Multi-dimensional data

- NumPy
- matrix multiplication, @
- np.linalg.solve, numpy.polyfit

pylab ?

- Guttag 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.”*

matplotlib.org/faq/usage_faq.html

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

www.numpy.org

docs.scipy.org/doc/numpy-dev/user/quickstart.html

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)

generate n uniformly space values

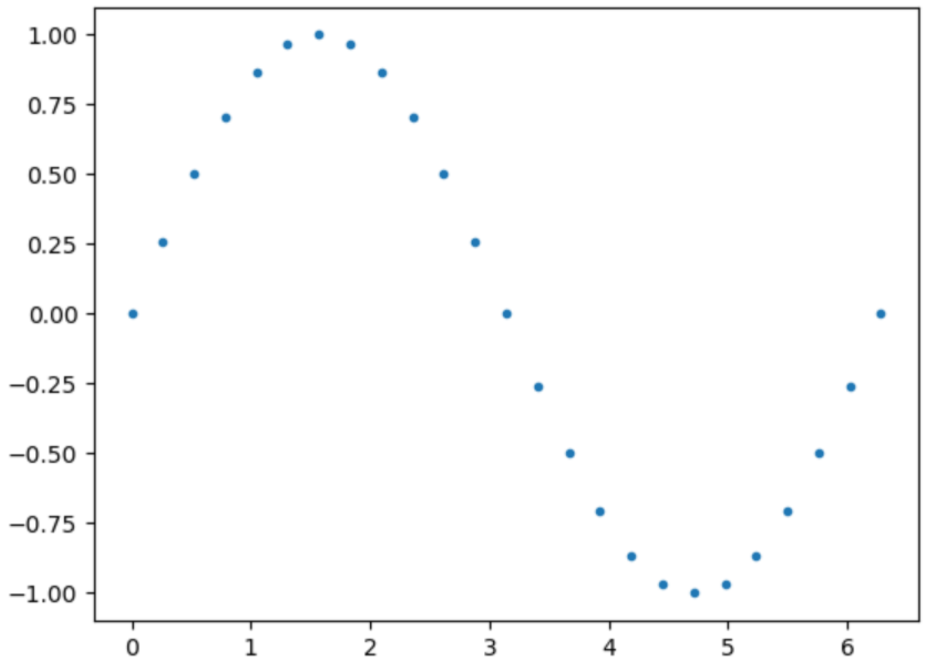
Plotting a function (example)

sin.py

```
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

```
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()
```



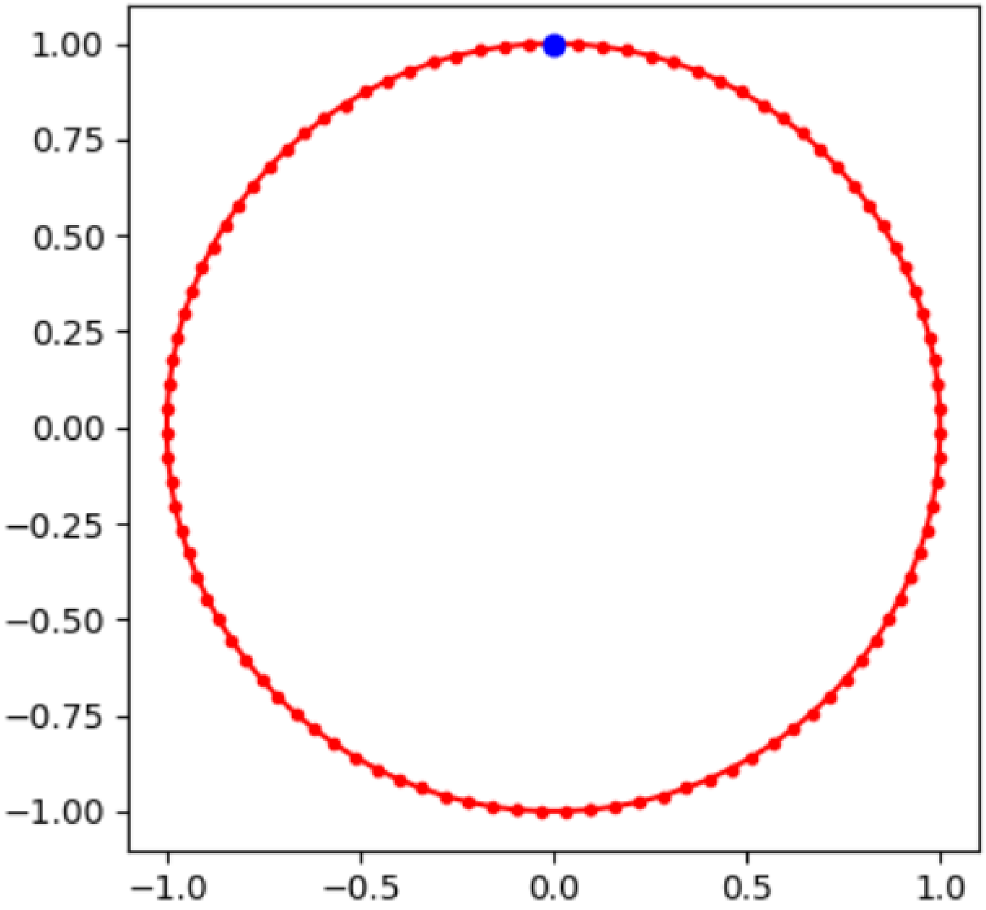
- `np.sin` applies the `sin` function to each element of `x`
- `pyplot` accepts numpy arrays

A circle

circle.py

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

a = np.linspace(0, 2*np.pi, 100)
plt.plot(np.sin(a), np.cos(a), 'r.-')
plt.plot(np.sin(a)[0], np.cos(a)[0], 'bo')
plt.show()
```

