CME 193: Introduction to Scientific Python

Lecture 5: Scipy, Matplotlib, and Recursion

Sven Schmit

`stanford.edu/~schmit/cme193`

Contents

What is SciPy?

SciPy is a library of algorithms and mathematical tools built to work with NumPy arrays.

- linear algebra - *scipy.linalg*
- statistics - *scipy.stats*
- optimization - *scipy.optimize*
- sparse matrices - *scipy.sparse*
- signal processing - *scipy.signal*
- etc.

Scipy Linear Algebra

Slightly different from numpy.linalg. Always uses BLAS/LAPACK support, so could be faster.

Some more functions.

Functions can be slightly different.

Scipy Optimization

- General purpose minimization: CG, BFGS, least-squares
- Constrainted minimization; non-negative least-squares
- Minimize using simulated annealing
- Scalar function minimization
- Root finding
- Check gradient function
- Line search

Scipy Statistics

- Mean, median, mode, variance, kurtosis
- Pearson correlation coefficient
- Hypothesis tests (ttest, Wilcoxon signed-rank test, Kolmogorov-Smirnov)
- Gaussian kernel density estimation

See also SciKits (or scikit-learn).

Scipy sparse

- Sparse matrix classes: CSC, CSR, etc.
- Functions to build sparse matrices
- sparse.linalg module for sparse linear algebra
- sparse.csgraph for sparse graph routines

Scipy signal

- Convolutions
- B-splines
- Filtering
- Continuous-time linear system
- Wavelets
- Peak finding

Scipy IO

Methods for loading and saving data

- Matlab files
- Matrix Market files (sparse matrices)
- Wav files

Contents

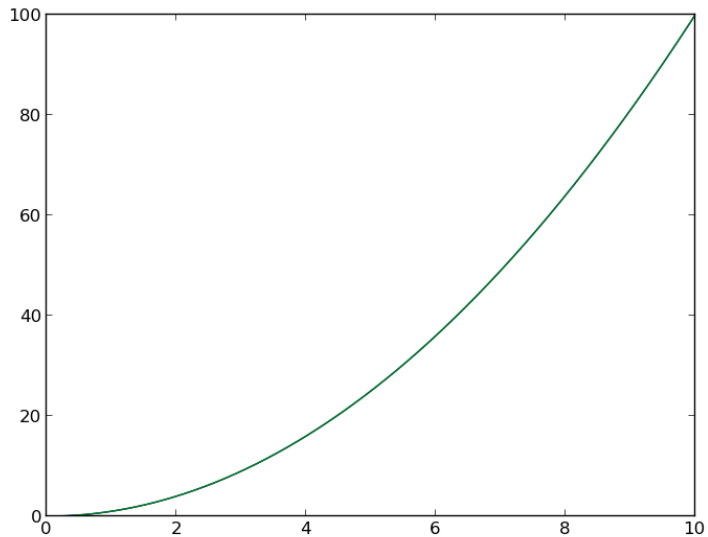
What is Matplotlib?

matplotlib.org: matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB or Mathematica), web application servers, and six graphical user interface toolkits.

- matplotlib is the standard Python plotting library
- We will prmarily be using `matplotlib.pyplot`
- Can create histograms, power spectra, bar charts, errorcharts, scatterplots, etc with a few lines of code

