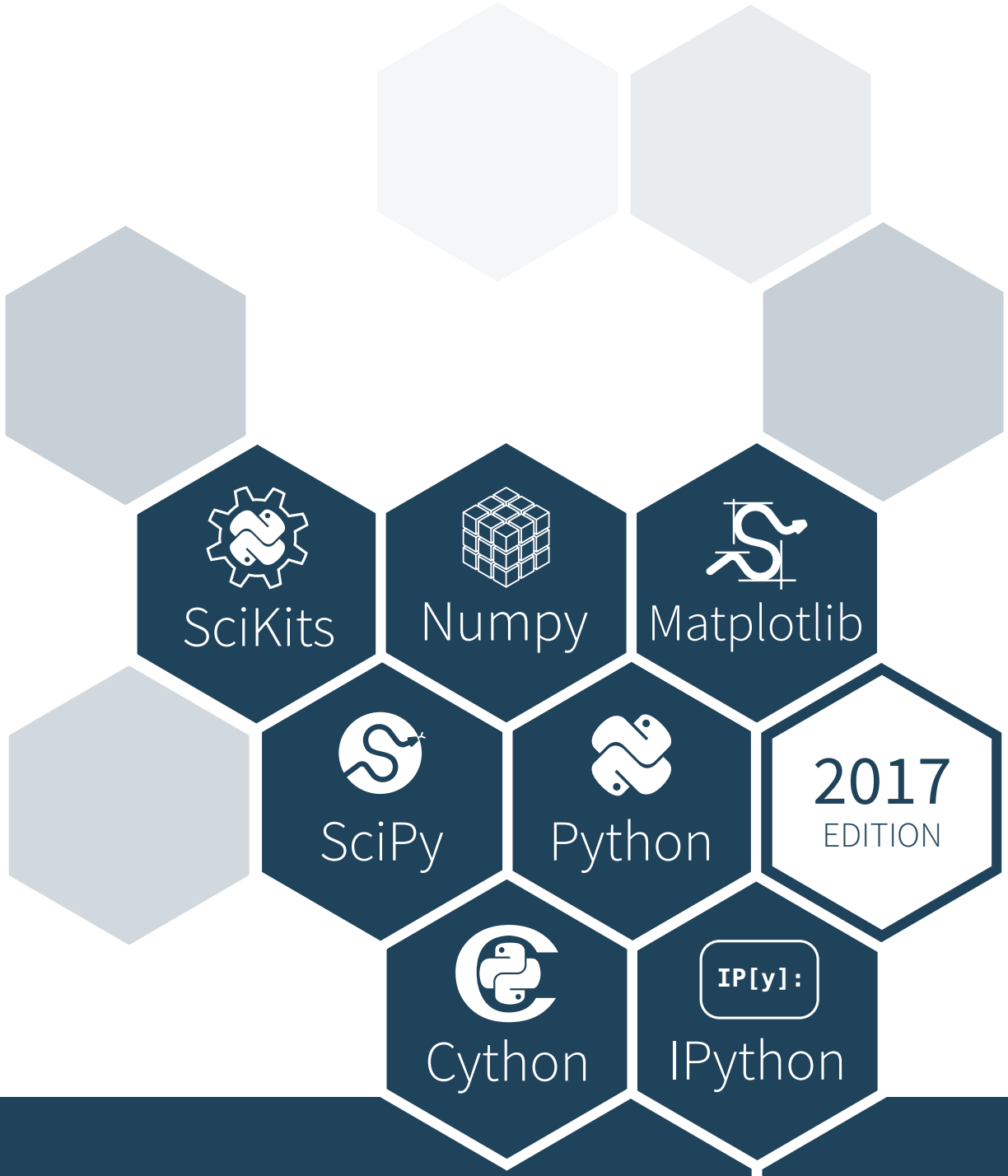


Scipy Lecture Notes

www.scipy-lectures.org

Edited by
Gaël Varoquaux
Emmanuelle Gouillart
Olaf Vahtras

Gaël Varoquaux • Emmanuelle Gouillart • Olav Vahtras
Christopher Burns • Adrian Chauve • Robert Cimrman • Christophe Combelles
Pierre de Buyl • Ralf Gommers • André Espaze • Zbigniew Jędrzejewski-Szmek
Valentin Haenel • Gert-Ludwig Ingold • Fabian Pedregosa • Didrik Pinte
Nicolas P. Rougier • Pauli Virtanen

and many others...

Contents

CHAPTER 1

NumPy: creating and manipulating numerical data

Authors: *Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux, and Pauli Virtanen*

This chapter gives an overview of NumPy, the core tool for performant numerical computing with Python.

1.1 The NumPy array object

Section contents

1.1.1 What are NumPy and NumPy arrays?

NumPy arrays

Python objects

- high-level number objects: integers, floating point
- containers: lists (costless insertion and append), dictionaries (fast lookup)

NumPy provides

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)
- Also known as *array oriented computing*

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

Tip: For example, An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Why it is useful: Memory-efficient container that provides fast numerical operations.

```
In [1]: L = range(1000)

In [2]: %timeit [i**2 for i in L]
1000 loops, best of 3: 403 us per loop

In [3]: a = np.arange(1000)

In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

NumPy Reference documentation

- On the web: http://docs.scipy.org/
- Interactive help:

```
In [5]: np.array?
String Form:<built-in function array>
Docstring:
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0, ...
```

- Looking for something:

```
>>> np.lookfor('create array')
Search results for 'create array'
--------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
```

```
In [6]: np.con*?
np.concatenate
np.conj
np.conjugate
np.convolve
```

Import conventions

The recommended convention to import numpy is:

```
>>> import numpy as np
```

1.1.2 Creating arrays

Manual construction of arrays

- **1-D**:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

- **2-D, 3-D, ...**:

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)     # returns the size of the first dimension
2

>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],

       [[3],
        [4]]])
>>> c.shape
(2, 2, 1)
```

Exercise: Simple arrays

- Create a simple two dimensional array. First, redo the examples from above. And then create your own: how about odd numbers counting backwards on the first row, and even numbers on the second?
- Use the functions `len()`, `numpy.shape()` on these arrays. How do they relate to each other? And to the `ndim` attribute of the arrays?

Functions for creating arrays

Tip: In practice, we rarely enter items one by one...

- Evenly spaced:

```
>>> a = np.arange(10) # 0 .. n-1  (!)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exclusive), step
>>> b
array([1, 3, 5, 7])
```

- or by number of points:

```
>>> c = np.linspace(0, 1, 6)   # start, end, num-points
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

- Common arrays:

```
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

- `np.random`: random numbers (Mersenne Twister PRNG):

```
>>> a = np.random.rand(4)       # uniform in [0, 1]
>>> a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])
```

```
>>> b = np.random.randn(4)       # Gaussian
>>> b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])

>>> np.random.seed(1234)         # Setting the random seed
```

Exercise: Creating arrays using functions

- Experiment with `arange`, `linspace`, `ones`, `zeros`, `eye` and `diag`.
- Create different kinds of arrays with random numbers.
- Try setting the seed before creating an array with random values.
- Look at the function `np.empty`. What does it do? When might this be useful?

1.1.3 Basic data types

You may have noticed that, in some instances, array elements are displayed with a trailing dot (e.g. `2.` vs `2`). This is due to a difference in the data-type used:

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

Tip: Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```

The **default** data type is floating point:

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

There are also other types:

Complex

```
>>> d = np.array([1+2j, 3+4j, 5+6*1j])
>>> d.dtype
dtype('complex128')
```

Bool

```
>>> e = np.array([True, False, False, True])
>>> e.dtype
dtype('bool')
```

Strings

```
>>> f = np.array(['Bonjour', 'Hello', 'Hallo',])
>>> f.dtype     # <--- strings containing max. 7 letters
dtype('S7')
```

Much more

- int32
- int64
- uint32
- uint64

1.1.4 Basic visualization

Now that we have our first data arrays, we are going to visualize them.

Start by launching IPython:

```
$ ipython
```

Or the notebook:

```
$ ipython notebook
```

Once IPython has started, enable interactive plots:

```
>>> %matplotlib
```

Or, from the notebook, enable plots in the notebook:

```
>>> %matplotlib inline
```

The `inline` is important for the notebook, so that plots are displayed in the notebook and not in a new window.

Matplotlib is a 2D plotting package. We can import its functions as below:

```
>>> import matplotlib.pyplot as plt  # the tidy way
```

And then use (note that you have to use `show` explicitly if you have not enabled interactive plots with `%matplotlib`):

```
>>> plt.plot(x, y)       # line plot
>>> plt.show()           # <-- shows the plot (not needed with interactive plots)
```

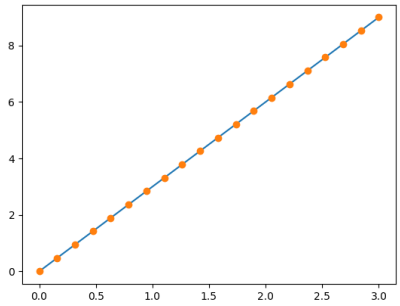
Or, if you have enabled interactive plots with `%matplotlib`:

```
>>> plt.plot(x, y)       # line plot
```

- **1D plotting**:

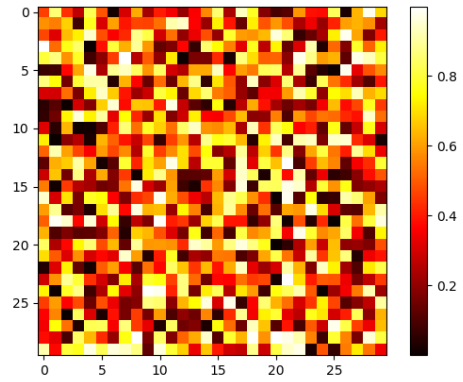
```
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at ...>]
```

```
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at ...>]
```



- **2D arrays** (such as images):

```
>>> image = np.random.rand(30, 30)
>>> plt.imshow(image, cmap=plt.cm.hot)
<matplotlib.image.AxesImage object at ...>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar object at ...>
```



See also:

More in the: *matplotlib chapter*

Exercise: Simple visualizations

- Plot some simple arrays: a cosine as a function of time and a 2D matrix.
- Try using the `gray` colormap on the 2D matrix.

1.1.5 Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
```

Warning: Indices begin at 0, like other Python sequences (and C/C++). In contrast, in Fortran or Matlab, indices begin at 1.

The usual python idiom for reversing a sequence is supported:

```
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

For multidimensional arrays, indexes are tuples of integers:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[2, 1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

Note:

- In 2D, the first dimension corresponds to **rows**, the second to **columns**.
- for multidimensional `a`, `a[0]` is interpreted by taking all elements in the unspecified dimensions.

Slicing: Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

Note that the last index is not included! :

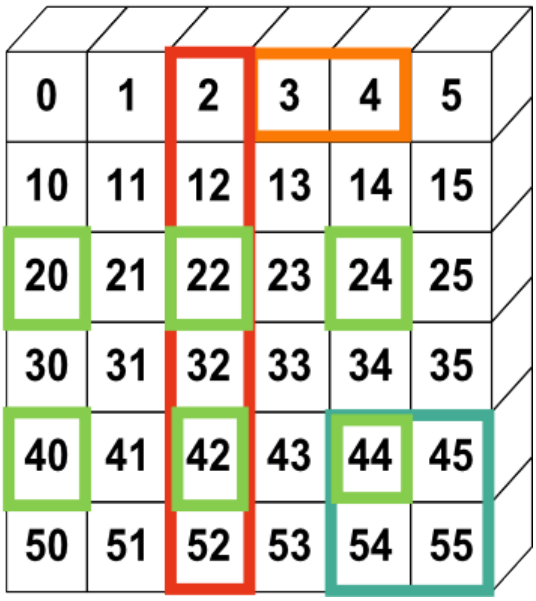
```
>>> a[:4]
array([0, 1, 2, 3])
```

All three slice components are not required: by default, *start* is 0, *end* is the last and *step* is 1:

```
>>> a[1:3]
array([1, 2])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```

A small illustrated summary of NumPy indexing and slicing...

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



You can also combine assignment and slicing:

```
>>> a = np.arange(10)
>>> a[5:] = 10
>>> a
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
>>> b = np.arange(5)
>>> a[5:] = b[::-1]
>>> a
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

Exercise: Indexing and slicing

- Try the different flavours of slicing, using `start`, `end` and `step`: starting from a linspace, try to obtain odd numbers counting backwards, and even numbers counting forwards.
- Reproduce the slices in the diagram above. You may use the following expression to create the array:

```
>>> np.arange(6) + np.arange(0, 51, 10)[:, np.newaxis]
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

Exercise: Array creation

Create the following arrays (with correct data types):

```
[[1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 2],
 [1, 6, 1, 1]]

[[0., 0., 0., 0., 0.],
 [2., 0., 0., 0., 0.],
 [0., 3., 0., 0., 0.],
 [0., 0., 4., 0., 0.],
 [0., 0., 0., 5., 0.],
 [0., 0., 0., 0., 6.]]
```

Par on course: 3 statements for each

Hint: Individual array elements can be accessed similarly to a list, e.g. `a[1]` or `a[1, 2]`.

Hint: Examine the docstring for `diag`.

Exercise: Tiling for array creation

Skim through the documentation for `np.tile`, and use this function to construct the array:

```
[[4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1],
 [4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1]]
```

1.1.6 Copies and views

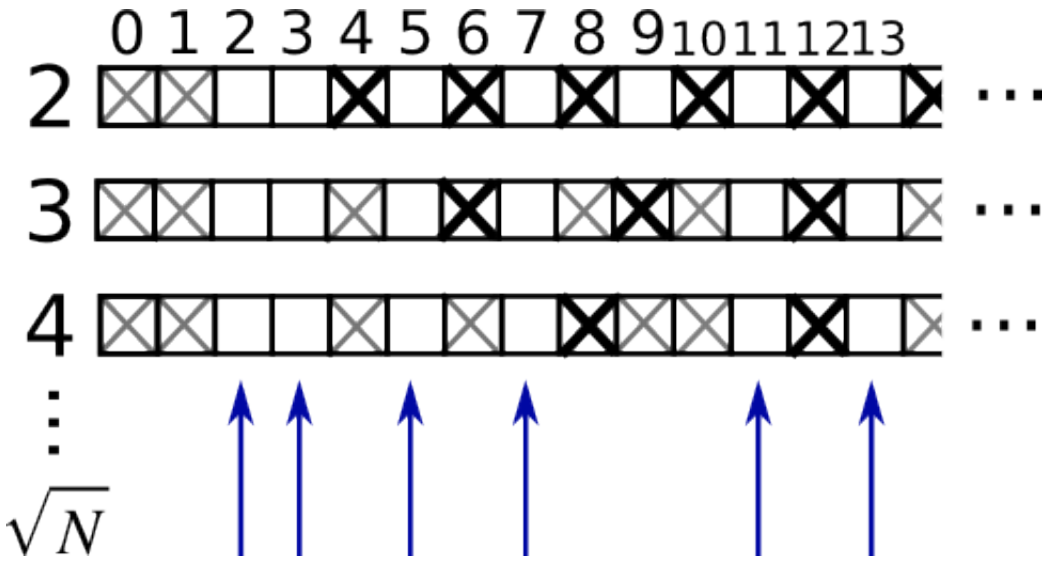
A slicing operation creates a **view** on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use `np.may_share_memory()` to check if two arrays share the same memory block. Note however, that this uses heuristics and may give you false positives.

When modifying the view, the original array is modified as well:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b
array([0, 2, 4, 6, 8])
>>> np.may_share_memory(a, b)
True
>>> b[0] = 12
>>> b
array([12,  2,  4,  6,  8])
>>> a   # (!)
array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9])

>>> a = np.arange(10)
>>> c = a[::2].copy()  # force a copy
>>> c[0] = 12
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> np.may_share_memory(a, c)
False
```

This behavior can be surprising at first sight... but it allows to save both memory and time.

Worked example: Prime number sieve



Compute prime numbers in 0–99, with a sieve

- Construct a shape (100,) boolean array `is_prime`, filled with True in the beginning:

```
>>> is_prime = np.ones((100,), dtype=bool)
```

- Cross out 0 and 1 which are not primes:

```
>>> is_prime[:2] = 0
```

- For each integer `j` starting from 2, cross out its higher multiples:

```
>>> N_max = int(np.sqrt(len(is_prime) - 1))
>>> for j in range(2, N_max + 1):
...     is_prime[2*j::j] = False
```

- Skim through `help(np.nonzero)`, and print the prime numbers
- Follow-up:
  - Move the above code into a script file named `prime_sieve.py`
  - Run it to check it works
  - Use the optimization suggested in the sieve of Eratosthenes:
  1. Skip `j` which are already known to not be primes
  2. The first number to cross out is j^2

1.1.7 Fancy indexing

Tip: NumPy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**). This method is called *fancy indexing*. It creates **copies not views**.

Using boolean masks

```
>>> np.random.seed(3)
>>> a = np.random.randint(0, 21, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
```

```
>>> (a % 3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> mask = (a % 3 == 0)
>>> extract_from_a = a[mask] # or,  a[a%3==0]
>>> extract_from_a           # extract a sub-array with the mask
array([ 3,  0,  9,  6,  0, 12])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

```
>>> a[a % 3 == 0] = -1
>>> a
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```

Indexing with an array of integers

```
>>> a = np.arange(0, 100, 10)
>>> a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

```
>>> a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
array([20, 30, 20, 40, 20])
```

New values can be assigned with this kind of indexing:

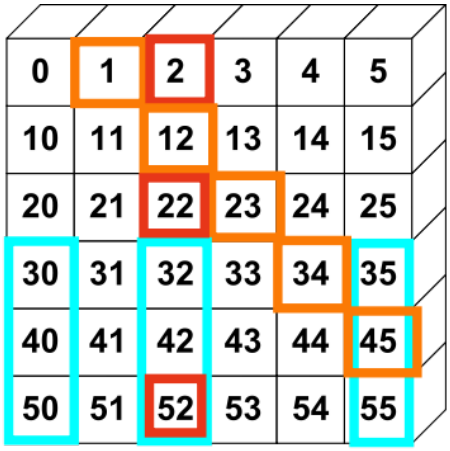
```
>>> a[[9, 7]] = -100
>>> a
array([   0,   10,   20,   30,   40,   50,   60, -100,   80, -100])
```

Tip: When a new array is created by indexing with an array of integers, the new array has the same shape as the array of integers:

```
>>> a = np.arange(10)
>>> idx = np.array([[3, 4], [9, 7]])
>>> idx.shape
(2, 2)
>>> a[idx]
array([[3, 4],
       [9, 7]])
```

The image below illustrates various fancy indexing applications

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Exercise: Fancy indexing

- Again, reproduce the fancy indexing shown in the diagram above.
- Use fancy indexing on the left and array creation on the right to assign values into an array, for instance by setting parts of the array in the diagram above to zero.

1.2 Numerical operations on arrays

Section contents

1.2.1 Elementwise operations

Basic operations

With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
```

All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
```

```
>>> a * b
array([ 2.,  4.,  6.,  8.])

>>> j = np.arange(5)
>>> 2**(j + 1) - j
array([ 2,  3,  6, 13, 28])
```

These operations are of course much faster than if you did them in pure python:

```
>>> a = np.arange(10000)
>>> %timeit a + 1
10000 loops, best of 3: 24.3 us per loop
>>> l = range(10000)
>>> %timeit [i+1 for i in l]
1000 loops, best of 3: 861 us per loop
```

Warning: Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c                   # NOT matrix multiplication!
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Note: Matrix multiplication:

```
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Exercise: Elementwise operations

- Try simple arithmetic elementwise operations: add even elements with odd elements
- Time them against their pure python counterparts using `%timeit`.
- Generate:
  - `[2**0, 2**1, 2**2, 2**3, 2**4]`
  - `a_j = 2^(3*j) - j`

Other operations

Comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
```

Tip: Array-wise comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> c = np.array([1, 2, 3, 4])
>>> np.array_equal(a, b)
False
>>> np.array_equal(a, c)
True
```

Logical operations:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)
>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
```

Transcendental functions:

```
>>> a = np.arange(5)
>>> np.sin(a)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
>>> np.log(a)
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436])
>>> np.exp(a)
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
```

Shape mismatches

```
>>> a = np.arange(4)
>>> a + np.array([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4) (2)
```

Broadcasting? We'll return to that *later*.

Transposition:

```
>>> a = np.triu(np.ones((3, 3)), 1)   # see help(np.triu)
>>> a
array([[ 0.,  1.,  1.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
>>> a.T
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  1.,  0.]])
```

Warning: The transposition is a view

As a results, the following code **is wrong** and will **not make a matrix symmetric**:

```
>>> a += a.T
```

It will work for small arrays (because of buffering) but fail for large one, in unpredictable ways.

Note: Linear algebra

The sub-module `numpy.linalg` implements basic linear algebra, such as solving linear systems, singular value decomposition, etc. However, it is not guaranteed to be compiled using efficient routines, and thus we recommend the use of `scipy.linalg`, as detailed in section *Linear algebra operations: scipy.linalg*

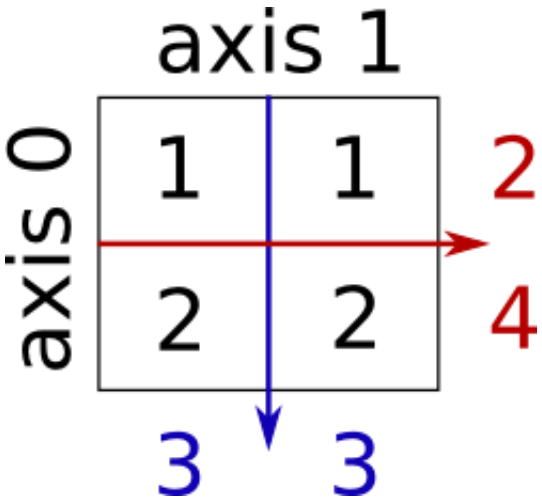
Exercise other operations

- Look at the help for `np.allclose`. When might this be useful?
- Look at the help for `np.triu` and `np.tril`.

1.2.2 Basic reductions

Computing sums

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```



Sum by rows and by columns:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)   # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1)   # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```

Tip: Same idea in higher dimensions:

```
>>> x = np.random.rand(2, 2, 2)
>>> x.sum(axis=2)[0, 1]
1.14764...
```

```
>>> x[0, 1, :].sum()
1.14764...
```

Other reductions

— works the same way (and take `axis=`)

Extrema:

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3

>>> x.argmin()  # index of minimum
0
>>> x.argmax()  # index of maximum
1
```

Logical operations:

```
>>> np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

Note: Can be used for array comparisons:

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True

>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
True
```

Statistics:

```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
1.75
>>> np.median(x)
1.5
>>> np.median(y, axis=-1) # last axis
array([ 2.,  5.])

>>> x.std()          # full population standard dev.
0.82915619758884995
```

... and many more (best to learn as you go).

Exercise: Reductions

- Given there is a `sum`, what other function might you expect to see?
- What is the difference between `sum` and `cumsum`?

Worked Example: data statistics

Data in `populations.txt` describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.

You can view the data in an editor, or alternatively in IPython (both shell and notebook):

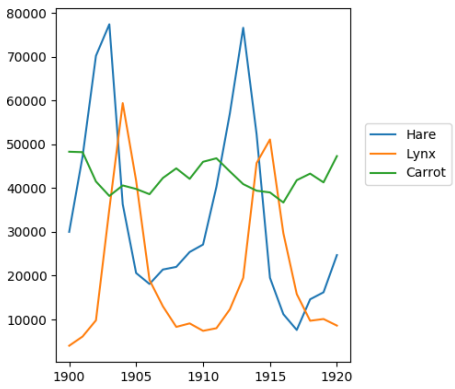
```
In [1]: !cat data/populations.txt
```

First, load the data into a NumPy array:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables
```

Then plot it:

```
>>> from matplotlib import pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
>>> plt.plot(year, hares, year, lynxes, year, carrots)
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
```



The mean populations over time:

```
>>> populations = data[:, 1:]
>>> populations.mean(axis=0)
array([ 34080.95238095,  20166.66666667,  42400.        ])
```

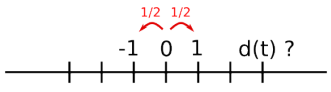
The sample standard deviations:

```
>>> populations.std(axis=0)
array([ 20897.90645809,  16254.59153691,   3322.50622558])
```

Which species has the highest population each year?:

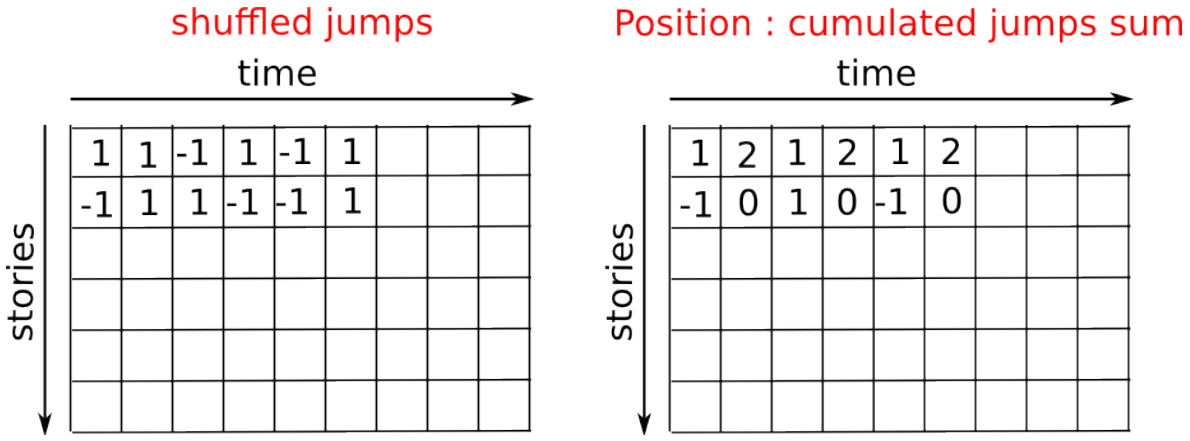
```
>>> np.argmax(populations, axis=1)
array([2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2])
```

Worked Example: diffusion using a random walk algorithm



Tip: Let us consider a simple 1D random walk process: at each time step a walker jumps right or left with equal probability.

We are interested in finding the typical distance from the origin of a random walker after `t` left or right jumps? We are going to simulate many "walkers" to find this law, and we are going to do so using array computing tricks: we are going to create a 2D array with the "stories" (each walker has a story) in one direction, and the time in the other:




```
>>> n_stories = 1000 # number of walkers
>>> t_max = 200      # time during which we follow the walker
```

We randomly choose all the steps 1 or -1 of the walk:

```
>>> t = np.arange(t_max)
>>> steps = 2 * np.random.randint(0, 1 + 1, (n_stories, t_max)) - 1 # +1 because the high␣
↪value is exclusive
>>> np.unique(steps) # Verification: all steps are 1 or -1
array([-1,  1])
```

We build the walks by summing steps along the time:

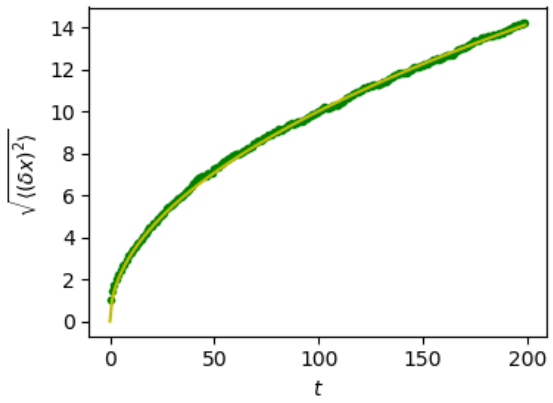
```
>>> positions = np.cumsum(steps, axis=1) # axis = 1: dimension of time
>>> sq_distance = positions**2
```

We get the mean in the axis of the stories:

```
>>> mean_sq_distance = np.mean(sq_distance, axis=0)
```

Plot the results:

```
>>> plt.figure(figsize=(4, 3))
<matplotlib.figure.Figure object at ...>
>>> plt.plot(t, np.sqrt(mean_sq_distance), 'g.', t, np.sqrt(t), 'y-')
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
>>> plt.xlabel(r"$t$")
<matplotlib.text.Text object at ...>
>>> plt.ylabel(r"$\sqrt{\langle (\delta x)^2 \rangle}$")
<matplotlib.text.Text object at ...>
>>> plt.tight_layout() # provide sufficient space for labels
```



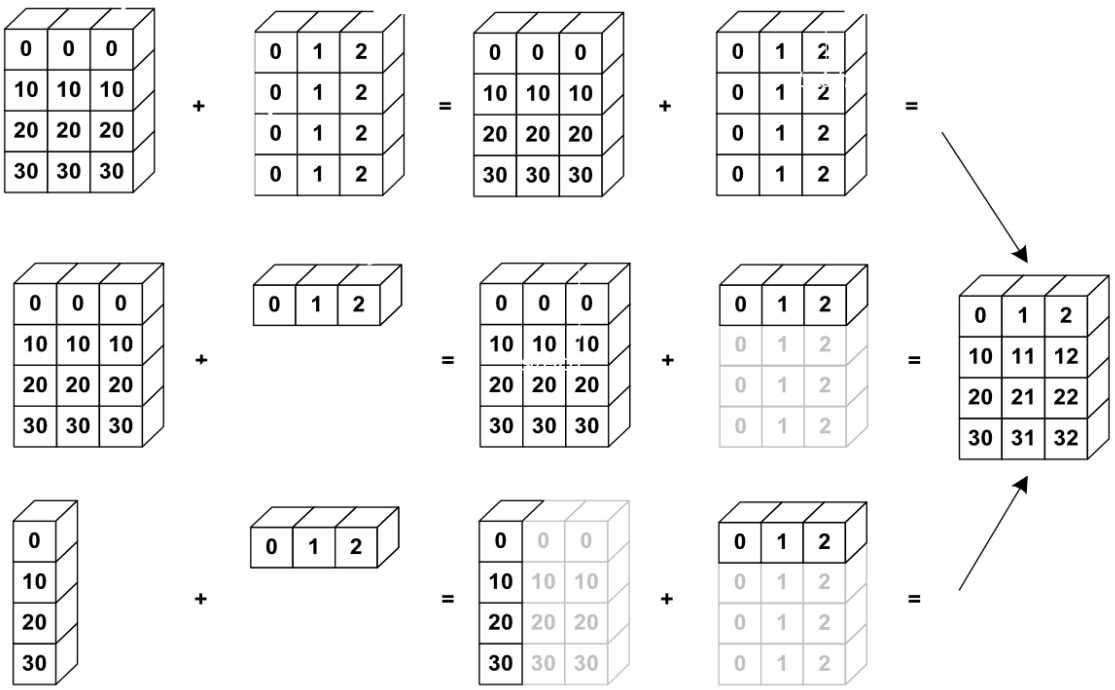
We find a well-known result in physics: the RMS distance grows as the square root of the time!

1.2.3 Broadcasting

- Basic operations on `numpy` arrays (addition, etc.) are elementwise
- This works on arrays of the same size.

  Nevertheless, It's also possible to do operations on arrays of different sizes if *NumPy* can transform these arrays so that they all have the same size: this conversion is called **broadcasting**.

The image below gives an example of broadcasting:



Let's verify:

```
>>> a = np.tile(np.arange(0, 40, 10), (3, 1)).T
>>> a
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
>>> b = np.array([0, 1, 2])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

We have already used broadcasting without knowing it!:

```
>>> a = np.ones((4, 5))
>>> a[0] = 2  # we assign an array of dimension 0 to an array of dimension 1
>>> a
array([[ 2.,  2.,  2.,  2.,  2.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

An useful trick:

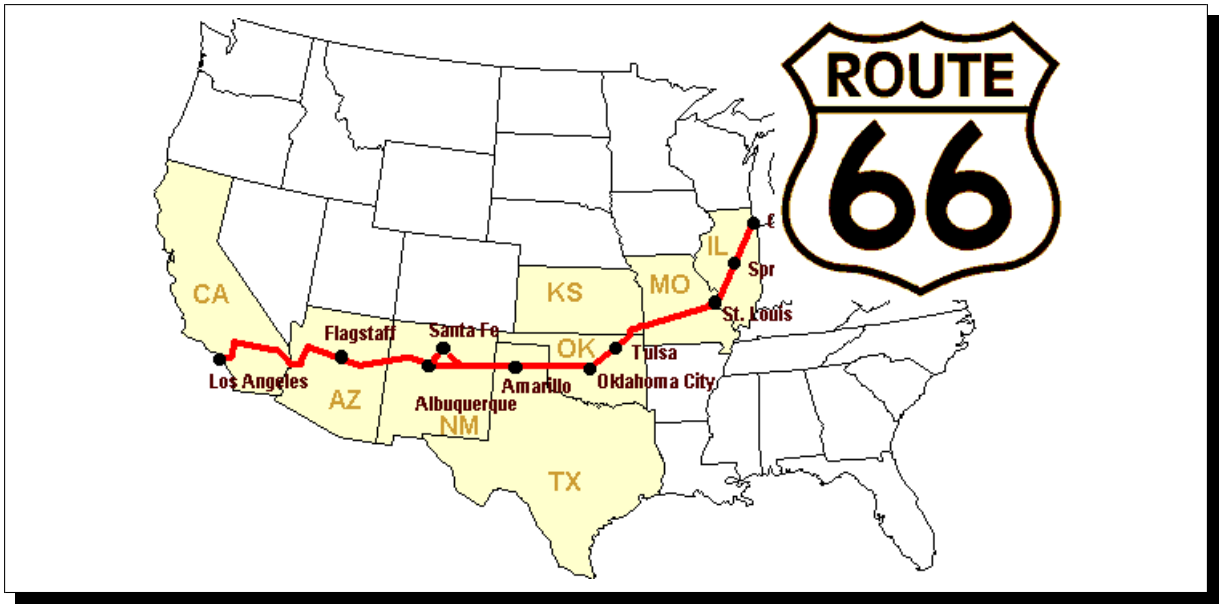
```
>>> a = np.arange(0, 40, 10)
>>> a.shape
(4,)
>>> a = a[:, np.newaxis]  # adds a new axis -> 2D array
>>> a.shape
(4, 1)
>>> a
array([[ 0],
       [10],
       [20],
       [30]])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Tip: Broadcasting seems a bit magical, but it is actually quite natural to use it when we want to solve a problem whose output data is an array with more dimensions than input data.

Worked Example: Broadcasting

Let's construct an array of distances (in miles) between cities of Route 66: Chicago, Springfield, Saint-Louis, Tulsa, Oklahoma City, Amarillo, Santa Fe, Albuquerque, Flagstaff and Los Angeles.

```
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544,
...        1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
array([[   0,  198,  303,  736,  871, 1175, 1475, 1544, 1913, 2448],
       [ 198,    0,  105,  538,  673,  977, 1277, 1346, 1715, 2250],
       [ 303,  105,    0,  433,  568,  872, 1172, 1241, 1610, 2145],
       [ 736,  538,  433,    0,  135,  439,  739,  808, 1177, 1712],
       [ 871,  673,  568,  135,    0,  304,  604,  673, 1042, 1577],
       [1175,  977,  872,  439,  304,    0,  300,  369,  738, 1273],
       [1475, 1277, 1172,  739,  604,  300,    0,   69,  438,  973],
       [1544, 1346, 1241,  808,  673,  369,   69,    0,  369,  904],
       [1913, 1715, 1610, 1177, 1042,  738,  438,  369,    0,  535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973,  904,  535,    0]])
```

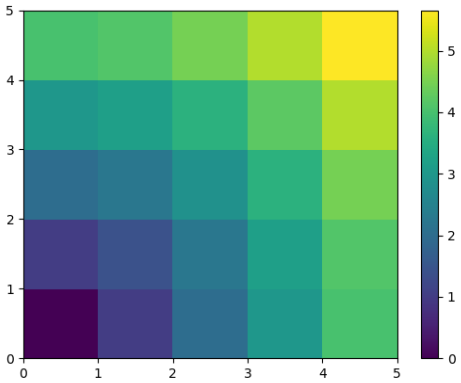


A lot of grid-based or network-based problems can also use broadcasting. For instance, if we want to compute the distance from the origin of points on a 10x10 grid, we can do

```
>>> x, y = np.arange(5), np.arange(5)[:, np.newaxis]
>>> distance = np.sqrt(x ** 2 + y ** 2)
>>> distance
array([[ 0.        ,  1.        ,  2.        ,  3.        ,  4.        ],
       [ 1.        ,  1.41421356,  2.23606798,  3.16227766,  4.12310563],
       [ 2.        ,  2.23606798,  2.82842712,  3.60555128,  4.47213595],
       [ 3.        ,  3.16227766,  3.60555128,  4.24264069,  5.        ],
       [ 4.        ,  4.12310563,  4.47213595,  5.        ,  5.65685425]])
```

Or in color:

```
>>> plt.pcolor(distance)
>>> plt.colorbar()
```



Remark : the `numpy.ogrid()` function allows to directly create vectors x and y of the previous example, with two "significant dimensions":

```
>>> x, y = np.ogrid[0:5, 0:5]
>>> x, y
(array([[0],
       [1],
       [2],
       [3],
       [4]]), array([[0, 1, 2, 3, 4]]))
```

```
>>> x.shape, y.shape
((5, 1), (1, 5))
>>> distance = np.sqrt(x ** 2 + y ** 2)
```

Tip: So, `np.ogrid` is very useful as soon as we have to handle computations on a grid. On the other hand, `np.mgrid` directly provides matrices full of indices for cases where we can't (or don't want to) benefit from broadcasting:

```
>>> x, y = np.mgrid[0:4, 0:4]
>>> x
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])
>>> y
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])
```

See also:

Broadcasting: discussion of broadcasting in the *Advanced NumPy* chapter.

1.2.4 Array shape manipulation

Flattening

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

Higher dimensions: last dimensions ravel out "first".

Reshaping

The inverse operation to flattening:

```
>>> a.shape
(2, 3)
>>> b = a.ravel()
>>> b = b.reshape((2, 3))
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
```

Or,

```
>>> a.reshape((2, -1))    # unspecified (-1) value is inferred
array([[1, 2, 3],
       [4, 5, 6]])
```

Warning: `ndarray.reshape` **may** return a view (cf `help(np.reshape)`), or copy

Tip:

```
>>> b[0, 0] = 99
>>> a
array([[99,  2,  3],
       [ 4,  5,  6]])
```

Beware: reshape may also return a copy!:

```
>>> a = np.zeros((3, 2))
>>> b = a.T.reshape(3*2)
>>> b[0] = 9
>>> a
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

To understand this you need to learn more about the memory layout of a numpy array.

Adding a dimension

Indexing with the `np.newaxis` object allows us to add an axis to an array (you have seen this already above in the broadcasting section):

```
>>> z = np.array([1, 2, 3])
>>> z
array([1, 2, 3])

>>> z[:, np.newaxis]
array([[1],
       [2],
       [3]])

>>> z[np.newaxis, :]
array([[1, 2, 3]])
```

Dimension shuffling

```
>>> a = np.arange(4*3*2).reshape(4, 3, 2)
>>> a.shape
(4, 3, 2)
>>> a[0, 2, 1]
5
>>> b = a.transpose(1, 2, 0)
>>> b.shape
(3, 2, 4)
>>> b[2, 1, 0]
5
```

Also creates a view:

```
>>> b[2, 1, 0] = -1
>>> a[0, 2, 1]
-1
```

Resizing

Size of an array can be changed with `ndarray.resize`:

```
>>> a = np.arange(4)
>>> a.resize((8,))
>>> a
array([0, 1, 2, 3, 0, 0, 0, 0])
```

However, it must not be referred to somewhere else:

```
>>> b = a
>>> a.resize((4,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize an array that has been referenced or is
referencing another array in this way.  Use the resize function
```

Exercise: Shape manipulations

- Look at the docstring for `reshape`, especially the notes section which has some more information about copies and views.
- Use `flatten` as an alternative to `ravel`. What is the difference? (Hint: check which one returns a view and which a copy)
- Experiment with `transpose` for dimension shuffling.

1.2.5 Sorting data

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

Note: Sorts each row separately!

In-place sort:

```
>>> a.sort(axis=1)
>>> a
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

Finding minima and maxima:

```
>>> a = np.array([4, 3, 1, 2])
>>> j_max = np.argmax(a)
>>> j_min = np.argmin(a)
>>> j_max, j_min
(0, 2)
```

Exercise: Sorting

- Try both in-place and out-of-place sorting.
- Try creating arrays with different dtypes and sorting them.
- Use `all` or `array_equal` to check the results.
- Look at `np.random.shuffle` for a way to create sortable input quicker.
- Combine `ravel`, `sort` and `reshape`.
- Look at the `axis` keyword for `sort` and rewrite the previous exercise.

1.2.6 Summary

What do you need to know to get started?

- Know how to create arrays : `array`, `arange`, `ones`, `zeros`.
- Know the shape of the array with `array.shape`, then use slicing to obtain different views of the array: `array[::2]`, etc. Adjust the shape of the array using `reshape` or flatten it with `ravel`.
- Obtain a subset of the elements of an array and/or modify their values with masks

  ```
  >>> a[a < 0] = 0
  ```

- Know miscellaneous operations on arrays, such as finding the mean or max (`array.max()`, `array.mean()`). No need to retain everything, but have the reflex to search in the documentation (online docs, `help()`, `lookfor()`)!!
- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more NumPy functions to handle various array operations.

Quick read

If you want to do a first quick pass through the Scipy lectures to learn the ecosystem, you can directly skip to the next chapter: *Matplotlib: plotting*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter, as well as to do some more *exercices*.

1.3 More elaborate arrays

Section contents

- *More data types*
- *Structured data types*
- *`maskedarray`: dealing with (propagation of) missing data*

1.3.1 More data types

Casting

"Bigger" type wins in mixed-type operations:

```
>>> np.array([1, 2, 3]) + 1.5
array([ 2.5,  3.5,  4.5])
```

Assignment never changes the type!

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a[0] = 1.9     # <-- float is truncated to integer
>>> a
array([1, 2, 3])
```

Forced casts:

```
>>> a = np.array([1.7, 1.2, 1.6])
>>> b = a.astype(int)  # <-- truncates to integer
>>> b
array([1, 1, 1])
```

Rounding:

```
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
>>> b = np.around(a)
>>> b                    # still floating-point
array([ 1.,  2.,  2.,  2.,  4.,  4.])
>>> c = np.around(a).astype(int)
>>> c
array([1, 2, 2, 2, 4, 4])
```

Different data type sizes

Integers (signed):

`int8`	8 bits
`int16`	16 bits
`int32`	32 bits (same as `int` on 32-bit platform)
`int64`	64 bits (same as `int` on 64-bit platform)

```
>>> np.array([1], dtype=int).dtype
dtype('int64')
>>> np.iinfo(np.int32).max, 2**31 - 1
(2147483647, 2147483647)
```

Unsigned integers:

`uint8`	8 bits
`uint16`	16 bits
`uint32`	32 bits
`uint64`	64 bits

```
>>> np.iinfo(np.uint32).max, 2**32 - 1
(4294967295, 4294967295)
```

Long integers

Python 2 has a specific type for 'long' integers, that cannot overflow, represented with an 'L' at the end. In Python 3, all integers are long, and thus cannot overflow.

```
>>> np.iinfo(np.int64).max, 2**63 - 1
(9223372036854775807, 9223372036854775807L)
```

Floating-point numbers:

`float16`	16 bits
`float32`	32 bits
`float64`	64 bits (same as `float`)
`float96`	96 bits, platform-dependent (same as `np.longdouble`)
`float128`	128 bits, platform-dependent (same as `np.longdouble`)

```
>>> np.finfo(np.float32).eps
1.1920929e-07
>>> np.finfo(np.float64).eps
2.2204460492503131e-16

>>> np.float32(1e-8) + np.float32(1) == 1
True
>>> np.float64(1e-8) + np.float64(1) == 1
False
```

Complex floating-point numbers:

`complex64`	two 32-bit floats
`complex128`	two 64-bit floats
`complex192`	two 96-bit floats, platform-dependent
`complex256`	two 128-bit floats, platform-dependent

Smaller data types

If you don't know you need special data types, then you probably don't.

Comparison on using `float32` instead of `float64`:

- Half the size in memory and on disk
- Half the memory bandwidth required (may be a bit faster in some operations)

```
In [1]: a = np.zeros((1e6,), dtype=np.float64)

In [2]: b = np.zeros((1e6,), dtype=np.float32)

In [3]: %timeit a*a
1000 loops, best of 3: 1.78 ms per loop

In [4]: %timeit b*b
1000 loops, best of 3: 1.07 ms per loop
```

- But: bigger rounding errors — sometimes in surprising places (i.e., don't use them unless you really need them)

1.3.2 Structured data types

sensor_code	(4-character string)
position	(float)
value	(float)

```
>>> samples = np.zeros((6,), dtype=[('sensor_code', 'S4'),
...                                 ('position', float), ('value', float)])
>>> samples.ndim
1
>>> samples.shape
(6,)
>>> samples.dtype.names
('sensor_code', 'position', 'value')

>>> samples[:] = [('ALFA',   1, 0.37), ('BETA', 1, 0.11), ('TAU', 1,   0.13),
...               ('ALFA', 1.5, 0.37), ('ALFA', 3, 0.11), ('TAU', 1.2, 0.13)]
>>> samples
array([('ALFA', 1.0, 0.37), ('BETA', 1.0, 0.11), ('TAU', 1.0, 0.13),
       ('ALFA', 1.5, 0.37), ('ALFA', 3.0, 0.11), ('TAU', 1.2, 0.13)],
      dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')])
```

Field access works by indexing with field names:

```
>>> samples['sensor_code']
array(['ALFA', 'BETA', 'TAU', 'ALFA', 'ALFA', 'TAU'],
      dtype='|S4')
>>> samples['value']
array([ 0.37,  0.11,  0.13,  0.37,  0.11,  0.13])
>>> samples[0]
('ALFA', 1.0, 0.37)

>>> samples[0]['sensor_code'] = 'TAU'
>>> samples[0]
('TAU', 1.0, 0.37)
```

Multiple fields at once:

```
>>> samples[['position', 'value']]
array([(1.0, 0.37), (1.0, 0.11), (1.0, 0.13), (1.5, 0.37), (3.0, 0.11),
       (1.2, 0.13)],
      dtype=[('position', '<f8'), ('value', '<f8')])
```

Fancy indexing works, as usual:

```
>>> samples[samples['sensor_code'] == 'ALFA']
array([('ALFA', 1.5, 0.37), ('ALFA', 3.0, 0.11)],
      dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')])
```

Note: There are a bunch of other syntaxes for constructing structured arrays, see here and here.

1.3.3 `maskedarray`: dealing with (propagation of) missing data

- For floats one could use NaN's, but masks work for all types:

```
>>> x = np.ma.array([1, 2, 3, 4], mask=[0, 1, 0, 1])
>>> x
```

```
masked_array(data = [1 -- 3 --],
             mask = [False  True False  True],
       fill_value = 999999)


>>> y = np.ma.array([1, 2, 3, 4], mask=[0, 1, 1, 1])
>>> x + y
masked_array(data = [2 -- -- --],
             mask = [False  True  True  True],
       fill_value = 999999)
```

- Masking versions of common functions:

```
>>> np.ma.sqrt([1, -1, 2, -2])
masked_array(data = [1.0 -- 1.41421356237... --],
             mask = [False  True False  True],
       fill_value = 1e+20)
```

Note: There are other useful *array siblings*

While it is off topic in a chapter on numpy, let's take a moment to recall good coding practice, which really do pay off in the long run:

Good practices

- Explicit variable names (no need of a comment to explain what is in the variable)
- Style: spaces after commas, around =, etc.

  A certain number of rules for writing "beautiful" code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).
- Except some rare cases, variable names and comments in English.

1.4 Advanced operations

Section contents

1.4.1 Polynomials

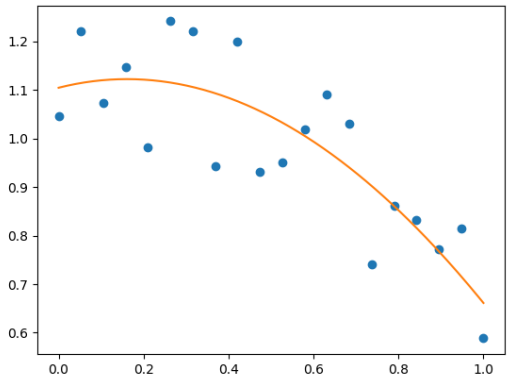
NumPy also contains polynomials in different bases:

For example, $3x^2 + 2x - 1$:

```
>>> p = np.poly1d([3, 2, -1])
>>> p(0)
-1
>>> p.roots
```

```
array([-1.        ,  0.33333333])
>>> p.order
2
```

```
>>> x = np.linspace(0, 1, 20)
>>> y = np.cos(x) + 0.3*np.random.rand(20)
>>> p = np.poly1d(np.polyfit(x, y, 3))

>>> t = np.linspace(0, 1, 200)
>>> plt.plot(x, y, 'o', t, p(t), '-')
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
```



See http://docs.scipy.org/doc/numpy/reference/routines.polynomials.poly1d.html for more.

More polynomials (with more bases)

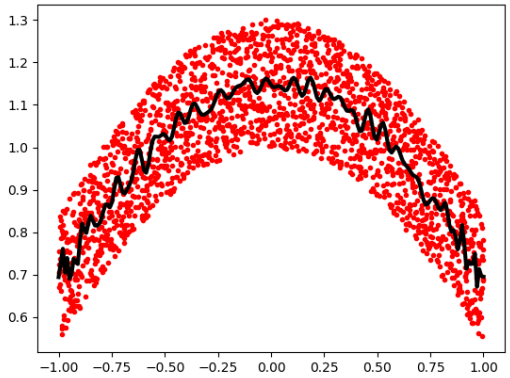
NumPy also has a more sophisticated polynomial interface, which supports e.g. the Chebyshev basis.

$3x^2 + 2x - 1$:

```
>>> p = np.polynomial.Polynomial([-1, 2, 3]) # coefs in different order!
>>> p(0)
-1.0
>>> p.roots()
array([-1.        ,  0.33333333])
>>> p.degree()  # In general polynomials do not always expose 'order'
2
```

Example using polynomials in Chebyshev basis, for polynomials in range `[-1, 1]`:

```
>>> x = np.linspace(-1, 1, 2000)
>>> y = np.cos(x) + 0.3*np.random.rand(2000)
>>> p = np.polynomial.Chebyshev.fit(x, y, 90)

>>> t = np.linspace(-1, 1, 200)
>>> plt.plot(x, y, 'r.')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t, p(t), 'k-', lw=3)
[<matplotlib.lines.Line2D object at ...>]
```



The Chebyshev polynomials have some advantages in interpolation.

1.4.2 Loading data files

Text files

Example: `populations.txt`:

```
# year  hare    lynx    carrot
1900    30e3    4e3     48300
1901    47.2e3  6.1e3   48200
1902    70.2e3  9.8e3   41500
1903    77.4e3  35.2e3  38200
```

```
>>> data = np.loadtxt('data/populations.txt')
>>> data
array([[  1900.,  30000.,   4000.,  48300.],
       [  1901.,  47200.,   6100.,  48200.],
       [  1902.,  70200.,   9800.,  41500.],
...
```

```
>>> np.savetxt('pop2.txt', data)
>>> data2 = np.loadtxt('pop2.txt')
```

Note: If you have a complicated text file, what you can try are:

- `np.genfromtxt`
- Using Python's I/O functions and e.g. regexps for parsing (Python is quite well suited for this)

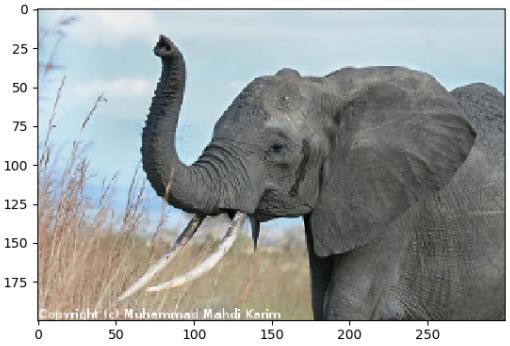
Reminder: Navigating the filesystem with IPython

```
In [1]: pwd      # show current directory
'/home/user/stuff/2011-numpy-tutorial'
In [2]: cd ex
'/home/user/stuff/2011-numpy-tutorial/ex'
In [3]: ls
populations.txt  species.txt
```

Images

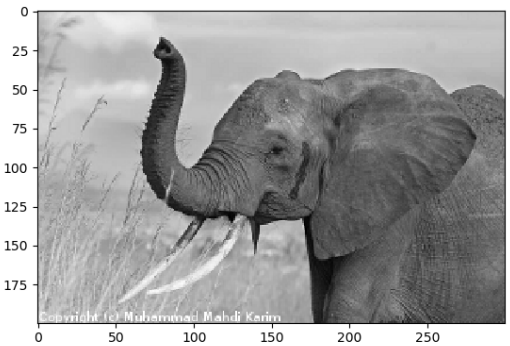
Using Matplotlib:

```
>>> img = plt.imread('data/elephant.png')
>>> img.shape, img.dtype
((200, 300, 3), dtype('float32'))
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at ...>
>>> plt.savefig('plot.png')

>>> plt.imsave('red_elephant.png', img[:,:,0], cmap=plt.cm.gray)
```



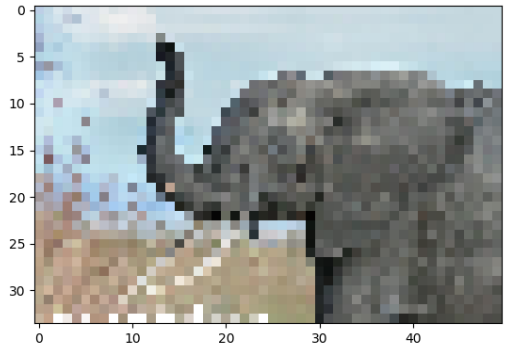
This saved only one channel (of RGB):

```
>>> plt.imshow(plt.imread('red_elephant.png'))
<matplotlib.image.AxesImage object at ...>
```



Other libraries:

```
>>> from scipy.misc import imsave
>>> imsave('tiny_elephant.png', img[::6,::6])
>>> plt.imshow(plt.imread('tiny_elephant.png'), interpolation='nearest')
<matplotlib.image.AxesImage object at ...>
```



NumPy's own format

NumPy has its own binary format, not portable but with efficient I/O:

```
>>> data = np.ones((3, 3))
>>> np.save('pop.npy', data)
>>> data3 = np.load('pop.npy')
```

Well-known (& more obscure) file formats

- HDF5: h5py, PyTables
- NetCDF: `scipy.io.netcdf_file`, netcdf4-python, ...
- Matlab: `scipy.io.loadmat`, `scipy.io.savemat`
- MatrixMarket: `scipy.io.mmread`, `scipy.io.mmwrite`
- IDL: `scipy.io.readsav`

... if somebody uses it, there's probably also a Python library for it.

Exercise: Text data files

Write a Python script that loads data from `populations.txt`:: and drop the last column and the first 5 rows. Save the smaller dataset to `pop2.txt`.

NumPy internals

If you are interested in the NumPy internals, there is a good discussion in *Advanced NumPy*.

1.5 Some exercises

1.5.1 Array manipulations

1. Form the 2-D array (without typing it in explicitly):

```
[[1,  6, 11],
 [2,  7, 12],
 [3,  8, 13],
 [4,  9, 14],
 [5, 10, 15]]
```

and generate a new array containing its 2nd and 4th rows.

2. Divide each column of the array:

```
>>> import numpy as np
>>> a = np.arange(25).reshape(5, 5)
```

elementwise with the array `b = np.array([1., 5, 10, 15, 20])`. (Hint: `np.newaxis`).

3. Harder one: Generate a 10 x 3 array of random numbers (in range [0,1]). For each row, pick the number closest to 0.5.

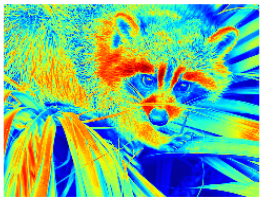
   - Use `abs` and `argsort` to find the column `j` closest for each row.
   - Use fancy indexing to extract the numbers. (Hint: `a[i,j]` – the array `i` must contain the row numbers corresponding to stuff in `j`.)

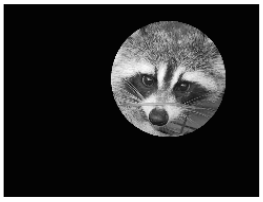
1.5.2 Picture manipulation: Framing a Face

Let's do some manipulations on numpy arrays by starting with an image of a racoon. `scipy` provides a 2D array of this image with the `scipy.misc.face` function:

```
>>> from scipy import misc
>>> face = misc.face(gray=True)  # 2D grayscale image
```

Here are a few images we will be able to obtain with our manipulations: use different colormaps, crop the image, change some parts of the image.



- Let's use the imshow function of pylab to display the image.

```
>>> import pylab as plt
>>> face = misc.face(gray=True)
>>> plt.imshow(face)
<matplotlib.image.AxesImage object at 0x...>
```

- **The face is displayed in false colors. A colormap must be** specified for it to be displayed in grey.

```
>>> plt.imshow(face, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>
```

- **Create an array of the image with a narrower centering** [for example,] remove 100 pixels from all the borders of the image. To check the result, display this new array with `imshow`.

```
>>> crop_face = face[100:-100, 100:-100]
```

- **We will now frame the face with a black locket. For this, we** need to create a mask corresponding to the pixels we want to be black. The center of the face is around (660, 330), so we defined the mask by this condition `(y-300)**2 + (x-660)**2`

```
>>> sy, sx = face.shape
>>> y, x = np.ogrid[0:sy, 0:sx] # x and y indices of pixels
>>> y.shape, x.shape
((768, 1), (1, 1024))
>>> centerx, centery = (660, 300) # center of the image
>>> mask = ((y - centery)**2 + (x - centerx)**2) > 230**2 # circle
```

then we assign the value 0 to the pixels of the image corresponding to the mask. The syntax is extremely simple and intuitive:

```
>>> face[mask] = 0
>>> plt.imshow(face)
<matplotlib.image.AxesImage object at 0x...>
```

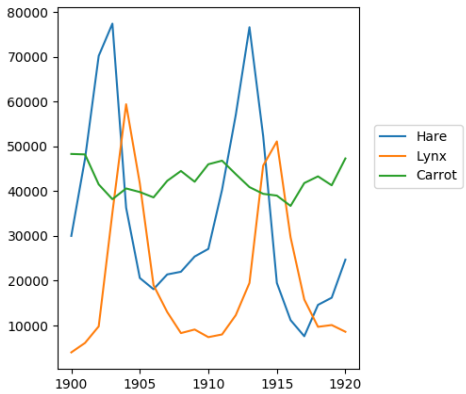
- **Follow-up: copy all instructions of this exercise in a script called** `face_locket.py` then execute this script in IPython with `%run face_locket.py`.

  Change the circle to an ellipsoid.

1.5.3 Data statistics

The data in `populations.txt` describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables

>>> import matplotlib.pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
<matplotlib.axes...Axes object at ...>
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at ...>, ...]
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at ...>
```



Computes and print, based on the data in `populations.txt`...

1. The mean and std of the populations of each species for the years in the period.
2. Which year each species had the largest population.
3. Which species has the largest population for each year. (Hint: `argsort` & fancy indexing of `np.array(['H', 'L', 'C'])`)
4. Which years any of the populations is above 50000. (Hint: comparisons and `np.any`)
5. The top 2 years for each species when they had the lowest populations. (Hint: `argsort`, fancy indexing)

6. Compare (plot) the change in hare population (see `help(np.gradient)`) and the number of lynxes. Check correlation (see `help(np.corrcoef)`).

... all without for-loops.

Solution: `Python source file`

1.5.4 Crude integral approximations

Write a function `f(a, b, c)` that returns $a^b - c$. Form a 24x12x6 array containing its values in parameter ranges `[0,1] x [0,1] x [0,1]`.

Approximate the 3-d integral

$$\int_0^1 \int_0^1 \int_0^1 (a^b - c) da db dc$$

over this volume with the mean. The exact result is: $\ln 2 - \frac{1}{2} \approx 0.1931\ldots$ — what is your relative error?

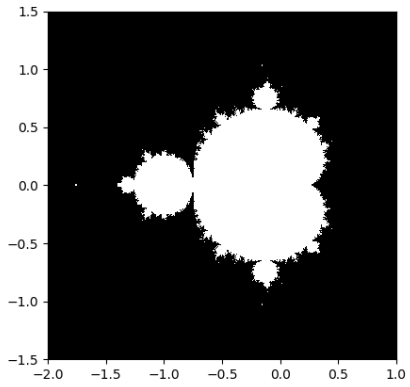
(Hints: use elementwise operations and broadcasting. You can make `np.ogrid` give a number of points in given range with `np.ogrid[0:1:20j]`.)

Reminder Python functions:

```
def f(a, b, c):
    return some_result
```

Solution: `Python source file`

1.5.5 Mandelbrot set



Write a script that computes the Mandelbrot fractal.

