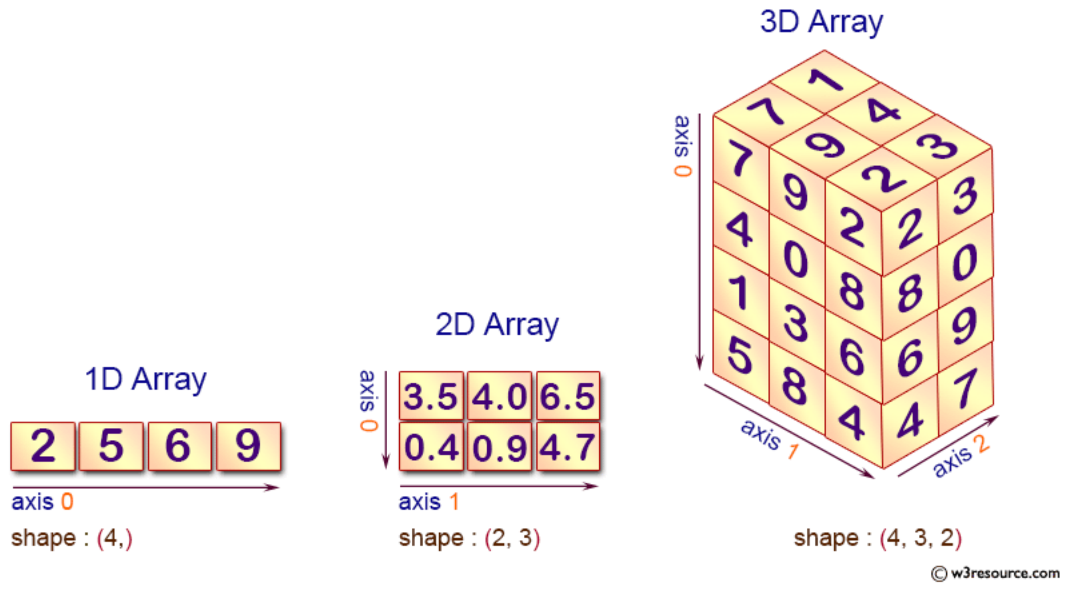
Vector Processing Using `numpy`
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/numpy/numpy.pdf



The numpy Library

- *Mathematicians think of lists and tables as vectors and matrices;*
- *Vectors are thought of as "things"*
- *Matrices represent linear transformations of vectors;*
- *The study of linear transformations is called linear algebra;*
- *Python's* `numpy` *library gives us tools for linear algebra;*
- *Vectors have norm (length), unit direction, pairwise angle;*
- *Matrix-vector multiplication A*x=b transforms x into b;*
- *Given A and b, we can usually figure out what x was;*
- *Insight into a matrix comes from LU, QR, SVD factorizations;*
- *Matrix eigenvalues indicate the structure of the transformation.*

1 A Library for Vectorized Computing

When our computing tasks involve lists or arrays of data, it is often awkward to set up space for the data, initialize it, and carry out the same operation on every data item one at a time.

In linear algebra, there is a standard language of vectors and matrices that allows us to express many actions in a simple and precise language. Rather than specifying each item of the vector by index, we might multiply all the entries of a vector v by 10 with the simple statement

$$w \leftarrow v * 10$$

Similar expressions can be used to add 20 to every element of a vector, to add two vectors of the same length, to find the maximum entry, and so on.

A computation that is carried out in this way is said to be *vectorized*. By using the tighter vector language, we come closer to representing the important features of our work, while dropping bookkeeping items such as indices. This makes it easier to write correct code; moreover, vectorized programs can be executed much faster. In Python, for example, vectorized operations will be passed off to a hidden, underlying compiled C code.

