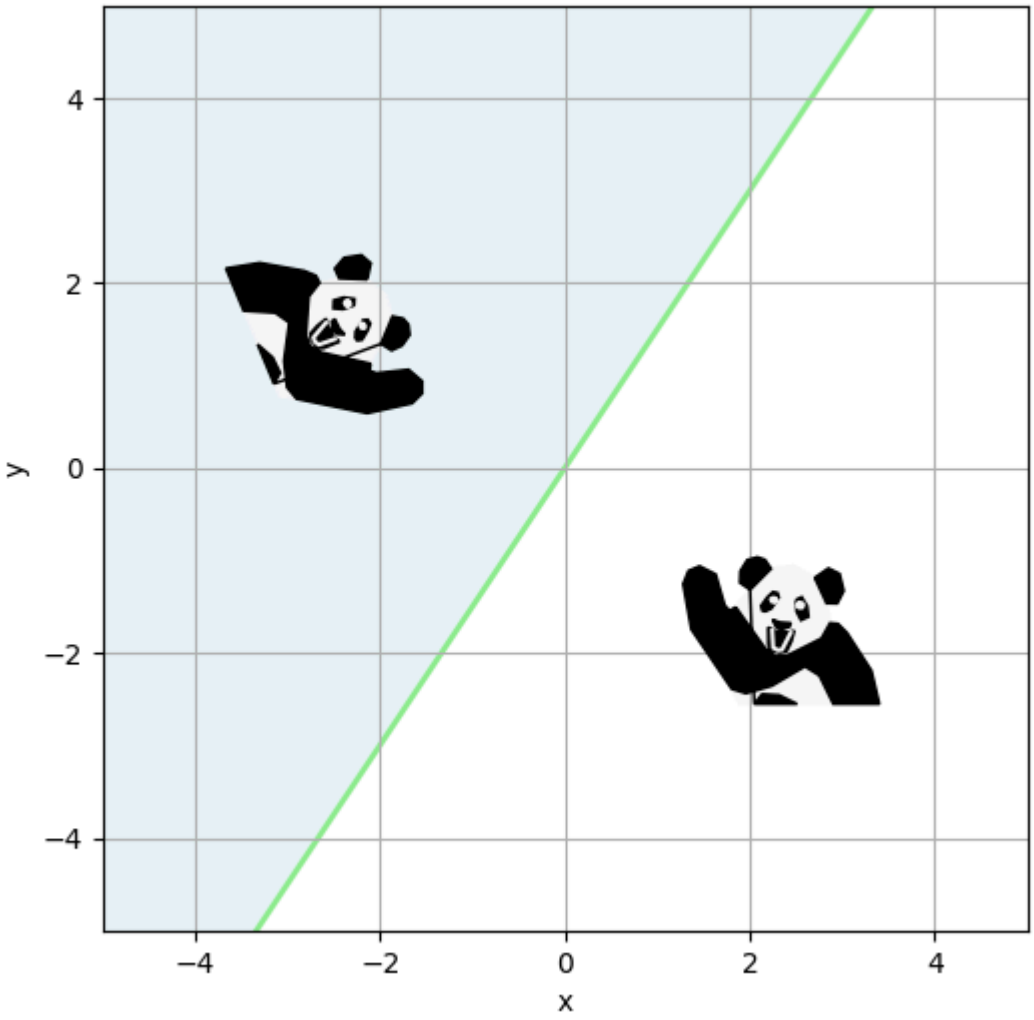
```
bear_white_fur = np.array([[ 2.229253 ,  2.4680387,  2.7017972,  2.829987
         2.7746897,  2.4052003,  2.2795237,  2.1639012,  1.847196 ,
         2.0030351,  2.229253 ,        np.nan,  1.8044659,  1.8974666,
         2.0491517,  2.1326865,  2.3175294,  2.5112591,  2.9028799,
         2.7520679,  2.5962289,  2.2367936,  1.9577914,  1.8044659],
       [-1.0902537, -1.0601388, -1.1881273, -1.4641809, -1.677495 ,
        -1.8431272, -2.028836 , -2.0363647, -1.9485295, -1.5043341,
        -1.3186252, -1.0902537,        np.nan, -2.3902152, -2.5527822,
        -2.5527822, -2.4463632, -2.4570104, -2.5527822, -2.5527822,
        -2.2446597, -2.1467861, -2.3575907, -2.4378972, -2.3902152]])

bear_black_fur = np.array([[ 2.0030351 ,  2.229253  ,  2.1639012 ,  2.080
         1.8974666 ,  1.8924396 ,  2.0030351 ,        np.nan,  2.7017972
         2.8500957 ,  2.9707453 ,  3.0159889 ,  2.94561   ,  2.8299874 ,
         2.7017972 ,        np.nan,  2.1639012 ,  2.2317666 ,  2.3147132
         2.299632  ,  2.2493613 ,  2.1890365 ,  2.1211711 ,  2.1337387 ,
         2.1639012 ,        np.nan,  2.4982011 ,  2.5610936 ,  2.6213642
         2.633986  ,  2.5536071 ,  2.5057417 ,  2.4982011 ,        np.na
         2.2468478 ,  2.3247673 ,  2.4429034 ,  2.4303357 ,  2.3448755 ,
         2.2820372 ,  2.2468478 ,        np.nan,  2.1966706 ,  2.2722074
         2.4055076 ,  2.481933  ,  2.449941  ,  2.4001756 ,  2.3237501 ,
         2.222442  ,  2.1984479 ,  2.1966706 ,        np.nan,  1.847196
         1.7818441 ,  1.7290599 ,  1.6310321 ,  1.4575984 ,  1.3369488 ,
         1.2791375 ,  1.3671112 ,  1.8044659 ,  1.9577914 ,  2.2367936 ,
         2.5962289 ,  2.7520679 ,  2.9028799 ,  3.4005595 ,  3.3150993 ,
         3.0511783 ,  2.9531506 ,  2.8676905 ,  2.7746897 ,  2.4052003 ,
         2.2795237 ,  2.1639012 ,  1.847196  ,        np.nan,  2.0491517
         2.5112591 ,  2.3175294 ,  2.1326865 ,  2.0491517 ],
```

```
       [-1.3186252 , -1.0902537 , -0.99238015, -0.96477475, -0.99488975,
        -1.1153494 , -1.2408283 , -1.3186252 ,         np.nan, -1.1881273
        -1.0852346 , -1.1454645 , -1.3286636 , -1.4666904 , -1.4641808 ,
        -1.1881273 ,         np.nan, -1.5545256 , -1.5219011 , -1.4014413
        -1.3512497 , -1.3412115 , -1.3989317 , -1.4917862 , -1.5419777 ,
        -1.5545256 ,         np.nan, -1.4265371 , -1.3964222 , -1.4968054
        -1.6097363 , -1.64738   , -1.5545256 , -1.4265371 ,         np.na
        -1.6423608 , -1.6699662 , -1.677495  , -1.7176483 , -1.7477632 ,
        -1.7176483 , -1.6423608 ,         np.nan, -1.7223509 , -1.7622781
        -1.7764744 , -1.7613908 , -1.8767359 , -1.9805465 , -1.9991791 ,
        -1.9672374 , -1.913114  , -1.7223509 ,         np.nan, -1.5043341
        -1.5444873 , -1.486767  , -1.1504836 , -1.0626484 , -1.11284   ,
        -1.2558858 , -1.7452537 , -2.3902152 , -2.4378972 , -2.3575907 ,
        -2.1467861 , -2.2446597 , -2.5527822 , -2.5527822 , -2.1919586 ,
        -1.7828973 , -1.6850238 , -1.677495  , -1.8431272 , -2.028836  ,
        -2.0363647 , -1.9485295 , -1.5043341 ,         np.nan, -2.5527822
        -2.5527822 , -2.4570104 , -2.4463632 , -2.5527822 ]])

bear_face = np.array([[ 2.2419927,  2.2526567,  2.3015334,  2.3477442,  2
                np.nan,  2.5258499,  2.5113971,  2.5327621,  2.5632387,
         2.5780058,  2.5726645,  2.5475292,  2.5258499,         np.nan,
         2.2858075,  2.2704121,  2.2402497,  2.2283105,  2.2484187,
         2.273554 ,  2.2858075],
       [-1.7605035, -1.9432811, -1.9707865, -1.9654629, -1.781798 ,
                np.nan, -1.4688862, -1.4942957, -1.5099806, -1.5112354,
        -1.4877081, -1.466063 , -1.4588479, -1.4688862,         np.nan,
        -1.4346933, -1.4506918, -1.4463002, -1.418381 , -1.4055194,
        -1.4083427, -1.4346933]])
```