The Mandelbrot iteration:

```
N_max = 50
some_threshold = 50

c = x + 1j*y

z = 0
for j in range(N_max):
    z = z**2 + c
```

Point (x, y) belongs to the Mandelbrot set if $|z| <$ `some_threshold`.

Do this computation by:

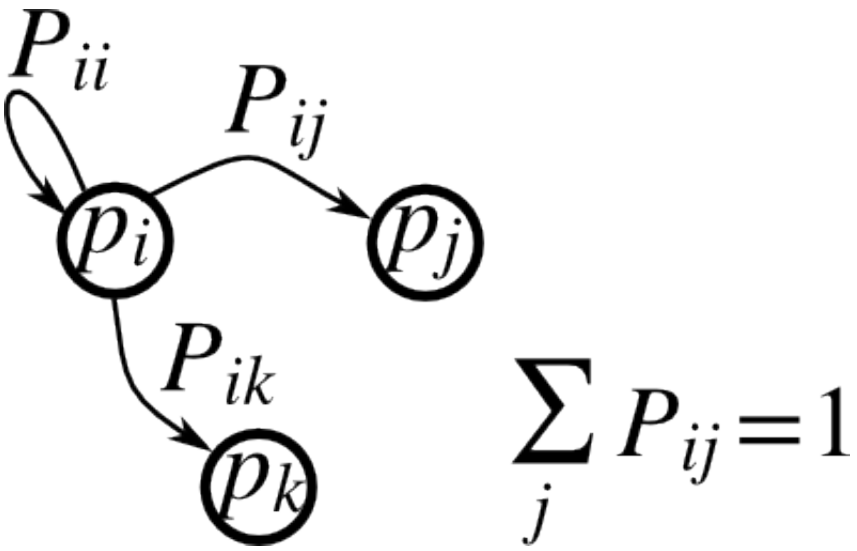
1. Construct a grid of c = x + 1j*y values in range [-2, 1] x [-1.5, 1.5]

2. Do the iteration
3. Form the 2-d boolean mask indicating which points are in the set
4. Save the result to an image with:

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(mask.T, extent=[-2, 1, -1.5, 1.5])
<matplotlib.image.AxesImage object at ...>
>>> plt.gray()
>>> plt.savefig('mandelbrot.png')
```

Solution: `Python source file`

1.5.6 Markov chain



Markov chain transition matrix `P`, and probability distribution on the states `p`:

1. `0 <= P[i,j] <= 1`: probability to go from state `i` to state `j`
2. Transition rule: $p_{new} = P^T p_{old}$
3. `all(sum(P, axis=1) == 1)`, `p.sum() == 1`: normalization

Write a script that works with 5 states, and:

- Constructs a random matrix, and normalizes each row so that it is a transition matrix.
- Starts from a random (normalized) probability distribution `p` and takes 50 steps => `p_50`
- Computes the stationary distribution: the eigenvector of `P.T` with eigenvalue 1 (numerically: closest to 1) => `p_stationary`

Remember to normalize the eigenvector — I didn't...

- Checks if `p_50` and `p_stationary` are equal to tolerance 1e-5

Toolbox: `np.random.rand`, `.dot()`, `np.linalg.eig`, reductions, `abs()`, `argmin`, comparisons, `all`, `np.linalg.norm`, etc.

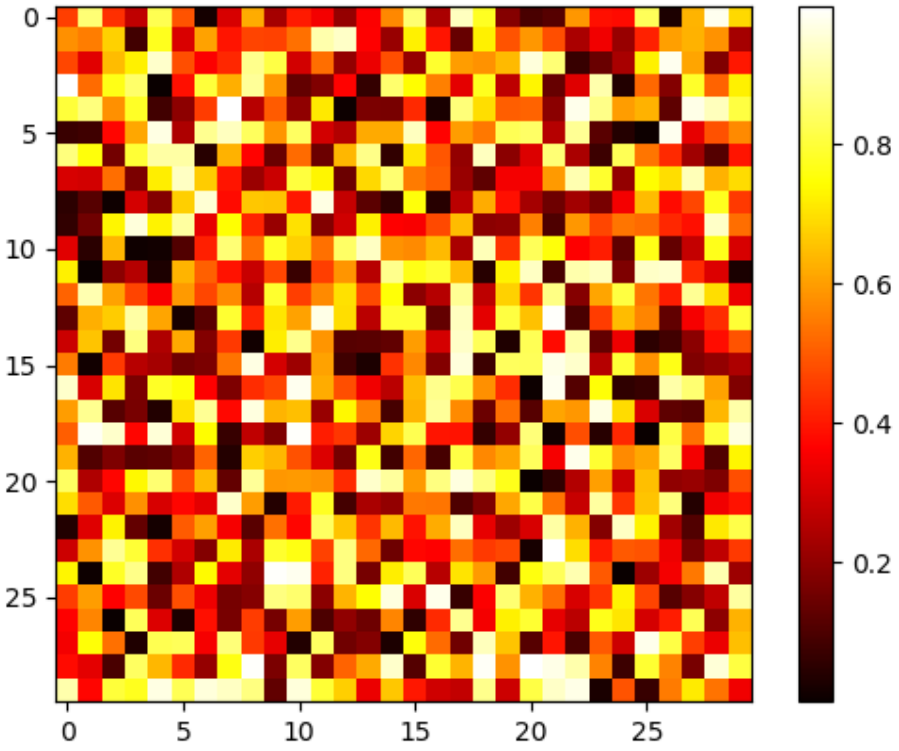
Solution: `Python source file`

1.6 Full code examples

1.6.1 Full code examples for the numpy chapter

2D plotting

Plot a basic 2D figure

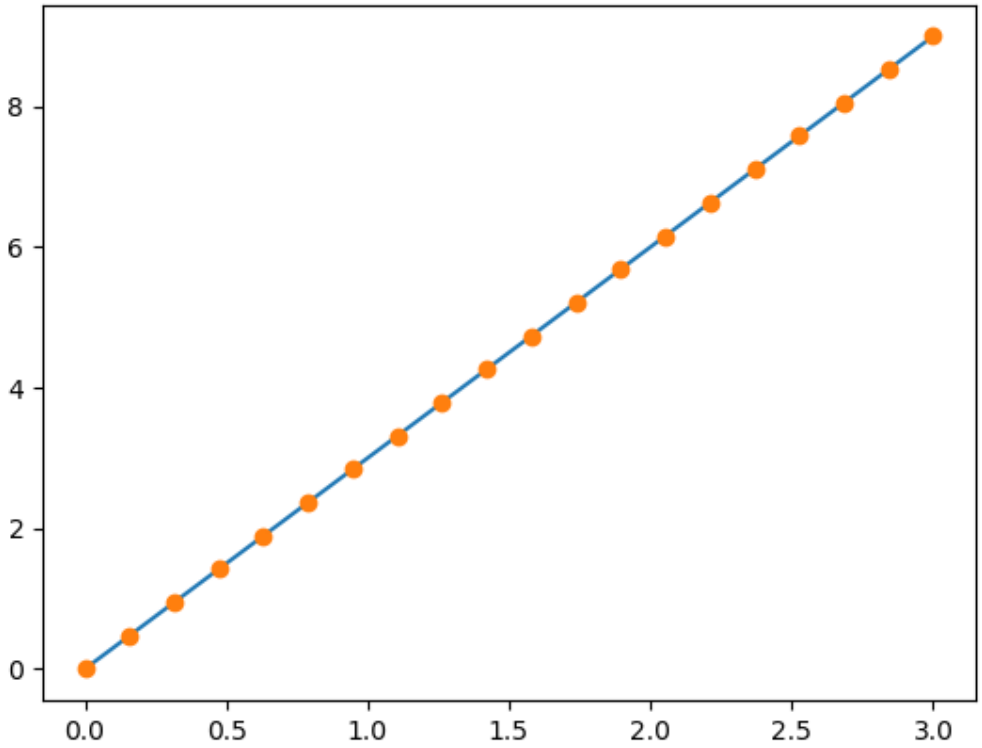



```
import numpy as np
import matplotlib.pyplot as plt

image = np.random.rand(30, 30)
plt.imshow(image, cmap=plt.cm.hot)
plt.colorbar()
plt.show()
```

Total running time of the script: (0 minutes 0.110 seconds)

1D plotting

Plot a basic 1D figure

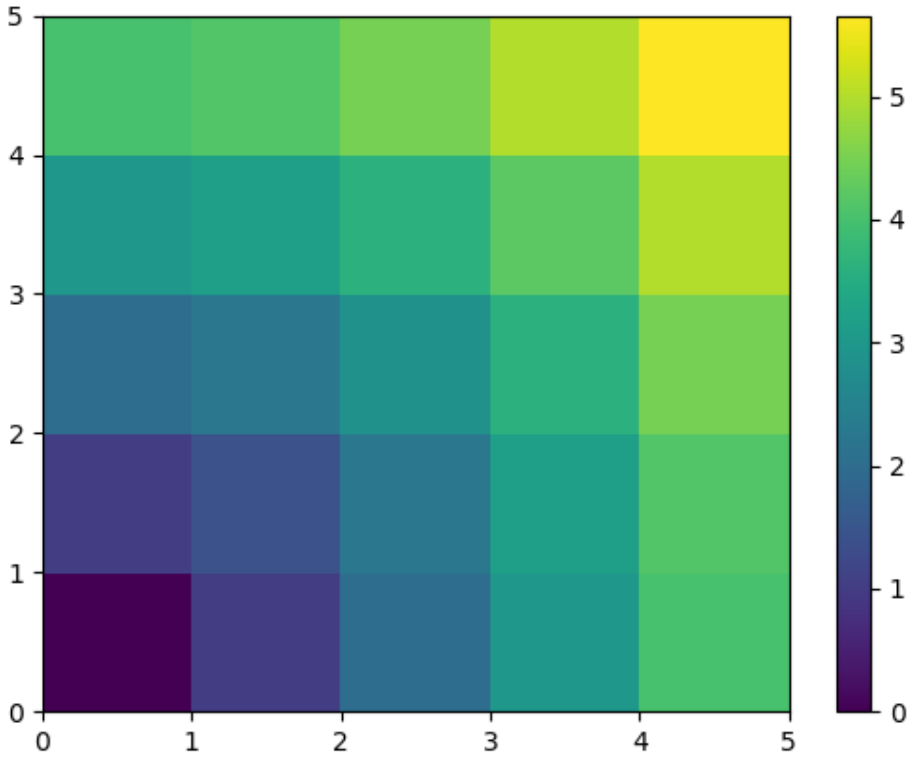



```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 3, 20)
y = np.linspace(0, 9, 20)
plt.plot(x, y)
plt.plot(x, y, 'o')
plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

Distances exercise

Plot distances in a grid

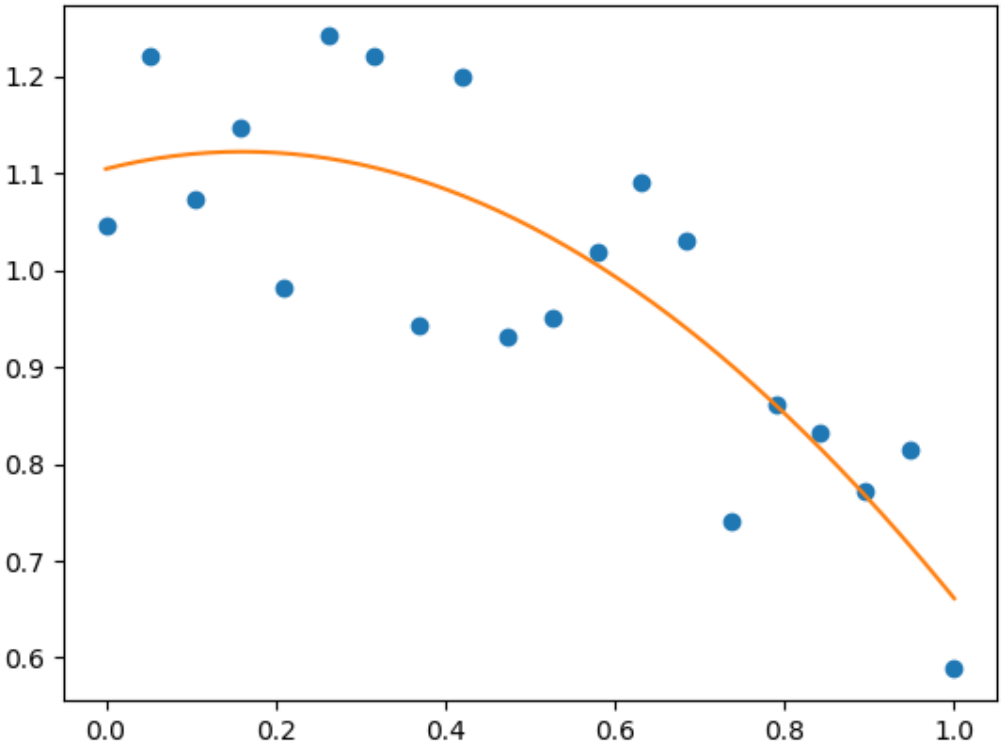



```
import numpy as np
import matplotlib.pyplot as plt

x, y = np.arange(5), np.arange(5)[:, np.newaxis]
distance = np.sqrt(x ** 2 + y ** 2)
plt.pcolor(distance)
plt.colorbar()
plt.show()
```

Total running time of the script: (0 minutes 0.116 seconds)

Fitting to polynomial

Plot noisy data and their polynomial fit

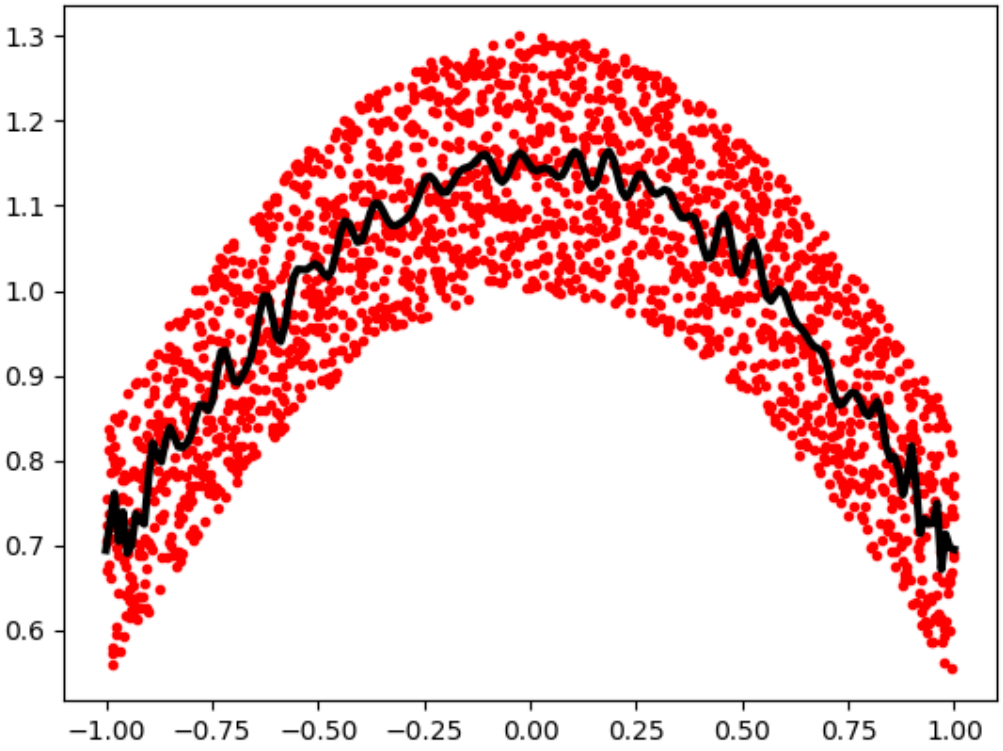



```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(12)

x = np.linspace(0, 1, 20)
y = np.cos(x) + 0.3*np.random.rand(20)
p = np.poly1d(np.polyfit(x, y, 3))

t = np.linspace(0, 1, 200)
plt.plot(x, y, 'o', t, p(t), '-')
plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

Fitting in Chebyshev basis

Plot noisy data and their polynomial fit in a Chebyshev basis




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

np.random.seed(0)

x = np.linspace(-1, 1, 2000)
y = np.cos(x) + 0.3*np.random.rand(2000)
p = np.polynomial.Chebyshev.fit(x, y, 90)

t = np.linspace(-1, 1, 200)
plt.plot(x, y, 'r.')
plt.plot(t, p(t), 'k-', lw=3)
plt.show()
```

Total running time of the script: (0 minutes 0.158 seconds)

Population exercise

Plot populations of hares, lynxes, and carrots


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

data = np.loadtxt('../data/populations.txt')
year, hares, lynxes, carrots = data.T

plt.axes([0.2, 0.1, 0.5, 0.8])
plt.plot(year, hares, year, lynxes, year, carrots)
plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
plt.show()
```

Total running time of the script: (0 minutes 0.055 seconds)

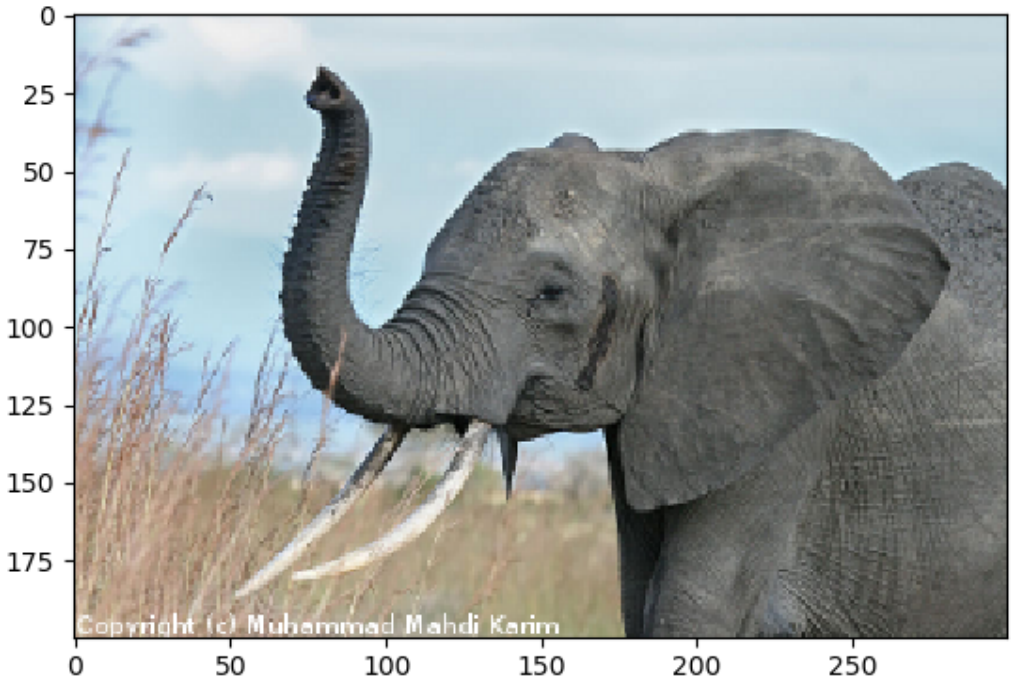
Reading and writing an elephant

Read and write images

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

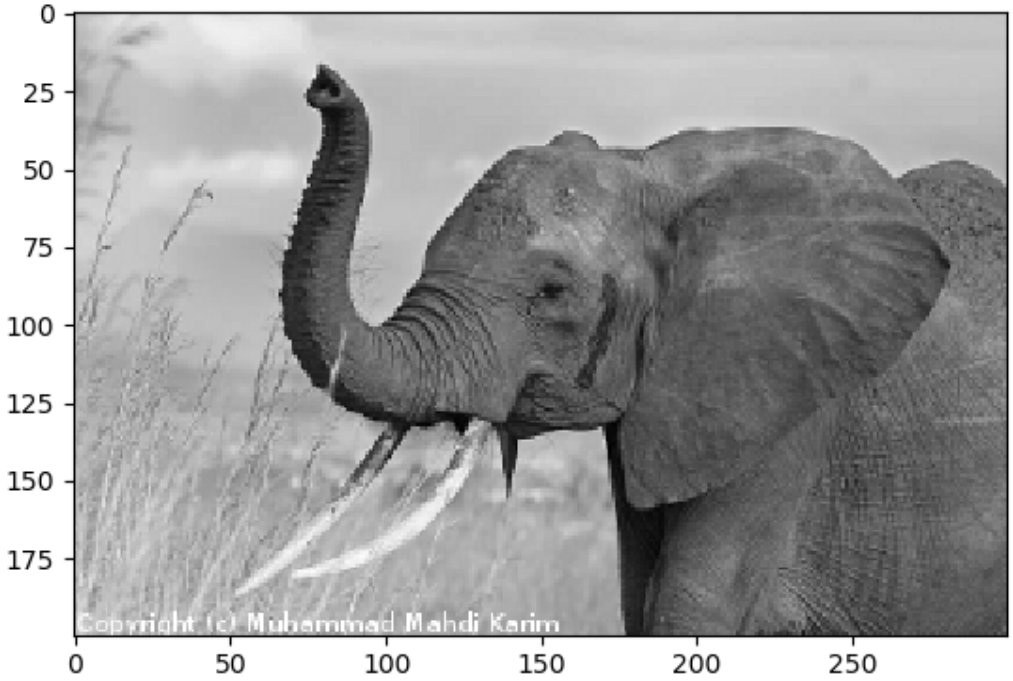
original figure

```
plt.figure()
img = plt.imread('../data/elephant.png')
plt.imshow(img)
```



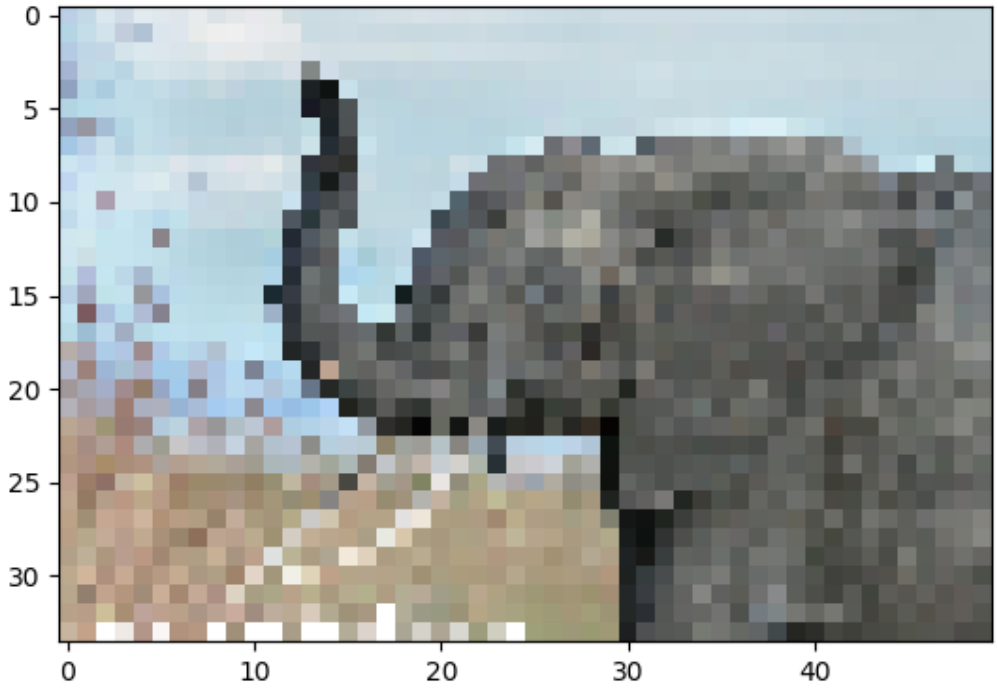
red channel displayed in grey

```
plt.figure()
img_red = img[:, :, 0]
plt.imshow(img_red, cmap=plt.cm.gray)
```



lower resolution

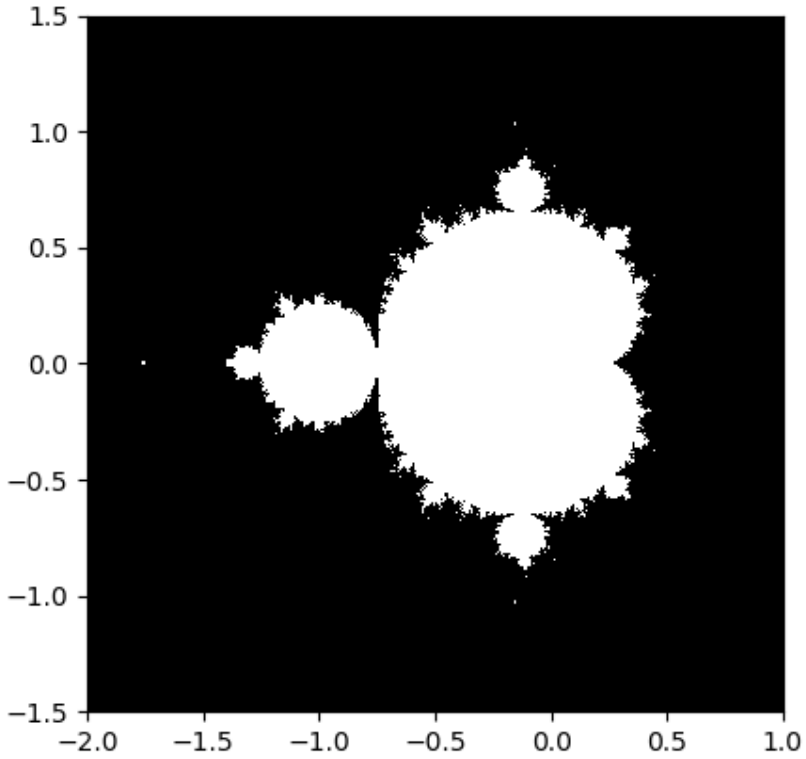
```
plt.figure()
img_tiny = img[::6, ::6]
plt.imshow(img_tiny, interpolation='nearest')

plt.show()
```



Total running time of the script: (0 minutes 0.217 seconds)

Mandelbrot set

Compute the Mandelbrot fractal and plot it

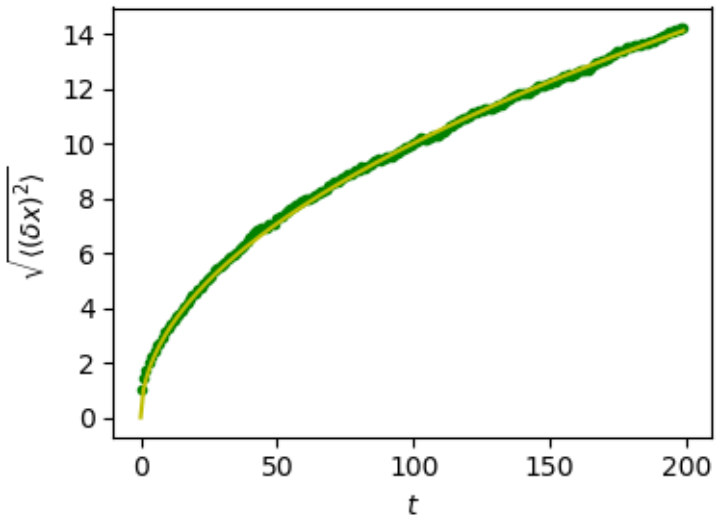



```
import numpy as np
import matplotlib.pyplot as plt
from numpy import newaxis

def compute_mandelbrot(N_max, some_threshold, nx, ny):
    # A grid of c-values
    x = np.linspace(-2, 1, nx)
    y = np.linspace(-1.5, 1.5, ny)

    c = x[:,newaxis] + 1j*y[newaxis,:]

    # Mandelbrot iteration

    z = c
    for j in range(N_max):
        z = z**2 + c

    mandelbrot_set = (abs(z) < some_threshold)

    return mandelbrot_set

mandelbrot_set = compute_mandelbrot(50, 50., 601, 401)

plt.imshow(mandelbrot_set.T, extent=[-2, 1, -1.5, 1.5])
plt.gray()
plt.show()
```

Total running time of the script: (0 minutes 0.110 seconds)

Random walk exercise

Plot distance as a function of time for a random walk together with the theoretical result




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

# We create 1000 realizations with 200 steps each
n_stories = 1000
t_max = 200

t = np.arange(t_max)
# Steps can be -1 or 1 (note that randint excludes the upper limit)
steps = 2 * np.random.randint(0, 1 + 1, (n_stories, t_max)) - 1

# The time evolution of the position is obtained by successively
# summing up individual steps. This is done for each of the
# realizations, i.e. along axis 1.
positions = np.cumsum(steps, axis=1)

# Determine the time evolution of the mean square distance.
sq_distance = positions**2
mean_sq_distance = np.mean(sq_distance, axis=0)

# Plot the distance d from the origin as a function of time and
# compare with the theoretically expected result where d(t)
# grows as a square root of time t.
plt.figure(figsize=(4, 3))
plt.plot(t, np.sqrt(mean_sq_distance), 'g.', t, np.sqrt(t), 'y-')
plt.xlabel(r"$t$")
plt.ylabel(r"$\sqrt{\langle (\delta x)^2 \rangle}$")
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.132 seconds)

CHAPTER 2

Matplotlib: plotting

Thanks

Many thanks to **Bill Wing** and **Christoph Deil** for review and corrections.

Authors: *Nicolas Rougier, Mike Müller, Gaël Varoquaux*

Chapter contents

2.1 Introduction

Tip: Matplotlib is probably the most used Python package for 2D-graphics. It provides both a quick way to visualize data from Python and publication-quality figures in many formats. We are going to explore matplotlib

in interactive mode covering most common cases.

2.1.1 IPython, Jupyter, and matplotlib modes

Tip: The Jupyter notebook and the IPython enhanced interactive Python, are tuned for the scientific-computing workflow in Python, in combination with Matplotlib:

For interactive matplotlib sessions, turn on the **matplotlib mode**

IPython console When using the IPython console, use:

```
In [1]: %matplotlib
```

Jupyter notebook In the notebook, insert, **at the beginning of the notebook** the following magic:

```
%matplotlib inline
```

2.1.2 pyplot

Tip: *pyplot* provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab™. Therefore, the majority of plotting commands in pyplot have Matlab™ analogs with similar arguments. Important commands are explained with interactive examples.

```
from matplotlib import pyplot as plt
```

2.2 Simple plot

Tip: In this section, we want to draw the cosine and sine functions on the same plot. Starting from the default settings, we'll enrich the figure step by step to make it nicer.

First step is to get the data for the sine and cosine functions:

```
import numpy as np

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)
```

`X` is now a numpy array with 256 values ranging from $-\pi$ to $+\pi$ (included). `C` is the cosine (256 values) and `S` is the sine (256 values).

To run the example, you can type them in an IPython interactive session:

```
$ ipython --pylab
```

This brings us to the IPython prompt:

```
IPython 0.13 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.
```

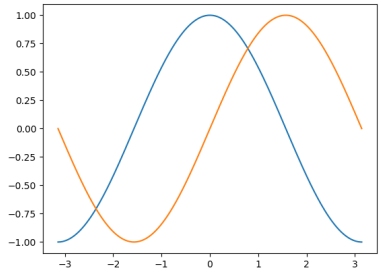
```
Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.
```

Tip: You can also download each of the examples and run it using regular python, but you will lose interactive data manipulation:

```
$ python plot_exercise_1.py
```

You can get source for each step by clicking on the corresponding figure.

2.2.1 Plotting with default settings



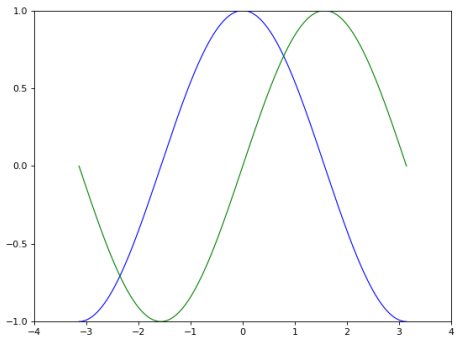
Hint: Documentation

- plot tutorial
- plot() command

Tip: Matplotlib comes with a set of default settings that allow customizing all kinds of properties. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on.

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

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C)
plt.plot(X, S)

plt.show()
```

2.2.2 Instantiating defaults



Hint: Documentation

- Customizing matplotlib

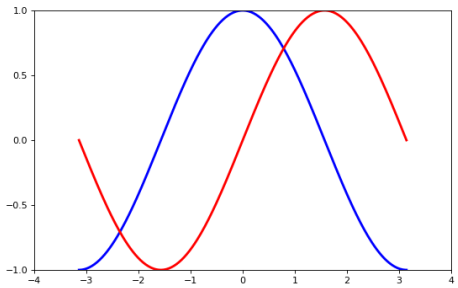
In the script below, we've instantiated (and commented) all the figure settings that influence the appearance of the plot.

Tip: The settings have been explicitly set to their default values, but now you can interactively play with the values to explore their affect (see *Line properties* and *Line styles* below).

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

# Create a figure of size 8x6 inches, 80 dots per inch
plt.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
plt.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

# Plot cosine with a blue continuous line of width 1 (pixels)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine with a green continuous line of width 1 (pixels)
plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
plt.xlim(-4.0, 4.0)

# Set x ticks
plt.xticks(np.linspace(-4, 4, 9, endpoint=True))

# Set y limits
plt.ylim(-1.0, 1.0)

# Set y ticks
plt.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Save figure using 72 dots per inch
# plt.savefig("exercise_2.png", dpi=72)
```

```
# Show result on screen
plt.show()
```

2.2.3 Changing colors and line widths



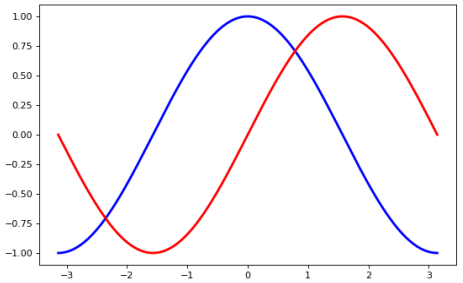
Hint: Documentation

- Controlling line properties
- Line API

Tip: First step, we want to have the cosine in blue and the sine in red and a slighty thicker line for both of them. We'll also slightly alter the figure size to make it more horizontal.

```
...
plt.figure(figsize=(10, 6), dpi=80)
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
...
```

2.2.4 Setting limits



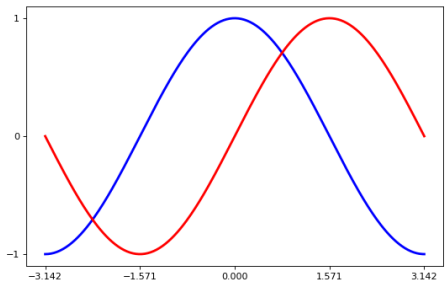
Hint: Documentation

- xlim() command
- ylim() command

Tip: Current limits of the figure are a bit too tight and we want to make some space in order to clearly see all data points.

```
...
plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)
...
```

2.2.5 Setting ticks



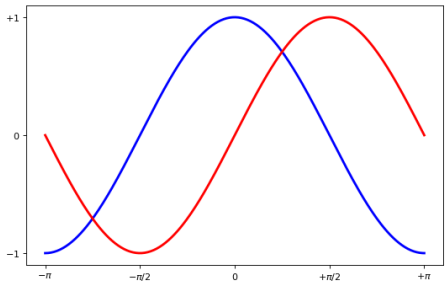
Hint: Documentation

- xticks() command
- yticks() command
- Tick container
- Tick locating and formatting

Tip: Current ticks are not ideal because they do not show the interesting values (+/-π,+/-π/2) for sine and cosine. We'll change them such that they show only these values.

```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
plt.yticks([-1, 0, +1])
...
```

2.2.6 Setting tick labels



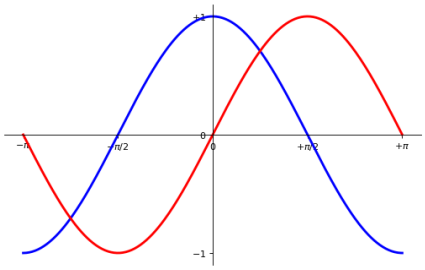
Hint: Documentation

- Working with text
- xticks() command
- yticks() command
- set_xticklabels()
- set_yticklabels()

Tip: Ticks are now properly placed but their label is not very explicit. We could guess that 3.142 is π but it would be better to make it explicit. When we set tick values, we can also provide a corresponding label in the second argument list. Note that we'll use latex to allow for nice rendering of the label.

```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
           [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.yticks([-1, 0, +1],
           [r'$-1$', r'$0$', r'$+1$'])
...
```

2.2.7 Moving spines



Hint: Documentation

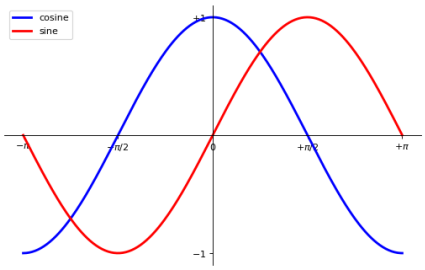
- Spines
- Axis container
- Transformations tutorial

Tip: Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions and until now, they were on the border of the axis. We'll change that since we want to have them in the middle. Since there are four of them (top/bottom/left/right), we'll discard the top and right by setting their color to none and we'll move the bottom and left ones to coordinate 0 in data space coordinates.

```
...
ax = plt.gca()  # gca stands for 'get current axis'
ax.spines['right'].set_color('none')
```

```
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
...
```

2.2.8 Adding a legend



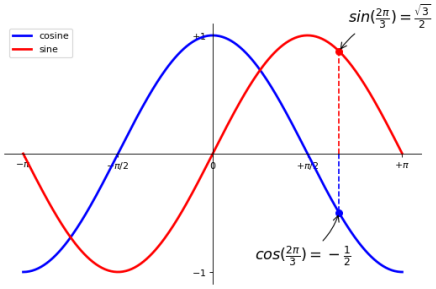
Hint: Documentation

- Legend guide
- legend() command
- Legend API

Tip: Let's add a legend in the upper left corner. This only requires adding the keyword argument label (that will be used in the legend box) to the plot commands.

```
...
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-", label="sine")

plt.legend(loc='upper left')
...
```

2.2.9 Annotate some points



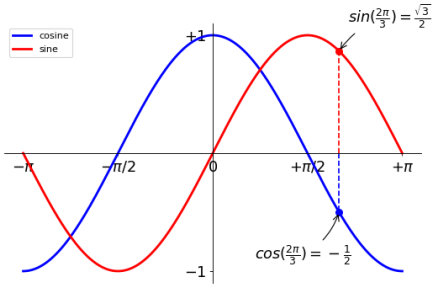
Hint: Documentation

- Annotating axis
- annotate() command

Tip: Let's annotate some interesting points using the annotate command. We chose the $2\pi/3$ value and we want to annotate both the sine and the cosine. We'll first draw a marker on the curve as well as a straight dotted line. Then, we'll use the annotate command to display some text with an arrow.

```
...

t = 2 * np.pi / 3
plt.plot([t, t], [0, np.cos(t)], color='blue', linewidth=2.5, linestyle="--")
plt.scatter([t, ], [np.cos(t), ], 50, color='blue')

plt.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$',
             xy=(t, np.cos(t)), xycoords='data',
             xytext=(-90, -50), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plt.plot([t, t],[0, np.sin(t)], color='red', linewidth=2.5, linestyle="--")
plt.scatter([t, ],[np.sin(t), ], 50, color='red')

plt.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
             xy=(t, np.sin(t)), xycoords='data',
             xytext=(+10, +30), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
...
```

2.2.10 Devil is in the details



Hint: Documentation

- Artists
- BBox

Tip: The tick labels are now hardly visible because of the blue and red lines. We can make them bigger and we can also adjust their properties such that they'll be rendered on a semi-transparent white background. This will allow us to see both the data and the labels.

```
...
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
```

```
    label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65))
...
```

2.3 Figures, Subplots, Axes and Ticks

A **"figure"** in matplotlib means the whole window in the user interface. Within this figure there can be **"subplots"**.

Tip: So far we have used implicit figure and axes creation. This is handy for fast plots. We can have more control over the display using figure, subplot, and axes explicitly. While subplot positions the plots in a regular grid, axes allows free placement within the figure. Both can be useful depending on your intention. We've already worked with figures and subplots without explicitly calling them. When we call plot, matplotlib calls `gca()` to get the current axes and gca in turn calls `gcf()` to get the current figure. If there is none it calls `figure()` to make one, strictly speaking, to make a `subplot(111)`. Let's look at the details.

2.3.1 Figures

Tip: A figure is the windows in the GUI that has "Figure #" as title. Figures are numbered starting from 1 as opposed to the normal Python way starting from 0. This is clearly MATLAB-style. There are several parameters that determine what the figure looks like:

Argument	Default	Description
`num`	`1`	number of figure
`figsize`	`figure.figsize`	figure size in inches (width, height)
`dpi`	`figure.dpi`	resolution in dots per inch
`facecolor`	`figure.facecolor`	color of the drawing background
`edgecolor`	`figure.edgecolor`	color of edge around the drawing background
`frameon`	`True`	draw figure frame or not

Tip: The defaults can be specified in the resource file and will be used most of the time. Only the number of the figure is frequently changed.

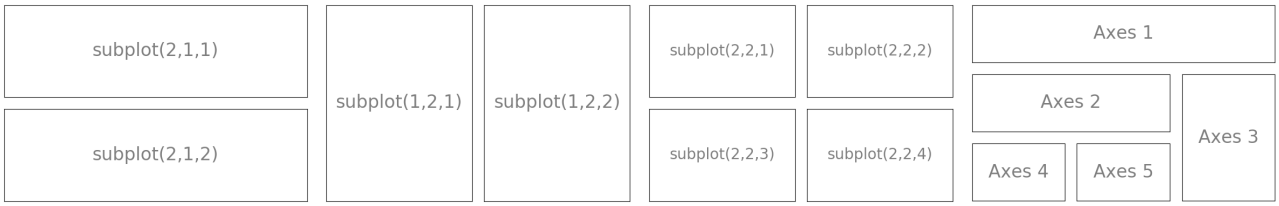
As with other objects, you can set figure properties also setp or with the set_something methods.

When you work with the GUI you can close a figure by clicking on the x in the upper right corner. But you can close a figure programmatically by calling close. Depending on the argument it closes (1) the current figure (no argument), (2) a specific figure (figure number or figure instance as argument), or (3) all figures (`"all"` as argument).

```
plt.close(1)     # Closes figure 1
```

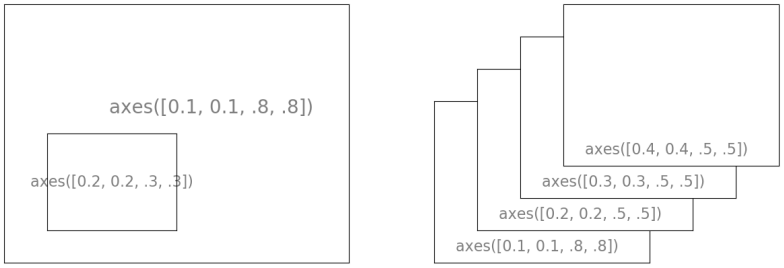
2.3.2 Subplots

Tip: With subplot you can arrange plots in a regular grid. You need to specify the number of rows and columns and the number of the plot. Note that the gridspec command is a more powerful alternative.



2.3.3 Axes

Axes are very similar to subplots but allow placement of plots at any location in the figure. So if we want to put a smaller plot inside a bigger one we do so with axes.



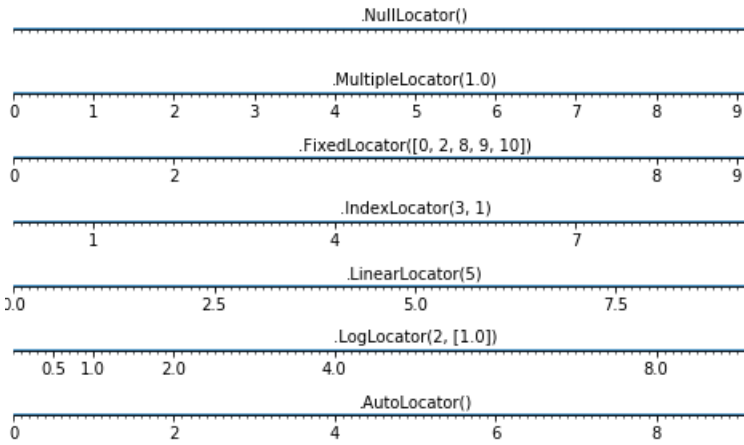
2.3.4 Ticks

Well formatted ticks are an important part of publishing-ready figures. Matplotlib provides a totally configurable system for ticks. There are tick locators to specify where ticks should appear and tick formatters to give ticks the appearance you want. Major and minor ticks can be located and formatted independently from each other. Per default minor ticks are not shown, i.e. there is only an empty list for them because it is as `NullLocator` (see below).

Tick Locators

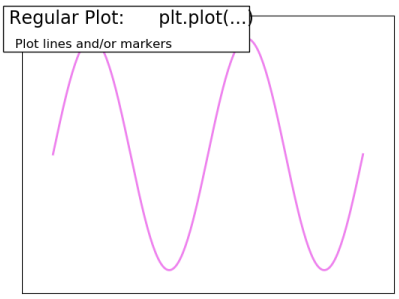
Tick locators control the positions of the ticks. They are set as follows:

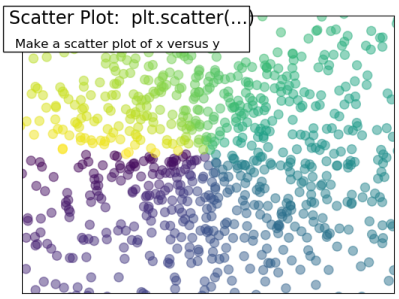
```
ax = plt.gca()
ax.xaxis.set_major_locator(eval(locator))
```

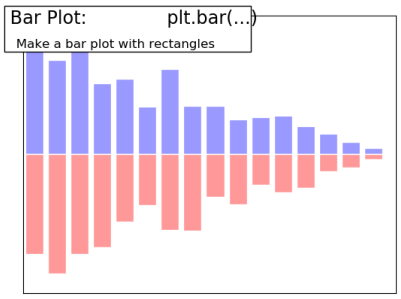


There are several locators for different kind of requirements:
All of these locators derive from the base class `matplotlib.ticker.Locator`. You can make your own locator deriving from it. Handling dates as ticks can be especially tricky. Therefore, matplotlib provides special locators in matplotlib.dates.

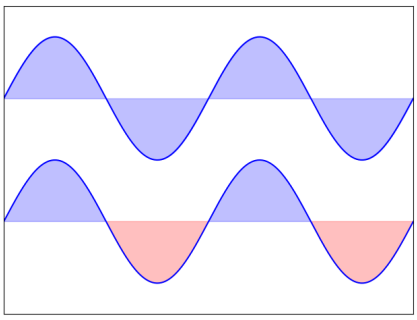
2.4 Other Types of Plots: examples and exercises







2.4.1 Regular Plots



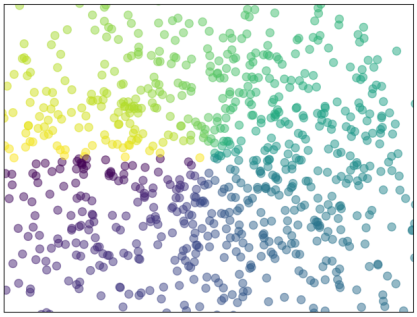
Starting from the code below, try to reproduce the graphic taking care of filled areas:

Hint: You need to use the fill_between command.

```
n = 256
X = np.linspace(-np.pi, np.pi, n, endpoint=True)
Y = np.sin(2 * X)

plt.plot(X, Y + 1, color='blue', alpha=1.00)
plt.plot(X, Y - 1, color='blue', alpha=1.00)
```

Click on the figure for solution.

2.4.2 Scatter Plots



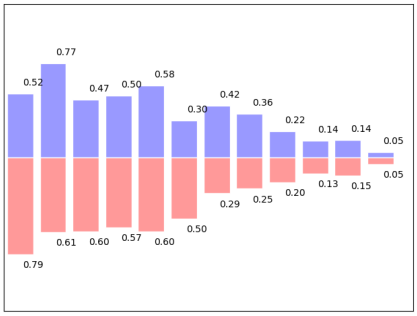
Starting from the code below, try to reproduce the graphic taking care of marker size, color and transparency.

Hint: Color is given by angle of (X,Y).

```
n = 1024
X = np.random.normal(0,1,n)
Y = np.random.normal(0,1,n)

plt.scatter(X,Y)
```

Click on figure for solution.

2.4.3 Bar Plots



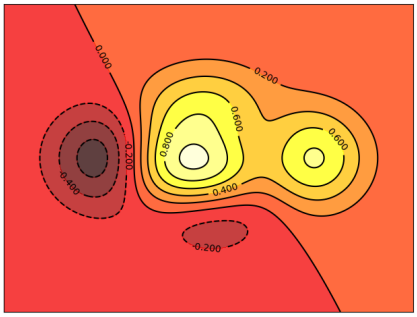
Starting from the code below, try to reproduce the graphic by adding labels for red bars.

Hint: You need to take care of text alignment.

```
n = 12
X = np.arange(n)
Y1 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
Y2 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)

plt.bar(X, +Y1, facecolor='#9999ff', edgecolor='white')
plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x, y in zip(X, Y1):
    plt.text(x + 0.4, y + 0.05, '%.2f' % y, ha='center', va='bottom')

plt.ylim(-1.25, +1.25)
```

Click on figure for solution.

2.4.4 Contour Plots



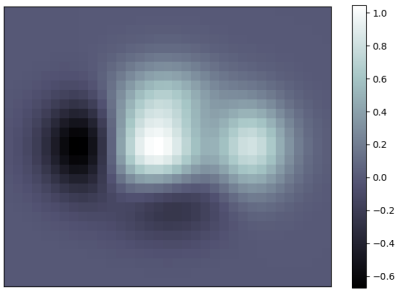
Starting from the code below, try to reproduce the graphic taking care of the colormap (see *Colormaps* below).

Hint: You need to use the clabel command.

```
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 -y ** 2)

n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)

plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='jet')
C = plt.contour(X, Y, f(X, Y), 8, colors='black', linewidth=.5)
```

Click on figure for solution.

2.4.5 Imshow



Starting from the code below, try to reproduce the graphic taking care of colormap, image interpolation and origin.

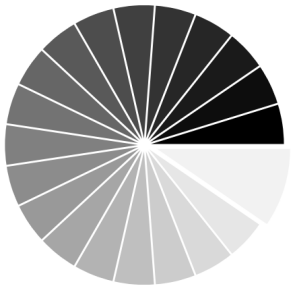
Hint: You need to take care of the `origin` of the image in the imshow command and use a colorbar

---

```
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)

n = 10
x = np.linspace(-3, 3, 4 * n)
y = np.linspace(-3, 3, 3 * n)
X, Y = np.meshgrid(x, y)
plt.imshow(f(X, Y))
```

Click on the figure for the solution.

2.4.6 Pie Charts



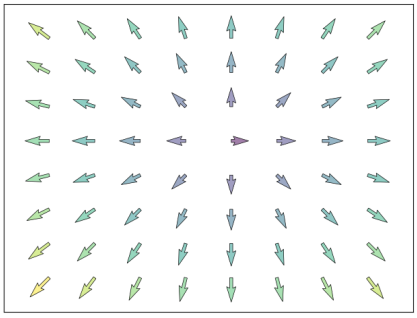
Starting from the code below, try to reproduce the graphic taking care of colors and slices size.

Hint: You need to modify Z.

```
Z = np.random.uniform(0, 1, 20)
plt.pie(Z)
```

Click on the figure for the solution.

2.4.7 Quiver Plots



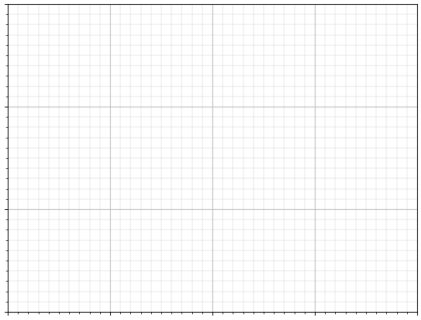
Starting from the code below, try to reproduce the graphic taking care of colors and orientations.

Hint: You need to draw arrows twice.

```
n = 8
X, Y = np.mgrid[0:n, 0:n]
plt.quiver(X, Y)
```

Click on figure for solution.

2.4.8 Grids



Starting from the code below, try to reproduce the graphic taking care of line styles.

```
axes = plt.gca()
axes.set_xlim(0, 4)
axes.set_ylim(0, 3)
axes.set_xticklabels([])
axes.set_yticklabels([])
```

Click on figure for solution.

2.4.9 Multi Plots

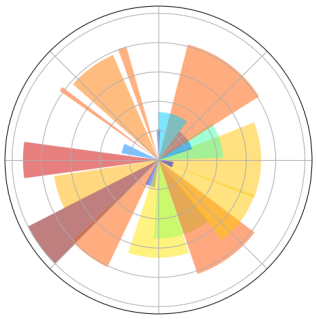
Starting from the code below, try to reproduce the graphic.

Hint: You can use several subplots with different partition.

```
plt.subplot(2, 2, 1)
plt.subplot(2, 2, 3)
plt.subplot(2, 2, 4)
```

Click on figure for solution.

2.4.10 Polar Axis



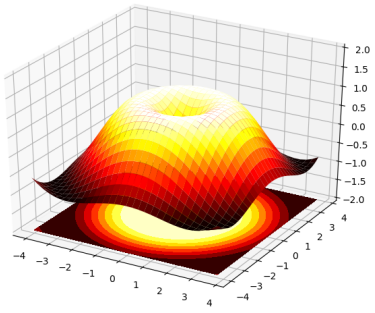
Hint: You only need to modify the `axes` line

Starting from the code below, try to reproduce the graphic.

