Computational Physics

Programming Style and Practices &
Visualizing Data via Plotting

Prof. Paul Eugenio
Department of Physics
Florida State University
Jan 29, 2019

http://hadron.physics.fsu.edu/~eugenio/comphy/

Announcements

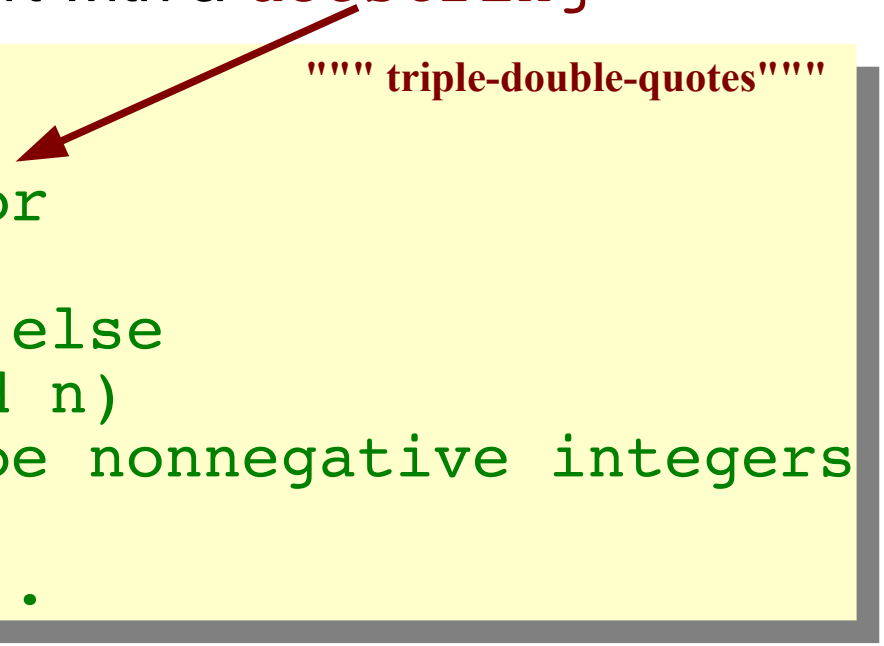
Reading Chapter 4

- Accuracy and Speed: pages 126 – 137
- Turn-In Questions
  - 2 questions on reading due next Tuesday

Programming Conventions to follow in this class

Documentation via `docstring`

- Include documenting comments in your programs
  - Each program should include a prolog comment block or `docstring`
  - Each function should start with a `docstring`



The use of docstring will have helpful advantages later on

Program Prolog via docstring

```
#! /usr/bin/env python
"""
 gcd.py calculates the greatest common divisor g(m,n)
 of two nonnegative integers m and n using Euclid prescription.


 Paul Eugenio
 PHZ4151C
 Exercise 2, Problem 4
 Jan 28, 2019
"""

from __future__ import division, print_function
```

Functions, variables, and constants

- Use meaningful variable names
  - Function names: lower_case_with_underscores
    - Examples: `catalan(), generate_data()`
    - Names to AVOID
      - l (lowercase el), O (uppercase oh), or I (uppercase eye)

Functions, variables, and constants

- Use meaningful variable names
  - Function names: lower_case_with_underscores
    - Examples: `catalan(), generate_data()`
    - Names to AVOID
      - l (lowercase el), O (uppercase oh), or I (uppercase eye)
  - Variables: b, B, lower, mixedCase
    - Examples: `N, i,  j, maxAngularMomentum`

Functions, variables, and constants

- Use meaningful variable names
  - Function names: lower_case_with_underscores
    - Examples: `catalan(), generate_data()`
    - Names to AVOID
      - l (lowercase el), O (uppercase oh), or I (uppercase eye)
  - Variables: b, B, lower, mixedCase
    - Examples: `N, i,  j, maxAngularMomentum`
  - Constants: CAPITAL_WITH_UNDERSCORES
    - Examples: `RADIUS_EARTH`,
      - exceptions for common use single constants
        - i.e., `g = 9.81, c = 3e8`