In [188…

```
# First load Pyplot, a graph plotting library.
%matplotlib inline
import matplotlib.pyplot as plt

# This is the matrix of Bear's basis vectors.
# (When we've done the exercise once, see what happns when we change Bear
bearBasis = np.array(
    [[1,   -1],
     [1.5, 2]])
# This line uses our code to build a transformation matrix for us to use.
T = build_reflection_matrix(bearBasis)

# Bear is drawn as a set of polygons, the vertices of which are placed as
# We have three of these non-square matrix lists: bear_white_fur, bear_bl
# We'll make new lists of vertices by applying the T matrix we've calcula
reflected_bear_white_fur = T @ bear_white_fur
reflected_bear_black_fur = T @ bear_black_fur
reflected_bear_face = T @ bear_face

# This next line runs a code to set up the graphics environment.
# Instead of using a predefined function, we'll create the plot manually.
fig, ax = plt.subplots(figsize=(6, 6))
ax.set_aspect('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)

# Calculate the slope of the mirror line and fill the background.
m = bearBasis[1,0] / bearBasis[0,0]
```

```
ax.fill_between([-5, 5], [-5*m, 5*m], 5, color='lightblue', alpha=0.3, zo

draw_mirror(bearBasis, ax)

# We'll first plot Bear, his white fur, his black fur, and his face.
ax.fill(bear_white_fur[0], bear_white_fur[1], color=bear_white, zorder=1)
ax.fill(bear_black_fur[0], bear_black_fur[1], color=bear_black, zorder=2)
ax.plot(bear_face[0], bear_face[1], color=bear_white, zorder=3)

# Next we'll plot Bear's reflection.
ax.fill(reflected_bear_white_fur[0], reflected_bear_white_fur[1], color=b
ax.fill(reflected_bear_black_fur[0], reflected_bear_black_fur[1], color=b
ax.plot(reflected_bear_face[0], reflected_bear_face[1], color=bear_white,

plt.show()
```



