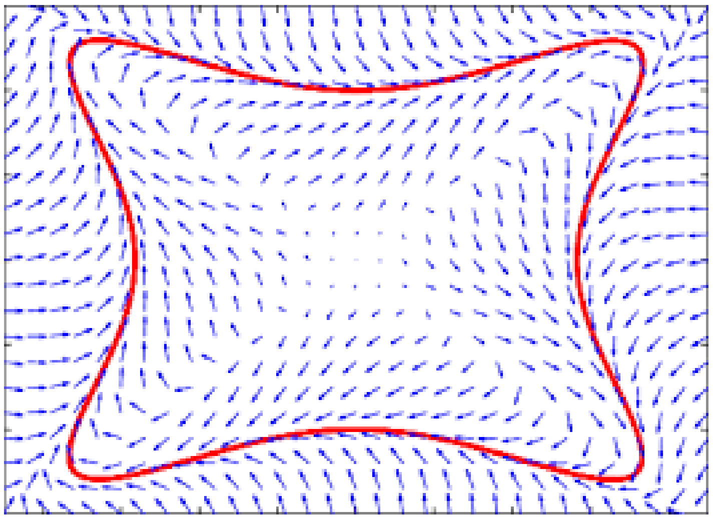
Vectors from the numpy Library
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/python_2025/vectors/vectors.pdf



Linear Algebra

- *Python's* `numpy` *library gives us tools for linear algebra;*
- *Vectors have norm (length), unit direction, pairwise angle;*
- *A matrix is a vector whose entries are rows;*
- `numpy` *math operators apply to an entire array;*
- *Matrix-vector and matrix-matrix multiplication functions exist;*
- *Linear solution or least squares approximations available;*
- *Factorizations like LU, QR, SVD are available;*
- *Eigenvalue can be computed.*

The `numpy()` library enables Python to do linear algebra efficiently. To carry out linear algebra operations, we need a simple way to create, manipulate, and display matrices and vectors. The Python `list()` data type is not suitable for this work. The `numpy()` library gives Python some of the same power as MATLAB. With a single command, we can carry out many linear algebra operations. *Some simple tasks we have in mind include:*

- *initialize the size, shape, value and data type of an array;*
- *find the shape of an array;*
- *find the number of rows and columns of an array;*
- *report or change any entry of an array:*
- *compute the norm of an array;*
- *compute the distance and angle between two vectors;*
- *multiply a vector by a matrix;*
- *solve a system of linear equations;*

In order to do our linear algebra this way, we have to begin by importing the library:

```
import numpy as np
```

It is customary to use `np` as the abbreviation for the `numpy()` library. Then to use a function such as `sum()` from this library, we use the name `np.sum()`.

1 A vector is a 1-dimensional numpy array

Let's begin by creating a simple example of a vector. We do this by invoking the `numpy()` function `array()`, which expects a list of comma-separated values enclosed in square brackets.

```
import numpy as np
x = np.array ( [ 1, 2, 3 ] )
```

There are several facts we can check for our array:

```
type ( x )
  <class 'numpy.ndarray'>
x.dtype
  dtype('int64')  # The values in x are presumed to be integers
x.shape
  (3,)        # x is a 1 dimensional array.  The first dimension is 3.
x.shape[0]
  3           # x.shape is a vector.  x.shape[0] is a number.
len ( x )
  3           # There are a total of 3 entries in x.
print ( x )
  [1 2 3]
```

Useful `numpy()` functions include

```
np.max ( x )
np.min ( x )
np.mean ( x )
np.sum ( x )
np.linalg.norm ( x )   #  Some numpy operations are in sublibrary "linalg"
```

Most `numpy()` functions can be applied to a vector element by element, creating a vector result:

```
a = np.exp ( x )
b = np.log ( x )
c = np.sin ( x )
d = np.sqrt ( x )
e = x + 10
f = 1 / x          # Why is this probably not what you want?
g = 1.0 / x
h = x**2
```

