Multi-dimensional data

- NumPy

- matrix multiplication, @

- numpy.linalg.solve, numpy.polyfit

- NumPy is a Python package for dealing with multi-dimensional data

pylab ?

- Guttag [2nd edition] uses pylab in the examples, but...

> "pylab is a convenience module that bulk imports matplotlib.pyplot (for plotting) and numpy (for mathematics and working with arrays) in a single name space. Although many examples use pylab, it is no longer recommended."

NumPy arrays (example)

Python shell

```
> range(0, 1, .3)
| TypeError: 'float' object cannot be
  interpreted as an integer
> [1 + i / 4 for i in range(5)]
| [1.0, 1.25, 1.5, 1.75, 2.0]
```

python only supports ranges of int

generate 5 uniform values in range [1,2]

Python shell

```
> import numpy as np
> np.arange(0, 1, 0.3)
| array([0. , 0.3, 0.6, 0.9])
> type(np.arange(0, 1, 0.3))
| <class 'numpy.ndarray'>
> help(numpy.ndarray)
| +2000 lines of text
> np.linspace(1, 2, 5)
| array([1.  , 1.25, 1.5 , 1.75, 2.  ])
```

numpy can generate ranges with float

returns a "NumPy array" (not a list)
"arange" ≡ "array range" and generates the array explicitly
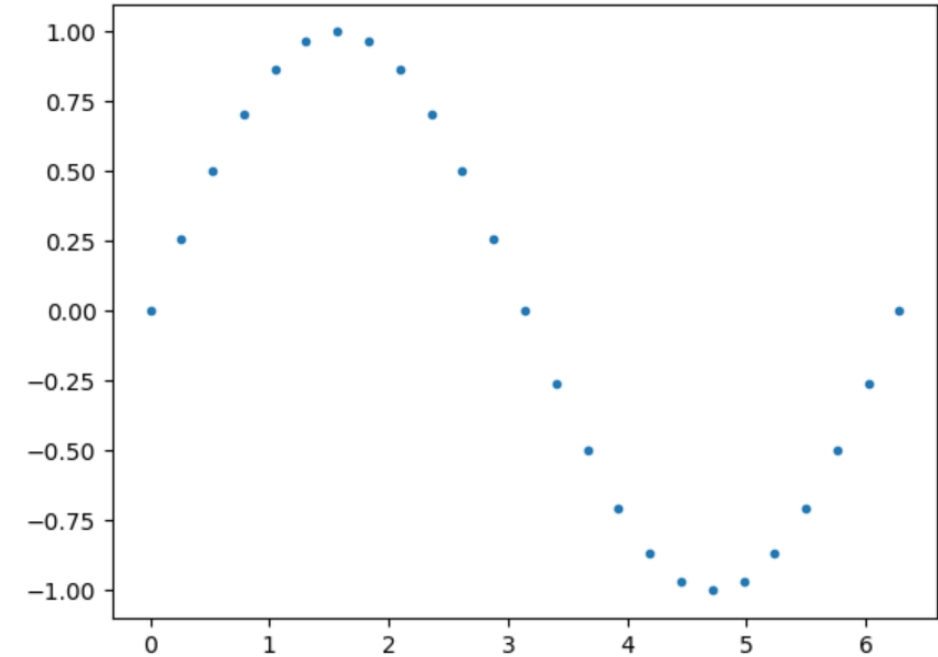
generate n uniformly spaced values

Plotting a function (example)

sin.py

```python
import matplotlib.pyplot as plt
import math
n = 25
x = [2 * math.pi * i / (n - 1) for i in range(n)]
y = [math.sin(v) for v in x]
plt.plot(x, y, '.')
plt.show()
```

sin_numpy.py

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2 * np.pi, 25)
y = np.sin(x)
plt.plot(x, y, '.')
plt.show()
```


- `np.sin` applies the `sin` function *to each* element of `x`
- `pyplot` accepts NumPy arrays
- `math.pi == np.pi` $\approx \dfrac{22}{7}$

A circle

```python
# circle.py
import matplotlib.pyplot as plt
import numpy as np

a = np.linspace(0, 2 * np.pi, 100)
x = np.sin(a)
y = np.cos(a)