Coding conventions

- Use the right type of variables
  - **`int`** for integer quantities: indices, quantum numbers, ..
  - **`float`** and **`complex`** for quantities that are real or complex
  - **`bool`** for truly boolean types

Coding conventions

- Use the right type of variables
  - **`int`** for integer quantities: indices, quantum numbers, ..
  - **`float`** and **`complex`** for quantities that are real or complex
  - **`bool`** for truly boolean types
- Import functions at the beginning of the program
  - easy to find, check, and add to them
  - ensures that functions are imported before use

Coding conventions

- Use the right type of variables
  - **`int`** for integer quantities: indices, quantum numbers, ..
  - **`float`** and **`complex`** for quantities that are real or complex
  - **`bool`** for truly boolean types
- Import functions at the beginning of the program
  - easy to find, check, and add to them
  - ensures that functions are imported before use
- Give your constants names
  - constants in your program should be given names and defined at the beginning for your program
    - G = 6.67384e-11 , Z = 28, BOHR_RADIUS = 5.2917721092e-11

Coding conventions

- Use the right type of variables
  - **`int`** for integer quantities: indices, quantum numbers, ..
  - **`float`** and **`complex`** for quantities that are real or complex
  - **`bool`** for truly boolean types
- Import functions at the beginning of the program
  - easy to find, check, and add to them
  - ensures that functions are imported before use
- Give your constants names
  - constants in your program should be given names and defined at the beginning for your program

```
G = 6.67384e-11                  # Newtonian constant of gravitation  [ m**2 /(kg * s**2) ]
Z = 28                            # Atomic Number for Nickel
BOHR_RADIUS = 5.2917721092e-11   #  units [m]
```

Divide and Conquer

- Employ user-defined function,
  - use for repeated and complicated operations
    - greatly increases the legibility of your programs
  - avoid overuse
    - a single line or two of code is often better left in the main part of the program

Good Programming Practices

- **Print out partial results & update throughout your program**
  - define a function and test it

Good Programming Practices

- **Print out partial results & update throughout your program**
  - define a function and test it
- **Lay out your programs clearly**
  - most of the work should NOT be in the main part of the program
    - i.e. Divide and Conquer!

Good Programming Practices

- **Print out partial results & update throughout your program**
  - define a function and test it
- **Lay out your programs clearly**
  - most of the work should NOT be in the main part of the program
    - i.e. Divide and Conquer!
- **Don't make your programs unnecessarily complicated**

Visualizing Data via Plotting

Plotting with Matplotlib

Matplotlib is the standard package for plotting in Python

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

def crossSection(t):
    return t*np.exp(-t)

x = np.linspace(0, 10, 101)
y = np.zeros(len(x))
for i in range(len(x)):
    y[i] = crossSection(x[i])