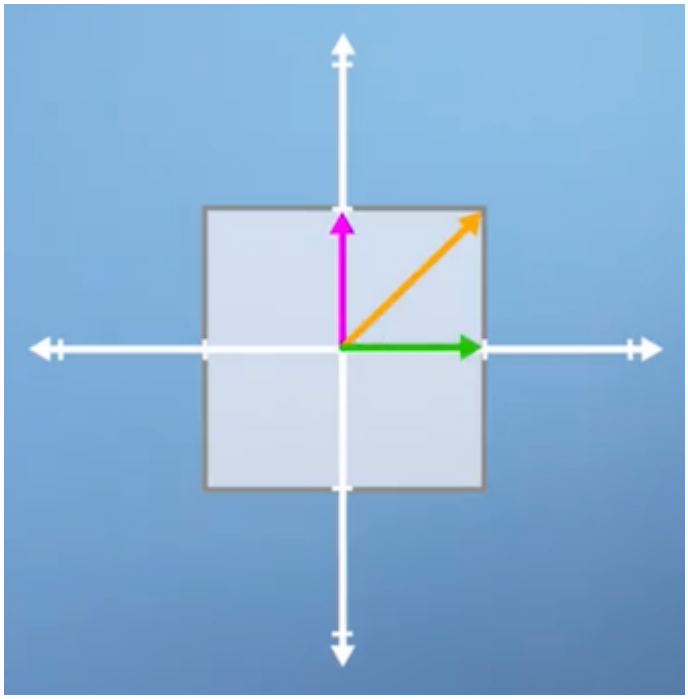
In [189…
```
T = np.array([[1,0],
              [2,-1]])
C = np.array([[1,2],
              [0,1]])
ans = inv(C) @ T @ C

T @ T @ T @ T @ T
```

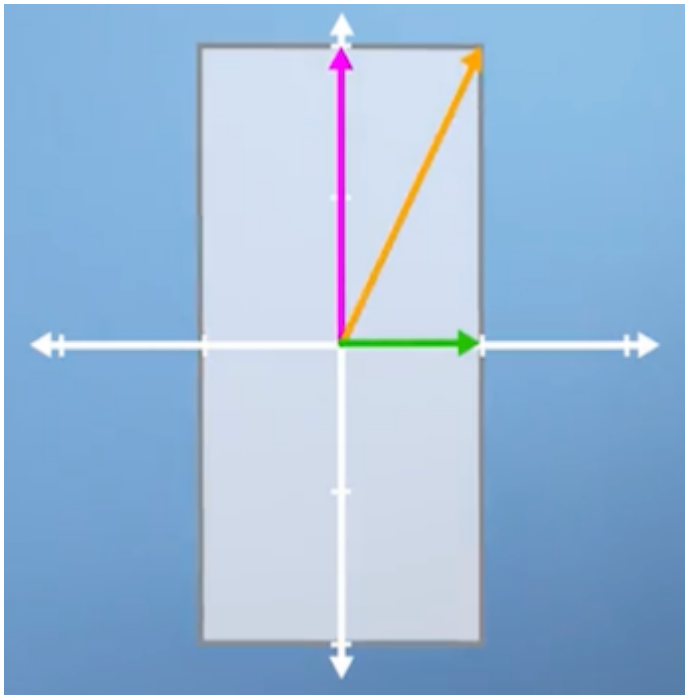
Out[189…
```
array([[ 1,  0],
       [ 2, -1]])
```