If we define a vector `y`, we can perform various arithmetic operations, which are done element by element. (By comparison, many such operations in MATLAB would require a special notation to indicate elementwise computation):

```
y = np.array ( [ 10, 20, 30 ] )
a = x + y
b = x * y
c = x / y
d = x**y
e = np.dot ( x, y )    # This is a dot product
```

In mathematics, and in MATLAB, we distinguish between row and column vectors. The numpy() library doesn't quite support this difference. The vectors we have created are all sort of row vectors. To create what behaves like a column vector of m entries, we would have to go to some trouble to create an $m \times 1$ matrix. It turns out that in Python, we generally won't need to worry about the distinction between row and column vectors that is rather important in MATLAB.

Occasionally, we need to set aside space for a vector before we know what we are going to put in it. or we need to create a vector of 0's, 1's, or random values. We can do any ot these tasks as follows:

```
a = np.empty ( 4 )
b = np.zeros ( 5 )
c = np.ones ( 6 )
d = np.random.rand ( 7 )
```

The entries of an array can be indexed. In an array of `n` entries, the first has index 0, the last index `n-1`.

```
a = np.array ( [ 10, 20, 30 ] )
a[0] = a[0] + a[2]
print ( a[0] )
print ( a[2] )
```

It is natural to use a `for()` loop to access each item of a vector:

```
a = np.array ( [ 10, 20, 30 ] )
n = a.shape[0]
for i in range ( 0, n ):
  a[i] = max ( a[i], 25 )     #  It would be wrong to use np.max() here!
print ( a )
```

Another version of the `for()` loop accesses entries by a temporary name, without using an index. (However, if we don't have an index, we can't put a computed result back into the array.)

```
a = np.array ( [ 10, 20, 30 ] )
for ai in a:       #  Automatically selects one element at a time
  print ( max ( ai, 15 ) )
```

One of the tricky featurs of Python involves the assignment operator. When an assignment statement involves arrays, and has the form `a=b`, then instead of copying the values of b into z

```
x = np.array ( [ 1, 2, 3 ] )
y = x                    #  Don't do this!
y = 2 * y
print ( x )              # x is the SAME as y; that's what "y=x" means!
print ( y )

x = np.array ( [ 1, 2, 3 ] )
z = z.copy ( )
z = 2 * z
print ( x )              # x is unchanged
print ( z )
```

2 Indexing a range of array entries

We have seen how to set a single entry of a vector to 7, or all the vectors to 0, 1, or random values. Sometimes we want to read or update just a subset of the vector entries. We can do this using indexing that is similar to what we have seen for the `for()` statement. In particular, suppose we begin with

```
x = np.array ( [ 500, 501, 502, 503, 504, 505, 506, 607, 508 ] )
```

We mentioned that the first entry is `x[0]`, and the last can be written as `x[8]` or `x[-1]`. Then, to set all the entries of `x` to 99, we could write

```
x[:] = 99
x[0:] = 99
x[0:9] = 99
x[0:-1] = 99    # wrong!
x[0:8] = 99     # wrong!
x = 99          # very wrong!
```

Recall that in Python, an index sequence `start:end` begins with the value `start` but the final index is `end-1`. If `start` is omitted, the sequence begins with the first index; If `end` is omitted, the sequence ends with the last index.

As with the `for()` statement, the index sequence can include an increment. To change every second element of `x` to 88, we can write any of these statements:

```
x[0:9:2] = 88
x[0::2] = 88
x[::2] = 88
```

Finally, and most confusingly, the increment can be negative. Look carefully at the following command, which makes a new vector by copying half of the elements of `x` in reverse order:

```
y = x[8:-1:-2]
```

The increment is -2. The seqnece starts at `x[8]`. Then it copies `x[6], x[4], x[2], x[0]` and stops, because the end index is -1. I don't blame you if you find this confusing at first!