```
plt.axes([0, 0, 1, 1])

N = 20
theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r / 10.))
    bar.set_alpha(0.5)
```

Click on figure for solution.

2.4.11 3D Plots



Starting from the code below, try to reproduce the graphic.

Hint: You need to use contourf

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
Y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
```

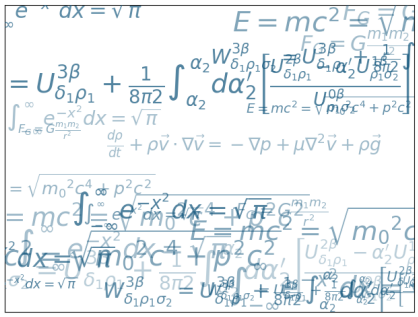
```
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

Click on figure for solution.

See also:

3D plotting with Mayavi

2.4.12 Text



Try to do the same from scratch !

Hint: Have a look at the matplotlib logo.

Click on figure for solution.

Quick read

If you want to do a first quick pass through the Scipy lectures to learn the ecosystem, you can directly skip to the next chapter: *Scipy : high-level scientific computing*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter later.

2.5 Beyond this tutorial

Matplotlib benefits from extensive documentation as well as a large community of users and developers. Here are some links of interest:

2.5.1 Tutorials

- Pyplot tutorial
- Introduction

- Controlling line properties
- Working with multiple figures and axes
- Working with text
- Image tutorial
- Startup commands
- Importing image data into Numpy arrays
- Plotting numpy arrays as images
- Text tutorial
- Text introduction
- Basic text commands
- Text properties and layout
- Writing mathematical expressions
- Text rendering With LaTeX
- Annotating text
- Artist tutorial
- Introduction
- Customizing your objects
- Object containers
- Figure container
- Axes container
- Axis containers
- Tick containers
- Path tutorial
- Introduction
- Bézier example
- Compound paths
- Transforms tutorial
- Introduction
- Data coordinates
- Axes coordinates
- Blended transformations
- Using offset transforms to create a shadow effect
- The transformation pipeline

2.5.2 Matplotlib documentation

- User guide
- FAQ
- Installation
- Usage
- How-To
- Troubleshooting
- Environment Variables
- Screenshots

2.5.3 Code documentation

The code is well documented and you can quickly access a specific command from within a python session:

```
>>> import matplotlib.pyplot as plt
>>> help(plt.plot)
Help on function plot in module matplotlib.pyplot:

plot(*args, **kwargs)
   Plot lines and/or markers to the
   :class:`~matplotlib.axes.Axes`.  *args* is a variable length
   argument, allowing for multiple *x*, *y* pairs with an
```

```
   optional format string.  For example, each of the following is
   legal::

       plot(x, y)         # plot x and y using default line style and color
       plot(x, y, 'bo')   # plot x and y using blue circle markers
       plot(y)            # plot y using x as index array 0..N-1
       plot(y, 'r+')      # ditto, but with red plusses

   If *x* and/or *y* is 2-dimensional, then the corresponding columns
   will be plotted.
...
```

2.5.4 Galleries

The matplotlib gallery is also incredibly useful when you search how to render a given graphic. Each example comes with its source.

2.5.5 Mailing lists

Finally, there is a user mailing list where you can ask for help and a developers mailing list that is more technical.

2.6 Quick references

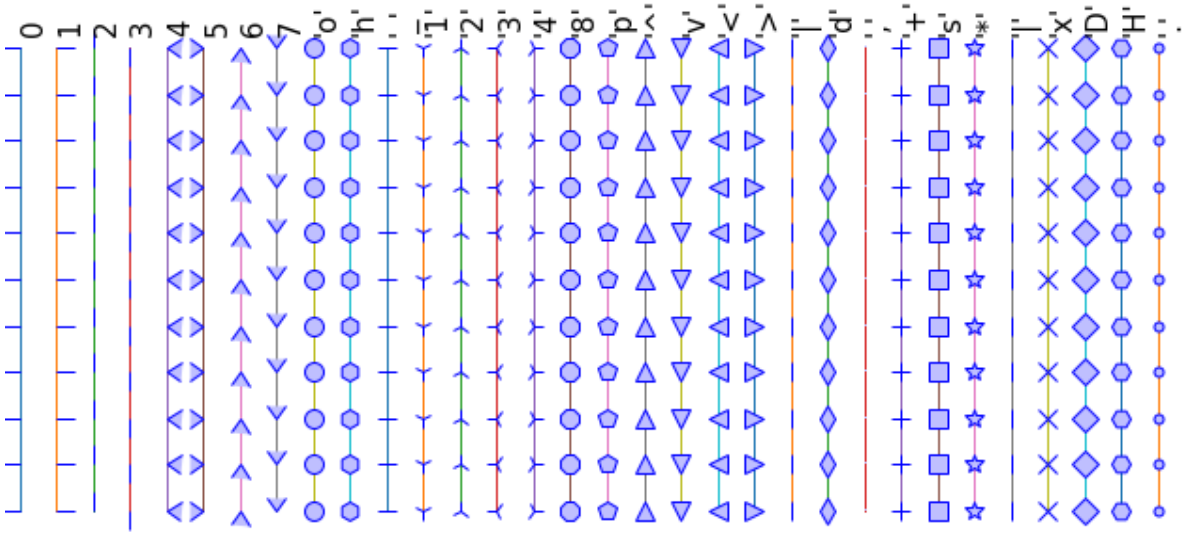
Here is a set of tables that show main properties and styles.

2.6.1 Line properties

Property	Description	Appearance
alpha (or a)	alpha transparency on 0-1 scale	
antialiased	True or False - use antialised rendering	Aliased Anti-aliased
color (or c)	matplotlib color arg	
linestyle (or ls)	see *Line properties*	
linewidth (or lw)	float, the line width in points	
solid_capstyle	Cap style for solid lines	
solid_joinstyle	Join style for solid lines	
dash_capstyle	Cap style for dashes	
dash_joinstyle	Join style for dashes	
marker	see *Markers*	
markeredgewidth (mew)	line width around the marker symbol	
markeredgecolor (mec)	edge color if a marker is used	
markerfacecolor (mfc)	face color if a marker is used	
markersize (ms)	size of the marker in points	

2.6.2 Line styles

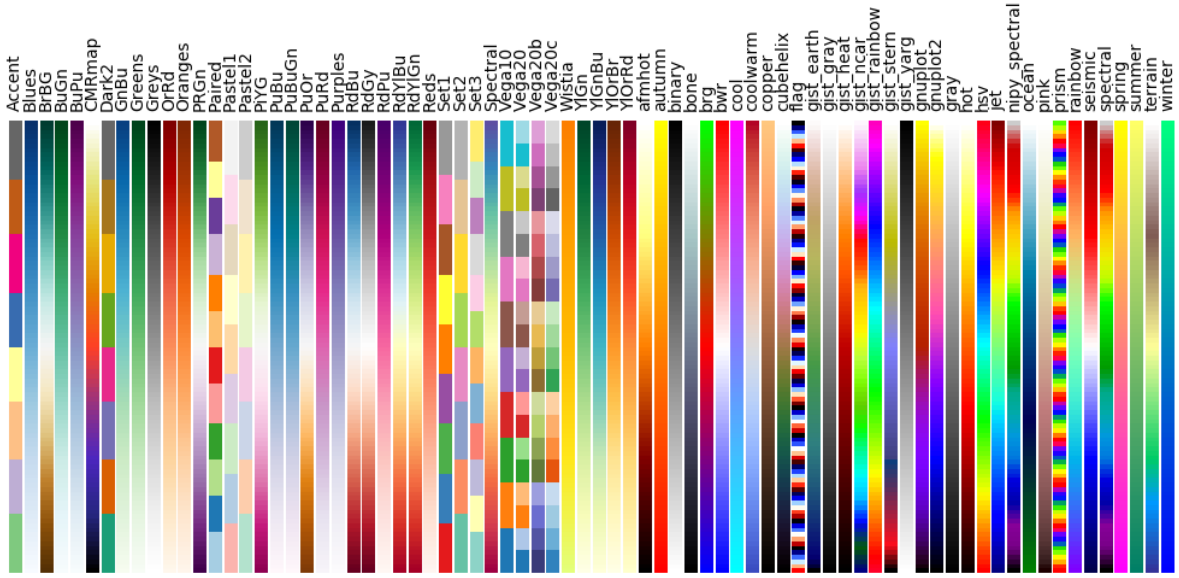
2.6.3 Markers



2.6.4 Colormaps

All colormaps can be reversed by appending `_r`. For instance, `gray_r` is the reverse of `gray`.

If you want to know more about colormaps, check the documentation on Colormaps in matplotlib.



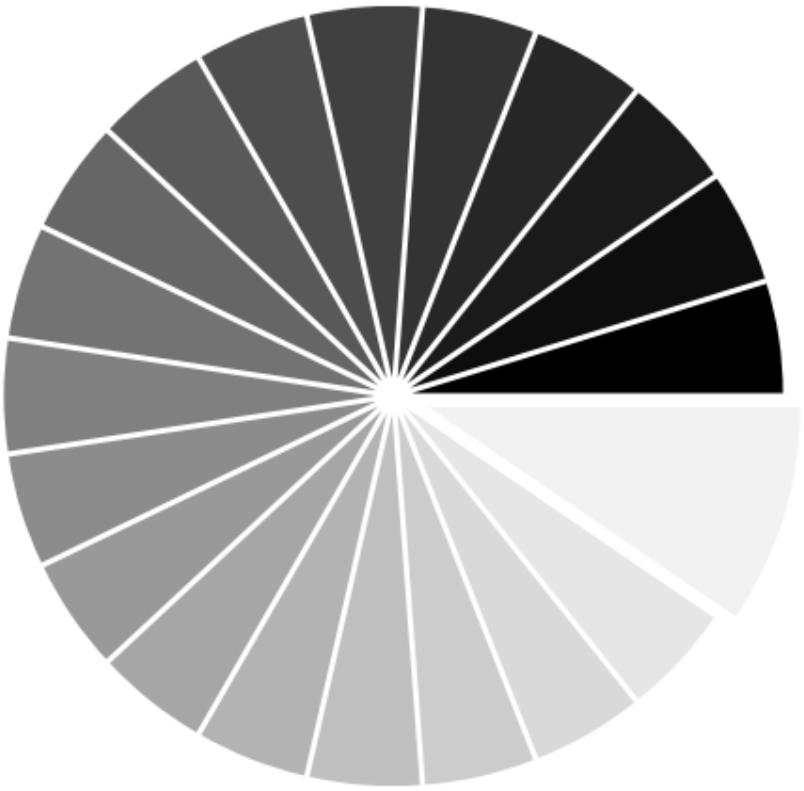
2.7 Full code examples

2.7.1 Code samples for Matplotlib

The examples here are only examples relevant to the points raised in this chapter. The matplotlib documentation comes with a much more exhaustive gallery.

Pie chart

A simple pie chart example with matplotlib.

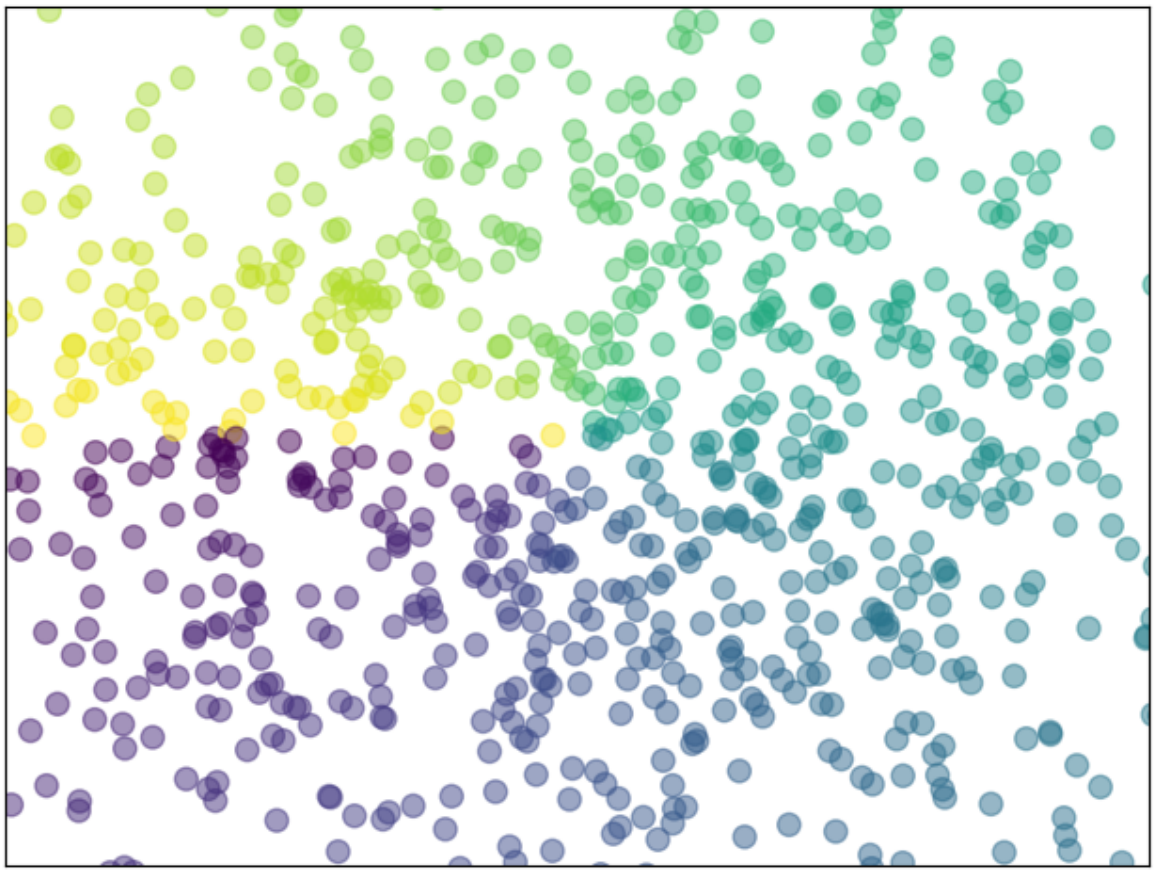


```
import numpy as np
import matplotlib.pyplot as plt

n = 20
Z = np.ones(n)
Z[-1] *= 2

plt.axes([0.025, 0.025, 0.95, 0.95])

plt.pie(Z, explode=Z*.05, colors = ['%f' % (i/float(n)) for i in range(n)])
plt.axis('equal')
plt.xticks(())
plt.yticks()

plt.show()
```

Total running time of the script: (0 minutes 0.071 seconds)

Plotting a scatter of points

A simple example showing how to plot a scatter of points with matplotlib.

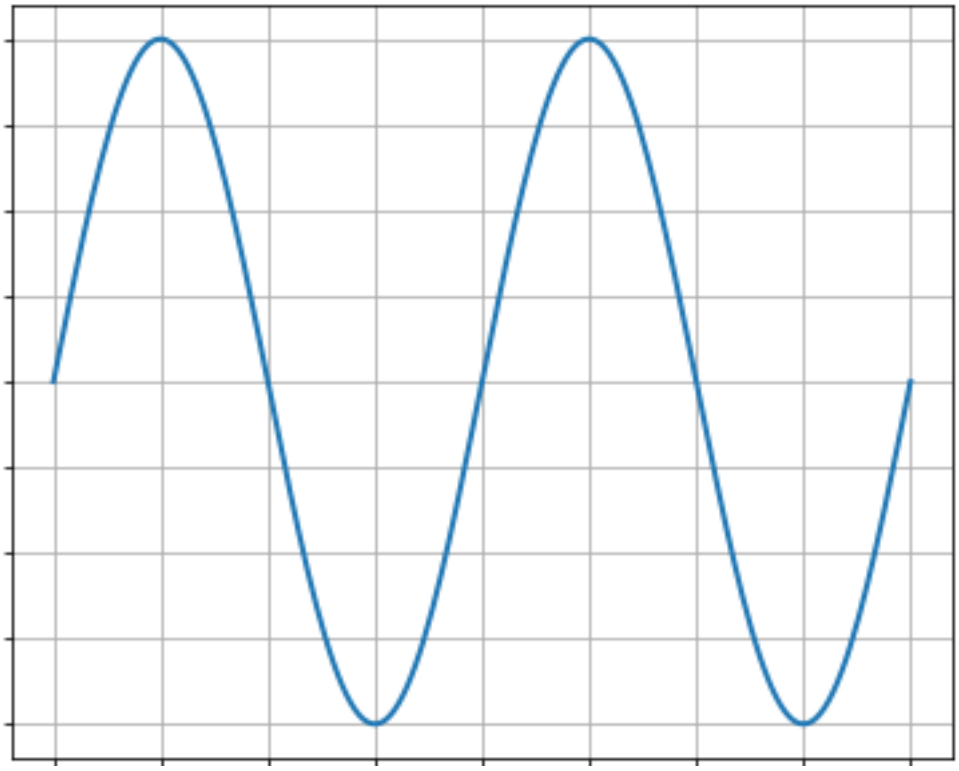


```
import numpy as np
import matplotlib.pyplot as plt

n = 1024
X = np.random.normal(0, 1, n)
Y = np.random.normal(0, 1, n)
T = np.arctan2(Y, X)

plt.axes([0.025, 0.025, 0.95, 0.95])
plt.scatter(X, Y, s=75, c=T, alpha=.5)

plt.xlim(-1.5, 1.5)
plt.xticks(())
plt.ylim(-1.5, 1.5)
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

A simple, good-looking plot

Demoing some simple features of matplotlib

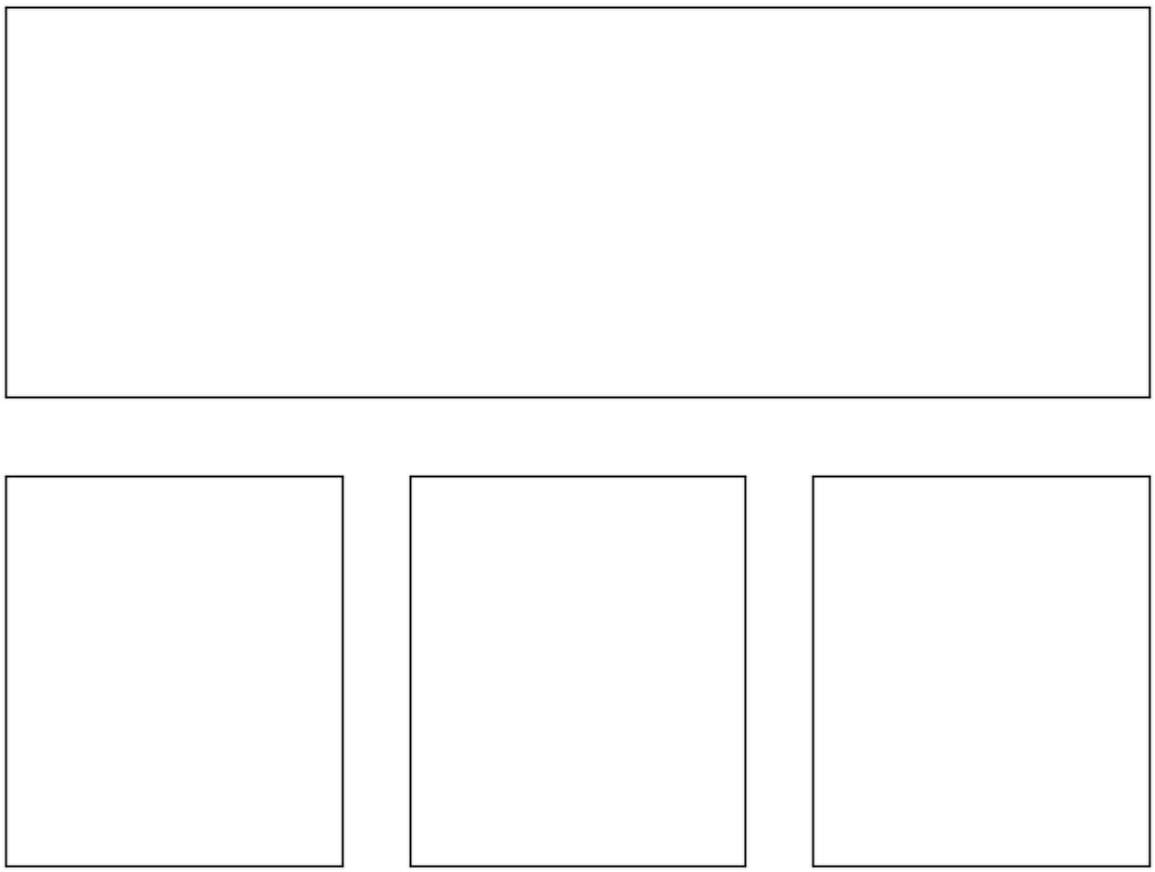


```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5, 4), dpi=72)
axes = fig.add_axes([0.01, 0.01, .98, 0.98])
X = np.linspace(0, 2, 200, endpoint=True)
Y = np.sin(2*np.pi*X)
plt.plot(X, Y, lw=2)
plt.ylim(-1.1, 1.1)
plt.grid()

plt.show()
```

Total running time of the script: (0 minutes 0.050 seconds)

Subplots

Show multiple subplots in matplotlib.

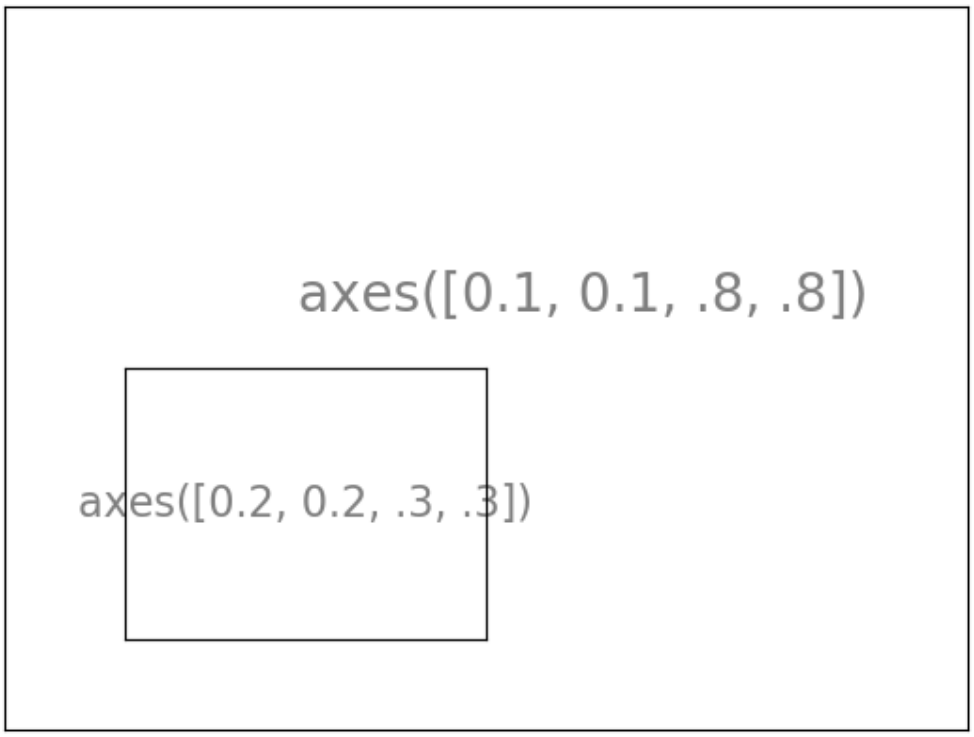


```
import matplotlib.pyplot as plt

fig = plt.figure()
fig.subplots_adjust(bottom=0.025, left=0.025, top = 0.975, right=0.975)

plt.subplot(2, 1, 1)
plt.xticks(()), plt.yticks(())

plt.subplot(2, 3, 4)
plt.xticks(())
plt.yticks(())

plt.subplot(2, 3, 5)
plt.xticks(())
plt.yticks(())

plt.subplot(2, 3, 6)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.191 seconds)

Simple axes example

This example shows a couple of simple usage of axes.

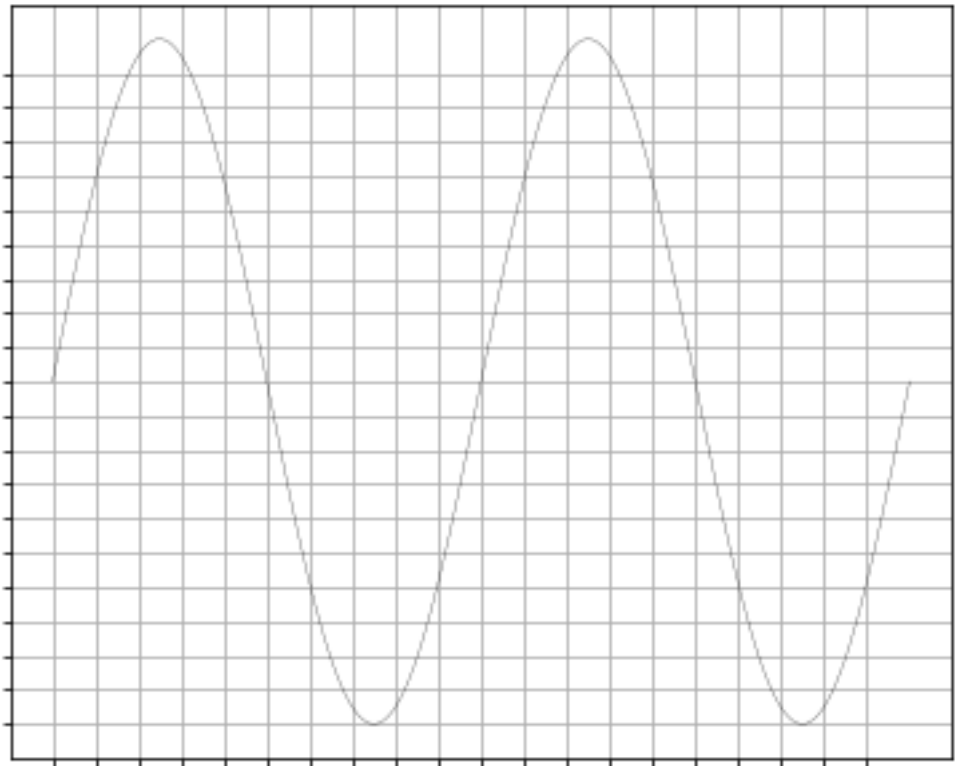



```
import matplotlib.pyplot as plt

plt.axes([.1, .1, .8, .8])
plt.xticks(())
plt.yticks(())
plt.text(.6, .6, 'axes([0.1, 0.1, .8, .8])', ha='center', va='center',
        size=20, alpha=.5)

plt.axes([.2, .2, .3, .3])
plt.xticks(())
plt.yticks(())
plt.text(.5, .5, 'axes([0.2, 0.2, .3, .3])', ha='center', va='center',
        size=16, alpha=.5)

plt.show()
```

Total running time of the script: (0 minutes 0.152 seconds)

A simple plotting example

A plotting example with a few simple tweaks

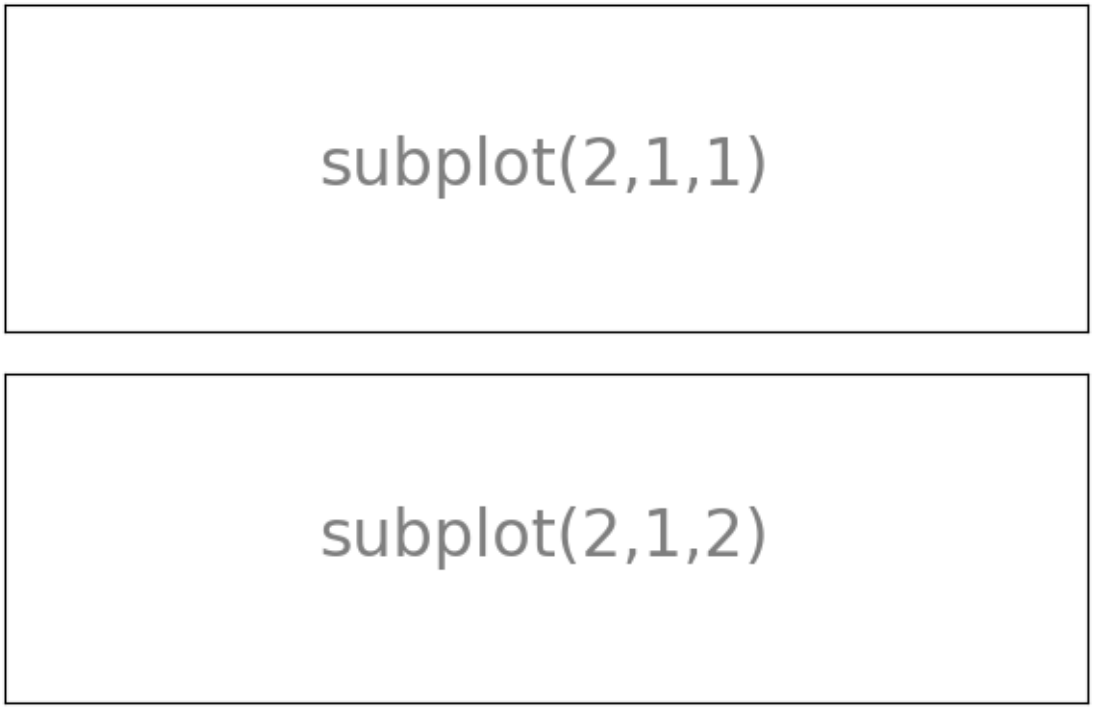


```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5, 4), dpi=72)
axes = fig.add_axes([0.01, 0.01, .98, 0.98])
x = np.linspace(0, 2, 200, endpoint=True)
y = np.sin(2 * np.pi * x)
plt.plot(x, y, lw=.25, c='k')
plt.xticks(np.arange(0.0, 2.0, 0.1))
plt.yticks(np.arange(-1.0, 1.0, 0.1))
plt.grid()
plt.show()
```

Total running time of the script: (0 minutes 0.111 seconds)

Horizontal arrangement of subplots

An example showing horizontal arrangement of subplots with matplotlib.




```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.subplot(2, 1, 1)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(2,1,1)', ha='center', va='center',
        size=24, alpha=.5)

plt.subplot(2, 1, 2)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(2,1,2)', ha='center', va='center',
        size=24, alpha=.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.102 seconds)

Subplot plot arrangement vertical

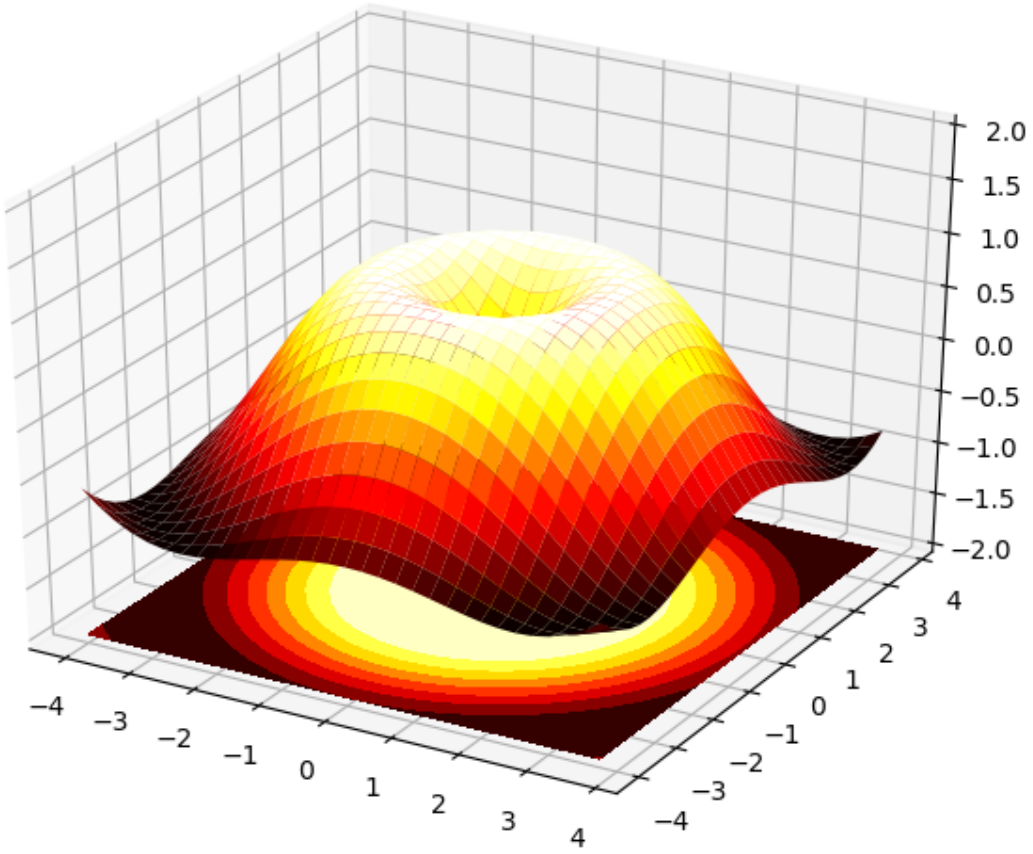
An example showing vertical arrangement of subplots with matplotlib.


```
import matplotlib.pyplot as plt


plt.figure(figsize=(6, 4))
plt.subplot(1, 2, 1)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(1,2,1)', ha='center', va='center',
        size=24, alpha=.5)

plt.subplot(1, 2, 2)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(1,2,2)', ha='center', va='center',
        size=24, alpha=.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.104 seconds)

3D plotting

A simple example of 3D plotting.

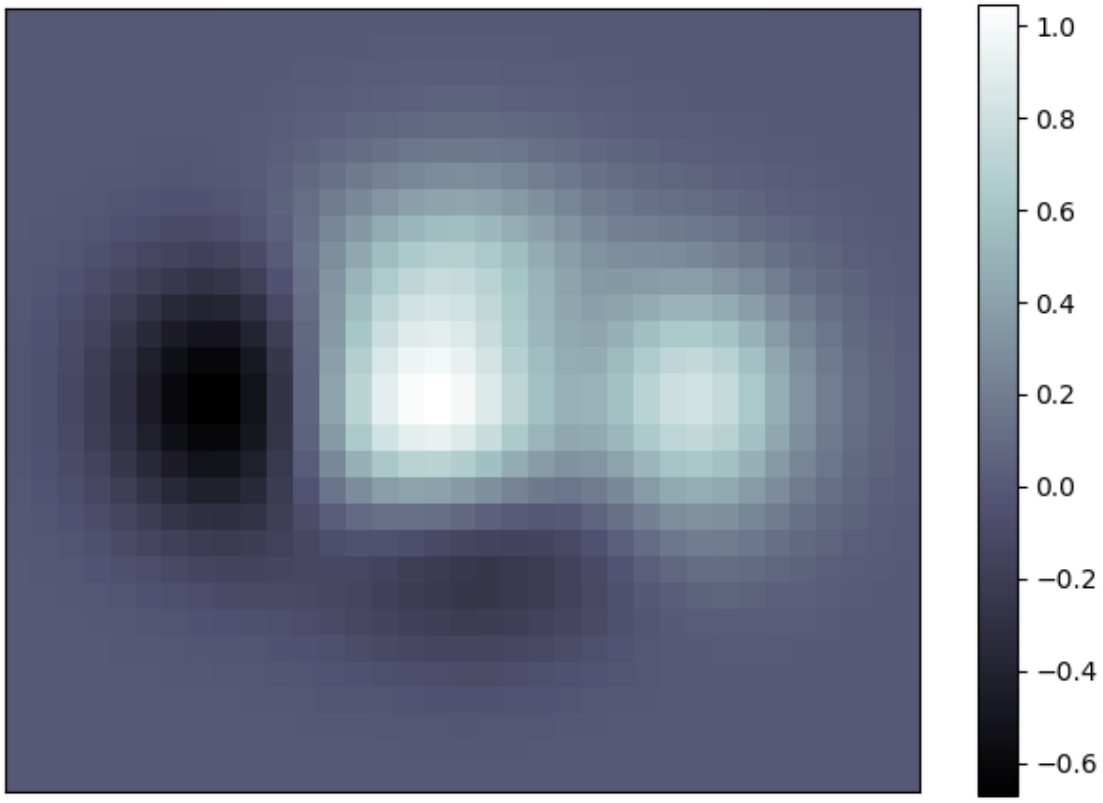



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
Y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X ** 2 + Y ** 2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.cm.hot)
ax.contourf(X, Y, Z, zdir='z', offset=-2, cmap=plt.cm.hot)
ax.set_zlim(-2, 2)

plt.show()
```

Total running time of the script: (0 minutes 0.076 seconds)

Imshow elaborate

An example demoing imshow and styling the figure.

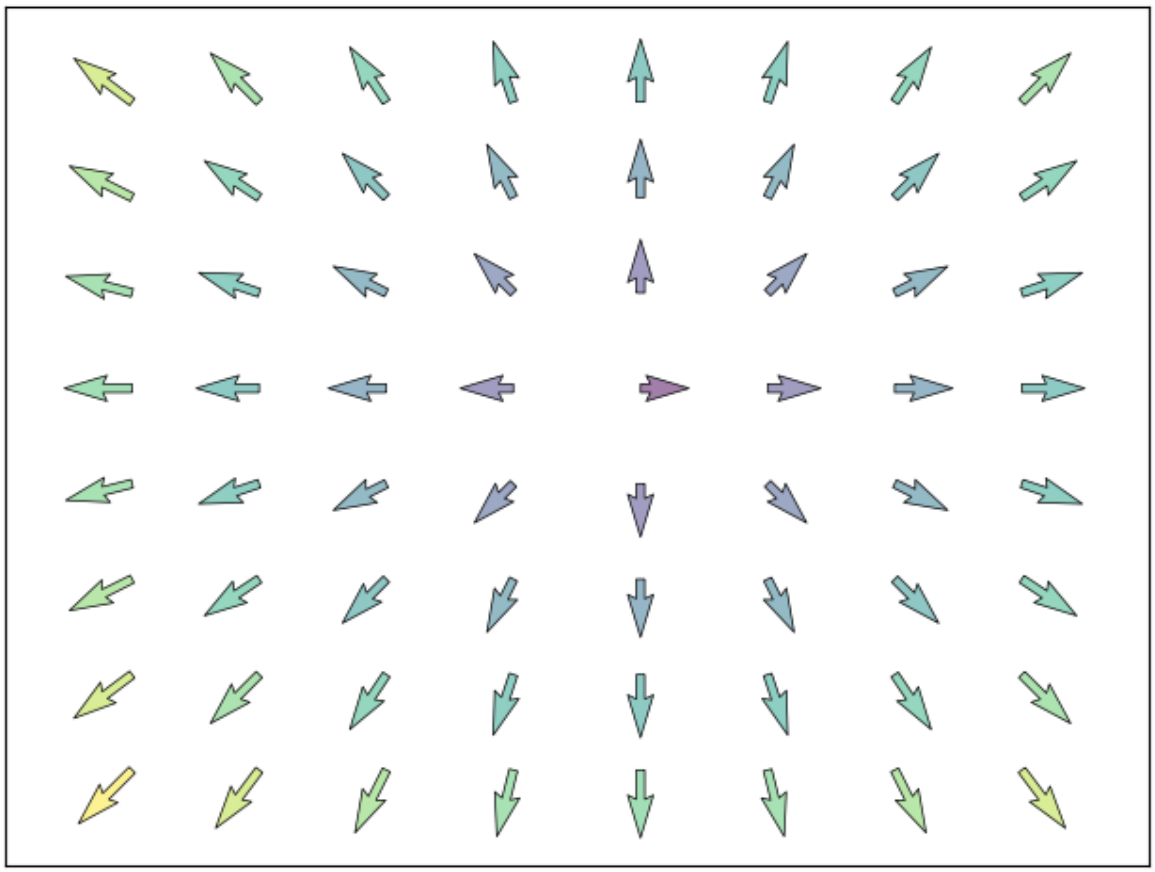



```
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3 ) * np.exp(-x ** 2 - y ** 2)

n = 10
x = np.linspace(-3, 3, 3.5 * n)
y = np.linspace(-3, 3, 3.0 * n)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

plt.axes([0.025, 0.025, 0.95, 0.95])
plt.imshow(Z, interpolation='nearest', cmap='bone', origin='lower')
plt.colorbar(shrink=.92)

plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 0.165 seconds)

Plotting a vector field: quiver

A simple example showing how to plot a vector field (quiver) with matplotlib.

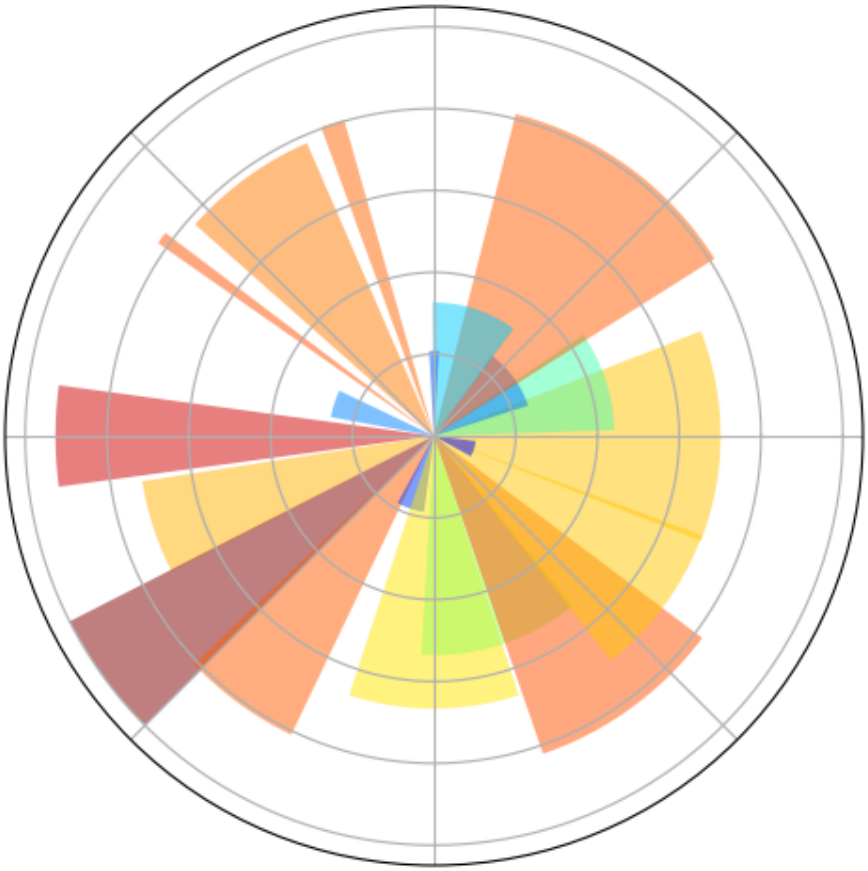


```
import numpy as np
import matplotlib.pyplot as plt

n = 8
X, Y = np.mgrid[0:n, 0:n]
T = np.arctan2(Y - n / 2., X - n/2.)
R = 10 + np.sqrt((Y - n / 2.0) ** 2 + (X - n / 2.0) ** 2)
U, V = R * np.cos(T), R * np.sin(T)

plt.axes([0.025, 0.025, 0.95, 0.95])
plt.quiver(X, Y, U, V, R, alpha=.5)
plt.quiver(X, Y, U, V, edgecolor='k', facecolor='None', linewidth=.5)

plt.xlim(-1, n)
plt.xticks(())
plt.ylim(-1, n)
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

Plotting in polar coordinnates

A simple example showing how to plot in polar coordinnates with matplotlib.

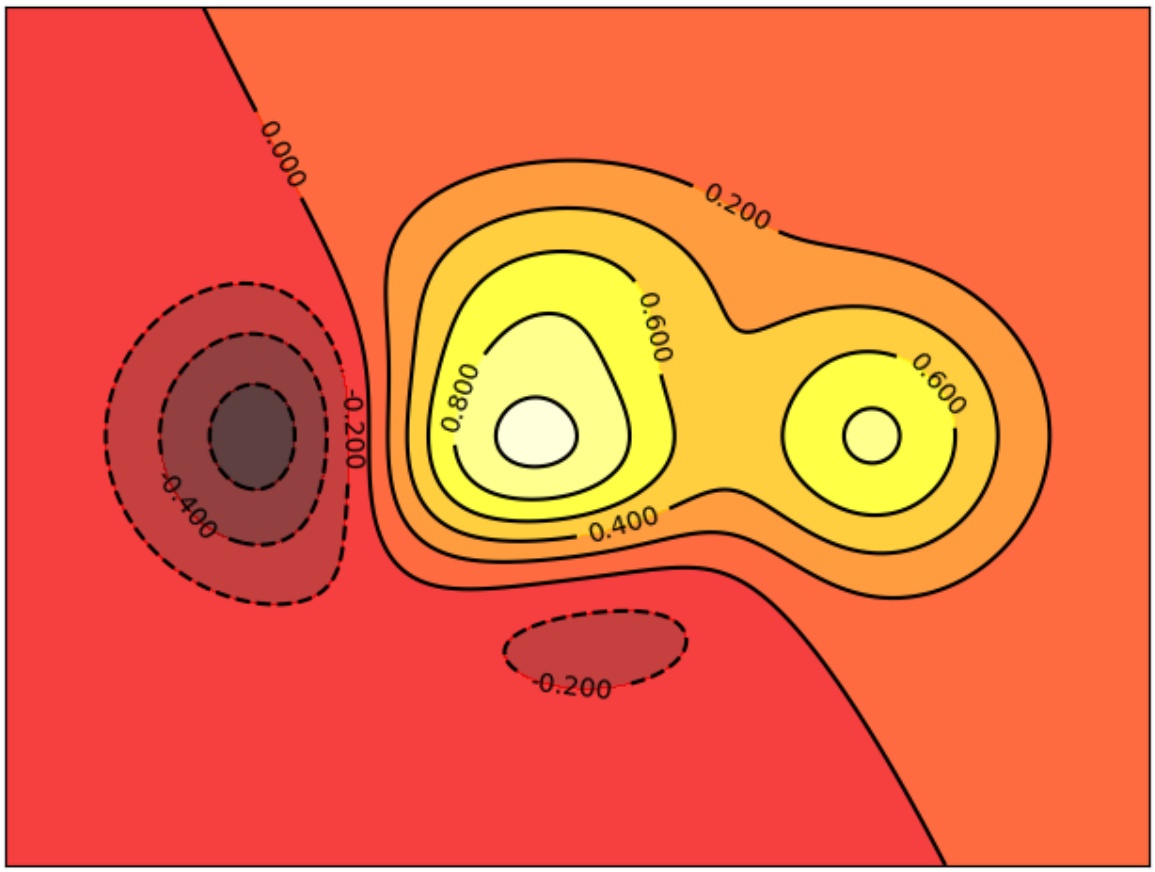


```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)

N = 20
theta = np.arange(0.0, 2 * np.pi, 2 * np.pi / N)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)

for r,bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r/10.))
    bar.set_alpha(0.5)

ax.set_xticklabels([])
ax.set_yticklabels([])
plt.show()
```

Total running time of the script: (0 minutes 0.075 seconds)

Displaying the contours of a function

An example showing how to display the contours of a function with matplotlib.

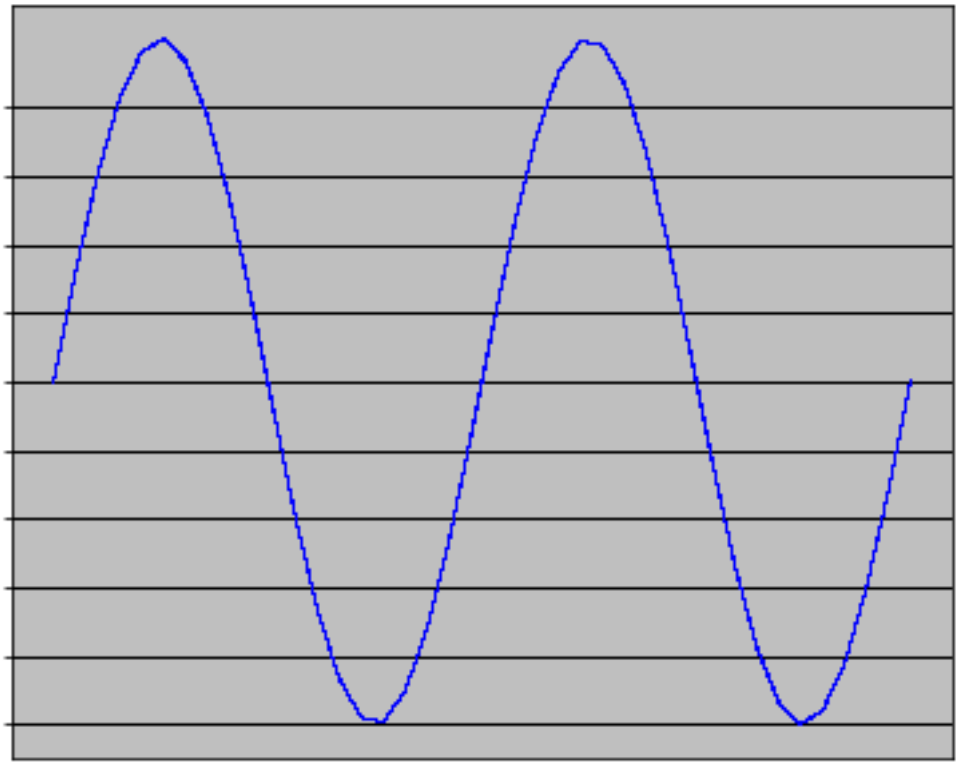



```
import numpy as np
import matplotlib.pyplot as plt

def f(x,y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-x**2 -y**2)

n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X,Y = np.meshgrid(x, y)

plt.axes([0.025, 0.025, 0.95, 0.95])

plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap=plt.cm.hot)
C = plt.contour(X, Y, f(X, Y), 8, colors='black', linewidth=.5)
plt.clabel(C, inline=1, fontsize=10)

plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 0.138 seconds)

A example of plotting not quite right

An “ugly” example of plotting.

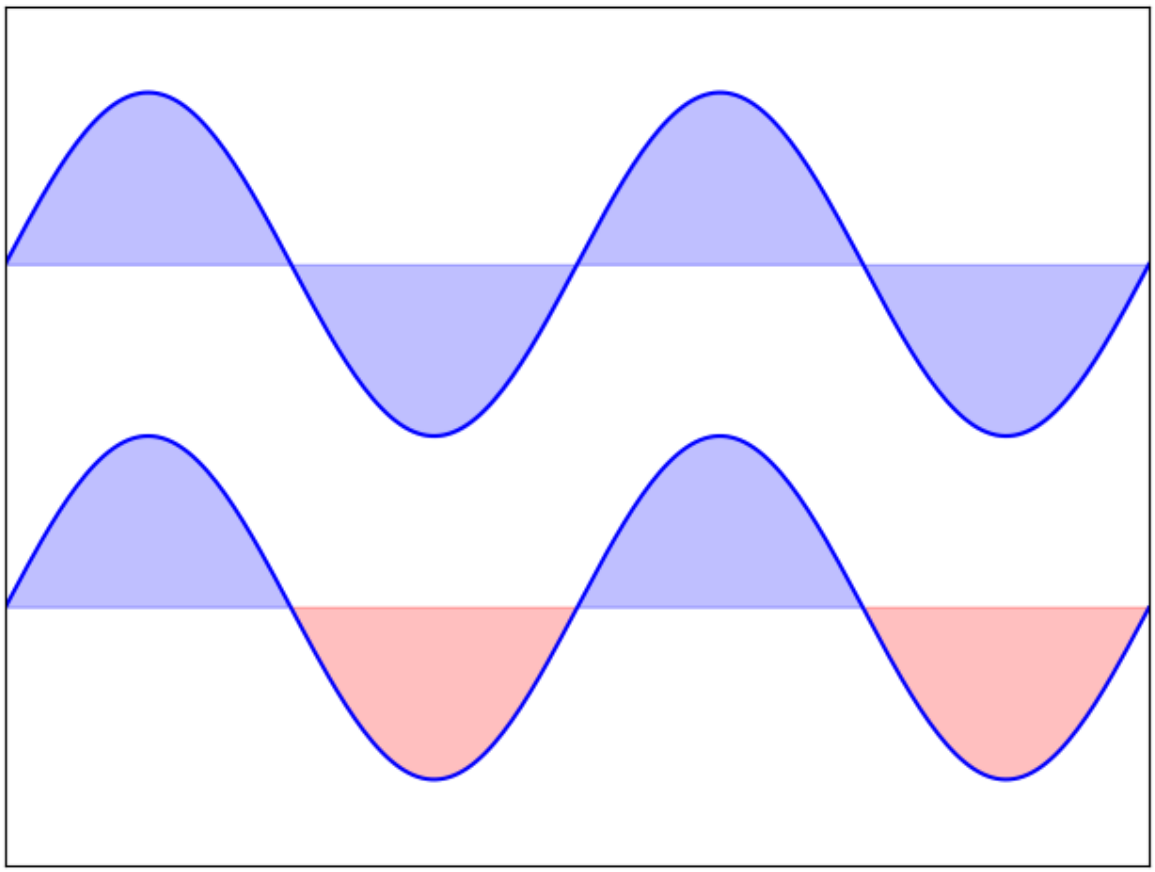


```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

matplotlib.rc('grid', color='black', linestyle='-', linewidth=1)

fig = plt.figure(figsize=(5,4),dpi=72)
axes = fig.add_axes([0.01, 0.01, .98, 0.98], axisbg='.75')
X = np.linspace(0, 2, 40, endpoint=True)
Y = np.sin(2 * np.pi * X)
plt.plot(X, Y, lw=.05, c='b', antialiased=False)

plt.xticks(())
plt.yticks(np.arange(-1., 1., 0.2))
plt.grid()
ax = plt.gca()

plt.show()
```

Total running time of the script: (0 minutes 0.065 seconds)

Plot and filled plots

Simple example of plots and filling between them with matplotlib.

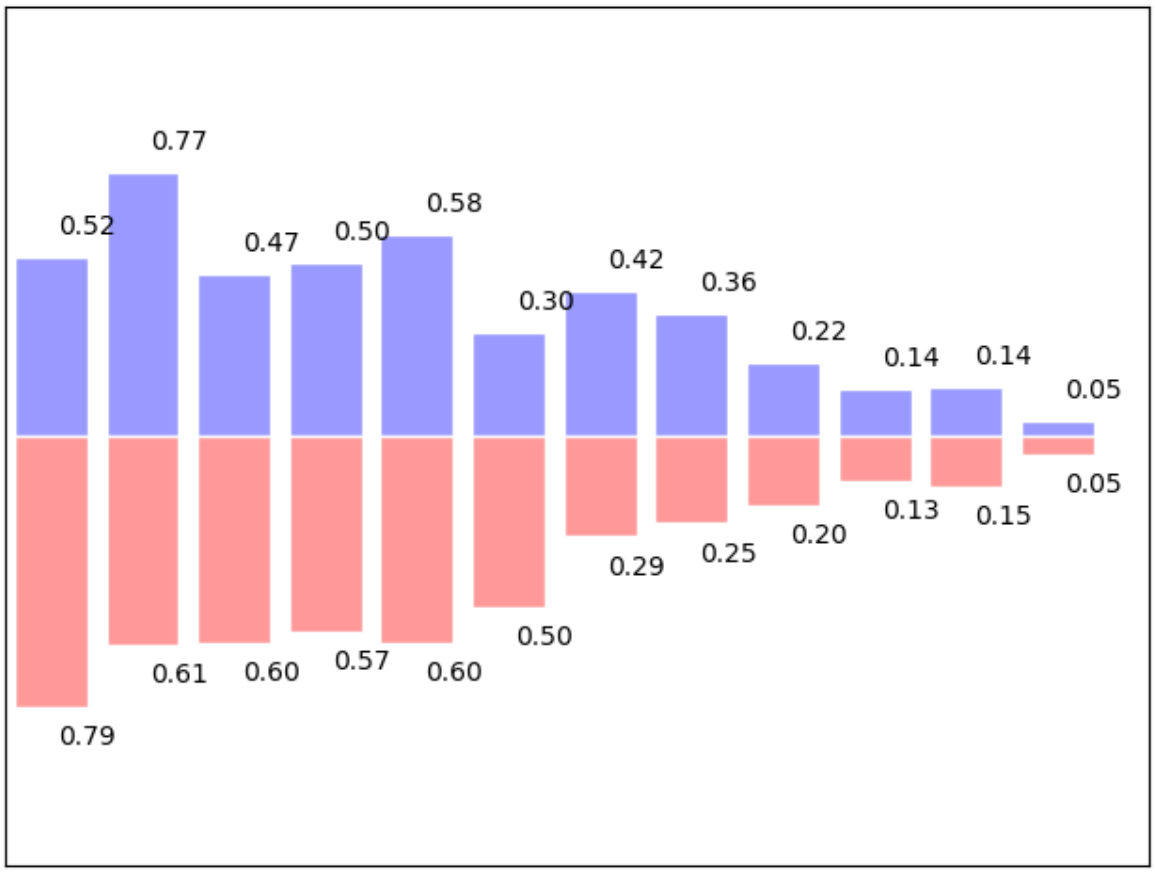


```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi, np.pi, n, endpoint=True)
Y = np.sin(2 * X)

plt.axes([0.025, 0.025, 0.95, 0.95])

plt.plot(X, Y + 1, color='blue', alpha=1.00)
plt.fill_between(X, 1, Y + 1, color='blue', alpha=.25)

plt.plot(X, Y - 1, color='blue', alpha=1.00)
plt.fill_between(X, -1, Y - 1, (Y - 1) > -1, color='blue', alpha=.25)
plt.fill_between(X, -1, Y - 1, (Y - 1) < -1, color='red',  alpha=.25)

plt.xlim(-np.pi, np.pi)
plt.xticks(())
plt.ylim(-2.5, 2.5)
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.059 seconds)

Bar plots

An example of bar plots with matplotlib.

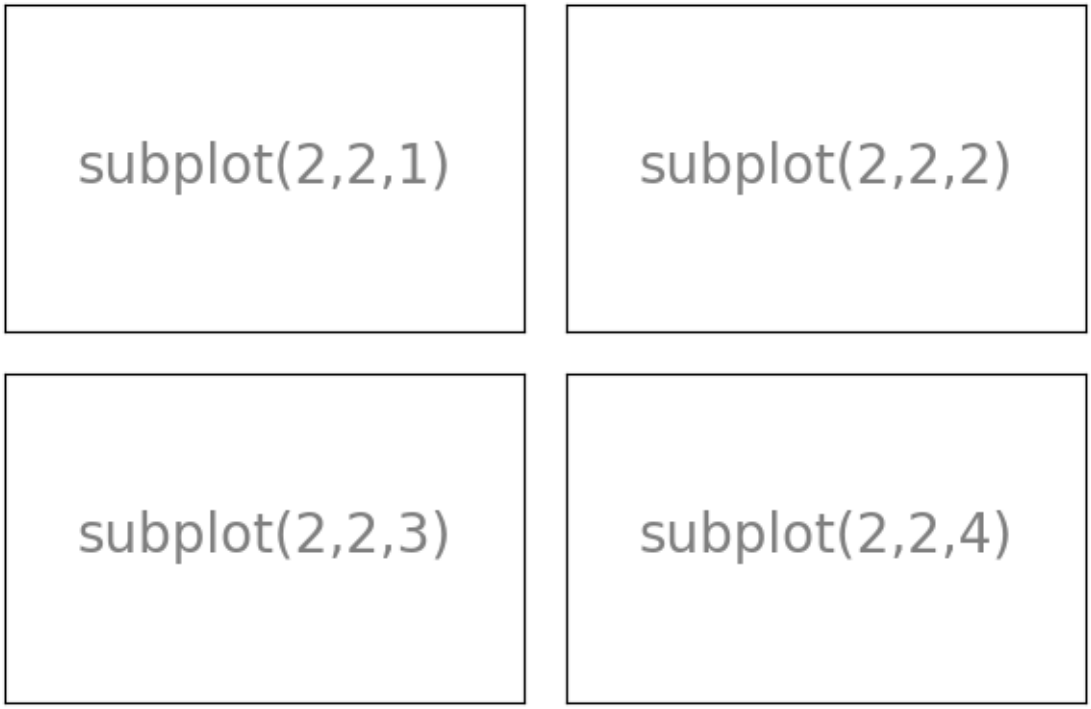



```
import numpy as np
import matplotlib.pyplot as plt

n = 12
X = np.arange(n)
Y1 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
Y2 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)

plt.axes([0.025, 0.025, 0.95, 0.95])
plt.bar(X, +Y1, facecolor='#9999ff', edgecolor='white')
plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x, y in zip(X, Y1):
    plt.text(x + 0.4, y + 0.05, '%.2f' % y, ha='center', va= 'bottom')

for x, y in zip(X, Y2):
    plt.text(x + 0.4, -y - 0.05, '%.2f' % y, ha='center', va= 'top')

plt.xlim(-.5, n)
plt.xticks(())
plt.ylim(-1.25, 1.25)
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.080 seconds)

Subplot grid

An example showing the subplot grid in matplotlib.

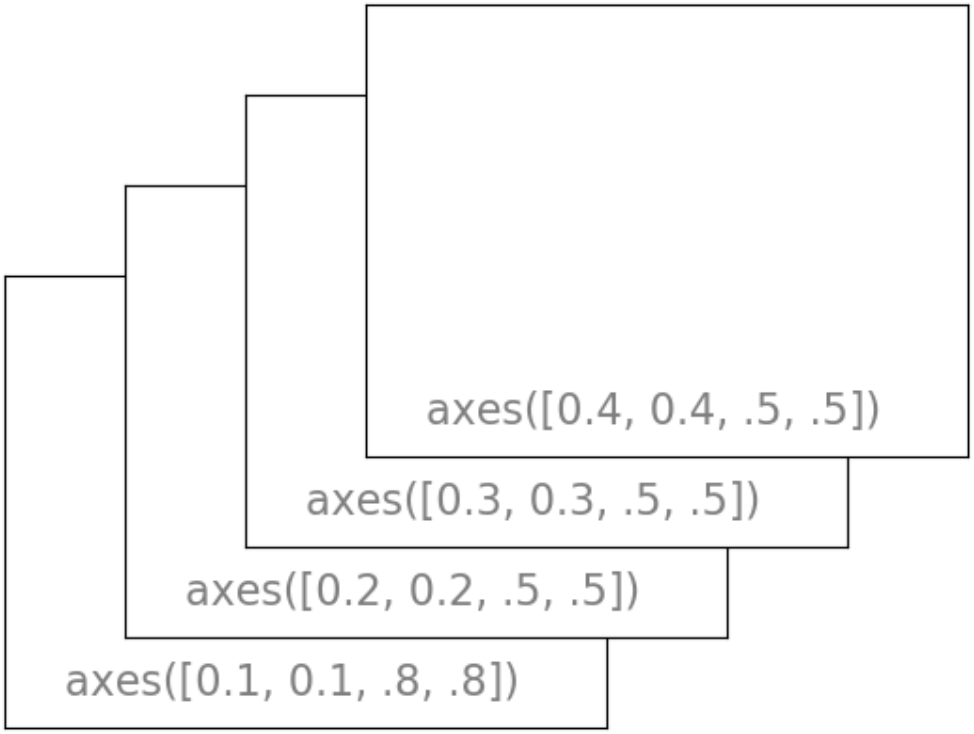



```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.subplot(2, 2, 1)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(2,2,1)', ha='center', va='center',
        size=20, alpha=.5)

plt.subplot(2, 2, 2)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(2,2,2)', ha='center', va='center',
        size=20, alpha=.5)

plt.subplot(2, 2, 3)
plt.xticks(())
plt.yticks(())

plt.text(0.5, 0.5, 'subplot(2,2,3)', ha='center', va='center',
        size=20, alpha=.5)

plt.subplot(2, 2, 4)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'subplot(2,2,4)', ha='center', va='center',
        size=20, alpha=.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.265 seconds)

Axes

This example shows various axes command to position matplotlib axes.

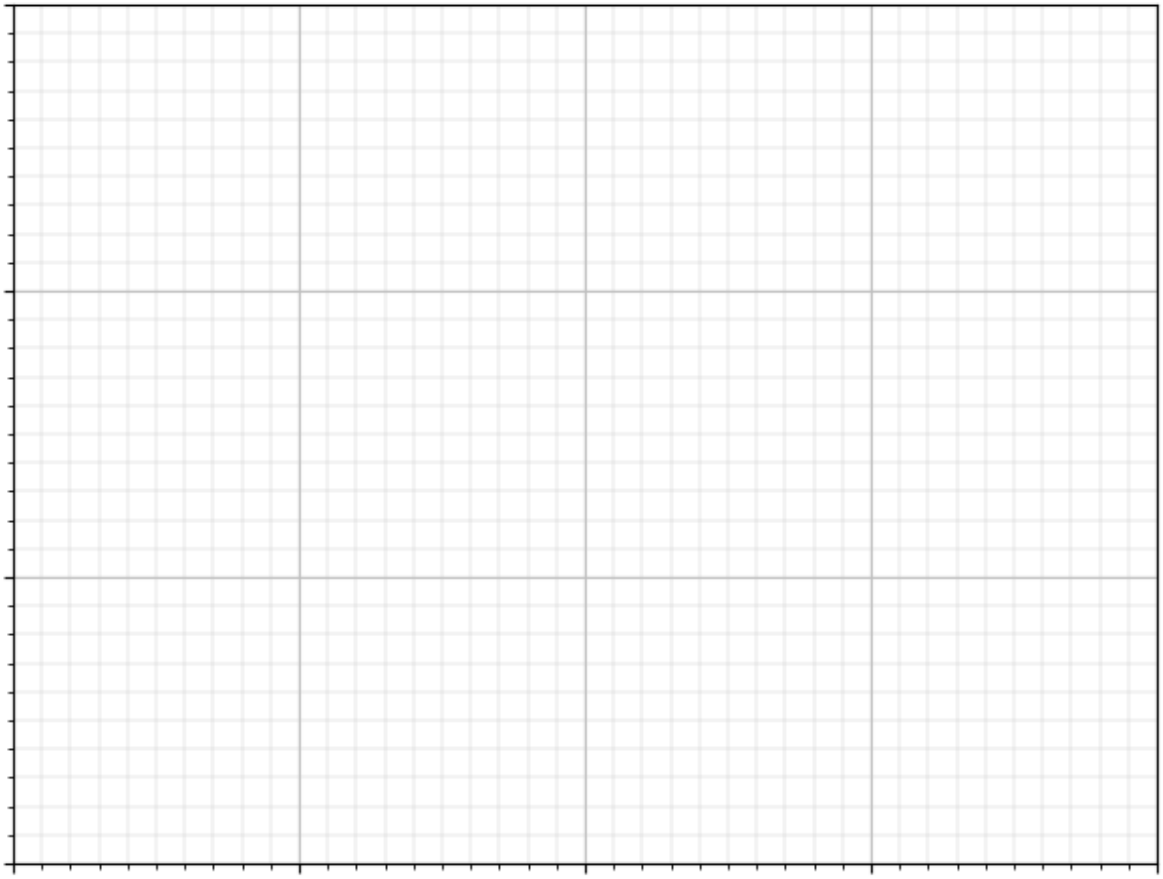



```
import matplotlib.pyplot as plt

plt.axes([.1, .1, .5, .5])
plt.xticks(())
plt.yticks(())
plt.text(0.1, 0.1, 'axes([0.1, 0.1, .8, .8])', ha='left', va='center',
        size=16, alpha=.5)

plt.axes([.2, .2, .5, .5])
plt.xticks(())
plt.yticks(())
plt.text(0.1, 0.1, 'axes([0.2, 0.2, .5, .5])', ha='left', va='center',
        size=16, alpha=.5)

plt.axes([0.3, 0.3, .5, .5])
plt.xticks(())
plt.yticks(())
plt.text(0.1, 0.1, 'axes([0.3, 0.3, .5, .5])', ha='left', va='center',
        size=16, alpha=.5)

plt.axes([.4, .4, .5, .5])
plt.xticks(())
plt.yticks(())
plt.text(0.1, 0.1, 'axes([0.4, 0.4, .5, .5])', ha='left', va='center',
        size=16, alpha=.5)

plt.show()
```

Total running time of the script: (0 minutes 0.192 seconds)

Grid

Displaying a grid on the axes in matploblib.