plt.plot(x, y)
plt.savefig("crossSection.png")
plt.show()
```

pyplot *is a module in* ***matplotlib*** *package*

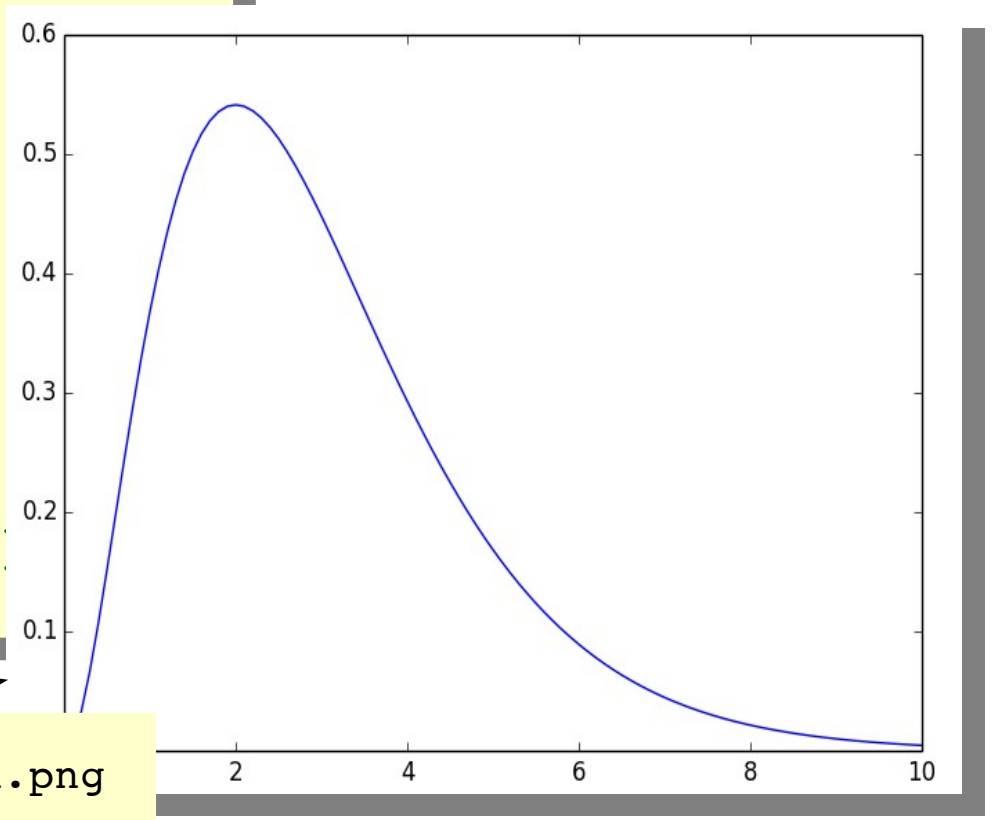
Create an array of linearly spaced values

Plotting with Matplotlib

Matplotlib is the standard package for plotting in Python

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

def crossSection(t, b=1):
    return t*np.exp(-b*t)

x = np.linspace(0, 10, 101
y = np.zeros(len(x))
for i in range(len(x)):
    y[i] = sigma(x[i])

plt.plot(x, y)
plt.savefig("crossSection.
plt.show()
```

create an array of linearly spaced values



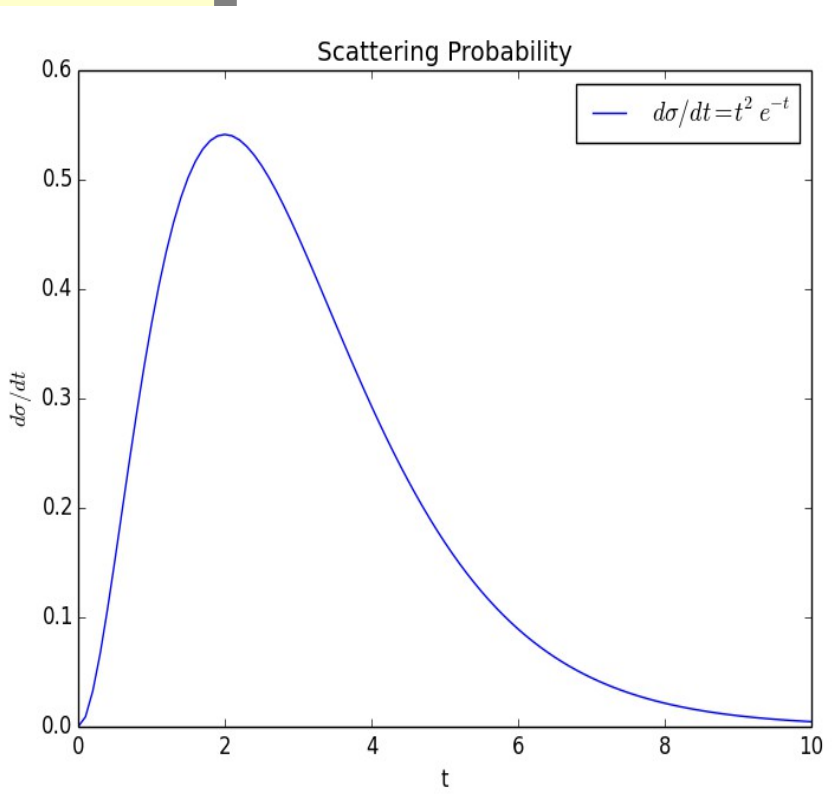

```
hpc-login-38 125% display crossSection.png
```