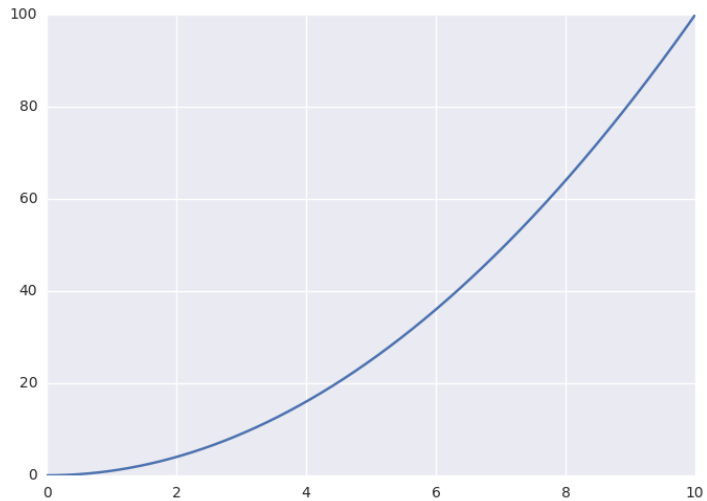
Scatter Plot

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

x = np.linspace(0, 10, 1000)
y = np.power(x, 2)
plt.plot(x, y)
plt.show()
```

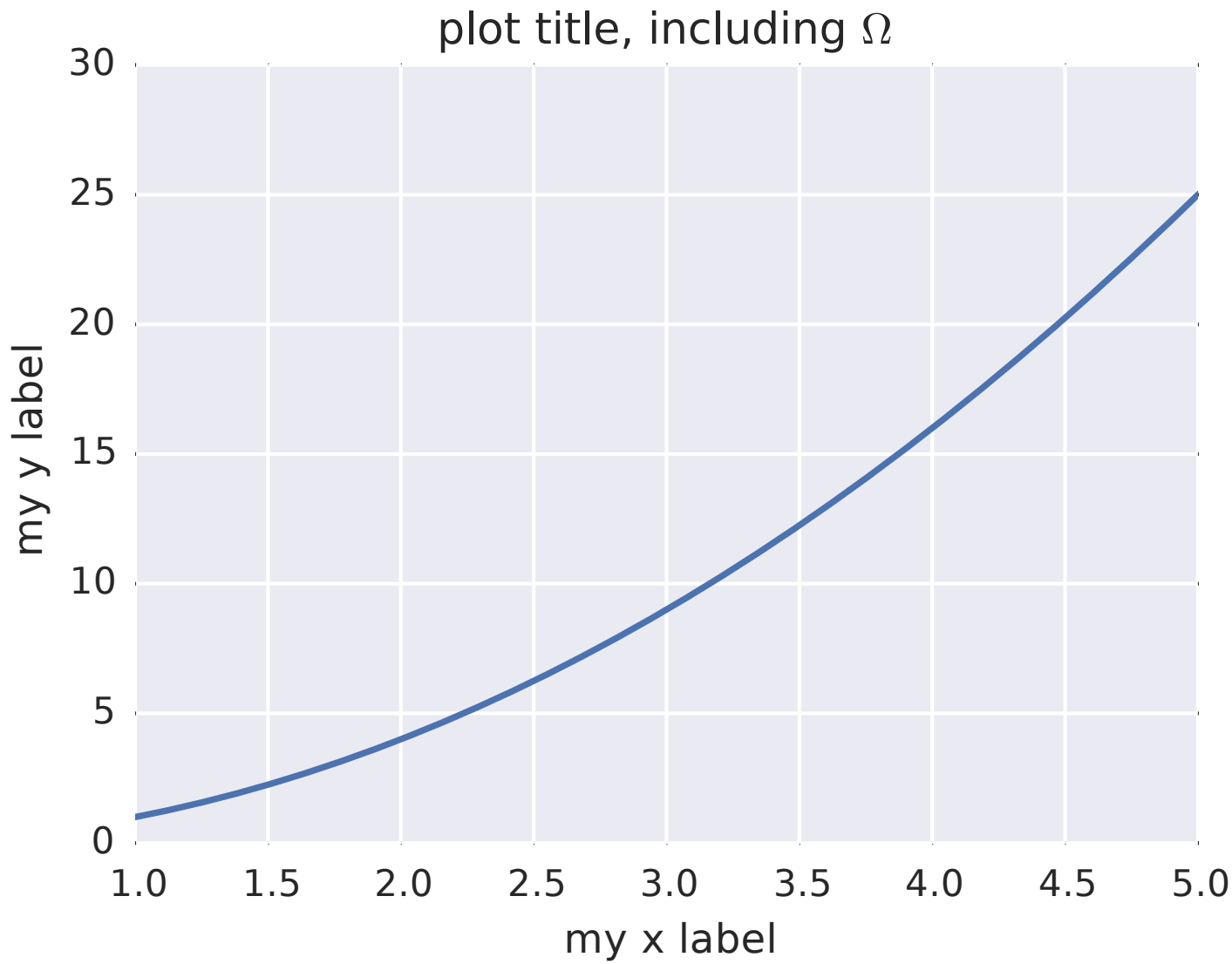


Seaborn makes plot pretty

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

x = np.linspace(0, 10, 1000)
y = np.power(x, 2)
plt.plot(x, y)
plt.show()
```



