Linear Algebra

Math - Linear Algebra

Linear Algebra is the branch of mathematics that studies vector spaces and linear transformations between vector spaces, such as rotating a shape, scaling it up or down, translating it (ie. moving it), etc.

Machine Learning relies heavily on Linear Algebra, so it is essential to understand what vectors and matrices are, what operations you can perform with them, and how they can be useful.

Before we start, let's make sure the following command runs without any errors to ensure that the interface you're working on works well with python2 and python3:

```
from __future__ import division, print_function, unicode_literals
```

Vectors

Definition

A vector is a quantity defined by a magnitude and a direction. For example, a rocket's velocity is a 3-dimensional vector: its magnitude is the speed of the rocket, and its direction is (hopefully) up. A vector can be represented by an array of numbers called *scalars*. Each scalar corresponds to the magnitude of the vector with regards to each dimension.

For example, say the rocket is going up at a slight angle: it has a vertical speed of 5,000 m/s, and also a slight speed towards the East at 10 m/s, and a slight speed towards the North at 50 m/s. The rocket's velocity may be represented by the following vector:

$$velocity = \begin{pmatrix} 10 \\ 50 \\ 5000 \end{pmatrix}$$

Note: by convention vectors are generally presented in the form of columns. Also, vector names are generally lowercase to distinguish them from matrices (which we will discuss below) and in bold (when possible) to distinguish them from simple scalar values such as

meters_per_second=5026

A list of N numbers may also represent the coordinates of a point in an N-dimensional space, so it is quite frequent to represent vectors as simple points instead of arrows. A vector with 1 element may be represented as an arrow or a point on an axis, a vector with 2 elements is an arrow or a point on a plane, a vector with 3 elements is an arrow or point in space, and a vector with N elements is an arrow or a point in an N-dimensional space… which most people find hard to imagine.

Purpose

Vectors have many purposes in Machine Learning, most notably to represent observations and predictions. For example, say we built a Machine Learning system to classify videos into 3 categories (good, spam, clickbait) based on what we know about them. For each video, we would have a vector representing what we know about it, such as:

$$video = \begin{pmatrix} 10.5 \\ 5.2 \\ 3.25 \\ 7.0 \end{pmatrix}$$

This vector could represent a video that lasts 10.5 minutes, but only 5.2% viewers watch for more than a minute, it gets 3.25 views per day on average, and it was flagged 7 times as spam. As you can see, each axis may have a different meaning.

Based on this vector, our Machine Learning system may predict that there is an 80% probability that it is a spam video, 18% that it is clickbait, and 2% that it is a good video. This could be represented as the following vector:

$$class\,probabilities = \begin{pmatrix} 0.80 \\ 0.18 \\ 0.02 \end{pmatrix}$$

Vectors in python

In python, a vector can be represented in many ways, the simplest being a regular python list of numbers:

In [1]:
```
[10.5, 5.2, 3.25, 7.0]
```
Out[1]:
```
[10.5, 5.2, 3.25, 7.0]
```

Since we plan to do quite a lot of scientific calculations, it is much better to use NumPy's `ndarray`, which provides a lot of convenient and optimized implementations of essential mathematical operations on vectors. For example:

In [2]:
```
import numpy as np
video = np.array([10.5, 5.2, 3.25, 7.0])
video
```
Out[2]:
```
array([10.5 ,  5.2 ,  3.25,  7.  ])
```
The size of a vector can be obtained using the `size` attribute:

In [3]:
```
video.size
```
Out[3]:
```
4
```

The ith element (also called *entry* or *item*) of a vector v is noted v_i.

Note that indices in mathematics generally start at 1, but in programming they usually start at 0. So to access $video_3$ programmatically, we would write:

In [4]:
```
video[2]  # 3rd element
```
Out[4]:
```
3.25
```

Plotting vectors

To plot vectors we will use matplotlib, so let's start by importing it:

In [5]:
```
%matplotlib inline
```

```
import matplotlib.pyplot as plt
```

2D vectors

Let's create a couple very simple 2D vectors to plot:

In [5]:

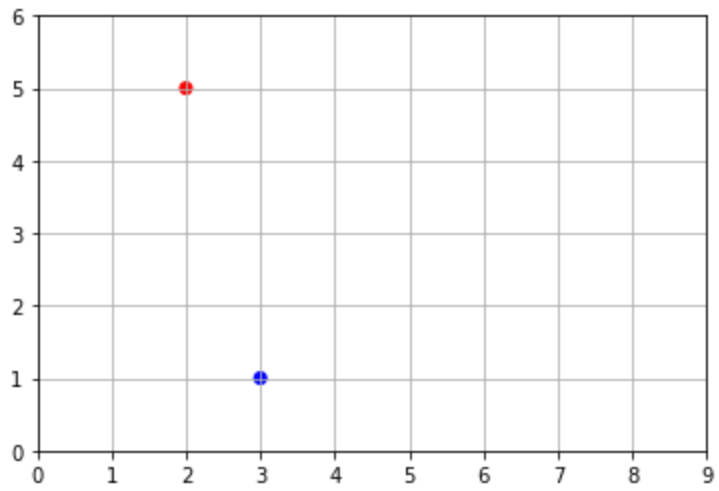
```
u = np.array([2, 5])
v = np.array([3, 1])
```

These vectors each have 2 elements, so they can easily be represented graphically on a 2D graph, for example as points:

In [6]:

```
x_coords, y_coords = zip(u, v)
plt.scatter(x_coords, y_coords, color=["r","b"])
plt.axis([0, 9, 0, 6])
plt.grid()
plt.show()
```

Out[6]:



Vectors can also be represented as arrows. Let's create a small convenience function to draw nice arrows:

In [7]:

```
def plot_vector2d(vector2d, origin=[0, 0], **options):
    return plt.arrow(origin[0], origin[1], vector2d[0], vector2d[1],
              head_width=0.2, head_length=0.3, length_includes_head=True, **options)
```

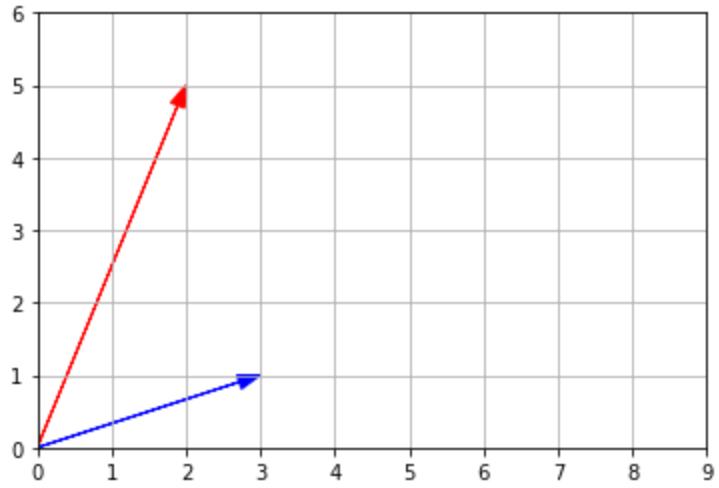
Now let's draw the vectors **u** and **v** as arrows:

In [8]:

```
plot_vector2d(u, color="r")
plot_vector2d(v, color="b")
```

```
plt.axis([0, 9, 0, 6])
plt.grid()
plt.show()
```

Out[8]:



3D vectors

Plotting 3D vectors is also relatively straightforward. First let's create two 3D vectors:

In [9]:

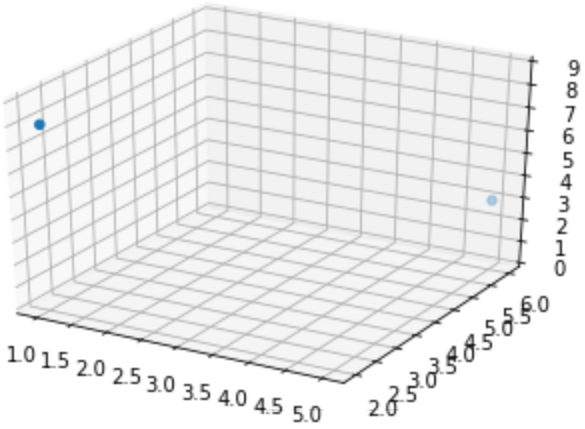
```
a = np.array([1, 2, 8])
b = np.array([5, 6, 3])
```

Now let's plot them using matplotlib's `Axes3D`:

In [10]:

```
from mpl_toolkits.mplot3d import Axes3D
subplot3d = plt.subplot(111, projection='3d')
x_coords, y_coords, z_coords = zip(a,b)
subplot3d.scatter(x_coords, y_coords, z_coords)
subplot3d.set_zlim3d([0, 9])
plt.show()
```

Out[10]:

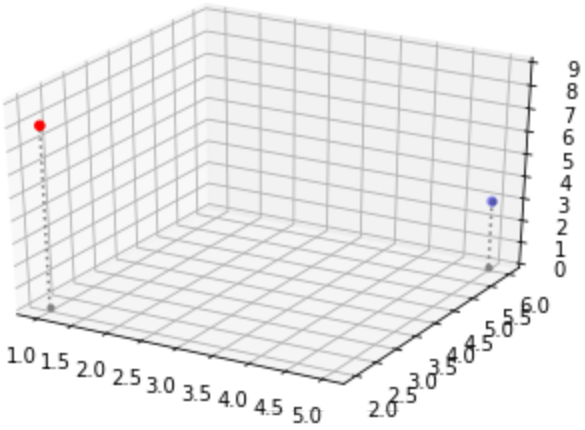


It is a bit hard to visualize exactly where in space these two points are, so let's add vertical lines. We'll create a small convenience function to plot a list of 3d vectors with vertical lines attached:

In [11]:

```
def plot_vectors3d(ax, vectors3d, z0, **options):
    for v in vectors3d:
        x, y, z = v
        ax.plot([x,x], [y,y], [z0, z], color="gray", linestyle='dotted', marker=".")
    x_coords, y_coords, z_coords = zip(*vectors3d)
    ax.scatter(x_coords, y_coords, z_coords, **options)

subplot3d = plt.subplot(111, projection='3d')
subplot3d.set_zlim([0, 9])
plot_vectors3d(subplot3d, [a,b], 0, color=("r","b"))
plt.show()
```

Out[11]:



Norm

The norm of a vector u, noted $\| u \|$, is a measure of the length (a.k.a. the magnitude) of

u . There are multiple possible norms, but the most common one (and the only one we will discuss here) is the Euclidean norm, which is defined as:

$$\| u \| = \sqrt{\sum_i u_i^2}$$

We could implement this easily in pure python, recalling that $\sqrt{x} = x^{1/2}$

In [12]:

```
def vector_norm(vector):
    squares = [element**2 for element in vector]
    return sum(squares)**0.5
print("||", u, "|| =")
vector_norm(u)
```

|| [2 5] || =

Out[12]:

5.385164807134504

However, it is much more efficient to use NumPy's norm function, available in the linalg (**Lin**ear **Alg**ebra) module:

In [13]:

```
import numpy.linalg as LA
LA.norm(u)
```

Out[13]:

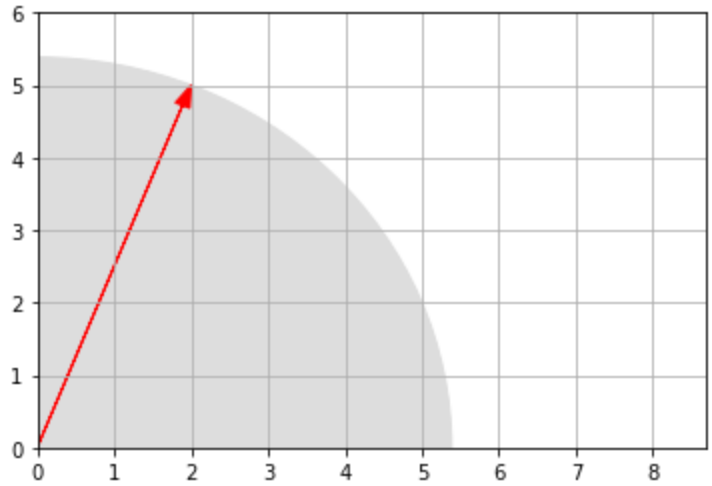
5.385164807134504

Let's plot a little diagram to confirm that the length of vector v is indeed ≈5.4:

In [14]:

```
radius = LA.norm(u)
plt.gca().add_artist(plt.Circle((0,0), radius, color="#DDDDDD"))
plot_vector2d(u, color="red")
plt.axis([0, 8.7, 0, 6])
plt.grid()
plt.show()
```

Out[14]:



Looks about right!

Addition

Vectors of the same size can be added together. Addition is performed *elementwise*:

In [15]:

```
print(" ", u)
print("+", v)
print("-"*10)
u + v
```

Out[15]:

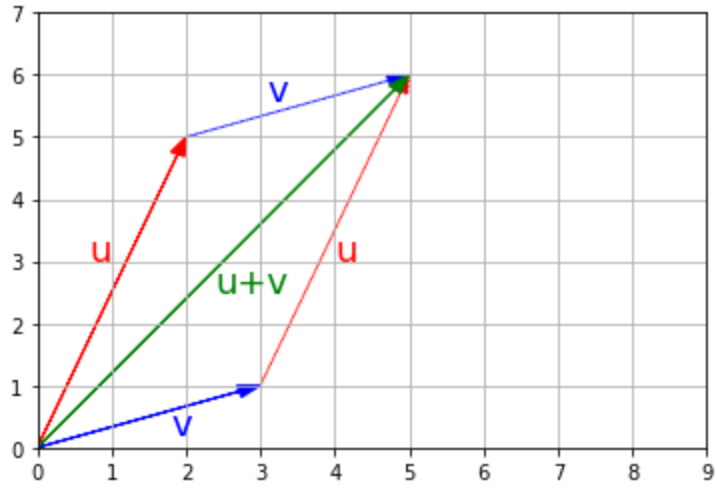
```
 [2 5]
+ [3 1]
----------
array([5, 6])
```

Let's look at what vector addition looks like graphically:

In [16]:

```
plot_vector2d(u, color="r")
plot_vector2d(v, color="b")
plot_vector2d(v, origin=u, color="b", linestyle="dotted")
plot_vector2d(u, origin=v, color="r", linestyle="dotted")
plot_vector2d(u+v, color="g")
plt.axis([0, 9, 0, 7])
plt.text(0.7, 3, "u", color="r", fontsize=18)
plt.text(4, 3, "u", color="r", fontsize=18)
plt.text(1.8, 0.2, "v", color="b", fontsize=18)
plt.text(3.1, 5.6, "v", color="b", fontsize=18)
```

```
plt.text(2.4, 2.5, "u+v", color="g", fontsize=18)
plt.grid()
plt.show()
```
Out[16]:

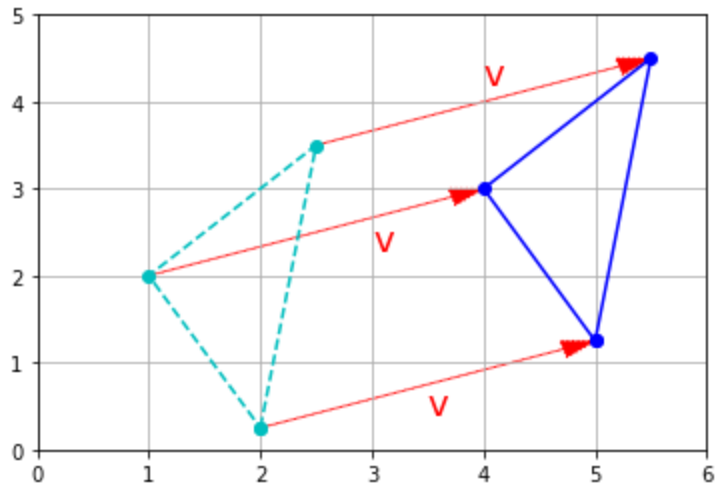


Vector addition is **commutative**, meaning that $u+v=v+u$. You can see it in the previous image: following u *then* v leads to the same point as following v *then* u.

Vector addition is also **associative**, meaning that $u+(v+w)=(u+v)+w$. If you have a shape defined by a number of points (vectors), and you add a vector v to all of these points, then the whole shape gets shifted by v. This is called a geometric translation:

In [17]:
```
t1 = np.array([2, 0.25])
t2 = np.array([2.5, 3.5])
t3 = np.array([1, 2])
x_coords, y_coords = zip(t1, t2, t3, t1)
plt.plot(x_coords, y_coords, "c--", x_coords, y_coords, "co")
plot_vector2d(v, t1, color="r", linestyle=":")
plot_vector2d(v, t2, color="r", linestyle=":")
plot_vector2d(v, t3, color="r", linestyle=":")
t1b = t1 + v
t2b = t2 + v
t3b = t3 + v
x_coords_b, y_coords_b = zip(t1b, t2b, t3b, t1b)
plt.plot(x_coords_b, y_coords_b, "b-", x_coords_b, y_coords_b, "bo")
plt.text(4, 4.2, "v", color="r", fontsize=18)
plt.text(3, 2.3, "v", color="r", fontsize=18)
plt.text(3.5, 0.4, "v", color="r", fontsize=18)
plt.axis([0, 6, 0, 5])
```

```
plt.grid()
plt.show()
```
Out[17]:



Finally, subtracting a vector is like adding the opposite vector.

Multiplication by a scalar

Vectors can be multiplied by scalars. All elements in the vector are multiplied by that number, for example:

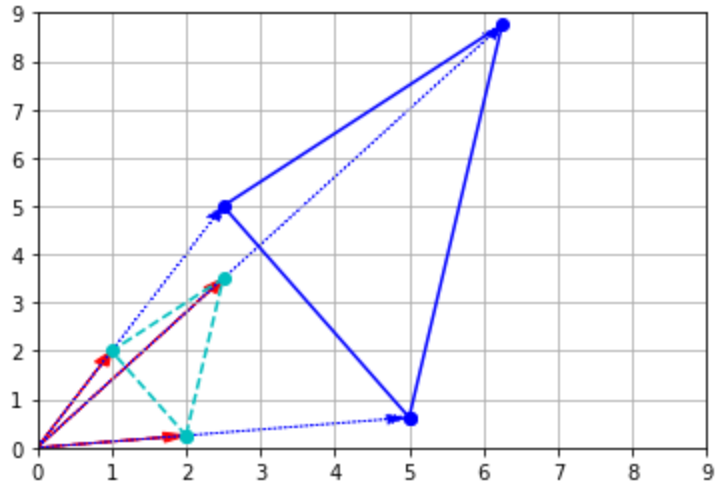
In [18]:
```
print("1.5 *", u, "=")
1.5 * u
```
Out[19]:
```
1.5 * [2 5] =
array([3. , 7.5])
```

Graphically, scalar multiplication results in changing the scale of a figure, hence the name *scalar*. The distance from the origin (the point at coordinates equal to zero) is also multiplied by the scalar. For example, let's scale up by a factor of `k = 2.5`:

In [20]:
```
k = 2.5
t1c = k * t1
t2c = k * t2
t3c = k * t3
plt.plot(x_coords, y_coords, "c--", x_coords, y_coords, "co")
plot_vector2d(t1, color="r")
plot_vector2d(t2, color="r")
plot_vector2d(t3, color="r")
x_coords_c, y_coords_c = zip(t1c, t2c, t3c, t1c)
```

```
plt.plot(x_coords_c, y_coords_c, "b-", x_coords_c, y_coords_c, "bo")
plot_vector2d(k * t1, color="b", linestyle=":")
plot_vector2d(k * t2, color="b", linestyle=":")
plot_vector2d(k * t3, color="b", linestyle=":")
plt.axis([0, 9, 0, 9])
plt.grid()
plt.show()
```

Out[20]:



As you might guess, dividing a vector by a scalar is equivalent to multiplying by its inverse:

$$\frac{u}{\lambda} = \frac{1}{\lambda} \times u$$

Scalar multiplication is **commutative**: λ×u=u×λ

It is also **associative**: λ1×(λ2×u)=(λ1×λ2)×u.

Finally, it is **distributive** over addition of vectors:

λ×(u+v)=λ×u+λ×v.

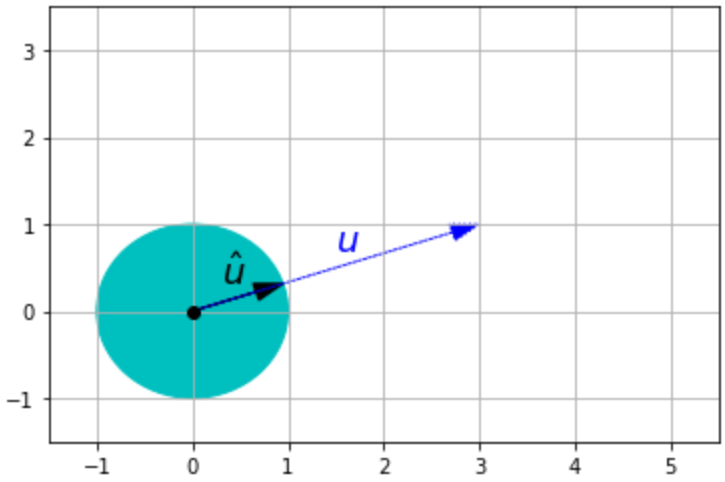
Zero, unit and normalized vectors

- A **zero-vector** is a vector full of 0s.
- A **unit vector** is a vector with a norm equal to 1.
- The normalized vector of a non-null vector u, noted $\widehat{u}$, is the unit vector that points in the same direction as u. It is equal to: $\widehat{u} = \frac{u}{\|u\|}$

In [21]:

```
plt.gca().add_artist(plt.Circle((0,0),1,color='c'))
plt.plot(0, 0, "ko")
plot_vector2d(v / LA.norm(v), color="k")
plot_vector2d(v, color="b", linestyle=":")
plt.text(0.3, 0.3, "$\hat{u}$", color="k", fontsize=18)
plt.text(1.5, 0.7, "$u$", color="b", fontsize=18)
plt.axis([-1.5, 5.5, -1.5, 3.5])
plt.grid()
plt.show()
```