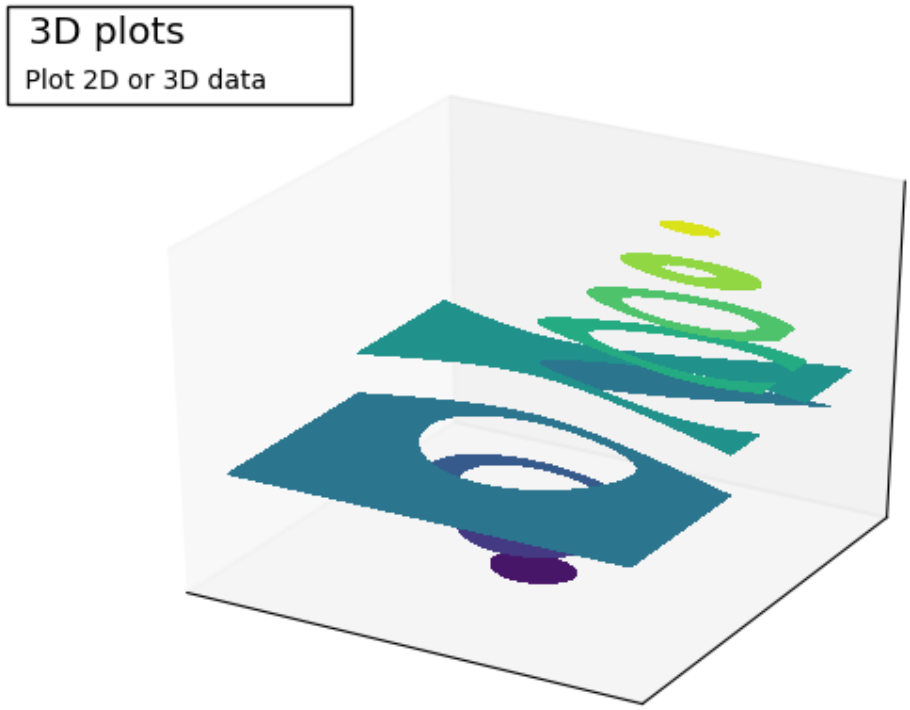


```
import matplotlib.pyplot as plt

ax = plt.axes([0.025, 0.025, 0.95, 0.95])

ax.set_xlim(0,4)
ax.set_ylim(0,3)
ax.xaxis.set_major_locator(plt.MultipleLocator(1.0))
ax.xaxis.set_minor_locator(plt.MultipleLocator(0.1))
ax.yaxis.set_major_locator(plt.MultipleLocator(1.0))
ax.yaxis.set_minor_locator(plt.MultipleLocator(0.1))
ax.grid(which='major', axis='x', linewidth=0.75, linestyle='-', color='0.75')
ax.grid(which='minor', axis='x', linewidth=0.25, linestyle='-', color='0.75')
ax.grid(which='major', axis='y', linewidth=0.75, linestyle='-', color='0.75')
ax.grid(which='minor', axis='y', linewidth=0.25, linestyle='-', color='0.75')
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
```

Total running time of the script: (0 minutes 0.066 seconds)

3D plotting

Demo 3D plotting with matplotlib and style the figure.




```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

ax = plt.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contourf(X, Y, Z)
ax.clabel(cset, fontsize=9, inline=1)

plt.xticks(())
plt.yticks(())
ax.set_zticks(())


ax.text2D(-0.05, 1.05, " 3D plots             \n",
          horizontalalignment='left',
          verticalalignment='top',
          bbox=dict(facecolor='white', alpha=1.0),
          family='Lint McCree Intl BB',
          size='x-large',
          transform=plt.gca().transAxes)

ax.text2D(-0.05, .975, " Plot 2D or 3D data",
          horizontalalignment='left',
          verticalalignment='top',
          family='Lint McCree Intl BB',
          size='medium',
          transform=plt.gca().transAxes)
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.060 seconds)

GridSpec

An example demoing gridspec

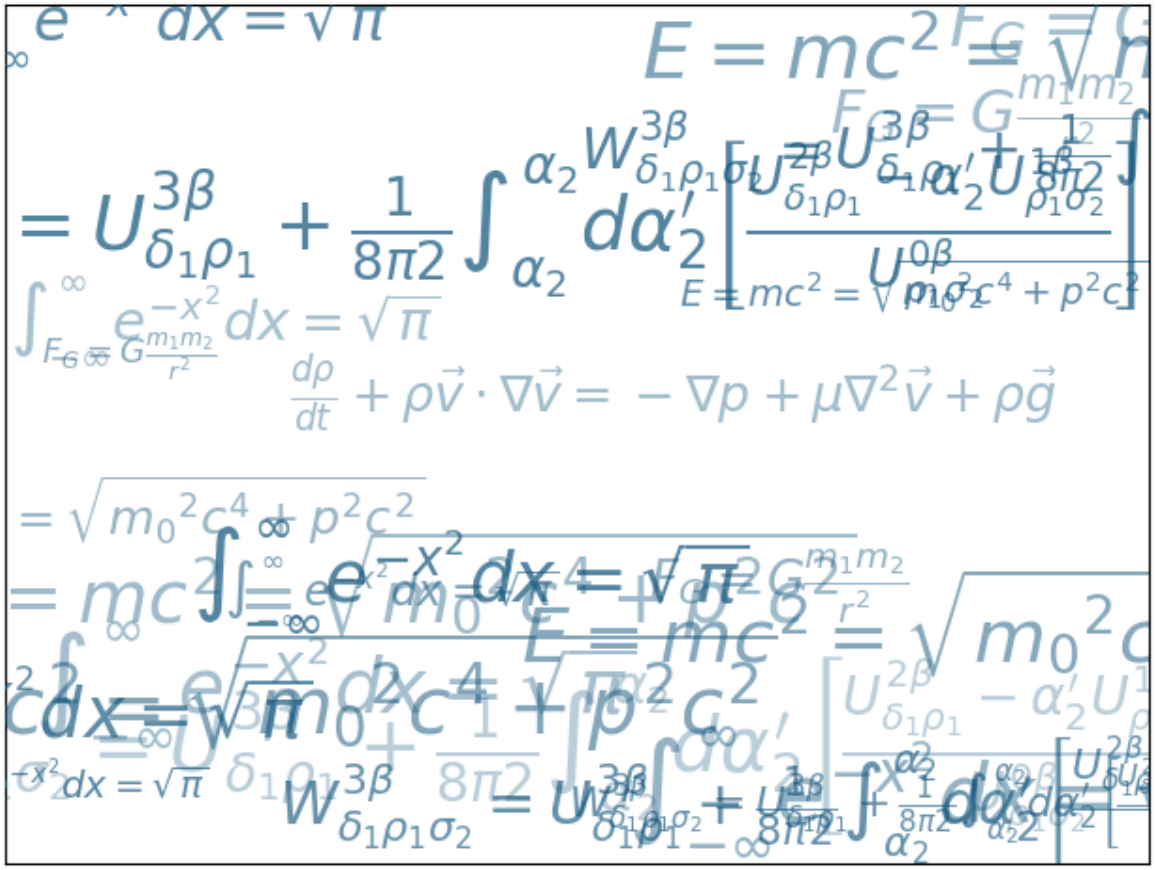

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

plt.figure(figsize=(6, 4))
G = gridspec.GridSpec(3, 3)

axes_1 = plt.subplot(G[0, :])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'Axes 1', ha='center', va='center', size=24, alpha=.5)

axes_2 = plt.subplot(G[1, :-1])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'Axes 2', ha='center', va='center', size=24, alpha=.5)

axes_3 = plt.subplot(G[1:, -1])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'Axes 3', ha='center', va='center', size=24, alpha=.5)

axes_4 = plt.subplot(G[-1, 0])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'Axes 4', ha='center', va='center', size=24, alpha=.5)
```

```
axes_5 = plt.subplot(G[-1, -2])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'Axes 5', ha='center', va='center', size=24, alpha=.5)

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.317 seconds)

Demo text printing

A example showing off elaborate text printing with matplotlib.



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


eqs = []
eqs.append((r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}
↪{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\delta_1␣
↪\rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]$
↪"))
eqs.append((r"$\frac{d\rho}{d t} + \rho \vec{v}\cdot\nabla\vec{v} = -\nabla p + \mu\nabla^2␣
↪\vec{v} + \rho \vec{g}$"))
eqs.append((r"$\int_{-\infty}^\infty e^{-x^2}dx=\sqrt{\pi}$"))
eqs.append((r"$E = mc^2 = \sqrt{{m_0}^2c^4 + p^2c^2}$"))
eqs.append((r"$F_G = G\frac{m_1m_2}{r^2}$"))
```

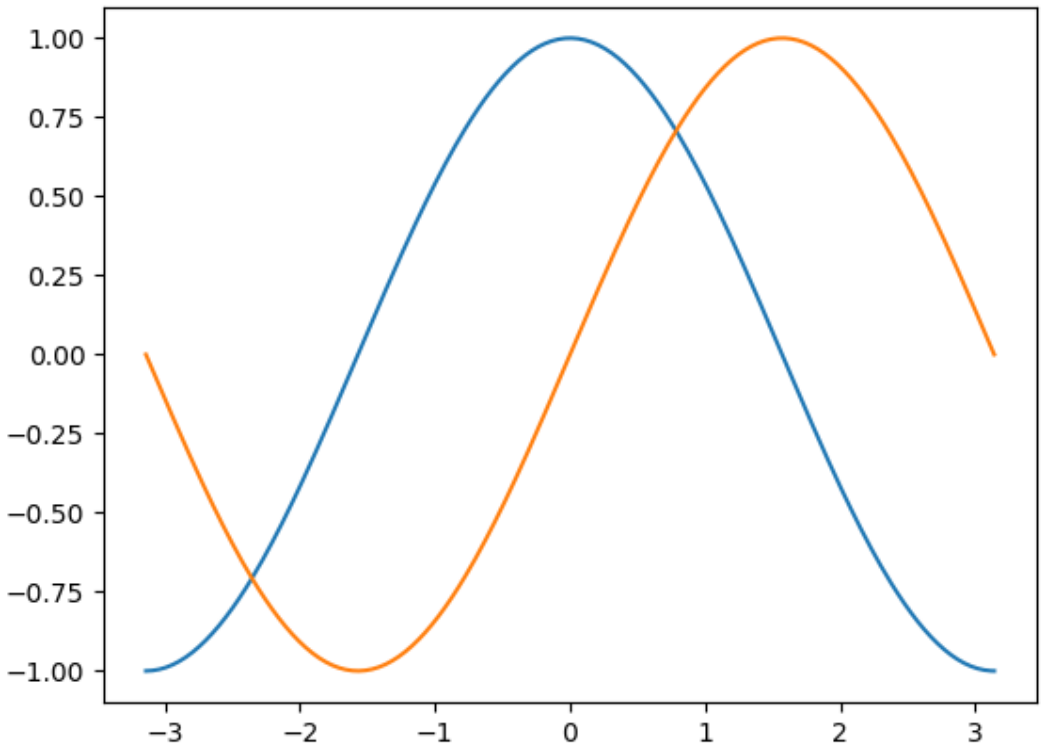
```
plt.axes([0.025, 0.025, 0.95, 0.95])

for i in range(24):
    index = np.random.randint(0, len(eqs))
    eq = eqs[index]
    size = np.random.uniform(12, 32)
    x,y = np.random.uniform(0, 1, 2)
    alpha = np.random.uniform(0.25, .75)
    plt.text(x, y, eq, ha='center', va='center', color="#11557c", alpha=alpha,
         transform=plt.gca().transAxes, fontsize=size, clip_on=True)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.058 seconds)

2.7.2 Code for the chapter's exercises

Excercise 1

Solution of the excercise 1 with matplotlib.



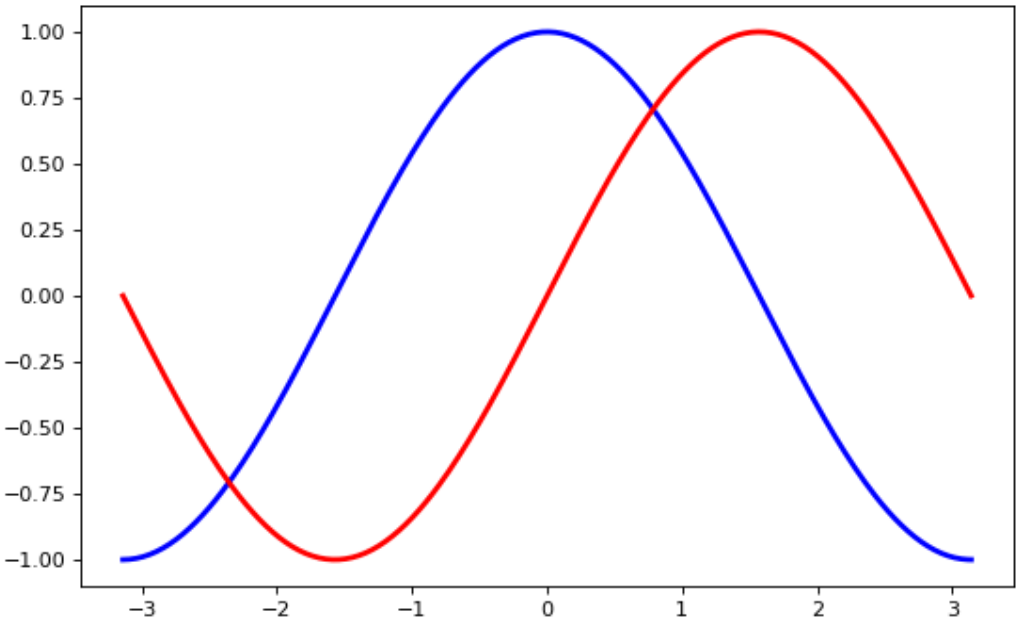

```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C,S = np.cos(X), np.sin(X)
```

```
plt.plot(X, C)
plt.plot(X,S)

plt.show()
```

Total running time of the script: (0 minutes 0.051 seconds)

Exercise 4

Exercise 4 with matplotlib.

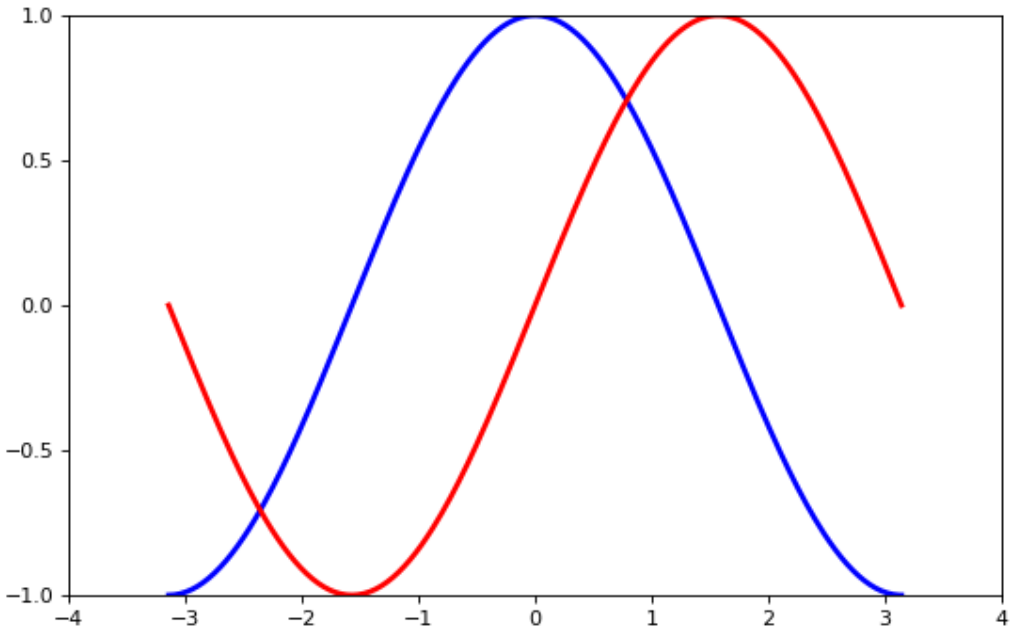



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
S = np.sin(X)
C = np.cos(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)

plt.show()
```

Total running time of the script: (0 minutes 0.052 seconds)

Exercise 3

Exercise 3 with matplotlib.

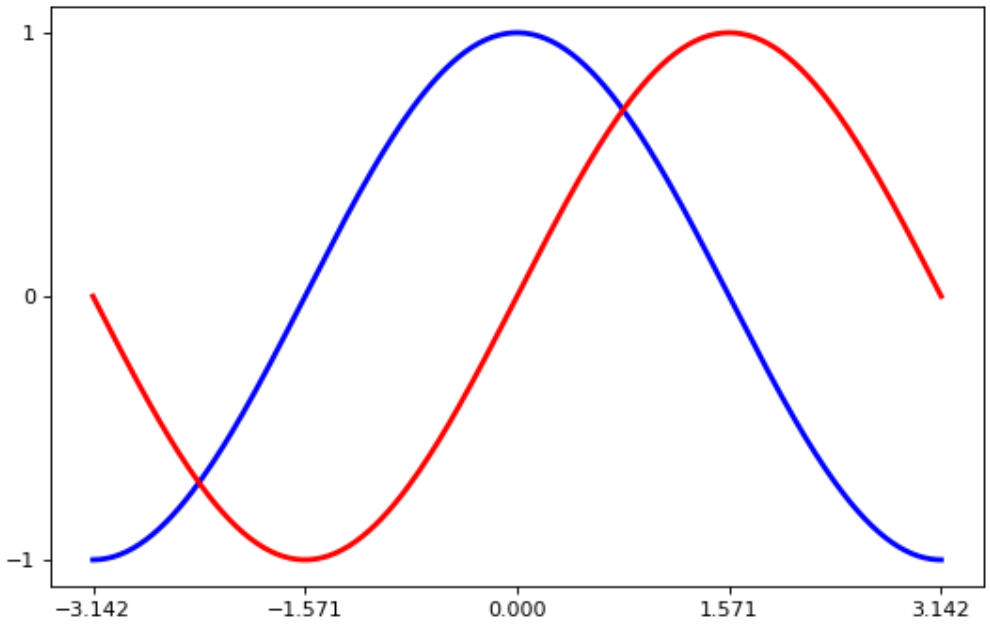



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(-4.0, 4.0)
plt.xticks(np.linspace(-4, 4, 9, endpoint=True))

plt.ylim(-1.0, 1.0)
plt.yticks(np.linspace(-1, 1, 5, endpoint=True))

plt.show()
```

Total running time of the script: (0 minutes 0.069 seconds)

Exercise 5

Exercise 5 with matplotlib.

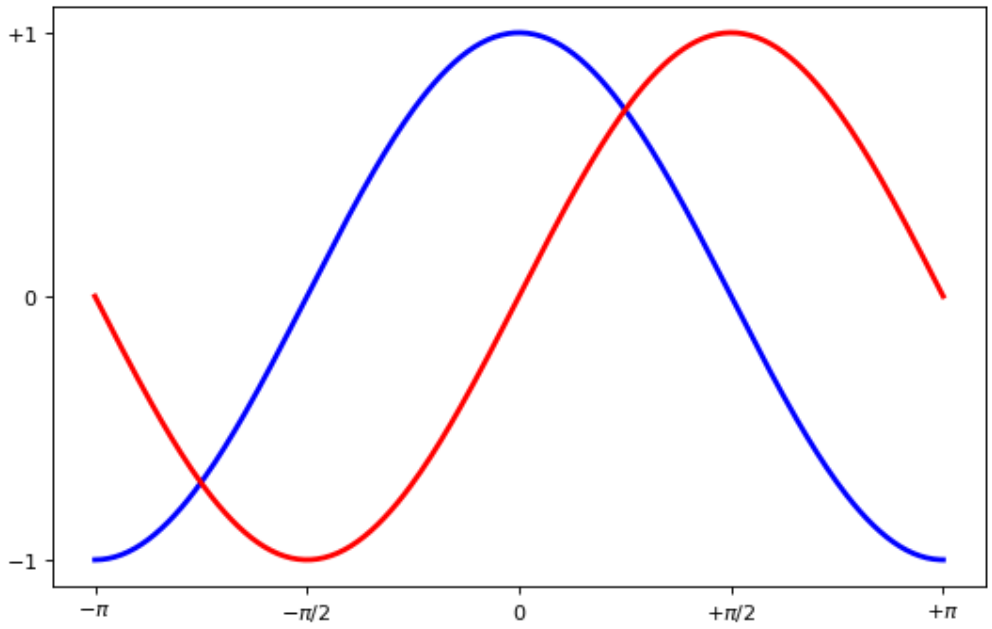



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
S = np.sin(X)
C = np.cos(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 0, +1])

plt.show()
```

Total running time of the script: (0 minutes 0.060 seconds)

Exercise 6

Exercise 6 with matplotlib.

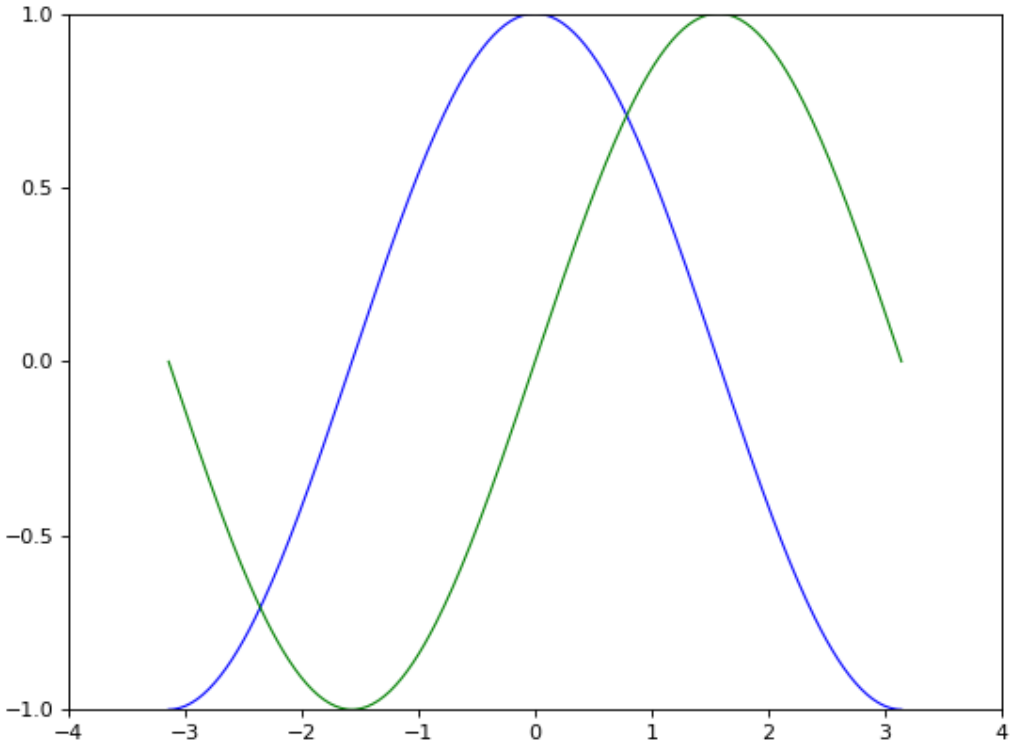



```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
          [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 0, +1],
          [r'$-1$', r'$0$', r'$+1$'])

plt.show()
```

Total running time of the script: (0 minutes 0.062 seconds)

Exercise 2

Exercise 2 with matplotlib.



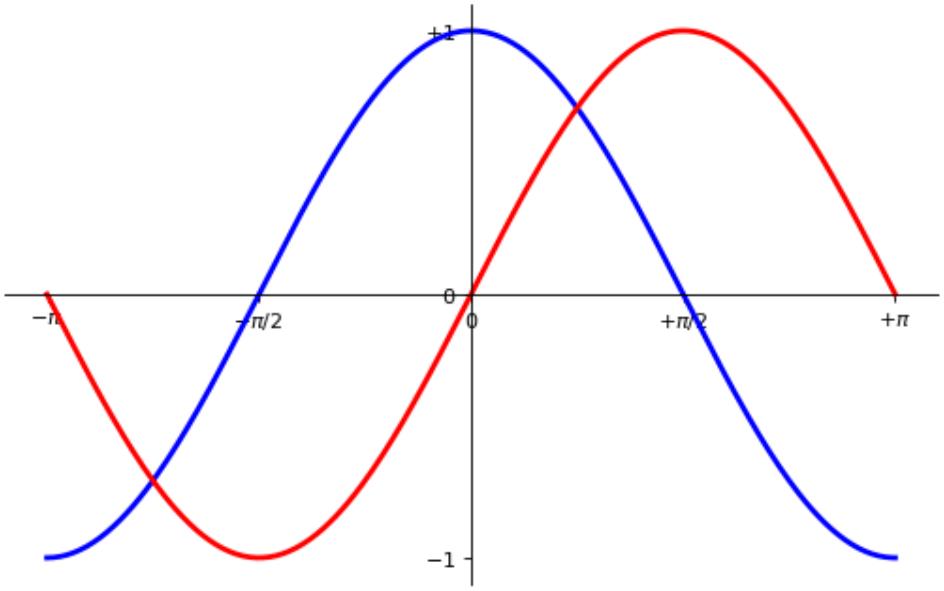

```
import numpy as np
import matplotlib.pyplot as plt

# Create a new figure of size 8x6 points, using 100 dots per inch
plt.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

# Plot cosine using blue color with a continuous line of width 1 (pixels)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine using green color with a continuous line of width 1 (pixels)
plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
plt.xlim(-4., 4.)

# Set x ticks
plt.xticks(np.linspace(-4, 4, 9, endpoint=True))

# Set y limits
plt.ylim(-1.0, 1.0)

# Set y ticks
plt.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Show result on screen
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.068 seconds)

Exercise 7

Exercise 7 with matplotlib



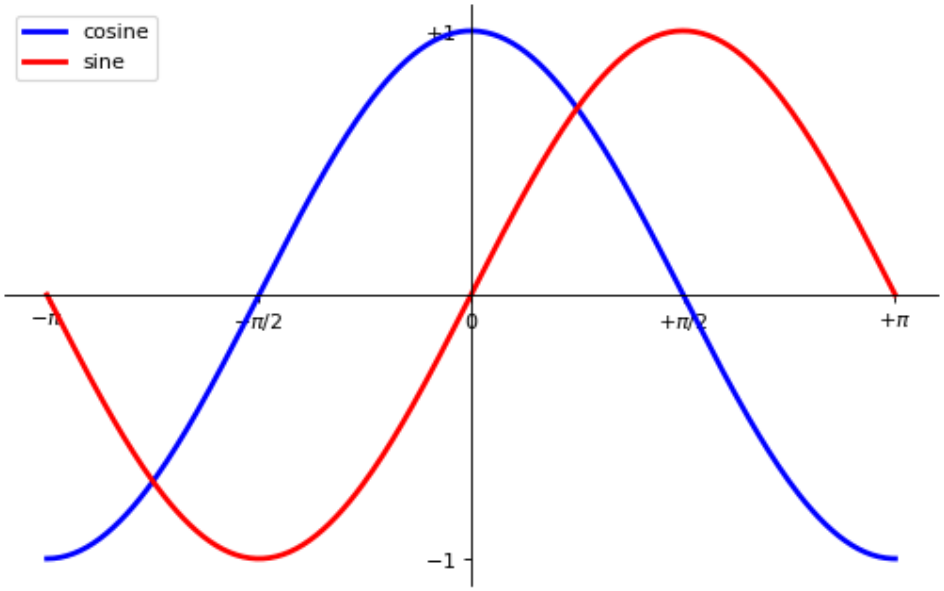

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-")

ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
          [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 0, +1],
```

```
          [r'$-1$', r'$0$', r'$+1$'])

plt.show()
```

Total running time of the script: (0 minutes 0.069 seconds)

Exercise 8

Exercise 8 with matplotlib.



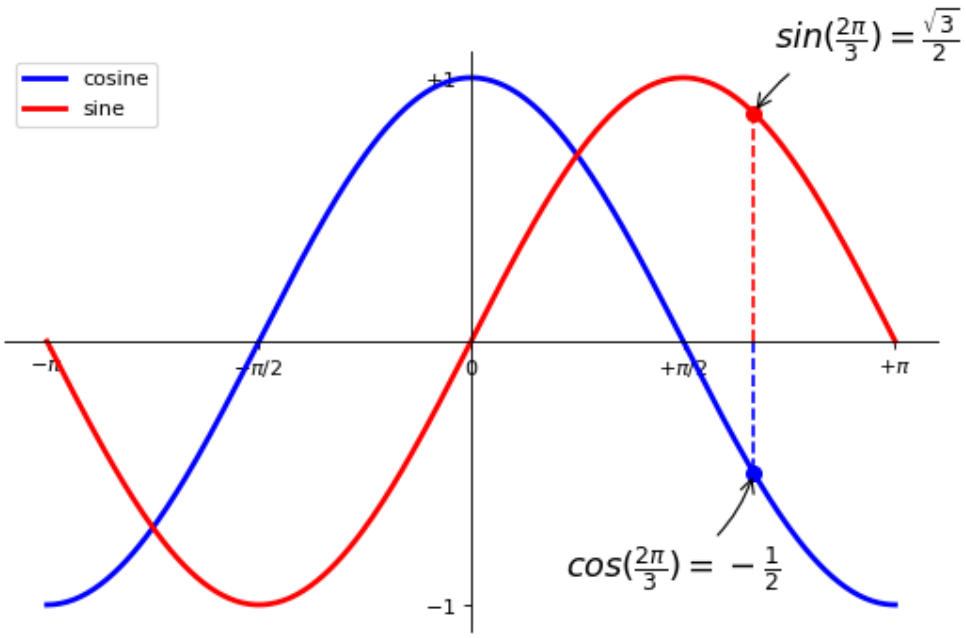

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-",  label="sine")

ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
          [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
```

```
plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, +1],
           [r'$-1$', r'$+1$'])

plt.legend(loc='upper left')

plt.show()
```

Total running time of the script: (0 minutes 0.071 seconds)

Exercise 9

Exercise 9 with matplotlib.



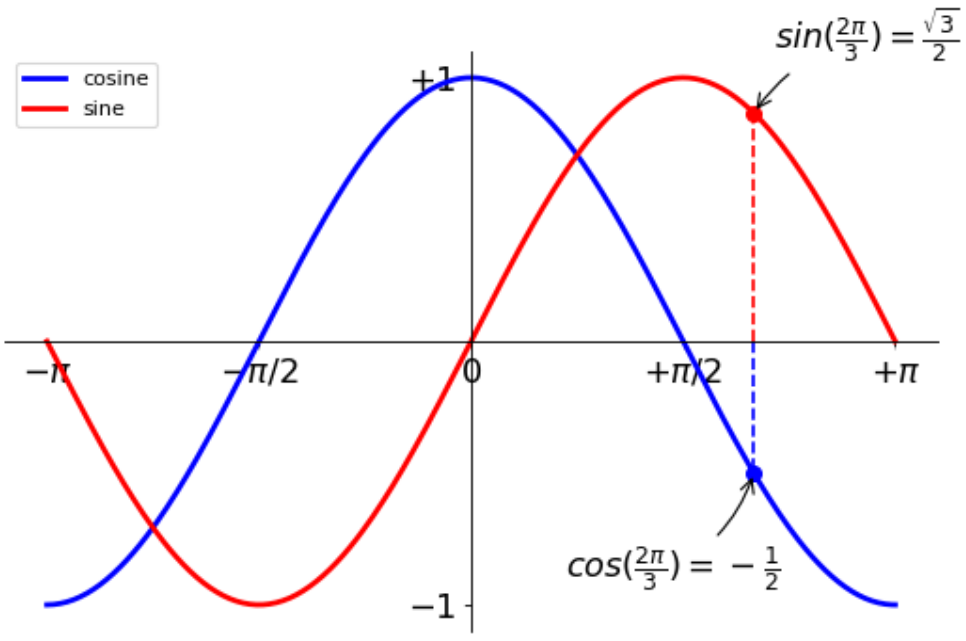

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-",  label="sine")

ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
```

```
plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
          [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, +1],
          [r'$-1$', r'$+1$'])

t = 2*np.pi/3
plt.plot([t, t], [0, np.cos(t)],
        color='blue', linewidth=1.5, linestyle="--")
plt.scatter([t, ], [np.cos(t), ], 50, color='blue')
plt.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
            xy=(t, np.sin(t)), xycoords='data',
            xytext=(+10, +30), textcoords='offset points', fontsize=16,
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plt.plot([t, t], [0, np.sin(t)],
        color='red', linewidth=1.5, linestyle="--")
plt.scatter([t, ], [np.sin(t), ], 50, color='red')
plt.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$', xy=(t, np.cos(t)),
            xycoords='data', xytext=(-90, -50), textcoords='offset points',
            fontsize=16,
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plt.legend(loc='upper left')

plt.show()
```

Total running time of the script: (0 minutes 0.076 seconds)

Exercise

Exercises with matplotlib.



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

plt.figure(figsize=(8, 5), dpi=80)
plt.subplot(111)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red", linewidth=2.5, linestyle="-",  label="sine")

ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data', 0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data', 0))

plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.xticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi],
          [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.ylim(C.min() * 1.1, C.max() * 1.1)
plt.yticks([-1, 1],
          [r'$-1$', r'$+1$'])

plt.legend(loc='upper left')

t = 2*np.pi/3
plt.plot([t, t], [0, np.cos(t)],
        color='blue', linewidth=1.5, linestyle="--")
plt.scatter([t, ], [np.cos(t), ], 50, color='blue')
plt.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
            xy=(t, np.sin(t)), xycoords='data',
            xytext=(10, 30), textcoords='offset points', fontsize=16,
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plt.plot([t, t], [0, np.sin(t)],
        color='red', linewidth=1.5, linestyle="--")
plt.scatter([t, ], [np.sin(t), ], 50, color ='red')
plt.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$', xy=(t, np.cos(t)),
            xycoords='data', xytext=(-90, -50),
            textcoords='offset points', fontsize=16,
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
    label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65 ))

plt.show()
```

Total running time of the script: (0 minutes 0.077 seconds)

2.7.3 Example demoing choices for an option

The colors matplotlib line plots

An example demoing the various colors taken by matplotlib's plot.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0.1, 1, .8], frameon=False)

for i in range(1,11):
    plt.plot([i, i], [0, 1], lw=1.5)

plt.xlim(0, 11)
plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 0.062 seconds)

Linewidth

Plot various linewidth with matplotlib.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, .1, 1, .8], frameon=False)

for i in range(1, 11):
    plt.plot([i, i], [0, 1], color='b', lw=i/2.)

plt.xlim(0, 11)
plt.ylim(0, 1)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.064 seconds)

Alpha: transparency

This example demonstrates using alpha for transparency.

```
import matplotlib.pyplot as plt

size = 256,16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
```

```
fig.patch.set_alpha(0)
plt.axes([0, 0.1, 1, .8], frameon=False)

for i in range(1, 11):
    plt.axvline(i, linewidth=1, color='blue', alpha= .25 + .75 * i / 10.)

plt.xlim(0, 11)
plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 0.066 seconds)

Aliased versus anti-aliased

This example demonstrates aliased versus anti-aliased text.


```
import matplotlib.pyplot as plt

size = 128, 16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)

plt.axes([0, 0, 1, 1], frameon=False)

plt.rcParams['text.antialiased'] = False
plt.text(0.5, 0.5, "Aliased", ha='center', va='center')

plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.050 seconds)

Aliased versus anti-aliased

The example shows aliased versus anti-aliased text.


```
import matplotlib.pyplot as plt

size = 128, 16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

plt.rcParams['text.antialiased'] = True
plt.text(0.5, 0.5, "Anti-aliased", ha='center', va='center')

plt.xlim(0, 1)
```

```
plt.ylim(0, 1)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.049 seconds)

Marker size

Demo the marker size control in matplotlib.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

for i in range(1, 11):
    plt.plot([i, ], [1, ], 's', markersize=i, markerfacecolor='w',
         markeredgewidth=.5, markeredgecolor='k')

plt.xlim(0, 11)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.067 seconds)

Marker edge width

Demo the marker edge widths of matplotlib's markers.

```
import matplotlib.pyplot as plt

size = 256, 16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

for i in range(1,11):
    plt.plot([i, ], [1, ], 's', markersize=5,
         markeredgewidth=1 + i/10., markeredgecolor='k', markerfacecolor='w')
plt.xlim(0, 11)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.064 seconds)

Marker edge color

Demo the marker edge color of matplotlib's markers.

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

size = 256,16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

for i in range(1, 11):
    r, g, b = np.random.uniform(0, 1, 3)
    plt.plot([i, ], [1, ], 's', markersize=5, markerfacecolor='w',
             markeredgewidth=1.5, markeredgecolor=(r, g, b, 1))

plt.xlim(0, 11)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.064 seconds)

Marker face color

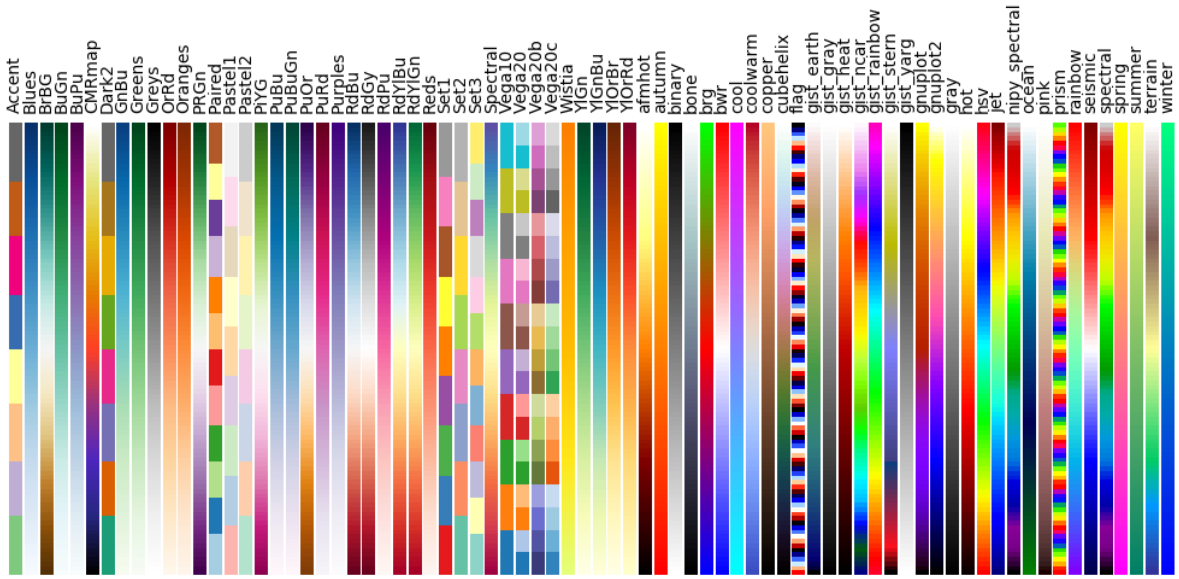
Demo the marker face color of matplotlib's markers.

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

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

for i in range(1, 11):
    r, g, b = np.random.uniform(0, 1, 3)
    plt.plot([i, ], [1, ], 's', markersize=8, markerfacecolor=(r, g, b, 1),
             markeredgewidth=.1,  markeredgecolor=(0, 0, 0, .5))
plt.xlim(0, 11)
plt.xticks(())
plt.yticks(())
plt.show()
```

Total running time of the script: (0 minutes 0.064 seconds)

Colormaps

An example plotting the matplotlib colormaps.




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

plt.rc('text', usetex=False)
a = np.outer(np.arange(0, 1, 0.01), np.ones(10))

plt.figure(figsize=(10, 5))
plt.subplots_adjust(top=0.8, bottom=0.05, left=0.01, right=0.99)
maps = [m for m in plt.cm.datad if not m.endswith("_r")]
maps.sort()
l = len(maps) + 1

for i, m in enumerate(maps):
    plt.subplot(1, l, i+1)
    plt.axis("off")
    plt.imshow(a, aspect='auto', cmap=plt.get_cmap(m), origin="lower")
    plt.title(m, rotation=90, fontsize=10, va='bottom')

plt.show()
```

Total running time of the script: (0 minutes 4.020 seconds)

Solid cap style

An example demoing the solide cap style in matplotlib.

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

size = 256, 16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

plt.plot(np.arange(4), np.ones(4), color="blue", linewidth=8,
        solid_capstyle='butt')
```

```
plt.plot(5 + np.arange(4), np.ones(4), color="blue", linewidth=8,
        solid_capstyle='round')

plt.plot(10 + np.arange(4), np.ones(4), color="blue", linewidth=8,
        solid_capstyle='projecting')

plt.xlim(0, 14)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

Solid joint style

An example showing the differen solid joint styles in matplotlib.

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

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

plt.plot(np.arange(3), [0, 1, 0], color="blue", linewidth=8,
        solid_joinstyle='miter')
plt.plot(4 + np.arange(3), [0, 1, 0], color="blue", linewidth=8,
        solid_joinstyle='bevel')
plt.plot(8 + np.arange(3), [0, 1, 0], color="blue", linewidth=8,
        solid_joinstyle='round')

plt.xlim(0, 12)
plt.ylim(-1, 2)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

Dash capstyle

An example demoing the dash capstyle.

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

size = 256, 16
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
```

```
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

plt.plot(np.arange(4), np.ones(4), color="blue", dashes=[15, 15],
        linewidth=8, dash_capstyle='butt')

plt.plot(5 + np.arange(4), np.ones(4), color="blue", dashes=[15, 15],
        linewidth=8, dash_capstyle='round')

plt.plot(10 + np.arange(4), np.ones(4), color="blue", dashes=[15, 15],
        linewidth=8, dash_capstyle='projecting')

plt.xlim(0, 14)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

Dash join style

Example demoing the dash join style.

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

size = 256, 16
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)
plt.axes([0, 0, 1, 1], frameon=False)

plt.plot(np.arange(3), [0, 1, 0], color="blue", dashes=[12, 5], linewidth=8,
        dash_joinstyle='miter')
plt.plot(4 + np.arange(3), [0, 1, 0], color="blue", dashes=[12, 5],
        linewidth=8, dash_joinstyle='bevel')
plt.plot(8 + np.arange(3), [0, 1, 0], color="blue", dashes=[12, 5],
        linewidth=8, dash_joinstyle='round')

plt.xlim(0, 12)
plt.ylim(-1, 2)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.334 seconds)

Linestyles

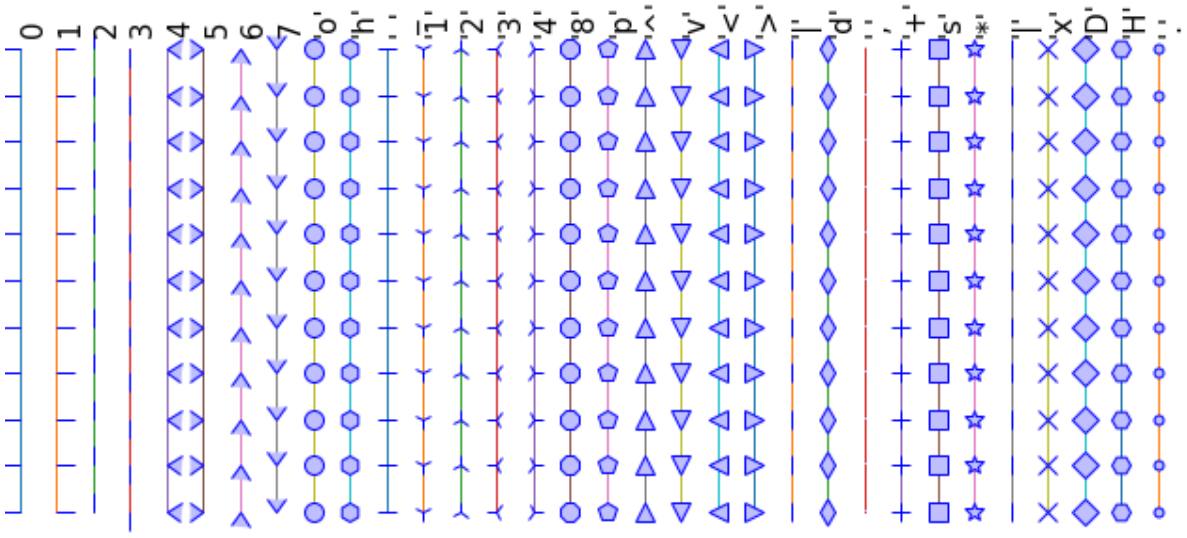
Plot the different line styles.

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

def linestyle(ls, i):
    X = i * .5 * np.ones(11)
    Y = np.arange(11)
    plt.plot(X, Y, ls, color=(.0, .0, 1, 1), lw=3, ms=8,
             mfc=(.75, .75, 1, 1), mec=(0, 0, 1, 1))
    plt.text(.5 * i, 10.25, ls, rotation=90, fontsize=15, va='bottom')

linestyles = ['-', '--', ':', '-.', '.', ',', 'o', '^', 'v', '<', '>', 's',
              '+', 'x', 'd', '1', '2', '3', '4', 'h', 'p', '|', '_', 'D', 'H']
n_lines = len(linestyles)

size = 20 * n_lines, 300
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
plt.axes([0, 0.01, 1, .9], frameon=False)

for i, ls in enumerate(linestyles):
    linestyle(ls, i)

plt.xlim(-.2, .2 + .5*n_lines)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.101 seconds)

Markers

Show the different markers of matplotlib.

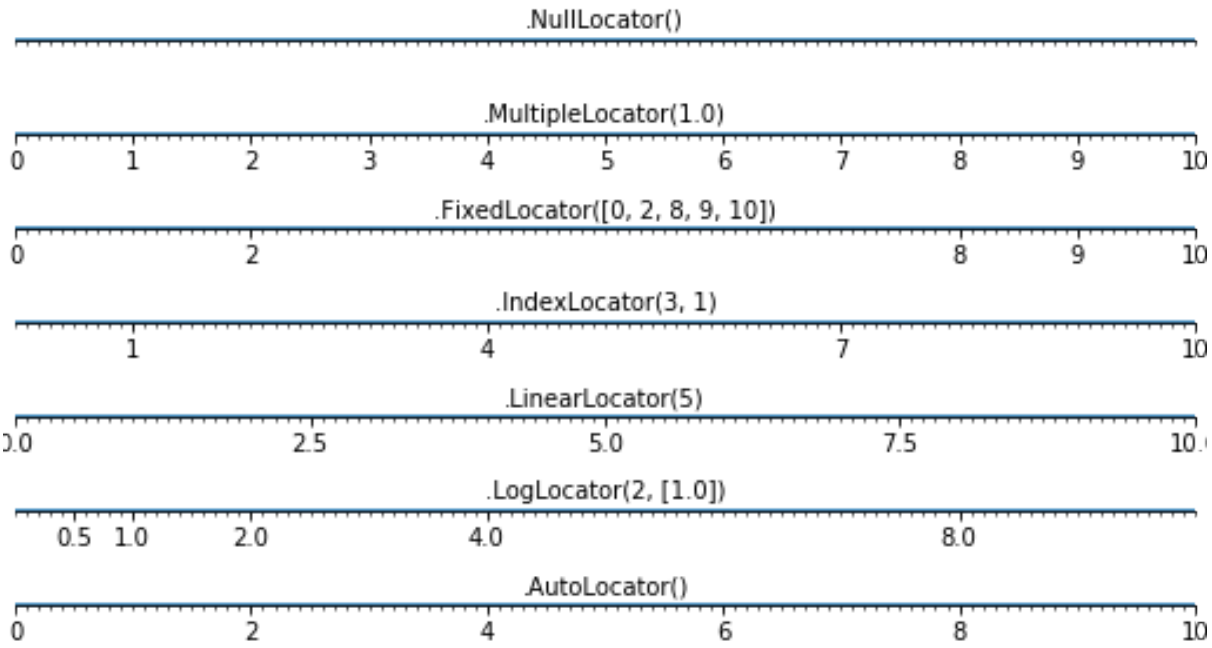



```
import numpy as np
import matplotlib.pyplot as plt

def marker(m, i):
    X = i * .5 * np.ones(11)
    Y = np.arange(11)

    plt.plot(X, Y, lw=1, marker=m, ms=10, mfc=(.75, .75, 1, 1),
             mec=(0, 0, 1, 1))
    plt.text(.5 * i, 10.25, repr(m), rotation=90, fontsize=15, va='bottom')

markers = [0, 1, 2, 3, 4, 5, 6, 7, 'o', 'h', '_', '1', '2', '3', '4',
           '8', 'p', '^', 'v', '<', '>', '|', 'd', ',', '+', 's', '*',
           '|', 'x', 'D', 'H', '.']

n_markers = len(markers)

size = 20 * n_markers, 300
dpi = 72.0
figsize= size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
plt.axes([0, 0.01, 1, .9], frameon=False)

for i, m in enumerate(markers):
    marker(m, i)

plt.xlim(-.2, .2 + .5 * n_markers)
plt.xticks(())
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.112 seconds)

Locators for tick on axis

An example demoing different locators to position ticks on axis for matplotlib.




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


def tickline():
    plt.xlim(0, 10), plt.ylim(-1, 1), plt.yticks([])
    ax = plt.gca()
    ax.spines['right'].set_color('none')
    ax.spines['left'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.spines['bottom'].set_position(('data',0))
    ax.yaxis.set_ticks_position('none')
    ax.xaxis.set_minor_locator(plt.MultipleLocator(0.1))
    ax.plot(np.arange(11), np.zeros(11))
    return ax

locators = [
                'plt.NullLocator()',
                'plt.MultipleLocator(1.0)',
                'plt.FixedLocator([0, 2, 8, 9, 10])',
                'plt.IndexLocator(3, 1)',
                'plt.LinearLocator(5)',
                'plt.LogLocator(2, [1.0])',
                'plt.AutoLocator()',
            ]

n_locators = len(locators)

size = 512, 40 * n_locators
dpi = 72.0
figsize = size[0] / float(dpi), size[1] / float(dpi)
fig = plt.figure(figsize=figsize, dpi=dpi)
fig.patch.set_alpha(0)


for i, locator in enumerate(locators):
    plt.subplot(n_locators, 1, i + 1)
    ax = tickline()
    ax.xaxis.set_major_locator(eval(locator))
```

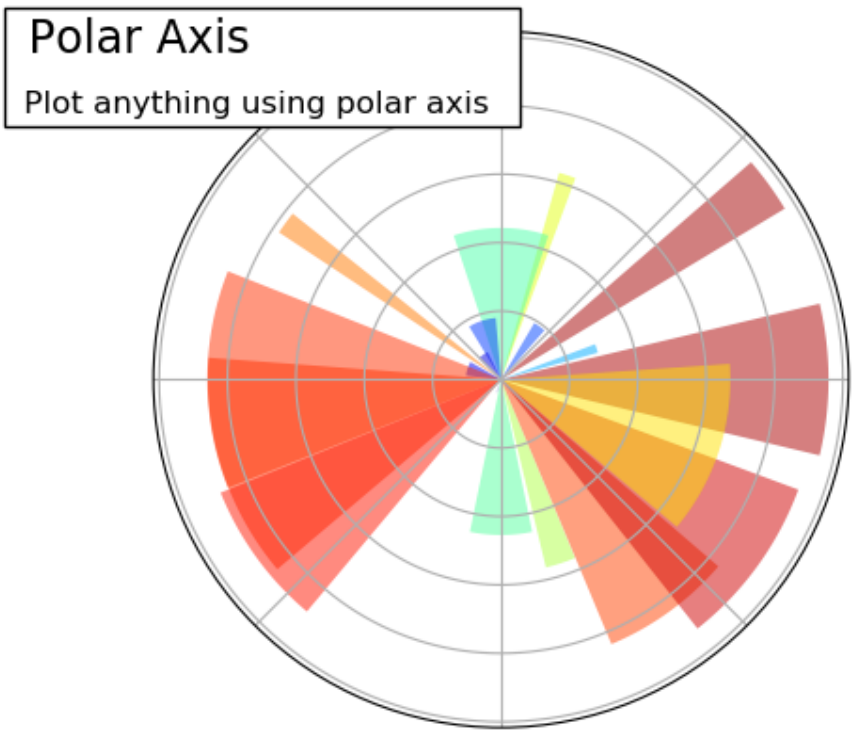
```
    plt.text(5, 0.3, locator[3:], ha='center')

plt.subplots_adjust(bottom=.01, top=.99, left=.01, right=.99)
plt.show()
```

Total running time of the script: (0 minutes 0.359 seconds)

2.7.4 Code generating the summary figures with a title

Plotting in polar, decorated

An example showing how to plot in polar coordinnate, and some decorations.



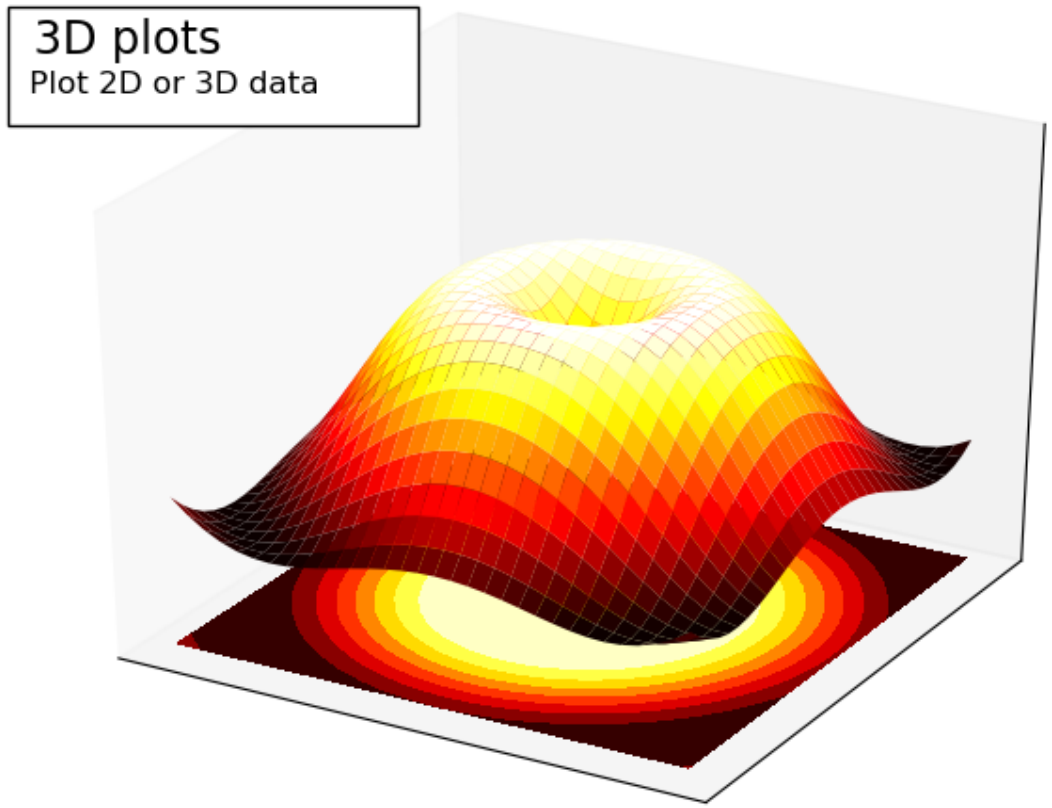

```
import numpy as np
import matplotlib.pyplot as plt

plt.subplot(1, 1, 1, polar=True)

N = 20
theta = np.arange(0.0, 2 * np.pi, 2 * np.pi / N)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)
plt.gca().set_xticklabels([])
plt.gca().set_yticklabels([])
```

```
plt.text(-0.2, 1.02, " Polar Axis                    \n",
      horizontalalignment='left',
      verticalalignment='top',
      size='xx-large',
      bbox=dict(facecolor='white', alpha=1.0),
      transform=plt.gca().transAxes)

plt.text(-0.2, 1.01, "\n\n Plot anything using polar axis ",
      horizontalalignment='left',
      verticalalignment='top',
      size='large',
      transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.173 seconds)

3D plotting vignette

Demo 3D plotting with matplotlib and decorate the figure.



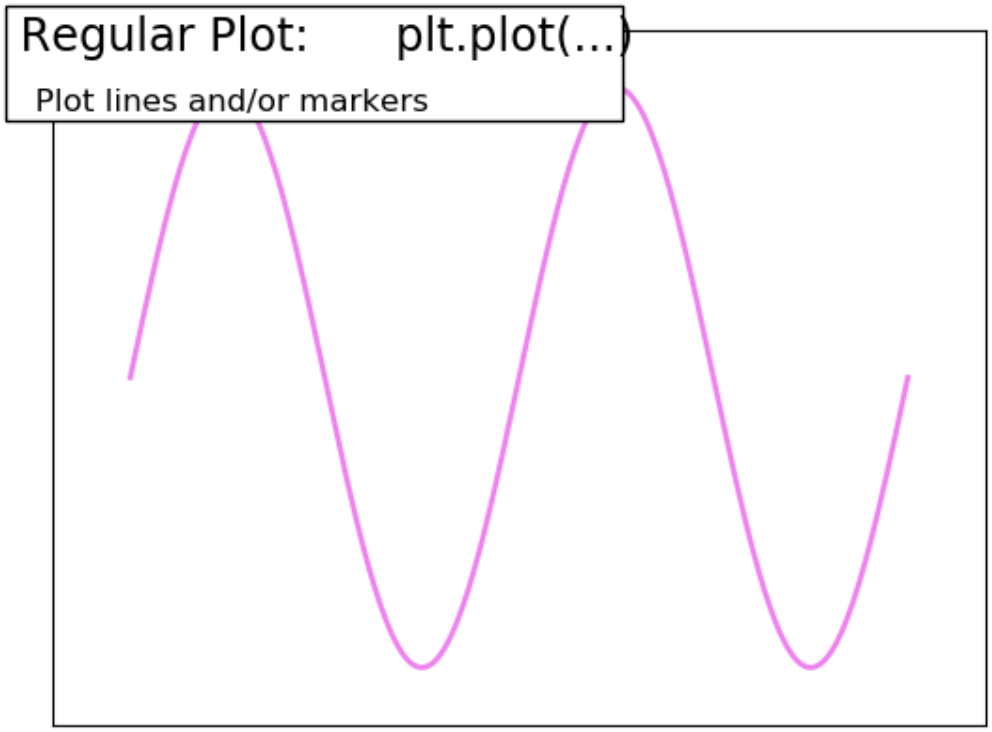

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
Y = np.arange(-4, 4, 0.25)
```

```
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X ** 2 + Y ** 2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.cm.hot)
ax.contourf(X, Y, Z, zdir='z', offset=-2, cmap=plt.cm.hot)
ax.set_zlim(-2, 2)
plt.xticks(())
plt.yticks(())
ax.set_zticks(())

ax.text2D(0.05, .93, " 3D plots              \n",
          horizontalalignment='left',
          verticalalignment='top',
          size='xx-large',
          bbox=dict(facecolor='white', alpha=1.0),
          transform=plt.gca().transAxes)

ax.text2D(0.05, .87, " Plot 2D or 3D data",
          horizontalalignment='left',
          verticalalignment='top',
          size='large',
          transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.076 seconds)

Plot example vignette

An example of plots with matplotlib, and added annotations.