Scatter Plot

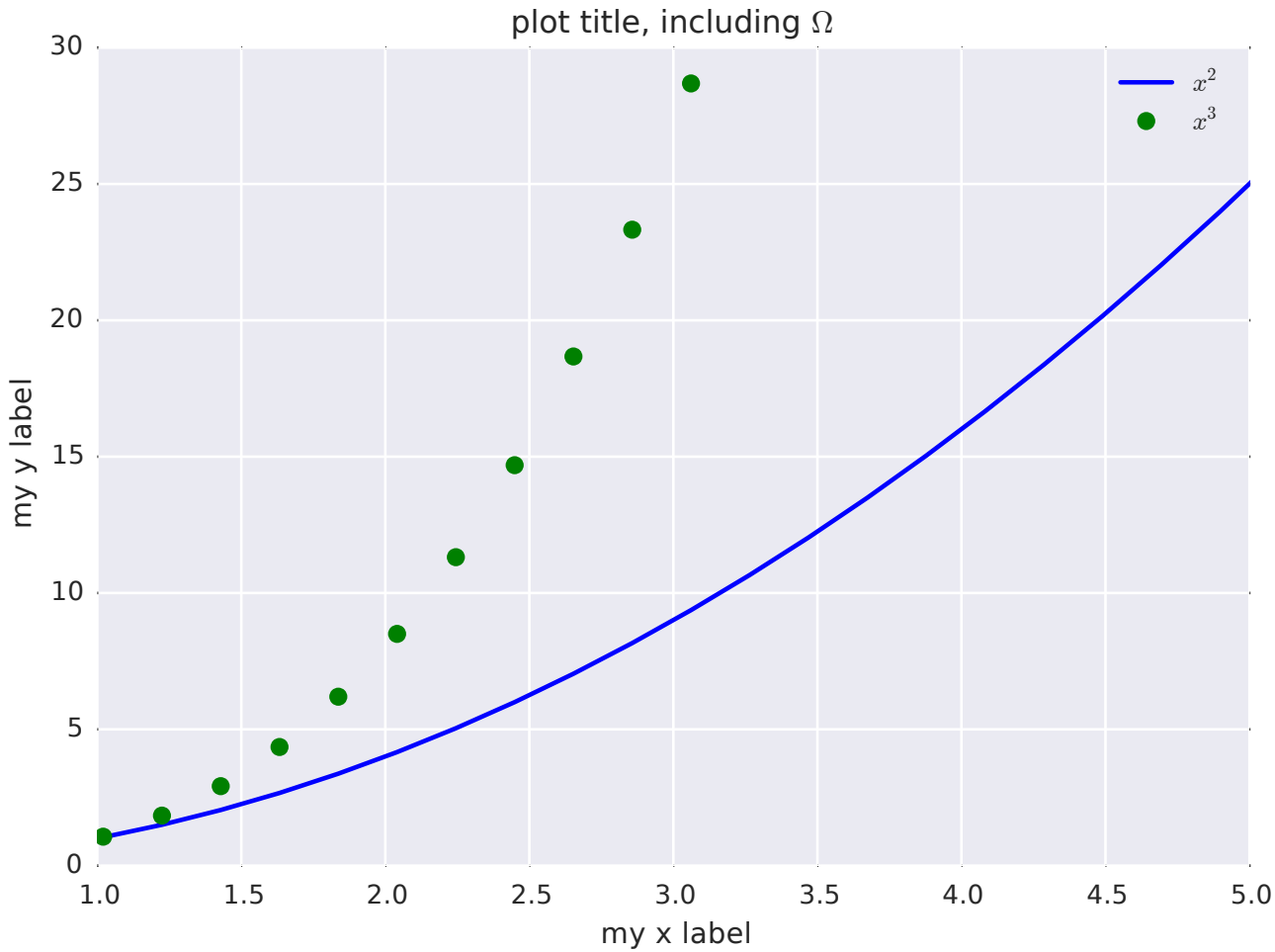


Scatter Plot

Adding titles and labels

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

f, ax = plt.subplots(1, 1, figsize=(5,4))

x = np.linspace(0, 10, 1000)
y = np.power(x, 2)
ax.plot(x, y)
ax.set_xlim((1, 5))
ax.set_ylim((0, 30))
ax.set_xlabel('my x label')
ax.set_ylabel('my y label')
ax.set_title('plot title, including $\Omega$')