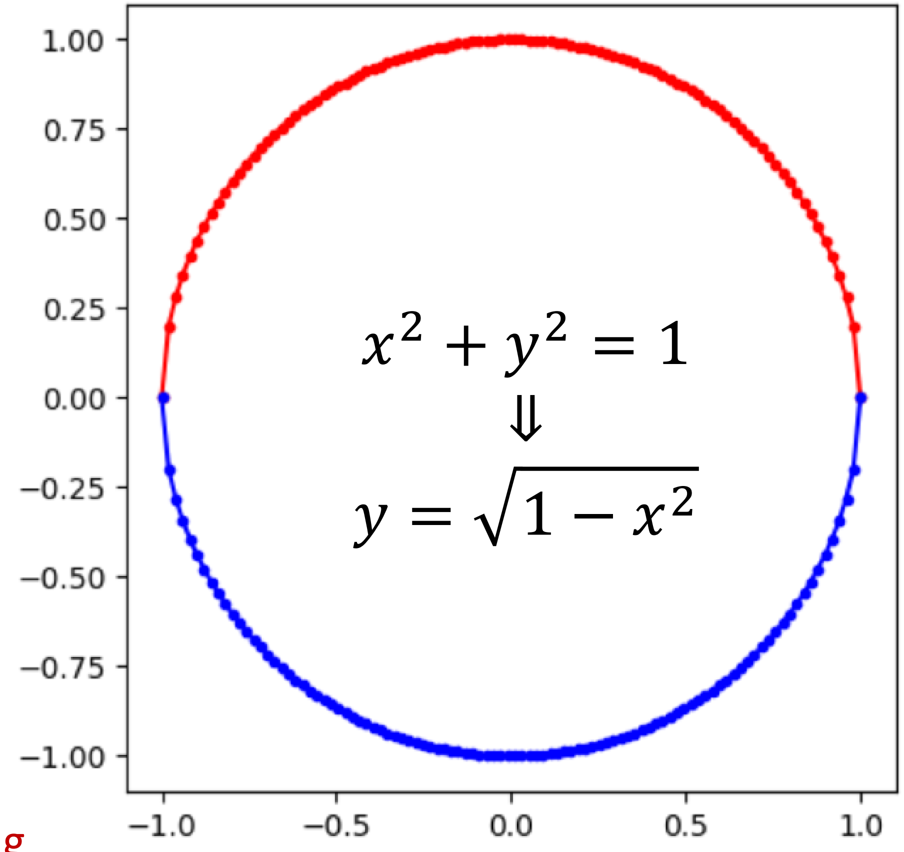
- `np.sin` applies the `sin` function to each element of `x`
- `pyplot` accepts numpy arrays



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` 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
- unary – numpy method `__neg__` that negates each element in `x`
- `np.sqrt` numpy method computing the square root of 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.random.random(3)
| array([0.73761651,
  0.60607355, 0.3614118 ])
```

