# *Python for Scientific Data Analysis*

# SciPy

SciPy stands for Scientific Python. SciPy is a scientific computation library that you will see referenced profusely in physics and astronomy coding applications. SciPy SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

Originally, it was written using its own data type (a "Numeric"). But then everyone realized that this wasn't a good idea. So now the basic data structure used by SciPy is a multidimensional array provided by the NumPy module (i.e. it uses NumPy underneath).

While a lot of the linear algebra operations in SciPy are also found in NumPy, SciPy is especially good for optimization, statistics, and signal processing/filtering of various flavors.

## Section 1: Optimization in SciPy

We have already discussed some of its linear algebra capabilities in the previous lectures, so we are going to switch gears slightly, first discussing its optimization capabilities.

The *optimization* subpackage within Scipy can be accessed by `from scipy import optimize`. Nearly always you want to do array operations when setting up an optimization problem, in which case, you need to load NumPy as well. So the typically preliminary steps for a SciPy optimization problem are to do the following at the beginning of your script:

```
import numpy as np
from scipy import optimize
```

### 1. Root Finding

#### *Motivation*

The root or zero of a function, $f(x)$, is an $x_r$ such that the function evaluated at $x_r$ equals 0 ( $f(x_r)$ =0).

Finding the roots of functions is important in many engineering applications such as signal processing and optimization. For simple functions such as $f(x) = ax^2 + bx + c$, you can analytically solve for roots using the *quadratic formula*:

$x_r$ = -b $\pm$ $\frac{\sqrt{b^2-4ac}}{2a}$ .

This gives the two roots of $f$, $x_r$, exactly.

But what about if you have a cubic function? Or quartic? Or just a very complicated-looking one? In that case, you need to appeal to numerical methods to find roots.

We will start of discussing root finding with `scipy.optimize`'s `fsolve` function. `fsolve` is pretty complicated and we are going to treat it as a black box for now.

Then we will discuss an important topic with any sort of optimization procedure -- tolerancing -- before going back to describing a few the wide array of optimization approaches (e.g. gradient descent; Newton-Raphson) and how these strategies are implemented in algorithms SciPy actually uses.

### *Basic Root Finding with SciPy.Optimize's `fsolve` and `root`*

Our first stop in the process if doing this is the aptly named `fsolve` function within `scipy.optimize` . As the documentation states "*fsolve is a wrapper around MINPACK's hybrd and hybrj algorithms.*" Here, MINPACK is a FORTRAN subroutine library for solving minimization problems which you may have heard about. So as with NumPy, SciPy does a lot of things that are basically friendly wrappers of algorithms in faster, simpler languages.

The actual algorithm used with `fsolve` is a version of *Powell's Hybrid Method* which is a combination of the Gauss-Newton algorithm (which is an extension of the Newton-Raphson method described below) and gradient descent (ditto). If you are really interested in the details of this method you can read about it here: https://en.wikipedia.org/wiki/Powell's_dog_leg_method . If you are *really really* interested (I'm not) you can also read about the specific implementation within MINPACK here: https://www.math.utah.edu/software/minpack/minpack/hybrj.html . For everyone else (everyone?), we will just move on and describe what it does with a few examples.

Let's take one example of root finding with `fsolve` . Consider the function:

$f(x) = 5 * sin(x) - x^3 + 2$

Now, let's solve it with `fsolve`. Here, the key commmand is `fsolve([function name],initial guess)`. That is, you have to provide `fsolve` with a starting guess. If there are multiple roots, this is akin to "find the nearest root to $x_o$".

Anyway, let's proceed to set up our optimization problem...

```
import numpy as np
from scipy import optimize

 #define our function as a lambda function

func=lambda x: 5*np.sin(x)-x**3.+2
```

Here, I've defined a lambda function *func* which is a function of *x*. We then will plug in this single variable into `fsolve` to gain a solution given a starting guess. Let's try a couple of initial guesses. First, let's try at zero. Now, we can easily figure out just from looking at the function, that 0 is not *actually* a root (the first two entries go to zero, but the 2 remains). But perhaps there is a solution close to it.

```
 #see how simple this is to write? I told you that lambda functions would
 be useful somehow!

 result=optimize.fsolve(func,0)
 print(result)
 #array([-0.42878611])
```

So, given an initial guess of 0, we find a root of about -0.429. We can then verify that this is a root as follows:

```
 func(result)
 #array([-1.33226763e-14])
```

So we get a number very close to 0. Looks like `fsolve` found something that is quite close to a root of our ugly $f(x)$ function.

Now, let's try a different initial guess ... how about 10 being a root? You can probably guess it isn't (the $x^3$ term is going to tank this idea) but can SciPy find a root with this initial guess? Yup!

```
result=optimize.fsolve(func,10)
#array([1.88968729])

func(result)
#array([0.])
```

In fact, by eye this looks like an *exact* solution.

