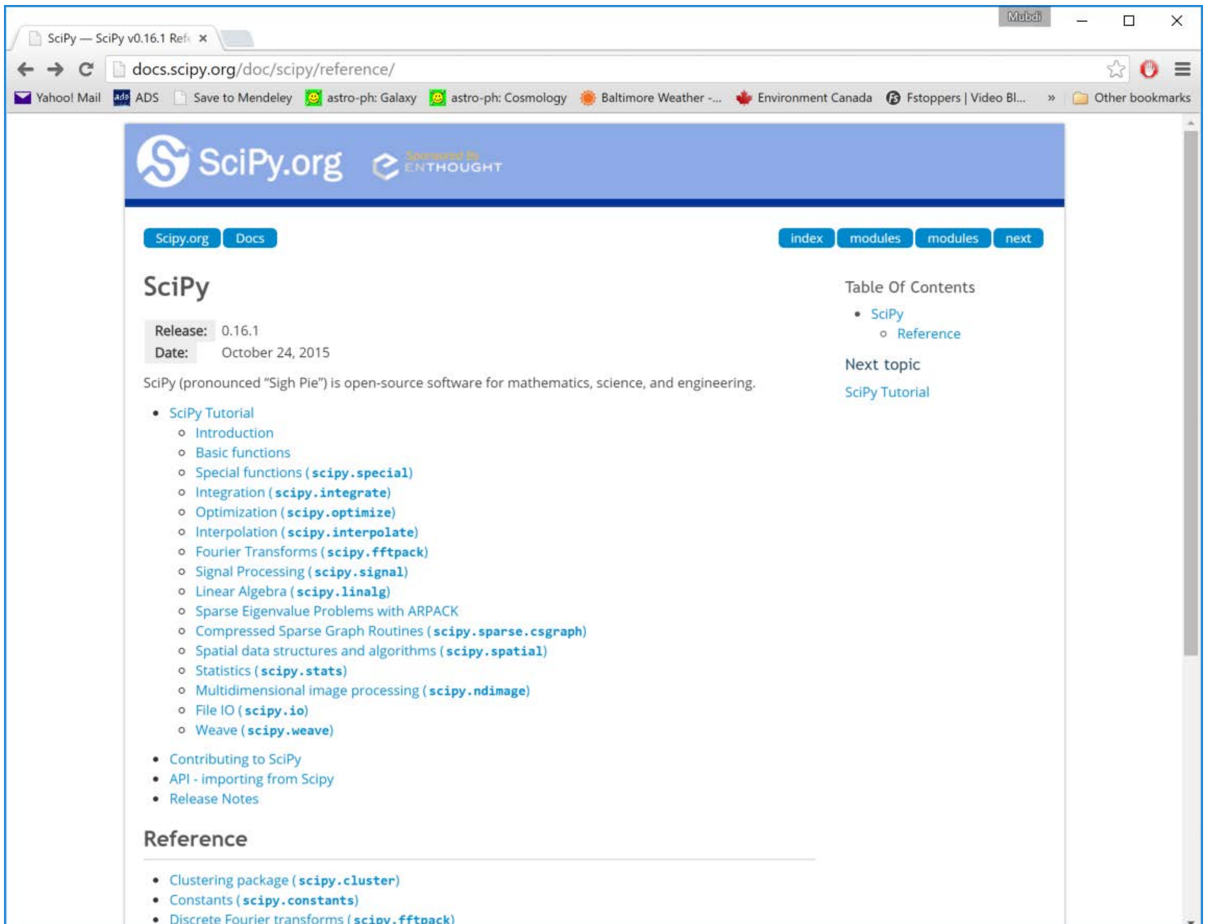
5. ADVANCED DATA TECHNIQUES

JHU Physics & Astronomy
Python Workshop 2015

Lecturer: Mubdi Rahman

SCIPY: FUNCTIONS YOU WANT, THE PACKAGE YOU NEED



The Docs: http://docs.scipy.org/doc/scipy/reference/

L.J. DURSI'S FIRST RULE OF PROGRAMMING

Rule #1: Don't code!

For most common algorithms or problems that exist, there are functions and modules that have been optimized and tested by large groups of people who know what they're doing. Use these rather than programming your own.

Scipy has a lot of these functions and algorithms ready for your use. In this lesson, we'll go through a few of these useful functions.

ORGANIZATION OF PACKAGES

Scipy

- Optimize/Fitting (`scipy.optimize`)
- Integration (`scipy.integrate`)
- Image Processing (`scipy.ndimage`)
- Linear Algebra (`scipy.linalg`)
- Statistics (`scipy.stats`)
- Much More…

INTERPOLATION

Importing the Functions:

```
from scipy import interpolate
```

Basic one-dimensional interpolation:

```
funct1 = interpolate.interp1d(
xvals, yvals, kind='linear', bounds_error=False,
fill_value=np.nan)
```

Kind options: 'linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic'

INTERPOLATION

Importing the Functions:

```
from scipy import interpolate
```

Basic one-dimensional interpolation:

```
funct1 = interpolate.interp1d(
ear’, bounds_error=False,
```

ero’, ‘slinear’, ‘quadratic’, ‘cubic’