plt.tight_layout()
plt.savefig('line_plot_plus.pdf')
```

Scatter Plot

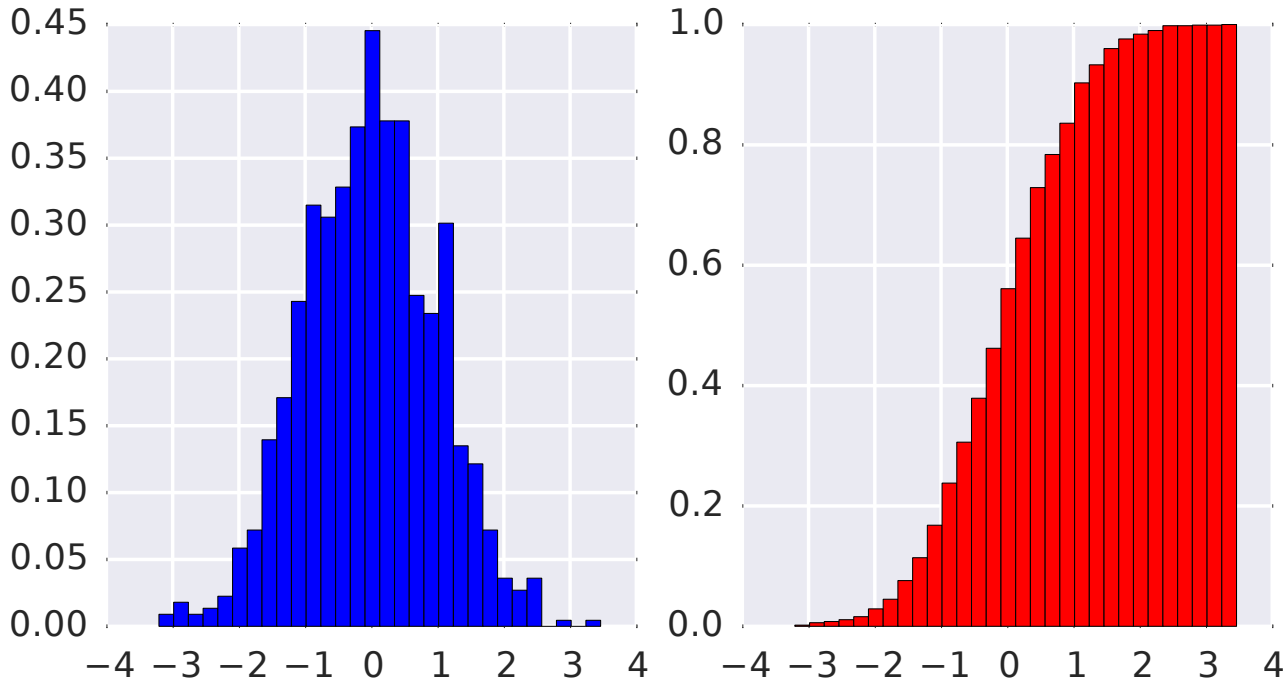


Scatter Plot

Adding multiple lines and a legend

```
x = np.linspace(0, 10, 50)
y1 = np.power(x, 2)
y2 = np.power(x, 3)

plt.plot(x, y1, 'b-', label='$x^2$')
plt.plot(x, y2, 'go', label='$x^3$')
plt.xlim((1, 5))
plt.ylim((0, 30))
plt.xlabel('my x label')
plt.ylabel('my y label')
plt.title('plot title, including $\Omega$')
plt.legend()