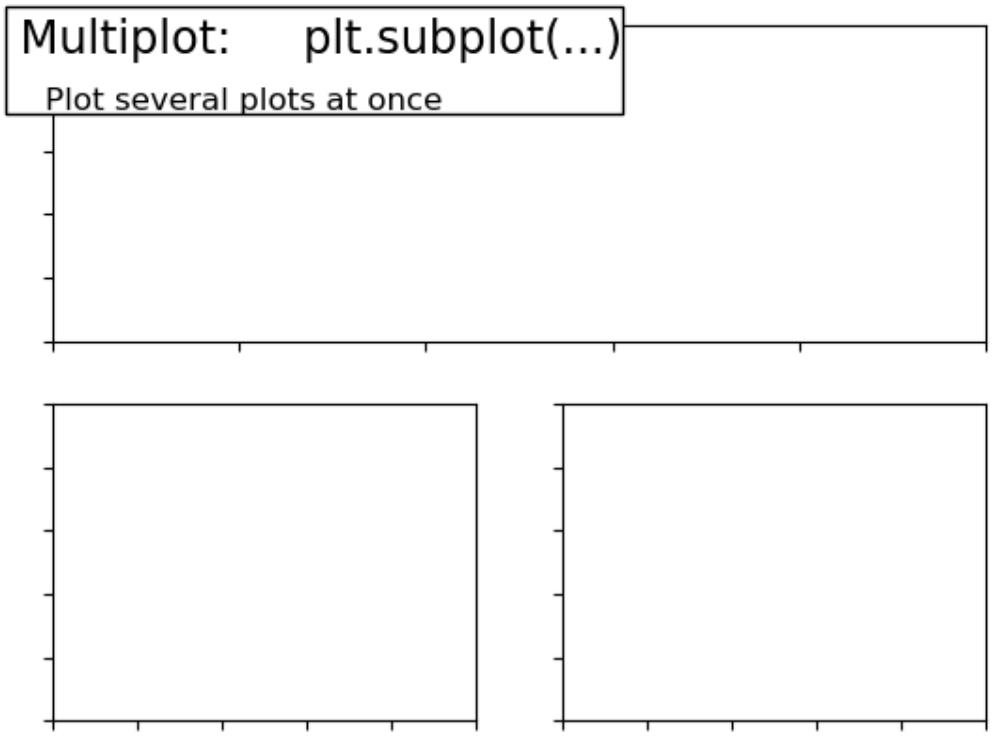

```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(0, 2, n)
Y = np.sin(2 * np.pi * X)

plt.plot (X, Y, lw=2, color='violet')
plt.xlim(-0.2, 2.2)
plt.xticks(())
plt.ylim(-1.2, 1.2)
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Regular Plot:      plt.plot(...)\n",
        horizontalalignment='left',
        verticalalignment='top',
        size='xx-large',
        transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n   Plot lines and/or markers ",
        horizontalalignment='left',
```

```
        verticalalignment='top',
        size='large',
        transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.051 seconds)

Multiple plots vignette

Demo multiple plots and style the figure.



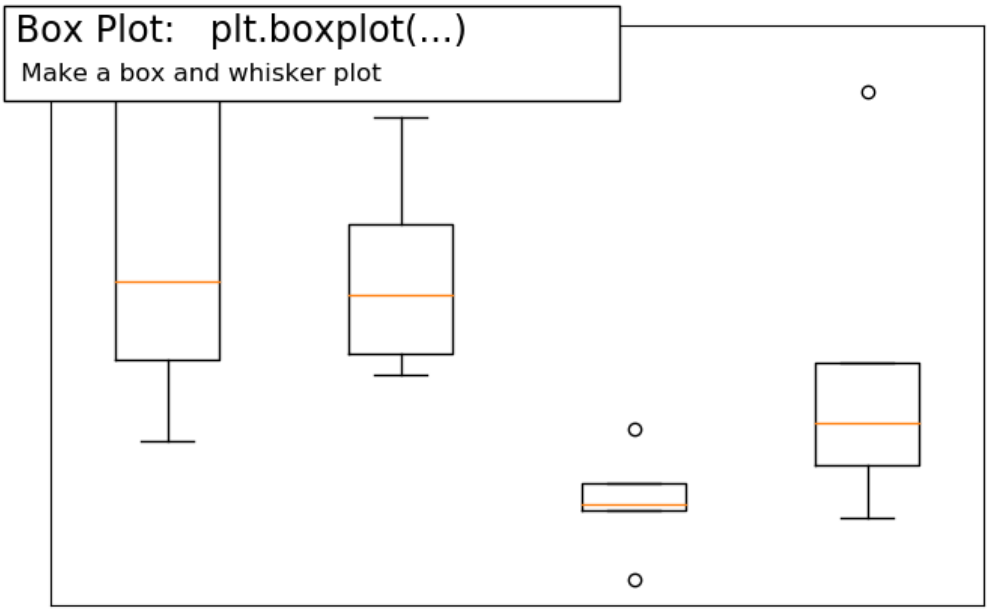

```
import matplotlib.pyplot as plt

ax = plt.subplot(2, 1, 1)
ax.set_xticklabels([])
ax.set_yticklabels([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .72),
                            width=.66, height=.34, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Multiplot:     plt.subplot(...)\n",
      horizontalalignment='left',
```

```
      verticalalignment='top',
      size='xx-large',
      transform=ax.transAxes)
plt.text(-0.05, 1.01, "\n\n    Plot several plots at once ",
      horizontalalignment='left',
      verticalalignment='top',
      size='large',
      transform=ax.transAxes)

ax = plt.subplot(2, 2, 3)
ax.set_xticklabels([])
ax.set_yticklabels([])

ax = plt.subplot(2, 2, 4)
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
```

Total running time of the script: (0 minutes 0.199 seconds)

Boxplot with matplotlib

An example of doing box plots with matplotlib



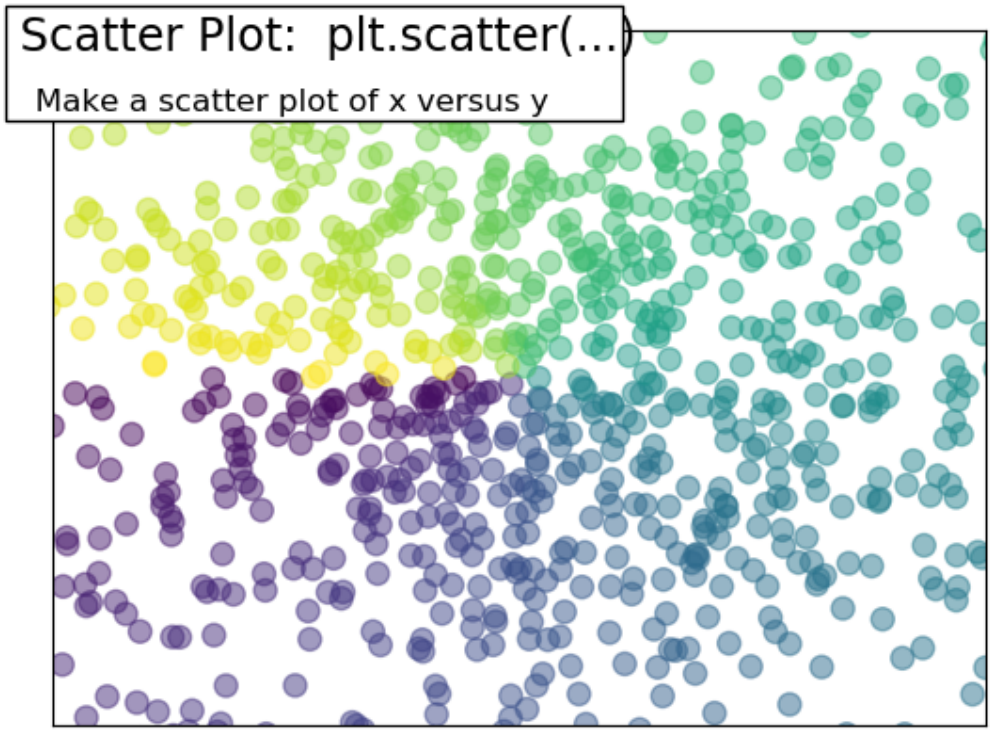

```
import numpy as np
import matplotlib.pyplot as plt


fig = plt.figure(figsize=(8, 5))
axes = plt.subplot(111)

n = 5
Z = np.zeros((n, 4))
X = np.linspace(0, 2, n, endpoint=True)
```

```
Y = np.random.random((n, 4))
plt.boxplot(Y)

plt.xticks(())
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Box Plot:   plt.boxplot(...)\n ",
        horizontalalignment='left',
        verticalalignment='top',
        size='xx-large',
        transform=axes.transAxes)

plt.text(-0.04, .98, "\n Make a box and whisker plot ",
        horizontalalignment='left',
        verticalalignment='top',
        size='large',
        transform=axes.transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.097 seconds)

Plot scatter decorated

An example showing the scatter function, with decorations.



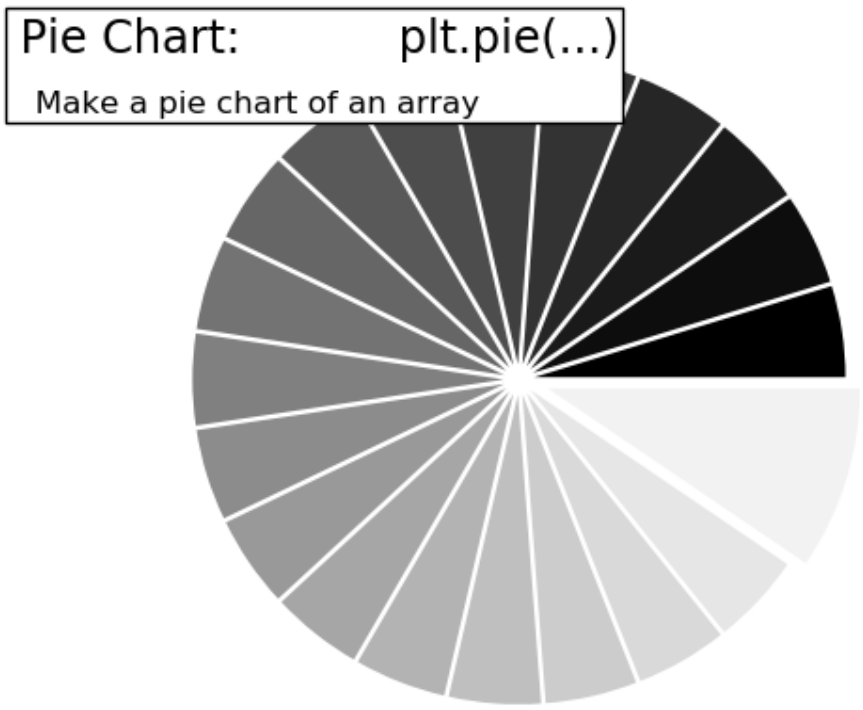

```
import numpy as np
import matplotlib.pyplot as plt

n = 1024
X = np.random.normal(0, 1, n)
Y = np.random.normal(0, 1, n)

T = np.arctan2(Y,X)

plt.scatter(X, Y, s=75, c=T, alpha=.5)
plt.xlim(-1.5, 1.5)
plt.xticks(())
plt.ylim(-1.5, 1.5)
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Scatter Plot:  plt.scatter(...)\n",
      horizontalalignment='left',
      verticalalignment='top',
      size='xx-large',
      transform=plt.gca().transAxes)
```

```
plt.text(-0.05, 1.01, "\n\n   Make a scatter plot of x versus y ",
      horizontalalignment='left',
      verticalalignment='top',
      size='large',
      transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.055 seconds)

Pie chart vignette

Demo pie chart with matplotlib and style the figure.



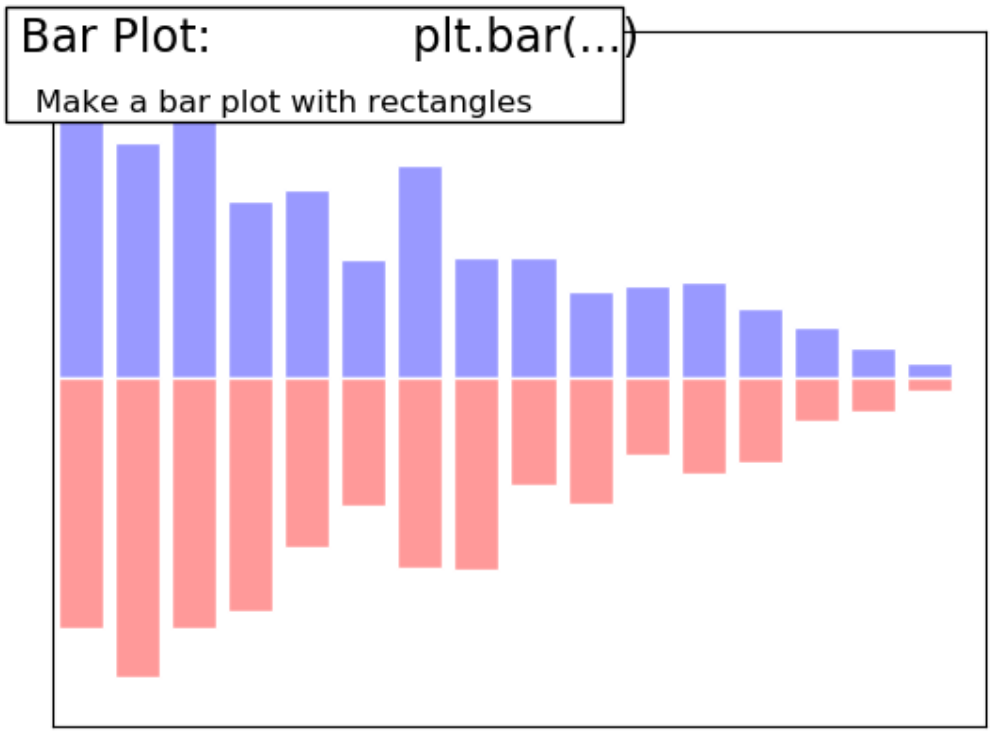

```
import numpy as np
import matplotlib.pyplot as plt

n = 20
X = np.ones(n)
X[-1] *= 2
plt.pie(X, explode=X*.05, colors = ['%f' % (i/float(n)) for i in range(n)])

fig = plt.gcf()
w, h = fig.get_figwidth(), fig.get_figheight()
r = h / float(w)

plt.xlim(-1.5, 1.5)
plt.ylim(-1.5 * r, 1.5 * r)
plt.xticks(())
plt.yticks(())
```

```
# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Pie Chart:           plt.pie(...)\n",
        horizontalalignment='left',
        verticalalignment='top',
        size='xx-large',
        transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n   Make a pie chart of an array ",
        horizontalalignment='left',
        verticalalignment='top',
        size='large',
        transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.131 seconds)

Bar plot advanced

An more elaborate bar plot example

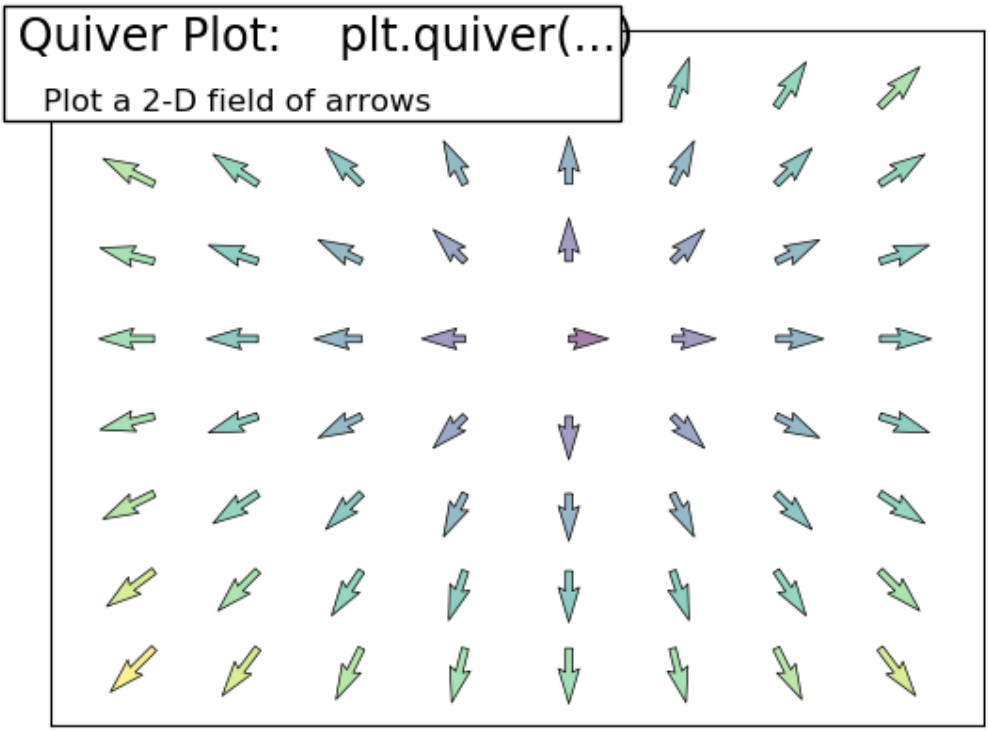


```
import numpy as np
import matplotlib.pyplot as plt

n = 16
X = np.arange(n)
Y1 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
Y2 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
plt.bar(X, Y1, facecolor='#9999ff', edgecolor='white')
plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')
plt.xlim(-.5, n)
plt.xticks(())
plt.ylim(-1, 1)
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Bar Plot:              plt.bar(...)\n",
      horizontalalignment='left',
      verticalalignment='top',
      size='xx-large',
      transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n   Make a bar plot with rectangles ",
      horizontalalignment='left',
      verticalalignment='top',
      size='large',
      transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.082 seconds)

Plotting quiver decorated

An example showing quiver with decorations.



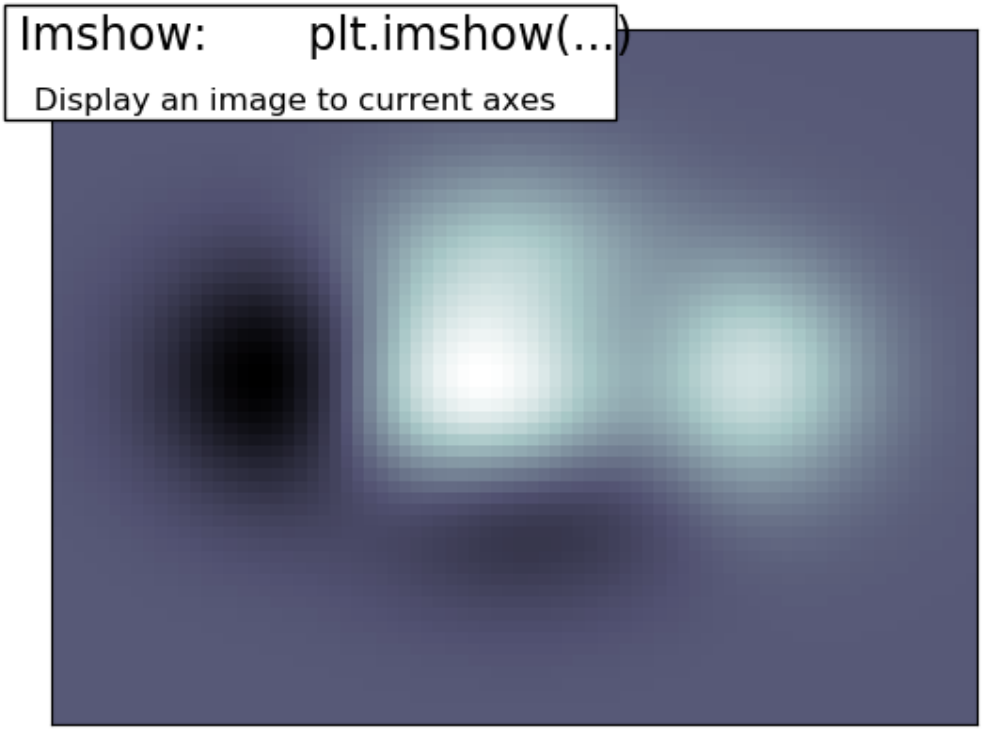

```
import numpy as np
import matplotlib.pyplot as plt

n = 8
X, Y = np.mgrid[0:n, 0:n]
T = np.arctan2(Y - n/ 2., X - n / 2.)
R = 10 + np.sqrt((Y - n / 2.) ** 2 + (X - n / 2.) ** 2)
U, V = R * np.cos(T), R * np.sin(T)

plt.quiver(X, Y, U, V, R, alpha=.5)
plt.quiver(X, Y, U, V, edgecolor='k', facecolor='None', linewidth=.5)

plt.xlim(-1, n)
plt.xticks(())
plt.ylim(-1, n)
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Quiver Plot:    plt.quiver(...)\n",
      horizontalalignment='left',
      verticalalignment='top',
      size='xx-large',
```

```
      transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n    Plot a 2-D field of arrows ",
      horizontalalignment='left',
      verticalalignment='top',
      size='large',
      transform=plt.gca().transAxes)


plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

Imshow demo

Demoing imshow



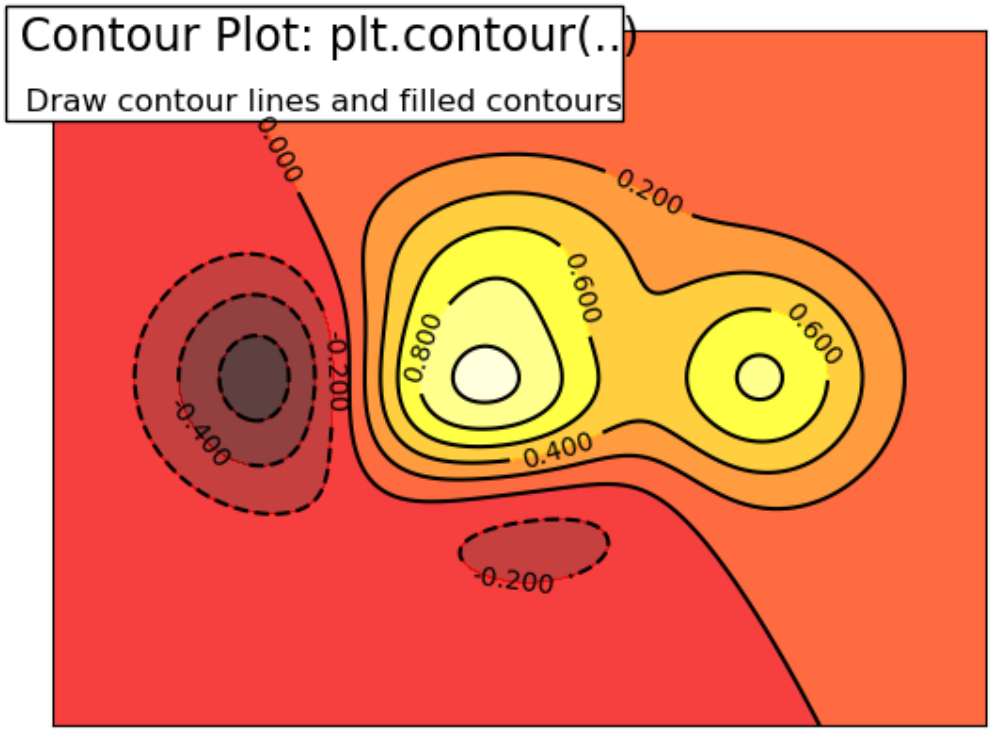

```
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)

n = 10
x = np.linspace(-3, 3, 8 * n)
y = np.linspace(-3, 3, 6 * n)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.imshow(Z, interpolation='nearest', cmap='bone', origin='lower')
plt.xticks(())
```

```
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Imshow:       plt.imshow(...)\n",
        horizontalalignment='left',
        verticalalignment='top',
        size='xx-large',
        transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n   Display an image to current axes ",
        horizontalalignment='left',
        verticalalignment='top',
        family='Lint McCree Intl BB',
        size='large',
        transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.053 seconds)

Display the contours of a function

An example demoing how to plot the contours of a function, with additional layout tweeks.



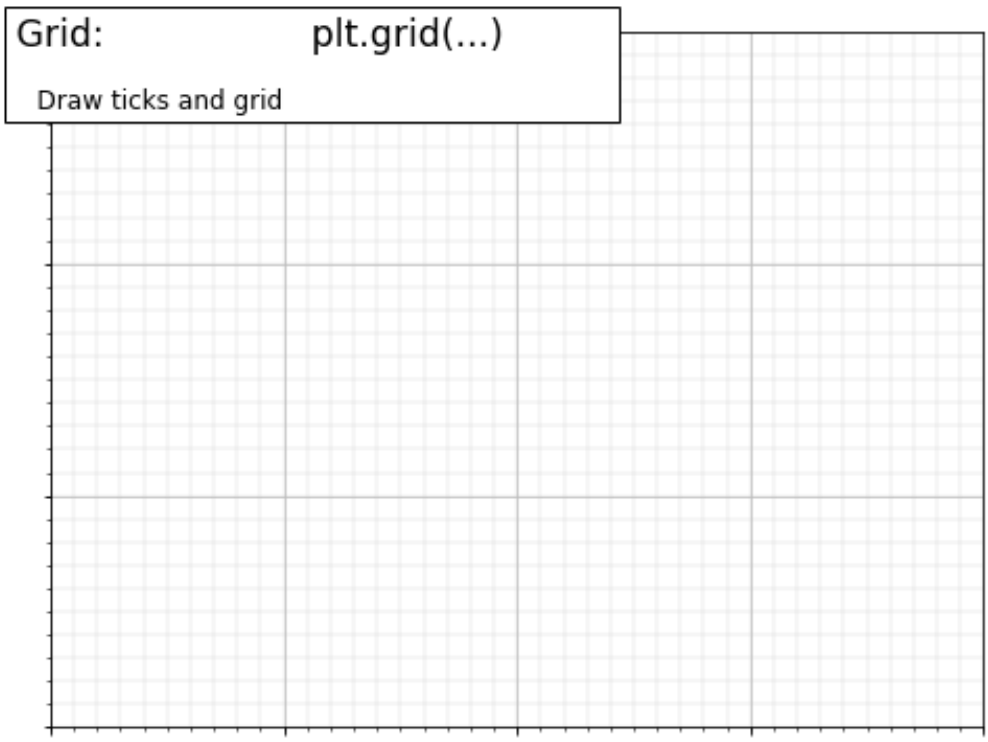

```
import numpy as np
import matplotlib.pyplot as plt

def f(x,y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)

n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)

plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap=plt.cm.hot)
C = plt.contour(X, Y, f(X,Y), 8, colors='black', linewidth=.5)
plt.clabel(C, inline=1, fontsize=10)
plt.xticks(())
plt.yticks(())


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Contour Plot: plt.contour(..)\n",
      horizontalalignment='left',
      verticalalignment='top',
      size='xx-large',
```

```
      transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n  Draw contour lines and filled contours ",
      horizontalalignment='left',
      verticalalignment='top',
      size='large',
      transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.137 seconds)

Grid elaborate

An example displaying a grid on the axes and tweaking the layout.



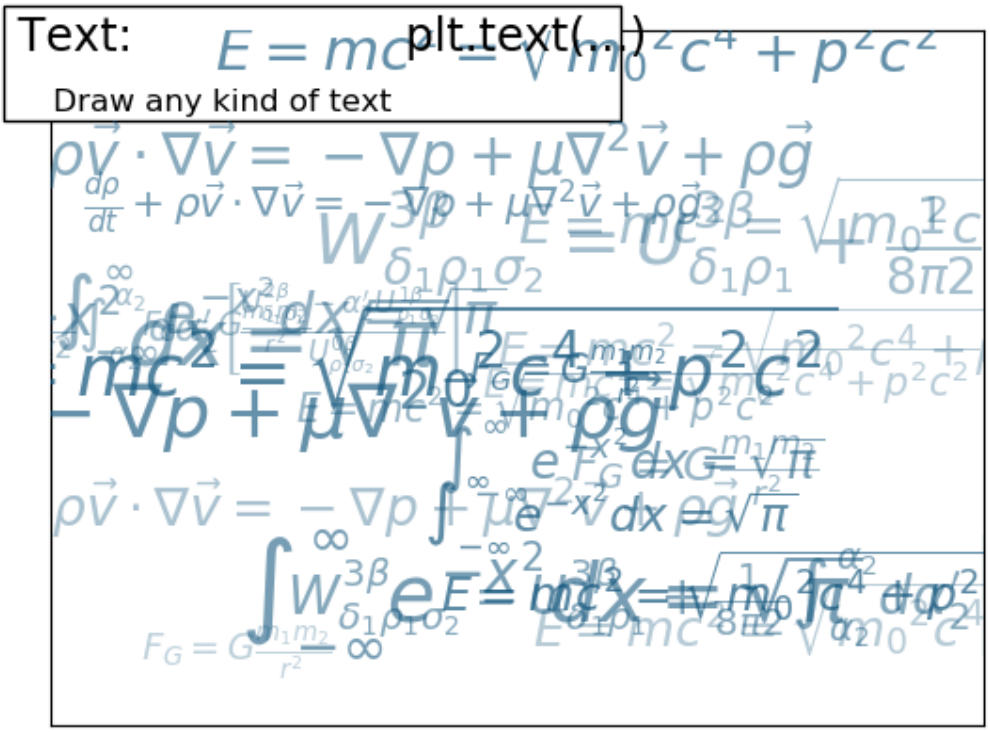

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator

fig = plt.figure(figsize=(8, 6), dpi=72, facecolor="white")
axes = plt.subplot(111)
axes.set_xlim(0, 4)
axes.set_ylim(0, 3)

axes.xaxis.set_major_locator(MultipleLocator(1.0))
axes.xaxis.set_minor_locator(MultipleLocator(0.1))
axes.yaxis.set_major_locator(MultipleLocator(1.0))
axes.yaxis.set_minor_locator(MultipleLocator(0.1))
axes.grid(which='major', axis='x', linewidth=0.75, linestyle='-', color='0.75')
axes.grid(which='minor', axis='x', linewidth=0.25, linestyle='-', color='0.75')
```

```
axes.grid(which='major', axis='y', linewidth=0.75, linestyle='-', color='0.75')
axes.grid(which='minor', axis='y', linewidth=0.25, linestyle='-', color='0.75')
axes.set_xticklabels([])
axes.set_yticklabels([])


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Grid:                    plt.grid(...)\n",
          horizontalalignment='left',
          verticalalignment='top',
          size='xx-large',
          transform=axes.transAxes)

plt.text(-0.05, 1.01, "\n\n    Draw ticks and grid ",
          horizontalalignment='left',
          verticalalignment='top',
          size='large',
          transform=axes.transAxes)
```

Total running time of the script: (0 minutes 0.066 seconds)

Text printing decorated

An example showing text printing and decorating the resulting figure.




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

fig = plt.figure()
plt.xticks(())
plt.yticks(())

eqs = []
eqs.append((r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \frac{1}
↪{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ U^{2\beta}_{\delta_1␣
↪\rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]$
↪"))
eqs.append((r"$\frac{d\rho}{d t} + \rho \vec{v}\cdot\nabla\vec{v} = -\nabla p + \mu\nabla^2␣
↪\vec{v} + \rho \vec{g}$"))
eqs.append((r"$\int_{-\infty}^\infty e^{-x^2}dx=\sqrt{\pi}$"))
eqs.append((r"$E = mc^2 = \sqrt{{m_0}^2c^4 + p^2c^2}$"))
eqs.append((r"$F_G = G\frac{m_1m_2}{r^2}$"))

for i in range(24):
    index = np.random.randint(0,len(eqs))
    eq = eqs[index]
    size = np.random.uniform(12,32)
    x,y = np.random.uniform(0,1,2)
    alpha = np.random.uniform(0.25,.75)
    plt.text(x, y, eq, ha='center', va='center', color="#11557c", alpha=alpha,
         transform=plt.gca().transAxes, fontsize=size, clip_on=True)


# Add a title and a box around it
from matplotlib.patches import FancyBboxPatch
ax = plt.gca()
```

```
ax.add_patch(FancyBboxPatch((-0.05, .87),
                            width=.66, height=.165, clip_on=False,
                            boxstyle="square,pad=0", zorder=3,
                            facecolor='white', alpha=1.0,
                            transform=plt.gca().transAxes))

plt.text(-0.05, 1.02, " Text:                    plt.text(...)\n",
        horizontalalignment='left',
        verticalalignment='top',
        size='xx-large',
        transform=plt.gca().transAxes)

plt.text(-0.05, 1.01, "\n\n     Draw any kind of text ",
        horizontalalignment='left',
        verticalalignment='top',
        size='large',
        transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (0 minutes 0.117 seconds)

CHAPTER 3

Scipy : high-level scientific computing

Authors: *Gaël Varoquaux, Adrien Chauve, Andre Espaze, Emmanuelle Gouillart, Ralf Gommers*

Scipy

The `scipy` package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

Tip: `scipy` can be compared to other standard scientific-computing libraries, such as the GSL (GNU Scientific Library for C and C++), or Matlab's toolboxes. `scipy` is the core package for scientific routines in Python; it is meant to operate efficiently on `numpy` arrays, so that numpy and scipy work hand in hand.

Before implementing a routine, it is worth checking if the desired data processing is not already implemented in Scipy. As non-professional programmers, scientists often tend to **re-invent the wheel**, which leads to buggy, non-optimal, difficult-to-share and unmaintainable code. By contrast, `Scipy`'s routines are optimized and tested, and should therefore be used when possible.

Chapters contents

Warning: This tutorial is far from an introduction to numerical computing. As enumerating the different submodules and functions in scipy would be very boring, we concentrate instead on a few examples to give a general idea of how to use `scipy` for scientific computing.

`scipy` is composed of task-specific sub-modules:

`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fftpack`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

Tip: They all depend on `numpy`, but are mostly independent of each other. The standard way of importing Numpy and these Scipy modules is:

```
>>> import numpy as np
>>> from scipy import stats  # same for other sub-modules
```

The main `scipy` namespace mostly contains functions that are really numpy functions (try `scipy.cos is np.cos`). Those are exposed for historical reasons; there's no reason to use `import scipy` in your code.

3.1 File input/output: `scipy.io`

Matlab files: Loading and saving:

```
>>> from scipy import io as spio
>>> a = np.ones((3, 3))
>>> spio.savemat('file.mat', {'a': a}) # savemat expects a dictionary
>>> data = spio.loadmat('file.mat')
```

```
>>> data['a']
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Warning: Python / Matlab mismatches, *eg* matlab does not represent 1D arrays

```
>>> a = np.ones(3)
>>> a
array([ 1.,  1.,  1.])
>>> spio.savemat('file.mat', {'a': a})
>>> spio.loadmat('file.mat')['a']
array([[ 1.,  1.,  1.]])
```

Notice the difference?

Image files: Reading images:

```
>>> from scipy import misc
>>> misc.imread('fname.png')
array(...)
>>> # Matplotlib also has a similar function
>>> import matplotlib.pyplot as plt
>>> plt.imread('fname.png')
array(...)
```

See also:

- Load text files: `numpy.loadtxt()`/`numpy.savetxt()`
- Clever loading of text/csv files: `numpy.genfromtxt()`/`numpy.recfromcsv()`
- Fast and efficient, but numpy-specific, binary format: `numpy.save()`/`numpy.load()`
- More advanced input/output of images in scikit-image: `skimage.io`

3.2 Special functions: `scipy.special`

Special functions are transcendental functions. The docstring of the `scipy.special` module is well-written, so we won't list all functions here. Frequently used ones are:

- Bessel function, such as `scipy.special.jn()` (nth integer order Bessel function)
- Elliptic function (`scipy.special.ellipj()` for the Jacobian elliptic function, ...)
- Gamma function: `scipy.special.gamma()`, also note `scipy.special.gammaln()` which will give the log of Gamma to a higher numerical precision.
- Erf, the area under a Gaussian curve: `scipy.special.erf()`

3.3 Linear algebra operations: `scipy.linalg`

Tip: The `scipy.linalg` module provides standard linear algebra operations, relying on an underlying efficient implementation (BLAS, LAPACK).

- The `scipy.linalg.det()` function computes the determinant of a square matrix:

```
>>> from scipy import linalg
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> linalg.det(arr)
-2.0
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.det(arr)
0.0
>>> linalg.det(np.ones((3, 4)))
Traceback (most recent call last):
...
ValueError: expected square matrix
```

- The `scipy.linalg.inv()` function computes the inverse of a square matrix:

```
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> iarr = linalg.inv(arr)
>>> iarr
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.allclose(np.dot(arr, iarr), np.eye(2))
True
```

Finally computing the inverse of a singular matrix (its determinant is zero) will raise `LinAlgError`:

```
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.inv(arr)
Traceback (most recent call last):
...
...LinAlgError: singular matrix
```

- More advanced operations are available, for example singular-value decomposition (SVD):

```
>>> arr = np.arange(9).reshape((3, 3)) + np.diag([1, 0, 1])
>>> uarr, spec, vharr = linalg.svd(arr)
```

The resulting array spectrum is:

```
>>> spec
array([ 14.88982544,   0.45294236,   0.29654967])
```

The original matrix can be re-composed by matrix multiplication of the outputs of `svd` with `np.dot`:

```
>>> sarr = np.diag(spec)
>>> svd_mat = uarr.dot(sarr).dot(vharr)
>>> np.allclose(svd_mat, arr)
True
```

SVD is commonly used in statistics and signal processing. Many other standard decompositions (QR, LU, Cholesky, Schur), as well as solvers for linear systems, are available in `scipy.linalg`.

3.4 Interpolation: `scipy.interpolate`

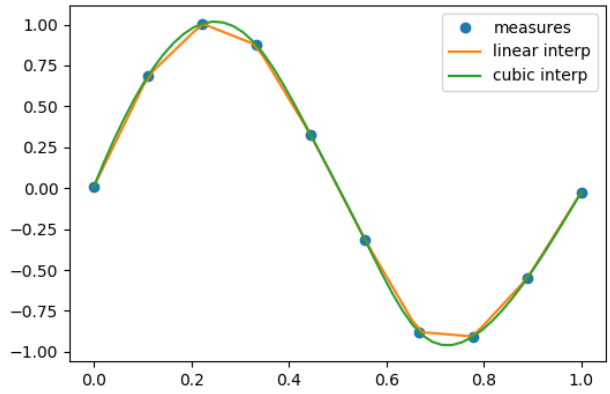
`scipy.interpolate` is useful for fitting a function from experimental data and thus evaluating points where no measure exists. The module is based on the FITPACK Fortran subroutines.

By imagining experimental data close to a sine function:

```
>>> measured_time = np.linspace(0, 1, 10)
>>> noise = (np.random.random(10)*2 - 1) * 1e-1
>>> measures = np.sin(2 * np.pi * measured_time) + noise
```

`scipy.interpolate.interp1d` can build a linear interpolation function:

```
>>> from scipy.interpolate import interp1d
>>> linear_interp = interp1d(measured_time, measures)
```



Then the result can be evaluated at the time of interest:

```
>>> interpolation_time = np.linspace(0, 1, 50)
>>> linear_results = linear_interp(interpolation_time)
```

A cubic interpolation can also be selected by providing the `kind` optional keyword argument:

```
>>> cubic_interp = interp1d(measured_time, measures, kind='cubic')
>>> cubic_results = cubic_interp(interpolation_time)
```

`scipy.interpolate.interp2d` is similar to `scipy.interpolate.interp1d`, but for 2-D arrays. Note that for the *interp* family, the interpolation points must stay within the range of given data points. See the summary exercise on *Maximum wind speed prediction at the Sprogø station* for a more advanced spline interpolation example.

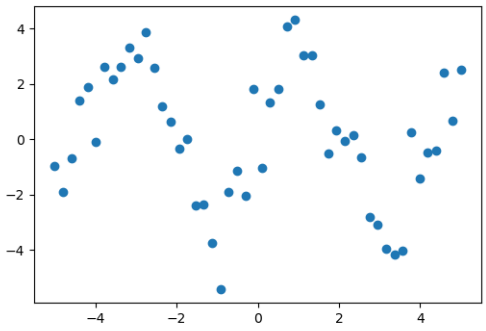
3.5 Optimization and fit: `scipy.optimize`

Optimization is the problem of finding a numerical solution to a minimization or equality.

Tip: The `scipy.optimize` module provides algorithms for function minimization (scalar or multi-dimensional), curve fitting and root finding.

```
>>> from scipy import optimize
```

3.5.1 Curve fitting

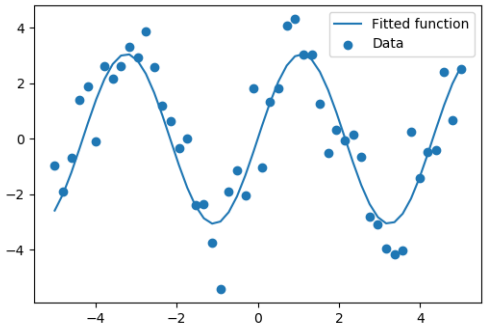


Suppose we have data on a sine wave, with some noise:

```
>>> x_data = np.linspace(-5, 5, num=50)
>>> y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=50)
```

If we know that the data lies on a sine wave, but not the amplitudes or the period, we can find those by least squares curve fitting. First we have to define the test function to fit, here a sine with unknown amplitude and period:

```
>>> def test_func(x, a, b):
...     return a * np.sin(b * x)
```



We then use `scipy.optimize.curve_fit()` to find a and b:

```
>>> params, params_covariance = optimize.curve_fit(test_func, x_data, y_data, p0=[2, 2])
>>> print(params)
[ 3.05931973  1.45754553]
```

Exercise: Curve fitting of temperature data

The temperature extremes in Alaska for each month, starting in January, are given by (in degrees Celcius):

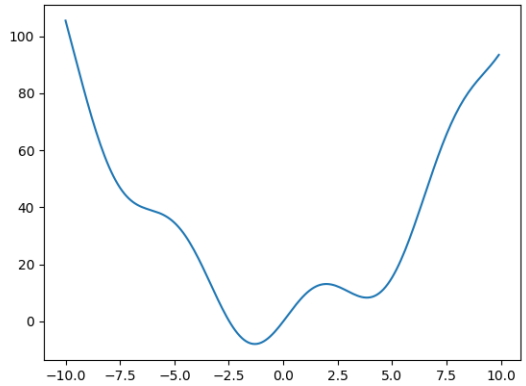
```
max:  17,  19,  21,  28,  33,  38, 37,  37,  31,  23,  19,  18
min: -62, -59, -56, -46, -32, -18, -9, -13, -25, -46, -52, -58
```

1. Plot these temperature extremes.
2. Define a function that can describe min and max temperatures. Hint: this function has to have a period of 1 year. Hint: include a time offset.
3. Fit this function to the data with `scipy.optimize.curve_fit()`.
4. Plot the result. Is the fit reasonable? If not, why?

5. Is the time offset for min and max temperatures the same within the fit accuracy?

solution

3.5.2 Finding the minimum of a scalar function



Let's define the following function:

```
>>> def f(x):
...     return x**2 + 10*np.sin(x)
```

and plot it:

```
>>> x = np.arange(-10, 10, 0.1)
>>> plt.plot(x, f(x))
>>> plt.show()
```

This function has a global minimum around -1.3 and a local minimum around 3.8.

Searching for minimum can be done with `scipy.optimize.minimize()`, given a starting point x0, it returns the location of the minimum that it has found:

result type

The result of `scipy.optimize.minimize()` is a compound object comprising all information on the convergence

```
>>> result = optimize.minimize(f, x0=0)
>>> result
      fun: -7.9458233756...
 hess_inv: array([[ 0.0858...]])
      jac: array([ -1.19209...e-06])
  message: 'Optimization terminated successfully.'
     nfev: 18
      nit: 5
     njev: 6
   status: 0
  success: True
        x: array([-1.30644...])
>>> result.x # The coordinate of the minimum
array([-1.30644...])
```

Methods: As the function is a smooth function, gradient-descent based methods are good options. The lBFGS algorithm is a good choice in general:

```
>>> optimize.minimize(f, x0=0, method="L-BFGS-B")
      fun: array([-7.94582338])
 hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
      jac: array([ -1.42108547e-06])
  message: ...'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 12
      nit: 5
   status: 0
  success: True
        x: array([-1.30644013])
```

Note how it cost only 12 functions evaluation above to find a good value for the minimum.

Global minimum: A possible issue with this approach is that, if the function has local minima, the algorithm may find these local minima instead of the global minimum depending on the initial point x0:

```
>>> res = optimize.minimize(f, x0=3, method="L-BFGS-B")
>>> res.x
array([ 3.83746709])
```

If we don't know the neighborhood of the global minimum to choose the initial point, we need to resort to costlier global optimization. To find the global minimum, we use `scipy.optimize.basinhopping()` (added in version 0.12.0 of Scipy). It combines a local optimizer with sampling of starting points:

```
>>> optimize.basinhopping(f, 0)
                    nfev: 1725
 minimization_failures: 0
                     fun: -7.9458233756152845
                       x: array([-1.30644001])
                 message: ['requested number of basinhopping iterations completed successfully']
                    njev: 575
                     nit: 100
```

Note: `scipy` used to contain the routine *anneal*, it has been removed in SciPy 0.16.0.

Constraints: We can constrain the variable to the interval `(0, 10)` using the "bounds" argument:

A list of bounds

As `minimize()` works in general with x multidimensionsal, the "bounds" argument is a list of bound on each dimension.

```
>>> res = optimize.minimize(f, x0=1,
...                         bounds=((0, 10), ))
>>> res.x
array([ 0.])
```

Tip: What has happened? Why are we finding 0, which is not a mimimum of our function.

Minimizing functions of several variables

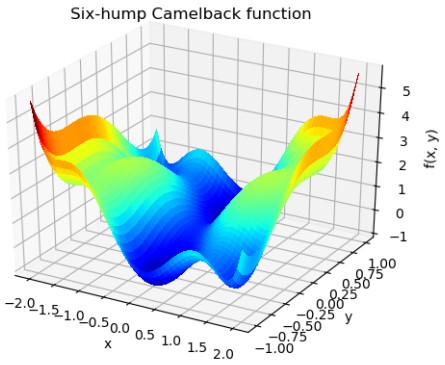
To minimize over several variables, the trick is to turn them into a function of a multi-dimensional variable (a vector). See for instance the exercise on 2D minimization below.

Note: `scipy.optimize.minimize_scalar()` is a function with dedicated methods to minimize functions of only one variable.

See also:

Finding minima of function is discussed in more details in the advanced chapter: *Mathematical optimization: finding minima of functions*.

Exercise: 2-D minimization



The six-hump camelback function

$$f(x,y) = (4 - 2.1x^2 + \frac{x^4}{3})x^2 + xy + (4y^2 - 4)y^2$$

has multiple global and local minima. Find the global minima of this function.

Hints:

- Variables can be restricted to $-2 < x < 2$ and $-1 < y < 1$.
- Use `numpy.meshgrid()` and `pylab.imshow()` to find visually the regions.
- Use `scipy.optimize.minimize()`, optionally trying out several of its 'methods'.

How many global minima are there, and what is the function value at those points? What happens for an initial guess of $(x, y) = (0, 0)$?

solution

3.5.3 Finding the roots of a scalar function

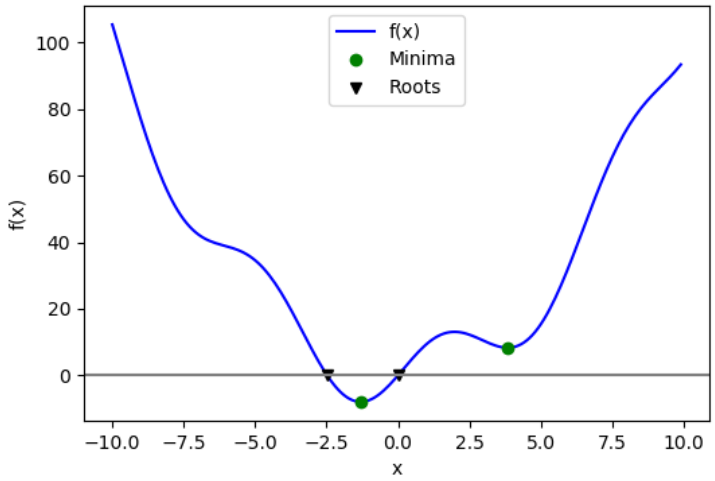
To find a root, i.e. a point where $f(x) = 0$, of the function f above we can use `scipy.optimize.root()`:

```
>>> root = optimize.root(f, x0=1)  # our initial guess is 1
>>> root    # The full result
    fjac: array([[-1.]])
     fun: array([ 0.])
 message: 'The solution converged.'
    nfev: 10
     qtf: array([  1.33310463e-32])
       r: array([-10.])
  status: 1
 success: True
       x: array([ 0.])
>>> root.x  # Only the root found
array([ 0.])
```

Note that only one root is found. Inspecting the plot of f reveals that there is a second root around -2.5. We find the exact value of it by adjusting our initial guess:

```
>>> root2 = optimize.root(f, x0=-2.5)
>>> root2.x
array([-2.47948183])
```

Note: `scipy.optimize.root()` also comes with a variety of algorithms, set via the "method" argument.



Now that we have found the minima and roots of `f` and used curve fitting on it, we put all those results together in a single plot:

See also:

You can find all algorithms and functions with similar functionalities in the documentation of `scipy.optimize`.

See the summary exercise on *Non linear least squares curve fitting: application to point extraction in topographical lidar data* for another, more advanced example.

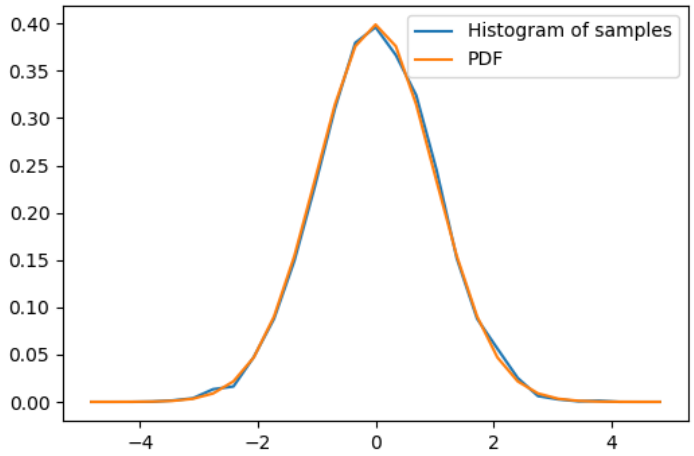
3.6 Statistics and random numbers: `scipy.stats`

The module `scipy.stats` contains statistical tools and probabilistic descriptions of random processes. Random number generators for various random process can be found in `numpy.random`.

3.6.1 Distributions: histogram and probability density function

Given observations of a random process, their histogram is an estimator of the random process's PDF (probability density function):

```
>>> samples = np.random.normal(size=1000)
>>> bins = np.arange(-4, 5)
>>> bins
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> histogram = np.histogram(samples, bins=bins, normed=True)[0]
>>> bins = 0.5*(bins[1:] + bins[:-1])
>>> bins
array([-3.5, -2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5])
>>> from scipy import stats
>>> pdf = stats.norm.pdf(bins)  # norm is a distribution object

>>> plt.plot(bins, histogram)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(bins, pdf)
[<matplotlib.lines.Line2D object at ...>]
```



The distribution objects

`scipy.stats.norm` is a distribution object: each distribution in `scipy.stats` is represented as an object. Here it's the normal distribution, and it comes with a PDF, a CDF, and much more.

If we know that the random process belongs to a given family of random processes, such as normal processes, we can do a maximum-likelihood fit of the observations to estimate the parameters of the underlying distribution. Here we fit a normal process to the observed data:

```
>>> loc, std = stats.norm.fit(samples)
>>> loc
-0.045256707...
>>> std
0.9870331586...
```

Exercise: Probability distributions

Generate 1000 random variates from a gamma distribution with a shape parameter of 1, then plot a histogram from those samples. Can you plot the pdf on top (it should match)?

Extra: the distributions have many useful methods. Explore them by reading the docstring or by using tab completion. Can you recover the shape parameter 1 by using the `fit` method on your random variates?

3.6.2 Mean, median and percentiles

The mean is an estimator of the center of the distribution:

```
>>> np.mean(samples)
-0.0452567074...
```

The median another estimator of the center. It is the value with half of the observations below, and half above:

```
>>> np.median(samples)
-0.0580280347...
```

Tip: Unlike the mean, the median is not sensitive to the tails of the distribution. It is "robust".

Exercise: Compare mean and median on samples of a Gamma distribution

Which one seems to be the best estimator of the center for the Gamma distribution?

The median is also the percentile 50, because 50% of the observation are below it:

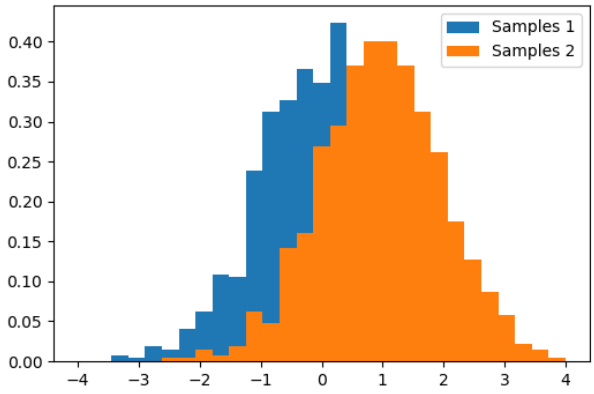
```
>>> stats.scoreatpercentile(samples, 50)
-0.0580280347...
```

Similarly, we can calculate the percentile 90:

```
>>> stats.scoreatpercentile(samples, 90)
1.2315935511...
```

Tip: The percentile is an estimator of the CDF: cumulative distribution function.

3.6.3 Statistical tests



A statistical test is a decision indicator. For instance, if we have two sets of observations, that we assume are generated from Gaussian processes, we can use a T-test to decide whether the means of two sets of observations are significantly different:

```
>>> a = np.random.normal(0, 1, size=100)
>>> b = np.random.normal(1, 1, size=10)
>>> stats.ttest_ind(a, b)
(array(-3.177574054...), 0.0019370639...)
```

Tip: The resulting output is composed of:

- The T statistic value: it is a number the sign of which is proportional to the difference between the two random processes and the magnitude is related to the significance of this difference.
- the *p value*: the probability of both processes being identical. If it is close to 1, the two process are almost certainly identical. The closer it is to zero, the more likely it is that the processes have different means.

See also:

The chapter on *statistics* introduces much more elaborate tools for statistical testing and statistical data loading and visualization outside of scipy.

3.7 Numerical integration: `scipy.integrate`

3.7.1 Function integrals

The most generic integration routine is `scipy.integrate.quad()`. To compute $\int_0^{\pi/2} sin(t)dt$:

```
>>> from scipy.integrate import quad
>>> res, err = quad(np.sin, 0, np.pi/2)
>>> np.allclose(res, 1)   # res is the result, is should be close to 1
True
>>> np.allclose(err, 1 - res)  # err is an estimate of the err
True
```

Other integration schemes are available: `scipy.integrate.fixed_quad()`, `scipy.integrate.quadrature()`, `scipy.integrate.romberg()`...

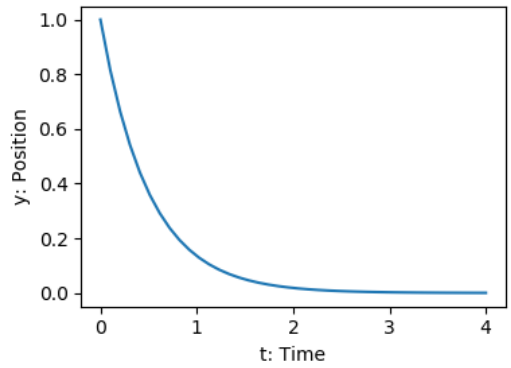
3.7.2 Integrating differential equations

`scipy.integrate` also features routines for integrating Ordinary Differential Equations (ODE). In particular, `scipy.integrate.odeint()` solves ODE of the form:

```
dy/dt = rhs(y1, y2, .., t0,...)
```

As an introduction, let us solve the ODE $\frac{dy}{dt} = -2y$ between $t = 0...4$, with the initial condition $y(t = 0) = 1$. First the function computing the derivative of the position needs to be defined:

```
>>> def calc_derivative(ypos, time):
...     return -2 * ypos
```



Then, to compute y as a function of time:

```
>>> from scipy.integrate import odeint
>>> time_vec = np.linspace(0, 4, 40)
>>> y = odeint(calc_derivative, y0=1, t=time_vec)
```

Let us integrate a more complex ODE: a damped spring-mass oscillator. The position of a mass attached to a spring obeys the 2nd order *ODE* $y'' + 2\varepsilon\omega_0 y' + \omega_0^2 y = 0$ with $\omega_0^2 = k/m$ with k the spring constant, m the mass and $\varepsilon = c/(2m\omega_0)$ with c the damping coefficient. We set:

```
>>> mass = 0.5  # kg
>>> kspring = 4  # N/m
>>> cviscous = 0.4  # N s/m
```

Hence:

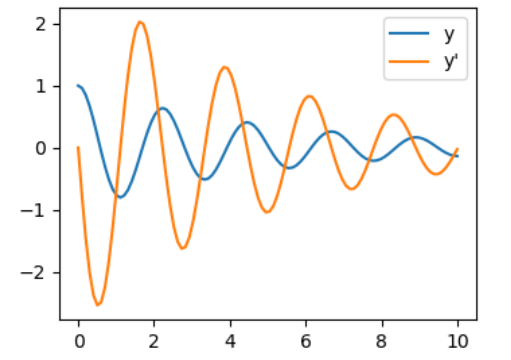
```
>>> eps = cviscous / (2 * mass * np.sqrt(kspring/mass))
>>> omega = np.sqrt(kspring / mass)
```

The system is underdamped, as:

```
>>> eps < 1
True
```

For `odeint()`, the 2nd order equation needs to be transformed in a system of two first-order equations for the vector $Y = (y, y')$: the function computes the velocity and acceleration:

```
>>> def calc_deri(yvec, time, eps, omega):
...     return (yvec[1], -eps * omega * yvec[1] - omega **2 * yvec[0])
```



Integration of the system follows:

```
>>> time_vec = np.linspace(0, 10, 100)
>>> yinit = (1, 0)
>>> yarr = odeint(calc_deri, yinit, time_vec, args=(eps, omega))
```

Tip: `scipy.integrate.odeint()` uses the LSODA (Livermore Solver for Ordinary Differential equations with Automatic method switching for stiff and non-stiff problems), see the ODEPACK Fortran library for more details.

See also:

Partial Differental Equations

There is no Partial Differential Equations (PDE) solver in Scipy. Some Python packages for solving PDE's are available, such as fipy or SfePy.

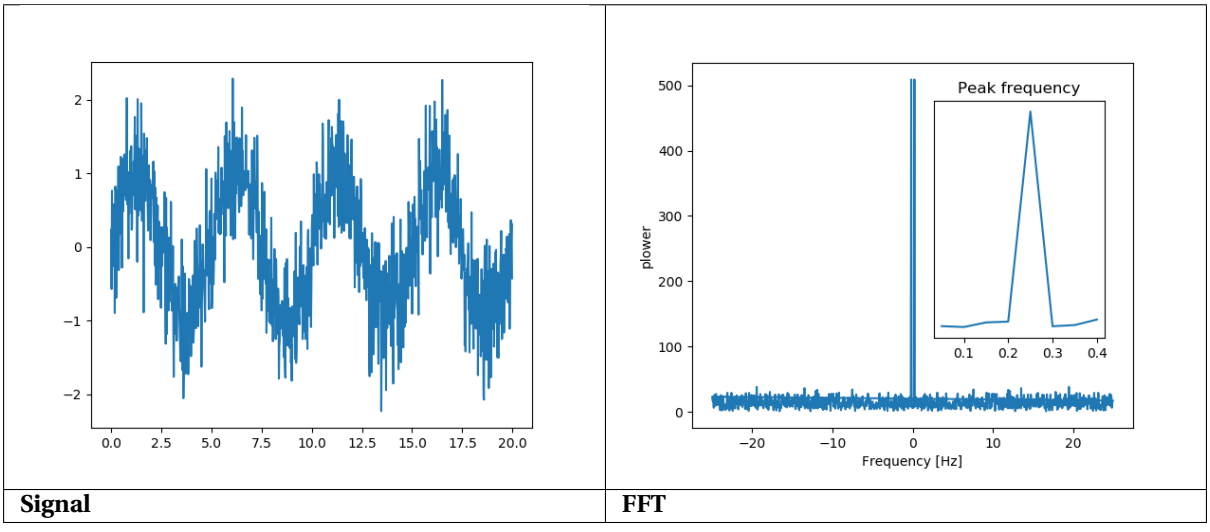
3.8 Fast Fourier transforms: `scipy.fftpack`

The `scipy.fftpack` module computes fast Fourier transforms (FFTs) and offers utilities to handle them. The main functions are:

- `scipy.fftpack.fft()` to compute the FFT
- `scipy.fftpack.fftfreq()` to generate the sampling frequencies
- `scipy.fftpack.ifft()` computes the inverse FFT, from frequency space to signal space

As an illustration, a (noisy) input signal (`sig`), and its FFT:

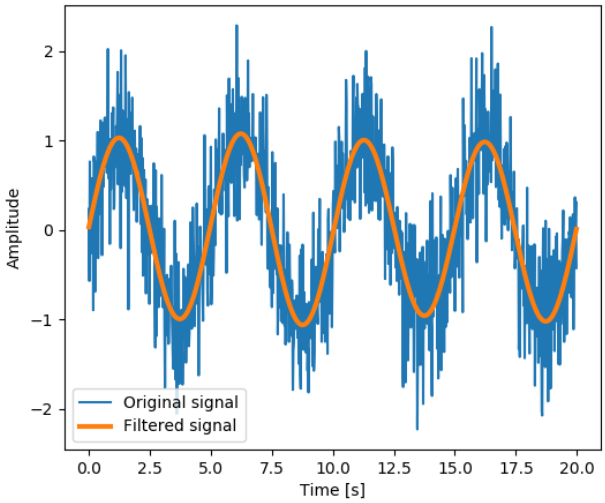
```
>>> from scipy import fftpack
>>> sig_fft = fftpack.fft(sig)
>>> freqs = fftpack.fftfreq(sig.size, d=time_step)
```



Signal	FFT

As the signal comes from a real function, the Fourier transform is symmetric.