E. Eigenvalues and Eigenvectors

E1. Introduction to Eigenvectors and Eigenvalues

- The German word ***"eigen"*** translates to ***"characteristic."*** In linear algebra, eigenvectors and eigenvalues are a way of understanding the characteristic properties of a linear transformation. Instead of looking at how a transformation affects every single vector, we look for special vectors that are only scaled, not changed in direction.
- The geometric intuition
  - When a linear transformation (like **scaling**, **shearing**, or **rotation**) is applied to a space, most vectors will be moved and their direction will be changed. However, there are ***some special vectors that remain on their original "span" or line***, even after the transformation. These special vectors are the eigenvectors.
    - Eigenvectors: The vectors whose direction **"remains"** unchanged by the transformation.
    - Eigenvalues: The scalar value that measures **how much the eigenvector's length is scaled** during the transformation.
  - For example, consider a vertical scaling transformation that doubles the height of every vector.
    - A vertical vector will still point straight up but its length will be doubled. It's an eigenvector with an eigenvalue of 2.
    - A horizontal vector will remain on the horizontal axis and its length will be unchanged. It's an eigenvector with an eigenvalue of 1.
    - A diagonal vector's direction will be changed by the transformation. It is not an eigenvector.





- Common examples
  - Shear transformation: A pure shear transformation, which shifts all vectors horizontally while keeping their height constant, only has one eigenvector: the horizontal vector. Its direction remains unchanged, while all other vectors are shifted.
  - Rotation: A pure rotation (e.g., a 90-degree rotation) has **"NO"** eigenvectors at all, because every vector's direction is changed by the rotation.

E2. Special Eigen-cases and higher dimensions

- Eigenvectors and eigenvalues are powerful tools for understanding transformations, and exploring special cases helps solidify this geometric

intuition. In these cases, the eigenvectors aren't always easy to spot, but they provide crucial insights into a transformation's nature.

- Uniform scaling
  - When a transformation scales every vector by the same amount in all directions, **all vectors are eigenvectors**. The transformation simply stretches or shrinks the entire space uniformly. For this kind of transformation, every vector is an eigenvector, and they all share the same **eigenvalue**, which is the scaling factor.
- Rotation
  - While a small rotation typically has no eigenvectors, a **180-degree rotation is a unique case**. In this scenario, every vector is an **eigenvector**, as each one **remains on its original span but points in the opposite direction**. The corresponding eigenvalue is **-1**, which signifies a length change of 1 and a change in pointing direction.
- Combined transformations
  - When transformations are combined (e.g., a shear and a scaling), the eigenvectors become less obvious. An eigenvector may not align with the standard axes, and you might need to find it through calculation. However, if you can find the eigenvectors, they still provide the most stable directions of the transformation.
- Extending to 3D: The axis of rotation
  - In three dimensions, the concept of eigenvectors takes on a new, physically meaningful role. For a 3D rotation, most vectors will change their direction. However, there will be one vector that remains completely unchanged—the vector that lies along the **axis of rotation**. This vector is the **eigenvector of the 3D rotation**, and its corresponding eigenvalue is **1**, because its length is not changed. Finding the eigenvector of a 3D rotation matrix is therefore equivalent to finding the axis around which the rotation occurs.

E3. Calculating Eigenvectors

- The Eigenvalue Equation
  - We have explored the geometric intuition of eigenvectors and eigenvalues. Now, we will formalize this concept into an algebraic expression that allows us to calculate them for any square matrix. The core idea is that applying a transformation matrix A to an eigenvector x is equivalent to simply scaling the same eigenvector by a scalar value λ. This is the fundamental **eigenvalue equation**:

$$Ax = \lambda x$$

    1. A is a square transformation matrix.
    2. x is the eigenvector (a non-zero vector).
    3. λ is the eigenvalue (a scalar).