What about if we give it a negative value?

```
result=optimize.fsolve(func,-10)
#array([-1.43481477])

func(result)
#array([2.03836947e-13])
```

So manually guessing solutions and getting the roots nearest to our guess is tedious. And we would be tempted to vectorize this problem. E.g.

```
import numpy as np
from scipy import optimize

 #define our function as a lambda function

func=lambda x: 5*np.sin(x)-x**3.+2
evalnumbers=np.arange(-100,100,0.5)
result=optimize.fsolve(func,evalnumbers)
roots=np.unique(result.round(decimals=6))
roots
#array([-1.434815, -0.428786,  1.889687])
func(roots)
#array([1.26732159e-06, 4.38162979e-07, 3.60173658e-06])
```

But what this is literally telling `optimize` is that your initial guess is the array np.arange(-100,100,0.5), whereas the function is a function of just one variable. Watch what happens when you decrease the spacing size:

```
import numpy as np
from scipy import optimize
func=lambda x: 5*np.sin(x)-x**3.+2
evalnumbers=np.arange(-100,100,0.5)
result=optimize.fsolve(func,evalnumbers)
evalnumbers=np.arange(-100,100,0.2) #2.5x finer grid
#RuntimeWarning: The iteration is not making good progress, as measured b
y the
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
roots=np.unique(result.round(decimals=6))
roots
#giant array
```

First, Python tells you that the solution did not converge. And the answer you get is very close to your initial guess in each case. Not helpful.

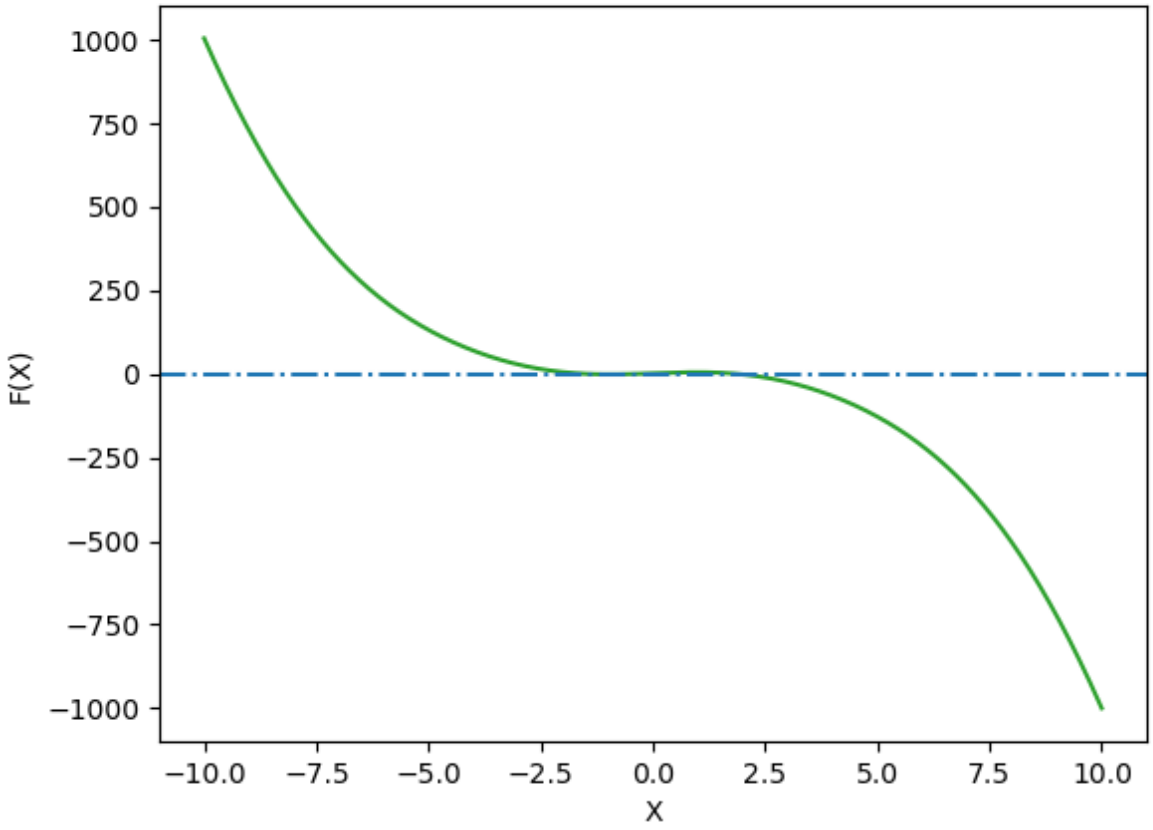
Now do the same in a for-loop:

```
import numpy as np
from scipy import optimize
func=lambda x: 5*np.sin(x)-x**3.+2
evalnumbers=np.arange(-100,100,0.5)

result=np.zeros(len(evalnumbers))
for ind,guess in enumerate(evalnumbers):
 result[ind]=optimize.fsolve(func,guess)

roots=np.unique(results.round(decimals=6))

roots
#array([-1.434815, -0.428786,  1.889687])
```