The peak signal frequency can be found with `freqs[power.argmax()]`



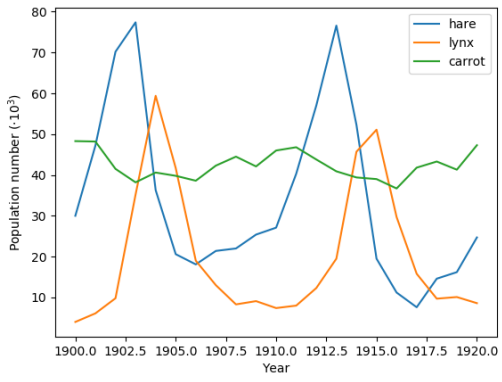
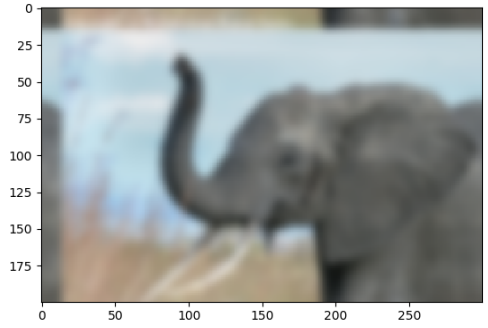
Setting the Fourrier component above this frequency to zero and inverting the FFT with `scipy.fftpack.ifft()`, gives a filtered signal.

Note: The code of this example can be found *here*

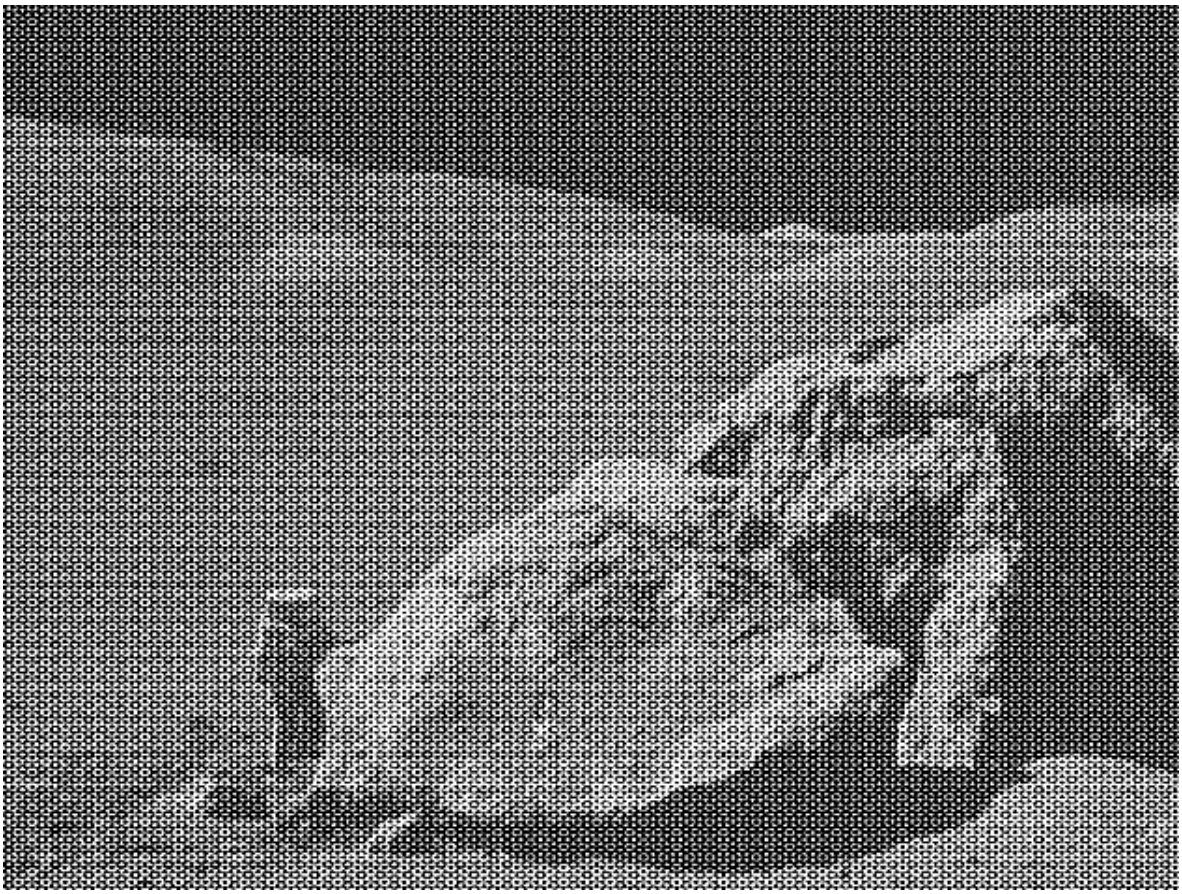
numpy.fft

Numpy also has an implementation of FFT (`numpy.fft`). However, the scipy one should be preferred, as it uses more efficient underlying implementations.

Fully worked examples:

Crude periodicity finding (*link*)	Gaussian image blur (*link*)
	

Exercise: Denoise moon landing image

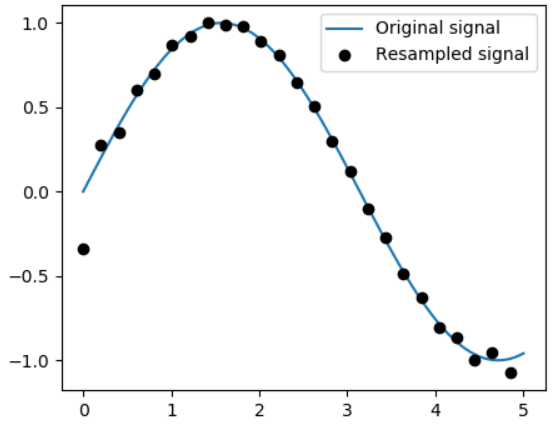


1. Examine the provided image `moonlanding.png`, which is heavily contaminated with periodic noise. In this exercise, we aim to clean up the noise using the Fast Fourier Transform.
2. Load the image using `pylab.imread()`.
3. Find and use the 2-D FFT function in `scipy.fftpack`, and plot the spectrum (Fourier transform of) the image. Do you have any trouble visualising the spectrum? If so, why?
4. The spectrum consists of high and low frequency components. The noise is contained in the high-frequency part of the spectrum, so set some of those components to zero (use array slicing).
5. Apply the inverse Fourier transform to see the resulting image.

Solution

3.9 Signal processing: `scipy.signal`

Tip: `scipy.signal` is for typical signal processing: 1D, regularly-sampled signals.

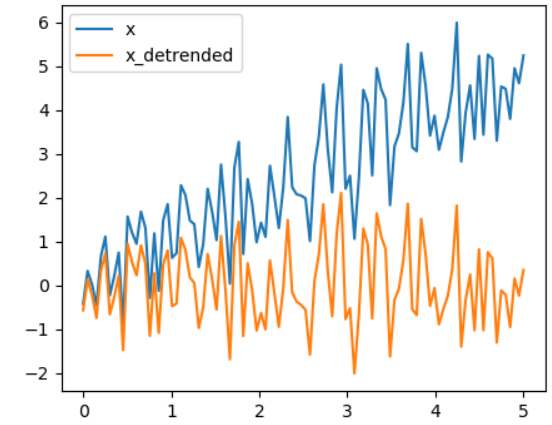


Resampling `scipy.signal.resample()`: resample a signal to n points using FFT.

```
>>> t = np.linspace(0, 5, 100)
>>> x = np.sin(t)

>>> from scipy import signal
>>> x_resampled = signal.resample(x, 25)

>>> plt.plot(t, x)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t[::4], x_resampled, 'ko')
[<matplotlib.lines.Line2D object at ...>]
```

Tip: Notice how on the side of the window the resampling is less accurate and has a rippling effect.

This resampling is different from the *interpolation* provided by `scipy.interpolate` as it only applies to regularly sampled data.



Detrending `scipy.signal.detrend()`: remove linear trend from signal:

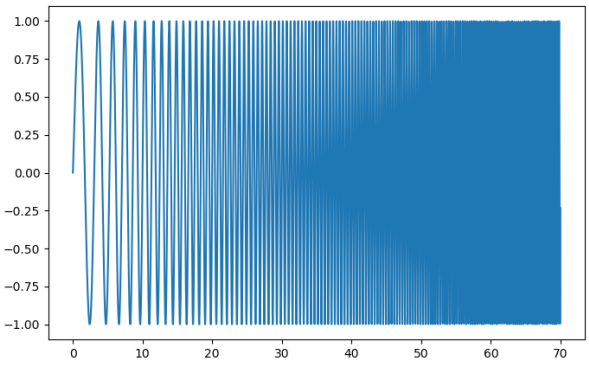
```
>>> t = np.linspace(0, 5, 100)
>>> x = t + np.random.normal(size=100)

>>> from scipy import signal
>>> x_detrended = signal.detrend(x)

>>> plt.plot(t, x)
```

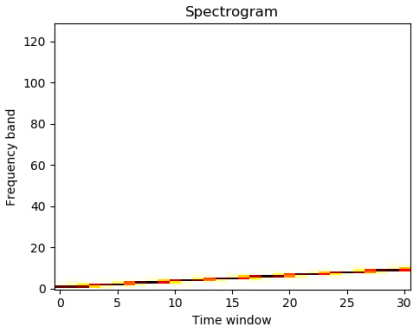
```
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t, x_detrended)
[<matplotlib.lines.Line2D object at ...>]
```

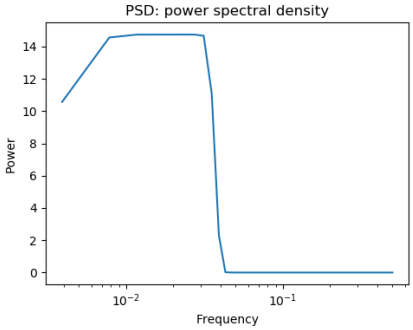
Filtering: For non-linear filtering, `scipy.signal` has filtering (median filter `scipy.signal.medfilt()`, Wiener `scipy.signal.wiener()`), but we will discuss this in the image section.

Tip: `scipy.signal` also has a full-blown set of tools for the design of linear filter (finite and infinite response filters), but this is out of the scope of this tutorial.

Spectral analysis: `scipy.signal.spectrogram()` compute a spectrogram –frequency spectrums over consecutive time windows–, while `scipy.signal.welch()` comptes a power spectrum density (PSD).







3.10 Image manipulation: `scipy.ndimage`

`scipy.ndimage` provides manipulation of n-dimensional arrays as images.

3.10.1 Geometrical transformations on images

Changing orientation, resolution, ..

```
>>> from scipy import misc  # Load an image
>>> face = misc.face(gray=True)

>>> from scipy import ndimage # Shift, roate and zoom it
>>> shifted_face = ndimage.shift(face, (50, 50))
>>> shifted_face2 = ndimage.shift(face, (50, 50), mode='nearest')
>>> rotated_face = ndimage.rotate(face, 30)
>>> cropped_face = face[50:-50, 50:-50]
>>> zoomed_face = ndimage.zoom(face, 2)
```

```
>>> zoomed_face.shape
(1536, 2048)
```

```
>>> plt.subplot(151)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>

>>> plt.imshow(shifted_face, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>

>>> plt.axis('off')
(-0.5, 1023.5, 767.5, -0.5)

>>> # etc.
```

3.10.2 Image filtering

Generate a noisy face:

```
>>> from scipy import misc
>>> face = misc.face(gray=True)
>>> face = face[:512, -512:]  # crop out square on right
>>> import numpy as np
>>> noisy_face = np.copy(face).astype(np.float)
>>> noisy_face += face.std() * 0.5 * np.random.standard_normal(face.shape)
```

Apply a variety of filters on it:

```
>>> blurred_face = ndimage.gaussian_filter(noisy_face, sigma=3)
>>> median_face = ndimage.median_filter(noisy_face, size=5)
>>> from scipy import signal
>>> wiener_face = signal.wiener(noisy_face, (5, 5))
```

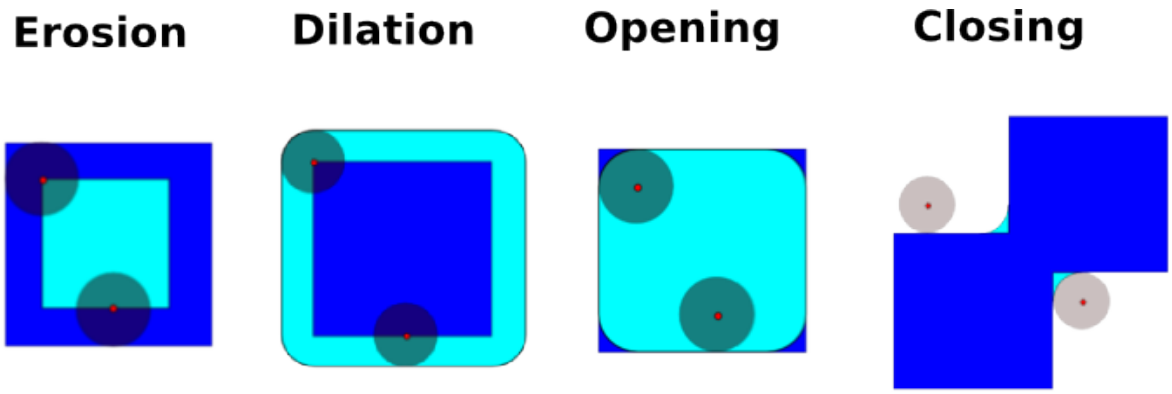
Other filters in `scipy.ndimage.filters` and `scipy.signal` can be applied to images.

Exercise

Compare histograms for the different filtered images.

3.10.3 Mathematical morphology

Tip: Mathematical morphology stems from set theory. It characterizes and transforms geometrical structures. Binary (black and white) images, in particular, can be transformed using this theory: the sets to be transformed are the sets of neighboring non-zero-valued pixels. The theory was also extended to gray-valued images.



Mathematical-morphology operations use a *structuring element* in order to modify geometrical structures.

Let us first generate a structuring element:

```
>>> el = ndimage.generate_binary_structure(2, 1)
>>> el
array([[False, True, False],
       [...True, True, True],
       [False, True, False]], dtype=bool)
>>> el.astype(np.int)
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```

- **Erosion** `scipy.ndimage.binary_erosion()`

```
>>> a = np.zeros((7, 7), dtype=np.int)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> #Erosion removes objects smaller than the structure
>>> ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
```

```
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

- **Dilation** `scipy.ndimage.binary_dilation()`

```
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

- **Opening** `scipy.ndimage.binary_opening()`

```
>>> a = np.zeros((5, 5), dtype=np.int)
>>> a[1:4, 1:4] = 1
>>> a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
>>> ndimage.binary_opening(a, structure=np.ones((3, 3))).astype(np.int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> ndimage.binary_opening(a).astype(np.int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

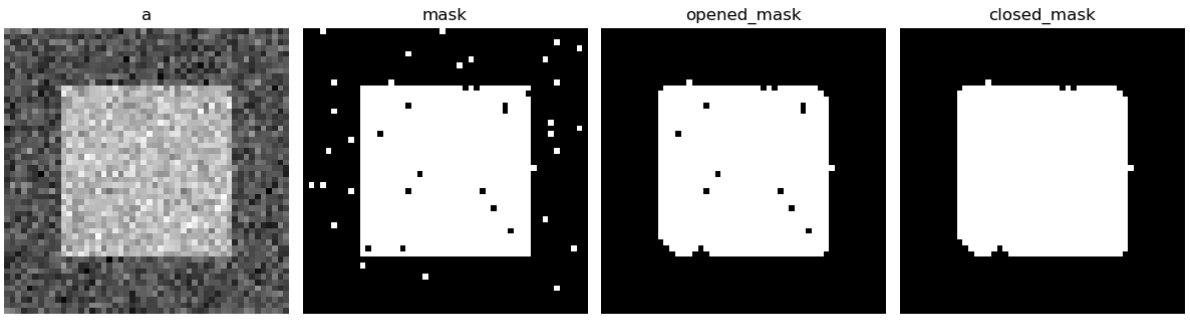
- **Closing:** `scipy.ndimage.binary_closing()`

Exercise

Check that opening amounts to eroding, then dilating.

An opening operation removes small structures, while a closing operation fills small holes. Such operations can therefore be used to "clean" an image.

```
>>> a = np.zeros((50, 50))
>>> a[10:-10, 10:-10] = 1
>>> a += 0.25 * np.random.standard_normal(a.shape)
>>> mask = a>=0.5
>>> opened_mask = ndimage.binary_opening(mask)
>>> closed_mask = ndimage.binary_closing(opened_mask)
```



Exercise

Check that the area of the reconstructed square is smaller than the area of the initial square. (The opposite would occur if the closing step was performed *before* the opening).

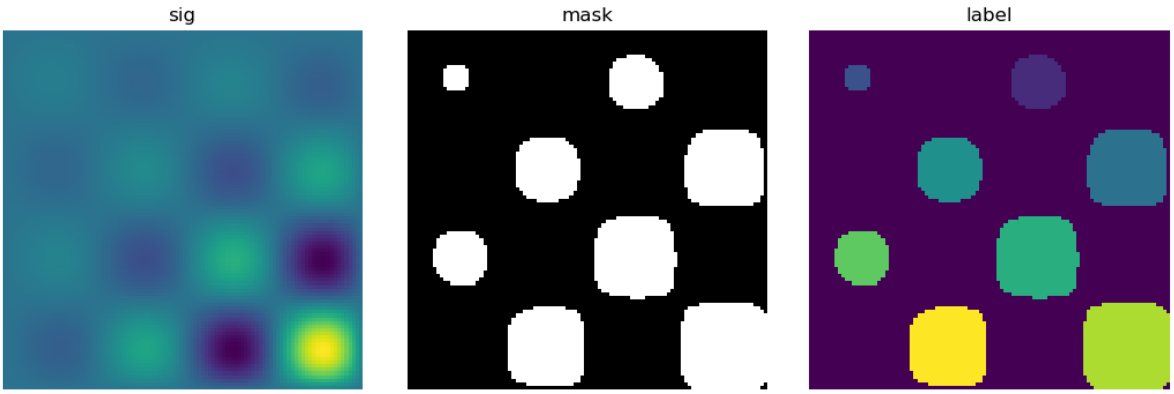
For *gray-valued* images, eroding (resp. dilating) amounts to replacing a pixel by the minimal (resp. maximal) value among pixels covered by the structuring element centered on the pixel of interest.

```
>>> a = np.zeros((7, 7), dtype=np.int)
>>> a[1:6, 1:6] = 3
>>> a[4, 4] = 2; a[2, 3] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 1, 3, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 3, 2, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_erosion(a, size=(3, 3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 3, 2, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

3.10.4 Connected components and measurements on images

Let us first generate a nice synthetic binary image.

```
>>> x, y = np.indices((100, 100))
>>> sig = np.sin(2*np.pi*x/50.) * np.sin(2*np.pi*y/50.) * (1+x*y/50.**2)**2
>>> mask = sig > 1
```

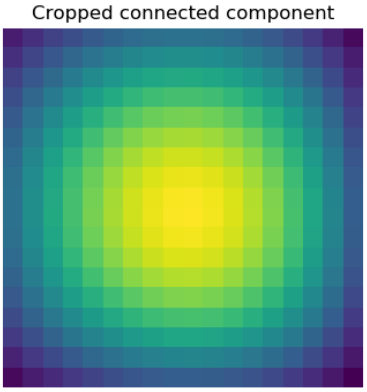


`scipy.ndimage.label()` assigns a different label to each connected component:

```
>>> labels, nb = ndimage.label(mask)
>>> nb
8
```

Now compute measurements on each connected component:

```
>>> areas = ndimage.sum(mask, labels, range(1, labels.max()+1))
>>> areas   # The number of pixels in each connected component
array([ 190.,   45.,  424.,  278.,  459.,  190.,  549.,  424.])
>>> maxima = ndimage.maximum(sig, labels, range(1, labels.max()+1))
>>> maxima  # The maximum signal in each connected component
array([  1.80238238,   1.13527605,   5.51954079,   2.49611818,
         6.71673619,   1.80238238,  16.76547217,   5.51954079])
```



Extract the 4th connected component, and crop the array around it:

```
>>> ndimage.find_objects(labels==4)
[(slice(30L, 48L, None), slice(30L, 48L, None))]
>>> sl = ndimage.find_objects(labels==4)
>>> from matplotlib import pyplot as plt
>>> plt.imshow(sig[sl[0]])
<matplotlib.image.AxesImage object at ...>
```

See the summary exercise on *Image processing application: counting bubbles and unmolten grains* for a more advanced example.

3.11 Summary exercises on scientific computing

The summary exercises use mainly Numpy, Scipy and Matplotlib. They provide some real-life examples of scientific computing with Python. Now that the basics of working with Numpy and Scipy have been introduced, the interested user is invited to try these exercises.

3.11.1 Maximum wind speed prediction at the Sprogø station

The exercise goal is to predict the maximum wind speed occurring every 50 years even if no measure exists for such a period. The available data are only measured over 21 years at the Sprogø meteorological station located in Denmark. First, the statistical steps will be given and then illustrated with functions from the scipy.interpolate module. At the end the interested readers are invited to compute results from raw data and in a slightly different approach.

Statistical approach

The annual maxima are supposed to fit a normal probability density function. However such function is not going to be estimated because it gives a probability from a wind speed maxima. Finding the maximum wind speed occurring every 50 years requires the opposite approach, the result needs to be found from a defined probability. That is the quantile function role and the exercise goal will be to find it. In the current model, it is supposed that the maximum wind speed occurring every 50 years is defined as the upper 2% quantile.

By definition, the quantile function is the inverse of the cumulative distribution function. The latter describes the probability distribution of an annual maxima. In the exercise, the cumulative probability `p_i` for a given year `i` is defined as `p_i = i/(N+1)` with `N = 21`, the number of measured years. Thus it will be possible to calculate the cumulative probability of every measured wind speed maxima. From those experimental points, the scipy.interpolate module will be very useful for fitting the quantile function. Finally the 50 years maxima is going to be evaluated from the cumulative probability of the 2% quantile.

Computing the cumulative probabilities

The annual wind speeds maxima have already been computed and saved in the numpy format in the file `examples/max-speeds.npy`, thus they will be loaded by using numpy:

```
>>> import numpy as np
>>> max_speeds = np.load('intro/summary-exercises/examples/max-speeds.npy')
>>> years_nb = max_speeds.shape[0]
```

Following the cumulative probability definition `p_i` from the previous section, the corresponding values will be:

```
>>> cprob = (np.arange(years_nb, dtype=np.float32) + 1)/(years_nb + 1)
```

and they are assumed to fit the given wind speeds:

```
>>> sorted_max_speeds = np.sort(max_speeds)
```

Prediction with UnivariateSpline

In this section the quantile function will be estimated by using the `UnivariateSpline` class which can represent a spline from points. The default behavior is to build a spline of degree 3 and points can have different weights according to their reliability. Variants are `InterpolatedUnivariateSpline` and `LSQUnivariateSpline` on which errors checking is going to change. In case a 2D spline is wanted, the `BivariateSpline` class family is provided. All those classes for 1D and 2D splines use the FITPACK Fortran subroutines, that's why a lower library access is available through the `splrep` and `splev` functions for respectively representing and evaluating a spline. Moreover interpolation functions without the use of FITPACK parameters are also provided for simpler use (see `interp1d`, `interp2d`, `barycentric_interpolate` and so on).

For the Sprogø maxima wind speeds, the `UnivariateSpline` will be used because a spline of degree 3 seems to correctly fit the data:

```
>>> from scipy.interpolate import UnivariateSpline
>>> quantile_func = UnivariateSpline(cprob, sorted_max_speeds)
```

The quantile function is now going to be evaluated from the full range of probabilities:

```
>>> nprob = np.linspace(0, 1, 1e2)
>>> fitted_max_speeds = quantile_func(nprob)
```

In the current model, the maximum wind speed occurring every 50 years is defined as the upper 2% quantile. As a result, the cumulative probability value will be:

```
>>> fifty_prob = 1. - 0.02
```

So the storm wind speed occurring every 50 years can be guessed by:

```
>>> fifty_wind = quantile_func(fifty_prob)
>>> fifty_wind
array(32.97989825...)
```

The results are now gathered on a Matplotlib figure:

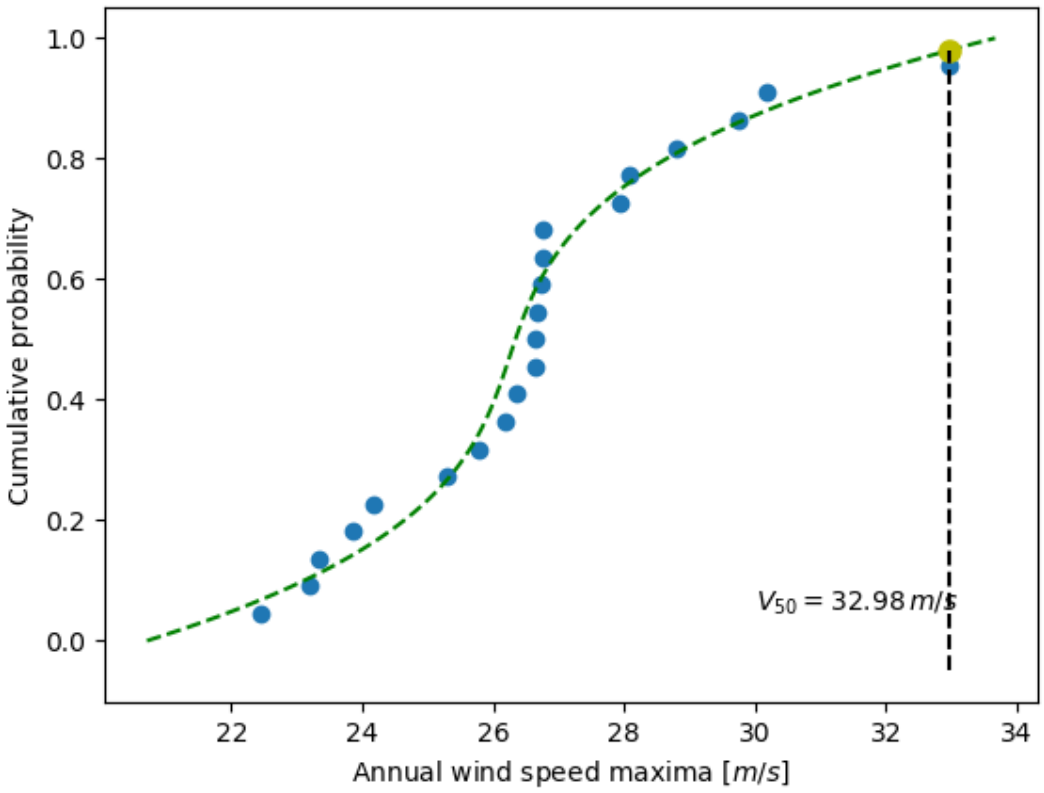


Fig. 3.1: Solution: Python source file

Exercise with the Gumbell distribution

The interested readers are now invited to make an exercise by using the wind speeds measured over 21 years. The measurement period is around 90 minutes (the original period was around 10 minutes but the file size has been reduced for making the exercise setup easier). The data are stored in numpy format inside the file `examples/sprog-windspeeds.npy`. Do not look at the source code for the plots until you have completed the exercise.

- The first step will be to find the annual maxima by using numpy and plot them as a matplotlib bar figure.

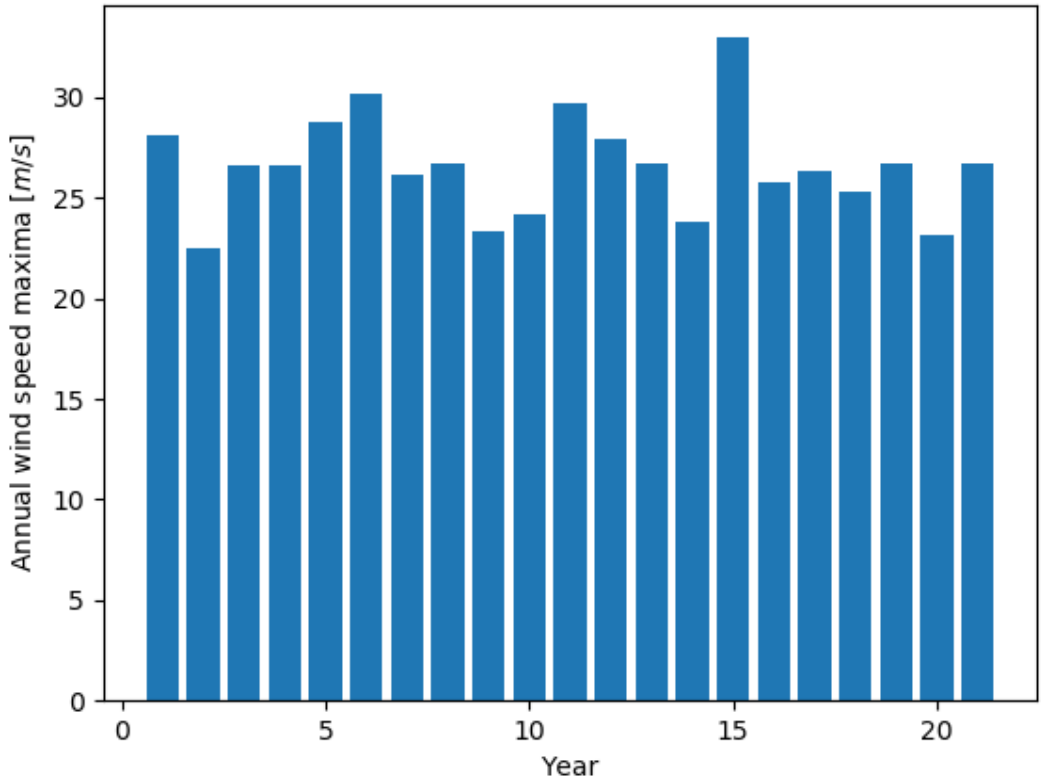


Fig. 3.2: Solution: Python source file

- The second step will be to use the Gumbell distribution on cumulative probabilities `p_i` defined as `-log( -log(p_i) )` for fitting a linear quantile function (remember that you can define the degree of the `UnivariateSpline`). Plotting the annual maxima versus the Gumbell distribution should give you the following figure.

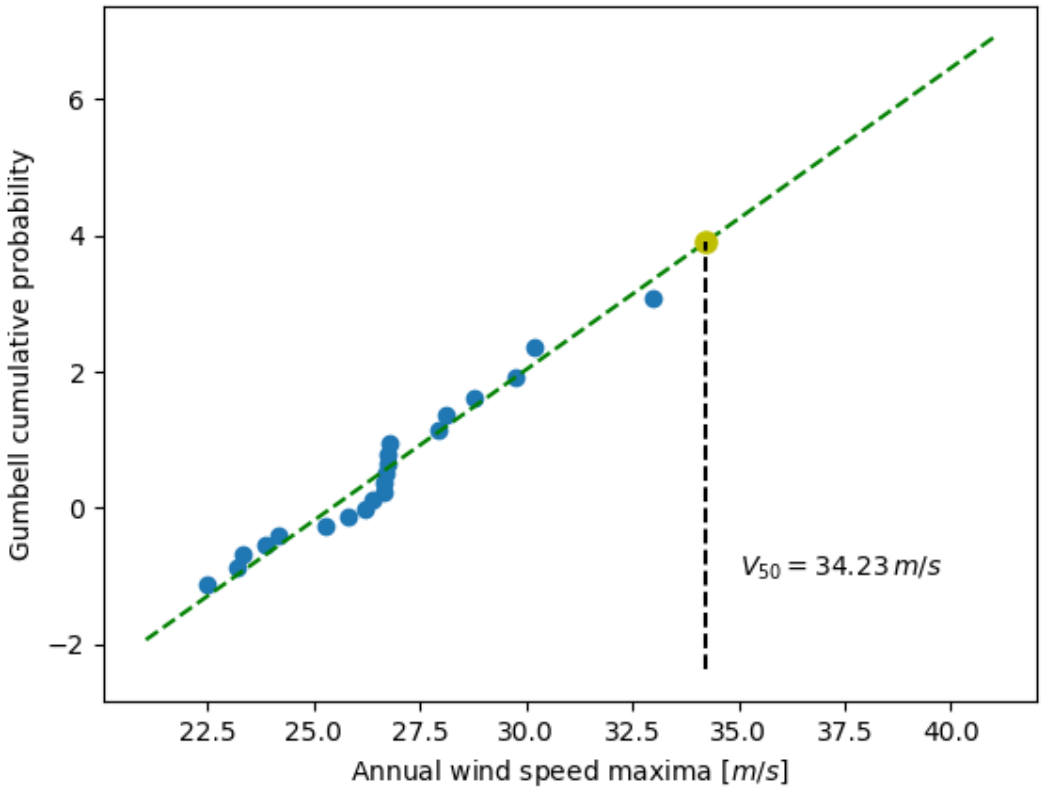


Fig. 3.3: Solution: Python source file

- The last step will be to find 34.23 m/s for the maximum wind speed occurring every 50 years.

3.11.2 Non linear least squares curve fitting: application to point extraction in topographical lidar data

The goal of this exercise is to fit a model to some data. The data used in this tutorial are lidar data and are described in details in the following introductory paragraph. If you're impatient and want to practice now, please skip it and go directly to *Loading and visualization*.

Introduction

Lidars systems are optical rangefinders that analyze property of scattered light to measure distances. Most of them emit a short light impulsion towards a target and record the reflected signal. This signal is then processed to extract the distance between the lidar system and the target.

Topographical lidar systems are such systems embedded in airborne platforms. They measure distances between the platform and the Earth, so as to deliver information on the Earth's topography (see[1] for more details).

[1] Mallet, C. and Bretar, F. Full-Waveform Topographic Lidar: State-of-the-Art. *ISPRS Journal of Photogrammetry and Remote Sensing* 64(1), pp.1-16, January 2009 http://dx.doi.org/10.1016/j.isprsjprs.2008.09.007

In this tutorial, the goal is to analyze the waveform recorded by the lidar system[2]. Such a signal contains peaks whose center and amplitude permit to compute the position and some characteristics of the hit target. When the footprint of the laser beam is around 1m on the Earth surface, the beam can hit multiple targets during the two-way propagation (for example the ground and the top of a tree or building). The sum of the contributions of each target hit by the laser beam then produces a complex signal with multiple peaks, each one containing information about one target.

One state of the art method to extract information from these data is to decompose them in a sum of Gaussian functions where each function represents the contribution of a target hit by the laser beam.

Therefore, we use the `scipy.optimize` module to fit a waveform to one or a sum of Gaussian functions.

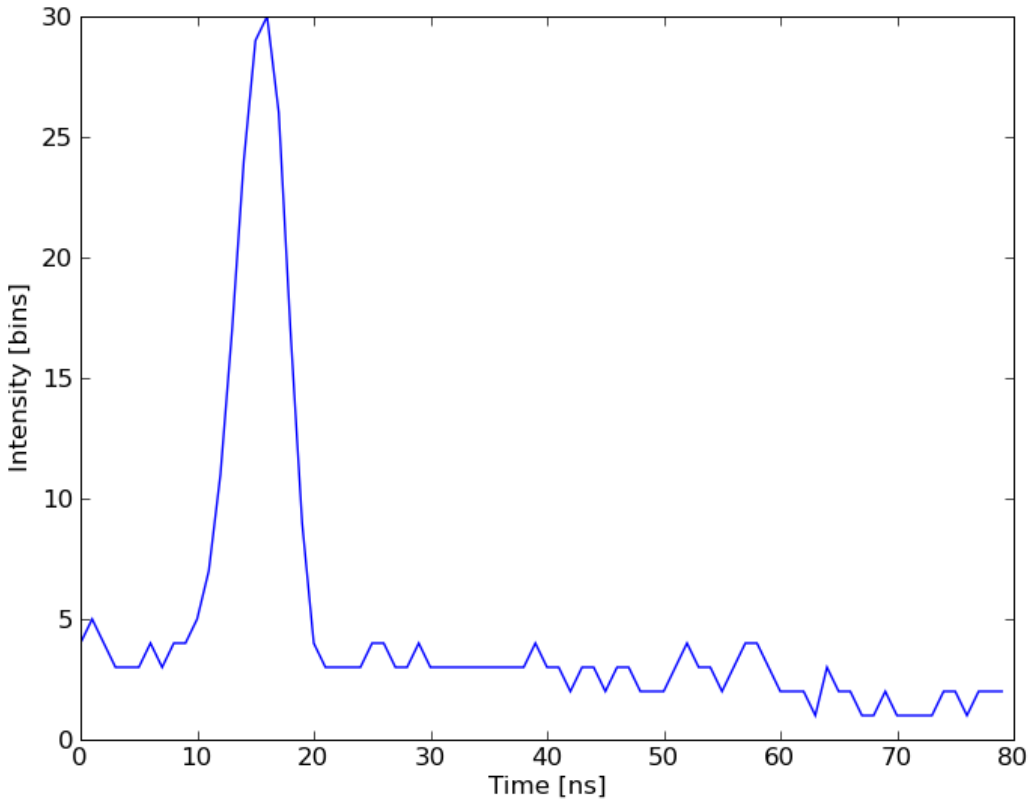
Loading and visualization

Load the first waveform using:

```
>>> import numpy as np
>>> waveform_1 = np.load('data/waveform_1.npy')
```

and visualize it:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(len(waveform_1))
>>> plt.plot(t, waveform_1)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.show()
```



As you can notice, this waveform is a 80-bin-length signal with a single peak.

[2] The data used for this tutorial are part of the demonstration data available for the FullAnalyze software and were kindly provided by the GIS DRAIX.

Fitting a waveform with a simple Gaussian model

The signal is very simple and can be modeled as a single Gaussian function and an offset corresponding to the background noise. To fit the signal with the function, we must:

- define the model
- propose an initial solution
- call `scipy.optimize.leastsq`

Model

A Gaussian function defined by

$$B + A\exp\left\{-\left(\frac{t-\mu}{\sigma}\right)^2\right\}$$

can be defined in python by:

```
>>> def model(t, coeffs):
...     return coeffs[0] + coeffs[1] * np.exp( - ((t-coeffs[2])/coeffs[3])**2 )
```

where

- `coeffs[0]` is B (noise)
- `coeffs[1]` is A (amplitude)
- `coeffs[2]` is μ (center)
- `coeffs[3]` is σ (width)

Initial solution

An approximative initial solution that we can find from looking at the graph is for instance:

```
>>> x0 = np.array([3, 30, 15, 1], dtype=float)
```

Fit

`scipy.optimize.leastsq` minimizes the sum of squares of the function given as an argument. Basically, the function to minimize is the residuals (the difference between the data and the model):

```
>>> def residuals(coeffs, y, t):
...     return y - model(t, coeffs)
```

So let's get our solution by calling `scipy.optimize.leastsq()` with the following arguments:

- the function to minimize
- an initial solution
- the additional arguments to pass to the function

```
>>> from scipy.optimize import leastsq
>>> x, flag = leastsq(residuals, x0, args=(waveform_1, t))
>>> print(x)
[  2.70363341  27.82020742  15.47924562   3.05636228]
```

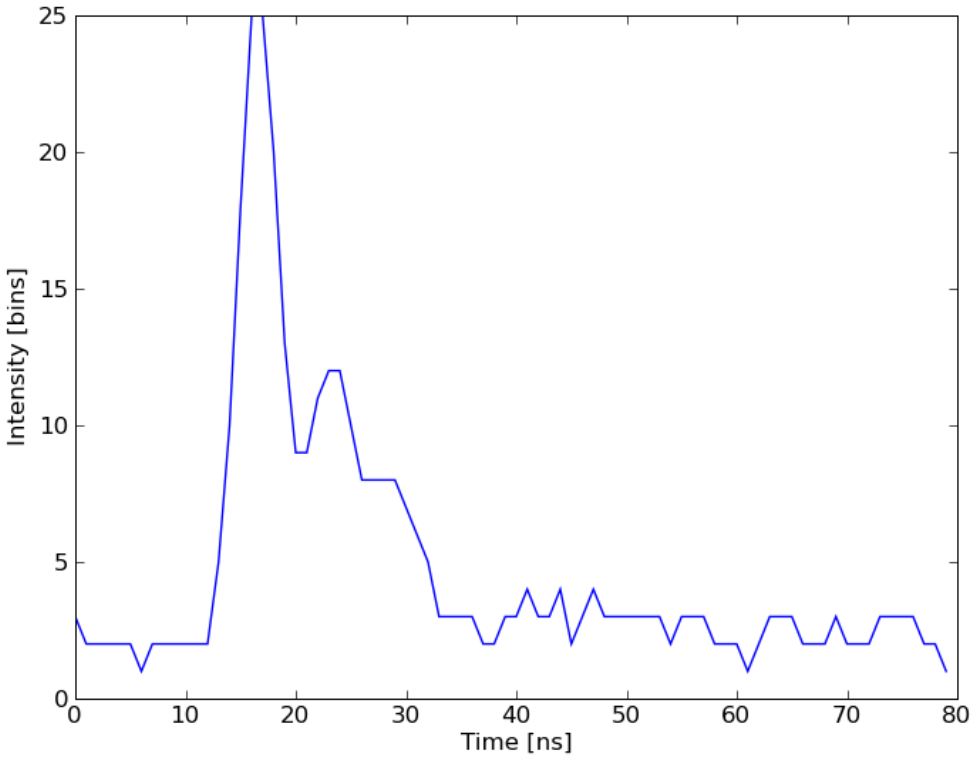
And visualize the solution:

```
>>> plt.plot(t, waveform_1, t, model(t, x))
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
>>> plt.legend(['waveform', 'model'])
<matplotlib.legend.Legend object at ...>
>>> plt.show()
```

Remark: from scipy v0.8 and above, you should rather use `scipy.optimize.curve_fit()` which takes the model and the data as arguments, so you don't need to define the residuals any more.

Going further

- Try with a more complex waveform (for instance `data/waveform_2.npy`) that contains three significant peaks. You must adapt the model which is now a sum of Gaussian functions instead of only one Gaussian peak.



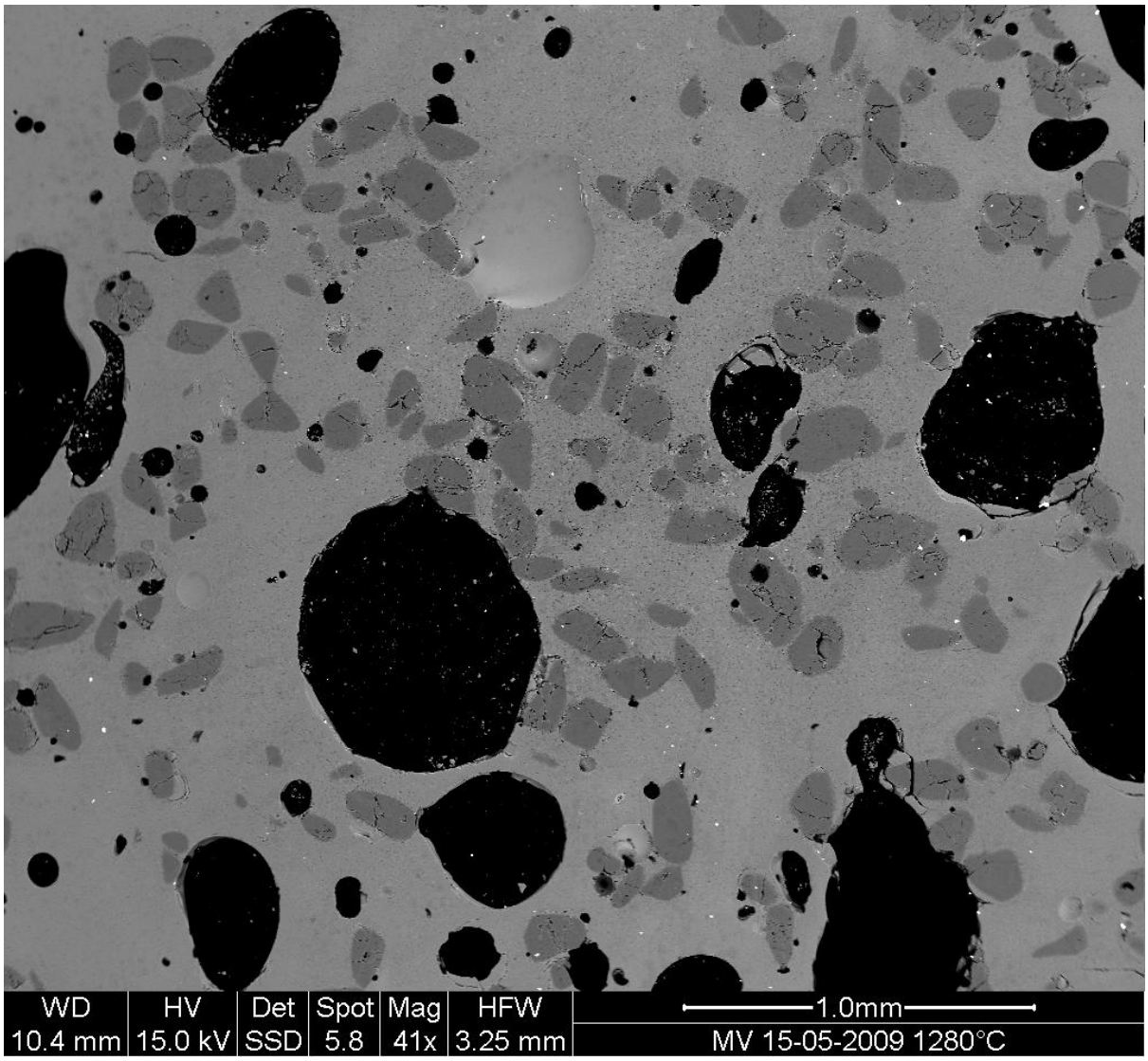
- In some cases, writing an explicit function to compute the Jacobian is faster than letting `leastsq` estimate it numerically. Create a function to compute the Jacobian of the residuals and use it as an input for `leastsq`.
- When we want to detect very small peaks in the signal, or when the initial guess is too far from a good solution, the result given by the algorithm is often not satisfying. Adding constraints to the parameters of the model enables to overcome such limitations. An example of *a priori* knowledge we can add is the sign of our variables (which are all positive).

  With the following initial solution:

  ```
  >>> x0 = np.array([3, 50, 20, 1], dtype=float)
  ```

  compare the result of `scipy.optimize.leastsq()` and what you can get with `scipy.optimize.fmin_slsqp()` when adding boundary constraints.

3.11.3 Image processing application: counting bubbles and unmolten grains



Statement of the problem

1. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the docstring of `imshow` to display the image with the "right" orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).

This Scanning Element Microscopy image shows a glass sample (light gray matrix) with some bubbles (on black) and unmolten sand grains (dark gray). We wish to determine the fraction of the sample covered by these three phases, and to estimate the typical size of sand grains and bubbles, their sizes, etc.

2. Crop the image to remove the lower panel with measure information.

3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.

4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option (homework): write a function that determines automatically the thresholds from the minima of the histogram.

5. Display an image in which the three phases are colored with three different colors.

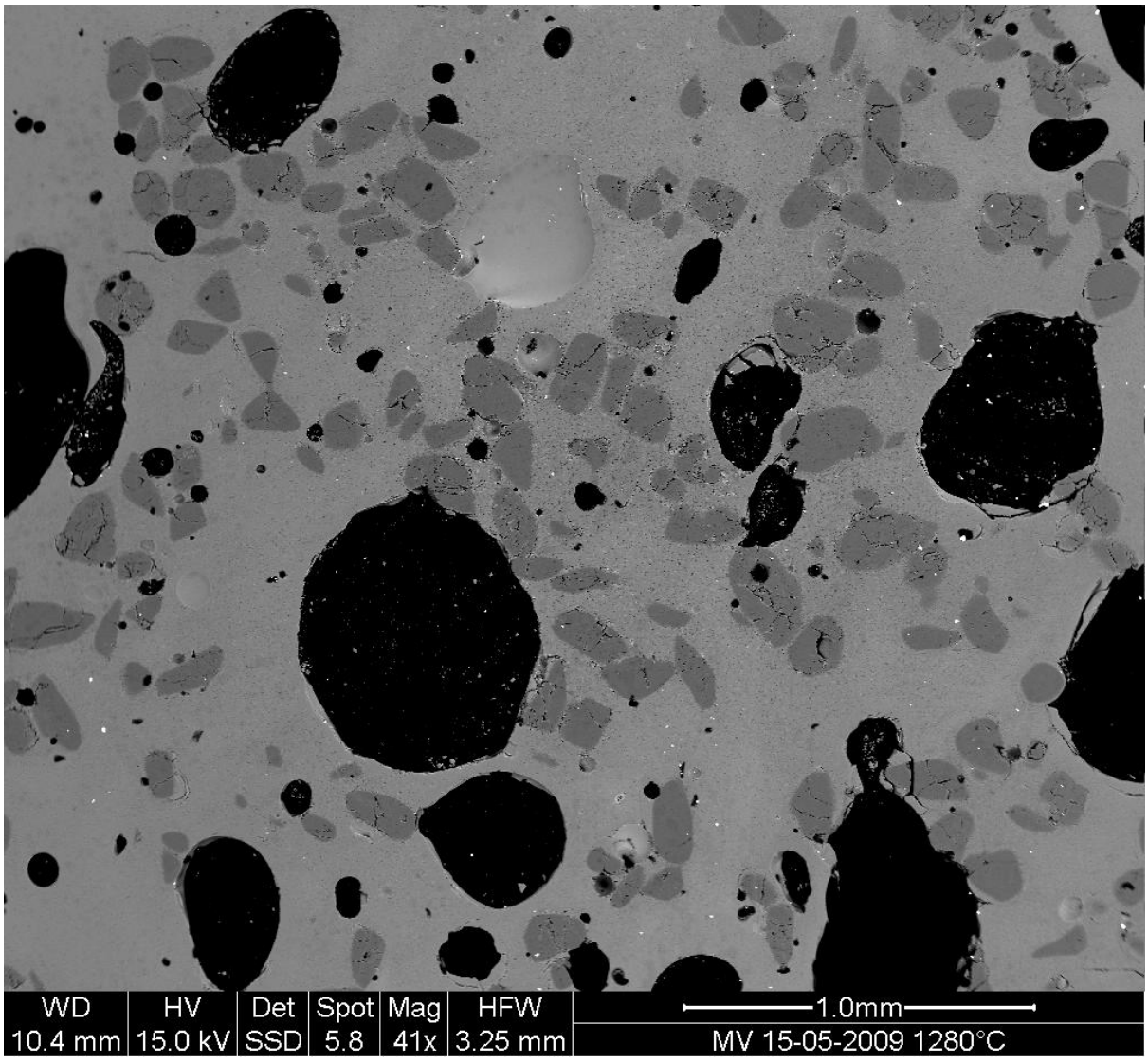
6. Use mathematical morphology to clean the different phases.

7. Attribute labels to all bubbles and sand grains, and remove from the sand mask grains that are smaller than 10 pixels. To do so, use `ndimage.sum` or `np.bincount` to compute the grain sizes.

8. Compute the mean size of bubbles.

Proposed solution

3.11.4 Example of solution for the image processing exercise: unmolten grains in glass



1. Open the image file MV_HFV_012.jpg and display it. Browse through the keyword arguments in the docstring of `imshow` to display the image with the "right" orientation (origin in the bottom left corner, and not the upper left corner as for standard arrays).

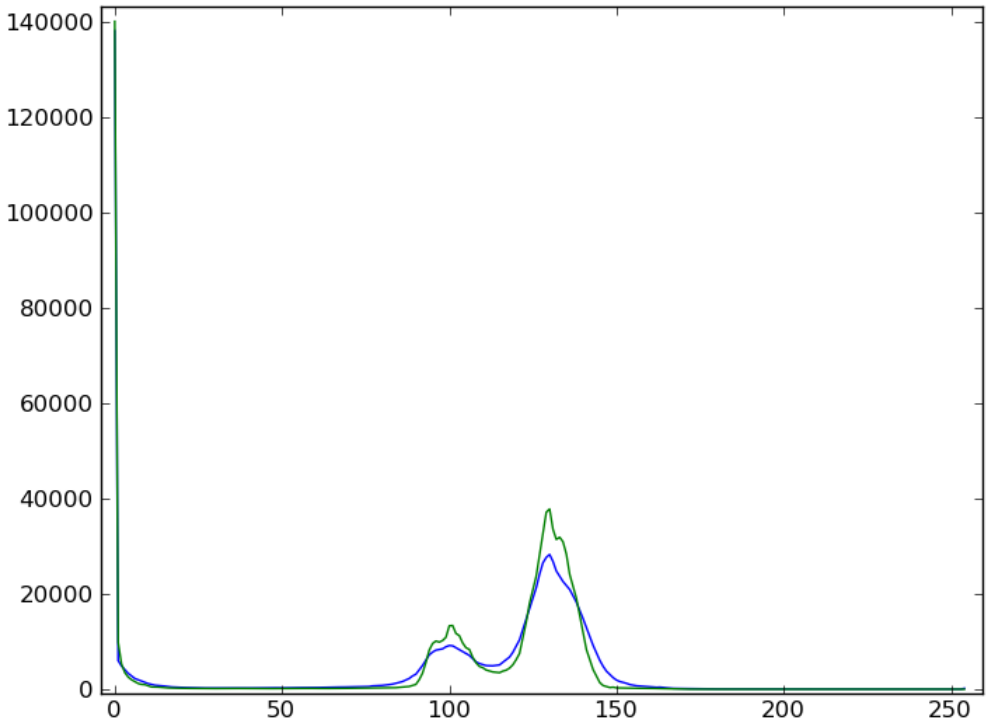
```
>>> dat = pl.imread('data/MV_HFV_012.jpg')
```

2. Crop the image to remove the lower panel with measure information.

```
>>> dat = dat[:-60]
```

3. Slightly filter the image with a median filter in order to refine its histogram. Check how the histogram changes.

```
>>> filtdat = ndimage.median_filter(dat, size=(7,7))
>>> hi_dat = np.histogram(dat, bins=np.arange(256))
>>> hi_filtdat = np.histogram(filtdat, bins=np.arange(256))
```

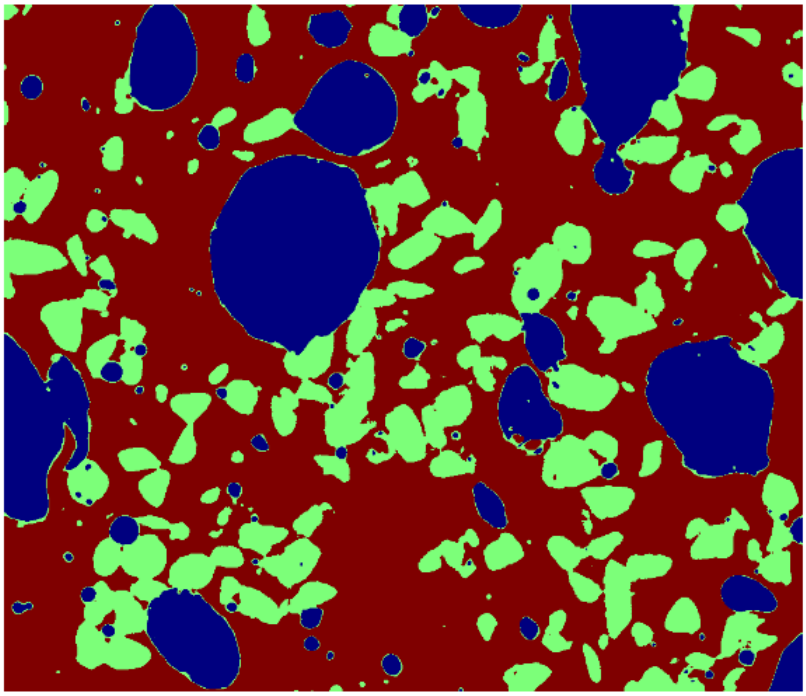


4. Using the histogram of the filtered image, determine thresholds that allow to define masks for sand pixels, glass pixels and bubble pixels. Other option (homework): write a function that determines automatically the thresholds from the minima of the histogram.

```
>>> void = filtdat <= 50
>>> sand = np.logical_and(filtdat > 50, filtdat <= 114)
>>> glass = filtdat > 114
```

5. Display an image in which the three phases are colored with three different colors.

```
>>> phases = void.astype(np.int) + 2*glass.astype(np.int) + 3*sand.astype(np.int)
```

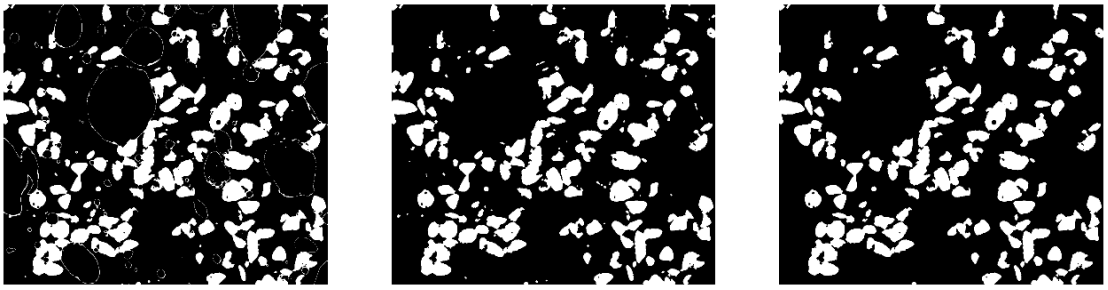


6. Use mathematical morphology to clean the different phases.

```
>>> sand_op = ndimage.binary_opening(sand, iterations=2)
```

7. Attribute labels to all bubbles and sand grains, and remove from the sand mask grains that are smaller than 10 pixels. To do so, use `ndimage.sum` or `np.bincount` to compute the grain sizes.

```
>>> sand_labels, sand_nb = ndimage.label(sand_op)
>>> sand_areas = np.array(ndimage.sum(sand_op, sand_labels, np.arange(sand_labels.
↪max()+1)))
>>> mask = sand_areas > 100
>>> remove_small_sand = mask[sand_labels.ravel()].reshape(sand_labels.shape)
```



8. Compute the mean size of bubbles.

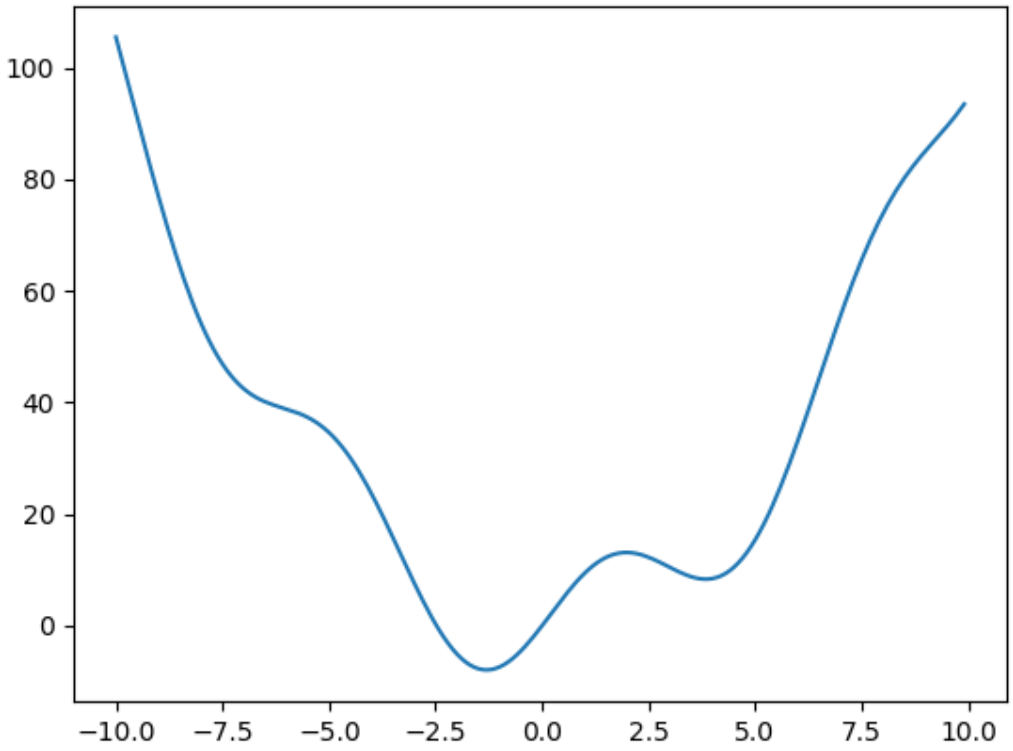
```
>>> bubbles_labels, bubbles_nb = ndimage.label(void)
>>> bubbles_areas = np.bincount(bubbles_labels.ravel())[1:]
>>> mean_bubble_size = bubbles_areas.mean()
>>> median_bubble_size = np.median(bubbles_areas)
>>> mean_bubble_size, median_bubble_size
(1699.875, 65.0)
```

3.12 Full code examples for the scipy chapter

3.12.1 Finding the minimum of a smooth function

Demos various methods to find the minimum of a function.