plt.plot(x, y, 'r.-')
plt.plot(x[0], y[0], 'bo')
plt.show()
```
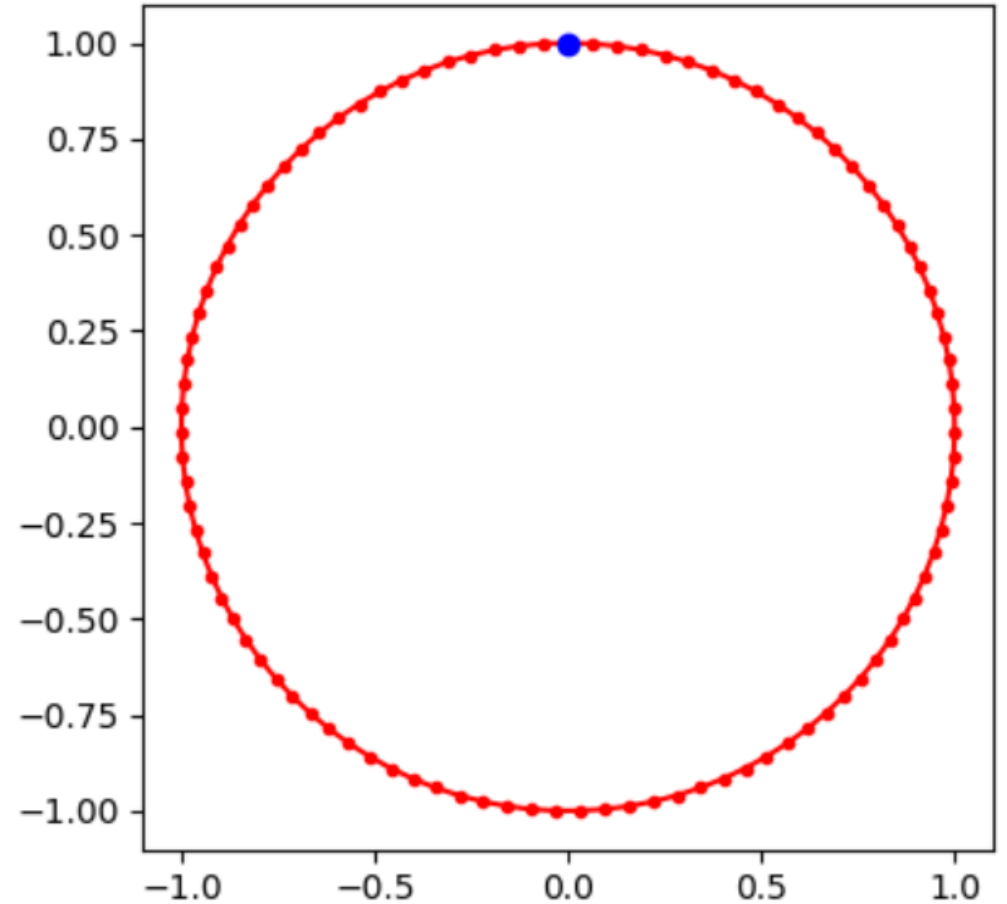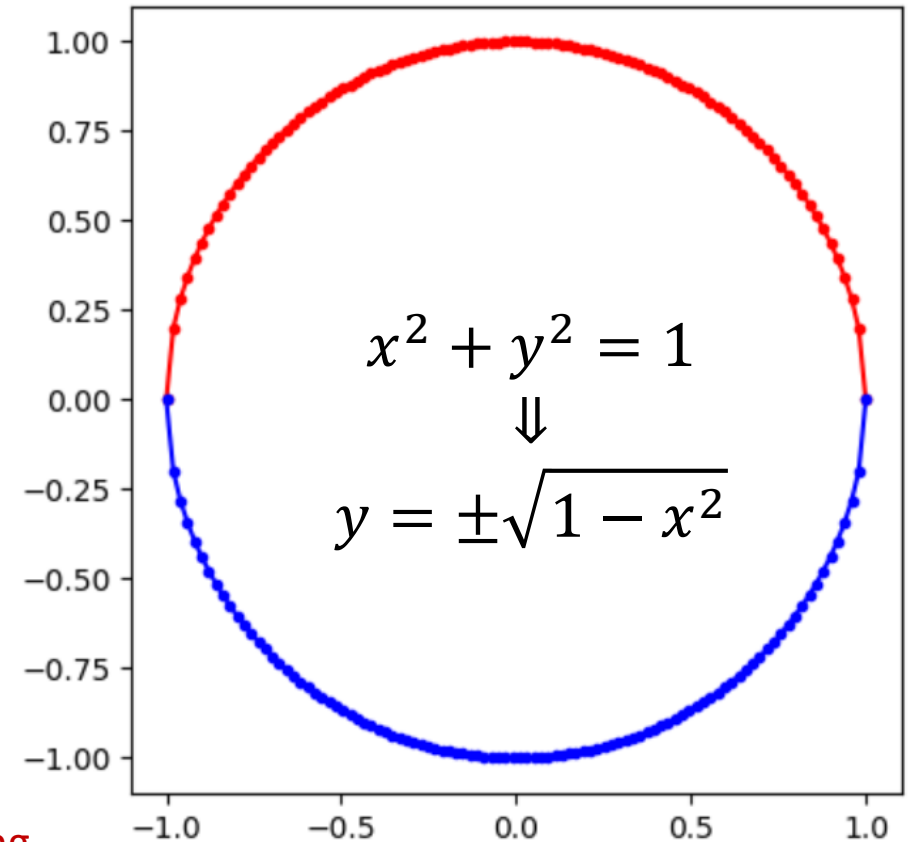
Two half circles

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

x = np.linspace(-1, 1, 100)
plt.plot(x, np.sqrt(1 - x ** 2), 'r.-')
plt.plot(x, -np.sqrt(1 - x ** 2), 'b.-')
plt.show()
```

compact expression computing
something quite comlicated

$$x^2 + y^2 = 1$$
$$\Downarrow$$
$$y = \pm\sqrt{1 - x^2}$$

- x is a NumPy array
- $**$ NumPy method __pow__ squaring each element in x
- binary – NumPy method __rsub__ that for each element e in x computes 1 - e
- np.sqrt NumPy method computing the square root of each element in x
- unary – NumPy method __neg__ that negates each element in x

Creating one-dimensional NumPy arrays

Python shell

```
> np.array([1, 2, 3])
| array([1, 2, 3])
> np.array((1, 2, 3))
| array([1, 2, 3])
> np.array(range(1, 4))
| array([1, 2, 3])
> np.arange(1., 4., 1.)
| array([1., 2., 3.])
> np.linspace(1, 3, 3)
| array([1., 2., 3.])
> np.zeros(3)
| array([0., 0., 0.])
> np.ones(3)
| array([1., 1., 1.])
> np.full(3, 7)
| array([7, 7, 7])
> np.random.random(3)
| array([0.73761651,
  0.60607355, 0.3614118 ])
```

```
> np.array([1, 2, 3]).dtype    # type of all values
| dtype('int32')
> np.arange(3, dtype='float')
| array([0., 1., 2.])
> np.arange(3, dtype='int16')   # 16 bit integers
| array([0, 1, 2], dtype=int16)
> np.arange(3, dtype='int32')   # 32 bit integers
| array([0, 1, 2])
> 1000 ** np.arange(5)
| array([1, 1000, 1000000, 1000000000, -727379968],
  dtype=int32)    # OOPS.. overflow
> 1000 ** np.arange(5, dtype='O')
| array([1, 1000, 1000000, 1000000000,
  1000000000000], dtype=object)   # Python integer
> np.arange(3, dtype='complex')
| array([0.+0.j, 1.+0.j, 2.+0.j])
```

⚠️ Elements of a NumPy array are not arbitrary precision integers by default – you can select between +25 number representations

https://numpy.org/doc/stable/reference/arrays.scalars.html

Supported dtypes

⚠ ▪ The supported dtypes is platform dependent

https://numpy.org/doc/stable/reference/arrays.scalars.html

Mantissa size in various numpy floats

```
> for data_type in ['half', 'float', 'single', 'double', 'longdouble', 'float32', 'float64']:
    x = np.array([1], dtype=data_type)
    for i in range(100):
      if x == x + (x / 2) ** i:
        break
    print(data_type, i - 1, 'bits mantissa')
| half 10 bits mantissa
| float 52 bits mantissa
| single 23 bits mantissa
| double 52 bits mantissa
| longdouble 52 bits mantissa
| float32 23 bits mantissa                 # platform independent
| float64 52 bits mantissa                 # platform independent
```

mantissa

$$1.\overbrace{0000100000}^{}{}_2$$

i

docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html

Creating multi-dimensional NumPy arrays

```
Python shell
```

```
> np.array([[1, 2, 3], [4, 5, 6]])
| array([[1, 2, 3],
|        [4, 5, 6]])
> np.array([[1, 2], [4, 5, 6]])
| ValueError: inhomogeneous shape ⚠
> np.arange(1, 7).reshape(2, 3)
| array([[1, 2, 3],
|        [4, 5, 6]])
> x = np.arange(12).reshape(2, 2, 3)
> x
| array([[[ 0,  1,  2],
|         [ 3,  4,  5]],
|
|        [[ 6,  7,  8],
|         [ 9, 10, 11]]])
> numpy.zeros((2, 5), dtype='int32')
| array([[0, 0, 0, 0, 0],
|        [0, 0, 0, 0, 0]])
```

```
> x.size
| 12
> x.ndim
| 3
> x.shape
| (2, 2, 3)
> x.dtype
| dtype('int32')
> x.flatten()
| array([0,1,2,3,4,5,6,7,8,9,10,11])
> list(x.flat)  # flat is an iterator
| [0,1,2,3,4,5,6,7,8,9,10,11]
> np.eye(3)     # diagonal 2D array
| array([[1., 0., 0.],
|        [0., 1., 0.],
|        [0., 0., 1.]])
```