```
> 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://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.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.ravel()
| array([ 0,  1,  2,  3,  4,  5,  6,
  7,  8,  9, 10, 11])
> list(x.flat)
| [ 0,  1,  2,  3,  4,  5,  6,  7,
  8,  9, 10, 11]
> np.eye(3)
| array([[1., 0., 0.],
|        [0., 1., 0.],
|        [0., 0., 1.]])
```

Numpy operations

Python shell

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

PEP 465 -- A dedicated infix operator for matrix multiplication

Universal functions

Python shell

```
> 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)
| array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], 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]
| 13
> x[1:4:2, 2:4:1]  # rows 1 and 3, and columns 2 and 3
| 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'>
```

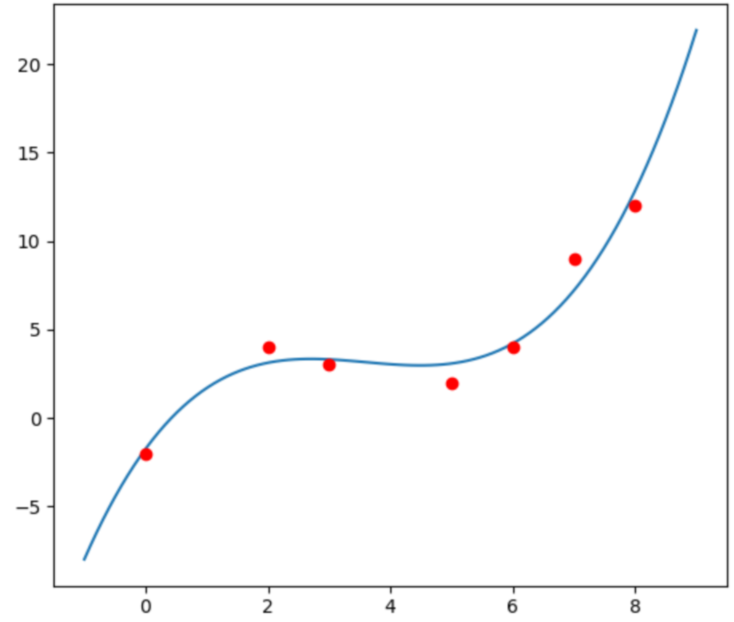
Linear algebra

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.inv(x)  # matrix inverse
| array([[-2. ,  1. ],
|        [ 1.5, -0.5]])
> numpy.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