Decorating the Plot

See http://matplotlib.org for more examples

```
import matplotlib.pyplot as plt
plt.rc('text', usetex=True)


plt.plot(x,y)
plt.legend([r"$d\sigma/dt = t^2 e^{-t}$"])
plt.title("Scattering Probability")
plt.xlabel("t")
plt.ylabel(r"$d\sigma/dt$")
plt.savefig("sigma-t.png")
plt.show()
```

Use this when you want to render Latex equations



Equation rendering with LaTex

set up with:

```
matplotlib.pyplot.rc('text', usetex=True)
```

then use:

```
r"<Latex-Equation>"
```

Multiple Curves

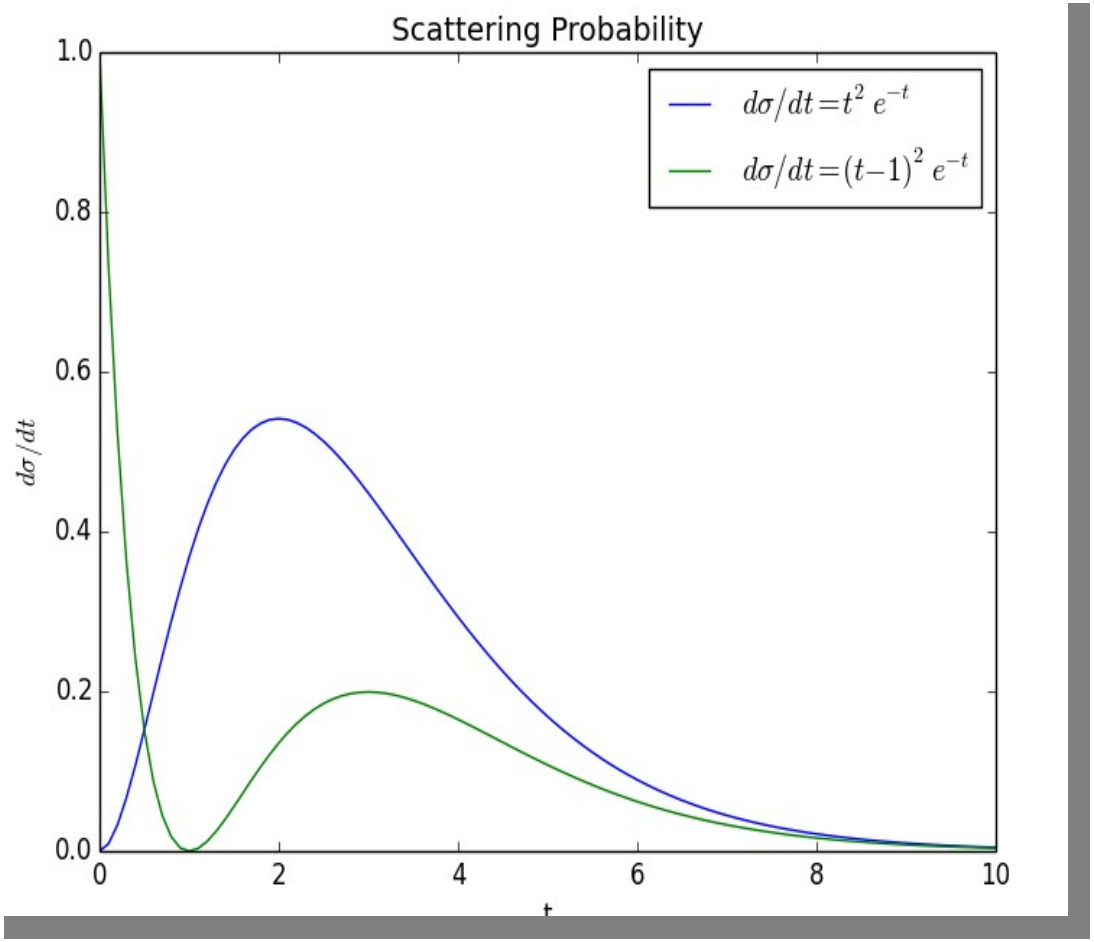
See http://matplotlib.org for more examples

```
b = 0
for i in range(len(x)):
    y[i] = crossSection(x[i], b)

plt.plot(x,y)



b = 1
for i in range(len(x)):
    y[i] = crossSection(x[i], b)

plt.plot(x,y)
```