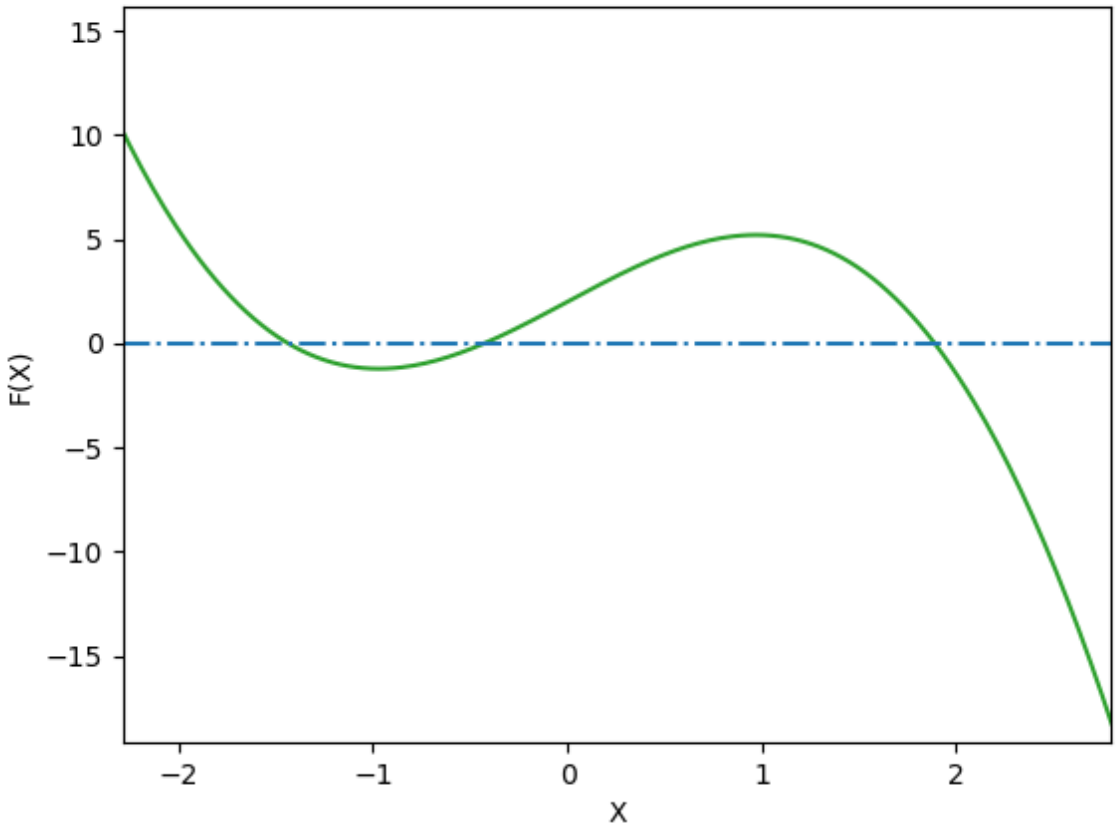
See, we recovered the original, suspected answer based off our our manual picking away at the problem.

Now, we can visualize the solutions using a plotting package like Matplotlib (don't worry about the syntax if you don't know it already: we will learn it later):

```
import matplotlib.pyplot as plt

rangeofnumbers=np.linspace(-10,10,1000) #an array of x values where we ev
aluate the function "func"
plt.plot(rangeofnumbers,func(rangeofnumbers),c='tab:green')
plt.xlabel('X')
plt.ylabel('F(X)')
plt.show()
```




Here, I've added a horizontal blue dash-dotted line. It is clear that f(x) crosses 0 somewhere around -1.5 and 2. Zooming in shows that it crosses near -0.4 as well. There are no other zeros (er, 'roots') of this function.




### *Basic Root Finding with SciPy.Optimize's `root`*

Okay, `fsolve` is one root-finding example. Another is aptly named `root`. We will briefly discuss the key features of `root`: the full documentation is here https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html#scipy.optimize.root)

A major difference with `root` vs `fsolve` is that `root` offers you *multiple* algorithms for root finding. The default method is the modified Powell method which -- like `fsolve` -- uses MINPACK routines under the hood. Some of the others are commonly used -- in fact you may have heard of them before.

Describing what they actually do is instructive at this point. But to avoid them becoming yet another black box, we need to describe basic concepts with root finding algorithms first. Then we can discuss the algorithms actually used and discussing specific examples. Note that these algorithms are often used in minimization problems (under `scipy.optimize.minimize`) which we will get to after root finding.

#### *Basic Concept: Gradient Descent*

Note that there is no "gradient descent" canned algorithm in SciPy but the approach is used by other algorithms and it is conceptually simple. With gradient descent and starting at point $x_o$, you take repeated steps in the opposite direction of the gradient (i.e. the negative gradient $\nabla$F) of the function at $x_o$, because this is the direction of steepest descent. For a small enough steps size $\gamma$ the change in function value can be approximated as linear: $x_{n+1} = x_n - \gamma_n \nabla \mathrm{F}(x_n)$.