View vs Copy

- Numpy is optimized to handle big multi-dimensional arrays

- To *avoid copying* data, **views** allows one to look at the underlying data in different ways (data can be shared by multiple views)

- `reshape`, `ravel` and slicing are examples creating views

- `flatten` and `ravel` both turn multiple dimensional arrays into one dimensional arrays but `flatten` creates an explicit copy whereas `ravel` creates a space efficient view

```
Python shell
> x = np.arange(6)
> y = x.reshape(2, 3)     # view
> y[0][0] = 42            # updates x
> x
| array([42, 1, 2, 3, 4, 5])
> y
| array([[42,  1,  2],
|        [ 3,  4,  5]])
> z = y.flatten()         # copy
> z[5] = 0
> z
| array([42, 1, 2, 3, 4, 0])
> x
| array([42, 1, 2, 3, 4, 5])
> w = y.ravel()           # view
> w[5] = -1
> w
| array([42, 1, 2, 3, 4, -1])
> x
| array([42, 1, 2, 3, 4, -1])
```

NumPy operations

```
> x = numpy.arange(3)
> x
| array([0, 1, 2])
> x + x   # elementwise addition
| array([0, 2, 4])
> 1 + x   # add integer to each element
| array([1, 2, 3])
> x * x   # elementwise multiplication
| array([0, 1, 4])
> np.dot(x, x)   # dot product
| 5
> np.cross([1, 2, 3], [3, 2, 1]) # cross product
| array([-4,  8, -4])
```

```
> a = np.arange(6).reshape(2,3)
> a
| array([[0, 1, 2],
|        [3, 4, 5]])
> a.T   # matrix transposition, view
| array([[0, 3],
|        [1, 4],
|        [2, 5]])
> a @ a.T   # matrix multiplication
| array([[ 5, 14],
|        [14, 50]])
> a += 1   # modifies view
> a
| array([[1, 2, 3],
|        [4, 5, 6]])
```

PEP 465 - A dedicated infix operator for matrix multiplication

Universal functions (apply to each entry)

```
Python shell
> x = np.array([[1, 2], [3, 4]])
> np.sin(x)   # also: cos, exp, sqrt, log, ceil, floor, abs
| array([[ 0.84147098,  0.90929743],
|        [ 0.14112001, -0.7568025 ]])
> np.sign(np.sin(x))
| array([[ 1.,   1.],
|        [ 1., -1.]])
> np.mod(np.arange(10), 3)   # same as: np.arange(10) % 3
| array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype=int32)
```

Axis

```
Python shell
> x = np.arange(1, 7).reshape(2, 3)
> x
| array([[1, 2, 3],
|        [4, 5, 6]])
> x.sum()   # = x.sum(axis=(0, 1))
| 21
> x.sum(axis=0)
| array([5, 7, 9])
> x.sum(axis=1)
| array([6, 15])
> x.min()   # = x.min(axis=(0, 1))
| 1
```

```
Python shell
> x.min(axis=0)
| array([1, 2, 3])
> x.min(axis=1)
| array([1, 4])
> x.cumsum()
| array([ 1,  3,  6, 10, 15, 21], dtype=int32)
> x.cumsum(axis=0)
| array([[1, 2, 3],
|        [5, 7, 9]], dtype=int32)
> x.cumsum(axis=1)
| array([[ 1,  3,  6],
|        [ 4,  9, 15]], dtype=int32)
```

Slicing

```
Python shell
> x = numpy.arange(20).reshape(4, 5)
> x
| array([[ 0,  1,  2,  3,  4],
|        [ 5,  6,  7,  8,  9],
|        [10, 11, 12, 13, 14],
|        [15, 16, 17, 18, 19]])
> x[2, 3]   # =  x[(2, 3)]
| 13
> x[1:4:2, 2:4:1]   # rows 1 and 3, and columns 2 and 3, view
| array([[ 7,  8],
|        [17, 18]])
> x[:, 3]
| array([ 3,  8, 13, 18])
> x[..., 3]   # ... is placeholder for ':' for all missing dimensions
| array([ 3,  8, 13, 18])
> type(...)
| <class 'ellipsis'>
```

Broadcasting (stretching arrays to get same size)

■ Numpy tries to apply *broadcasting*, if array shapes do not match, i.e. adds missing leading dimensions and repeats a dimension with only one element:

```
  [[1],[2]] + [10,20]         column + row vector
≡ [[1],[2]] + [[10,20]]              both ndim = 2
≡ [[1,1],[2,2]] + [[10,20]]
≡ [[1,1],[2,2]] + [[10,20],[10,20]]
≡ [[11,21],[12,22]]
```

■ To prevent unexpected broadcasting, add an assertion to your program:

⚠️
```
    assert x.shape == y.shape
```

docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Python shell

```
> x = np.array([[1, 2, 3], [4, 5, 6]])
> y = np.array([1, 2, 3])   # one row
> z = np.array([[1], [2]]) # one column
> x + 3   # add 3 to each entry
| array([[4, 5, 6],
|        [7, 8, 9]])
> x + y   # add y to each row
| array([[2, 4, 6],
|        [5, 7, 9]])
> x + z   # add z to each column
| array([[2, 3, 4],
|        [6, 7, 8]])
> y + z   # 2 rows with y + 3 columns with z
| array([[2, 3, 4],
|        [3, 4, 5]])
> z == z.T   # [[1,1],[2,2]] == [[1,2],[1,2]]
| array([[ True, False],
|        [False,  True]])
```

Masking

```
Python shell
> x = np.arange(1, 11).reshape(2, 5)
> x
| array([[ 1,  2,  3,  4,  5],
|        [ 6,  7,  8,  9, 10]])
> x % 3
| array([[1, 2, 0, 1, 2],
|        [0, 1, 2, 0, 1]], dtype=int32)
> x % 3 == 0
| array([[False, False,  True, False, False],
|        [ True, False, False,  True, False]])
> x[x % 3 == 0]   # use Boolean matrix to select entries
| array([3, 6, 9])
> x[:, x.sum(axis=0) % 3 == 0]   # columns with sum divisible by 3
| array([[ 2,  5],
|        [ 7, 10]])
```