Multiple Curves

See http://matplotlib.org for more examples

```
b = 0
for i in range(len(x)):
    y[i] = crossSection(x

plt.plot(x,y)



b = 1
for i in range(len(x)):
    y[i] = crossSection(x

plt.plot(x,y)
```



Matplotlib & Numpy Vectorization

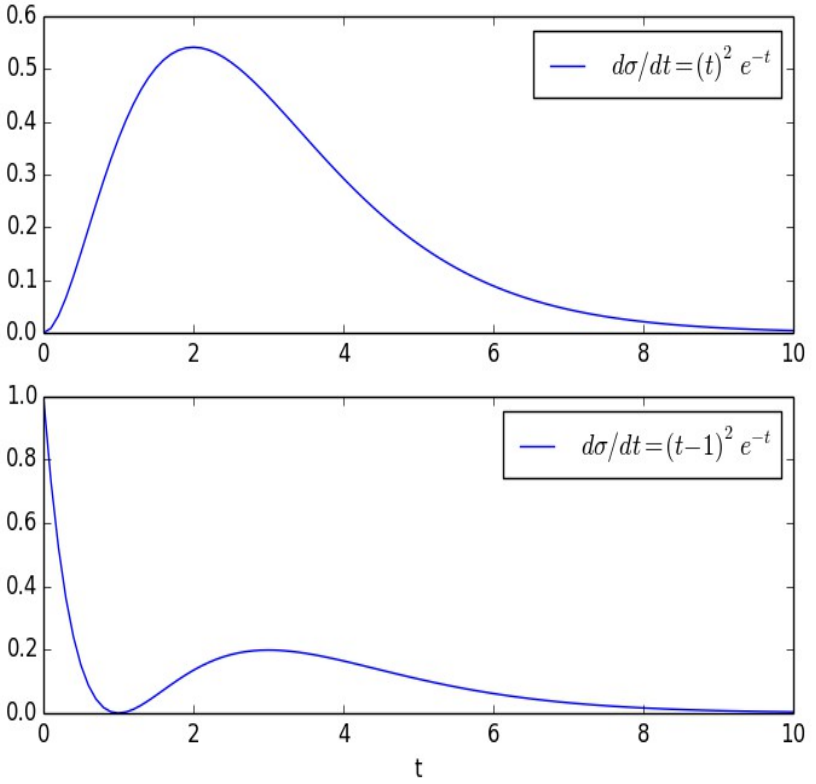
Python preferred programming

```
...


x = np.linspace(0,3,31)
y = crossSection(x)

plt.plot(x,y)

plt.title("Scattering")
plt.xlabel("t")
plt.ylabel(r"$d\sigma/dt$")

plt.savefig(“sigma.png”)
plt.show()
```

Notice the use of vectorization!

```
# data assignment by elements
x = np.linspace(0, 10, 101)
y = np.zeros(len(x))
for i in range(len(x)):
    y[i] = crossSection(x[i])
```