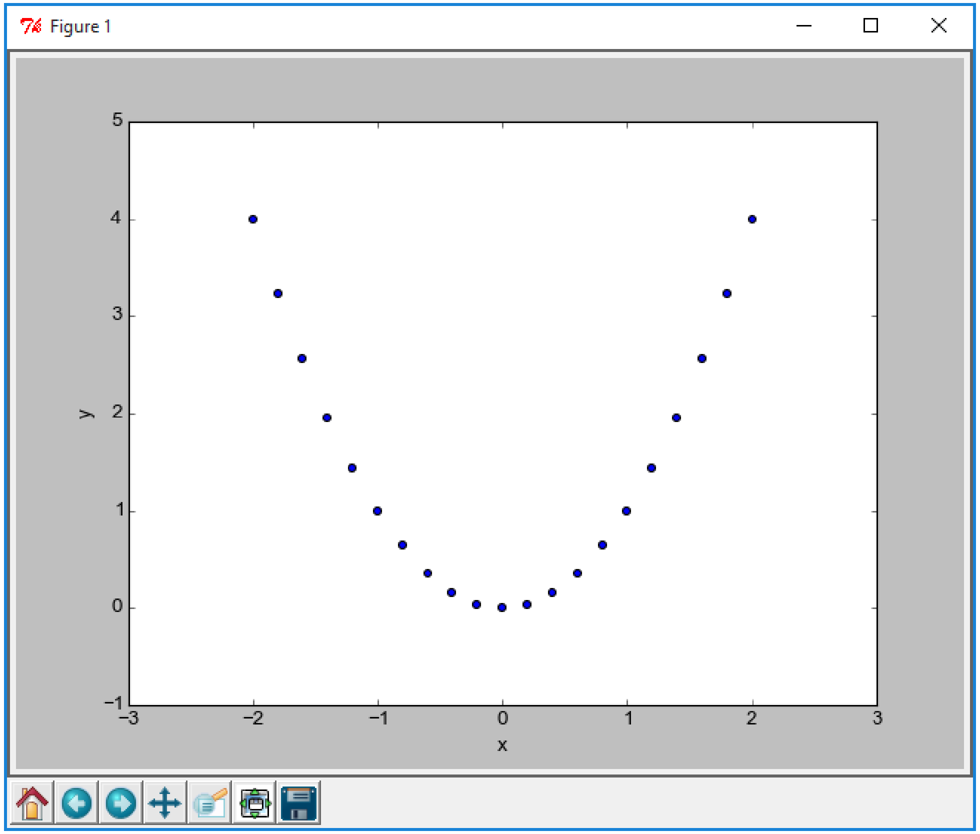
PRO TIP:

By default, the interpolation will fail if you go beyond the minimum and maximum points. The bounds_error keywork helps deal with this.

INTERPOLATION

Given a simple function:

```
funct1 =
interpolate.interp1d(
  xvals, yvals,
  kind='nearest'
)
```

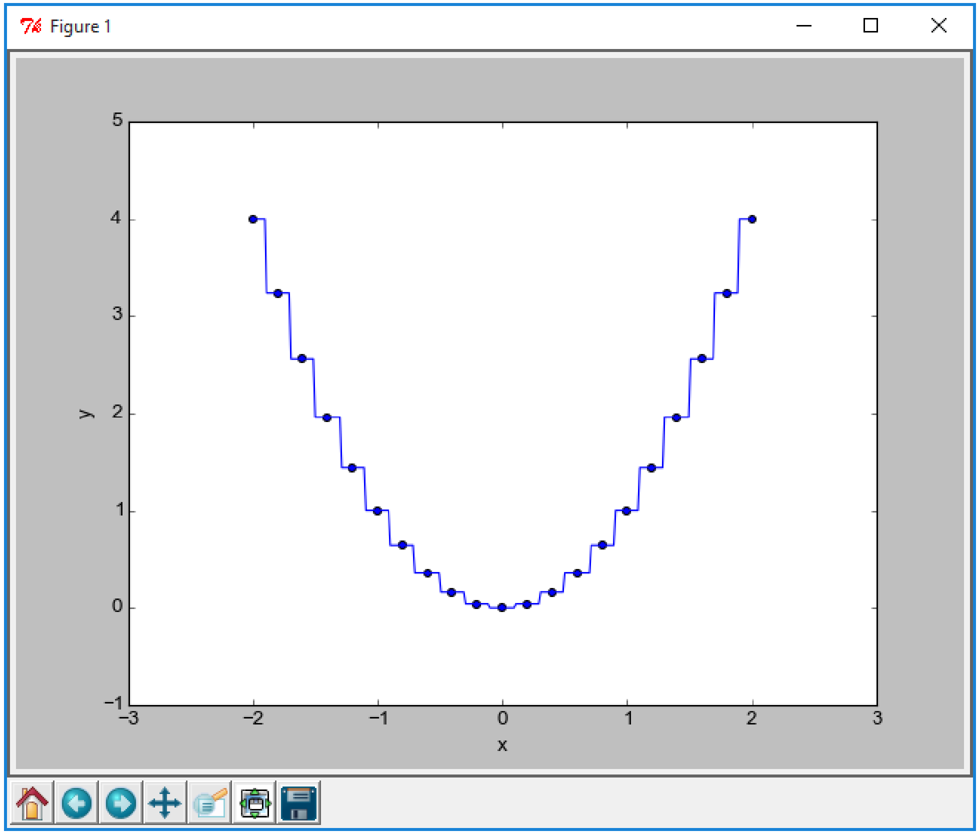


INTERPOLATION

Given a simple function:

```
funct1 =
interpolate.interp1d(
  xvals, yvals,
  kind='nearest'
)
```