Numpy is fast... but be aware of dtype ⚠️

Python shell

```
> sum([x**2 for x in range(1000000)])
| 333332833333500000
> (np.arange(1000000)**2).sum()
| 584144992     # wrong since overflow when default dtype='int32'
> (np.arange(1000000, dtype='int64')**2).sum()
| 333332833333500000   # 64 bit integers do not overflow

> import timeit from timeit
> timeit('sum([x**2 for x in range(1000000)])', number=1)
| 0.5614346340007614
> timeit('(np.arange(1000000)**2).sum()', setup='import numpy as np', number=1)
| 0.014362967000124627  # ridiculous fast but also wrong result...
> timeit('(np.arange(1000000, dtype='int64')**2).sum()',
  setup='import numpy as np', number=1)
| 0.048017077999247704  # fast and correct

> np.iinfo(np.int32).min
| -2147483648
> np.iinfo(np.int32).max
| 2147483647
```

numpy.int32 – 32 bit signed two's-complement integers

32 bits $b_{31}b_{30}b_{29}b_{28}\cdots b_2 b_1 b_0$
represent the value

$$-b_{31}\cdot 2^{31} + \sum_{i=0}^{30} b_i \cdot 2^i$$

1000000000000000000000000000000000	-2147483648
10000000000000000000000000000001	-2147483647
⋮	
11111111111111111111111111111110	-2
11111111111111111111111111111111	-1
00000000000000000000000000000000	0
00000000000000000000000000000001	1
⋮	
01111111111111111111111111111110	2147483646
01111111111111111111111111111111	2147483647

Note: There is one more negative number than positive

Python shell

```
> np.int32(- 2 ** 31)
| -2147483648
> np.int32(- 2 ** 31) + 1
| -2147483647
> np.int32(- 2 ** 31) - 1
| 2147483647
> np.int32(2 ** 31)
| OverflowError: Python int too
  large to convert to C long
> np.int32(2 ** 31 - 1)
| 2147483647
> np.int32(2 ** 31 - 1) + 1
| -2147483648
> np.abs(np.int32(-2147483647))
| 2147483647
> np.abs(np.int32(-2147483648))
| -2147483648
```

numpy.org/doc/stable/user/basics.types.html
en.wikipedia.org/wiki/Two's_complement

Linear algebra

numpy.matrix

⚠️ *"It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future."*

```
Python shell
> x = np.arange(1, 5, dtype=float).reshape(2, 2)
> x
| array([[1., 2.],
|        [3., 4.]])
> x.T   # matrix transpose
| array([[1., 3.],
|        [2., 4.]])
> np.linalg.det(x)   # matrix determinant
| -2.0000000000000004
> np.linalg.inv(x)   # matrix inverse
| array([[-2. ,  1. ],
|        [ 1.5, -0.5]])
> np.linalg.eig(x)   # eigenvalues and eigenvectors
| (array([-0.37228132,  5.37228132]),
|  array([[-0.82456484, -0.41597356], [0.56576746, -0.90937671]]))
> y = np.array([[5.], [7.]])
> np.linalg.solve(x, y)   # solve linear matrix equations
| array([[-3.],      # z1
|        [ 4.]])     # z2
```
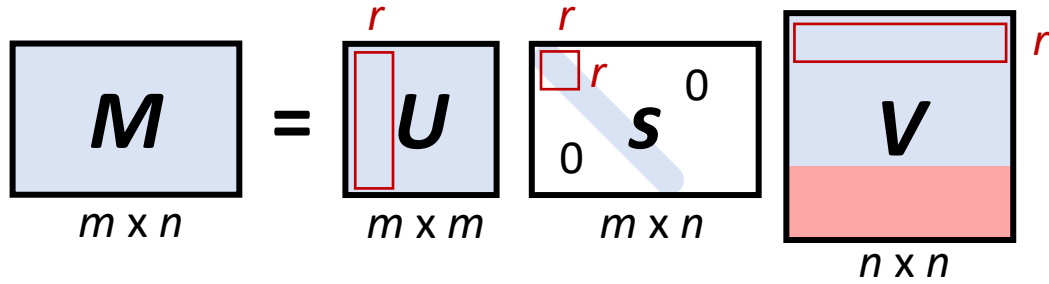
$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}$$

Singular value decomposition, `np.linalg.svd`

$$M = U \quad S \quad V$$

$m \times n$ $m \times m$ $m \times n$ $n \times n$

- *U* and *V* unitary matrix ($UU^T = I$)
- *S* diagonal matrix, decreasing singular values