    - This equation states that the output of the transformation Ax remains in the same span as the original vector x.
- Deriving the Characteristic Equation
    - To find the solutions to the eigenvalue equation, we must rearrange it to solve for λ and x. The following steps show the standard derivation:
        1. **Rearrange the equation**: Move all terms to one side of the equation.

$$Ax - \lambda x = 0$$

        2. **Factor out the eigenvector**: We cannot directly subtract a scalar (λ) from a matrix (A). To make the operation valid, we introduce the identity matrix (I), which acts as the multiplicative identity for matrices.

$$Ax - \lambda Ix = 0$$

$$(A - \lambda I)x = 0$$

        3. The condition for a non-trivial solution: This equation shows that a matrix ($A - \lambda I$) multiplied by a vector x equals the zero vector. This can happen in two scenarios:
            - The trivial solution, where $x = 0$. We are not interested in this case, as eigenvectors are, by definition, non-zero vectors.
            - The matrix ($A - \lambda I$) is a **singular matrix**, meaning it collapses space and has a determinant of zero. This is the condition we need to solve for our eigenvalues.
    - This leads to the characteristic equation:

$$\det(A - \lambda I) = 0$$

      The eigenvalues (λ) are the values that satisfy this equation.
- Steps to Find Eigenvalues and Eigenvectors
    - Once we have the characteristic equation, we can follow a systematic process to find the eigenvalues and their corresponding eigenvectors.
        1. Set up the Characteristic Equation: For a 2x2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the characteristic equation is:

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \det \begin{bmatrix} a - \lambda & b \\ c & d - \lambda \end{bmatrix} = 0$$

           This simplifies to the **characteristic polynomial**:

$$(a - \lambda)(d - \lambda) - bc = 0$$

$$\lambda^2 - (a + d)\lambda + (ad - bc) = 0$$

           We can solve this quadratic equation to find the values of λ.
        2. Solve for the Eigenvalues (λ): Find the roots of the characteristic polynomial. The number of eigenvalues will be equal to the dimension of the matrix. Note that a rotation matrix like the 90-degree rotation

$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ results in the characteristic equation $\lambda^2 + 1 = 0$, which has no real solutions for λ, confirming there are no real eigenvectors.

3. Solve for the Eigenvectors (x): Substitute each eigenvalue (λ) we found back into the original equation: $(A - \lambda I)x = 0$. This will give we a system of linear equations to solve for the eigenvector x. Since the system is singular, it will have infinite solutions (all vectors in the same span), so we will typically express the eigenvector in terms of a parameter, like t.

- Example: A Vertical Scaling Transformation
  - Let's apply this method to a vertical scaling matrix, $A = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$.

    Characteristic Equation:

    $$\det \begin{bmatrix} 1-\lambda & 0 \\ 0 & 2-\lambda \end{bmatrix} = (1-\lambda)(2-\lambda) - (0)(0) = 0$$

    Solve for Eigenvalues:

    $$(1-\lambda)(2-\lambda) = 0$$

    This gives us two eigenvalues: $\lambda_1 = 1$ and $\lambda_2 = 2$. Solve for Eigenvectors: For $\lambda_1 = 1$:

    $$\left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} - 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

    This results in the equations $0x_1 + 0x_2 = 0$ and $0x_1 + 1x_2 = 0$. The second equation tells us $x_2 = 0$, and the first equation has no constraint on x_1. Thus, the eigenvector is any vector of the form $\begin{bmatrix} t \\ 0 \end{bmatrix}$, where $t \neq 0$. This confirms our geometric intuition: the horizontal vector is an eigenvector with an eigenvalue of 1.

    For $\lambda_2 = 2$:

    $$\left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} - 2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

    This results in the equations $-1x_1 + 0x_2 = 0$ and $0x_1 + 0x_2 = 0$. The first equation tells us $x_1 = 0$, and the second has no constraint on x_2. Thus, the eigenvector is any vector of the form $\begin{bmatrix} 0 \\ t \end{bmatrix}$, where $t \neq 0$. This confirms that the vertical vector is an eigenvector with an eigenvalue of 2.