Nearest Neighbour interpolation

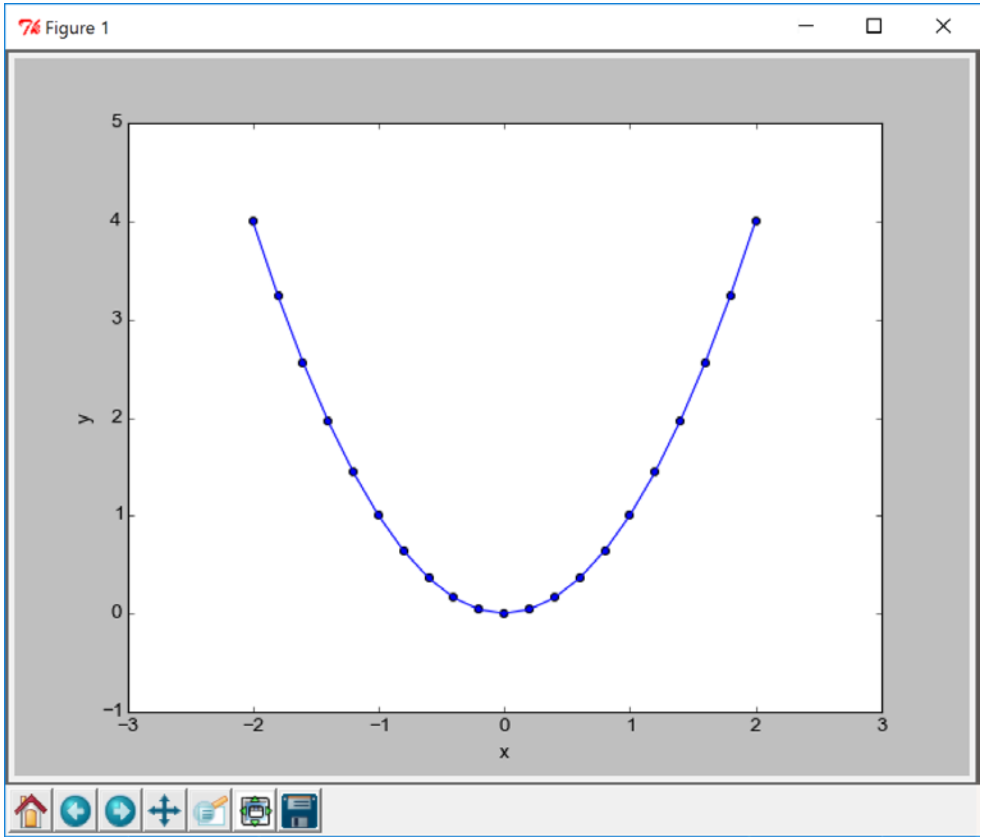


INTERPOLATION

Given a simple function:

```
funct1 =
interpolate.interp1d(
  xvals, yvals,
  kind='linear'
)
```

Linear interpolation

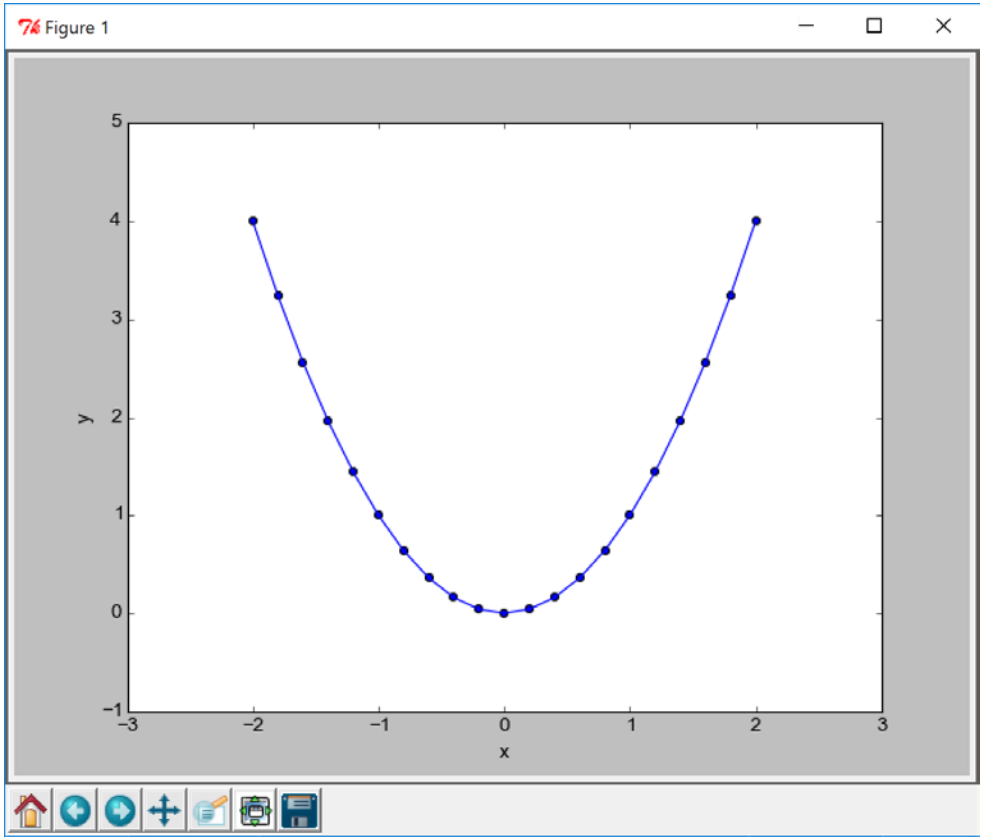


INTERPOLATION

Given a simple function:

```
funct1 =
interpolate.interp1d(
  xvals, yvals,
  kind='slinear'
)
```

Linear Spline
(first order)



NDIMAGE

Library of functions useful for dealing with N-dimensional images. In particular, we'll be using the filters for smoothing. Also includes functions to interpolate and manipulate images. Importing the library:

```
from scipy import ndimage
```

Includes basic filters:

```
ndimage.gaussian_filter(…)
ndimage.median_filter(…)
```

Or using generic convolution:

```
ndimage.convolve(…)
```

NDIMAGE

Library of functions useful for dealing with N-dimensional images. In particular, we'll be using the filters for smoothing. Also includes functions to interpolate and manipulate images. Importing the library:

```
from scipy import ndimage
```

Includes basic filters:

```
ndimage.gaussian_filter(…)
ndimage.median_filter(…)
```

Or using generic convolution:

```
ndimage.convolve(…)
```

PRO TIP:

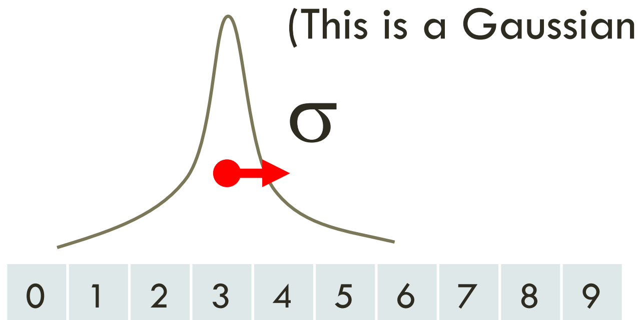
These functions work just as well on 2-D or 3-D arrays as they do on 1-D arrays.