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

def f(x):
    return x**2 + 10*np.sin(x)


x = np.arange(-10, 10, 0.1)
plt.plot(x, f(x))
```



Now find the minimum with a few methods

```
from scipy import optimize

# The default (Nelder Mead)
print(optimize.minimize(f, x0=0))
```

Out:

```
fun: -7.945823375615215
 hess_inv: array([[ 0.08589237]])
```

```
      jac: array([ -1.19209290e-06])
  message: 'Optimization terminated successfully.'
     nfev: 18
      nit: 5
     njev: 6
   status: 0
  success: True
        x: array([-1.30644012])
```

```
print(optimize.minimize(f, x0=0, method="L-BFGS-B"))
```

Out:

```
fun: array([-7.94582338])
 hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
      jac: array([ -1.42108547e-06])
  message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 12
      nit: 5
   status: 0
  success: True
        x: array([-1.30644013])
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.056 seconds)

3.12.2 Detrending a signal

`scipy.signal.detrend()` removes a linear trend.

Generate a random signal with a trend

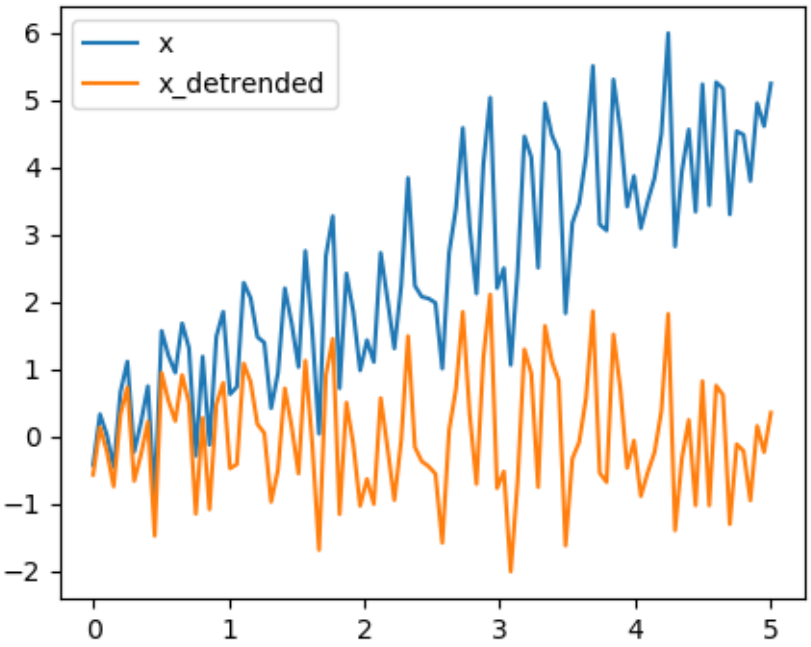
```
import numpy as np
t = np.linspace(0, 5, 100)
x = t + np.random.normal(size=100)
```

Detrend

```
from scipy import signal
x_detrended = signal.detrend(x)
```

Plot

```
from matplotlib import pyplot as plt
plt.figure(figsize=(5, 4))
plt.plot(t, x, label="x")
plt.plot(t, x_detrended, label="x_detrended")
plt.legend(loc='best')
plt.show()
```



Total running time of the script: (0 minutes 0.245 seconds)

3.12.3 Resample a signal with scipy.signal.resample

`scipy.signal.resample()` uses FFT to resample a 1D signal.

Generate a signal with 100 data point

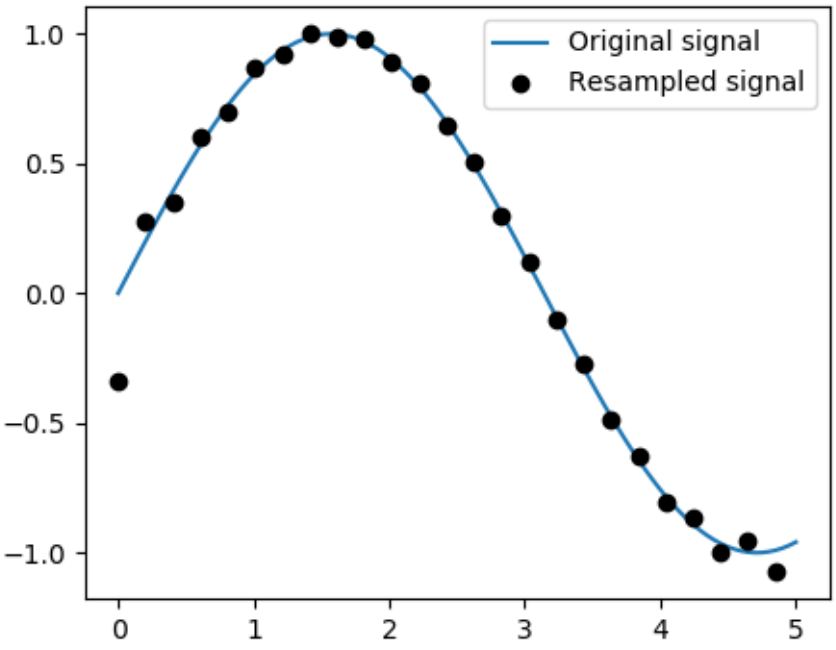
```
import numpy as np
t = np.linspace(0, 5, 100)
x = np.sin(t)
```

Downsample it by a factor of 4

```
from scipy import signal
x_resampled = signal.resample(x, 25)
```

Plot

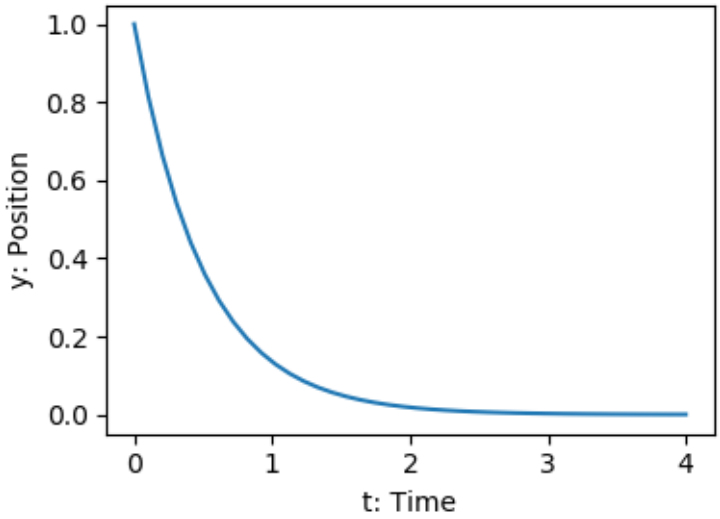
```
from matplotlib import pyplot as plt
plt.figure(figsize=(5, 4))
plt.plot(t, x, label='Original signal')
plt.plot(t[::4], x_resampled, 'ko', label='Resampled signal')

plt.legend(loc='best')
plt.show()
```



Total running time of the script: (0 minutes 0.055 seconds)

3.12.4 Integrating a simple ODE

Solve the ODE dy/dt = -2y between t = 0..4, with the initial condition y(t=0) = 1.

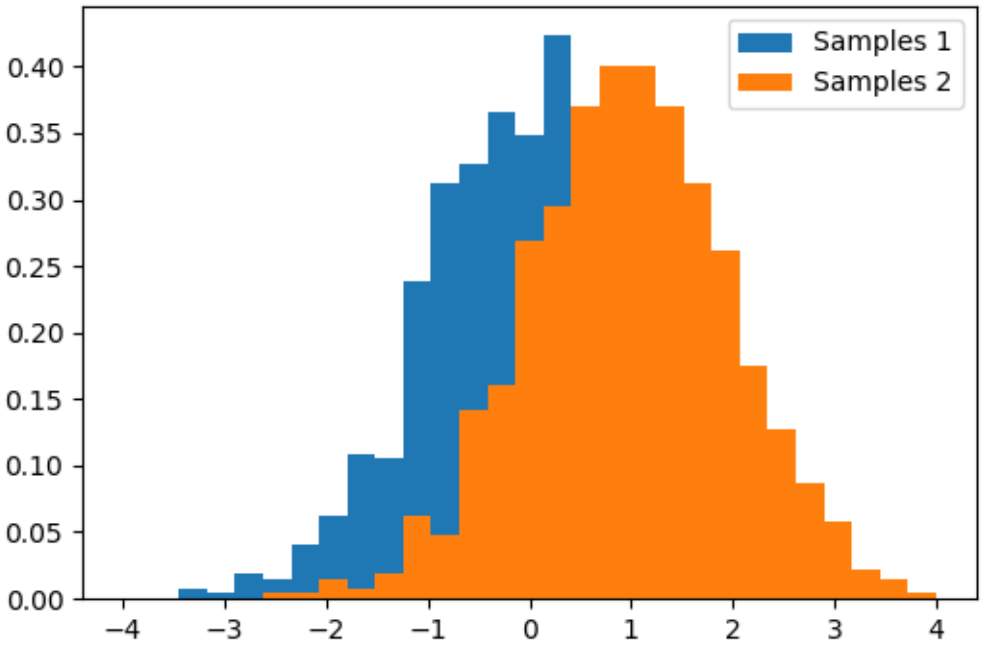



```
import numpy as np
from scipy.integrate import odeint
from matplotlib import pyplot as plt

def calc_derivative(ypos, time):
    return -2*ypos

time_vec = np.linspace(0, 4, 40)
```

```
yvec = odeint(calc_derivative, 1, time_vec)

plt.figure(figsize=(4, 3))
plt.plot(time_vec, yvec)
plt.xlabel('t: Time')
plt.ylabel('y: Position')
plt.tight_layout()
```

Total running time of the script: (0 minutes 0.083 seconds)

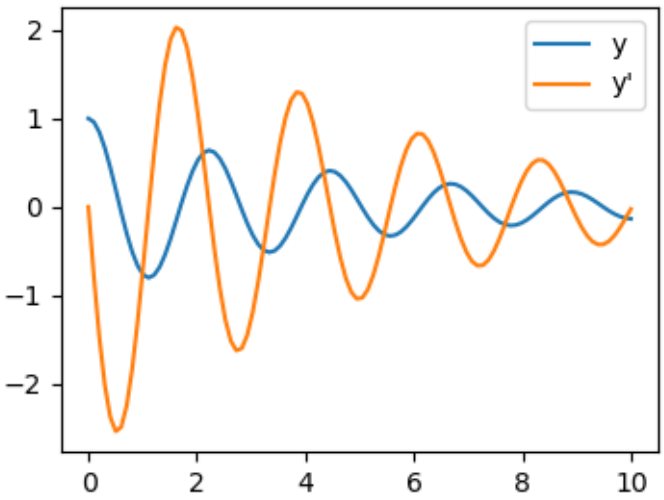
3.12.5 Comparing 2 sets of samples from Gaussians




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

# Generates 2 sets of observations
samples1 = np.random.normal(0, size=1000)
samples2 = np.random.normal(1, size=1000)

# Compute a histogram of the sample
bins = np.linspace(-4, 4, 30)
histogram1, bins = np.histogram(samples1, bins=bins, normed=True)
histogram2, bins = np.histogram(samples2, bins=bins, normed=True)

plt.figure(figsize=(6, 4))
plt.hist(samples1, bins=bins, normed=True, label="Samples 1")
plt.hist(samples2, bins=bins, normed=True, label="Samples 2")
plt.legend(loc='best')
plt.show()
```

Total running time of the script: (0 minutes 0.107 seconds)

3.12.6 Integrate the Damped spring-mass oscillator

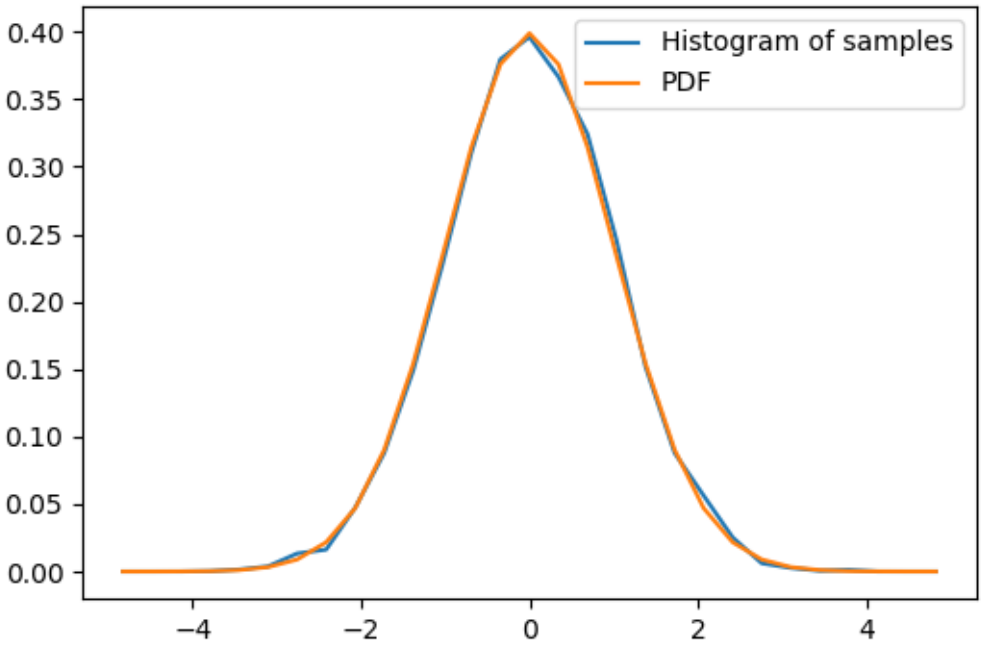



```
import numpy as np
from scipy.integrate import odeint
from matplotlib import pyplot as plt

mass = 0.5  # kg
kspring = 4  # N/m
cviscous = 0.4  # N s/m


eps = cviscous / (2 * mass * np.sqrt(kspring/mass))
omega = np.sqrt(kspring / mass)


def calc_deri(yvec, time, eps, omega):
    return (yvec[1], -eps * omega * yvec[1] - omega **2 * yvec[0])

time_vec = np.linspace(0, 10, 100)
yinit = (1, 0)
yarr = odeint(calc_deri, yinit, time_vec, args=(eps, omega))

plt.figure(figsize=(4, 3))
plt.plot(time_vec, yarr[:, 0], label='y')
plt.plot(time_vec, yarr[:, 1], label="y'")
plt.legend(loc='best')
plt.show()
```

Total running time of the script: (0 minutes 0.055 seconds)

3.12.7 Normal distribution: histogram and PDF

Explore the normal distribution: a histogram built from samples and the PDF (probability density function).




```
import numpy as np

# Sample from a normal distribution using numpy's random number generator
samples = np.random.normal(size=10000)

# Compute a histogram of the sample
bins = np.linspace(-5, 5, 30)
histogram, bins = np.histogram(samples, bins=bins, normed=True)

bin_centers = 0.5*(bins[1:] + bins[:-1])

# Compute the PDF on the bin centers from scipy distribution object
from scipy import stats
pdf = stats.norm.pdf(bin_centers)

from matplotlib import pyplot as plt
plt.figure(figsize=(6, 4))
plt.plot(bin_centers, histogram, label="Histogram of samples")
plt.plot(bin_centers, pdf, label="PDF")
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.056 seconds)

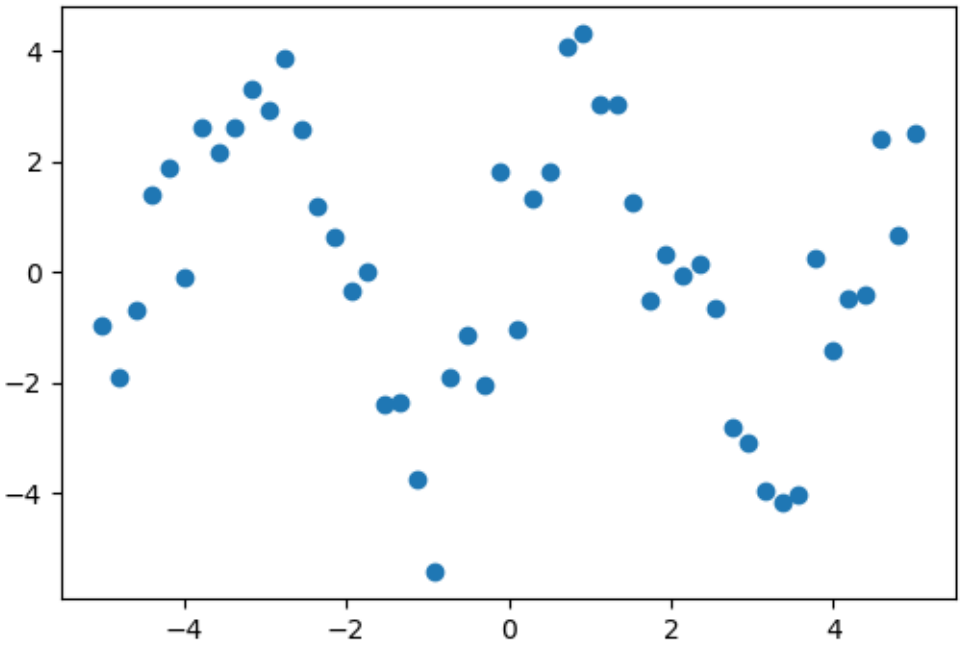
3.12.8 Curve fitting

Demos a simple curve fitting

First generate some data

```
import numpy as np

# Seed the random number generator for reproducibility
np.random.seed(0)
```

```
x_data = np.linspace(-5, 5, num=50)
y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=50)

# And plot it
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data)
```



Now fit a simple sine function to the data

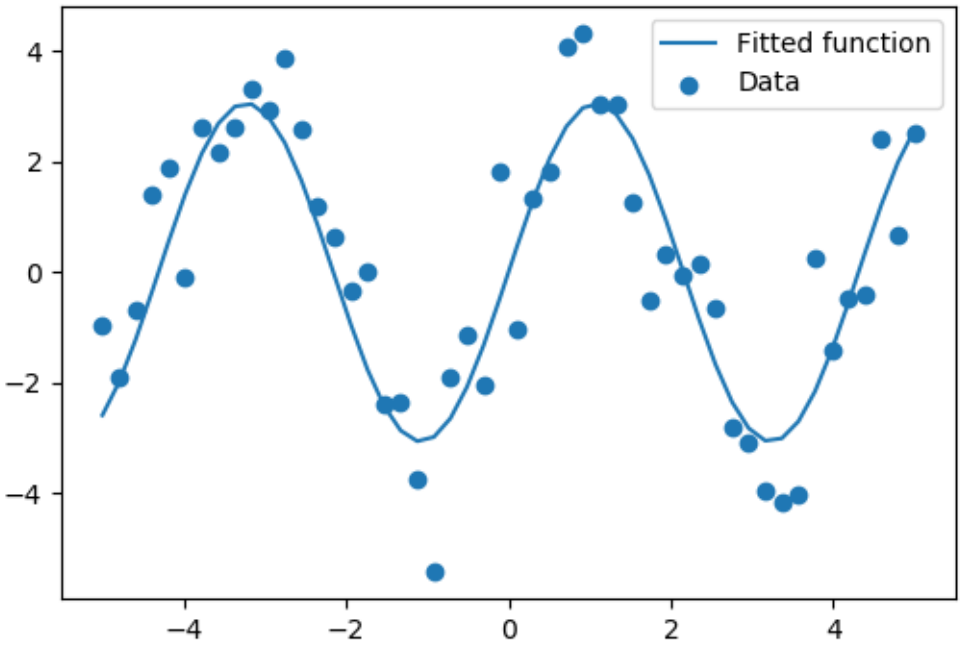
```
from scipy import optimize

def test_func(x, a, b):
    return a * np.sin(b * x)

params, params_covariance = optimize.curve_fit(test_func, x_data, y_data,
                                               p0=[2, 2])

print(params)
```

Out:

```
[ 3.05931973  1.45754553]
```

And plot the resulting curve on the data

```
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_data, test_func(x_data, params[0], params[1]),
         label='Fitted function')

plt.legend(loc='best')

plt.show()
```



Total running time of the script: (0 minutes 0.107 seconds)

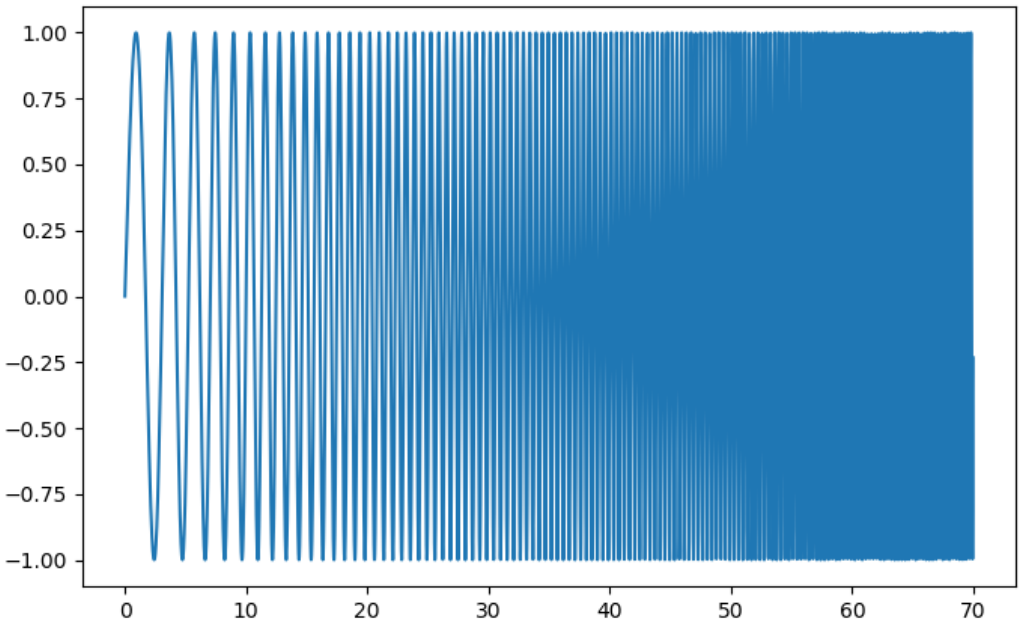
3.12.9 Spectrogram, power spectral density

Demo spectrogram and power spectral density on a frequency chirp.

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

Generate a chirp signal

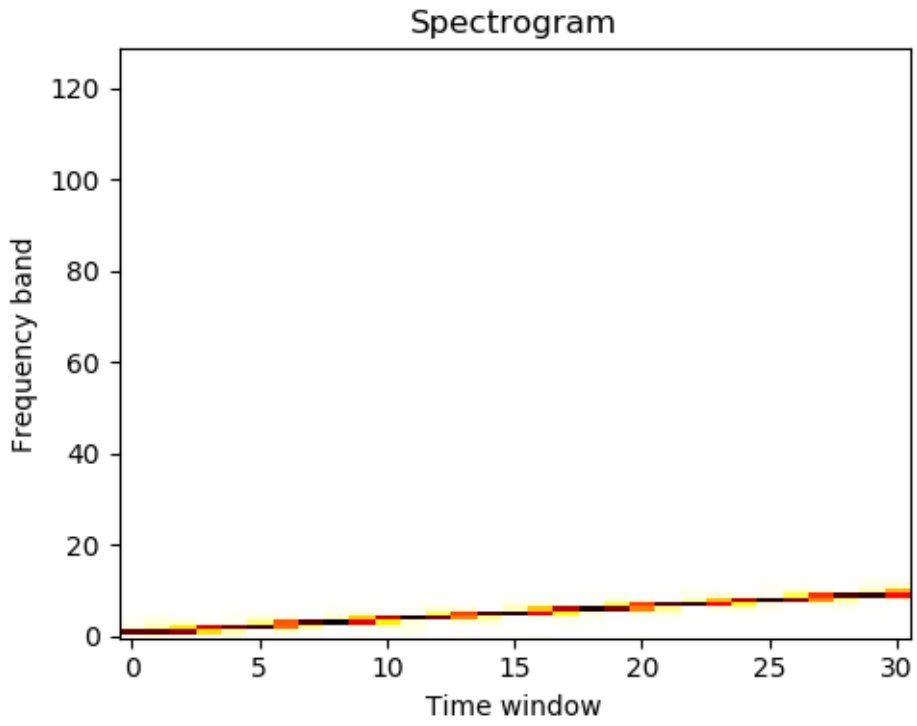
```
# Seed the random number generator
np.random.seed(0)

time_step = .01
time_vec = np.arange(0, 70, time_step)

# A signal with a small frequency chirp
sig = np.sin(0.5 * np.pi * time_vec * (1 + .1 * time_vec))

plt.figure(figsize=(8, 5))
plt.plot(time_vec, sig)
```



Compute and plot the spectrogram

The spectrum of the signal on consecutive time windows

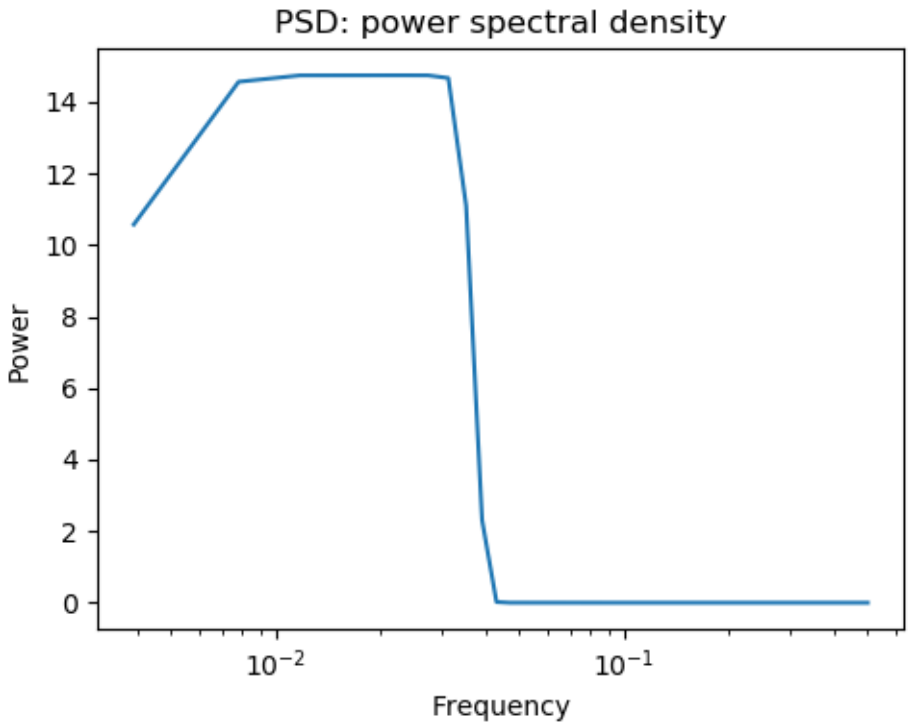
```
from scipy import signal
freqs, times, spectrogram = signal.spectrogram(sig)

plt.figure(figsize=(5, 4))
plt.imshow(spectrogram, aspect='auto', cmap='hot_r', origin='lower')
plt.title('Spectrogram')
plt.ylabel('Frequency band')
plt.xlabel('Time window')
plt.tight_layout()
```



Compute and plot the power spectral density (PSD)

The power of the signal per frequency band

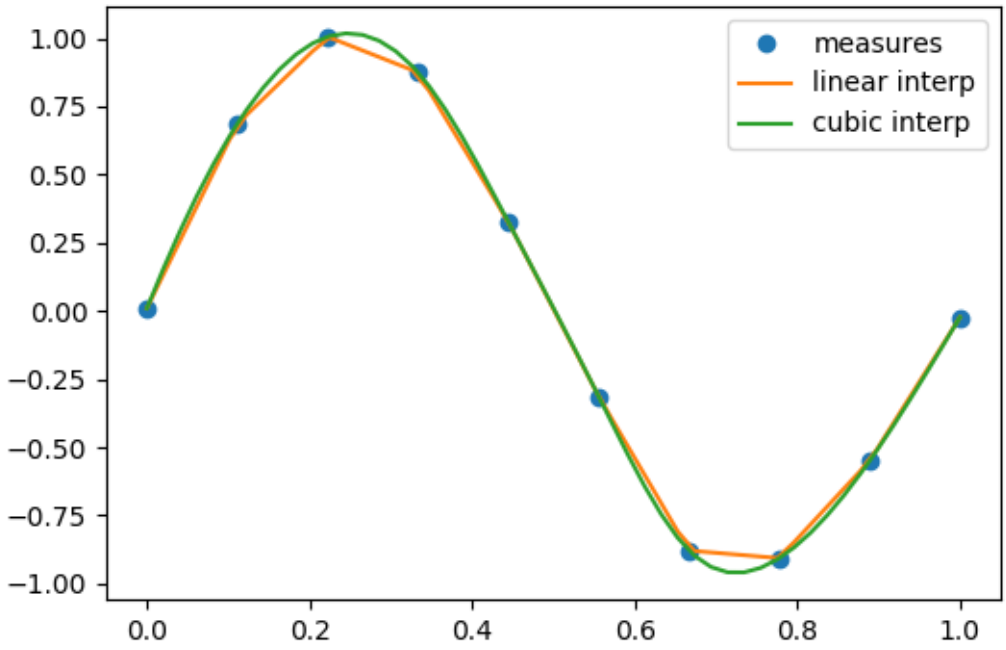
```
freqs, psd = signal.welch(sig)

plt.figure(figsize=(5, 4))
plt.semilogx(freqs, psd)
plt.title('PSD: power spectral density')
plt.xlabel('Frequency')
plt.ylabel('Power')
plt.tight_layout()
```




```
plt.show()
```

Total running time of the script: (0 minutes 0.428 seconds)

3.12.10 A demo of 1D interpolation

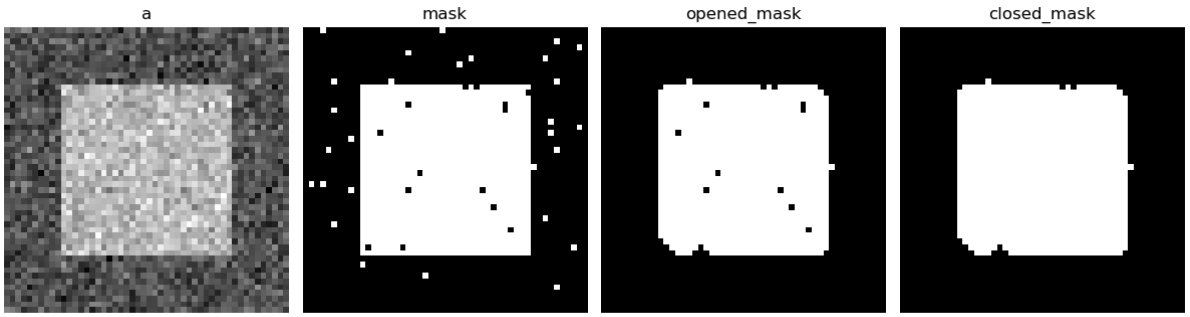


```
# Generate data
import numpy as np
np.random.seed(0)
measured_time = np.linspace(0, 1, 10)
noise = 1e-1 * (np.random.random(10)*2 - 1)
measures = np.sin(2 * np.pi * measured_time) + noise

# Interpolate it to new time points
from scipy.interpolate import interp1d
linear_interp = interp1d(measured_time, measures)
interpolation_time = np.linspace(0, 1, 50)
linear_results = linear_interp(interpolation_time)
cubic_interp = interp1d(measured_time, measures, kind='cubic')
cubic_results = cubic_interp(interpolation_time)

# Plot the data and the interpolation
from matplotlib import pyplot as plt
plt.figure(figsize=(6, 4))
plt.plot(measured_time, measures, 'o', ms=6, label='measures')
plt.plot(interpolation_time, linear_results, label='linear interp')
plt.plot(interpolation_time, cubic_results, label='cubic interp')
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.058 seconds)

3.12.11 Demo mathematical morphology

A basic demo of binary opening and closing.




```
# Generate some binary data
import numpy as np
np.random.seed(0)
a = np.zeros((50, 50))
a[10:-10, 10:-10] = 1
a += 0.25 * np.random.standard_normal(a.shape)
mask = a>=0.5

# Apply mathematical morphology
from scipy import ndimage
opened_mask = ndimage.binary_opening(mask)
closed_mask = ndimage.binary_closing(opened_mask)

# Plot
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
```

```
plt.imshow(a, cmap=plt.cm.gray)
plt.axis('off')
plt.title('a')

plt.subplot(142)
plt.imshow(mask, cmap=plt.cm.gray)
plt.axis('off')
plt.title('mask')

plt.subplot(143)
plt.imshow(opened_mask, cmap=plt.cm.gray)
plt.axis('off')
plt.title('opened_mask')

plt.subplot(144)
plt.imshow(closed_mask, cmap=plt.cm.gray)
plt.title('closed_mask')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.99)

plt.show()
```

Total running time of the script: (0 minutes 0.202 seconds)

3.12.12 Plot geometrical transformations on images

Demo geometrical transformations of images.

```
# Load some data
from scipy import misc
face = misc.face(gray=True)

# Apply a variety of transformations
from scipy import ndimage
from matplotlib import pyplot as plt
shifted_face = ndimage.shift(face, (50, 50))
shifted_face2 = ndimage.shift(face, (50, 50), mode='nearest')
rotated_face = ndimage.rotate(face, 30)
cropped_face = face[50:-50, 50:-50]
zoomed_face = ndimage.zoom(face, 2)
zoomed_face.shape

plt.figure(figsize=(15, 3))
plt.subplot(151)
plt.imshow(shifted_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(152)
plt.imshow(shifted_face2, cmap=plt.cm.gray)
plt.axis('off')
```

```
plt.subplot(153)
plt.imshow(rotated_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(154)
plt.imshow(cropped_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(155)
plt.imshow(zoomed_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.99)

plt.show()
```

Total running time of the script: (0 minutes 1.133 seconds)

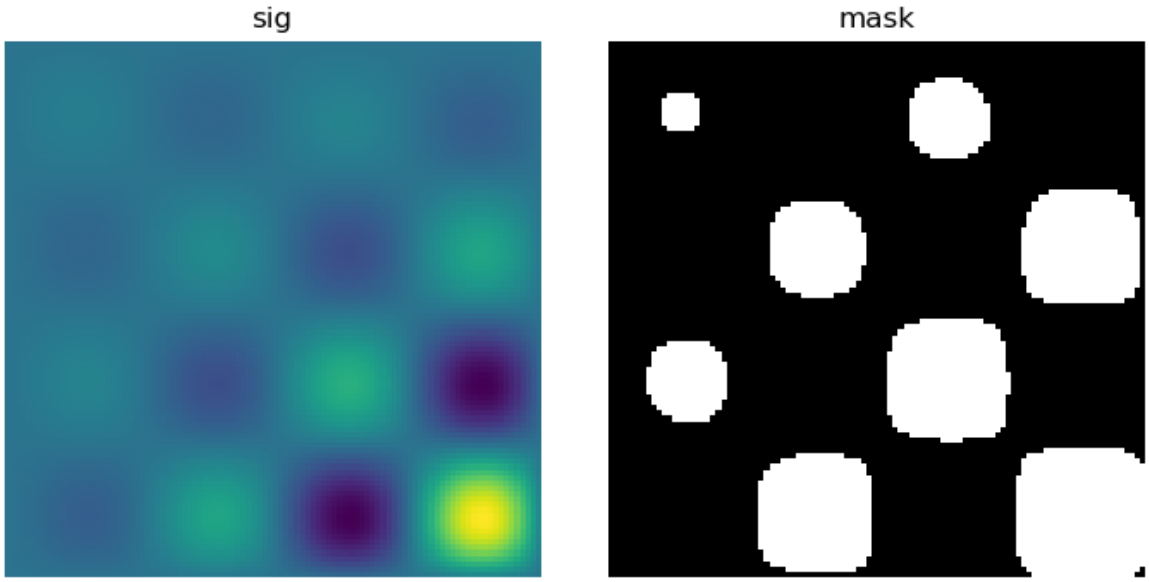
3.12.13 Demo connected components

Extracting and labeling connected components in a 2D array

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

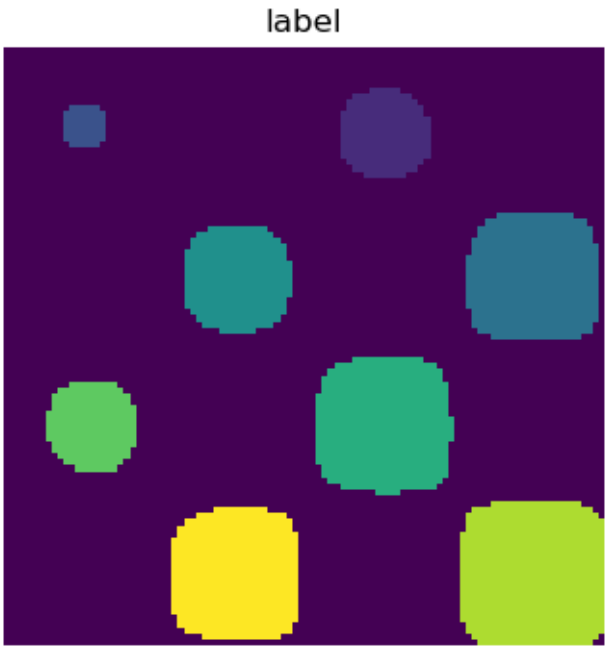
Generate some binary data

```
np.random.seed(0)
x, y = np.indices((100, 100))
sig = np.sin(2*np.pi*x/50.) * np.sin(2*np.pi*y/50.) * (1+x*y/50.**2)**2
mask = sig > 1

plt.figure(figsize=(7, 3.5))
plt.subplot(1, 2, 1)
plt.imshow(sig)
plt.axis('off')
plt.title('sig')

plt.subplot(1, 2, 2)
plt.imshow(mask, cmap=plt.cm.gray)
plt.axis('off')
plt.title('mask')
plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.9)
```



Label connected components

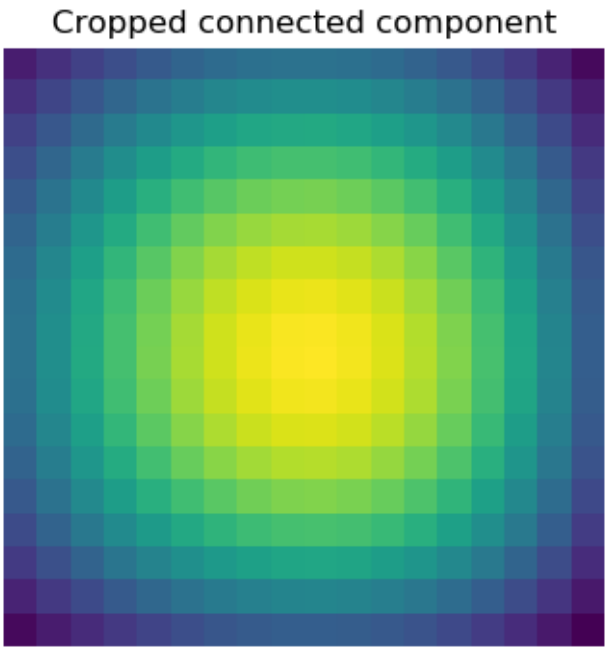
```
from scipy import ndimage
labels, nb = ndimage.label(mask)

plt.figure(figsize=(3.5, 3.5))
plt.imshow(labels)
plt.title('label')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.9)
```



Extract the 4th connected component, and crop the array around it

```
sl = ndimage.find_objects(labels==4)
plt.figure(figsize=(3.5, 3.5))
plt.imshow(sig[sl[0]])
plt.title('Cropped connected component')
```

```
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.9)

plt.show()
```



Total running time of the script: (0 minutes 0.264 seconds)

3.12.14 Minima and roots of a function

Demos finding minima and roots of a function.

Define the function

```
import numpy as np

x = np.arange(-10, 10, 0.1)
def f(x):
    return x**2 + 10*np.sin(x)
```

Find minima

```
from scipy import optimize

# Global optimization
grid = (-10, 10, 0.1)
xmin_global = optimize.brute(f, (grid, ))
print("Global minima found %s" % xmin_global)

# Constrain optimization
xmin_local = optimize.fminbound(f, 0, 10)
print("Local minimum found %s" % xmin_local)
```

Out:

```
Global minima found [-1.30641113]
Local minimum found 3.8374671195
```

Root finding

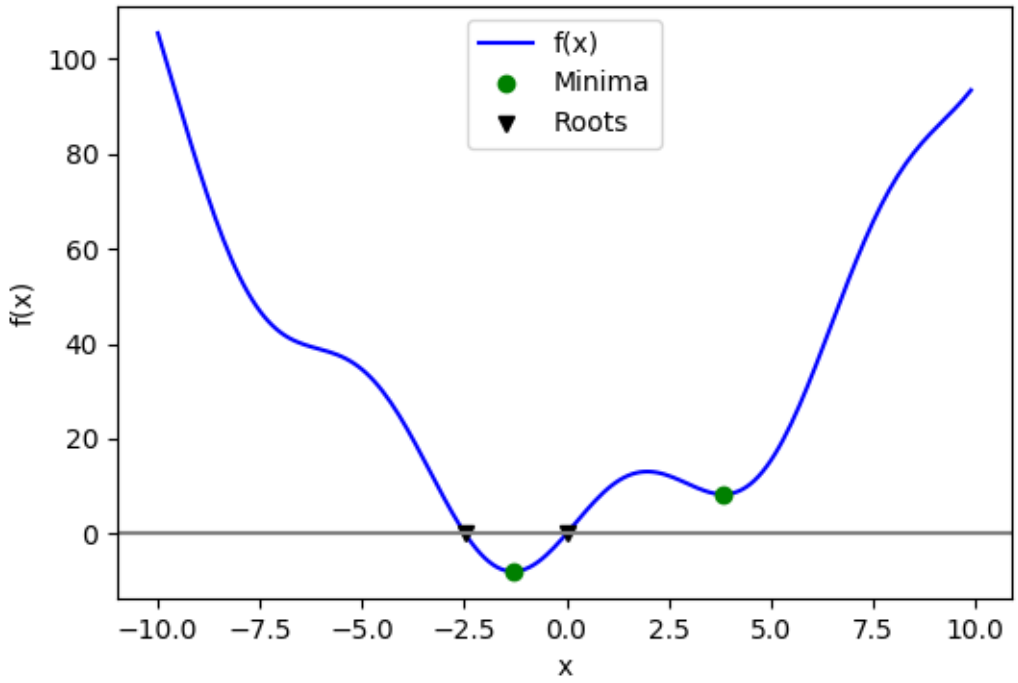
```
root = optimize.root(f, 1)  # our initial guess is 1
print("First root found %s" % root.x)
root2 = optimize.root(f, -2.5)
print("Second root found %s" % root2.x)
```

Out:

```
First root found [ 0.]
Second root found [-2.47948183]
```

Plot function, minima, and roots

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot(111)

# Plot the function
ax.plot(x, f(x), 'b-', label="f(x)")

# Plot the minima
xmins = np.array([xmin_global[0], xmin_local])
ax.plot(xmins, f(xmins), 'go', label="Minima")

# Plot the roots
roots = np.array([root.x, root2.x])
ax.plot(roots, f(roots), 'kv', label="Roots")

# Decorate the figure
ax.legend(loc='best')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.axhline(0, color='gray')
plt.show()
```



Total running time of the script: (0 minutes 0.062 seconds)

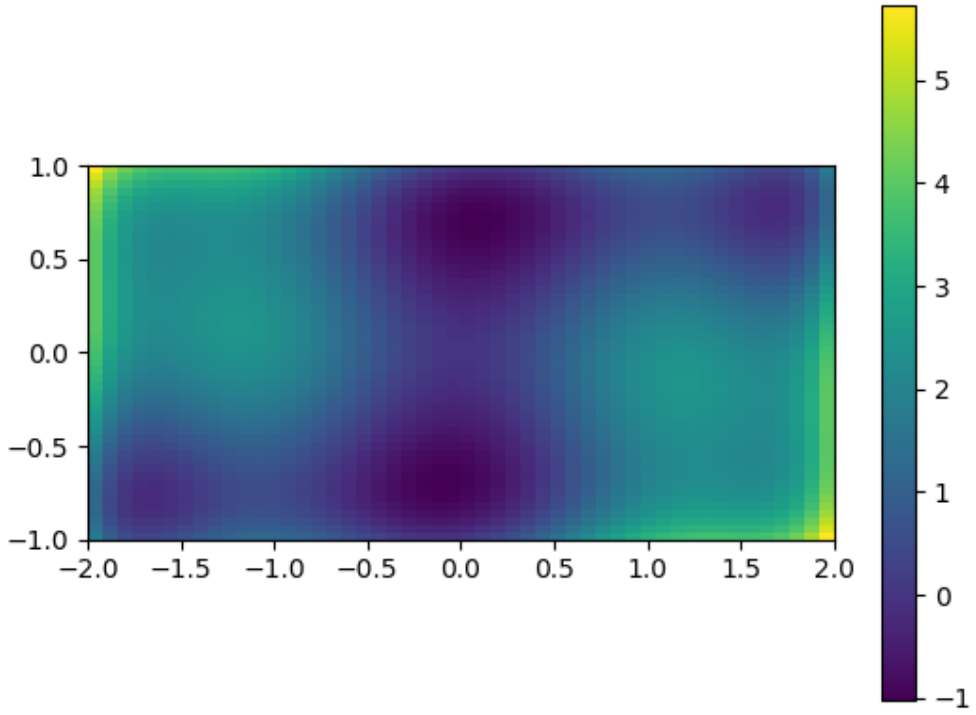
3.12.15 Optimization of a two-parameter function

```
import numpy as np


# Define the function that we are interested in
def sixhump(x):
    return ((4 - 2.1*x[0]**2 + x[0]**4 / 3.) * x[0]**2 + x[0] * x[1]
            + (-4 + 4*x[1]**2) * x[1] **2)

# Make a grid to evaluate the function (for plotting)
x = np.linspace(-2, 2)
y = np.linspace(-1, 1)
xg, yg = np.meshgrid(x, y)
```

A 2D image plot of the function

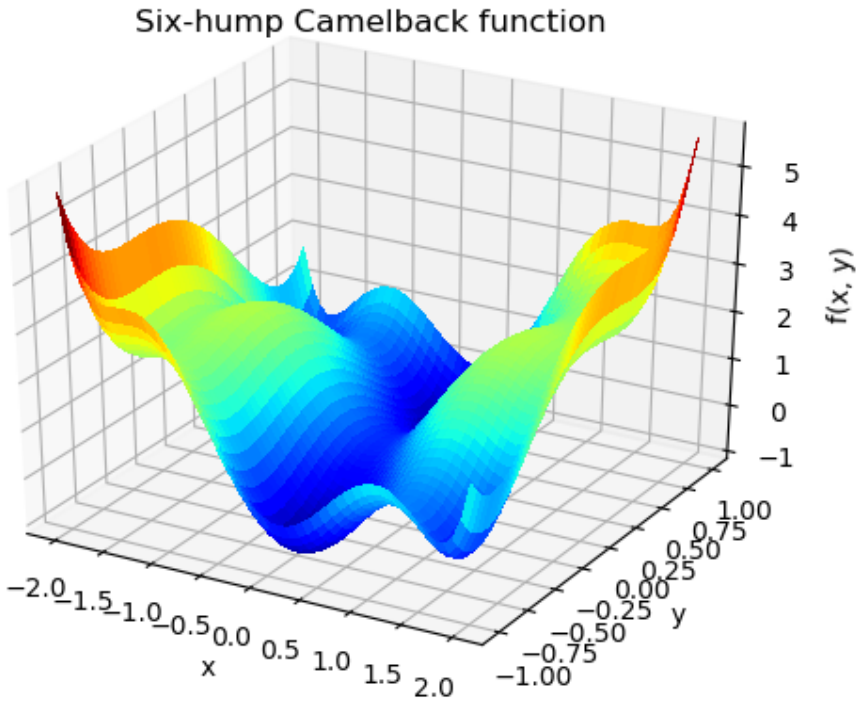
Simple visualization in 2D

```
import matplotlib.pyplot as plt
plt.figure()
plt.imshow(sixhump([xg, yg]), extent=[-2, 2, -1, 1])
plt.colorbar()
```



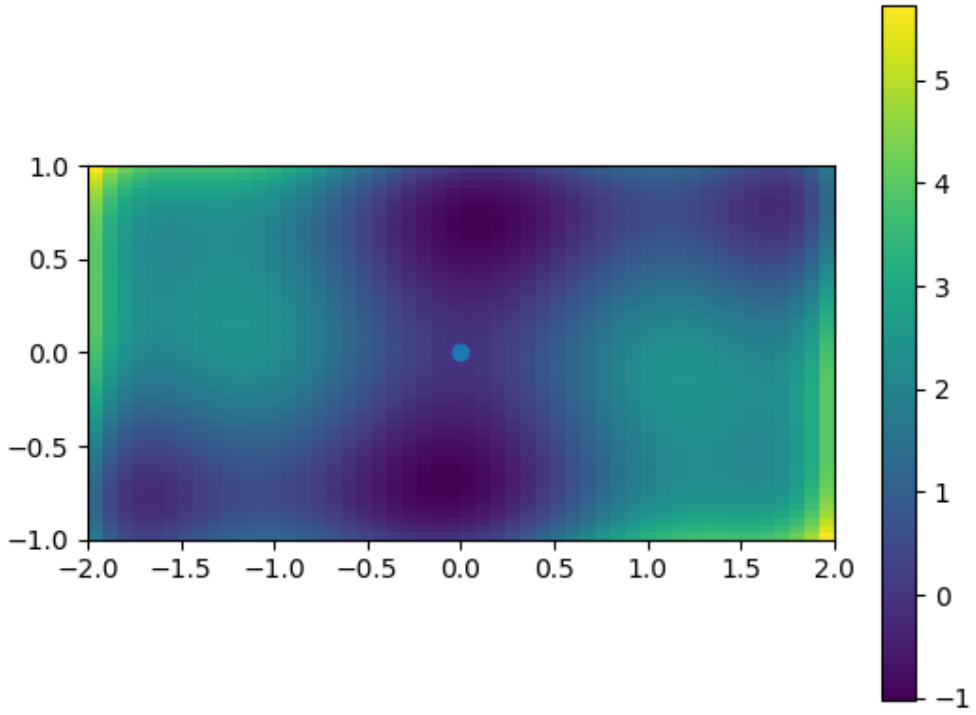
A 3D surface plot of the function

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(xg, yg, sixhump([xg, yg]), rstride=1, cstride=1,
                       cmap=plt.cm.jet, linewidth=0, antialiased=False)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.set_title('Six-hump Camelback function')
```



Find the minima

```
from scipy import optimize

x_min = optimize.minimize(sixhump, x0=[0, 0])

plt.figure()
# Show the function in 2D
plt.imshow(sixhump([xg, yg]), extent=[-2, 2, -1, 1])
plt.colorbar()
# And the minimum that we've found:
plt.scatter(x_min.x[0], x_min.x[1])

plt.show()
```



Total running time of the script: (0 minutes 0.388 seconds)

3.12.16 Plot filtering on images

Demo filtering for denoising of images.


```
# Load some data
from scipy import misc
face = misc.face(gray=True)
face = face[:512, -512:]  # crop out square on right

# Apply a variety of filters
from scipy import ndimage
from scipy import signal
from matplotlib import pyplot as plt

import numpy as np
```

```
noisy_face = np.copy(face).astype(np.float)
noisy_face += face.std() * 0.5 * np.random.standard_normal(face.shape)
blurred_face = ndimage.gaussian_filter(noisy_face, sigma=3)
median_face = ndimage.median_filter(noisy_face, size=5)
wiener_face = signal.wiener(noisy_face, (5, 5))

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
plt.imshow(noisy_face, cmap=plt.cm.gray)
plt.axis('off')
plt.title('noisy')

plt.subplot(142)
plt.imshow(blurred_face, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Gaussian filter')

plt.subplot(143)
plt.imshow(median_face, cmap=plt.cm.gray)
plt.axis('off')
plt.title('median filter')

plt.subplot(144)
plt.imshow(wiener_face, cmap=plt.cm.gray)
plt.title('Wiener filter')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.99)

plt.show()
```

Total running time of the script: (0 minutes 0.653 seconds)

3.12.17 Plotting and manipulating FFTs for filtering

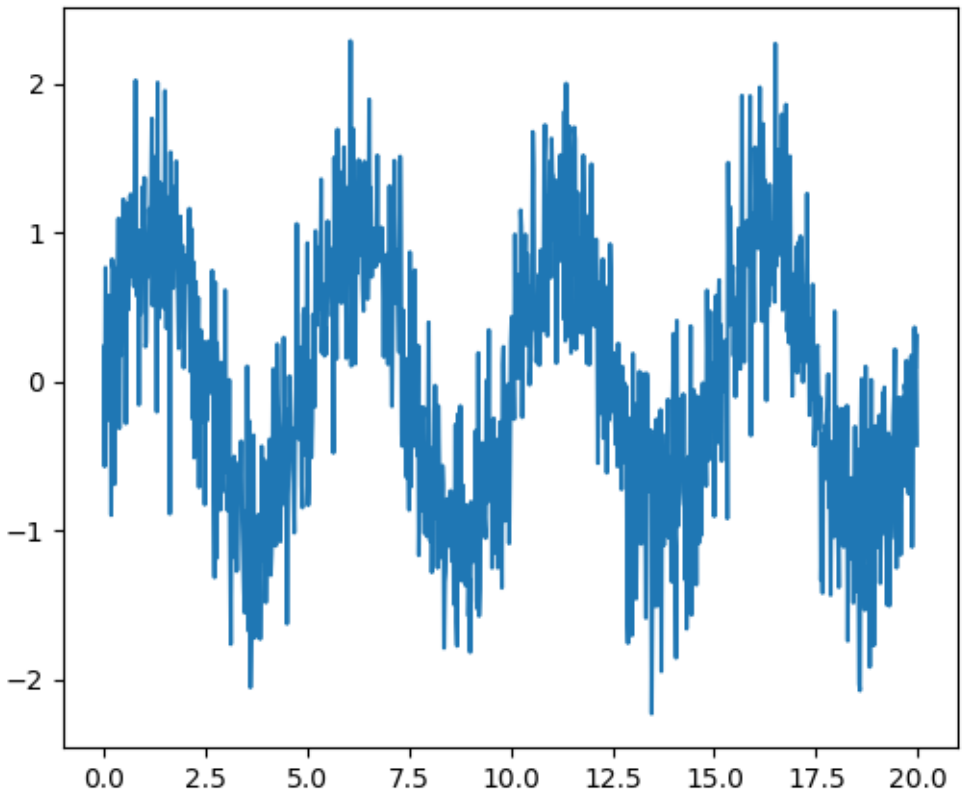
Plot the power of the FFT of a signal and inverse FFT back to reconstruct a signal.

This example demonstrate `scipy.fftpack.fft()`, `scipy.fftpack.fftfreq()` and `scipy.fftpack.ifft()`. It implements a basic filter that is very suboptimal, and should not be used.

```
import numpy as np
from scipy import fftpack
from matplotlib import pyplot as plt
```

Generate the signal

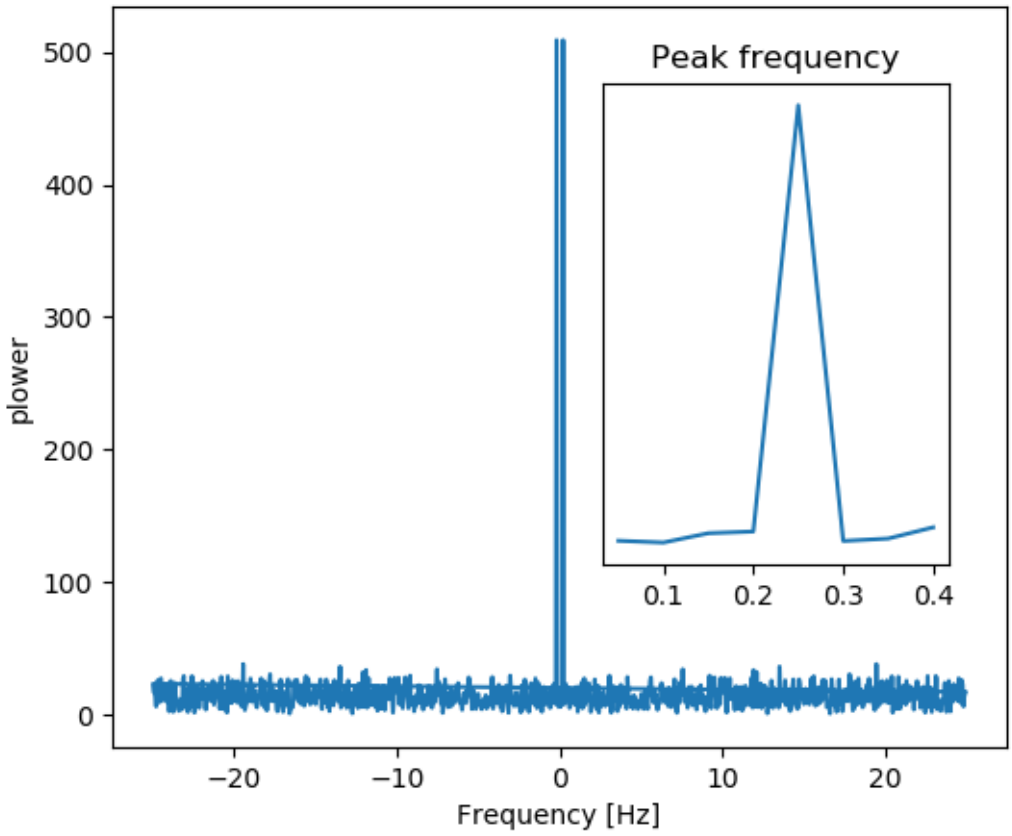
```
# Seed the random number generator
np.random.seed(1234)

time_step = 0.02
period = 5.

time_vec = np.arange(0, 20, time_step)
sig = (np.sin(2 * np.pi / period * time_vec)
       + 0.5 * np.random.randn(time_vec.size))

plt.figure(figsize=(6, 5))
plt.plot(time_vec, sig, label='Original signal')
```



Compute and plot the power

```
# The FFT of the signal
sig_fft = fftpack.fft(sig)

# And the power (sig_fft is of complex dtype)
power = np.abs(sig_fft)

# The corresponding frequencies
sample_freq = fftpack.fftfreq(sig.size, d=time_step)

# Plot the FFT power
plt.figure(figsize=(6, 5))
plt.plot(sample_freq, power)
plt.xlabel('Frequency [Hz]')
plt.ylabel('plower')

# Find the peak frequency: we can focus on only the positive frequencies
pos_mask = np.where(sample_freq > 0)
freqs = sample_freq[pos_mask]
peak_freq = freqs[power[pos_mask].argmax()]

# Check that it does indeed correspond to the frequency that we generate
# the signal with
np.allclose(peak_freq, 1./period)

# An inner plot to show the peak frequency
axes = plt.axes([0.55, 0.3, 0.3, 0.5])
```

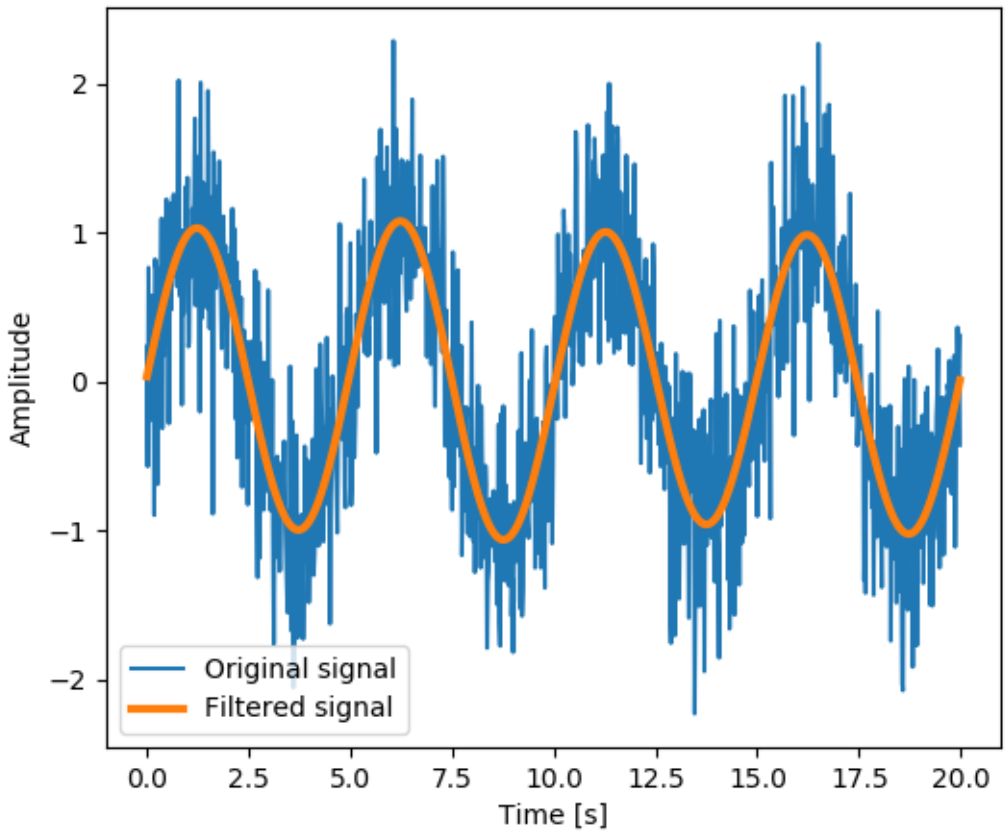
```
plt.title('Peak frequency')
plt.plot(freqs[:8], power[:8])
plt.setp(axes, yticks=[])

# scipy.signal.find_peaks_cwt can also be used for more advanced
# peak detection
```



Remove all the high frequencies

We now remove all the high frequencies and transform back from frequencies to signal.