E4. Changing to the Eigen basis: Diagonalization

- Diagonalization for Efficient Matrix Operations
  - When we need to apply a linear transformation repeatedly, such as a particle's movement over many time steps, multiplying the transformation matrix by itself over and over becomes computationally expensive. Diagonalization provides a clever and efficient way to handle this by leveraging eigenvectors and eigenvalues. The core idea is to change to a special coordinate system (an eigen-basis) where the transformation simplifies to a simple scaling operation, perform the calculation, and then transform back.
  - The Challenge: Raising a Matrix to a Power
    - Imagine we want to find the position of a particle after n time steps, where each step is described by a transformation matrix T. This requires calculating T^n.

$$v_n = T^n v_0$$

    - Multiplying a non-diagonal matrix by itself n times is computationally intensive. However, if the matrix were a diagonal matrix (where all non-diagonal elements are zero), raising it to the power of n is trivial—you simply raise each diagonal element to the power of n.
  - The Solution: The Power of an Eigen-basis
    - The key insight is that applying a transformation T is equivalent to a series of three steps:
      1. **Change of Basis (A Matrix with Eigenvectors as Columns)**: Transform a vector into a new coordinate system called the eigen-basis. The **columns** of this basis conversion matrix, let's call it C, are the **eigenvectors** of T. The inverse of this matrix, C^{-1}, is what takes a vector from our original basis into the eigen-basis.
      2. **Diagonal Transformation (Eigenvalues as the Diagonal Eletments)**: In this special eigen-basis, the transformation T is no longer complex. It becomes a simple scaling operation represented by a diagonal **matrix** D. The diagonal elements of D are the corresponding **eigenvalues** of T.
      3. **Change Back to Original Basis (Inverse Matrix)**: Transform the resulting vector back to the original coordinate system using the matrix C.
    - This relationship can be expressed by the equation:

$$T = CDC^{-1}$$

  - Diagonalization for Powers
    - Using the equation above, we can now express T^n as follows:

$$T^2 = (CDC^{-1})(CDC^{-1})$$

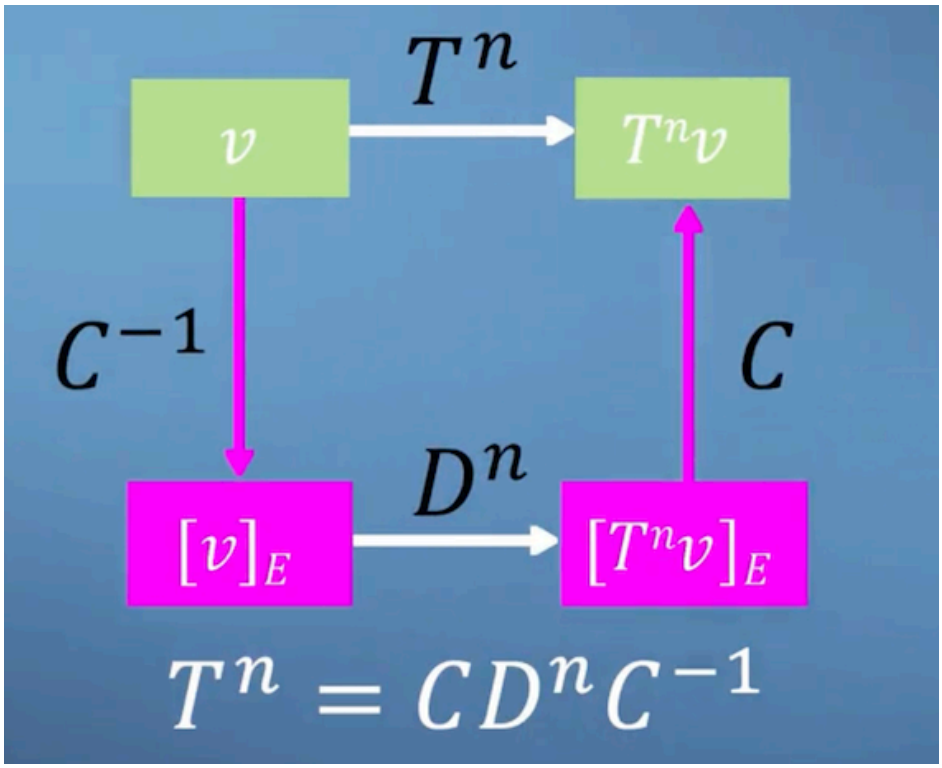
    - The adjacent $C^{-1}C$ in the middle cancels out, as any matrix multiplied by its inverse equals the identity matrix (I).

$$T^2 = CDDC^{-1} = CD^2C^{-1}$$

- We can generalize this to any power n:

$$T^n = CD^nC^{-1}$$

- This formula allows us to compute T^n efficiently. Instead of performing millions of matrix multiplications, we:
  - Find the **eigenvectors** and construct the matrix C.
  - Find the **eigenvalues** and construct the **diagonal matrix** D.
  - Raise the diagonal matrix D to the **power of** n (which is easy).
  - Perform three matrix multiplications: C, D^n, and C^{-1}.



- Example: A 2D Transformation
  - Let's illustrate with a simple example: a transformation matrix $T = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$. We want to find T^2.