This seems straightforward, though in practice finding the right step size $\gamma$ is challenging (too small: algorithm is slow; too large: algorithm overshoots solution).

***Basic Concept: Newton-Raphson Method***

The Newton-Raphson method (or "Newton's Method") also rests on the simple idea of iterative approximation. Here, if $f$ is differentiable, then we start with an initial guess of the root at $x_o$: an improved root should come from: an estimate of $x_1 = x_o - f(x_o)/f'(x_o)$. Then the next estimate will be $x_2 = x_1 - f(x_1)/f'(x_1)$ and so on. Note that if the *second* derivative is calculated, this becomes *Halley's method*.

A variant of the Newton-Raphson method is the *secant method*, where we start with *two* initial values: $x_n = x_{n-1} - f(x_{n-1})/Q(x_{n-1}, x_{n-2})$, where $Q(x_{n-1}, x_{n-2}) = \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}$

Note that if $Q$ is close to the instantaneous derivative of $f$ ($f'(x)$) then the secant method differs very little from the standard Newton-Raphson method. While the secant method usually requires more iterations to converge than Newton-Raphson, each iteration is usually faster (since calculating a derivative is usually more computationally expensive than calculating $Q$).

In any case, the relevant SciPy function is `scipy.optimize.newton`. Depending on the input parameters, what it actually executes is either classic Newton-Raphson, secant, or Halley's method.

***Levenberg-Marquardt Algorithm***

Approaches like Newton-Raphson and gradient descent motivate algorithms actually used by SciPy.

An example is the "Levenberg-Marquardt" algorithm, a classic method for solving non-linear least square problems (root-finding is an example of this). LM is generally fast and a good choice if you have a good idea of the solution but need precise numbers.

Qualitatively, the way that LM works is fairly straightforward and instructive. Briefly, if you have a function evaluated at points $i$ such that $y$ = f($x_i$, $\beta$), then LM finds the parameters of $\beta$ such that the sum of the squares of the deviations from $y$ are minimized: $\beta$ = argmin$_\beta$ $\sum_{i=1}^{m}$ [$y_i$ - f($x_i$, $\beta$)]

$^2$. LM iterates to this solution.

The manner in which LM iterates to a solution leverages on the Jacobian matrix $\mathbf{J}$ of the function (think "partial derivative with respect to everything") and non-negative "damping" factor:

$$J^T J + \lambda I)\delta = J^T [y - f(\beta)].$$

LM allows you to adjust $\lambda$ at each iteration. If the sum of the squared deviations from $y$ is rapid, then $\lambda$ is reduced in size, which brings the algorithm closer to the Gauss-Newton method. If an iteration results in a slow reduction in the sum of the squared deviations, then $\lambda$ can be increased, giving a step closer to the gradient descent direction.

To do this, the parameter vector $\beta$ is replaced by a new, slightly modified entry $\beta' = \beta + \delta$ at each iteration. The original function is then approximated as f($x_i$, $\beta$+$\delta$) ≈ f($x_i$, $\beta$) + $\mathbf{J}_i \delta$, where $\mathbf{J}_i$ is the gradient of $f$ with respect to $\beta$.

The LM algorithm called be utilized from

`result=scipy.optimize.root([func],method='lm')`

An example:

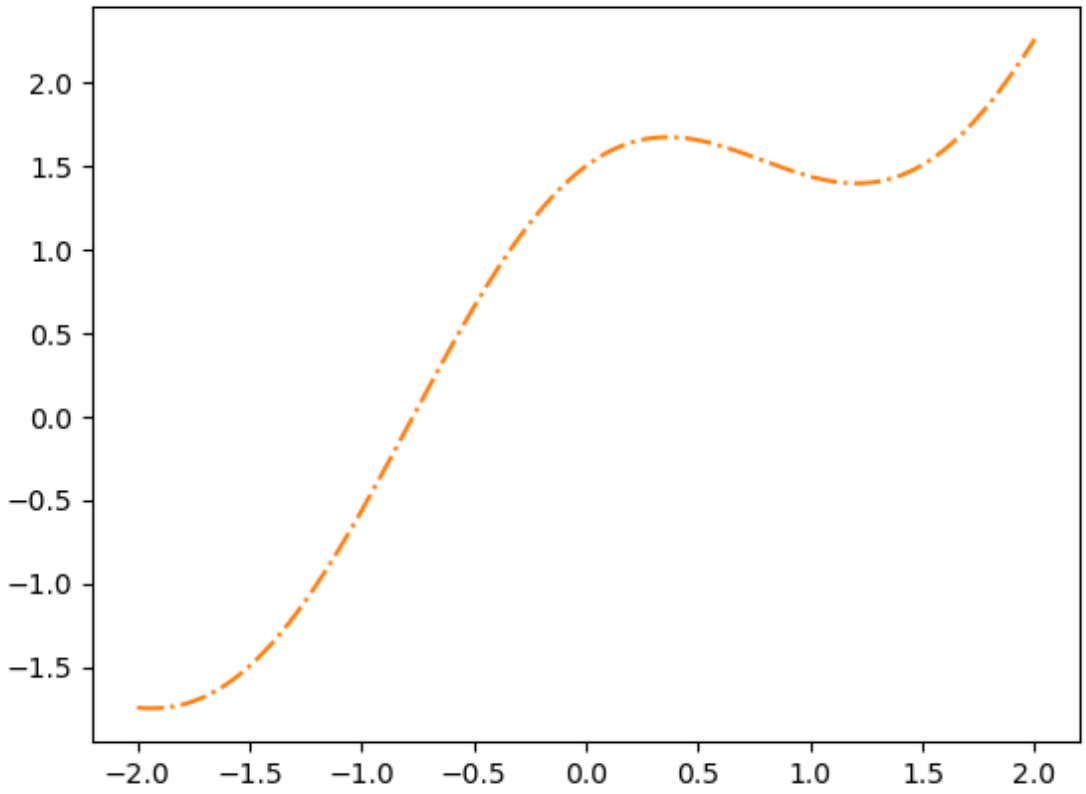
```
func=lambda x: x+1.5*np.cos(x)**2.
sol=optimize.root(func,0.2,method='lm')
print(sol.x)
#array([-0.77123776])

#sol.x has the numerical value of the solution
func(sol.x)
#array([1.11022302e-16])
```