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}$$

numpy.polyfit



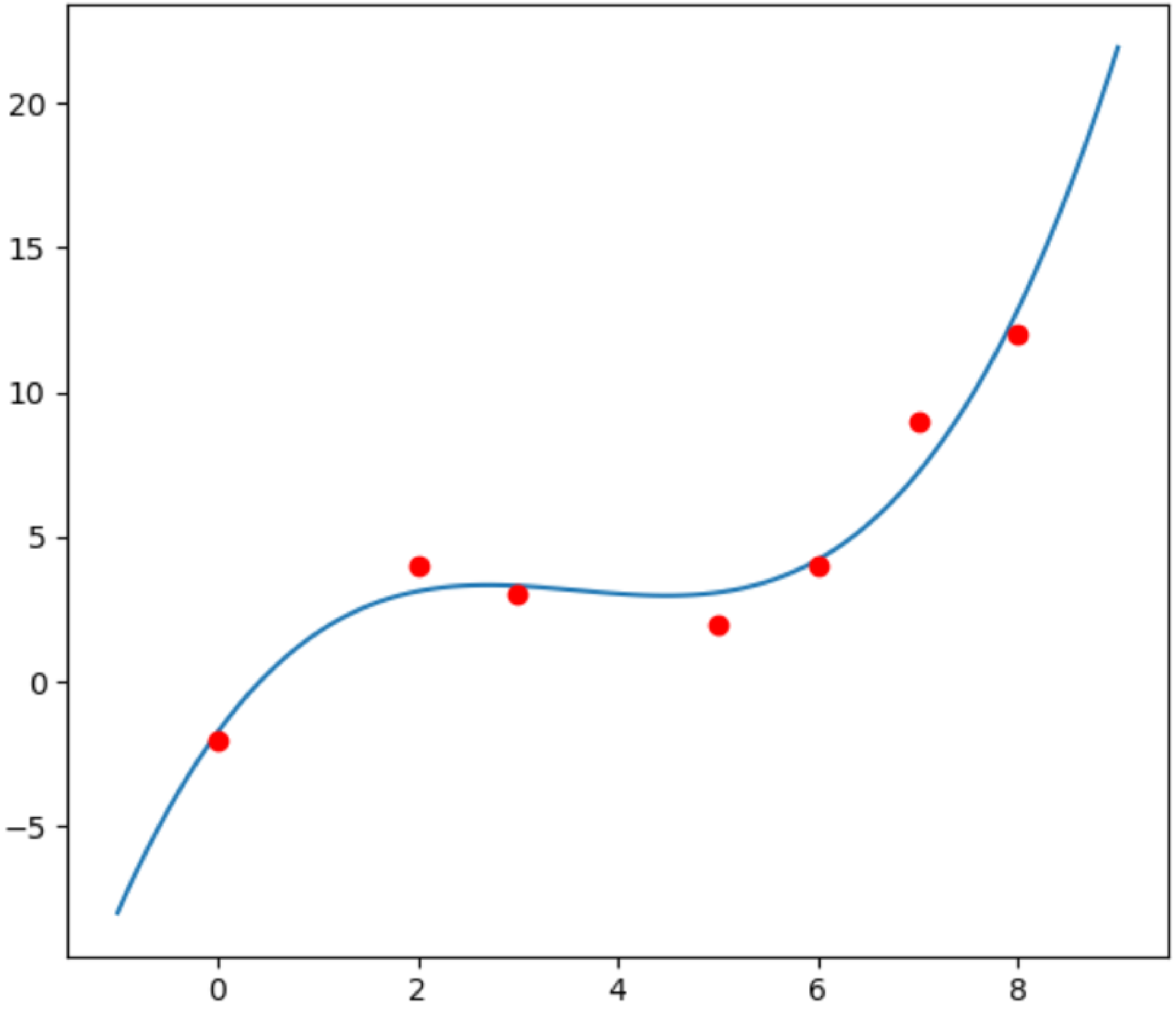
- Given n points with $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$
- Find polynomial p of degree d that minimizes

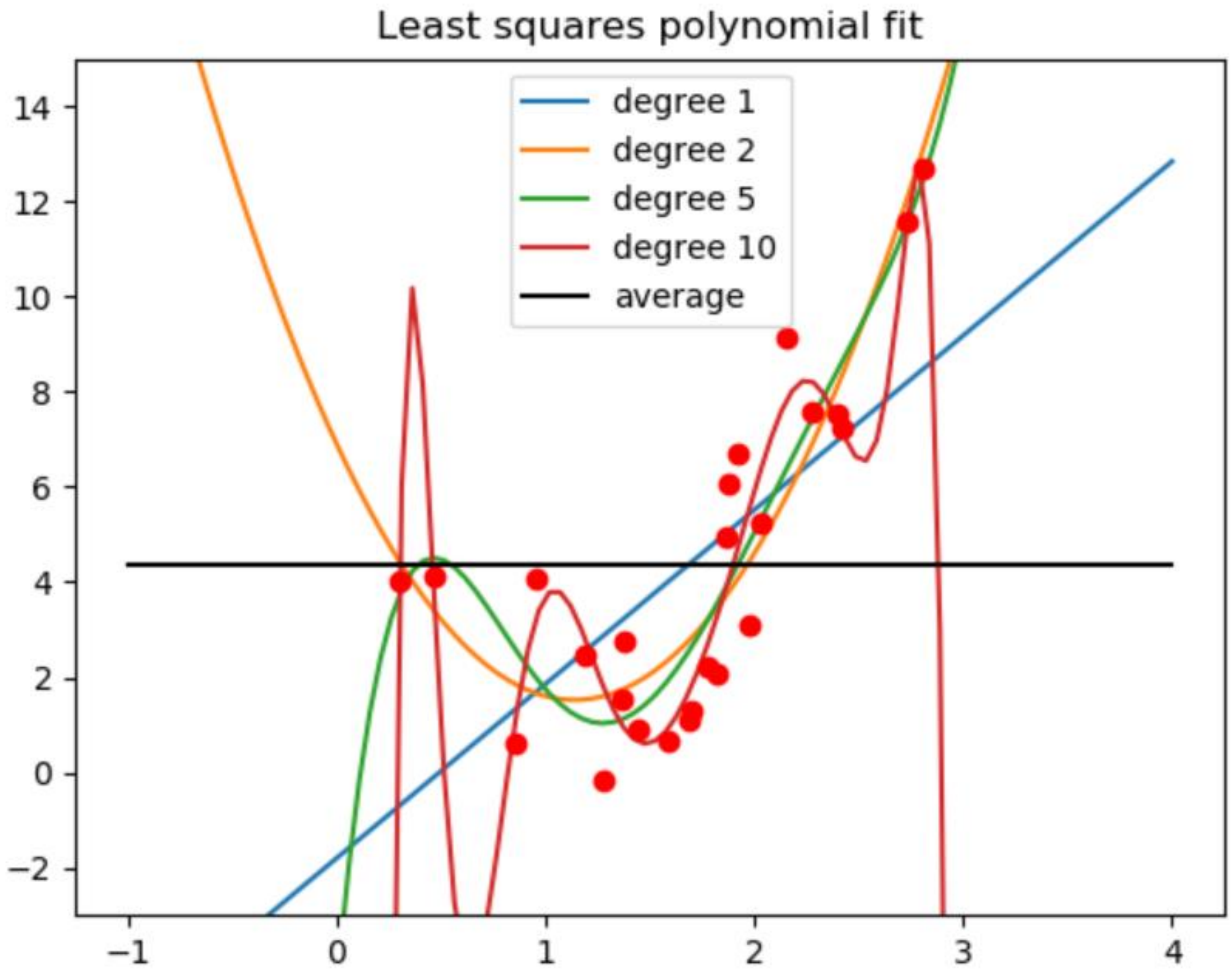
$$\sum_{i=0}^{n-1} (y_i - p(x_i))^2$$

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

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

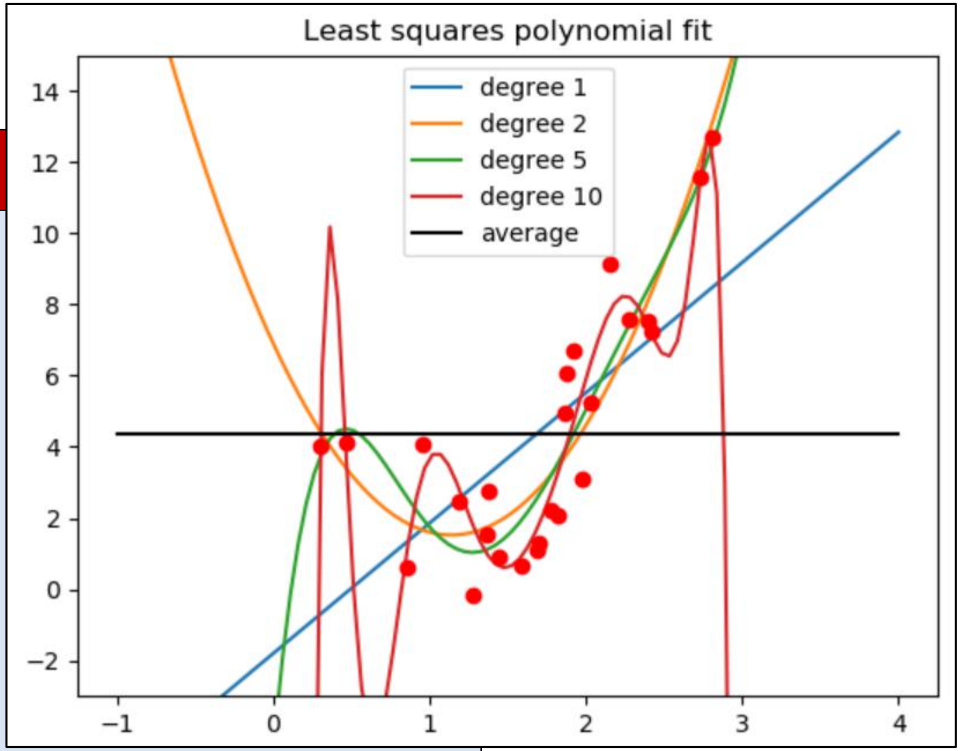
```
import matplotlib.pyplot as plt
import numpy as np