The `numpy` library:

- creates the datatypes corresponding to vectors, matrices, and higher dimensional arrays;
- defines rules for arithmetic operations involving them;
- adds a vectorized version of most mathematical functions;
- adds "convenience" operators to make vector work efficient;

2 Starting `numpy`, getting help

We import the library using a standard abbreviation:

```
import numpy as np
```

There is brief help on the library, and on the sublibraries:

```
help ( np )
help ( doc )
help ( lib )
help ( random )
help ( linalg )
help ( fft )
help ( polynomial )
```

and of course, if you know the name of a function, you can request specific help

```
help ( np.linspace )
```

3 Making vectors

Let's create a few vectors of length 3:

```
a = np.array ( [ 1.2, 3.4, 5.6 ] )
e = np.empty ( 3 )
l = np.linspace ( 0.0, 1.0, 3 )
o = np.ones ( 3 )
r = np.random.rand ( 3 )
z = np.zeros ( 3 )
```

Some familiar functions can be used on these new objects:

- `print()` displays a vector;
- `len()` return 3 for each of vectors above;
- `type()` returns `<class 'numpy.ndarray'>`;

Some new `numpy` attributes are available as well. Consider our `numpy` array `a`. We have

- `a.dtype` returns the data type (int, float, complex...)

- `a.ndim` returns 1 (these are 1-dimensional vectors;
- `a.shape` returns (3,);
- `a.size` returns 3 (total number of entries);

We can append values to a `numpy` array, but the syntax is a little different than we saw before for Python lists:

- np.append (array, values) appends values to array;

4 Arithmetic and Indexing

We can do arithmetic on all the entries of an array at once.

```
u = 2 * v
u = 3 * u + 1
w = u + v
w = u * v  #  elementwise: w[i] = u[i] * v[i]
v = u**2
```

Individual entries of an array are accessed by the usual 0-based indexing scheme:

```
u = np.array ( [ 100, 200, 300 ] )
u[1] = 99
```

A range of entries are accessed by the usual Python range conventions:

- `u[i:j]` entries `i` through `j-1`;
- `u[i:]` entries `i` to the end;
- `u[:k]` all entries up to, but not including `k`;
- `u[-1]` the last entry;
- `u[k::2]` every other entry, starting at `k`;
- `u[::-1]` the entire array in reverse order;

5 Functions and Operators

Given a `numpy` vector (1 dimensional array) `v`, we can ask for various statistical quantities:

```
min ( v )
max ( v )
mean ( v )
prod ( v )  #  product of all the entries
var ( v )
std ( v )
sum ( v )  #  sum of all the entries
```

The `numpy` library also redefines math functions so that they can operate on each element of an array:

```
u = np.abs ( v )
u = np.cos ( np.pi * v )
u = np.log ( v )
u = np.sqrt ( v**2 + 1 )
w = np.exp ( u + v )
```

The *dot product* of two n-vectors u and v is

$$u \cdot v = \sum_{i=0}^{i<n} u_i \, v_i$$

Using the function `np.dot()`, we can compute

```
udotv = np.dot ( u, v )
```

Later, the dot product will tell us the norm of a vector, whether two vectors are perpendicular or parallel, and can also be used to compute matrix-vector products.

6 Detecting conditions

The numpy logical vector operators:

- ~ (not) reverses all logical values;
- & (and) returns True for pairs of true values;
- | (or) returns True for each pair of values if at least one is true;

A logical expression involving a numpy array returns a numpy array of True and False values:

```
  x = [ 0, 1, 2, 3, 4, 5, 6, 7 ]
  test = ( x % 3 == 0 )
  print ( test )
  [ True, False, False, True, False, False, True, False ]
```

A vector of logical expressions can be used to select those entries of an array corresponding to True values:

```
  x = [ 0, 1, 2, 3, 4, 5, 6, 7 ]
  test = ( x % 3 == 0 )
  print ( test )
  [ True, False, False, True, False, False, True, False ]
  print ( x[test] )
  [ 0, 3, 6 ]
```

The functions `np.any()` and `np.all` check an array and return True if `any` or `all` values satisfy some condition.

```
np.all ( x <= 1.0 )
np.any ( x == 0.0 )
```