3 Selecting some vector elements using a condition

Suppose we have a vector and we want to identify all the small elements, and maybe even copy them into a new vector. We can identify the small elements with a simple logical function, which returns a logical vector of `True` and `False` values.

```
x = np.random.rand ( 10 )
small = x < 0.5
print ( small )
```

We can copy the small elements into a new array by using the logical vector:

```
y = x[small]
```

If you are brave, you can combine these into a single step:

```
y = x [ x < 0.5 ]
```

We can even ask for elements between 0.20 and 0.60, but we have to be careful about how we phrase this statement:

```
y = x [ 0.20 < x and x < 0.6 ]  # Fails, "and" doesn't work here.
y = x [ 0.20 < x & x < 0.6 ]    # Fails, because & operator is confused
y = x [ ( 0.20 < x) & ( x < 0.6 ) ]  # Works
```

It is easy to see how to ask for values that are greater than 0.60 **or** less than 0.20 by using the `|` operator, or the logical `not`:

```
z = x [ ( x < 0.20 ) | ( 0.5 < x ) ]
z = x [ not [ ( 0.20 < x) & ( x < 0.6 ) ] ]
```

It's easy to get tripped up by the syntax, the parentheses, and the correct logical operators, but a well-written selection expression is an efficient means of getting the information you want.

4 The vector norm

In linear algebra, a vector `v` of size `n` has direction, and a length or norm. The `numpy()` library provides tools to determine both these quantities.

There are several ways of defining a norm, with the most common being the Euclidean norm, the square root of the sum of the squares of the entries. The `linalg` sublibrary of the `numpy()` library provides the function `norm()` to compute this. By default, the Euclidean norm is computed, but other options can be requested by including a second argument.

```
import numpy as np
v = np.random.rand ( 4 )
t0 = np.linalg.norm ( v )         # Euclidean
t1 = np.linalg.norm ( v, 1 )      # L1, sum of absolute values
t2 = np.linalg.norm ( v, 2 )      # L2, Euclidean
t3 = np.linalg.norm ( v, 'fro' ) # Frobenius, = Euclidean
t5 = np.linalg.norm ( v, inf )    # Loo, maximum absolute value
```

While the Frobenius and Euclidean norms are the same for vectors, we will see that they differ when we come to look at matrices.

For a vector in 2D, we think of the direction as an angle. But this idea doesn't generalize well. Instead, it is better to describe the direction of a vector by a vector of unit norm. This vector `u` is simply the quotient of `v` divided by its Euclidean norm.

This give us the following decomposition of any nonzero vector into a length t and direction u , which mathematically is

$$v = ||v|| \cdot \overrightarrow{v}$$

as

```
t = np.linalg.norm ( v )
u = v / t
v = t * u
```

we can ask:

- Can we add w and v?
- Is w equal to v?
- Is w simply a multiple of v (longer? shorter?)
- Does w lie somewhat or very little, along the direction of v?
- What is the angle between v and w?

Geometrically, we can think of adding two vectors as a kind of two-stage walk. We start at the origin, then walk in the direction and length of vector w. Once we reach the end of that first stage, we continue to walk from there, but now in the direction and length of v. You can make a corresponding plot by drawing the vector w starting at the origin, and then adding a picture of v that starts at the tip of w.

Algebraically, things are very simple. The sum of w and v is a new vector we can call u, whose components are found by adding the corresponding pairs of components of w and v:

$$u_i = w_i + v_i$$

It turns out that w is equal to v exactly if the norm of $v - w$ is zero. So this is easy to check! In fact, $||v - w||$ measures the magnitude or distance between the two vectors, so the size of this quantity is an indication of how close they are.

To answer the other questions, we need to know about the vector *inner product*, usually known by the more familiar name of *dot product*:

$$< v, w >= \sum_{i=1}^{n} v(i) * w(i)$$

It turns out that

$$< v, w >= ||v|| \cdot ||w|| \cdot \cos(\alpha)$$

where α is the angle between the two vectors. In particular, this means that

$$\cos(\alpha) = \frac{< v, w >}{||v|| \, ||w||}$$

If w is simply a multiple of v, then

- $\cos(\alpha) = 1$ (the vectors point in the same direction), or
- $\cos(\alpha) = -1$ (the vectors point in opposite directions.)

