SciPy, NumPy, Matplotlib and Pyfits

Adrian Price-Whelan (NYU)

SciPy Package

- Includes NumPy
    - Linear Algebra
    - Random numbers
- Statistics package
- Integrated in Matplotlib, PyFITS
- Interpolation, integration, data io

NumPy

- Most fundamental object: **numpy.ndarray**
  - Can be multidimensional
  - Act like matrices, not like Python 'List's
  - Can be sliced, indexed, and iterated

```
arrayExample = numpy.array([1,2,3,4,5])
```

NumPy

- Numpy arrays store binary data, as described by the *numpy.dtype* objects
- *dtype* objects have a name and a string equivalent
    - np.int32 ⟷ i4
    - np.int64 ⟷ i8
    - np.float32 ⟷ f (32-bit floating point)
    - np.float64 ⟷ d (64-bit floating point)

NumPy

- *dtype* objects have a name and a string equivalent
  - np.int32 ⟷ i4
  - np.int64 ⟷ i8
  - np.float32 ⟷ f (32-bit floating point)
  - np.float64 ⟷ d (64-bit floating point)
- Can always convert shorter bit length to longer, but not the other way around

NumPy Arrays

- Indexed by [axis=0,axis=1,...,axis=n]
  - `apwArray[3,2]` ⟷ 92
- Sliced same as Lists
  - `apwArray[3:5,:2]` ⟷

1	8
7	6

  - `apwArray[:2,::2]` ⟷

0	4
5	2
7	6
6	5

apwArray

axis=1

axis=0

0	4	2
2	55	4
5	2	7
1	8	92
7	6	0
90	2	5
6	5	2

shape=(7,3)

NumPy Arrays

- `apwArray.size` ⟷ `21`
  - Total number of items
- `apwArray.shape` ⟷ `(7,3)`
  - Shape of array as Tuple
- `apwArray.ndim` ⟷ `2`
  - Num of dimensions
- Converting to List, or to array
  - `apwList = list(apwArray)`
  - `apwArray = numpy.array(apwList)`

apwArray

axis=1

axis=0

0	4	2
2	55	4
5	2	7
1	8	92
7	6	0
90	2	5
6	5	2

shape=(7,3)

NumPy Arrays

- Iterating over arrays can be done in many ways
- If the array is multidimensional, you have to determine how you want to iterate

apwArray

0	4	2
2	55	4
5	2	7

```
for row in apwArray:
    print row
```

→

0	4	2

2	55	4

5	2	7

```
for element in apwArray.flat:
    print element
```

→

0
4
2
2
55
4
5
2
7

Indexing with Boolean Arrays

- In IDL, the WHERE() function is probably what you have used most often

```
import numpy as np
foo = np.arange(10)

print foo[(foo > 2) & (foo < 7)]
```

[3 4 5 6]

  - Parentheses are important - the logical operators bind more tightly than comparison operators

- Can also index an array using an array

```
import numpy as np

spam = np.arange(30)[::3]
eggs = np.arange(100) * 4

print eggs[spam]
```

[0 12 24 36 48 60 72 84 96 108]

(basic) Linear Algebra

- `numpy.dot(array1,array2)`
  - Dot product between two arrays (matrix multiplication)
- Different from `array1*array2`
  - This operation works **element-wise**
- Similarly, `array1+array2`

$$\text{numpy.dot}\left(\begin{bmatrix}1 & 8\\ 7 & 6\end{bmatrix}, \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\right) \downarrow \begin{bmatrix}25 & 34\\ 25 & 38\end{bmatrix}$$

$$\begin{bmatrix}1 & 8\\ 7 & 6\end{bmatrix} * \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} \leftrightarrow \begin{bmatrix}1 & 16\\ 21 & 24\end{bmatrix}$$