Now, this solution obviously works (residuals are at machine precision level). But be careful. What if you had set the initial guess to be a bit different?

```
func=lambda x: x+1.5*np.cos(x)**2.
sol2=optimize.root(func,0.4,method='lm')
print(sol2.x)
#[1.20593908]

func(sol2.x)
#array([1.3969155])
```

The answer is obviously not a root of the function $f(x)$. So what happened? The answer is

clearer if we plot $f(x)$ vs an array of numbers:




Here, we see that x=1.206 corresponds to a *local minimum* of f(x) but not a zero. That is, the LM algorithm is very good at finding minima of functions. But if you give it an initial guess too far from the right answer, the solution is likely to be faulty.

Now, what about if we use this crazy "hybrid Powell's method" instead? We get the 'right' answer:

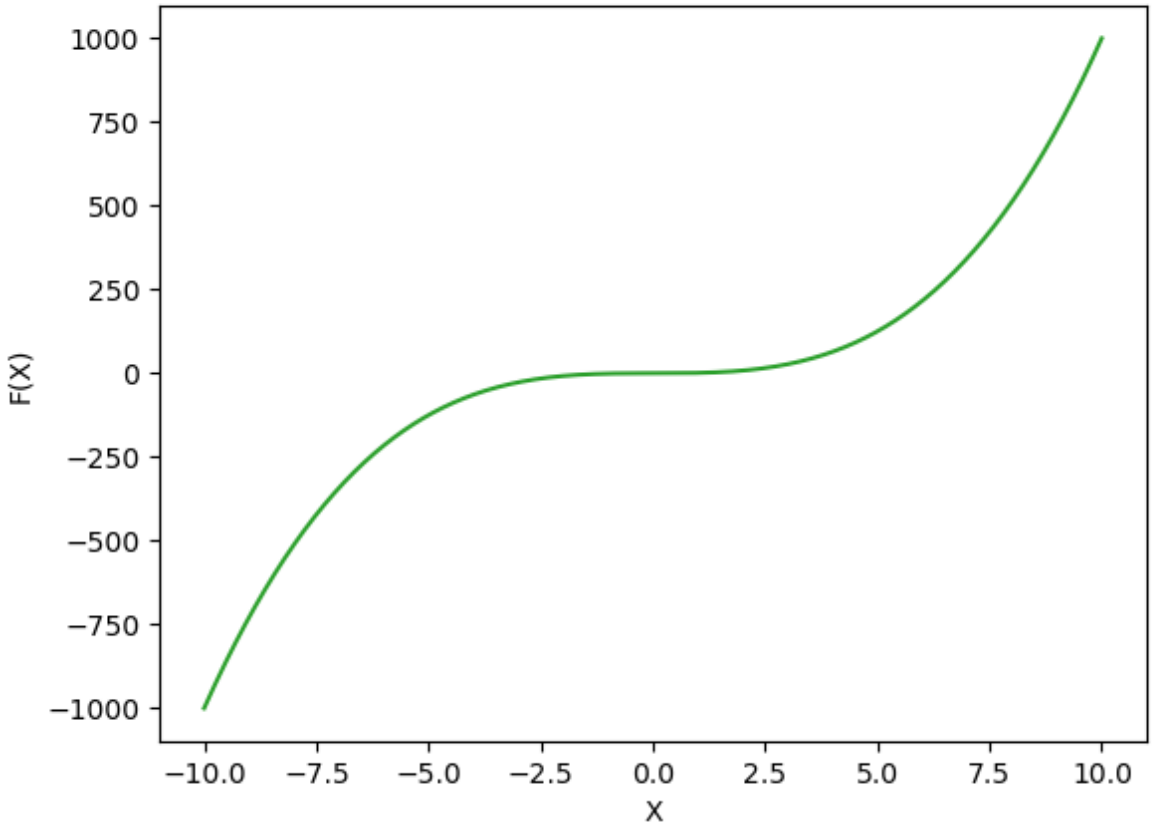
```
func=lambda x: x+1.5*np.cos(x)**2.
sol2=optimize.root(func,0.4,method='hybr')
print(sol2.x)
#array([-0.77123776])

func(sol2.x)
#array([1.11022302e-16])
```

The other functions possible with root have varying accuracy. E.g. "krylov" and "df-sane" do well; "excitingmixing" doesnt'. Be very careful about what algorithm you use for root finding!