We can have more complicated conditions:

```
np.all ( 0.0 <= sin(x) )
np.any ( x**2 == 1.0 )
```

To check if every element of the array `x` is in the interval [0,1]:

```
np.all ( ( 0.0 <= x ) & ( x <= 1.0 ) )
```

To check any x positive or even?

```
np.any ( ( 0.0 <= x ) | ( x % 2 == 0 ) )
```

7 Locating a minimum (or maximum)

The functions `np.argmin()` and `np.argmax()` returns the indexes of the array which correspond to the minimum and maximum values. If you are computing $y = f(x)$, you might want to know the minimum value of `y`, but perhaps also the index of the `y` vector where this occurred. Using that same index on `x` gives you the `x` value at which the minimum occurred.

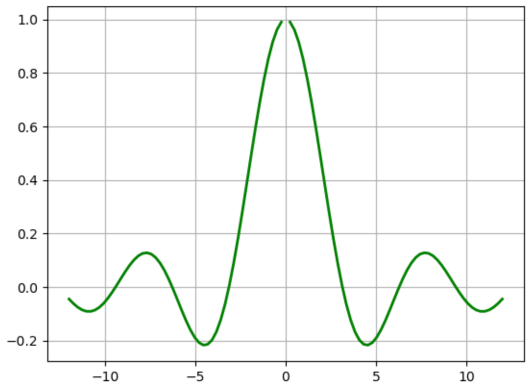
```
ymin = np,min ( y )
imin = np.argmin ( y )
xmin = x[imin]
```

8 Plotting the minimum and maximum

To see how we might use the functions that identify the minimum and maximum, let's consider the so-called "sinc" function $f(x) = \frac{\sin(x)}{x}$, on the domain $-12 \leq x \leq 12$. We will sample it at 101 points, creating vectors x and y.

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

n = 101
x = np.linspace ( -12, 12, n )
y = np.sin ( x ) / x     # We get a divide-by-zero complaint
plt.plot ( x, y, 'g-' )
plt.show ( )
```



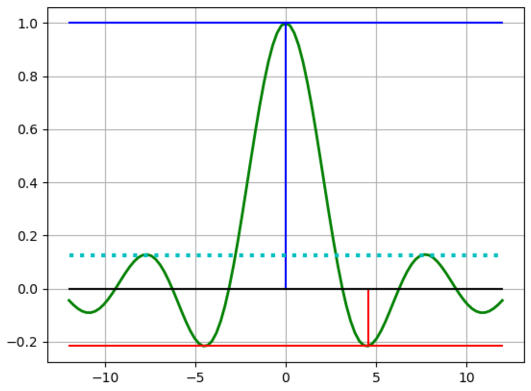
Now suppose we try to find the value of the maximum, and its location. (We know `ymax = 1` and `xmax = 0`:

```
ymax = np.max ( y )
imax = np.argmax ( y )
xmax = x[imax]
print ( ymax, imax, xmax )
nan 50 0.0                  # This output is surprising
```

The value of `ymax` is coming out as `nan`, that is, "not-a-number". Of course, at $x = 0$, we are computing $0/0$. There is actually easy to fix. We don't even need to know that this happens at index 50. We can simply say that that for any plot point with $x = 0$, the y value should be replaced by 1:

```
y = np.sin(x) / x
y[x==0] = 1.0
ymax = np.max ( y )
imax = np.argmax ( y )
xmax = x[imax]
print ( ymax, imax, xmax )
1 50 0.0                  # This output makes sense
```

Similar commands get us the minimum information, and we can also compute the mean or average value.