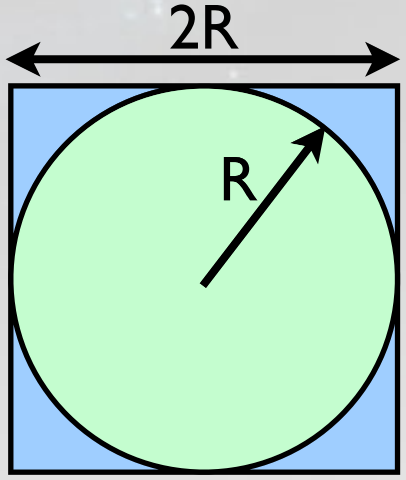
$$\begin{bmatrix}1 & 8\\ 7 & 6\end{bmatrix} + \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} \leftrightarrow \begin{bmatrix}2 & 10\\ 10 & 10\end{bmatrix}$$

numpy.random

- Useful package for handling random numbers
  - *rand, randint*
- Also includes statistical distributions
  - *normal, binomial, poisson, uniform*

Exercise: *PiExercise.py*

- One of the simplest examples of a Monte Carlo method is to use a random sampling to estimate the value of Pi
- If you draw N samples from a 2D square grid, and ask how many drawn points lie in a circle inscribed in the grid, ≈N x π/4 will lie in the circle



Circle Area = πR^2

Square Area = $4R^2$

Ratio Circle / Square = π / 4

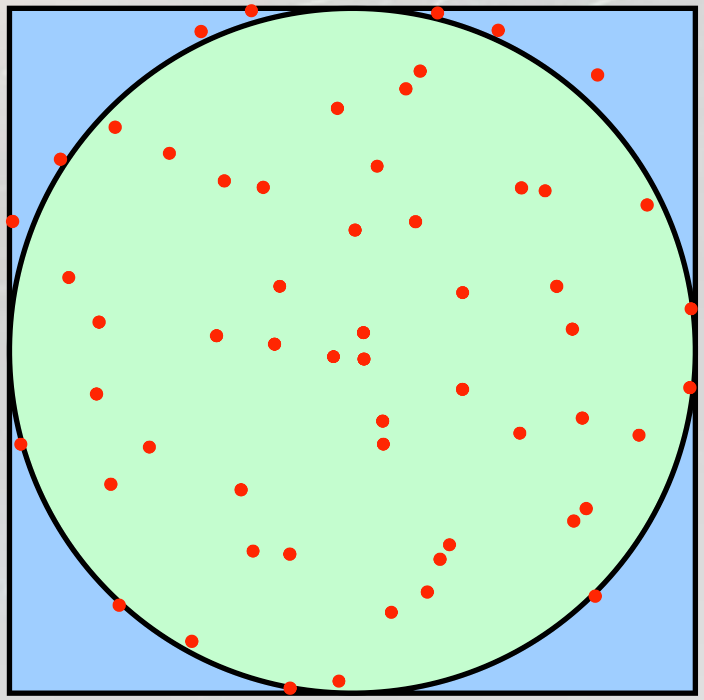
Exercise: *PiExercise.py*

- A visualization where red dots are individual draws

Number in Circle = 33

Total Number = 41

Pi ≈ 4 x 33 / 41 ≈ 3.2



Circle Area = πR^2

Square Area = $4R^2$

Ratio Circle / Square = π / 4

Exercise: *PiExercise.py*

```
import numpy as np

nSamples = 1000000
numInCircle = 0.0
for i in range(nSamples):
    x = np.random.rand()
    y = np.random.rand()

    if np.sqrt(x**2 + y**2) < 1.0:
        numInCircle += 1

pi = 4.0 * numInCircle / nSamples
print "One method, pi = %f" % pi
```

NumPy Arrays

- A few useful built in functions (there are *many* others, but you can look them up as you need them)
  - `numpy.zeros(shape,dtype)` (ndarray full of zeros)
  - `numpy.ones(shape,dtype)` (ndarray full of ones)
  - `numpy.eye(shape,dtype)` (Identity matrix)
  - `numpy.transpose(array)` or *arrayExample.T*
  - `numpy.linspace(start, stop, num_elements)`
  - `print numpy.linspace(0,1,5)`