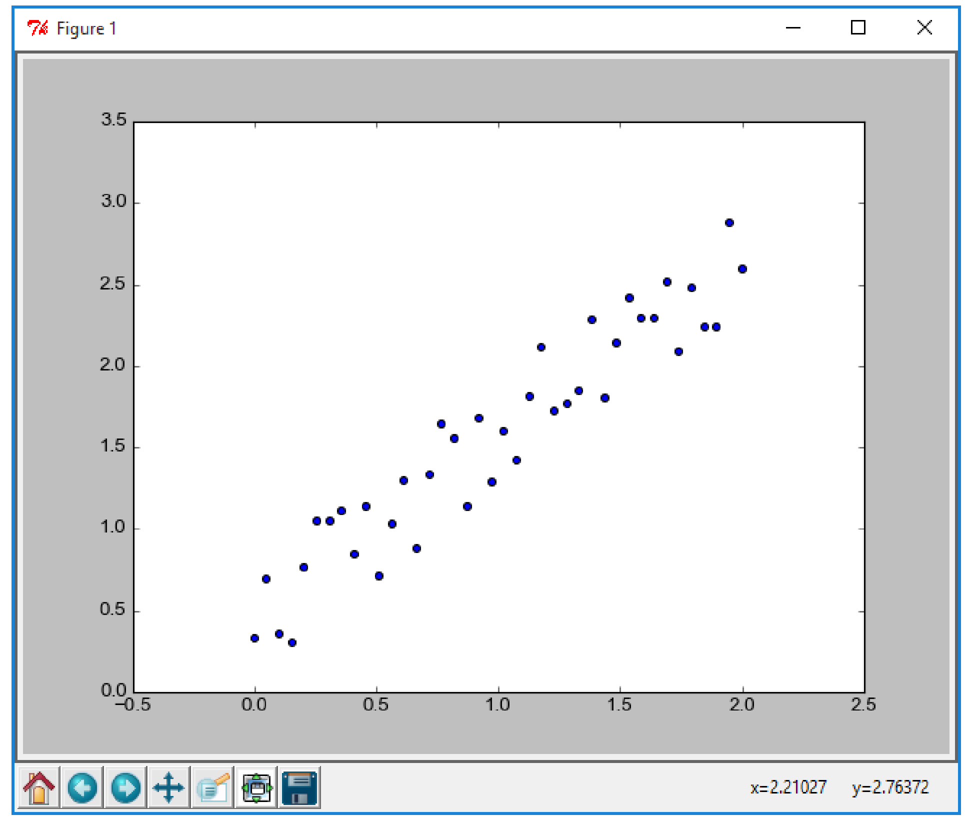
SMOOTHING/FILTERING

Given an array:

```
ndimage.gaussian_filter(
arr1, sigma=1.0)
```

(This is a Gaussian.)