1. Non-diagonalization approach: Directly calculating T^2 is straightforward for this simple matrix:

$$T = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} (1)(1) + (1)(0) & (1)(1) + (1)(2) \\ (0)(1) + (2)(0) & (0)(1) + (2)(2) \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & 4 \end{bmatrix}$$

2. Diagonalization approach:

   First, we find the eigenvectors and eigenvalues of T. The eigenvalues are $\lambda_1 = 1$ and $\lambda_2 = 2$. The corresponding eigenvectors are $x_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $x_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

- A. **Construct Matrices**:
  - **Eigenvector matrix** $C = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

    - **Diagonal eigenvalue matrix** $D = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$
    - **Inverse matrix** $C^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$
- B. **Calculate** D^2:

$$D^2 = \begin{bmatrix} 1^2 & 0 \\ 0 & 2^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$$

- C. **Apply the full formula**:

$$T^2 = CD^2C^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

$$T^2 = \begin{bmatrix} (1)(1) + (1)(0) & (1)(0) + (1)(4) \\ (0)(1) + (1)(0) & (0)(0) + (1)(4) \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

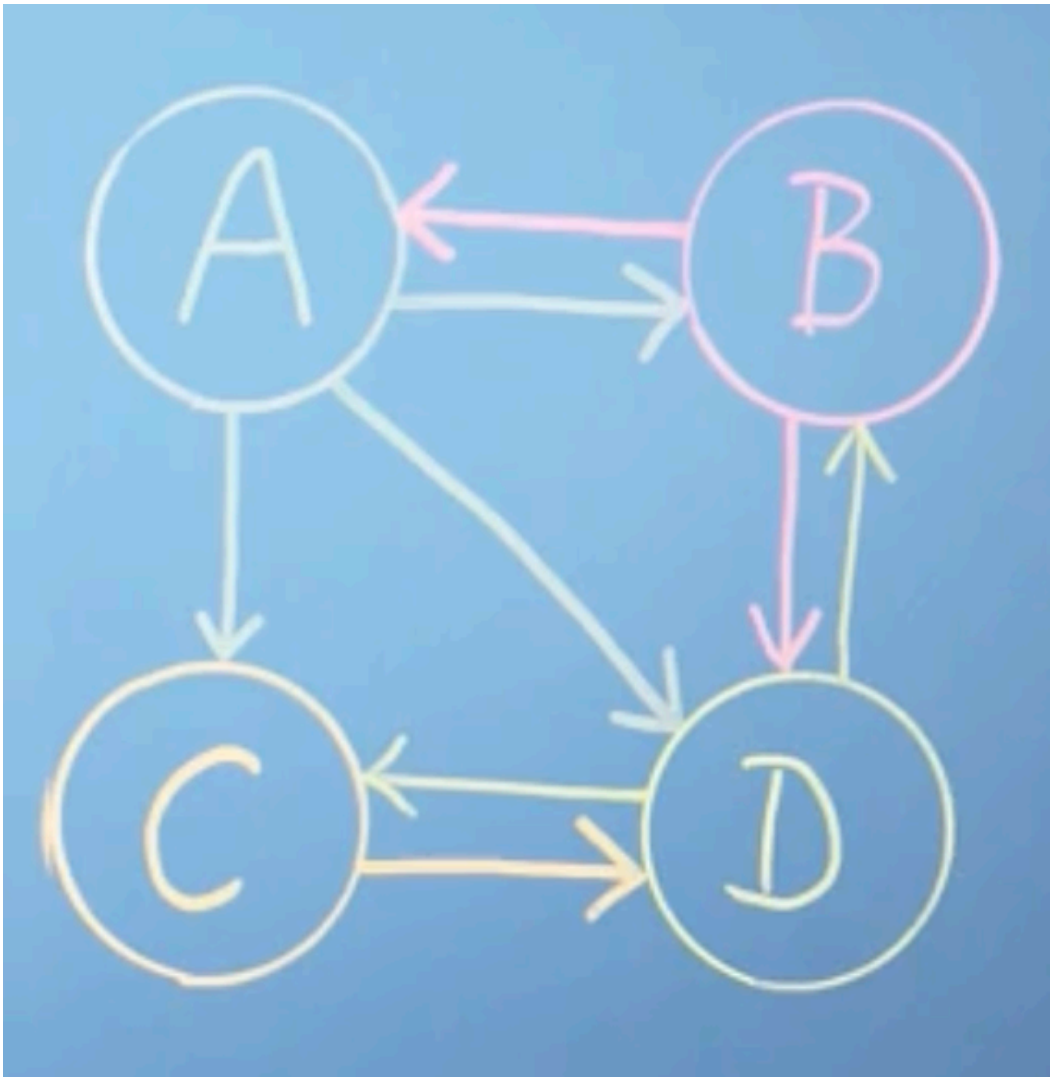
$$T^2 = \begin{bmatrix} (1)(1) + (4)(0) & (1)(-1) + (4)(1) \\ (0)(1) + (4)(0) & (0)(-1) + (4)(1) \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & 4 \end{bmatrix}$$