    ↳ `array([0,0.25,0.5,0.75,1.0])`

Exercise: *PendExercise.py*

- ODE Integration with SciPy

```
from scipy import integrate
import numpy as np
import matplotlib.pyplot as plt

# Linearized Damped Harmonic Oscillator (pendulum)
# theta_vec:    vector (array) containing theta and dTheta/dt
# t:            time
# Q:            quality factor
def damped_pendulum(theta_vec,t,Q):
    theta_dot = theta_vec[1]
    theta = theta_vec[0]
    return np.array([theta_dot,-np.sin(theta)-theta_dot/Q])

# linspace works like in Matlab - pass it starting value, end value, and
#   how many timesteps in between
t = np.linspace(0,50,1000)
Qs = [0.5,0.2,5]

# Initial conditions
theta0 = np.array([0.5,-0.5])
for Q in Qs:
    thetaSolved = integrate.odeint(damped_pendulum, theta0, t, args=(Q,))
    plt.plot(t, thetaSolved[:,0], label=r"$Q$ = %s" % str(Q) )

plt.title("Time Series for Damped Pendulum")
plt.xlabel(r"$time$")
plt.ylabel(r"$\theta(t)$")
plt.legend()
plt.show()
```

Matplotlib

- Plotting data is straightforward (usually) with Matplotlib
  - The plotting package is `matplotlib.pyplot`, so I usually `import matplotlib.pyplot as plt`
- Simplest plot is two lines of code:

```
plt.plot(xdata, ydata)

plt.show()
```

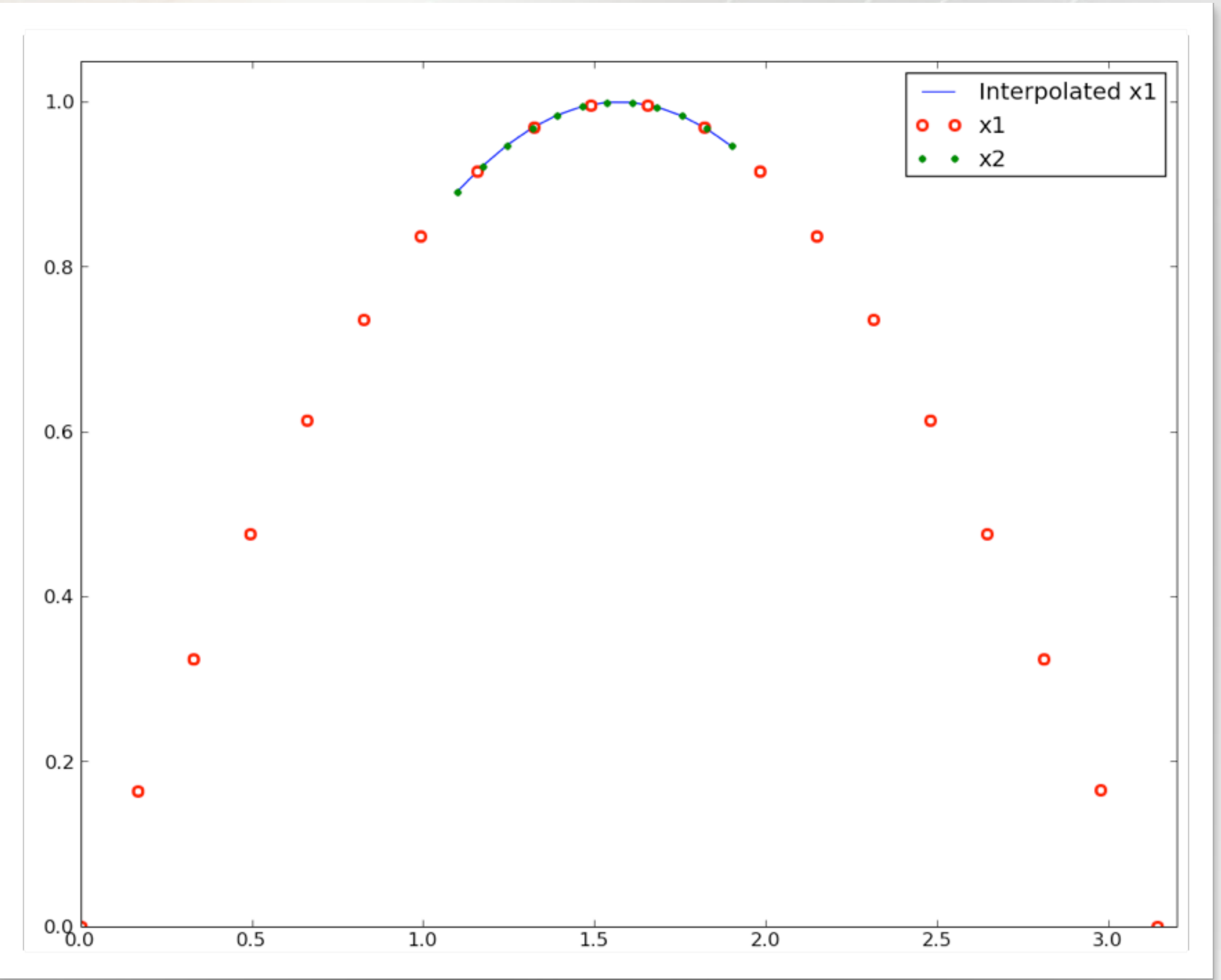
(or `plt.savefig('filename.ext')`)