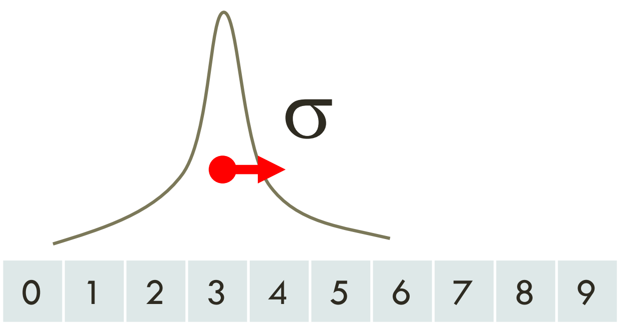


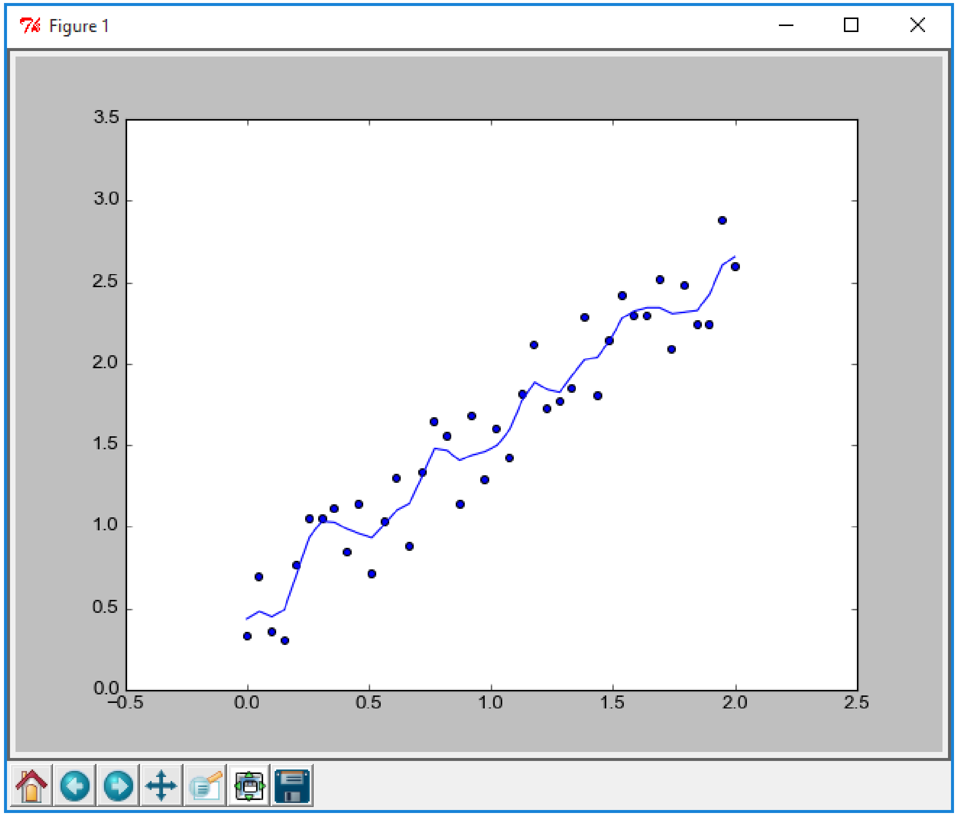


SMOOTHING/FILTERING

Given an array:

```
ndimage.gaussian_filter(
arr1, sigma=1.0)
```

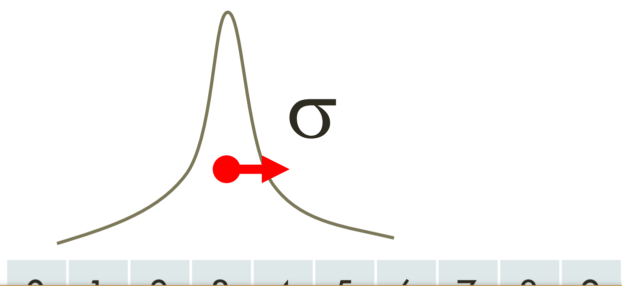


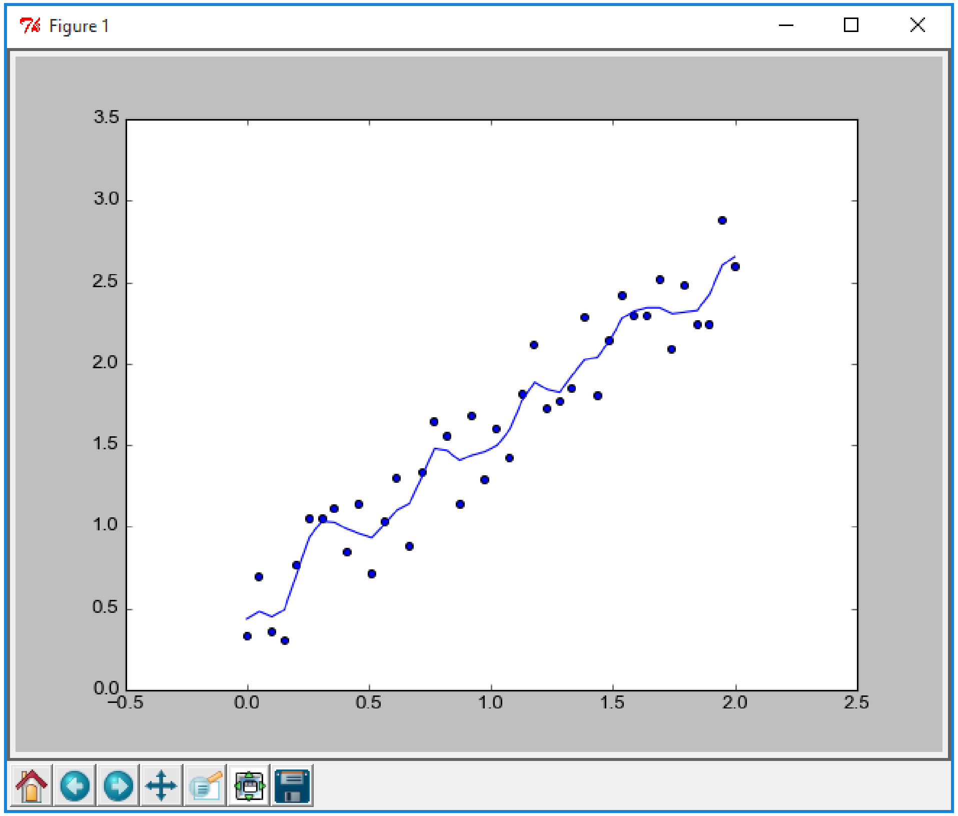


SMOOTHING/FILTERING

Given an array:

```
ndimage.gaussian_filter(
arr1, sigma=1.0)
```





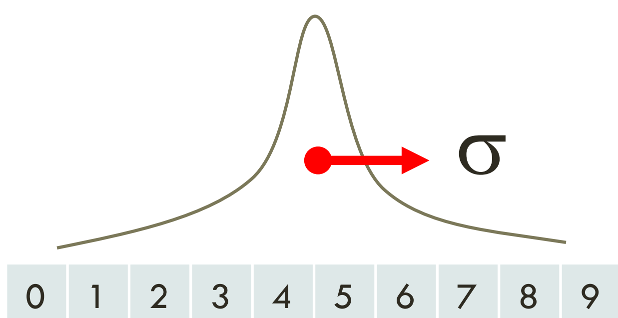
PRO TIP:

Values in the filter are in pixel units.