Multiple SubPlots



columns

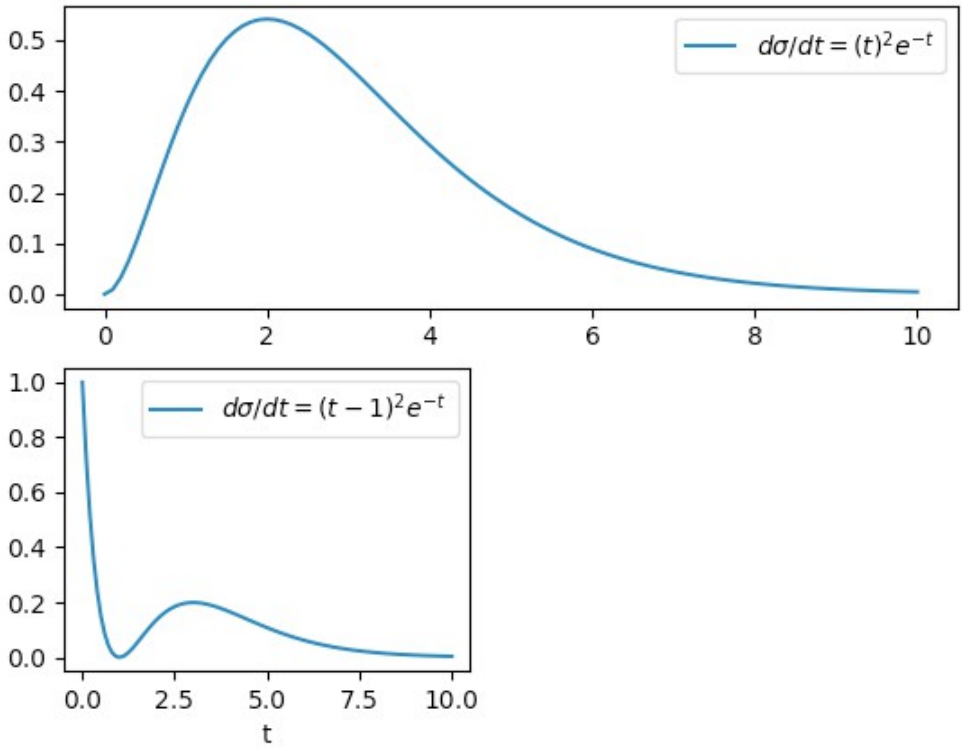
rows

plot #

```
plt.figure(1)

plt.subplot(211)

x = np.linspace(0,10,101)
b = 0
y = crossSection(x, b)
plt.plot(x,y)
plt.legend([r"$d\sigma/dt = (t)^2 e^{-t}$"])

b = 1
y = crossSection(x, b)

plt.subplot(212)
plt.plot(x,y)
plt.xlabel("t")
plt.legend([r"$d\sigma/dt = (t-1)^2 e^{-t}$"])

plt.show()
```

sets up
second plot

Multiple SubPlots



columns

rows

plot #

```
plt.figure(1)

plt.subplot(211)

x = np.linspace(0,10,101)
b = 0
y = crossSection(x, b)
plt.plot(x,y)
plt.legend([r"$d\sigma/dt = (t)^2 e^{-t}$"])

b = 1
y = crossSection(x, b)

plt.subplot(223)
plt.plot(x,y)
plt.xlabel("t")
plt.legend([r"$d\sigma/dt = (t-1)^2 e^{-t}$"])

plt.show()
```