np.ndarray shape=(520, 800) dtype=uint8 min=0 max=252
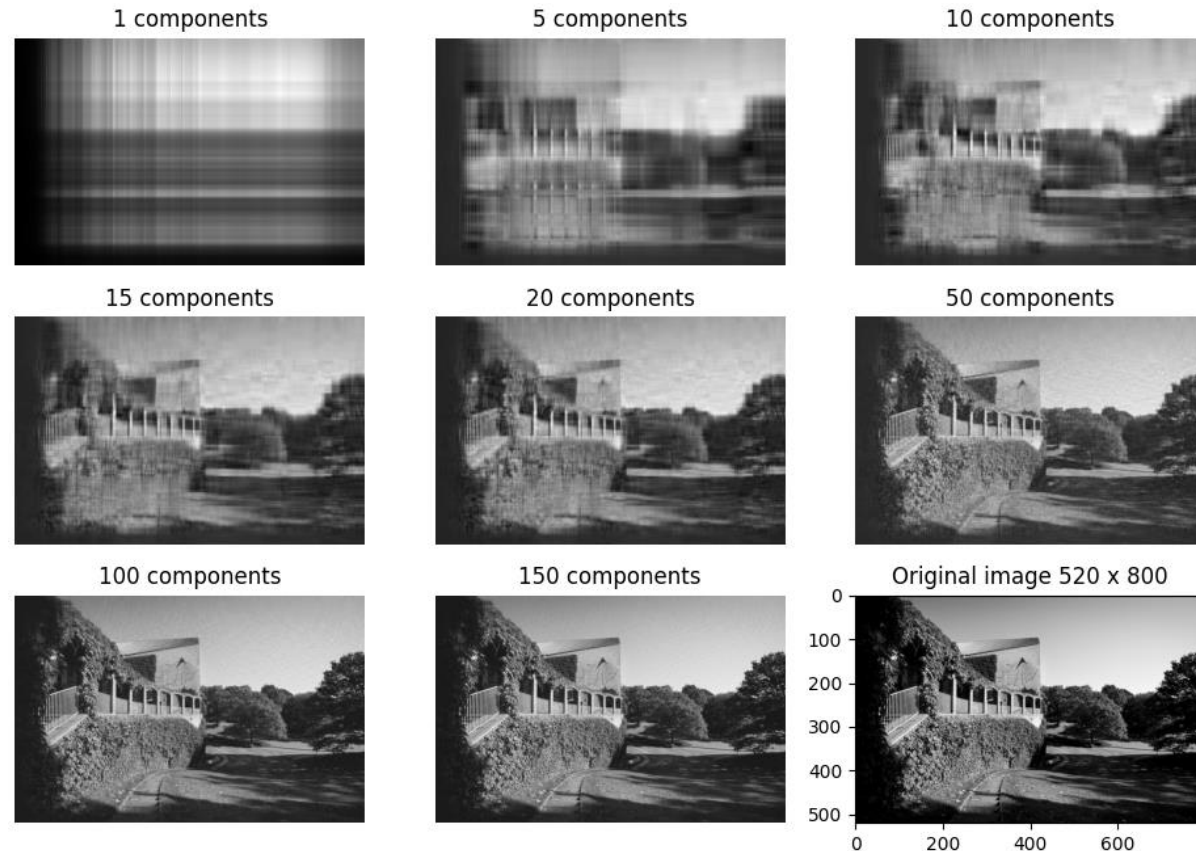
image-reconstruction.py

```python
import numpy as np
import cv2          # Computer Vision, opencv.org
import matplotlib.pyplot as plt

color = cv2.imread('university.jpg')            # color image
gray = cv2.cvtColor(color, cv2.COLOR_BGR2GRAY)  # convert to gray

u, s, v = np.linalg.svd(gray)    # Calculating the SVD

for i, r in enumerate([1, 5, 10, 15, 20, 50, 100, 150], start=1):
    rank_r = u[:, :r] @ np.diag(s[:r]) @ v[:r, :]
    plt.subplot(3, 3, i)
    plt.imshow(rank_r, cmap='gray', vmin=0, vmax=255)
    plt.title(f'{r} components')
    plt.axis('off')
plt.subplot(3, 3, 9)
plt.imshow(gray, cmap='gray')
plt.title(f'Original image {gray.shape[0]} x {gray.shape[1]}')
plt.show()
```


1 components   5 components   10 components   15 components   20 components   50 components   100 components   150 components   Original image 520 x 800

... and in color

image-reconstruction-color.py

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread

color = imread('university.jpg')
color = color / 255   # convert integers 0..255 to floats 0..1

plt.subplot(4, 2, 8)
plt.imshow(color)
plt.axis('off')
plt.title(f'Original')

height, width, colors = color.shape

u, s, v = np.linalg.svd(color.reshape((height, width * colors)),
                        full_matrices=False)

for i, r in enumerate([1, 2, 5, 10, 25, 50, 125], start=1):
    rank_r = u[:, :r] @ np.diag(s[:r]) @ v[:r, :]
    plt.subplot(4, 2, i)
    plt.imshow(rank_r.reshape((height, width, colors)))
    plt.title(f'{r} components')
    plt.axis('off')
plt.show()
```

shape=(520, 800, 3)
3 = (red, blue, green)

shape=(520, 2400)

v shape=(520, 2400)
instead of (2400, 2400)

shape=(520, 800, 3)

1 components

2 components

5 components

10 components

25 components

50 components

125 components

Original

... and in color (stacked)

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread

color = imread('university.jpg')
color = color / 255   # convert integers 0..255 to floats 0..1

plt.subplot(4, 2, 8)
plt.imshow(color)
plt.axis('off')
plt.title(f'Original')

u, s, v = np.linalg.svd(color.transpose(2, 0, 1), full_matrices=False)

print(f'{u.shape=} {s.shape=} {v.shape=}')

for i, r in enumerate([1, 2, 5, 10, 25, 50, 125], start=1):
    rank_r = (u[:, :, :r] * s[:, None, :r]) @ v[:, :r, :]
    plt.subplot(4, 2, i)
    plt.imshow(rank_r.transpose(1, 2, 0))
    plt.title(f'{r} components')
    plt.axis('off')
plt.show()
```
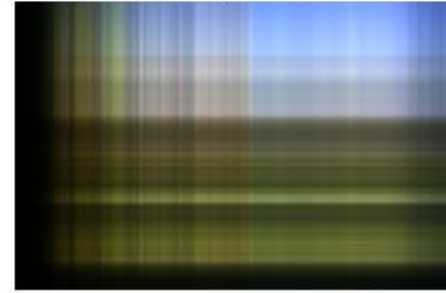
shape = (520, 800, 3)

shape = (3, 520, 800)

element-wise multiplication (*), broadcasting (None), stacked for each color (:)

shape = (520, 800, 3)

Python shell

```
| u.shape=(3, 520, 520) s.shape=(3, 520) v.shape=(3, 520, 800)
```


1 components

2 components

5 components

10 components

25 components

50 components

125 components

Original

matplotlib.image.imread   cv2.imread   cv2.imread corrected

color-image.py

```python
import matplotlib.pyplot as plt
import matplotlib.image
import cv2

img1 = matplotlib.image.imread('university.jpg')
img2 = cv2.imread('university.jpg')    # cv2 uses BGR instead of RGB
img3 = img2[:, :, ::-1]                # change color order BGR to RGB

Images = [(img1, 'matplotlib.image.imread'), (img2, 'cv2.imread'), (img3, 'cv2.imread corrected')]

for i, (img, title) in enumerate(images, start=1):
    plt.subplot(1, 3, i)
    plt.imshow(img)
    plt.axis('off')
    plt.title(title)

plt.show()
```
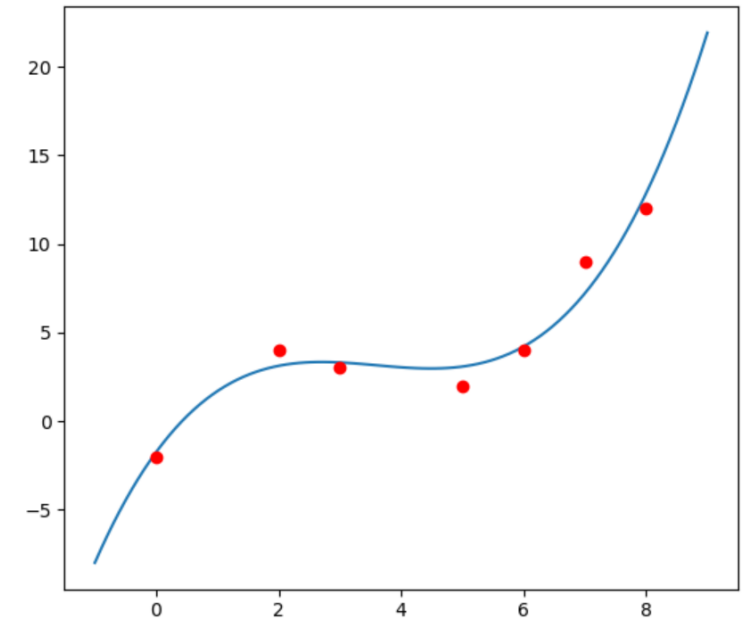
numpy.polyfit