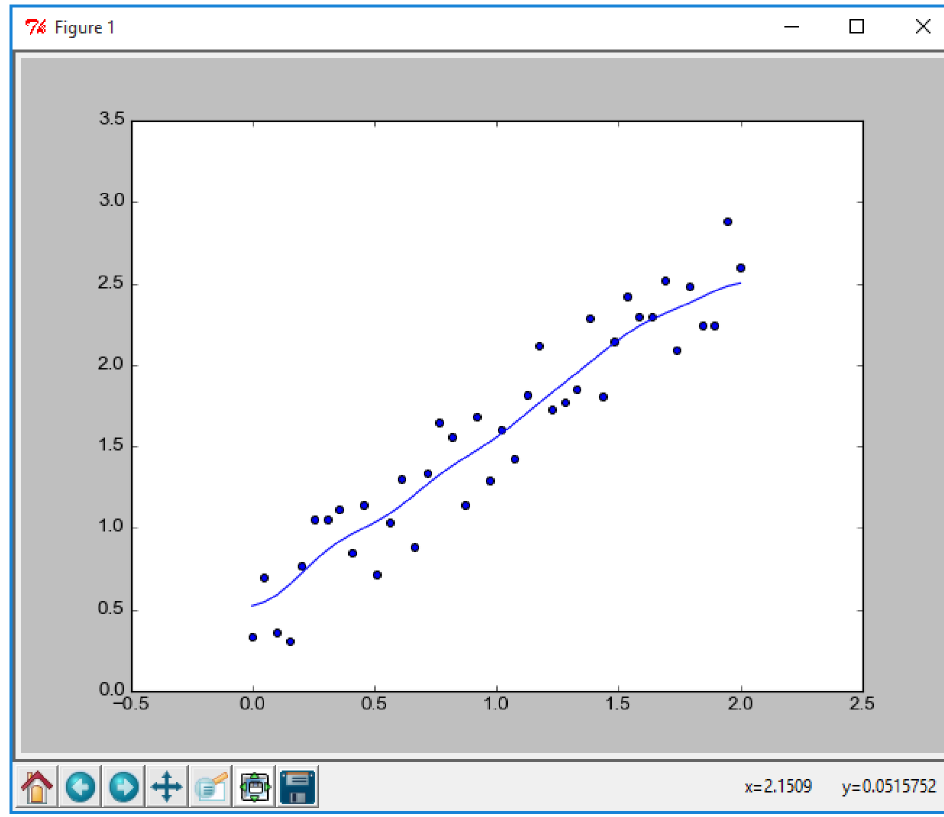
SMOOTHING/FILTERING

Given an array:

```
ndimage.gaussian_filter(
arr1, sigma=3.0)
```



(This is also a Gaussian.
I can draw in PowerPoint)

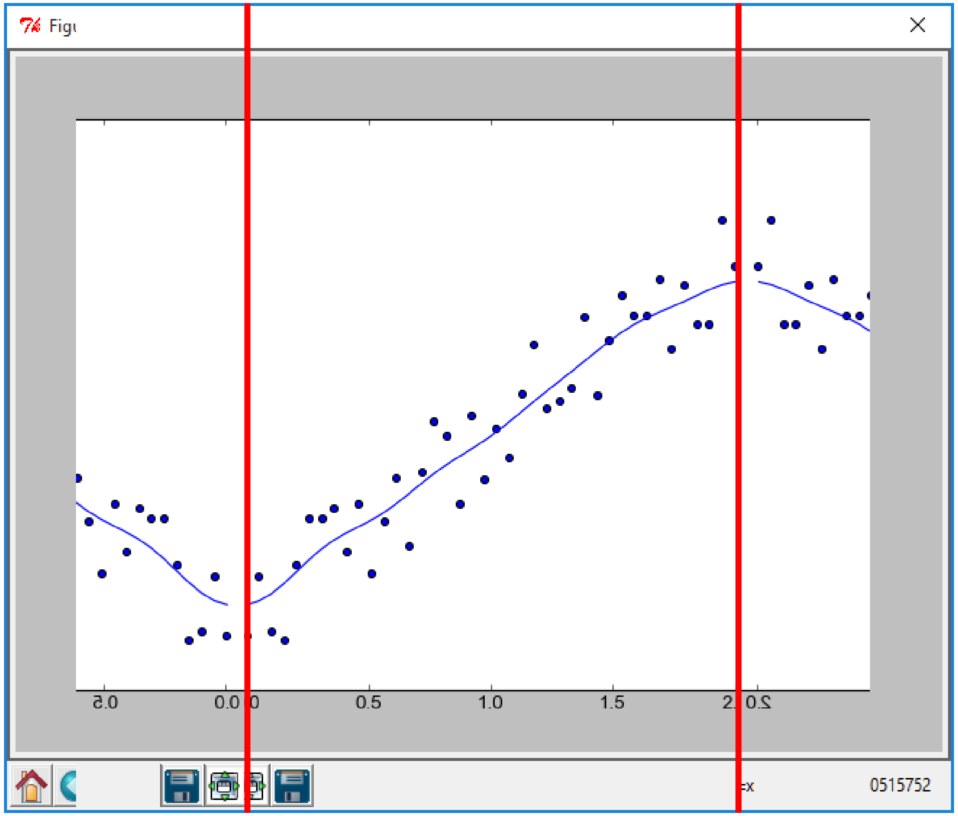


SMOOTHING/FILTERING

Given an array:

```
ndimage.gaussian_filter(
arr1, sigma=3.0,
mode='reflect')
```

The "mode" of the filtering indicates what happens at the edges of the dataset. "Reflect" treats the edges of the domain as the inverse of the dataset.