Out[21]:



Dot product

Definition

The dot product (also called *scalar product* or *inner product* in the context of the Euclidean space) of two vectors u and v is a useful operation that comes up fairly often in linear algebra. It is noted $u \cdot v$ or sometimes $\langle u|v \rangle$ or $(u|v)$, and it is defined as:

$u \cdot v = \| u \| \times \| v \| \times \cos(\theta)$

where θ is the angle between u and v. Another way to calculate the dot product is

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

In python

The dot product is pretty simple to implement:

In [22]:
```
def dot_product(v1, v2):
    return sum(v1i * v2i for v1i, v2i in zip(v1, v2))
dot_product(u, v)
```

Out[22]:
11

But a *much* more efficient implementation is provided by NumPy with the `dot` function:

In [23]:
```
np.dot(u,v)
```
Out[23]:
11

Equivalently, you can use the `dot` method of `ndarrays`:

In [24]:
```
u.dot(v)
```
Out[24]:
11

Caution: the `*` operator will perform an *elementwise* multiplication, *NOT* a dot product:

In [26]:
```
print("  ",u)
print("* ",v, "(NOT a dot product)")
print("-"*10)
u * v
```
Out[26]:
```
  [2 5]
*  [3 1] (NOT a dot product)
----------
array([6, 5])
```

Main properties

- The dot product is **commutative**: $u \cdot v = v \cdot u$
- The dot product is only defined between two vectors, not between a scalar and a vector. This means that we cannot chain dot products: for example, the expression $u \cdot v \cdot w$ is not defined since $u \cdot v$ is a scalar and w is a vector.
- This also means that the dot product is **NOT associative**: $(u \cdot v) \cdot w \neq u \cdot (v \cdot w)$ since neither are defined.
- However, the dot product is **associative with regards to scalar multiplication**: $\lambda \times (u \cdot v) = (\lambda \times u) \cdot v = u \cdot (\lambda \times v)$

- Finally, the dot product is **distributive** over addition of vectors: $u \cdot (v+w)=u \cdot v+u \cdot w$

Calculating the angle between vectors

One of the many uses of the dot product is to calculate the angle between two non-zero vectors. Looking at the dot product definition, we can deduce the following formula:

$$\theta = arc\cos\left(\frac{u \cdot v}{\|u\| \times \|v\|}\right)$$

Note that if $u \cdot v=0$, it follows that $\theta=\pi/2$. In other words, if the dot product of two non-null vectors is zero, it means that they are orthogonal.Let's use this formula to calculate the angle between u and v (in radians):

In [27]:
```
def vector_angle(u, v):
    cos_theta = u.dot(v) / LA.norm(u) / LA.norm(v)
    return np.arccos(np.clip(cos_theta, -1, 1))

theta = vector_angle(u, v)
print("Angle =", theta, "radians")
print("      =", theta * 180 / np.pi, "degrees")
```
Out[27]:
```
Angle = 0.8685393952858895 radians
      = 49.76364169072618 degrees
```

Note: due to small floating point errors, `cos_theta` may be very slightly outside of the $[-1,1]$ interval, which would make `arccos` fail. This is why we clipped the value within the range, using NumPy's `clip` function.

Projecting a point onto an axis

The dot product is also very useful to project points onto an axis. The projection of vector

v onto u's axis is given by this formula:

$$proj_u v = \frac{u \cdot v}{\|u\|^2} \times u$$

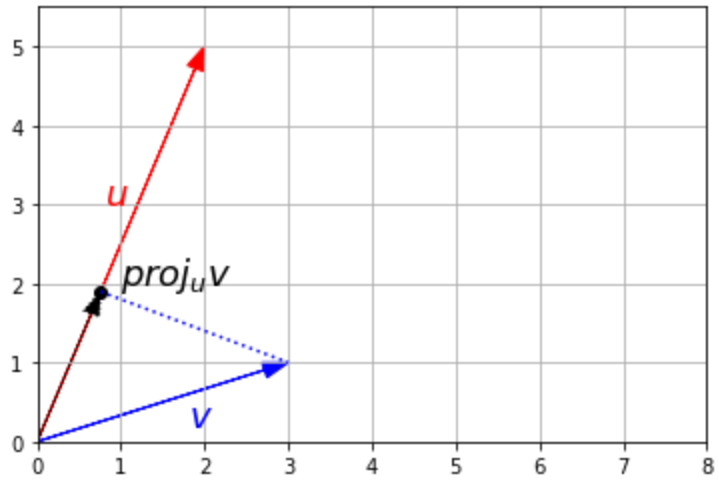
Which is equivalent to:

$$proj_u v = (v \cdot \hat{u}) \times \hat{u}$$

In [28]:

```
u_normalized = u / LA.norm(u)
proj = v.dot(u_normalized) * u_normalized

plot_vector2d(u, color="r")
plot_vector2d(v, color="b")

plot_vector2d(proj, color="k", linestyle=":")
plt.plot(proj[0], proj[1], "ko")

plt.plot([proj[0], v[0]], [proj[1], v[1]], "b:")

plt.text(1, 2, "$proj_u v$", color="k", fontsize=18)
plt.text(1.8, 0.2, "$v$", color="b", fontsize=18)
plt.text(0.8, 3, "$u$", color="r", fontsize=18)

plt.axis([0, 8, 0, 5.5])
plt.grid()
plt.show()
```

Out[28]:



Matrices

A matrix is a rectangular array of scalars (ie. any number: integer, real or complex) arranged in rows and columns, for example:

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

You can also think of a matrix as a list of vectors: the previous matrix contains either 2 horizontal 3D vectors or 3 vertical 2D vectors.

Matrices are convenient and very efficient to run operations on many vectors at a time. We will also see that they are great at representing and performing linear transformations such rotations, translations and scaling.

Matrices in python

In python, a matrix can be represented in various ways. The simplest is just a list of python lists:

In [29]:

```
[
    [10, 20, 30],
    [40, 50, 60]
]
```

Out[29]:

```
[[10, 20, 30], [40, 50, 60]]
```

A much more efficient way is to use the NumPy library which provides optimized implementations of many matrix operations:

In [30]:

```
A = np.array([
    [10,20,30],
    [40,50,60]
])
A
```

Out[30]:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

By convention matrices generally have uppercase names, such as A.

In the rest of this tutorial, we will assume that we are using NumPy arrays (type `ndarray`) to represent matrices.

Size

The size of a matrix is defined by its number of rows and number of columns. It is noted $rows \times columns$.For example, the matrix A above is an example of a 2×3 matrix: 2 rows, 3 columns.

Caution: a 3×2 matrix would have 3 rows and 2 columns.