- Given n points with $(x_0, y_0), ..., (x_{n-1}, y_{n-1})$
- Find polynomial p of degree d that minimizes

$$\sum_{i=0}^{n-1} (y_i - p(x_i))^2$$

- known as least squares fit / linear regression / polynomial regression

docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

fit.py

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

x = [0, 2, 3, 5, 6, 7, 8]
y = [-2, 4, 3, 2, 4, 9, 12]


coefficients = np.polyfit(x, y, 3)


fx = np.linspace(-1, 9, 100)
fy = np.polyval(coefficients, fx)

plt.plot(fx, fy, '-')
plt.plot(x, y, 'ro')

plt.show()
```
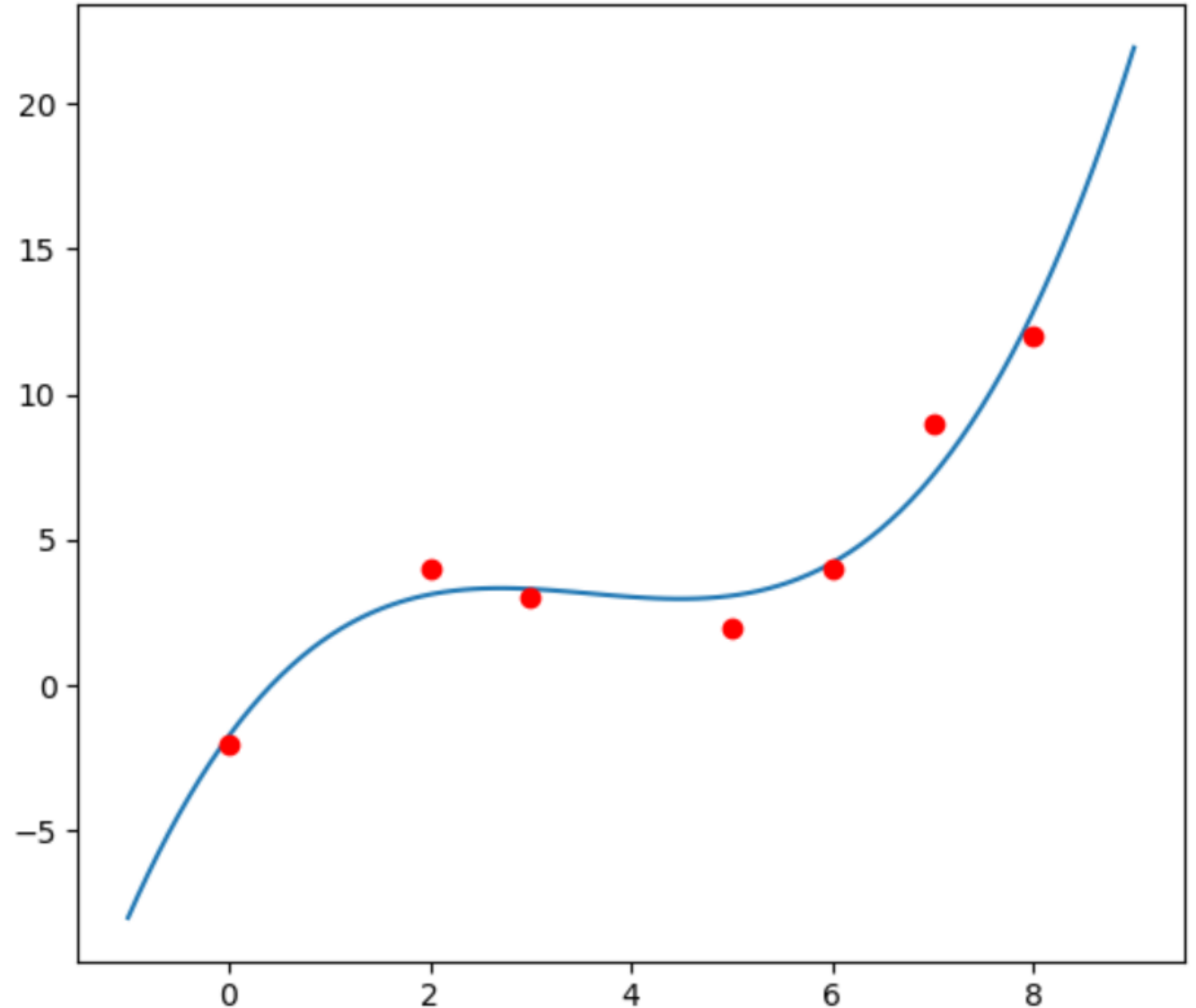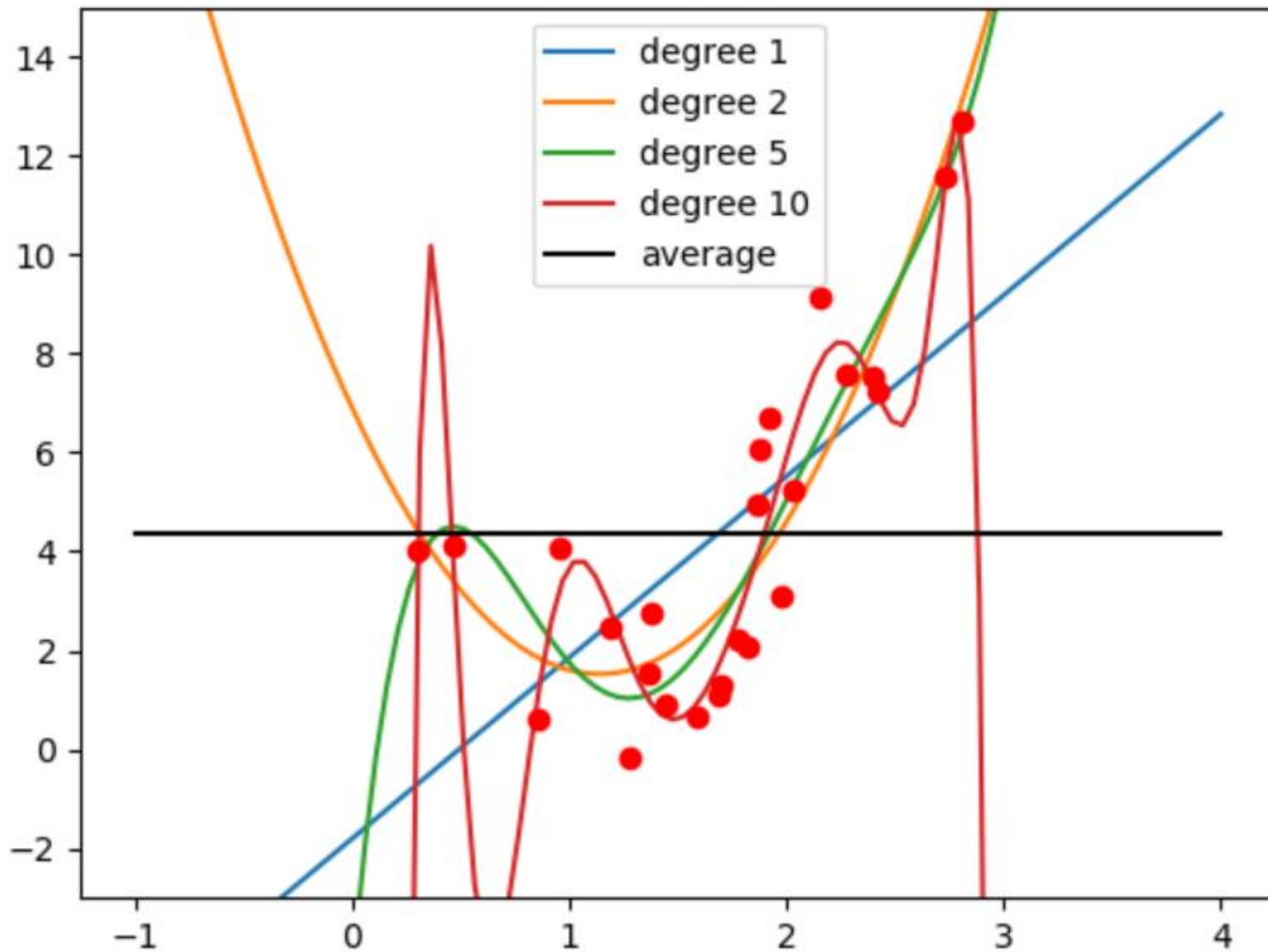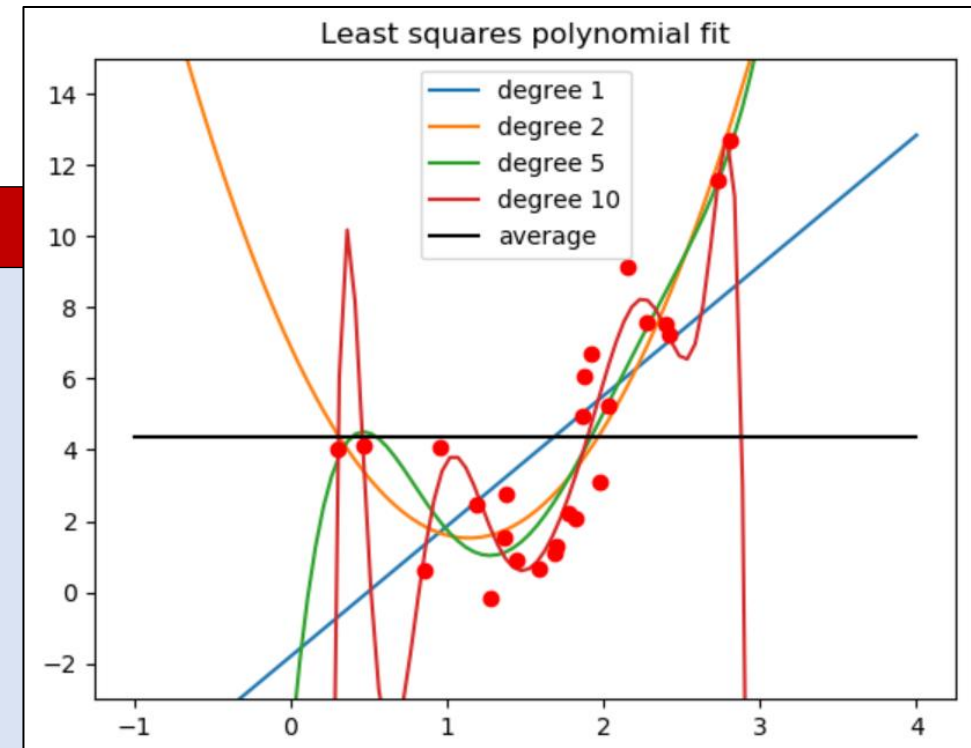
degree

Least squares polynomial fit

Least squares polynomial fit