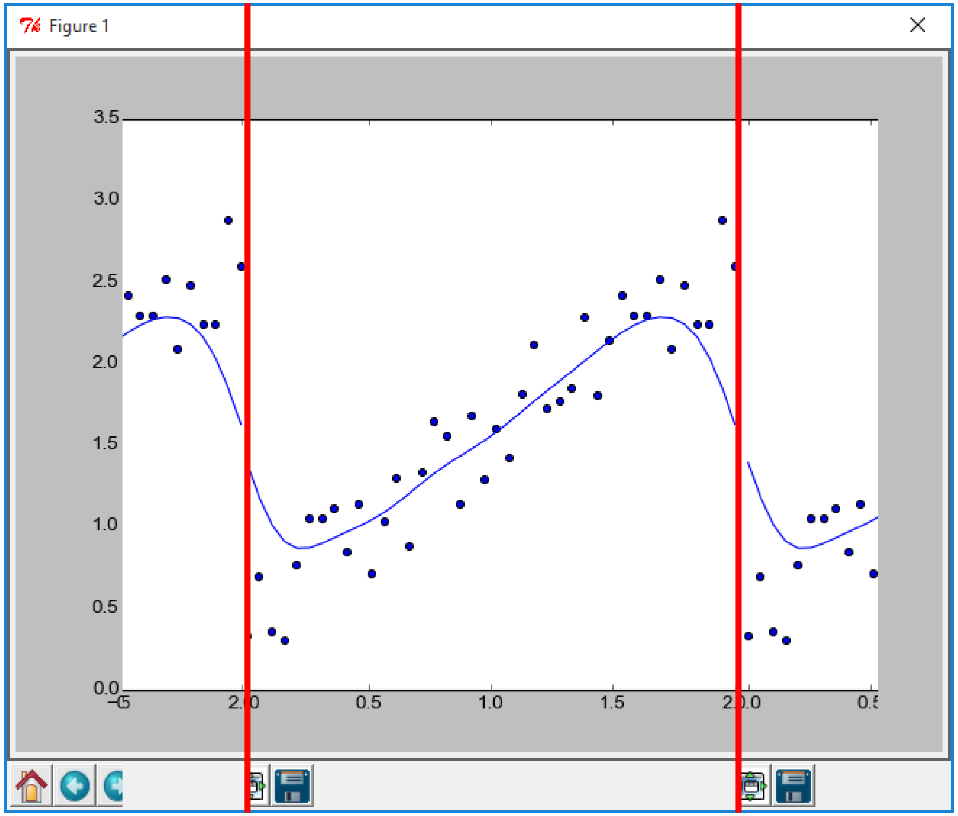


SMOOTHING/FILTERING

Given an array:

```
ndimage.gaussian_filter(
arr1, sigma=3.0,
mode='wrap')
```

"Wrap" treats the data like a tiled patchwork, with the data repeating itself on either end.



INTEGRATION

Importing the Functions:

```
from scipy import integrate
```

Functions that integrate fixed samples (i.e., numpy arrays):

```
integrate.cumtrapz(…) # Composite trapezoidal
integrate.simps(…) # Simpson’s Rule
integrate.romb(…) # Romberg integration
```

Functions that integrate functions:

```
integrate.quad(…) # General Purpose Integration
integrate.nquad(…) # Multiple Variable
integrate.quadrature(…) # Fixed Tolerance Integration
```

INTEGRATION

Composite Trapezoidal (Cumulative)

```
intarr = integrate.cumtrapz(yarr, x=xarr)
```

This function returns an array (size one less than original array). To get the final integrated value of the entire array:

```
total = intarr[-1]
```

INTEGRATION

Composite Trapezoidal (Cumulative)

```
intarr = integrate.cumtrapz(yarr, x=xarr)
```

This function returns an array (size one less than original array). To get the final integrated value of the entire array:

```
total = intarr[-1]
```

PRO TIP:

This also works when an array isn't evenly spaced. Just make sure to pass an array of x values.

INTEGRATION

Integrating functions (using quad):

```
# Creating function to integrate:
funct1 = lambda x: x**2

# Integrating the function over range [min, max]
total, err = integrate.quad(funct1, min, max)
```

Can integrate from negative infinity to positive infinity through:

```
total, err = integrate.quad(funct1, -np.inf, np.inf)
```

STATISTICS

Importing the Functions:

```
from scipy import stats
```

Provides access to a variety of useful functions:

```
stats.mode(arr1) # Modal Value

# Statistical measures
stats.skew(arr1), stats.kurtosis(arr1), …

# Trimmed Mean, Standard Deviation
stats.tmean(arr1, limits=[min, max]), stats.tstd(…)

# Percentile -> Score
stats.scoreatpercentile(arr1, percentile)
```

OPTIMIZE/FITTING

Importing the Functions:

```
from scipy import optimize
```

Unlike the other operations, curve fitting is **quite complex.** Consequently, there are a number of different functions and algorithms available to handle any number of situations. **Be sure that the fitting method you're using is doing what you think it is.**

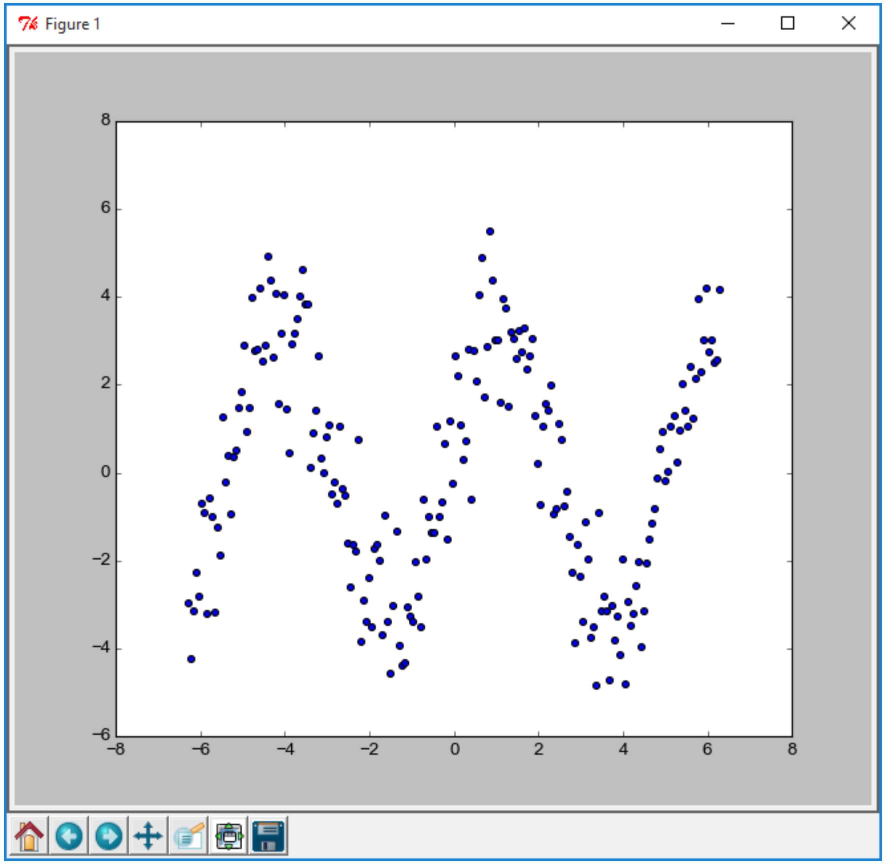
The basic functions you should know about:

```
optimize.curve_fit(…) # Fit a defined curve to data

# Minimized the sum of square of an equation
optimize.leastsq(…)

optimize.minimize(…) # Minimize a function
```

OPTIMIZE/FITTING

Basic curve fitting, using curve_fit:

```
# Function to fit to:
funct1 = lambda x,a,b,c:
  a*np.sin(b*x + c)
```