sets up second plot with 2x2 spacing

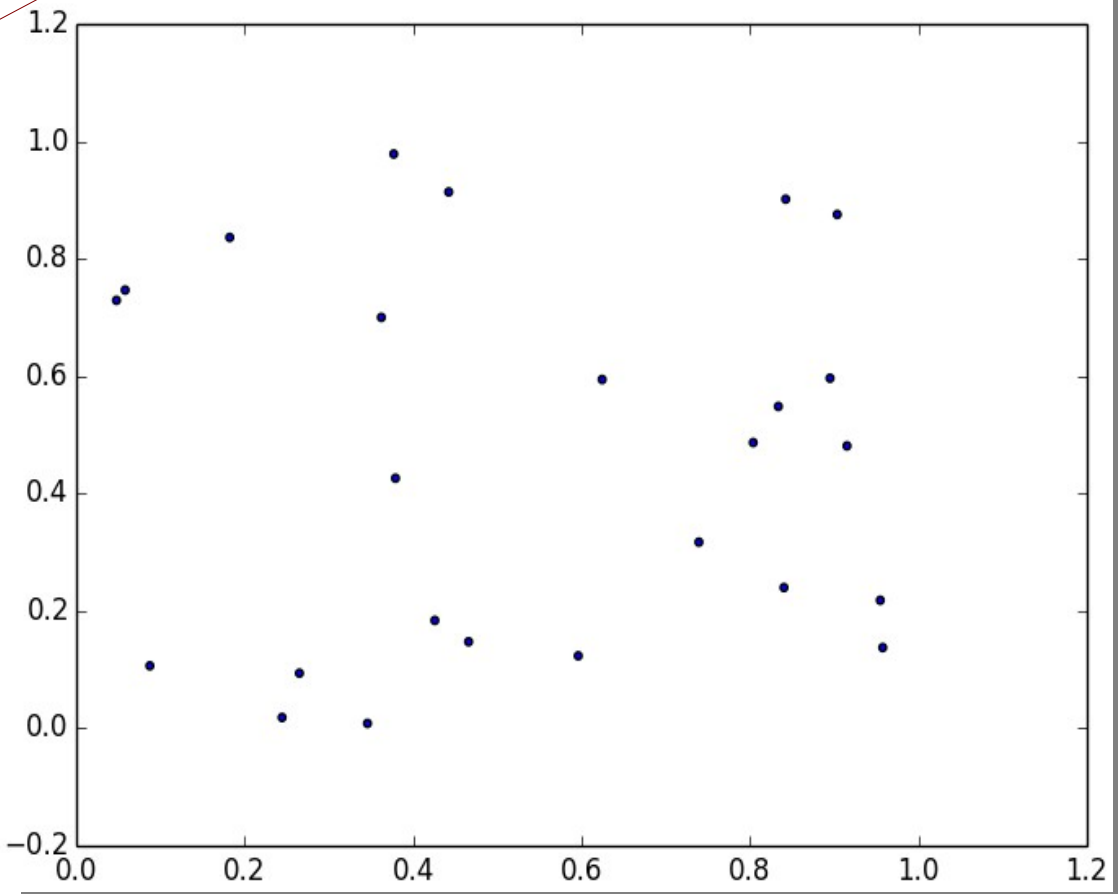
See Plotting Examples

- http://hadron.physics.fsu.edu/~eugenio/comphy/examples/
  - `plot1.py`
  - `plot2.py`
  - `plot3.py`
  - `plot4.py`