```
  plt.clf ( )
  plt.plot ( x, y, 'g-', linewidth = 2 )
  plt.plot ( [x1,x2], [ymax,ymax], 'b-' )
  plt.plot ( [xmax,xmax], [0.0,ymax], 'b-' )
  plt.plot ( [x1,x2], [ymin,ymin], 'r-' )
  plt.plot ( [xmin,xmin], [0.0,ymin], 'r-' )
  plt.plot ( [x1,x2], [ymean,ymean], 'c:', linewidth = 3 )
  plt.plot ( [x1,x2], [0.0,0.0], 'k-' )
```

Notice that the `np.argmin()` function only returned one value, when apparently the function hit its minimum value twice. We can imagine situations in which we actually wanted to catch many values that satisfy some condition. We can easily do that with logical expressions.

Suppose we were interested in all the `x` values for which the function $y = f(x)$ is less than or equal to zero. The expression

```
y < 0
```

The expression

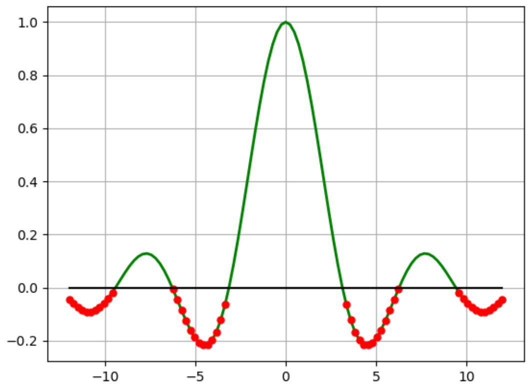
```
y[y < 0 ]
```

is a vector of all the negative `y` values. The expression

```
x[y < 0 ]
```

is a vector of all the `x` values that correspond to negative `y` values.
Let's go back to our sinc() function and mark all the negative data points:

```
  plt.plot ( x, y, 'g-', linewidth = 2 )
  plt.plot ( x[y<0], y[y<0], 'r.', markersize = 10 )
  plt.plot ( [x1,x2], [0.0,0.0], 'k-' )
```



9 Making matrices

Now it's time to move to two dimensions, and see how **numpy** arrays can be used to create, modify and analyze matrices.

An $m \times n$ mathematical matrix can be represented by a **numpy** array of dimensions (m, n). We can create matrices by commands like:

```
E = np.empty ( [ 3, 2 ] )
I = np.identity ( 3 )
O = np.ones ( [ 3, 2 ] )
R = np.random.rand ( 3, 2 )    # Does not bracket the dimensions!
Z = np.zeros ( [ 3, 2 ] )
```

For small matrices we can enter the values in a list of lists. For a mathematical matrix

$$A = \begin{bmatrix} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \\ 40 & 41 & 42 & 43 \end{bmatrix}$$

we can enter the Python commands:

```
A = np.array ( [ \
   [  0,  1,  2,  3 ], \
   [ 10, 11, 12, 13 ], \
   [ 20, 21, 22, 23 ], \
   [ 30, 31, 32, 32 ], \
   [ 40, 41, 42, 43 ] ] )
```

Some new **numpy** array attributes are available as well:

- `A.ndim` tells us that A is a 2-dimensional array;
- `A.shape` statement returns (5,4);
- `A.size` returns 20 (total number of entries);

To index a particular item, instead of using $[i][j]$, **numpy** arrays use the more familiar $[i, j]$ notation.

We have already seen some examples of how Python indexing works. For our sample matrix `A`,

```
A[0,:] = [ 0, 1, 2, 3]
A[:,1] = [ 1, 11, 21, 31, 41 ]
A[1,2] = 12
```

For a two-dimensional array, we have the `np.transpose()` operator:

```
B = np.transpose(A)
[   [ 0, 10, 20, 30, 40 ],
    [ 1, 11, 21, 31, 41 ],
    [ 2, 12, 32, 32, 42 ],
    [ 3, 13, 23, 33, 43 ] ]
```

If `x` is a vector of length n, and `A` is an $m \times n$ matrix, we can use the `np.dot()` operator to carry out matrix-vector multiplication, $b = A * x$:

```
x = [ 1, 2, 3, 4 ]
b = np.dot ( A, x )
[ 20, 120, 220, 320, 420 ]
```

10 Slicing a Matrix

Let's suppose the 5×4 matrix A has the following entries:

$$A = \begin{bmatrix} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \\ 40 & 41 & 42 & 43 \end{bmatrix}$$

What is the value of each of the following expressions?

1. A[2, :]
2. A[:, 1]
3. A[1:, 1:]
4. A[1:-1, 1:] (Why does this differ from previous item?)
5. A[::2, :]
6. A[2:, :2]
7. A[1::2, ::2]

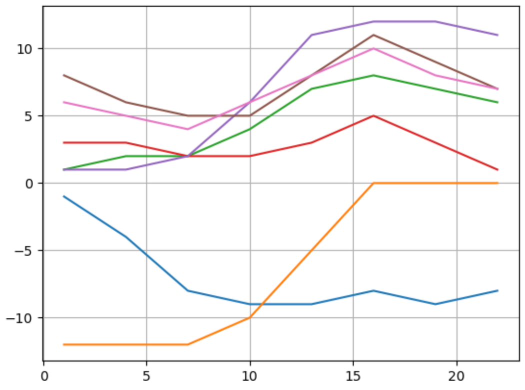
11 Plotting Temperature Data

Some `numpy` nfunctions can be applied to a matrix in a variety of ways. To start with, consider the `np.max()` function. Let's take as our data an array T that actually measures the temperature every 3 hours, over a week.

```
T = np.array ( [ \
   [ -1, -4, -8, -9, -9, -8, -9, -8 ], \
   [-12,-12,-12,-10, -5,  0,  0,  0 ], \
   [  1,  2,  2,  4,  7,  8,  7,  6 ], \
   [  3,  3,  2,  2,  3,  5,  3,  1 ], \
   [  1,  1,  2,  6, 11, 12, 12, 11 ], \
   [  8,  6,  5,  5,  8, 11,  9,  7 ], \
   [  6,  5,  4,  6,  8, 10,  8,  7 ] ] )
```

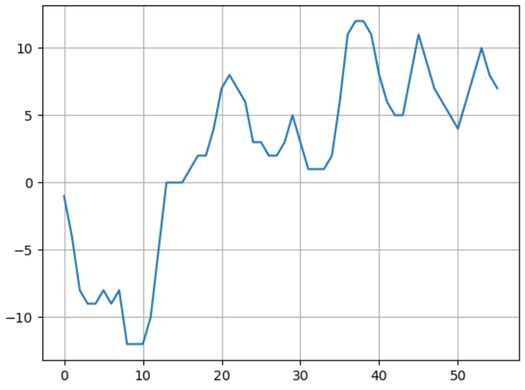
We could look at this data day by day, and plot it that way:

```
h = np.linspace ( 1, 22, 8 )  # 24 hour time
plt.clf ( )
for day in range ( 0, 7 ):
  plt.plot ( h, T[day,:] )
plt.grid ( True )
plt.show ( )
plt.close ( )
```



If we want a single plot over the whole week, we need to "flatten" the matrix, that is, to make a vector by stringing the rows together:

```
Tweek = T.flatten ( )
plt.plot ( Tweek )
plt.show ( )
```



12 Analyzing Temperature Data

Now that we have our temperature data, we might want to ask for the minimum, average, and maximums

- for each day

- for each measured hour;
- over the whole week.

```
min_day = np.min ( T, axis = 0 )
min_hour = np.min ( T, axis = 1 )
min_week = np.min ( T )
```

and our results are:

```
min(T) daily  =  [-12 -12 -12 -10  -9  -8  -9  -8]
min(T) hourly =  [ -9 -12   1   1   1   5   4]
min(T) weekly =  -12
```

You should see that `axis=0` computes the minimum value for each row, while `axis=1` does the same for columns, and with no axis specified, the minimum is over the whole set of data.

You can get similar results using `np.max()`, `np.mean()`, and `np.sum()`.

13 Making X and Y Spatial Matrices for Plotting

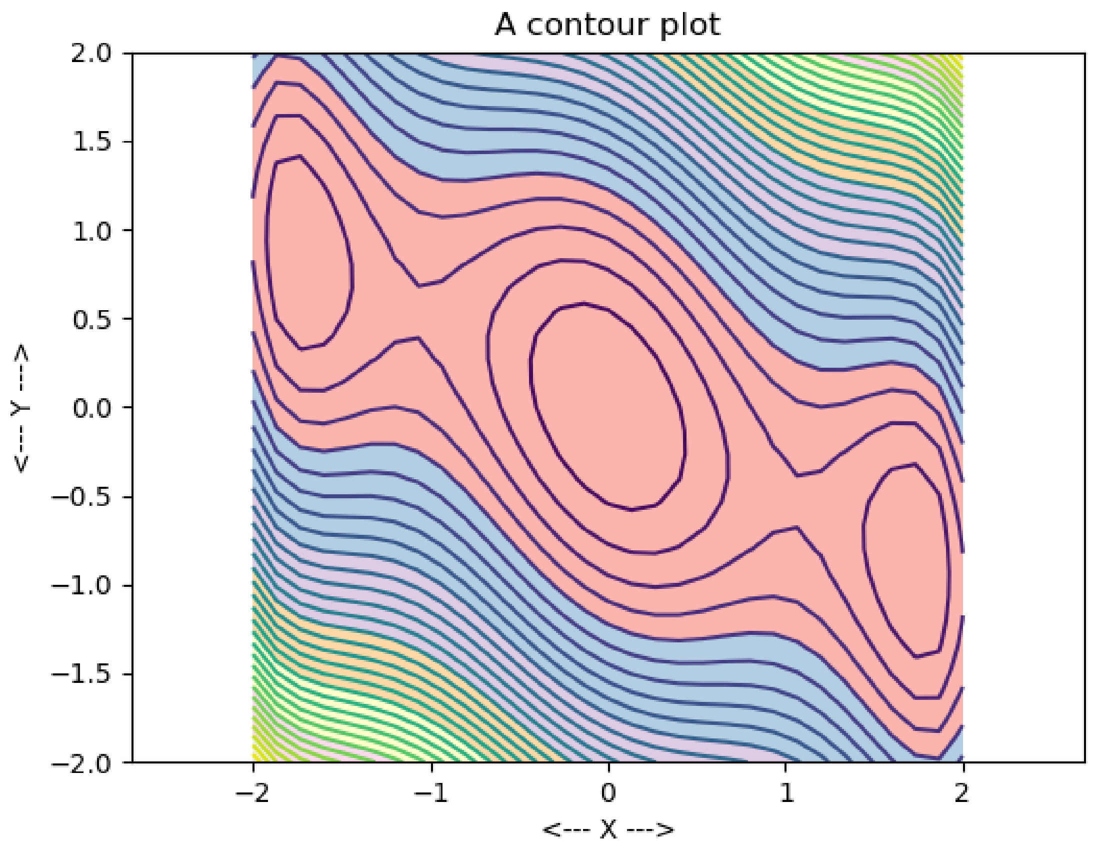
A standard way of sampling a function $z = f(x, y)$ is to define a grid of m equally spaced points over the x range, and n equally spaced points over the y range, evaluate the function $z_{i,j} = f(x_i, y_j)$ and somehow create a visual display of this information.

The `numpy` library allows us to write such a process in an efficient way. Here, we would like to sample the function $f(x, y) = 2x^2 + 1.05x^4 + x^6/6 = xy + y^2$ over the square $-2 \leq x, y \leq +2$ and then make a contour plot.

```
xvec = np.linspace ( -2.0, 2.0, 31 )
yvec = np.linspace ( -2.0, 2.0, 31 )

X, Y = np.meshgrid ( xvec, yvec )
Z = 2 * X**2 - 1.05 * X**4 + X**6 / 6 + X * Y + Y**2

plt.clf ( )
plt.contourf ( X, Y, Z )     # filled regions
plt.contour ( X, Y, Z, levels = 35 )  # contour lines
plt.show ( )
```



More about these kind of tricks when we resume our discussion of plotting.