To get a matrix's size in NumPy:

In [31]:

A.shape
Out[31]:
(2, 3)
Caution: the size attribute represents the number of elements in the ndarray, not the matrix's size:
In [32]:
A.size
Out[32]:
6

Element indexing

The number located in the ith row, and jth column of a matrix X is sometimes noted $X_{i,j}$ or X_{ij}, but there is no standard notation, so people often prefer to explicitly name the elements, like this: "*let* $X = (x_{i,j})\ 1 \le i \le m, 1 \le j \le n$. This means that X is equal to:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & x_{m,3} & \cdots & x_{m,n} \end{bmatrix}$$

However we will use the $X_{i,j}$ notation, as it matches fairly well NumPy's notation. Note that in math indices generally start at 1, but in programming they usually start at 0. So to access $A_{2,3}$ programmatically, we need to write this:
In [33]:
A[1,2] *# 2nd row, 3rd column*
Out[33]:
60

The ith row vector is sometimes noted M_i or $M_{i,\cdot}$
Mi,∗, but again there is no standard notation so people often prefer to explicitly define their own names, for example: "*let* $\boldsymbol{x}_i$ *be the* ith *row vector of matrix* X". We will use the $M_{i,\cdot}$, for the same reason as above. For example, to access $A_{2,\cdot}$(ie.A's 2nd row vector):
In [34]:
A[1, :] *# 2nd row vector (as a 1D array)*

Out[34]:
array([40, 50, 60])

Similarly, the jth column vector is sometimes noted M^j or $M_{*,j}$, but there is no standard notation. We will use $M_{*,j}$. For example, to access $A_{*,3}$(ie. A's 3rd column vector):

In [35]:
```
A[:, 2]  # 3rd column vector (as a 1D array)
```
Out[35]:
array([30, 60])

Note that the result is actually a one-dimensional NumPy array: there is no such thing as a *vertical* or *horizontal* one-dimensional array. If you need to actually represent a row vector as a one-row matrix (ie. a 2D NumPy array), or a column vector as a one-column matrix, then you need to use a slice instead of an integer when accessing the row or column, for example:

In [36]:
```
A[1:2, :]  # rows 2 to 3 (excluded): this returns row 2 as a one-row matrix
```
Out[36]:
array([[40, 50, 60]])

In [37]:
```
A[:, 2:3]  # columns 3 to 4 (excluded): this returns column 3 as a one-column matrix
```
Out[37]:
array([[30],
       [60]])

Square, triangular, diagonal and identity matrices

A **square matrix** is a matrix that has the same number of rows and columns, for example a

3×3 matrix:

$$\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}$$

An **upper triangular matrix** is a special kind of square matrix where all the elements *below* the main diagonal (top-left to bottom-right) are zero, for example:

$$\begin{bmatrix} 4 & 9 & 2 \\ 0 & 5 & 7 \\ 0 & 0 & 6 \end{bmatrix}$$

Similarly, a **lower triangular matrix** is a square matrix where all elements *above* the main diagonal are zero, for example:

$$\begin{bmatrix} 4 & 0 & 0 \\ 3 & 5 & 0 \\ 8 & 1 & 6 \end{bmatrix}$$

A **triangular matrix** is one that is either lower triangular or upper triangular.
A matrix that is both upper and lower triangular is called a **diagonal matrix**, for example:

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

You can construct a diagonal matrix using NumPy's diag function:

In [38]:
```
np.diag([4, 5, 6])
```
Out[38]:
```
array([[4, 0, 0],
       [0, 5, 0],
       [0, 0, 6]])
```
If you pass a matrix to the diag function, it will happily extract the diagonal values:

In [39]:
```
D = np.array([
       [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
   ])
np.diag(D)
```
Out[39]:
```
array([1, 5, 9])
```
Finally, the **identity matrix** of size n, noted I_n

In, is a diagonal matrix of size n×n with 1's in the main diagonal, for example I_3:

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

Numpy's eye function returns the identity matrix of the desired size:

In [40]:
```
np.eye(3)
```
Out[40]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

The identity matrix is often noted simply
I (instead of I_n) when its size is clear given the context. It is called the *identity* matrix because multiplying a matrix with it leaves the matrix unchanged as we will see below.

Adding matrices

If two matrices Q and R have the same size m×n, they can be added together. Addition is performed *elementwise*: the result is also a m×n matrix S where each element is the sum of the elements at the corresponding position:

$$S_{i,j} = Q_{i,j} + R_{i,j}$$

$$S = \begin{bmatrix} Q11+R11 & Q12+R12 & Q13+R13 & \cdots & Q1n+R1n \\ Q21+R21 & Q22+R22 & Q23+R23 & \cdots & Q2n+R2n \\ Q31+R31 & Q32+R32 & Q33+R33 & \cdots & Q3n+R3n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Qm1+Rm1 & Qm2+Rm2 & Qm3+Rm3 & \cdots & Qmn+Rmn \end{bmatrix}$$

For example, let's create a 2×3 matrix B and compute A+B:

In [41]:
```
B = np.array([[1,2,3], [4, 5, 6]])
B
```
Out[41]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```
In [42]:
```
A
```
Out[42]:
```
array([[10, 20, 30],
       [40, 50, 60]])
```

In [43]:
```
A + B
```
Out[43]:
```
array([[11, 22, 33],
       [44, 55, 66]])
```
Addition is *commutative*, meaning that A+B=B+A:

In [44]:
```
B + A
```

Out[44]:

```
array([[11, 22, 33],
       [44, 55, 66]])
```

It is also *associative*, meaning that A+(B+C)=(A+B)+C:

In [45]:

```
C = np.array([[100,200,300], [400, 500, 600]])
A + (B + C)
```

Out[45]:

```
array([[111, 222, 333],
       [444, 555, 666]])
```

In [46]:

```
(A + B) + C
```

Out[46]:

```
array([[111, 222, 333],
       [444, 555, 666]])
```

Scalar multiplication

A matrix M can be multiplied by a scalar λ. The result is noted λM, and it is a matrix of the same size as M with all elements multiplied by λ:

$$\lambda M = \begin{bmatrix} \lambda \times M11 & \lambda \times M12 & \lambda \times M13 & \cdots & \lambda \times M1n \\ \lambda \times M21 & \lambda \times M22 & \lambda \times M23 & \cdots & \lambda \times M2n \\ \lambda \times M31 & \lambda \times M32 & \lambda \times M33 & \cdots & \lambda \times M3n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda \times Mm1 & \lambda \times Mm2 & \lambda \times Mm3 & \cdots & \lambda \times Mmn \end{bmatrix}$$

A more concise way of writing this is:

$$(\lambda M)_{i,j} = \lambda (M)_{i,j}$$

In NumPy, simply use the `*` operator to multiply a matrix by a scalar. For example:

In [47]:

```
2 * A
```

Out[47]:

```
array([[ 20,  40,  60],
       [ 80, 100, 120]])
```

Scalar multiplication is also defined on the right hand side, and gives the same result: Mλ=λM. For example:

In [48]:
```
A * 2
```
Out[48]:
```
array([[ 20,  40,  60],
       [ 80, 100, 120]])
```
This makes scalar multiplication **commutative**.It is also **associative**, meaning that

$\alpha(\beta M)=(\alpha\times\beta)M$, where α and β are scalars. For example:

In [49]:
```
2 * (3 * A)
```
Out[49]:
```
array([[ 60, 120, 180],
       [240, 300, 360]])
```
In [50]:
```
(2 * 3) * A
```
Out[50]:
```
array([[ 60, 120, 180],
       [240, 300, 360]])
```
Finally, it is **distributive over addition** of matrices, meaning that $\lambda(Q+R)=\lambda Q+\lambda R$:

In [51]:
```
2 * (A + B)
```
Out[51]:
```
array([[ 22,  44,  66],
       [ 88, 110, 132]])
```
In [52]:
```
2 * A + 2 * B
```
Out[52]:
```
array([[ 22,  44,  66],
       [ 88, 110, 132]])
```

Matrix multiplication

So far, matrix operations have been rather intuitive. But multiplying matrices is a bit more involved.