```
polyfit.py

import matplotlib.pyplot as plt
import numpy as np

x = 3 * np.random.random(25)
noise = np.random.random(x.size) ** 2
y = 5 * x ** 2 - 12 * x + 7 + 5 * noise

for degree in [1, 2, 5, 10]:
    coefficients = np.polyfit(x, y, degree)
    fx = np.linspace(-1, 4, 100)
    fy = np.polyval(coefficients, fx)
    plt.plot(fx, fy, '-', label=f'degree {degree}')

avg = np.average(y)
plt.plot(x, y, 'ro')
plt.plot([-1, 4], [avg, avg], 'k-', label='average')
plt.ylim(-3, 15)
plt.title('Least squares polynomial fit')
plt.legend()
plt.show()
```
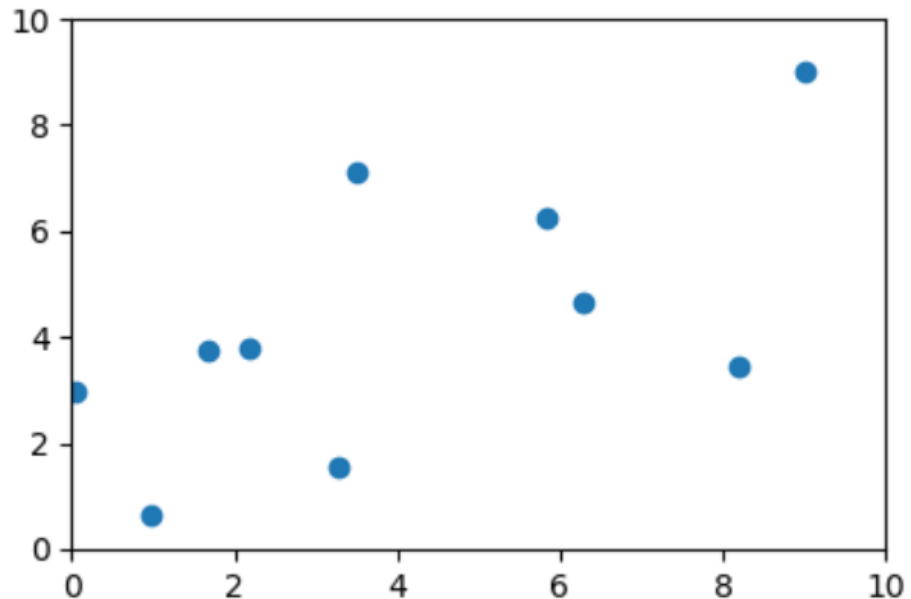
Animating bouncing balls

- matplotlib figures can be animated using

  `matplotlib.animation.FuncAnimation`

  that as arguments takes the figure to be updated/
  redrawn, a function to call for each update, and an
  interval in milliseconds between updates

balls.py