plt.savefig('line_plot_plus2.pdf')
```

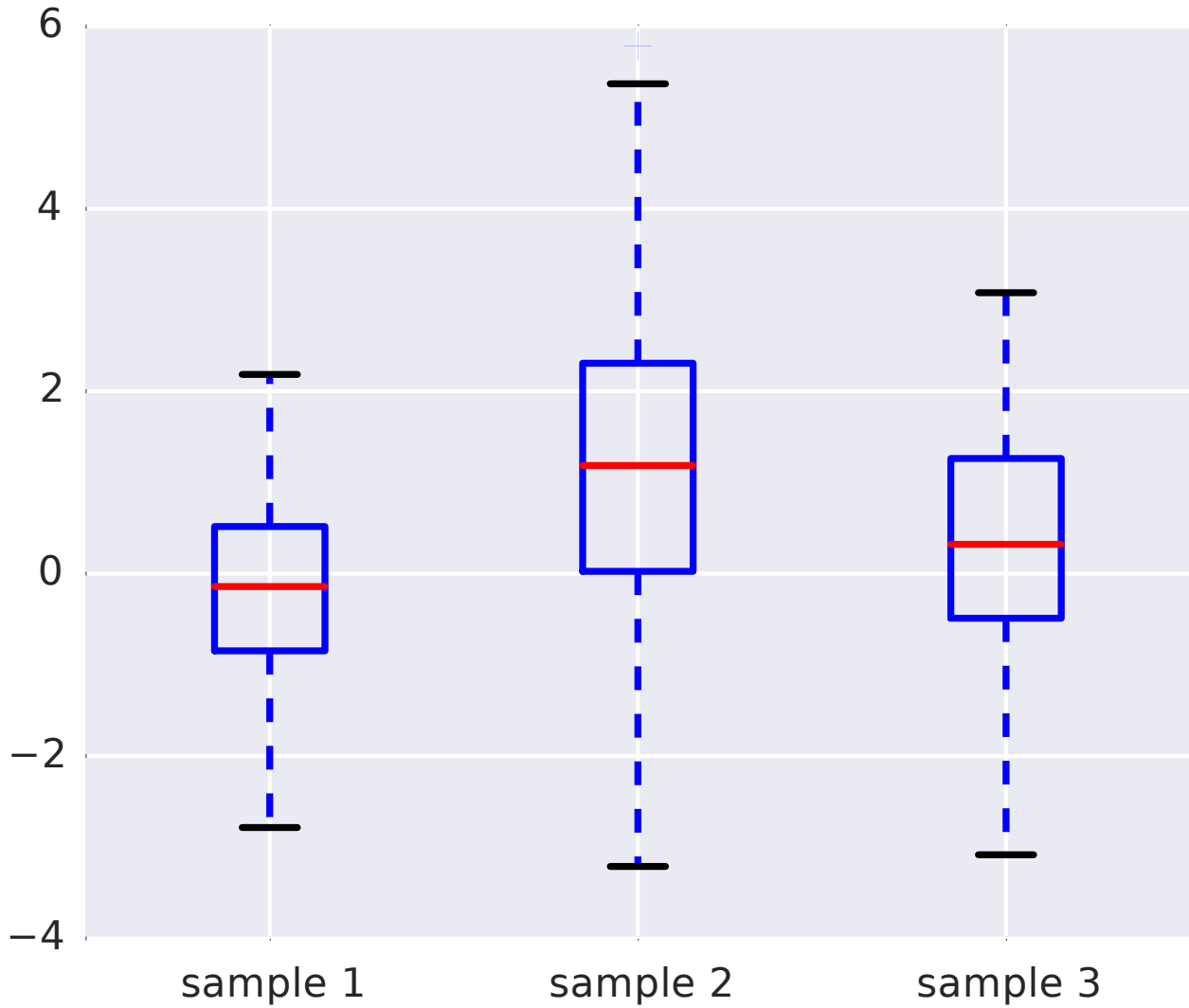
Histogram


0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4 −3 −2 −1 0 1 2 3 4
1.0
0.8
0.6
0.4
0.2
0.0
−4 −3 −2 −1 0 1 2 3 4

Histogram

```
data = np.random.randn(1000)

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(6,3))

# histogram (pdf)
ax1.hist(data, bins=30, normed=True, color='b')

# empirical cdf
ax2.hist(data, bins=30, normed=True, color='r',
         cumulative=True)

plt.savefig('histogram.pdf')
```

Box Plot



Box Plot

```
samp1 = np.random.normal(loc=0., scale=1., size=100)
samp2 = np.random.normal(loc=1., scale=2., size=100)
samp3 = np.random.normal(loc=0.3, scale=1.2, size=100)

f, ax = plt.subplots(1, 1, figsize=(5,4))