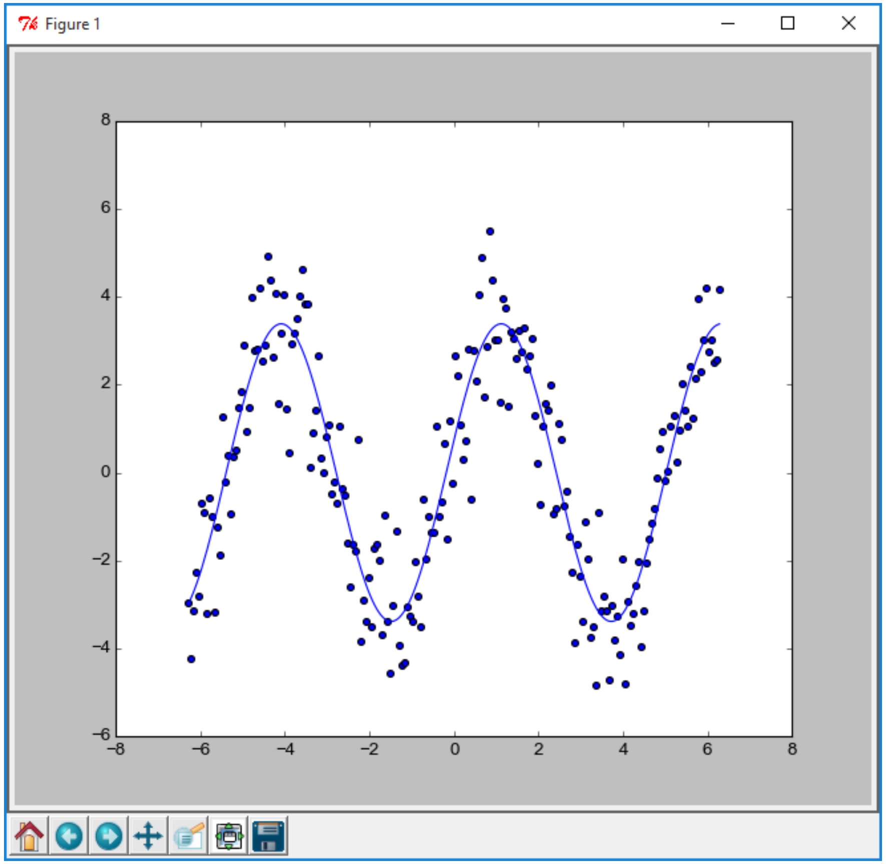
OPTIMIZE/FITTING

Basic curve fitting, using curve_fit:

```
# Function to fit to:
funct1 = lambda x,a,b,c:
  a*np.sin(b*x + c)
```

Assuming data in arrays named data_x and data_y:

```
# Fitting the data:
param, covar =
optimize.curve_fit(
    funct1, data_x, data_y
)
```



OPTIMIZE/FITTING

We can use the “minimize” function to be more flexible for fitting, minimizing the “least-square” function:

$$\sum (data - model)^2$$

Or if there are errors that you want to weight the fitting by:

$$\sum \left(\frac{data - model}{error}\right)^2$$

Both of these functions are always positive (for real numbers), and minimizing them ensures that you have adopted the best-fit parameters

OPTIMIZE/FITTING

Taking the equation from earlier:

$$y = a\sin(bx + c)$$

Which converts into:

```
funct1 = lambda x,a,b,c: a*np.sin(b*x + c)
```

OPTIMIZE/FITTING

Taking the equation from earlier:

$$y = a\sin(bx + c)$$

Which converts into:

```
funct1 = lambda x,a,b,c: a*np.sin(b*x + c)
```

We can create a (single variable) function to minimize:

```
funct2 = lambda par:
  np.sum((data_y - funct1(data_x, *par))**2)
```

Or with errors:

```
funct2 = lambda par:
  np.sum(((data_y - funct1(data_x, *par))/err)**2)
```

OPTIMIZE/FITTING

Taking the equation from earlier:

$$y = a \sin(bx + c)$$

Which converts into:

```
funct1 = lambda x,a,b,c: a*np.sin(b*x + c)
```

We can create a (single variable) function to minimize:

```
funct2 = lambda par:
t1(data_x, *par))**2)
```

```
ct1(data_x, *par))/err)**2)
```

PRO TIP:

If you have a tuple, list, or array that contains all the parameters you want to pass to a function in order, you can pass it by using the asterisk (*).

OPTIMIZE/FITTING

Taking the equation from earlier:

$$y = a\sin(bx + c)$$

Which converts into:

```
funct1 = lambda x,a,b,c: a*np.sin(b*x + c)
```

We can create a (single variable) function to minimize:

```
funct2 = lambda par:
                    t1(data_x, *par))**2)
```

```
                    ct1(data_x, *par))/err)**2)
```

PRO TIP 2:

Notice that the minimizing functions have only one argument, which is a 1-D vector of the required parameters.

OPTIMIZE/FITTING

Piecing this into the minimize_scalar function:

```
result = optimize.minimize(funct2, x0=initialguess)
```

The initial guess is an array with the same size as the parameters you want to fit.

This function abstracts a variety of different algorithms with different possible parameters. For instance you can use the following to define bounds for the fitting:

```
result = optimize.minimize(
  funct2, x0=initialguess, method='L-BFGS-B',
  bounds=((0, 5), (0, 2), (0,3))
)
```

OPTIMIZE/FITTING

Once you have the result, you have lots of information provided to you as a dictionary:

```
result[‘x’] # The final parameters of the fit
result[‘success’] # Whether the fit was successful
result[‘nit’] # Number of Iterations Performed
result[‘jac’] # The jacobian of the fit
```

OPTIMIZE/FITTING

Once you have the result, you have lots of information provided to you as a dictionary:

```
result['x'] # The final parameters of the fit
result['success'] # Whether the fit was successful
result['nit'] # Number of It
result['jac'] # The jacobia
```

PRO TIP:

When fitting, especially this way, check to ensure convergence.

EXERCISE TIME!

Rubber ducky, you're the one. You make bath time so much fun!