### *A Short Tangent on Tolerancing*

For simple functions such as $f(x) = x^2 - 9$, the roots are clearly 3 and –3 as we can determine analytically. However, for other functions determining an analytic, or exact, solution for the roots of functions can be difficult. Here's one particularly evil function, $f(x) = x^3 + x^2 cos(x)$, written as: `func=lambda x: x**(3.)+1e-3*x**(2.)*np.cos(x+0.01)`

If we plot this function over the interval x = -10 to 10 we get:




There's a lot of space (x = -7.5 to 2.5) where we are ever so close to finding a root ... but not quite there.

Now, consider also the function $f(x) = 1/x$. This one does *not* have a root. If you let SciPy choose the tolerance level you will get an error the first time:

```
 func=lambda x: 1/x
 result=optimize.fsolve(func,-5,xtol=1e-8)
 #RuntimeWarning: The number of calls to function has reached maxfev = 40
0.
  warnings.warn(msg, RuntimeWarning)
```

Now, if you run it again, you might not get an error message and might not have any idea that you have a convergence error.

To check for these things turn on `full_output`

```
result=optimize.fsolve(func,-5,xtol=1e-8,full_output=True)
```

Which returns ...

```
(array([-8.80118398e+83]), {'nfev': 400, 'fjac': array([[-1.]]), 'r': arr
ay([5.46865746e-168]), 'qtf': array([1.83842764e-84]), 'fvec': array([-1.
13621077e-84])}, 2, 'The number of calls to function has reached maxfev =
 400.')
```

This is a somewhat trivial example, but you will likely encounter numerical problems in your research where you need a "good enough" solution. That's where tolerancing comes in.

For computing roots, we want an $xr$ such that $f(xr)$ is very close to 0. Therefore |$f(x)$| is a possible choice for the measure of error since the smaller it is, the likelier we are to a root. Also if we assume that $xi$ is the $i$th guess of an algorithm for finding a root, then |$xi$+1−$xi$| is another possible choice for measuring error, since we expect the improvements between subsequent guesses to diminish as it approaches a solution. As will be demonstrated in the following examples, these different choices have their advantages and disadvantages.

E.g. Let error be measured by $e$=|$f(x)$| and tol be the acceptable level of error. The function $f(x)$=$x^2$ +tol/2 has no real roots. However, |$f$(0)|=tol/2 and is therefore acceptable as a solution for a root finding program.

Another general example: Let error be measured by $e$=|$xi$+1−$xi$| and tol be the acceptable level of error. The function $f(x) = 1/x$ has no real roots as we found before. But the guesses $xi$= −tol/4 and $xi$+1=tol/4 have an error of $e$=tol/2 and is an acceptable solution for a computer program.

The use of tolerance and converging criteria must be done very carefully and in the context of the program that uses them.

## 2. Finding Minima

Many of the algorithms used for root finding are also useful for finding minima of functions. E.g. the Levenberg-Marquardt algorithm is used to find minima. See the `scipy.optimize.minimize` documentation for the full list of algorithms at your disposal.

Here, we discuss in a bit more detail the Nelder-Mead algorithm, also known as the "Amoeba" or "downhill simplex" algorithm. This algorithm has a long history (I encountered it eons ago in the standard text "Numerical Recipes"). Its wide use warrants more detailed discussion.

***Nelder-Mead Simplex Algorithm (aka "Amoeba" Algorithm)***

The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. It is a direct search method (based on function comparison) and is often applied to nonlinear optimization problems for which derivatives may not be known. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