ax.boxplot((samp1, samp2, samp3))
ax.set_xticklabels(['sample 1', 'sample 2', 'sample 3'])
plt.savefig('boxplot.pdf')
```

Image Plot

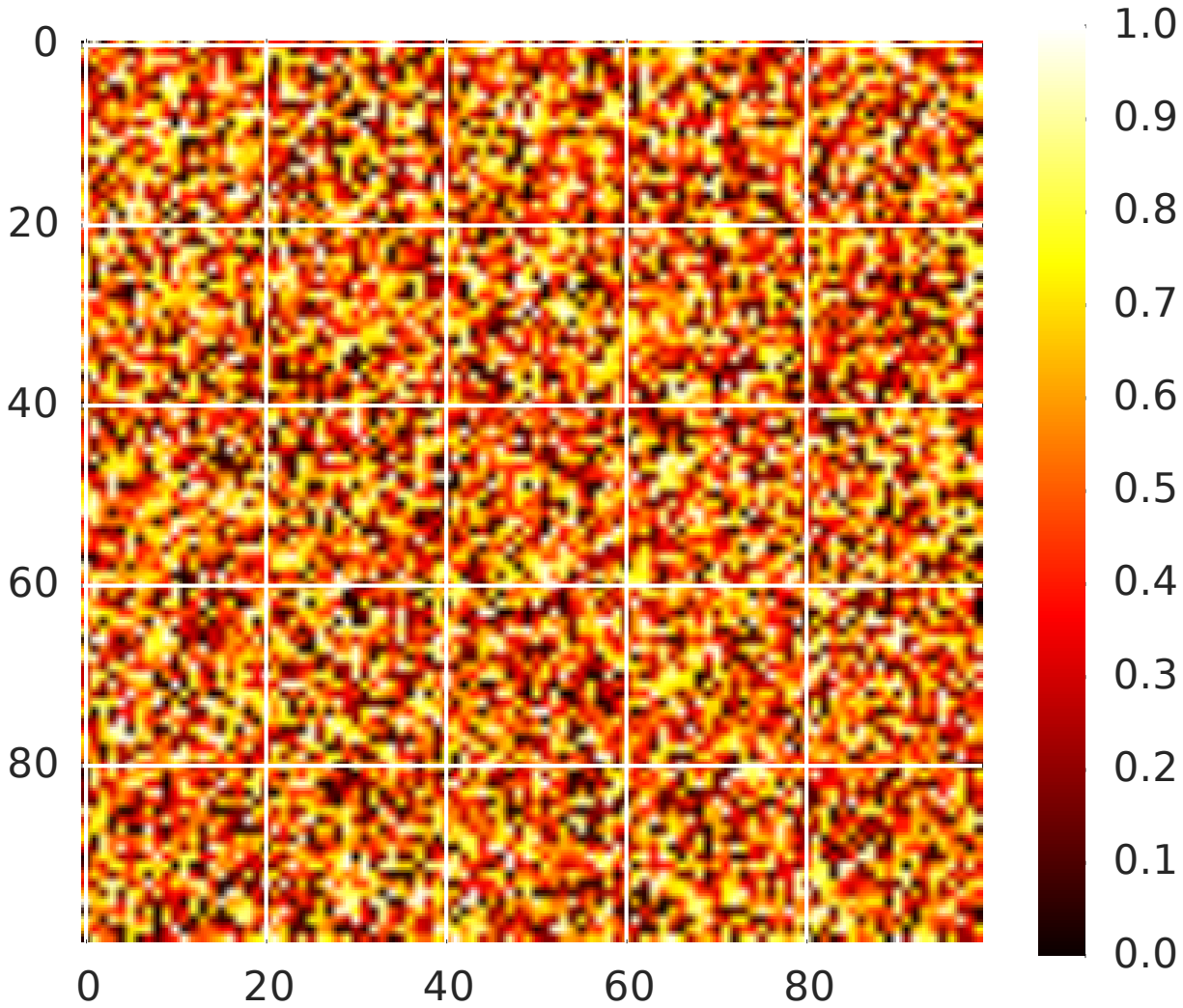
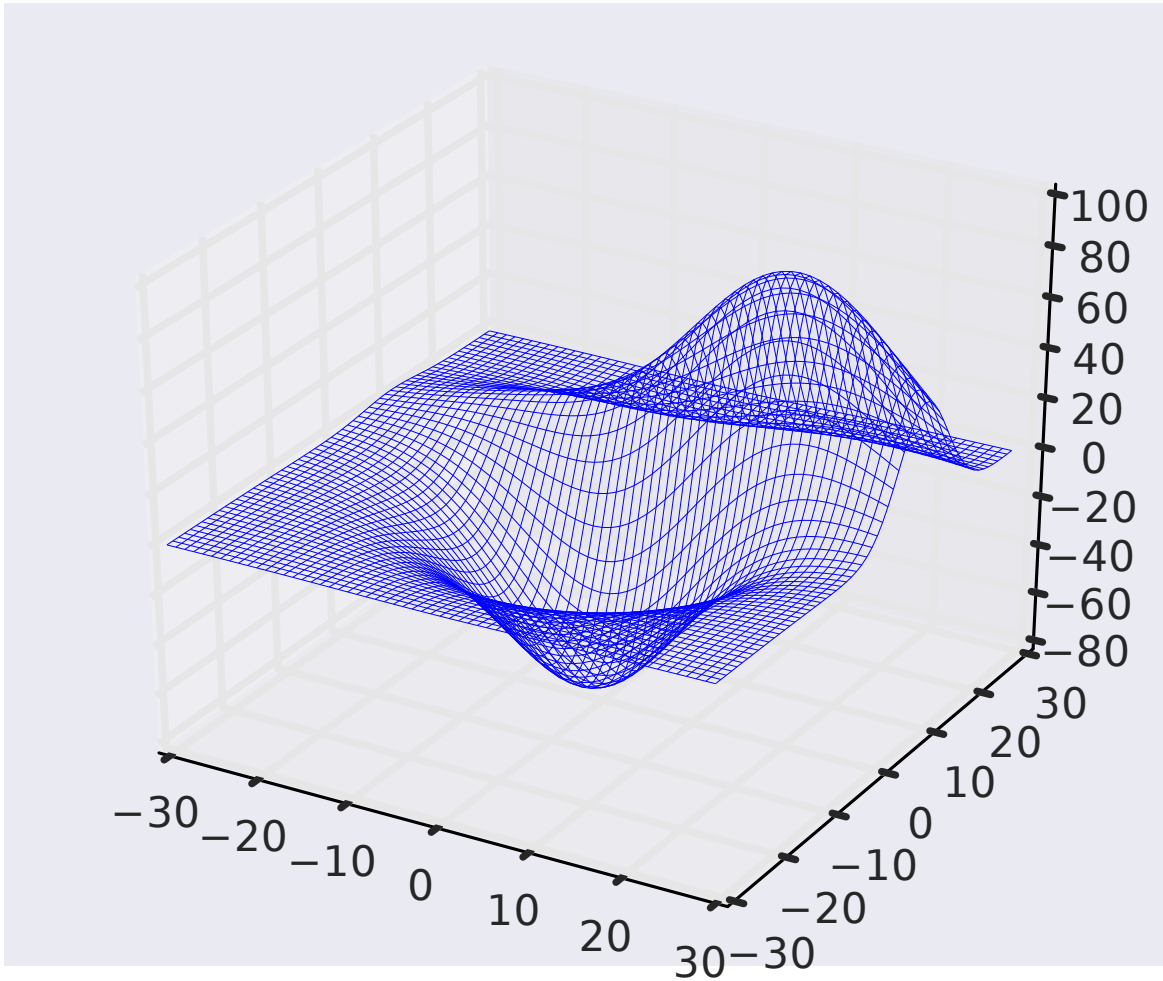


Image Plot