A matrix Q of size m×n can be multiplied by a matrix R of size n×q. It is noted simply as QR without a multiplication sign or dot. The result P is an m×q matrix where each element is computed as a sum of products:

$$P_{i,j} = \sum_{k=1}^{n} Q_{i,k} \times R_{k,j}$$

The element at position i,j in the resulting matrix is the sum of the products of elements in row

i of matrix Q by the elements in column j of matrix R.

P=

$$\begin{bmatrix} Q11R11+Q12R21+\cdots+Q1nRn1 & Q11R12+Q12R22+\cdots+Q1nRn2 & \cdots & Q11R1q+Q12R2q+\cdots+Q1nRnq \\ Q21R11+Q22R21+\cdots+Q2nRn1 & Q21R12+Q22R22+\cdots+Q2nRn2 & \cdots & Q21R1q+Q22R2q+\cdots+Q2nRnq \\ \vdots & \vdots & \ddots & \vdots \\ Qm1R11+Qm2R21+\cdots+QmnRn1 & Qm1R12+Qm2R22+\cdots+QmnRn2 & \cdots & Qm1R1q+Qm2R2q+\cdots+QmnRnq \end{bmatrix}$$

You may notice that each element $P_{i,j}$ is the dot product of the row vector $Q_{i,*}$ and the column vector $R_{*,j}$:

$$P_{i,j} = Q_{i,*} \cdot R_{*,j}$$

So we can rewrite P more concisely as:

$$P=\begin{bmatrix} Q_{1,*} \cdot R_{*,1} & Q_{1,*} \cdot R_{*,2} & \cdots & Q_{1,*} \cdot R_{*,q} \\ Q_{2,*} \cdot R_{*,1} & Q_{2,*} \cdot R_{*,2} & \cdots & Q_{2,*} \cdot R_{*,q} \\ \vdots & \vdots & \ddots & \vdots \\ Qm,* \cdot R*,1 & Q_{m,*} \cdot R_{*,2} & \cdots & Q_{m,*} \cdot R_{*,q} \end{bmatrix}$$

Let's multiply two matrices in NumPy, using ndarray's dot method:

$$E=AD=\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}\begin{bmatrix} 2 & 3 & 5 & 7 \\ 11 & 13 & 17 & 19 \\ 23 & 29 & 31 & 37 \end{bmatrix}=\begin{bmatrix} 930 & 1160 & 1320 & 1560 \\ 2010 & 2510 & 2910 & 3450 \end{bmatrix}$$

In [53]:

```
D = np.array([
       [ 2,  3,  5,  7],
       [11, 13, 17, 19],
       [23, 29, 31, 37]
    ])
E = A.dot(D)
E
```

Out[53]:

```
array([[ 930, 1160, 1320, 1560],
       [2010, 2510, 2910, 3450]])
```

Let's check this result by looking at one element, just to be sure: looking at $E_{2,3}$ for example, we need to multiply elements in A's 2^{nd} row by elements in D's 3rd column, and sum up these products:

In [54]:
```
40*5 + 50*17 + 60*31
```
Out[54]:
2910

In [55]:
```
E[1,2]  # row 2, column 3
```
Out[55]:
2910

Looks good! You can check the other elements until you get used to the algorithm.We multiplied a

2×3 matrix by a 3×4 matrix, so the result is a 2×4 matrix. The first matrix's number of columns has to be equal to the second matrix's number of rows. If we try to multiple

D by A, we get an error because D has 4 columns while A has 2 rows:

In [56]:
```
try:
    D.dot(A)
except ValueError as e:
    print("ValueError:", e)
```
Out[56]:
ValueError: shapes (3,4) and (2,3) not aligned: 4 (dim 1) != 2 (dim 0)

This illustrates the fact that **matrix multiplication is *NOT* commutative**: in general QR≠RQ

In fact, QR and RQ are only *both* defined if Q has size m×n and R has size n×m. Let's look at an example where both *are* defined and show that they are (in general) *NOT* equal:

In [57]:
```
F = np.array([
        [5,2],
        [4,1],
        [9,3]
    ])
A.dot(F)
```
Out[57]:
```
array([[400, 130],
       [940, 310]])
```
In [58]:
```
F.dot(A)
```
Out[58]:

```
array([[130, 200, 270],
       [ 80, 130, 180],
       [210, 330, 450]])
```

On the other hand, **matrix multiplication *is* associative**, meaning that Q(RS)=(QR)S. Let's create a 4×5 matrix G to illustrate this:

In [59]:

```
G = np.array([
       [8,  7,  4,  2,  5],
       [2,  5,  1,  0,  5],
       [9, 11, 17, 21,  0],
       [0,  1,  0,  1,  2]])
A.dot(D).dot(G)     # (AB)G
```

Out[59]:

```
array([[21640, 28390, 27320, 31140, 13570],
       [47290, 62080, 60020, 68580, 29500]])
```

In [60]:

```
A.dot(D.dot(G))     # A(BG)
```

Out[60]:

```
array([[21640, 28390, 27320, 31140, 13570],
       [47290, 62080, 60020, 68580, 29500]])
```

It is also ***distributive* over addition** of matrices, meaning that (Q+R)S=QS+RS. For example:

In [61]:

```
(A + B).dot(D)
```

Out[61]:

```
array([[1023, 1276, 1452, 1716],
       [2211, 2761, 3201, 3795]])
```

In [62]:

```
A.dot(D) + B.dot(D)
```

Out[62]:

```
array([[1023, 1276, 1452, 1716],
       [2211, 2761, 3201, 3795]])
```

The product of a matrix M by the identity matrix (of matching size) results in the same matrix M. More formally, if M is an m×n matrix, then:

$$M I_n = I_m M = M$$

This is generally written more concisely (since the size of the identity matrices is unambiguous given the context):

$$MI = IM = M$$

For example:

In [63]:
```
A.dot(np.eye(3))
```
Out[63]:
```
array([[10., 20., 30.],
       [40., 50., 60.]])
```
In [64]:
```
np.eye(2).dot(A)
```
Out[64]:
```
array([[10., 20., 30.],
       [40., 50., 60.]])
```
Caution: NumPy's * operator performs element wise multiplication, *NOT* a matrix multiplication:
In [65]:
```
A * B   # NOT a matrix multiplication
```
Out[65]:
```
array([[ 10,  40,  90],
       [160, 250, 360]])
```

The @ infix operator

Python 3.5 introduced the @ infix operator for matrix multiplication, and NumPy 1.10 added support for it. If you are using Python 3.5+ and NumPy 1.10+, you can simply write A @ D instead of A.dot(D), making your code much more readable (but less portable). This operator also works for vector dot products.
In [66]:
```
import sys
print("Python version: {}.{}.{}".format(*sys.version_info))
print("Numpy version:", np.version.version)

# Uncomment the following line if your Python version is ≥3.5
# and your NumPy version is ≥1.10:

A @ D
```
Out[66]:
```
Python version: 3.7.4
Numpy version: 1.17.2
array([[ 930, 1160, 1320, 1560],
       [2010, 2510, 2910, 3450]])
```
Note: Q @ R is actually equivalent to Q.__matmul__(R) which is implemented by NumPy as np.matmul(Q, R), not as Q.dot(R). The main difference is that matmul does not support scalar multiplication, while dot does, so you can write Q.dot(3), which is equivalent to Q * 3, but you cannot write Q @ 3 (more details).