Example: *InterpExample.py*

- The goal of this example is to compare two data sets, and see if they are drawn from the same function
- We start with 4 arrays of data: t1, x1, t2, x2
- The datasets are sampled differently: we need to interpolate one to compare to the other
  - We use `scipy.interpolate.interp1d(x, y)`
  - Where x and y are related by y=f(x)

Example: *InterpExample.py*

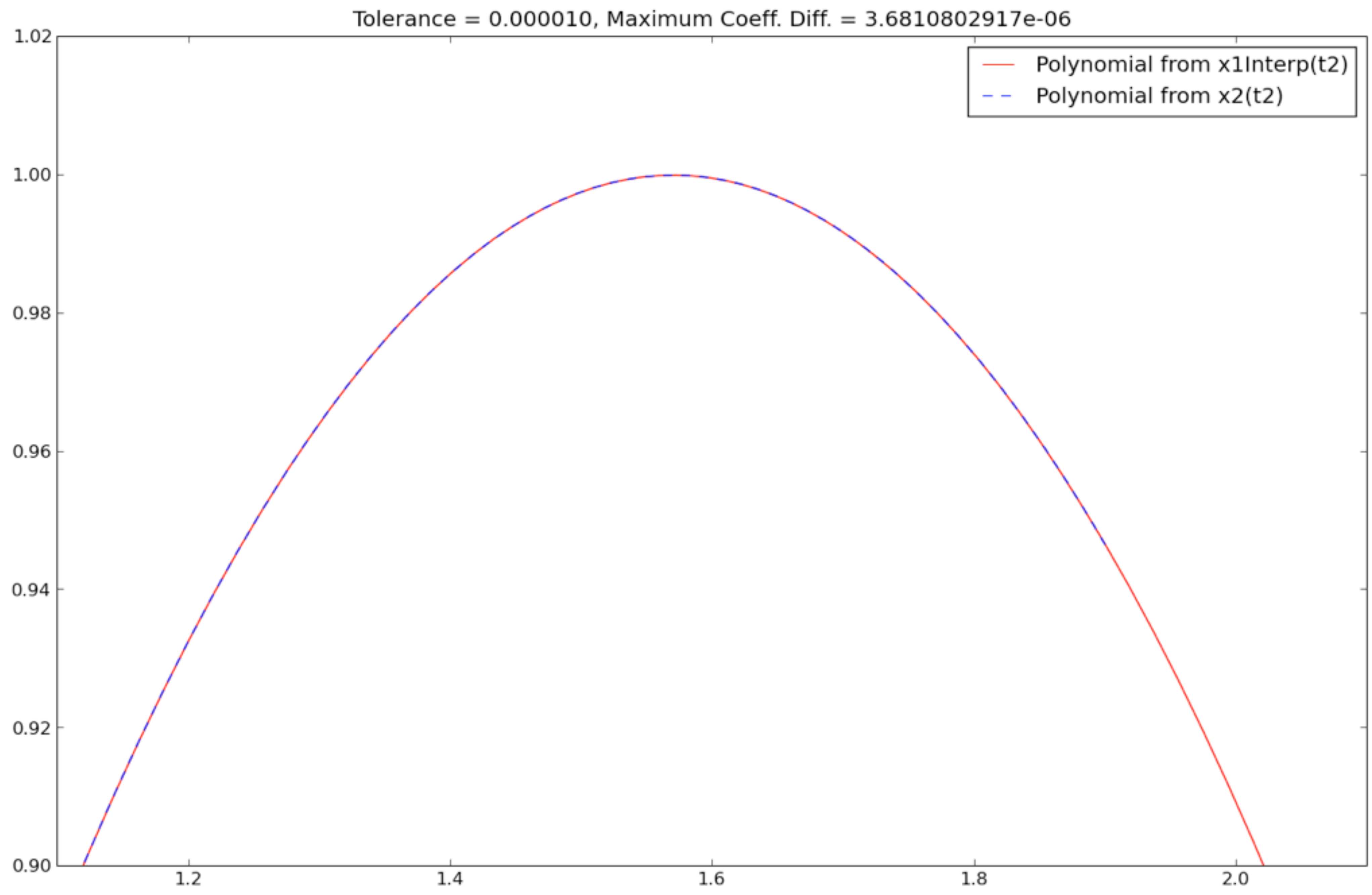
- Once we have the interpolation object `x1Interp` we can do `x1Interp(t2)` to get the values of x1 interpolated onto the grid from x2