```
A = np.random.random((100, 100))

plt.imshow(A)
plt.hot()
plt.colorbar()

plt.savefig('imageplot.pdf')
```

Wire Plot



Wire Plot

matplotlib toolkits extend funtionality for other kinds of visualization

```
from mpl_toolkits.mplot3d import axes3d

ax = plt.subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.1)
ax.plot_wireframe(X, Y, Z, linewidth=0.1)

plt.savefig('wire.pdf')
```

Contents

Back to control flow

To execute repetitive code, we have relied on `for` and `while` loops.

Furthermore, we used `if` statements to handle conditional statements.

These statements are rather straightforward and easy to understand.

Recursion

Recursive function solve problems by reducing them to smaller problems of the same form.

Hence, recursive functions call themselves.

- New paradigm
- Powerful tool
- Divide-and-conquer
- Beautiful

Recursion

Recursive function solve problems by reducing them to smaller problems of the same form.

Hence, recursive functions call themselves.

- New paradigm
- Powerful tool
- Divide-and-conquer
- Beautiful

First example

Suppose we want to add two numbers a and b, but we can only add or subtract one.

First example

Non-recursive solution:

```
def add(a, b):
    while b > 0:
        a += 1
        b -= 1
    return a
```

First example

Recursive solution:

- Simple case: b = 0, return a
- Else, we can return 1 + add(a, b-1)

```
def add(a, b):
    if b == 0:
        # base case
        return a
    # recursive step
    return add(a, b-1) + 1
```

First example

Recursive solution:

- Simple case: b = 0, return a
- Else, we can return 1 + add(a, b-1)