```
high_freq_fft = sig_fft.copy()
high_freq_fft[np.abs(sample_freq) > peak_freq] = 0
filtered_sig = fftpack.ifft(high_freq_fft)

plt.figure(figsize=(6, 5))
plt.plot(time_vec, sig, label='Original signal')
plt.plot(time_vec, filtered_sig, linewidth=3, label='Filtered signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')

plt.legend(loc='best')
```



Note This is actually a bad way of creating a filter: such brutal cut-off in frequency space does not control distorsion on the signal.

Filters should be created using the scipy filter design code

```
plt.show()
```

Total running time of the script: (0 minutes 0.210 seconds)

3.12.18 Solutions of the exercises for scipy

Crude periodicity finding

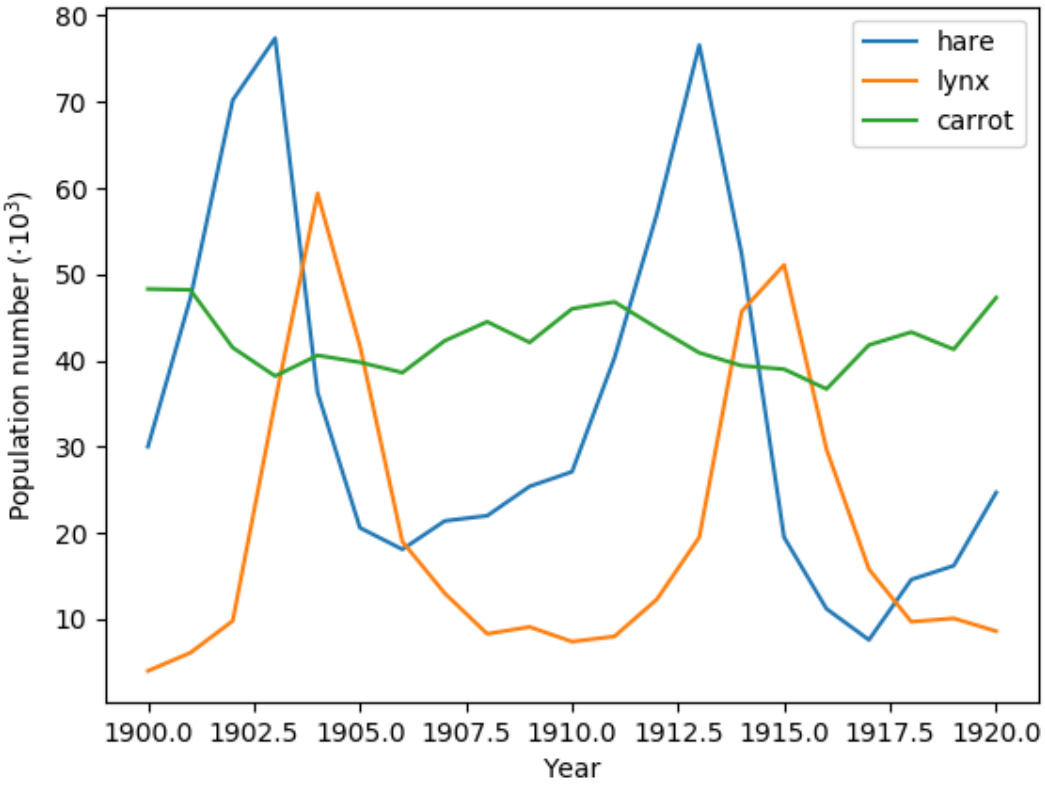
Discover the periods in evolution of animal populations (`../../data/populations.txt`)

Load the data

```
import numpy as np
data = np.loadtxt('../../../../data/populations.txt')
years = data[:, 0]
populations = data[:, 1:]
```

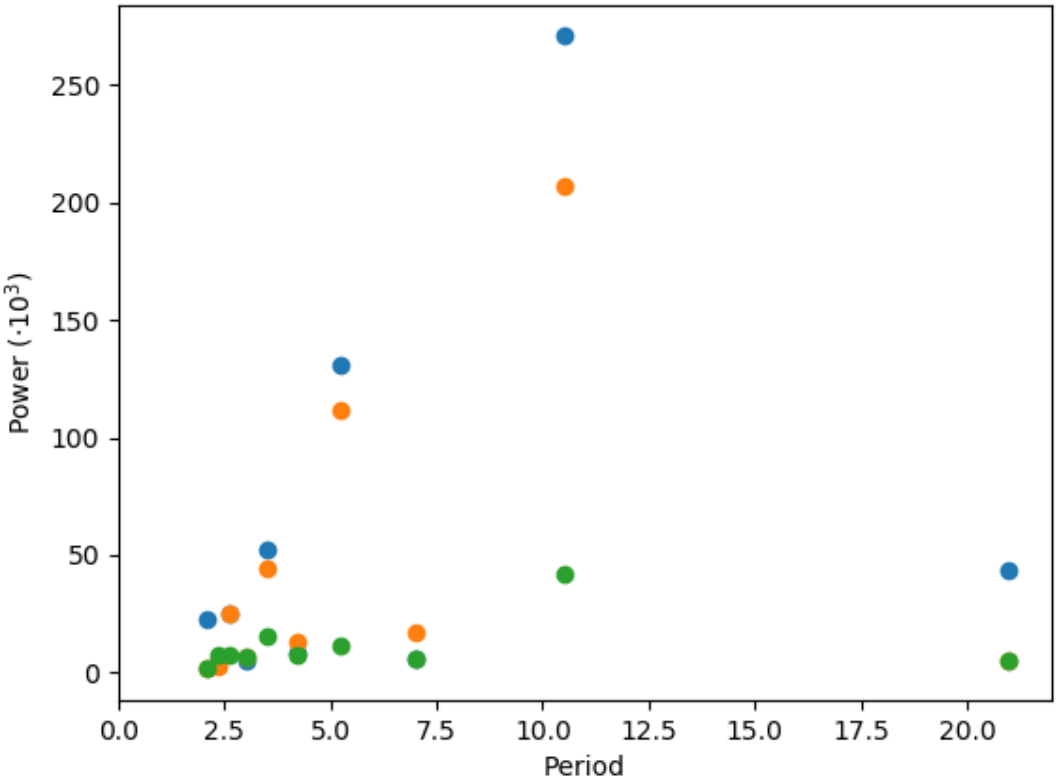
Plot the data

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(years, populations * 1e-3)
plt.xlabel('Year')
plt.ylabel('Population number ($\cdot10^3$)')
plt.legend(['hare', 'lynx', 'carrot'], loc=1)
```



Plot its periods

```
from scipy import fftpack

ft_populations = fftpack.fft(populations, axis=0)
frequencies = fftpack.fftfreq(populations.shape[0], years[1] - years[0])
periods = 1 / frequencies

plt.figure()
plt.plot(periods, abs(ft_populations) * 1e-3, 'o')
plt.xlim(0, 22)
plt.xlabel('Period')
plt.ylabel('Power ($\cdot10^3$)')

plt.show()
```



There's probably a period of around 10 years (obvious from the plot), but for this crude a method, there's not enough data to say much more.

Total running time of the script: (0 minutes 0.109 seconds)

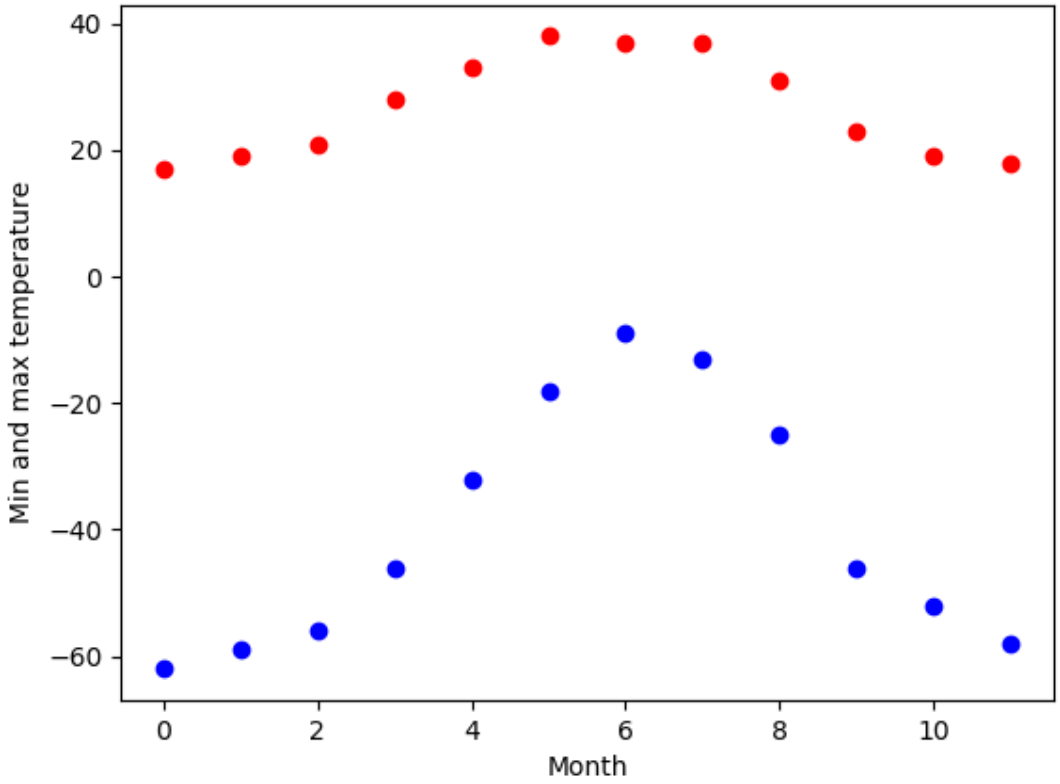
Curve fitting: temperature as a function of month of the year

We have the min and max temperatures in Alaska for each months of the year. We would like to find a function to describe this yearly evolution.

For this, we will fit a periodic function.

The data

```
import numpy as np

temp_max = np.array([17,  19,  21,  28,  33,  38, 37,  37,  31,  23,  19,  18])
temp_min = np.array([-62, -59, -56, -46, -32, -18, -9, -13, -25, -46, -52, -58])

import matplotlib.pyplot as plt
months = np.arange(12)
plt.plot(months, temp_max, 'ro')
plt.plot(months, temp_min, 'bo')
plt.xlabel('Month')
plt.ylabel('Min and max temperature')
```

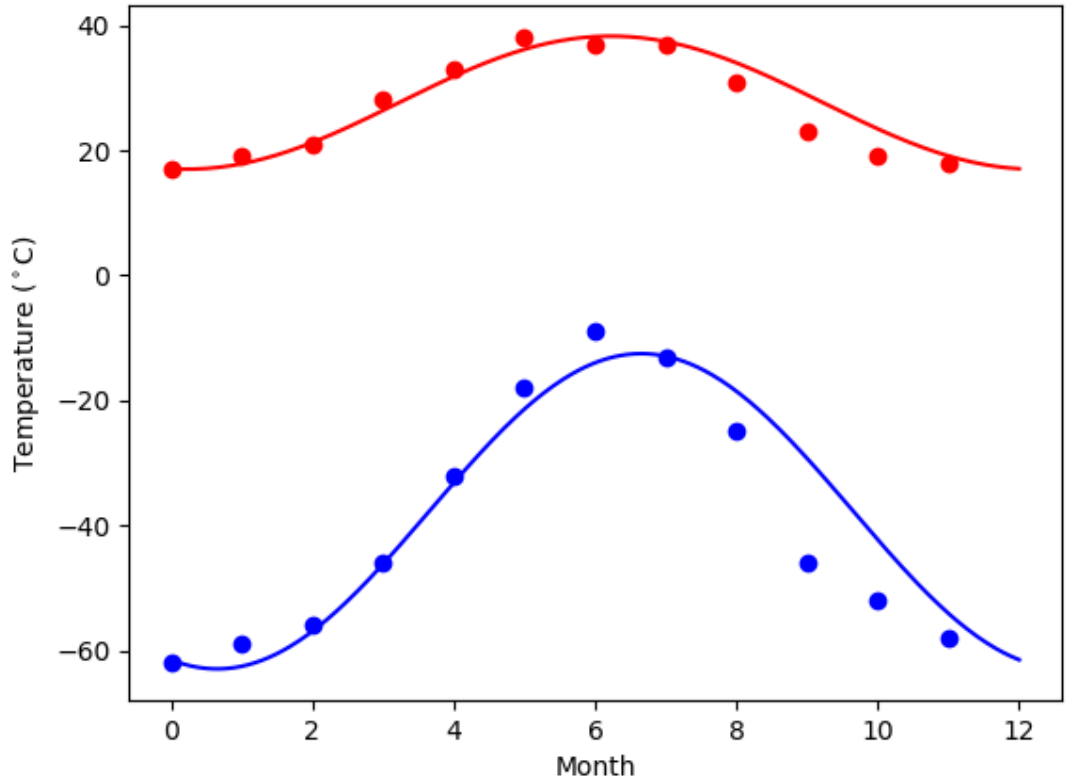


Fitting it to a periodic function

```
from scipy import optimize
def yearly_temps(times, avg, ampl, time_offset):
    return (avg
            + ampl * np.cos((times + time_offset) * 2 * np.pi / times.max()))

res_max, cov_max = optimize.curve_fit(yearly_temps, months,
                                      temp_max, [20, 10, 0])
res_min, cov_min = optimize.curve_fit(yearly_temps, months,
                                      temp_min, [-40, 20, 0])
```

Plotting the fit

```
days = np.linspace(0, 12, num=365)

plt.figure()
plt.plot(months, temp_max, 'ro')
plt.plot(days, yearly_temps(days, *res_max), 'r-')
plt.plot(months, temp_min, 'bo')
plt.plot(days, yearly_temps(days, *res_min), 'b-')
plt.xlabel('Month')
plt.ylabel('Temperature ($^\circ$C)')

plt.show()
```



Total running time of the script: (0 minutes 0.109 seconds)

Simple image blur by convolution with a Gaussian kernel

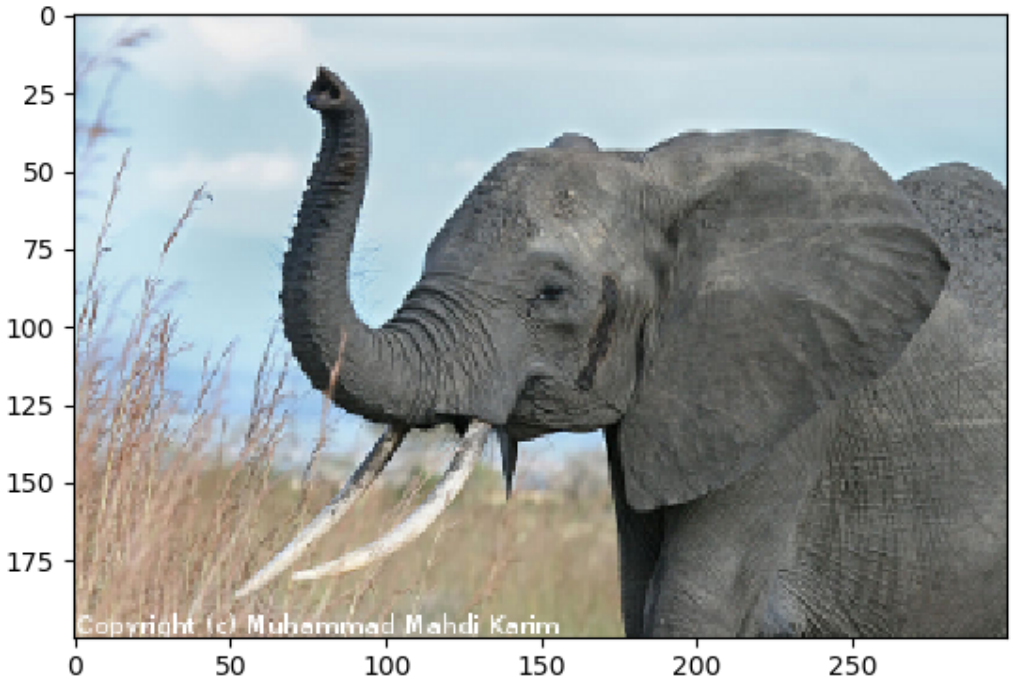
Blur an an image (`../../../../data/elephant.png`) using a Gaussian kernel.

Convolution is easy to perform with FFT: convolving two signals boils down to multiplying their FFTs (and performing an inverse FFT).

```
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt
```

The original image

```
# read image
img = plt.imread('../../../../data/elephant.png')
plt.figure()
plt.imshow(img)
```



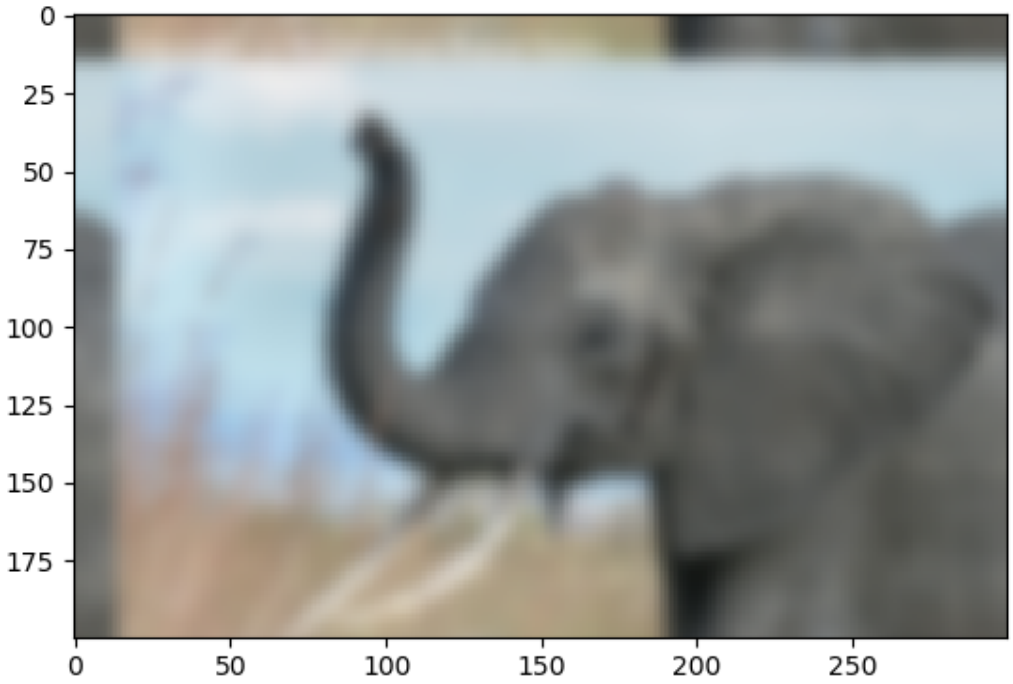
Prepare an Gaussian convolution kernel

```
# First a 1-D  Gaussian
t = np.linspace(-10, 10, 30)
bump = np.exp(-0.1*t**2)
bump /= np.trapz(bump) # normalize the integral to 1

# make a 2-D kernel out of it
kernel = bump[:, np.newaxis] * bump[np.newaxis, :]
```

Implement convolution via FFT

```
# Padded fourier transform, with the same shape as the image
# We use :func:`scipy.signal.fftpack.fft2` to have a 2D FFT
kernel_ft = fftpack.fft2(kernel, shape=img.shape[:2], axes=(0, 1))

# convolve
img_ft = fftpack.fft2(img, axes=(0, 1))
# the 'newaxis' is to match to color direction
img2_ft = kernel_ft[:, :, np.newaxis] * img_ft
img2 = fftpack.ifft2(img2_ft, axes=(0, 1)).real

# clip values to range
img2 = np.clip(img2, 0, 1)

# plot output
plt.figure()
```

```
plt.imshow(img2)
```



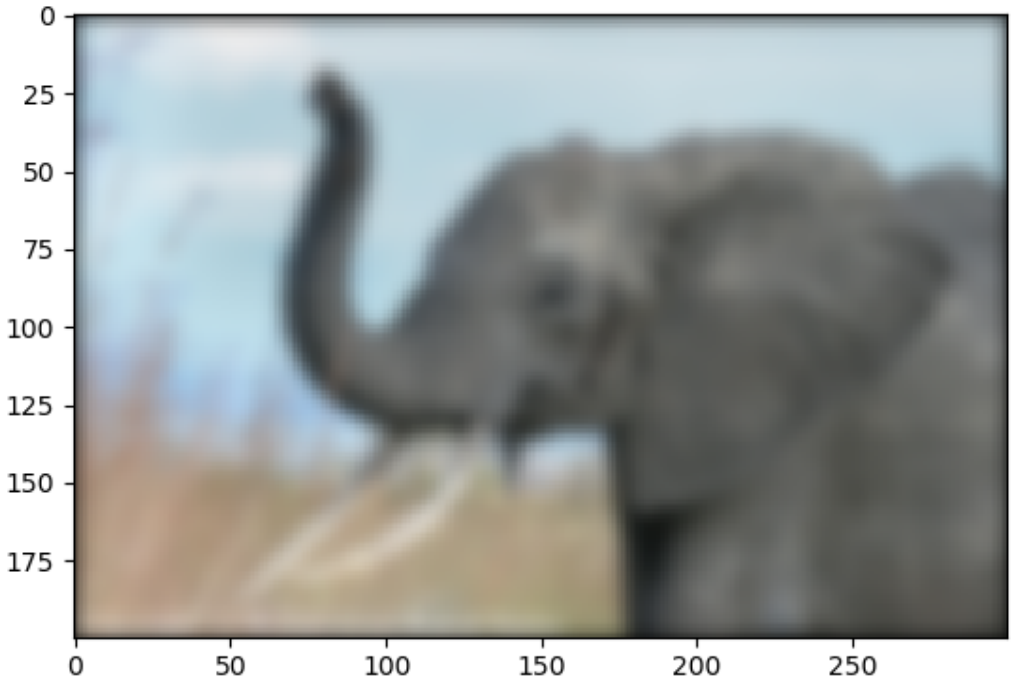
Further exercise (only if you are familiar with this stuff):

A "wrapped border" appears in the upper left and top edges of the image. This is because the padding is not done correctly, and does not take the kernel size into account (so the convolution "flows out of bounds of the image"). Try to remove this artifact.

A function to do it: `scipy.signal.fftconvolve()`

The above exercise was only for didactic reasons: there exists a function in scipy that will do this for us, and probably do a better job: `scipy.signal.fftconvolve()`

```
from scipy import signal
# mode='same' is there to enforce the same output shape as input arrays
# (ie avoid border effects)
img3 = signal.fftconvolve(img, kernel[:, :, np.newaxis], mode='same')
plt.figure()
plt.imshow(img3)
```



Note that we still have a decay to zero at the border of the image. Using `scipy.ndimage.gaussian_filter()` would get rid of this artifact

```
plt.show()
```

Total running time of the script: (0 minutes 0.200 seconds)

Image denoising by FFT

Denoise an image (`../../../../data/moonlanding.png`) by implementing a blur with an FFT.

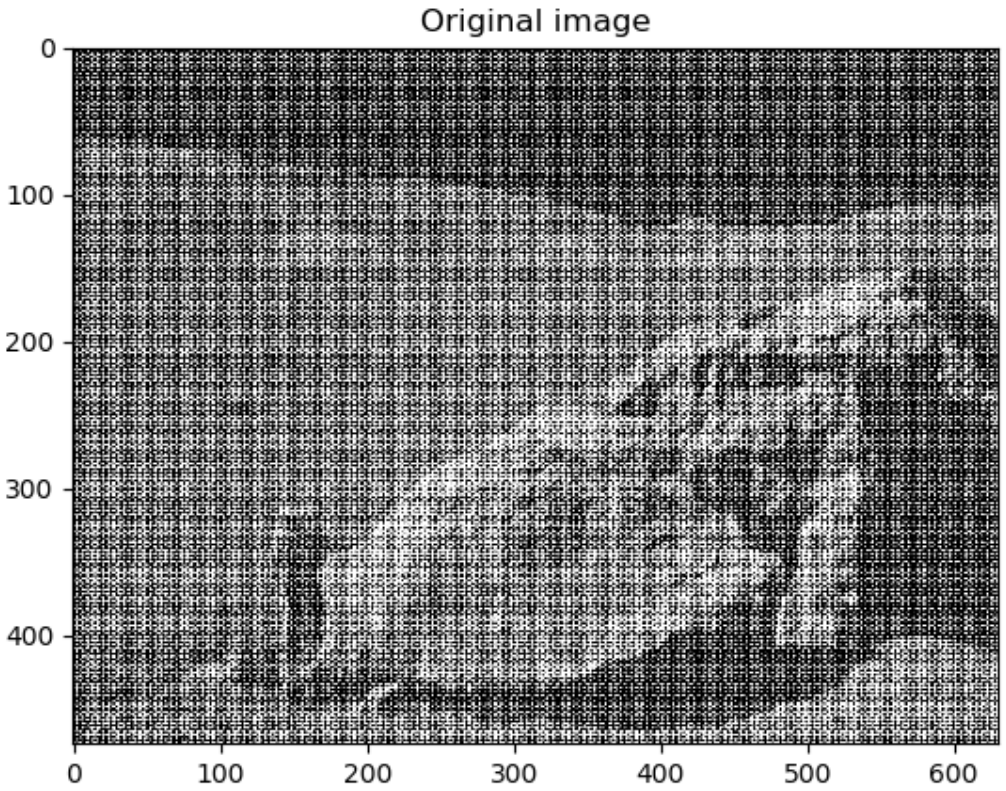
Implements, via FFT, the following convolution:

$$f_1(t) = \int dt' \, K(t-t') f_0(t')$$

$$\tilde{f}_1(\omega) = \tilde{K}(\omega) \tilde{f}_0(\omega)$$

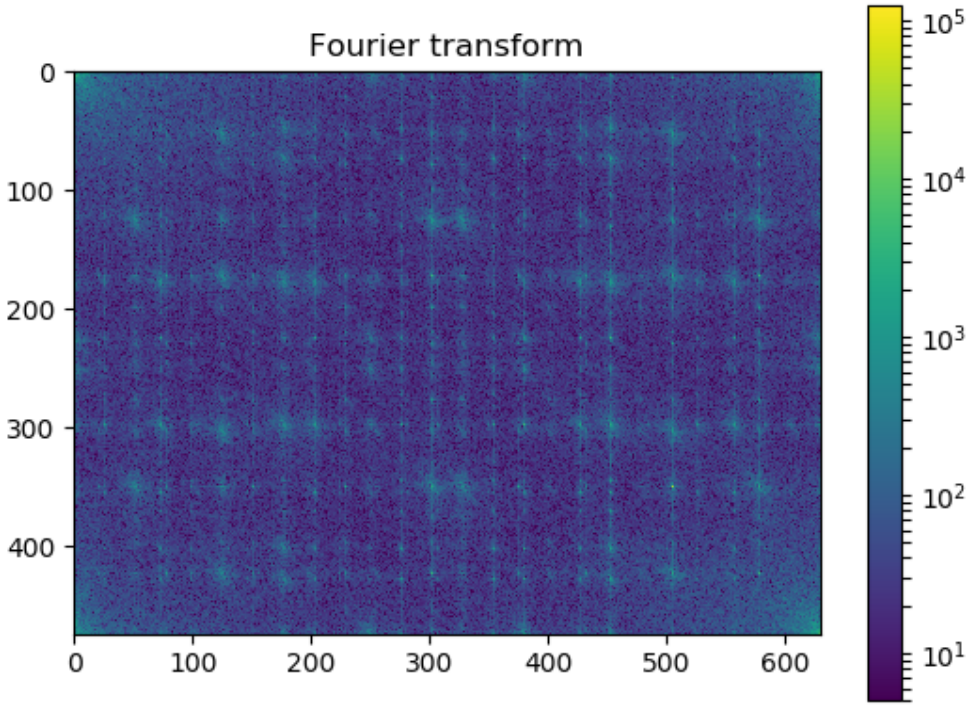
Read and plot the image

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

im = plt.imread('../../../../data/moonlanding.png').astype(float)

plt.figure()
plt.imshow(im, plt.cm.gray)
plt.title('Original image')
```



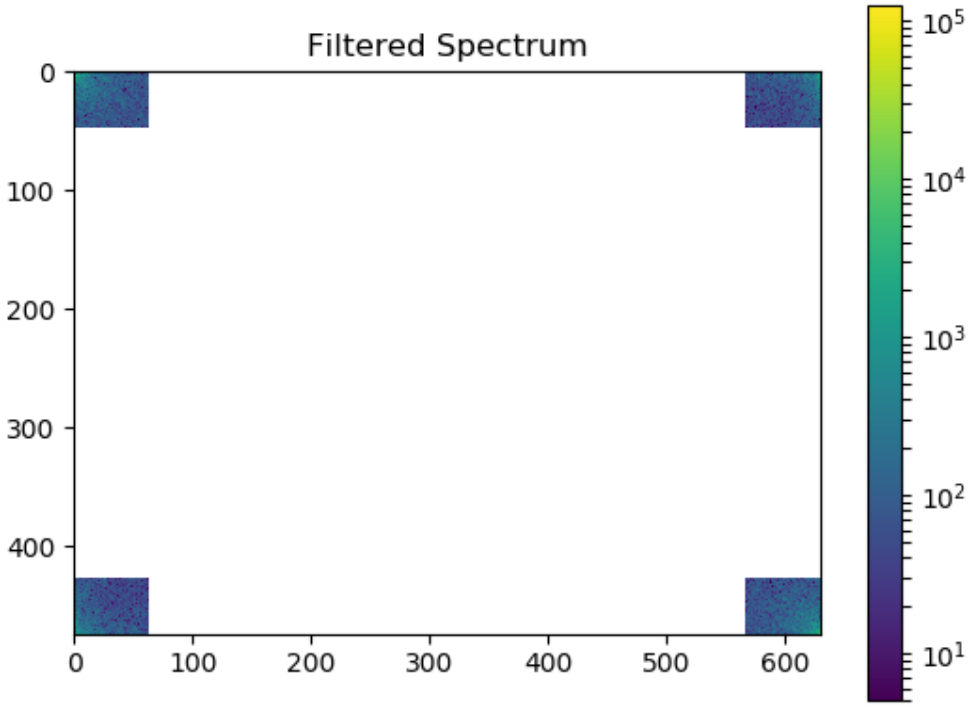
Compute the 2d FFT of the input image

```
from scipy import fftpack
im_fft = fftpack.fft2(im)

# Show the results

def plot_spectrum(im_fft):
    from matplotlib.colors import LogNorm
    # A logarithmic colormap
    plt.imshow(np.abs(im_fft), norm=LogNorm(vmin=5))
    plt.colorbar()

plt.figure()
plot_spectrum(im_fft)
plt.title('Fourier transform')
```



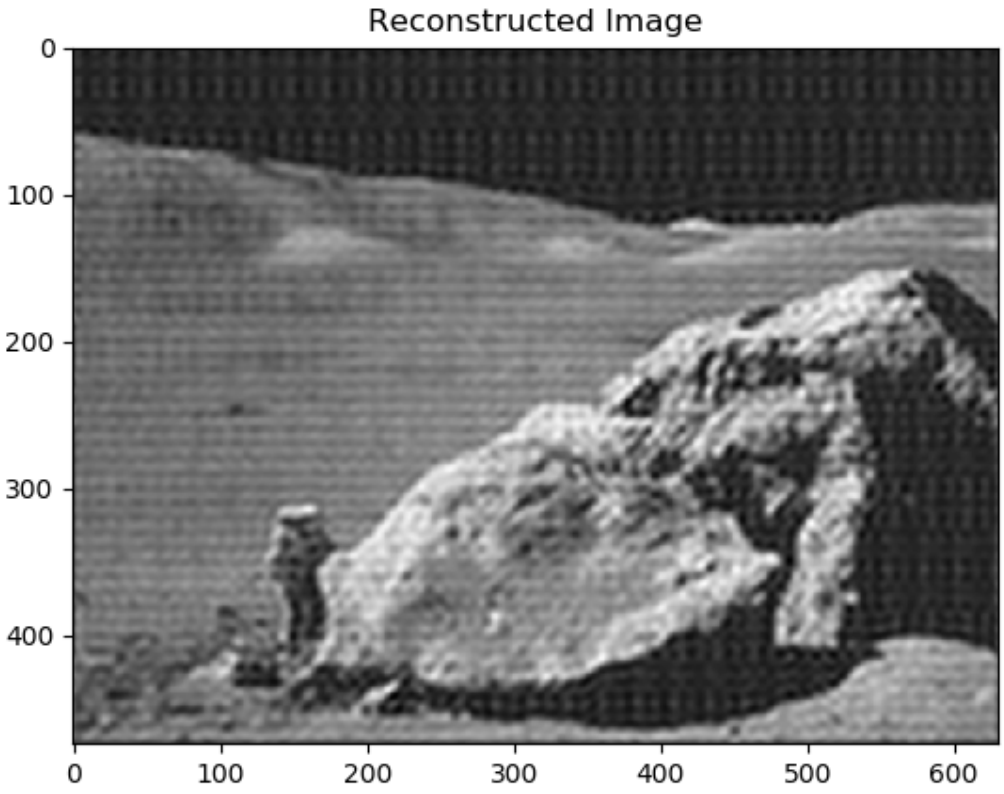
Filter in FFT

```
# In the lines following, we'll make a copy of the original spectrum and
# truncate coefficients.

# Define the fraction of coefficients (in each direction) we keep
keep_fraction = 0.1

# Call ff a copy of the original transform. Numpy arrays have a copy
# method for this purpose.
im_fft2 = im_fft.copy()

# Set r and c to be the number of rows and columns of the array.
r, c = im_fft2.shape

# Set to zero all rows with indices between r*keep_fraction and
# r*(1-keep_fraction):
im_fft2[int(r*keep_fraction):int(r*(1-keep_fraction))] = 0

# Similarly with the columns:
im_fft2[:, int(c*keep_fraction):int(c*(1-keep_fraction))] = 0

plt.figure()
plot_spectrum(im_fft2)
plt.title('Filtered Spectrum')
```



Reconstruct the final image

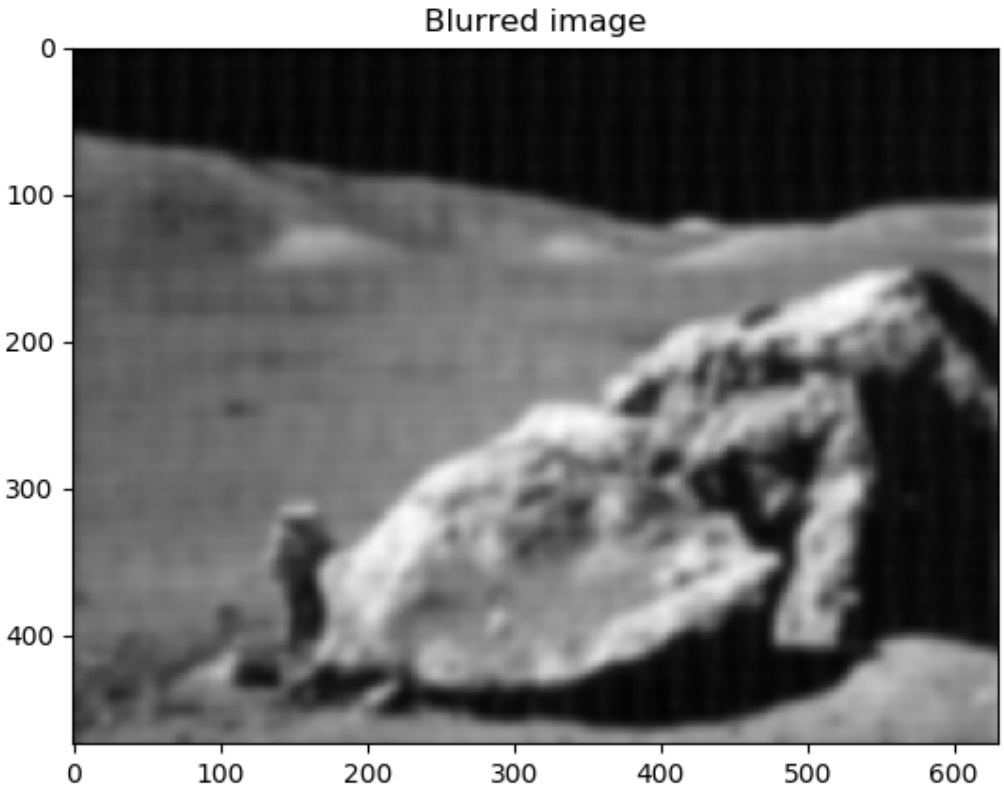
```
# Reconstruct the denoised image from the filtered spectrum, keep only the
# real part for display.
im_new = fftpack.ifft2(im_fft2).real

plt.figure()
plt.imshow(im_new, plt.cm.gray)
plt.title('Reconstructed Image')
```



Easier and better: `scipy.ndimage.gaussian_filter()`

Implementing filtering directly with FFTs is tricky and time consuming. We can use the Gaussian filter from `scipy.ndimage`

```
from scipy import ndimage
im_blur = ndimage.gaussian_filter(im, 4)

plt.figure()
plt.imshow(im_blur, plt.cm.gray)
plt.title('Blurred image')

plt.show()
```



Total running time of the script: (0 minutes 0.620 seconds)

See also:

References to go further

- Some chapters of the advanced and the packages and applications parts of the scipy lectures
- The scipy cookbook

CHAPTER 4

Getting help and finding documentation

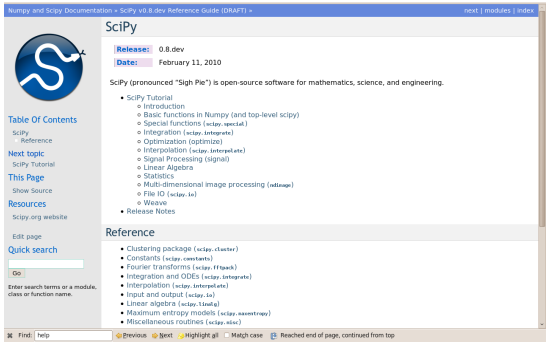
Author: *Emmanuelle Gouillart*

Rather than knowing all functions in Numpy and Scipy, it is important to find rapidly information throughout the documentation and the available help. Here are some ways to get information:

- In Ipython, `help function` opens the docstring of the function. Only type the beginning of the function's name and use tab completion to display the matching functions.

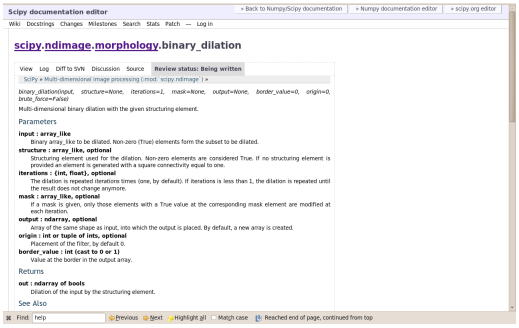
```
In [204]: help np.v
np.vander     np.vdot       np.version    np.void0      np.vstack
np.var        np.vectorize  np.void       np.vsplit

In [204]: help np.vander
```

In Ipython it is not possible to open a separated window for help and documentation; however one can always open a second `Ipython` shell just to display help and docstrings...

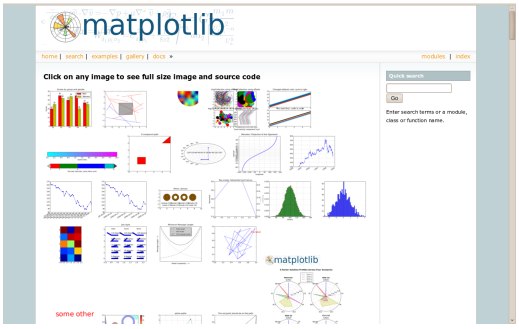


- Numpy's and Scipy's documentations can be browsed online on http://docs.scipy.org/doc. The `search` button is quite useful inside the reference documentation of the two packages (http://docs.scipy.org/doc/numpy/reference/ and http://docs.scipy.org/doc/scipy/reference/).

Tutorials on various topics as well as the complete API with all docstrings are found on this website.



- Numpy's and Scipy's documentation is enriched and updated on a regular basis by users on a wiki http://docs.scipy.org/doc/numpy/. As a result, some docstrings are clearer or more detailed on the wiki, and you may want to read directly the documentation on the wiki instead of the official documentation website. Note that anyone can create an account on the wiki and write better documentation; this is an easy way to contribute to an open-source project and improve the tools you are using!
- Scipy central http://central.scipy.org/ gives recipes on many common problems frequently encountered, such as fitting data points, solving ODE, etc.



- Matplotlib's website http://matplotlib.org/ features a very nice **gallery** with a large number of plots, each of them shows both the source code and the resulting plot. This is very useful for learning by example. More standard documentation is also available.

Finally, two more "technical" possibilities are useful as well:

- In Ipython, the magical function `%psearch` search for objects matching patterns. This is useful if, for example, one does not know the exact name of a function.

```
In [3]: import numpy as np
In [4]: %psearch np.diag*
np.diag
np.diagflat
np.diagonal
```

- numpy.lookfor looks for keywords inside the docstrings of specified modules.

```
In [45]: numpy.lookfor('convolution')
Search results for 'convolution'
--------------------------------
numpy.convolve
    Returns the discrete, linear convolution of two one-dimensional
sequences.
numpy.bartlett
    Return the Bartlett window.
numpy.correlate
    Discrete, linear correlation of two 1-dimensional sequences.
In [46]: numpy.lookfor('remove', module='os')
Search results for 'remove'
---------------------------
os.remove
```

```
    remove(path)
os.removedirs
    removedirs(path)
os.rmdir
    rmdir(path)
os.unlink
    unlink(path)
os.walk
    Directory tree generator.
```

- If everything listed above fails (and Google doesn't have the answer)… don't despair! Write to the mailing-list suited to your problem: you should have a quick answer if you describe your problem well. Experts on scientific python often give very enlightening explanations on the mailing-list.
  - **Numpy discussion** (numpy-discussion@scipy.org): all about numpy arrays, manipulating them, indexation questions, etc.
  - **SciPy Users List** (scipy-user@scipy.org): scientific computing with Python, high-level data processing, in particular with the scipy package.
  - matplotlib-users@lists.sourceforge.net for plotting with matplotlib.

CHAPTER 5

Debugging code

Author: *Gaël Varoquaux*

This section explores tools to understand better your code base: debugging, to find and fix bugs.

It is not specific to the scientific Python community, but the strategies that we will employ are tailored to its needs.

Prerequisites

- Numpy
- IPython
- nosetests
- pyflakes
- gdb for the C-debugging part.

Chapter contents

5.1 Avoiding bugs

5.1.1 Coding best practices to avoid getting in trouble

Brian Kernighan

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

- We all write buggy code. Accept it. Deal with it.
- Write your code with testing and debugging in mind.
- Keep It Simple, Stupid (KISS).
  - What is the simplest thing that could possibly work?
- Don't Repeat Yourself (DRY).
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
  - Constants, algorithms, etc...
- Try to limit interdependencies of your code. (Loose Coupling)
- Give your variables, functions and modules meaningful names (not mathematics names)

5.1.2 pyflakes: fast static analysis

They are several static analysis tools in Python; to name a few:

- pylint
- pychecker
- pyflakes
- flake8

Here we focus on *pyflakes*, which is the simplest tool.

- **Fast, simple**
- Detects syntax errors, missing imports, typos on names.

Another good recommendation is the *flake8* tool which is a combination of pyflakes and pep8. Thus, in addition to the types of errors that pyflakes catches, flake8 detects violations of the recommendation in PEP8 style guide.

Integrating pyflakes (or flake8) in your editor or IDE is highly recommended, it **does yield productivity gains**.

Running pyflakes on the current edited file

You can bind a key to run pyflakes in the current buffer.

- **In kate** Menu: 'settings -> configure kate

  - In plugins enable 'external tools'
  - In external Tools', add *pyflakes*:

```
kdialog --title "pyflakes %filename" --msgbox "$(pyflakes %filename)"
```

- **In TextMate**

  Menu: TextMate -> Preferences -> Advanced -> Shell variables, add a shell variable:

```
TM_PYCHECKER = /Library/Frameworks/Python.framework/Versions/Current/bin/pyflakes
```

  Then *Ctrl-Shift-V* is binded to a pyflakes report

- **In vim** In your *.vimrc* (binds F5 to *pyflakes*):

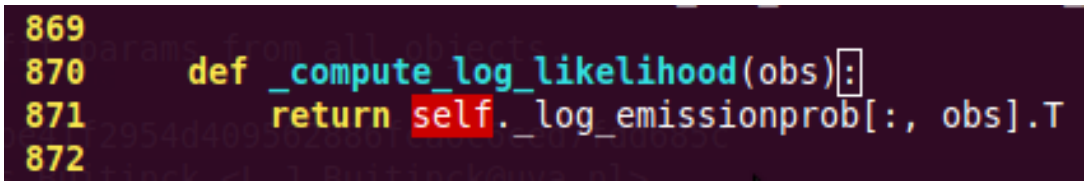
```
autocmd FileType python let &mp = 'echo "*** running % ***" ; pyflakes %'
autocmd FileType tex,mp,rst,python imap <Esc>[15~ <C-O>:make!^M
autocmd FileType tex,mp,rst,python map  <Esc>[15~ :make!^M
autocmd FileType tex,mp,rst,python set autowrite
```

- **In emacs** In your *.emacs* (binds F5 to *pyflakes*):

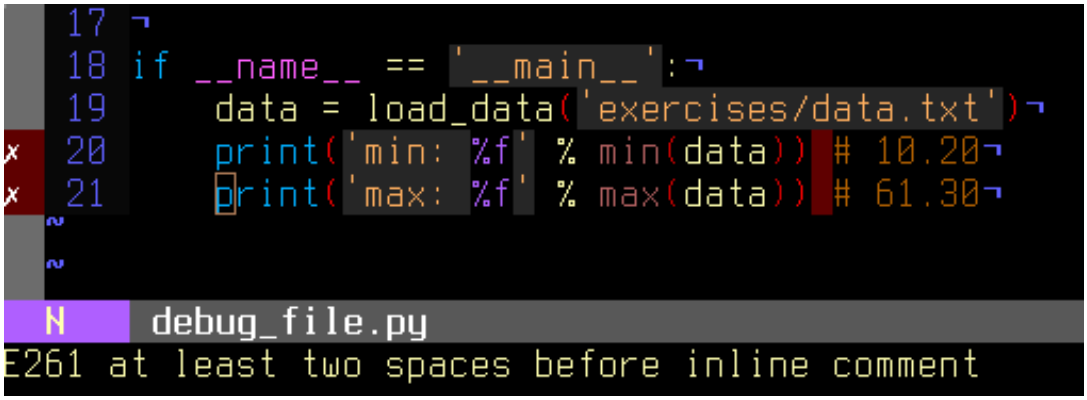
```
(defun pyflakes-thisfile () (interactive)
       (compile (format "pyflakes %s" (buffer-file-name)))
)

(define-minor-mode pyflakes-mode
    "Toggle pyflakes mode.
    With no argument, this command toggles the mode.
    Non-null prefix argument turns on the mode.
    Null prefix argument turns off the mode."
    ;; The initial value.
    nil
    ;; The indicator for the mode line.
    " Pyflakes"
    ;; The minor mode bindings.
    '( ([f5] . pyflakes-thisfile) )
)

(add-hook 'python-mode-hook (lambda () (pyflakes-mode t)))
```

A type-as-go spell-checker like integration

- **In vim**
  - Use the pyflakes.vim plugin:
    1. download the zip file from http://www.vim.org/scripts/script.php?script_id=2441
    2. extract the files in `~/.vim/ftplugin/python`
    3. make sure your vimrc has `filetype plugin indent on`



  - Alternatively: use the syntastic plugin. This can be configured to use `flake8` too and also handles on-the-fly checking for many other languages.



- **In emacs** Use the flymake mode with pyflakes, documented on http://www.plope.com/Members/chrism/flymake-mode : add the following to your .emacs file:

```
(when (load "flymake" t)
        (defun flymake-pyflakes-init ()
        (let* ((temp-file (flymake-init-create-temp-buffer-copy
                           'flymake-create-temp-inplace))
            (local-file (file-relative-name
                         temp-file
                         (file-name-directory buffer-file-name))))
            (list "pyflakes" (list local-file))))

        (add-to-list 'flymake-allowed-file-name-masks
                '("\\.py\\'" flymake-pyflakes-init)))

(add-hook 'find-file-hook 'flymake-find-file-hook)
```

5.2 Debugging workflow

If you do have a non trivial bug, this is when debugging strategies kick in. There is no silver bullet. Yet, strategies help:

> **For debugging a given problem, the favorable situation is when the problem is isolated in a small number of lines of code, outside framework or application code, with short modify-run-fail cycles**

1. Make it fail reliably. Find a test case that makes the code fail every time.
2. Divide and Conquer. Once you have a failing test case, isolate the failing code.
   - Which module.
   - Which function.
   - Which line of code.

   => isolate a small reproducible failure: a test case
3. Change one thing at a time and re-run the failing test case.
4. Use the debugger to understand what is going wrong.
5. Take notes and be patient. It may take a while.

Note: Once you have gone through this process: isolated a tight piece of code reproducing the bug and fix the bug using this piece of code, add the corresponding code to your test suite.

5.3 Using the Python debugger

The python debugger, `pdb`: https://docs.python.org/library/pdb.html, allows you to inspect your code interactively.

Specifically it allows you to:

- View the source code.
- Walk up and down the call stack.
- Inspect values of variables.
- Modify values of variables.
- Set breakpoints.

print

Yes, `print` statements do work as a debugging tool. However to inspect runtime, it is often more efficient to use the debugger.

5.3.1 Invoking the debugger

Ways to launch the debugger:

1. Postmortem, launch debugger after module errors.
2. Launch the module with the debugger.
3. Call the debugger inside the module

Postmortem

Situation: You're working in IPython and you get a traceback.

Here we debug the file `index_error.py`. When running it, an `IndexError` is raised. Type `%debug` and drop into the debugger.

```
In [1]: %run index_error.py
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in <module>()
      6
      7 if __name__ == '__main__':
----> 8     index_error()
      9

/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in index_error()
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':

IndexError: list index out of range

In [2]: %debug
> /home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py(5)index_error()
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
```

```
      6

ipdb> list
      1 """Small snippet to raise an IndexError."""
      2
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':
      8     index_error()
      9

ipdb> len(lst)
6
ipdb> print lst[len(lst)-1]
r
ipdb> quit

In [3]:
```

Post-mortem debugging without IPython

In some situations you cannot use IPython, for instance to debug a script that wants to be called from the command line. In this case, you can call the script with `python -m pdb script.py`:

```
$ python -m pdb index_error.py
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(1)<module>()
-> """Small snippet to raise an IndexError."""
(Pdb) continue
Traceback (most recent call last):
File "/usr/lib/python2.6/pdb.py", line 1296, in main
    pdb._runscript(mainpyfile)
File "/usr/lib/python2.6/pdb.py", line 1215, in _runscript
    self.run(statement)
File "/usr/lib/python2.6/bdb.py", line 372, in run
    exec cmd in globals, locals
File "<string>", line 1, in <module>
File "index_error.py", line 8, in <module>
    index_error()
File "index_error.py", line 5, in index_error
    print lst[len(lst)]
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(5)index_error()
-> print lst[len(lst)]
(Pdb)
```

Step-by-step execution

Situation: You believe a bug exists in a module but are not sure where.

For instance we are trying to debug `wiener_filtering.py`. Indeed the code runs, but the filtering does not work well.

- Run the script in IPython with the debugger using `%run -d wiener_filtering.p`:

```
In [1]: %run -d wiener_filtering.py
*** Blank or comment
```

```
*** Blank or comment
*** Blank or comment
Breakpoint 1 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_
↪filtering.py:4
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()
```

- Set a break point at line 34 using `b 34`:

```
ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(4)
↪<module>()
      3
1---> 4 import numpy as np
      5 import scipy as sp

ipdb> b 34
Breakpoint 2 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_
↪filtering.py:34
```

- Continue execution to next breakpoint with `c(ont(inue))`:

```
ipdb> c
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.
↪py(34)iterated_wiener()
     33     """
2--> 34     noisy_img = noisy_img
     35     denoised_img = local_mean(noisy_img, size=size)
```

- Step into code with `n(ext)` and `s(tep)`: `next` jumps to the next statement in the current execution context, while `step` will go across execution contexts, i.e. enable exploring inside function calls:

```
ipdb> s
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.
↪py(35)iterated_wiener()
2    34     noisy_img = noisy_img
---> 35     denoised_img = local_mean(noisy_img, size=size)
     36     l_var = local_var(noisy_img, size=size)

ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.
↪py(36)iterated_wiener()
     35     denoised_img = local_mean(noisy_img, size=size)
---> 36     l_var = local_var(noisy_img, size=size)
     37     for i in range(3):
```

- Step a few lines and explore the local variables:

```
ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.
↪py(37)iterated_wiener()
     36     l_var = local_var(noisy_img, size=size)
---> 37     for i in range(3):
     38         res = noisy_img - denoised_img
ipdb> print l_var
[[5868 5379 5316 ..., 5071 4799 5149]
 [5013  363  437 ...,  346  262 4355]
 [5379  410  344 ...,  392  604 3377]
 ...,
 [ 435  362  308 ...,  275  198 1632]
 [ 548  392  290 ...,  248  263 1653]
 [ 466  789  736 ..., 1835 1725 1940]]
```

```
ipdb> print l_var.min()
0
```

Oh dear, nothing but integers, and 0 variation. Here is our bug, we are doing integer arithmetic.

Raising exception on numerical errors

When we run the `wiener_filtering.py` file, the following warnings are raised:

```
In [2]: %run wiener_filtering.py
wiener_filtering.py:40: RuntimeWarning: divide by zero encountered in divide
    noise_level = (1 - noise/l_var )
```

We can turn these warnings in exception, which enables us to do post-mortem debugging on them, and find our problem more quickly:

```
In [3]: np.seterr(all='raise')
Out[3]: {'divide': 'print', 'invalid': 'print', 'over': 'print', 'under': 'ignore'}
In [4]: %run wiener_filtering.py
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
/home/esc/anaconda/lib/python2.7/site-packages/IPython/utils/py3compat.pyc in execfile(fname,
↪ *where)
    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.
↪py in <module>()
     55 pl.matshow(noisy_face[cut], cmap=pl.cm.gray)
     56
---> 57 denoised_face = iterated_wiener(noisy_face)
     58 pl.matshow(denoised_face[cut], cmap=pl.cm.gray)
     59

/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.
↪py in iterated_wiener(noisy_img, size)
     38         res = noisy_img - denoised_img
     39         noise = (res**2).sum()/res.size
---> 40         noise_level = (1 - noise/l_var )
     41         noise_level[noise_level<0] = 0
     42         denoised_img += noise_level*res

FloatingPointError: divide by zero encountered in divide
```

Other ways of starting a debugger

- **Raising an exception as a poor man break point**

  If you find it tedious to note the line number to set a break point, you can simply raise an exception at the point that you want to inspect and use IPython's `%debug`. Note that in this case you cannot step or continue the execution.

- **Debugging test failures using nosetests**

  You can run `nosetests --pdb` to drop in post-mortem debugging on exceptions, and `nosetests --pdb-failure` to inspect test failures using the debugger.

  In addition, you can use the IPython interface for the debugger in nose by installing the nose plugin ipdbplugin. You can than pass `--ipdb` and `--ipdb-failure` options to nosetests.

- **Calling the debugger explicitly**

  Insert the following line where you want to drop in the debugger:

```
import pdb; pdb.set_trace()
```

Warning: When running `nosetests`, the output is captured, and thus it seems that the debugger does not work. Simply run the nosetests with the `-s` flag.

Graphical debuggers and alternatives

- For stepping through code and inspecting variables, you might find it more convenient to use a graphical debugger such as winpdb.
- Alternatively, pudb is a good semi-graphical debugger with a text user interface in the console.
- Also, the pydbgr project is probably worth looking at.

5.3.2 Debugger commands and interaction

`l(list)`	Lists the code at the current position
`u(p)`	Walk up the call stack
`d(own)`	Walk down the call stack
`n(ext)`	Execute the next line (does not go down in new functions)
`s(tep)`	Execute the next statement (goes down in new functions)
`bt`	Print the call stack
`a`	Print the local variables
`!command`	Execute the given **Python** command (by opposition to pdb commands

Warning: Debugger commands are not Python code

You cannot name the variables the way you want. For instance, if in you cannot override the variables in the current frame with the same name: **use different names than your local variable when typing code in the debugger**.

Getting help when in the debugger

Type `h` or `help` to access the interactive help:

```
ipdb> help

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pdef   r        tbreak   w
a      c          continue  exit    l     pdoc   restart  u        whatis
alias  cl         d         h       list  pinfo  return   unalias  where
args   clear      debug     help    n     pp     run      unt
b      commands   disable   ignore  next  q      s        until
break  condition  down      j       p     quit   step     up

Miscellaneous help topics:
==========================
exec  pdb
```

```
Undocumented commands:
======================
retval  rv
```

5.4 Debugging segmentation faults using gdb

If you have a segmentation fault, you cannot debug it with pdb, as it crashes the Python interpreter before it can drop in the debugger. Similarly, if you have a bug in C code embedded in Python, pdb is useless. For this we turn to the gnu debugger, gdb, available on Linux.

Before we start with gdb, let us add a few Python-specific tools to it. For this we add a few macros to our `~/.gdbinit`. The optimal choice of macro depends on your Python version and your gdb version. I have added a simplified version in `gdbinit`, but feel free to read DebuggingWithGdb.

To debug with gdb the Python script `segfault.py`, we can run the script in gdb as follows

```
$ gdb python
...
(gdb) run segfault.py
Starting program: /usr/bin/python segfault.py
[Thread debugging using libthread_db enabled]

Program received signal SIGSEGV, Segmentation fault.
_strided_byte_copy (dst=0x8537478 "\360\343G", outstrides=4, src=
    0x86c0690 <Address 0x86c0690 out of bounds>, instrides=32, N=3,
    elsize=4)
        at numpy/core/src/multiarray/ctors.c:365
365             _FAST_MOVE(Int32);
(gdb)
```

We get a segfault, and gdb captures it for post-mortem debugging in the C level stack (not the Python call stack). We can debug the C call stack using gdb's commands:

```
(gdb) up
#1  0x004af4f5 in _copy_from_same_shape (dest=<value optimized out>,
    src=<value optimized out>, myfunc=0x496780 <_strided_byte_copy>,
    swap=0)
at numpy/core/src/multiarray/ctors.c:748
748         myfunc(dit->dataptr, dest->strides[maxaxis],
```

As you can see, right now, we are in the C code of numpy. We would like to know what is the Python code that triggers this segfault, so we go up the stack until we hit the Python execution loop:

```
(gdb) up
#8  0x080ddd23 in call_function (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/core/
↪arrayprint.py, line 156, in _leading_trailing (a=<numpy.ndarray at remote 0x85371b0>, _nc=
↪<module at remote 0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:3750
3750    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c

(gdb) up
#9  PyEval_EvalFrameEx (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/core/
↪arrayprint.py, line 156, in _leading_trailing (a=<numpy.ndarray at remote 0x85371b0>, _nc=
↪<module at remote 0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:2412
```

```
2412    in ../Python/ceval.c
(gdb)
```

Once we are in the Python execution loop, we can use our special Python helper function. For instance we can find the corresponding Python code:

```
(gdb) pyframe
/home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.py (158): _leading_
↪trailing
(gdb)
```

This is numpy code, we need to go up until we find code that we have written:

```
(gdb) up
...
(gdb) up
#34 0x080dc97a in PyEval_EvalFrameEx (f=
    Frame 0x82f064c, for file segfault.py, line 11, in print_big_array (small_array=<numpy.
↪ndarray at remote 0x853ecf0>, big_array=<numpy.ndarray at remote 0x853ed20>), throwflag=0)␣
↪at ../Python/ceval.c:1630
1630    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c
(gdb) pyframe
segfault.py (12): print_big_array
```

The corresponding code is:

```
def make_big_array(small_array):
    big_array = stride_tricks.as_strided(small_array,
                                         shape=(2e6, 2e6), strides=(32, 32))
    return big_array

def print_big_array(small_array):
    big_array = make_big_array(small_array)
```

Thus the segfault happens when printing `big_array[-10:]`. The reason is simply that `big_array` has been allocated with its end outside the program memory.

Note: For a list of Python-specific commands defined in the *gdbinit*, read the source of this file.

Wrap up exercise

The following script is well documented and hopefully legible. It seeks to answer a problem of actual interest for numerical computing, but it does not work... Can you debug it?

Python source code: `to_debug.py`