```python
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from numpy import zeros, maximum, minimum
from numpy.random import random

g = 0.01
N = 10
x, y = 10.0 * random(N), 1.0 + 9.0 * random(N)
dx, dy = random(N) / 5, zeros(N)

fig = plt.figure()
plt.xlim(0, 10)
plt.ylim(0, 10)
balls, = plt.plot(x, y, 'o') # returns Line2D obj

def move(frame):
    global x, y, dx, dy

    x += dx
    bounce = (x > 10.0) | (x < 0.0) # numpy mask
    dx[bounce] = -dx[bounce]
    x = minimum(10.0, maximum(0.0, x))

    y += dy
    bounce = y < 0.0   # numpy mask
    y[bounce] -= dy[bounce]
    dy[bounce] = -dy[bounce]
    dy -= g

    balls.set_data(x, y)   # update positions

# removing 'ani =' causes program to fail...
ani = FuncAnimation(fig, move, interval=25)
plt.show()
```
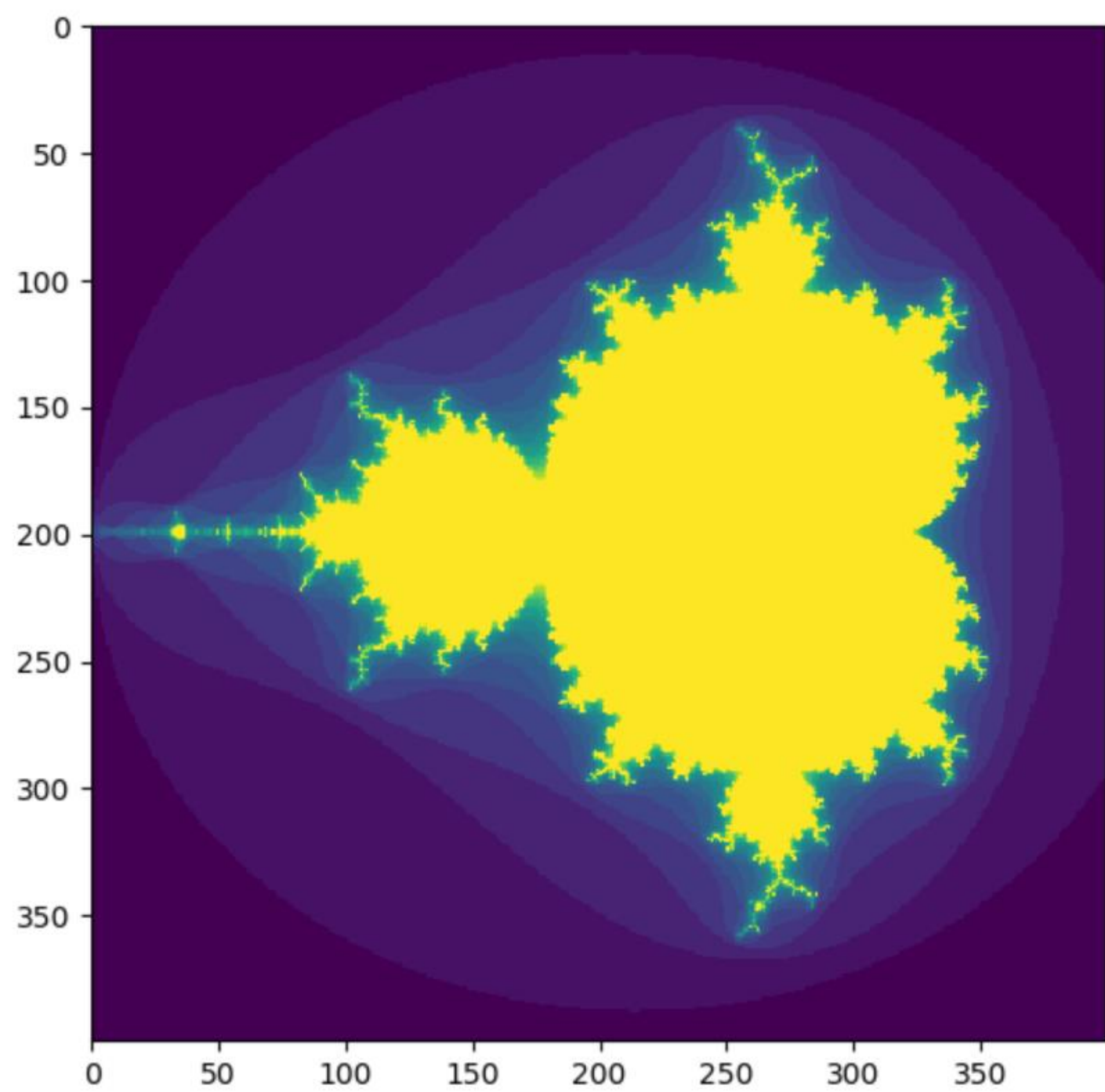
mandelbrot.py

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

def mandelbrot(h, w, maxit=20):
    '''Returns an image of the Mandelbrot fractal of size (h, w).'''

    x = np.linspace(-2.0, 0.8, w).reshape(1, w)    # row vector
    y = np.linspace(-1.4, 1.4, h).reshape(h, 1)    # column vector
    c = x + y * 1j                                 # broadcast & complex
    z = c
    divtime = np.full(z.shape, maxit, dtype=int)   # all values = maxit

    for i in range(maxit):
        z = z * z + c                              # elementwise
        diverge = z * np.conj(z) > 4               # who is diverging
        div_now = diverge & (divtime == maxit)     # who is diverging now
        divtime[div_now] = i                       # note when
        z[diverge] = 0                             # limit divergence
    return divtime                                 #    (avoids overflows)

plt.imshow(mandelbrot(400, 400))
plt.show()
```

code from docs.scipy.org/doc/numpy/user/quickstart.html