```
def add(a, b):
    if b == 0:
        # base case
        return a
    # recursive step
    return add(a, b-1) + 1
```

Base case and recursive steps

Recursive functions consist of two parts:

base case The base case is the trivial case that can be dealt with easily.

recursive step The recursive step brings us slightly closer to the base case and calls the function itself again.

Palindromes

A palindrome is a word that reads the same from both ways, such as *radar* or *level*.

Let's write a function that checks whether a given word is a palindrome.

The recursive idea

Given a word, such as *level*, we check:

- whether the first and last character are the same
- whether the string with first and last character removed are the same

Base case

What's the base case in this case?

- The empty string is a palindrome
- Any 1 letter string is a palindrome

Base case

What's the base case in this case?

- The empty string is a palindrome
- Any 1 letter string is a palindrome

Implementation

```
def is_palin(s):
    '''returns True iff s is a palindrome'''
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and is_palin(s[1:-1])
```

What is an iterative solution?

Implementation

```
def is_palin(s):
    '''returns True iff s is a palindrome'''
    if len(s) <= 1:
        return True
    return s[0] == s[-1] and is_palin(s[1:-1])
```

What is an iterative solution?

Numerical integration

Suppose we want to numerically integrate some function f:

$$A = \int_a^b f(x)\,dx$$

Trapezoid rule:

$$\begin{aligned}
A &= \int_a^b f(x)\,dx \\
&= \int_a^{r_1} f(x)\,dx + \int_{r_1}^{r_2} f(x)\,dx + \ldots + \int_{r_{n-1}}^{b} f(x)\,dx \\
&\approx \frac{h}{2}\left((f(a)+f(r_1)) + (f(r_1)+f(r_2)) + \ldots + (f(b)+f(r_{n-1}))\right) \\
&= \frac{h}{2}(f(a)+f(b)) \quad + \quad h(f(r_1)+f(r_2)+\ldots+f(r_{n-1}))
\end{aligned}$$

Numerical integration

Suppose we want to numerically integrate some function f:

$$A = \int_a^b f(x)\,dx$$

Trapezoid rule:

$$\begin{aligned}
A &= \int_a^b f(x)\,dx \\
&= \int_a^{r_1} f(x)\,dx + \int_{r_1}^{r_2} f(x)\,dx + \ldots + \int_{r_{n-1}}^{b} f(x)\,dx \\
&\approx \frac{h}{2}\left((f(a)+f(r_1)) + (f(r_1)+f(r_2)) + \ldots + (f(b)+f(r_{n-1}))\right) \\
&= \frac{h}{2}(f(a)+f(b)) \quad + \quad h(f(r_1)+f(r_2)+\ldots+f(r_{n-1}))
\end{aligned}$$

Trapezoid rule

```
def trapezoid(f, a, b, N):
    '''integrates f over [a,b] using N steps'''
    if a > b:
        a, b = b, a
    # step size
    h = float(b-a)/N
    # running sum
    s = h/2 * (f(a) + f(b))
    for k in xrange(1, N-1):
        s += h * f(a + h*k)
    return s
```

Adaptive integration

Idea: Adaptively space points based on local curvature of function

```
def ada_int(f, a, b, tol=1.0e-6, n=5, N=10):
    area = trapezoid(f, a, b, N)
    check = trapezoid(f, a, b, n)
    if abs(area - check) > tol:
        # bad accuracy, add more points to interval
        m = (b + a) / 2.0
        area = ada_int(f, a, m) + ada_int(f, m, b)
    return area
```

Note: we do not need to use trapezoid rule.

Adaptive integration

Idea: Adaptively space points based on local curvature of function

```
def ada_int(f, a, b, tol=1.0e-6, n=5, N=10):
    area = trapezoid(f, a, b, N)
    check = trapezoid(f, a, b, n)
    if abs(area - check) > tol:
        # bad accuracy, add more points to interval
        m = (b + a) / 2.0
        area = ada_int(f, a, m) + ada_int(f, m, b)
    return area
```

Note: we do not need to use trapezoid rule.

Pitfalls

Recursion can be very powerful, but there are some pitfalls:

- Have to ensure you always reach the base case.
- Each successive call of the algorithm must be solving a simpler problem
- The number of function calls shouldn't explode. (see exercises)
- An iterative algorithm is always faster due to overhead of function calls. (However, the iterative solution might be much more complex)

Contents

Exercises