Matrix transpose

The transpose of a matrix M is a matrix noted M^T such that the ith row in M^T is equal to the ith column in M:

$$A^T = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}^T = \begin{bmatrix} 10 & 40 \\ 20 & 50 \\ 30 & 60 \end{bmatrix}$$

In other words, $(A^T)_{i,j} = A_{j,i}$

Obviously, if M is an m×n matrix, then M^T is an n×m matrix.

Note: there are a few other notations, such as M^t,M',, or ${}^tM$.In NumPy, a matrix's transpose can be obtained simply using the T attribute:

In [67]:

```
A
```

Out[67]:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

In [68]:

```
A.T
```

Out[68]:

```
array([[10, 40],
       [20, 50],
       [30, 60]])
```

As you might expect, transposing a matrix twice returns the original matrix:

In [69]:

```
A.T.T
```

Out[69]:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

Transposition is distributive over addition of matrices, meaning that $(Q + R)^T = Q^T + R^T$. For example:

In [70]:

```
(A + B).T
```

Out[70]:

```
array([[11, 44],
```

```
       [22, 55],
       [33, 66]])
```
In [71]:
```
A.T + B.T
```
Out[71]:
```
array([[11, 44],
       [22, 55],
       [33, 66]])
```
Moreover, $(Q.R)^T = R^T.Q^T$. Note that the order is reversed. For example:

In [72]:
```
(A.dot(D)).T
```
Out[72]:
```
array([[ 930, 2010],
       [1160, 2510],
       [1320, 2910],
       [1560, 3450]])
```
In [73]:
```
D.T.dot(A.T)
```

Out[73]:
```
array([[ 930, 2010],
       [1160, 2510],
       [1320, 2910],
       [1560, 3450]])
```
A **symmetric matrix**

M is defined as a matrix that is equal to its transpose: XMT=M. This definition implies that it must be a square matrix whose elements are symmetric relative to the main diagonal, for example:

$$\begin{bmatrix} 17 & 22 & 27 & 49 \\ 22 & 29 & 36 & 0 \\ 27 & 36 & 45 & 2 \\ 49 & 0 & 2 & 99 \end{bmatrix}$$

The product of a matrix by its transpose is always a symmetric matrix, for example:

In [74]:
```
D.dot(D.T)
```
Out[74]:
```
array([[  87,  279,  547],
       [ 279,  940, 1860],
       [ 547, 1860, 3700]])
```

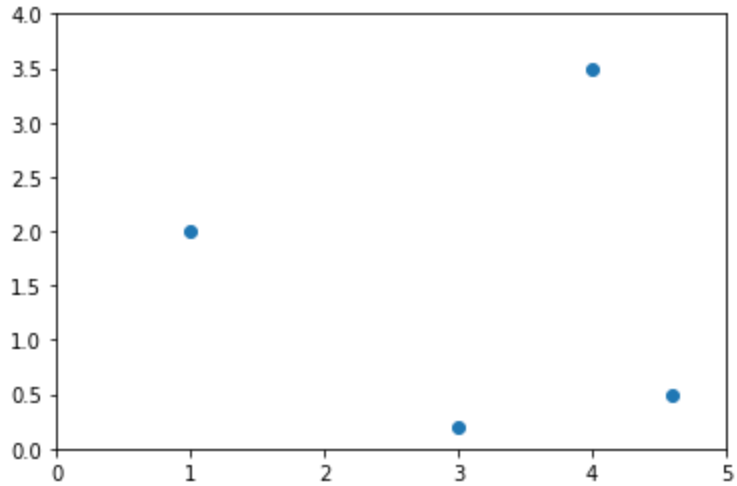
Plotting a matrix

We have already seen that vectors can be represented as points or arrows in N-dimensional space. Is there a good graphical representation of matrices? Well you can simply see a matrix as a list of vectors, so plotting a matrix results in many points or arrows. For example, let's create a 2×4 matrix `P` and plot it as points:

In [75]:

```
P = np.array([
        [3.0, 4.0, 1.0, 4.6],
        [0.2, 3.5, 2.0, 0.5]
    ])
x_coords_P, y_coords_P = P
plt.scatter(x_coords_P, y_coords_P)
plt.axis([0, 5, 0, 4])
plt.show()
```

Out[75]:



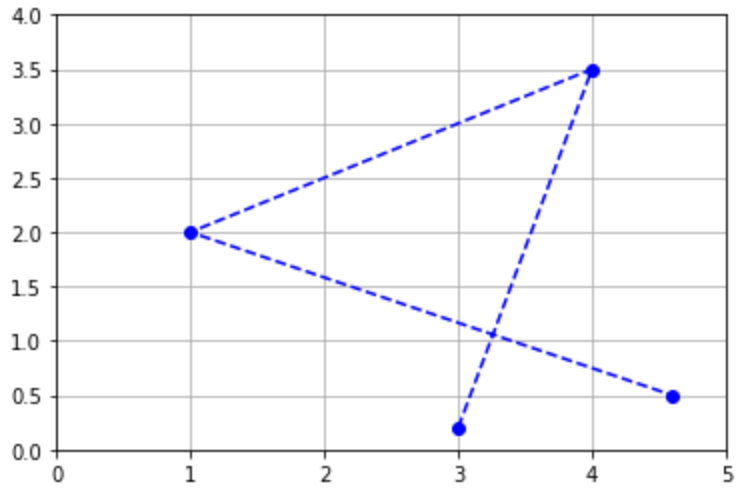
Of course we could also have stored the same 4 vectors as row vectors instead of column vectors, resulting in a 4×2 matrix (the transpose of P, in fact). It is really an arbitrary choice.

Since the vectors are ordered, you can see the matrix as a path and represent it with connected dots:

In [76]:

```
plt.plot(x_coords_P, y_coords_P, "bo")
plt.plot(x_coords_P, y_coords_P, "b--")
plt.axis([0, 5, 0, 4])
plt.grid()
plt.show()
```

Out[76]:



Matrix inverse

Now that we understand that a matrix can represent any linear transformation, a natural question is: can we find a transformation matrix that reverses the effect of a given transformation matrix F?

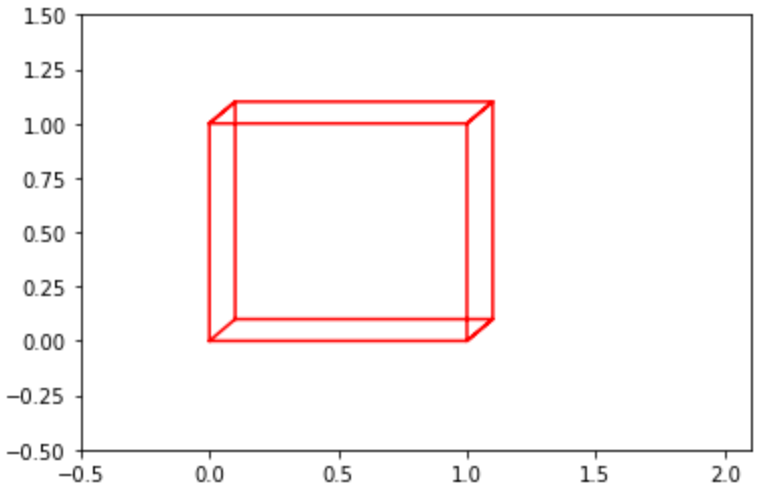
The answer is yes… sometimes! When it exists, such a matrix is called the **inverse** of F, and it is noted F^{-1}.