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()
```

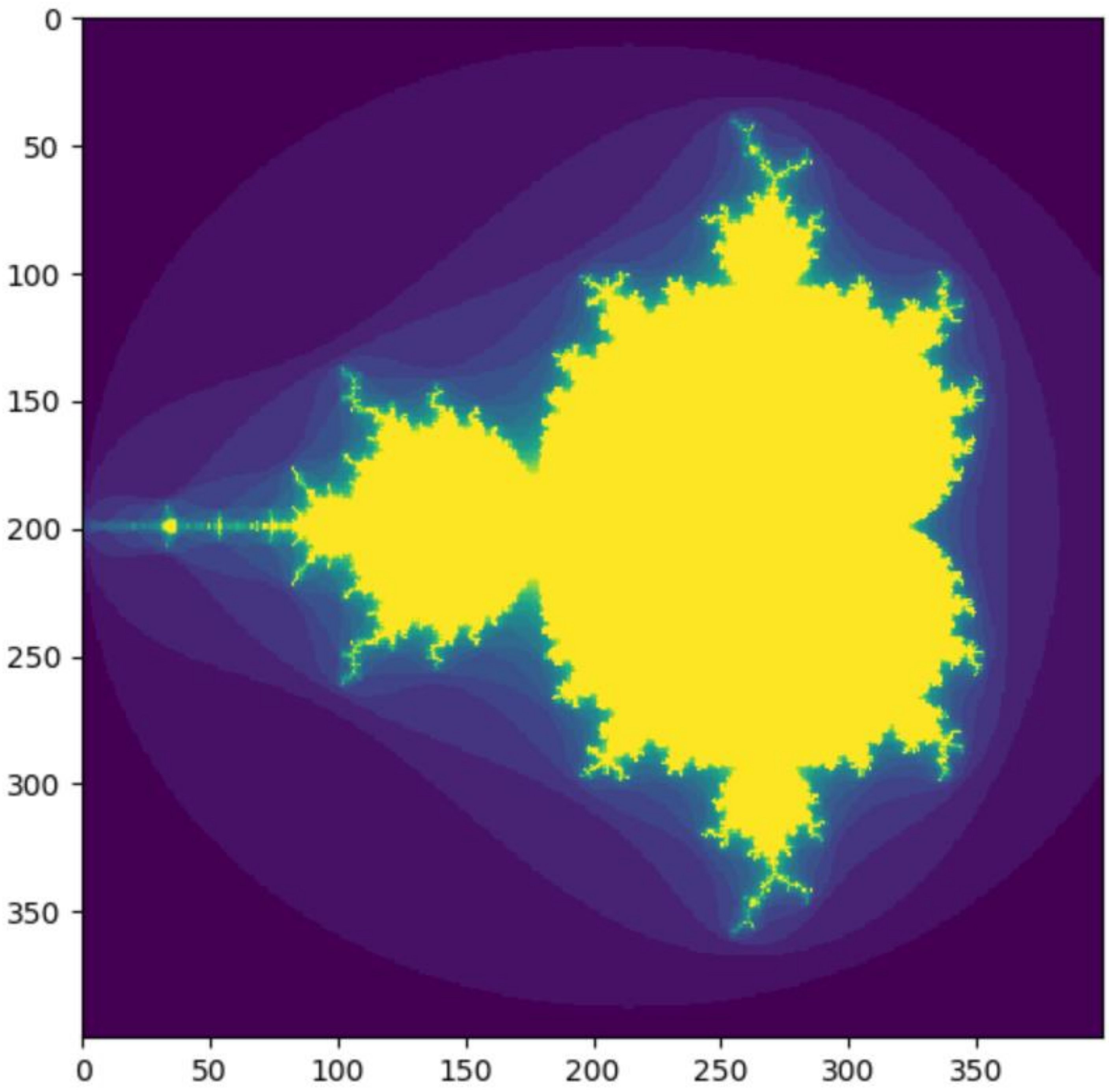



Least squares polynomial fit
degree 1
degree 2
degree 5
degree 10
average
14
12
10
8
6
4
2
0
−2
−1
0
1
2
3
4

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="degree %s" % degree)

avg = np.average(y)
plt.plot(x, y, 'ro')
plt.plot([-1, 4], [avg, avg], 'k-', label="average")
ax = plt.gca()
ax.set_ylim(-3, 15)
plt.title('Least squares polynomial fit')
plt.legend()
plt.show()
```




0
50
100
150
200
250
300
350
0
50
100
150
200
250
300
350

mandelbrot.py

```
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)."""
    y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
    c = x+y*1j
    z = c
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
        z = z**2 + c
        diverge = z*np.conj(z) > 2**2            # who is diverging
        div_now = diverge & (divtime==maxit)  # who is diverging now
        divtime[div_now] = i                     # note when
        z[diverge] = 2                           # avoid diverging too much

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

code from https://docs.scipy.org/doc/numpy-dev/user/quickstart.html