Example: *InterpExample.py*

- Now we fit to the interpolated `x1` data over the time-steps from `t2`, and compare this fitted curve to a fit over `t2, x2`
- We'll use the Scipy function `scipy.polyfit(xdata, ydata,polynomial_order)`, which returns polynomial coefficients
- If we arbitrarily set a tolerance at 5 decimal places, we can compare these coefficients to see if the fits agree to within this tolerance

Example: *InterpExample.py*



Matplotlib

- At the bottom of *InterpExample.py*, we are going to add some fancier plotting
- A few useful functions to know are *xlabel*, *ylabel*, *title*, and *hist*
- *xlabel* and *ylabel* accept text as an argument, and label the x or y axes
  - They also accept Latex tags using $:
    - `plt.xlabel(r"$\beta = 5$")`

SciPy and PyFITS

- Remember that using pyfits, you can get the header of an HDU
  - `hduList[0].header`
- Then from the header, extract a specific value, for instance 'BUNIT'
  - `hduList[0].header['BUNIT']`
- Also, from tables you can extract entire columns
  - `hduList[5].data.field('MAG')`

More Advanced Matplotlib

- Subplots
  - If you want a figure with multiple plots, subplot is the way to do it

Number of rows of plots
Number of columns of plots
Index representing the current plot

```
plt.subplot(221)
plt.plot(xdata1, ydata1)

plt.subplot(222)
plt.plot(xdata2, ydata2)

etc...
```

Even More Advanced Matplotlib

- Figures - the object oriented way

```
fig = plt.figure(dpi=100) # can also set figsize=(width,height)

axis1 = fig.add_subplot(221)
axis1.plot(xdata1, ydata1)
axis1.set_title("Dead Parrot")
axis1.set_xlabel("Spanish")
axis1.set_ylabel("Inquisition")

axis2 = fig.add_subplot(222)
axis2.plot(xdata2, ydata2)
```

Exercise: *SpectraExercise.py*

- The goal is to read from an *spPlate* file, which contains multiple spectra from different sources, (galaxies, quasars, sky, etc...) and generate some plots
    - (Details are in *SpectraExercise.py*)