Only square matrices can be inverted. This makes sense when you think about it: if you have a transformation that reduces the number of dimensions, then some information is lost and there is no way that you can get it back. For example, say you use a 2×3 matrix to project a 3D object onto a plane. The result may look like this:

In [77]:

```
plt.plot([0, 0, 1, 1, 0, 0.1, 0.1, 0, 0.1, 1.1, 1.0, 1.1, 1.1, 1.0, 1.1, 0.1],
         [0, 1, 1, 0, 0, 0.1, 1.1, 1.0, 1.1, 1.1, 1.0, 1.1, 0.1, 0, 0.1, 0.1],
         "r-")
plt.axis([-0.5, 2.1, -0.5, 1.5])
plt.show()
```

Out[77]:



Looking at this image, it is impossible to tell whether this is the projection of a cube or the projection of a narrow rectangular object. Some information has been lost in the projection.

Even square transformation matrices can lose information.
This transformation matrix performs a projection onto the horizontal axis. Our polygon gets entirely flattened out so some information is entirely lost and it is impossible to go back to the original polygon using a linear transformation. In other words,
$F_{project}$ has no inverse. Such a square matrix that cannot be inverted is called a **singular matrix** (aka degenerate matrix). If we ask NumPy to calculate its inverse, it raises an exception.

In [78]:

```
#Inverse of an invertible matrix. Below function throws an error if the matrix is not invertible.
#Note: Matrix is invertible only if and only if the determinant of the square matrix is non zero.
"""Definition: A square matrix (A)n × n is said to be an invertible matrix if and only if there
exists another square matrix (B)n × n such that AB=BA=In"""
m = np.array([
       [2, -3],
       [4, -7]
   ])
np.linalg.inv(m)
```

Out[78]:

```
array([[ 3.5, -1.5],
       [ 2. , -1. ]])
```

Determinant

The determinant of a square matrix M, noted $\det(M)$ or $\det M$ or $|M|$ is a value that can be calculated from its elements $(M_{i,j})$ using various equivalent methods. One of the simplest methods is this recursive approach:

$$|M| = M_{1,1} \times |M_{(1,1)}| - M_{2,1} \times |M_{(2,1)}| + M_{3,1} \times |M_{(3,1)}| - M_{4,1} \times |M_{(4,1)}| + \cdots \pm M_{n,1} \times |M_{(n,1)}|$$

- Where $M_{(i,j)}$ is the matrix M without row i and column j.

For example, let's calculate the determinant of the following 3×3 matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Using the method above, we get:

$$|M| = 1 \times |\begin{bmatrix} 5 & 6 \\ 8 & 0 \end{bmatrix}| - 2 \times |\begin{bmatrix} 4 & 6 \\ 7 & 0 \end{bmatrix}| + 3 \times |\begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}|$$

Now we need to compute the determinant of each of these 2×2 matrices (these determinants are called **minors**):

$$\left|\begin{bmatrix} 5 & 6 \\ 8 & 0 \end{bmatrix}\right| = 5 \times 0 - 6 \times 8 = -48$$

$$\left|\begin{bmatrix} 4 & 6 \\ 7 & 0 \end{bmatrix}\right| = 4 \times 0 - 6 \times 7 = -42$$

$$\left|\begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix}\right| = 4 \times 8 - 5 \times 7 = -3$$

Now we can calculate the final result:

$$|M| = 1 \times (-48) - 2 \times (-42) + 3 \times (-3) = 27$$

To get the determinant of a matrix, you can call NumPy's `det` function in the `numpy.linalg` module:

In [79]:

```
M = np.array([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
```

```
])
LA.det(M)
```
Out[79]:
```
27.0
```
One of the main uses of the determinant is to *determine* whether a square matrix can be inverted or not: if the determinant is equal to 0, then the matrix *cannot* be inverted (it is a singular matrix), and if the determinant is not 0, then it *can* be inverted.

Eigenvectors and eigenvalues

An **eigenvector** of a square matrix M (also called a **characteristic vector**) is a non-zero vector that remains on the same line after transformation by the linear transformation associated with M. A more formal definition is any vector v such that:

$$M \cdot v = \lambda \times v$$

Where λ is a scalar value called the **eigenvalue** associated to the vector v. For example, any horizontal vector remains horizontal after applying the shear mapping (as you can see on the image above), so it is an eigenvector of M. A vertical vector ends up tilted to the right, so vertical vectors are *NOT* eigenvectors of M.

If we look at the squeeze mapping, we find that any horizontal or vertical vector keeps its direction (although its length changes), so all horizontal and vertical vectors are eigenvectors of $F_{squeeze}$. However, rotation matrices have no eigenvectors at all (except if the rotation angle is 0° or 180°, in which case all non-zero vectors are eigenvectors).

NumPy's `eig` function returns the list of unit eigenvectors and their corresponding eigenvalues for any square matrix. Let's look at the eigenvectors and eigenvalues of the squeeze mapping matrix $F_{squeeze}$:

In [80]:
```
eigenvalues, eigenvectors = LA.eig(F_squeeze)
eigenvalues # [λ0, λ1, …]
```
Out[80]:
```
array([1.4       , 0.71428571])
```
In [81]:
```
eigenvectors # [v0, v1, …]
```
Out[81]:
```
array([[1., 0.],
       [0., 1.]])
```
Indeed the horizontal vectors are stretched by a factor of 1.4, and the vertical vectors are shrunk by a factor of 1/1.4=0.714…, so far so good. Let's look at the shear mapping matrix F_{shear}:

In [82]:
```
eigenvalues2, eigenvectors2 = LA.eig(F_shear)
eigenvalues2 # [λ0, λ1, …]
```
Out[82]:
```
array([1., 1.])
```
In [83]:
```
eigenvectors2 # [v0, v1, …]
```
Out[83]:
```
array([[ 1.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  1.48029737e-16]])
```
Wait, what!? We expected just one unit eigenvector, not two. The second vector is almost equal to $\begin{pmatrix} -1 \\ 0 \end{pmatrix}$, which is on the same line as the first vector. This is due to floating point errors. We can safely ignore vectors that are (almost) collinear (ie. on the same line).

Trace

The trace of a square matrix M, noted tr(M) is the sum of the values on its main diagonal. For example:

In [84]:
```
D = np.array([
       [100, 200, 300],
       [ 10,  20,  30],
       [  1,   2,   3],
   ])
np.trace(D)
```
Out[84]:
```
123
```
The trace does not have a simple geometric interpretation (in general), but it has a number of properties that make it useful in many areas:

- $tr(A+B)=tr(A)+tr(B)$
- $tr(A \cdot B)=tr(B \cdot A)$
- $tr(A \cdot B \cdot \cdots \cdot Y \cdot Z)=tr(Z \cdot A \cdot B \cdot \cdots \cdot Y)$
- $tr(A^T \cdot B)=tr(A \cdot B^T)=tr(B^T \cdot A)=tr(B \cdot A^T)=\Sigma_{i,j} X_{i,j} \times Y_{i,j}$
- …

It does, however, have a useful geometric interpretation in the case of projection matrices (such as $F_{project}$ that we discussed earlier): it corresponds to the number of dimensions after projection. For example:

In [85]:

```
np.trace(F_project)
```

Out[85]:

1

In this article we have covered the following topics:

- Vectors
- Norm
- Dot product
- Matrices