- Both methods yield the same result, but the diagonalization approach highlights a powerful conceptual tool that becomes essential for larger matrices and higher powers. While this example is simple, the method shines when dealing with high-dimensional problems in fields like machine learning.

E5. Page Rank: A Practical Application of Egienproblems

- The PageRank algorithm, developed by Google founders **Larry Page** and **Sergey Brin**, revolutionized search engine technology by ranking webpages based on their importance. The central idea is that a page is more important if it receives links from other important pages. This seemingly circular problem can be elegantly solved using eigen theory.

1. Constructing the Link Matrix (L)



To model a network of webpages (A, B, C, D), we can represent the links between them using a link matrix L.

- Each column of the matrix represents the outgoing links from a specific page.
- Each entry in a column is a probability, calculated by dividing the number of links to a specific page by the total number of links on the originating page.
- The sum of the entries in each column must equal 1.

For the given example:

- Page A links to B, C, and D (3 links).
- Page B links to A and C (2 links).
- Page C links to A and B (2 links).
- Page D links to B and C (2 links).

The corresponding link vectors are:

- $L_A = \begin{bmatrix} 0 \\ 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$ (links from A)

- $L_B = \begin{bmatrix} 1/2 \\ 0 \\ 0 \\ 1/2 \end{bmatrix}$ (links from A)
- $L_C = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ (links from A)
- $L_D = \begin{bmatrix} 0 \\ 1/2 \\ 1/2 \\ 0 \end{bmatrix}$ (links from A)

We then form the link matrix L by using these vectors as its columns.

$$L = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 1 & 0 \end{bmatrix}$$

2. The Rank Equation

   The importance of each page is stored in a vector r. The rank of a page (e.g., page A) is the sum of the ranks of all pages that link to it, weighted by the link probability. This can be expressed as a summation for a single page:

   $$r_A = \sum_{j=1}^{n} L_{A,j} r_j$$

   This states that the rank of page A (r_A) is the sum of the ranks of all other pages (r_j) that link to it, weighted by the probability of those links ($L_{A,j}$).

   - r_A: The rank of page A.
   - $L_{A,j}$: The probability of a link from page j to page A. This corresponds to the element in the A-th row and j-th column of the link matrix L.
   - r_j: The rank of page j.

   This system of equations for all pages can be expressed as a simple matrix multiplication:

   $$r = Lr$$

   This is the definition of an **eigenproblem**, which implies that r is an eigenvector of matrix L with a corresponding **eigenvalue of 1**.

3. The Solution and Damping Factor

   The **Power Method** is used to solve for r iteratively. We start with an initial guess, typically assuming all ranks are equal, and then repeatedly apply the transformation:

   $$r^{i+1} = L_r^i$$

This process converges to the PageRank vector.

A crucial addition is the **damping factor**, d, which is a value between 0 and 1. This factor, typically 0.85, accounts for a user randomly typing in a new address instead of clicking a link. It adds an additional term to our iterative formula:

$$r^{i+1} = dL_r^i + \frac{(1-d)}{n}$$

The damping factor ensures a stable and fast convergence, while the use of a sparse matrix (where most entries are zero) allows for efficient calculations on a massive scale.

Python code: Page Rank

In [190…
```python
import numpy as np
import numpy.linalg as la

def pageRank(linkMatrix, d) :
    n = linkMatrix.shape[0]
    M = d * linkMatrix + (1-d)/n * np.ones([n, n])
    r = 100 * np.ones(n) / n

    lastR = r
    r = M @ r

    while la.norm(lastR - r) > 0.01 :
        lastR = r
        r = M @ r

    return r
```

In [191…
```python
linkMatrix = np.array([
    [0, 0, 1, 0.5],
    [0.5, 0, 0, 0],
    [0.5, 1, 0, 0.5],
    [0, 0, 0, 0]
])

d = 0.85

rank_vector = pageRank(linkMatrix, d)
rank_vector
```

Out[191…
```
array([37.97275017, 19.8871091 , 38.39014073,  3.75      ])
```