On the other hand, if $\cos(\alpha) = 0$, then the vectors are perpendicular.

In general, two vectors will have $\cos(\alpha)$ somewhere between -1 and +1. Values near +1 or -1 mean the two directions are close, while values near 0 mean they have little relation. In our work, we will often need to use this measurement to decide whether two objects, represented by vectors, are closely related or not.

In Python, we can compute the vector dot product of v and w using `np.dot(v,w)`

```
import numpy as np
w = np.random.rand ( 2 )
x = np.random.rand ( 2 )
wdotx = np.dot ( w, x )
print ( 'dot(w,x) = ', wdotx )
```

and the cosine of α is also easy:

```
import numpy as np
w = np.random.rand ( 2 )
w_norm = np.linalg.norm ( w )
x = np.random.rand ( 2 )
x_norm = np.linalg.norm ( x )
cwx = np.dot ( w, x ) / w_norm / x_norm
print ( 'cos(w,x) = ', cwx )
```

Notice in these examples that w and x are defined as simple numpy arrays (essentially lists). We did not try to define them as row or column vectors.

In mathematics, and in MATLAB, the vector dot product can only be applied to a row vector dotted with a column vector. If the dot product of two column vectors, say c_1 and c_2 is desired, then the first factor must be transposed before the dot product is computed. Mathematically this might be written as

$$\text{c1dotc2} = c_1^T \cdot c_2$$

or in MATLAB, as

```
c1dotc = c1' * c2
```

If you are used to the conventions of mathematics or MATLAB, you should be aware that the `np.dot()` function, can be applied to simple `numpy` arrays as long as they have the same extent (number of entries).

5 The projection of one vector on another

Given any two vectors, it is unlikely that they are equal, but we would still like some information about whether they are closely related or not. We already know how to compute the angle between vectors u and v. However, another way to view this relationship is to measure what is called the *projection* of v onto u. What we want to know is, how much of vector v is going in the direction of u?

We are going to compute a dot product between v and the *direction* of u. In other words, we compute a number which we might call α:

$$\alpha =< v, \hat{u} >$$

Now we can think of $v = v_1 + v_2$, where the v_1 component is in the direction u, and the v_2 component is perpendicular to u. Thus we can also compute v_2:

$$\begin{aligned} v_1 &= \alpha * \hat{u} \\ v &= v_1 + v_2 = \alpha * \hat{u} + v_2 \\ v_2 &= v - \alpha * \hat{u} \\ v &= v_1 + v_2 = \alpha * \hat{u} + (v - \alpha * \hat{u}) \end{aligned}$$

As an example of projection, let $u = (3, 4)$, and $v = (5, 1)$:

```
u = np.array ( [ 3, 4 ] )
u_norm = np.linalg.norm ( u )
uhat = u / u_norm
uhat_norm = 1.0
v = np.array ( [ 5, 1 ] )
v_norm = np.linalg.norm ( v )

beta = np.dot ( v, uhat )
v1 = beta * uhat
v2 = v - v1

v1_norm = np.linalg.norm ( v1 )
v2_norm = np.linalg.norm ( v2 )

cos1 = np.dot ( v1, uhat ) / v1_norm / uhat_norm
angle1 = np.arccos ( cos1 )     # Python may complain cos1 illegal input for arccos
cos2 = np.dot ( v2, uhat ) / v2_norm / uhat_norm
angle2 = np.arccos ( cos2 )
```

You may find that the computation of `angle1` fails, because the value of `cos1` is illegal as input to `np.arccos()`. If so, look carefully at the value of `cos1`, explain what is wrong, and figure out a way to fix the problem.

Projection decomposes a vector v into two components v_1 and v_2 which are perpendicular to each other. This means that the three vectors v, v_1, and v_2 form a 90 degree triangle, and hence we must have $||v||^2 = ||v_1||^2 + ||v_2||^2$. You can verify this from the previous calculations by printing and comparing `v_norm**2` and `v1_norm**2+v2_norm**2`.