Scatter Plots

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

# Generate data...
x = np.random.random(25)
y = np.random.random(25)

# Plot...
plt.scatter(x, y, c="b", s=10)


plt.show()
```

point size

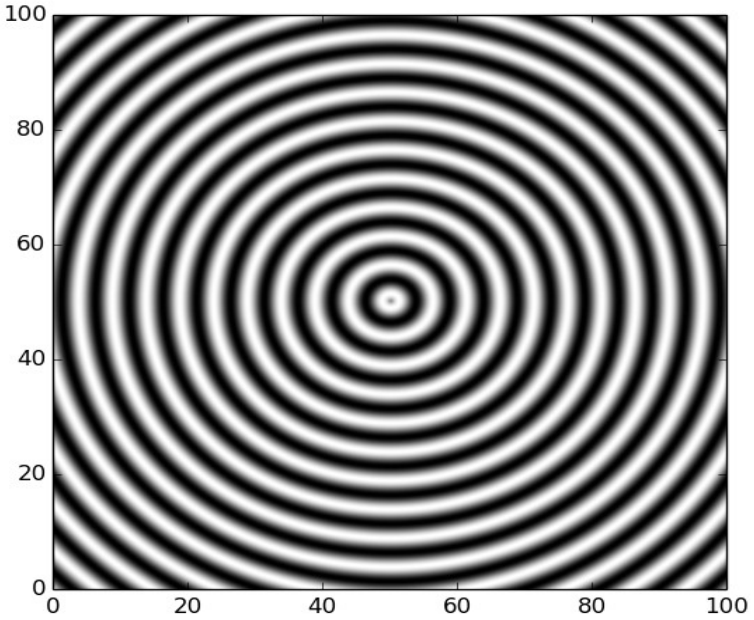


color

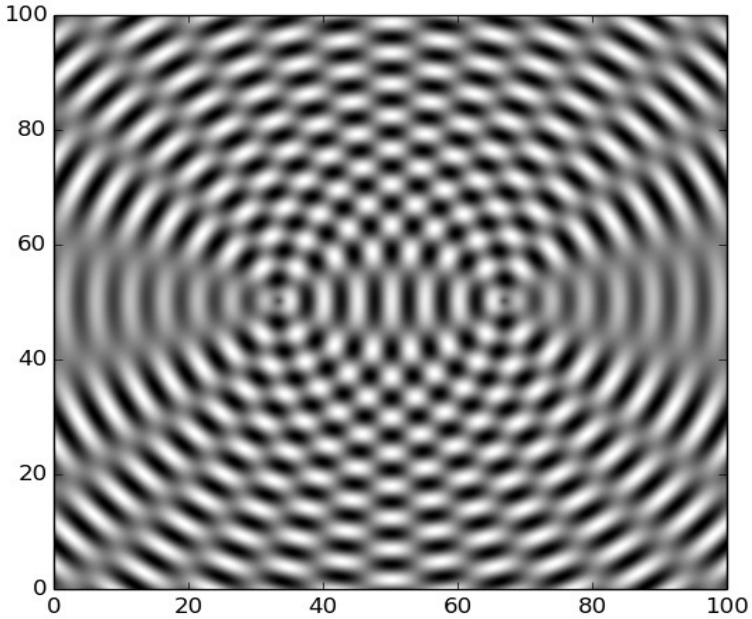
b: blue	m: magenta
g: green	y: yellow
r: red	k: black
c: cyan	w:white

Density Plots

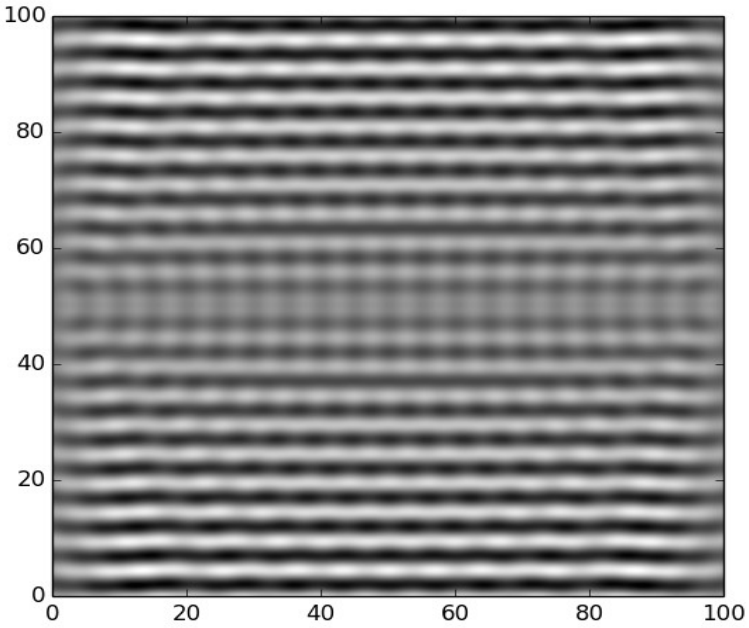
```
hpc-login 590% wavedrops.py
Enter a number of wave sources: 1
```




```
hpc-login 590% wavedrops.py
Enter a number of wave sources: 2
```




```
hpc-login 590% wavedrops.py
Enter a number of wave sources: 100
```



Density Plots

wavedrops.py

For brevity, all comments are left out
See Textbook example ripples.py for details

```
 #! /usr/bin/env python

from __future__ import division, print_function

import numpy as np
import matplotlib.pyplot as plt

waveLength = 5.0
k = 2*np.pi/waveLength
waveAmp = 1.0

size = 100.0
points = 500
spacing = size/points

numberOfSources = int(input("Enter a number of wave sources: "))

xSource = []
ySource = []
```

continued next slide

Density Plots

```
for i in range(numberOfSources):
    xSource += [(i+1)*size/(numberOfSources+1)]
    ySource += [size/2]

waveImage = np.zeros([points,points], float)

for i in range(points):
    y = spacing*i
    for j in range(points):
        x = spacing*j
        for n in range(numberOfSources):
            r = np.sqrt( (x-xSource[n])**2 + \
                         (y-ySource[n])**2 )
            waveImage[i,j] += waveAmp*np.sin(k*r)

plt.imshow(waveImage, origin="lower",
extent=[0,size,0,size])
plt.gray()

plt.savefig("density.png")
plt.show()
```

Let's get working