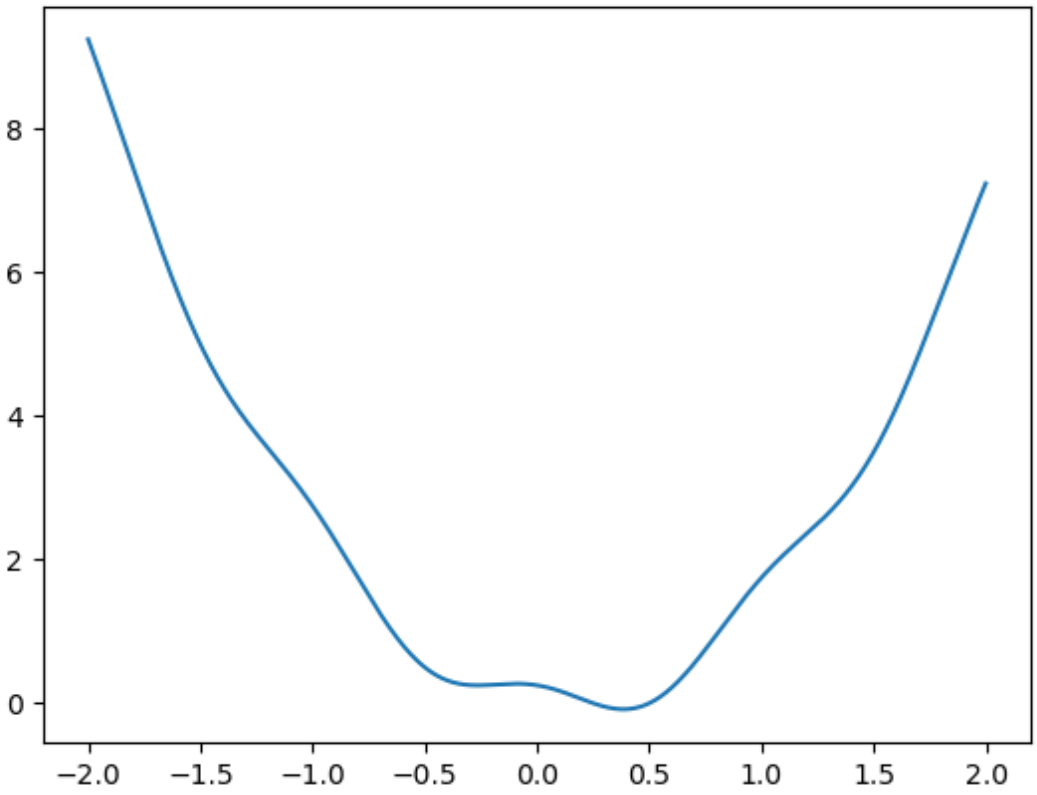
The algorithm leverages on the concept of a "simplex" which is basically jargon for "triangle in an arbitrary number of dimensions" (e.g. a true triangle in a plane; a tetrahedron in 3-D space). In $n$ dimensions, the simplex algorithm requires $n + 1$ test points. The simplest approach is to replace the worst point with a point reflected through the centroid of the remaining n points. If this point is better than the best current point, then we can try stretching exponentially out along this line. If you model the behavior of the simplex through n-dimensional space over time, it takes on the appearance of an amoeba propagating through a region of space. Hence the nickname.

Here's a really simple 1-D example of its implementation of the function $f(x) = 2x^2 - 0.5x + 0.25 * cos(\pi x^2)$, where we choose several starting points

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

func= lambda x: 2*x**2-0.5*x+0.25*np.cos(x*2*np.pi)

xstart=-1

result=optimize.minimize(func,xstart,method="nelder-mead")
print(result.x) #the estimated minimum
#[0.38486328]

xstart=-5
result=optimize.minimize(func,xstart,method="nelder-mead")
print(result.x) #the estimated minimum
#[0.3848877]

xstart=2
result=optimize.minimize(func,xstart,method="nelder-mead")
print(result.x) #the estimated minimum
#[0.38486328]
```

To visually confirm:

```
xrange=np.arange(-2,2,0.001)
plt.plot(xrange,func(xrange))
plt.show()
```



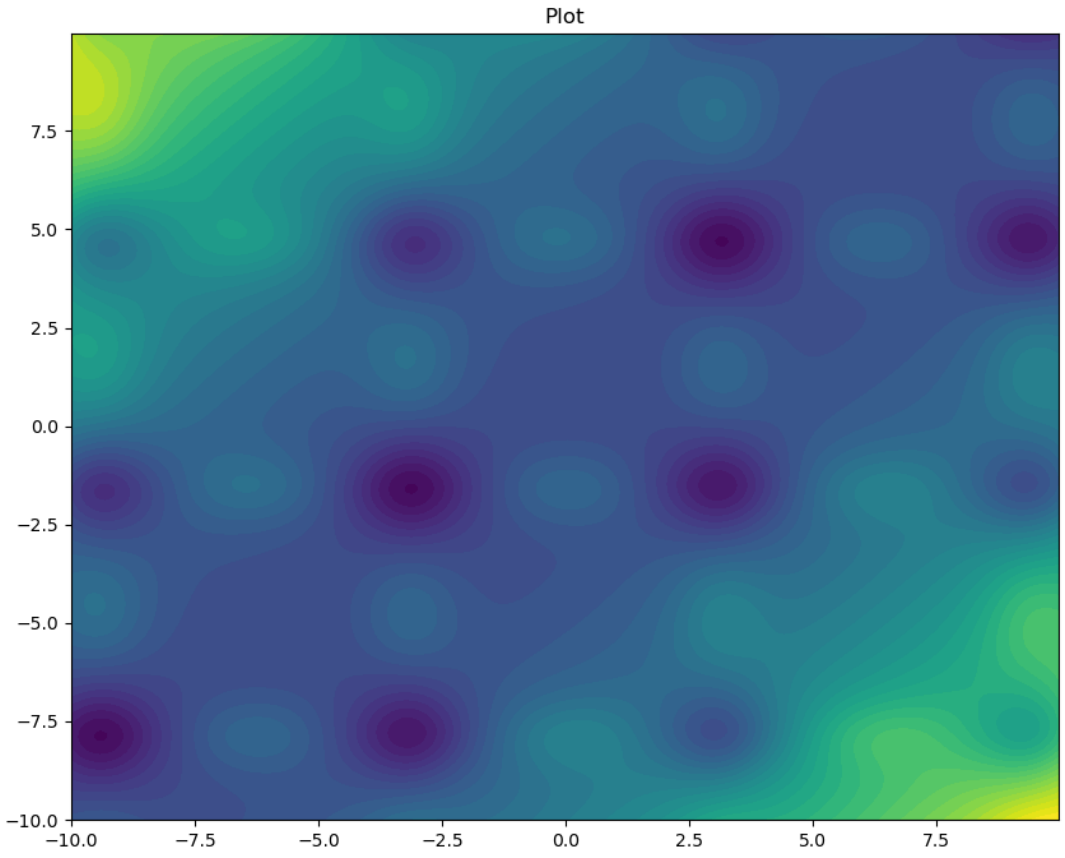

It appears as if the N-M algorithm has found the minimum of the function and avoided another local minimum at about -0.4 in the process.

Here's a more complex two-dimensional implementation, $f(x) = sin(x_1)e^{1-cos(x_o)^2} + cos(x_o)e^{1-sin(x_o)^2} + (x_o - x_1)^2$:

```
import numpy as np
from matplotlib import pyplot as plt
from scipy import optimize

#function
def f(x):
 out=np.sin(x[1])*np.exp(1-np.cos(x[0]))**2+np.cos(x[0])*np.exp(1-np.sin(
x[1]))**2+(x[0]-x[1])**2
 return out
```

We can do a contour plot of this function(x,y). You will immediately see the challenges with the N-D in this case.

```
x = np.arange(-10,10, delta)
y = np.arange(-10,10, delta)
X, Y = np.meshgrid(x, y)
Z=f([X,Y])
fig, ax = plt.subplots(figsize=(10, 8))
CS = ax.contourf(X, Y, Z, 50) #filled contour plot
ax.set_title('Plot')
plt.show()
```




There are multiple local minima within the space of the graph and it is very easy for N-D to get stuck in one.

The starting assumptions significantly affect the computed minimum. E.g.

```
#first guess
result=optimize.minimize(f,[5,2.5],method='nelder-mead')
result
#final_simplex: (array([[3.16033818, 4.69362966],
       [3.16031092, 4.69362937],
```

```
       [3.16030507, 4.69369765]]), array([-106.78773361, -106.78773361, -
106.78773351]))
           fun: -106.78773361168493
       message: 'Optimization terminated successfully.'
          nfev: 84
           nit: 45
        status: 0
       success: True
             x: array([3.16033818, 4.69362966])

result=optimize.minimize(f,[5,5],method='nelder-mead')
#same result

result=optimize.minimize(f,[-5,3],method='nelder-mead')

result
#final_simplex: (array([[-3.04674585,  4.61749862],
       [-3.04678746,  4.6175339 ],
       [-3.04673353,  4.61759561]]), array([-48.99140283, -48.99140273, -
48.99140262]))
           fun: -48.99140282891358
       message: 'Optimization terminated successfully.'
          nfev: 84
           nit: 44
        status: 0
       success: True
             x: array([-3.04674585,  4.61749862])


result=optimize.minimize(f,[-1.25,2.5],method='nelder-mead')
result
#final_simplex: (array([[0.65062186, 0.9201995 ],
       [0.65069767, 0.92011556],
       [0.65063794, 0.92023084]]), array([2.46724066, 2.46724066, 2.46724
066]))
           fun: 2.4672406559318256
       message: 'Optimization terminated successfully.'
          nfev: 91
           nit: 47
        status: 0
       success: True
             x: array([0.65062186, 0.9201995 ])

result=optimize.minimize(f,[-2.5,-2.5],method='nelder-mead')
result
```

```
#final_simplex: (array([[-3.12288422, -1.58956166],
       [-3.12281825, -1.58950298],
       [-3.12284585, -1.58947105]]), array([-106.78773355, -106.78773349,
 -106.78773339]))
           fun: -106.78773355031949
       message: 'Optimization terminated successfully.'
          nfev: 68
           nit: 35
        status: 0
       success: True
             x: array([-3.12288422, -1.58956166])
```

There are two minima of the same value here (at about 3.16,4.69 and -3.12, -1.59). The basic point is that N-D can easily get trapped in local minima or generally 'poor' solutions.

## 3. Curve Fitting (with SciPy *and* NumPy)

The SciPy provides a `curve_fit` function in its optimization library to fit the data with a given function. This method applies non-linear least squares to fit the data and extract the optimal parameters out of it.

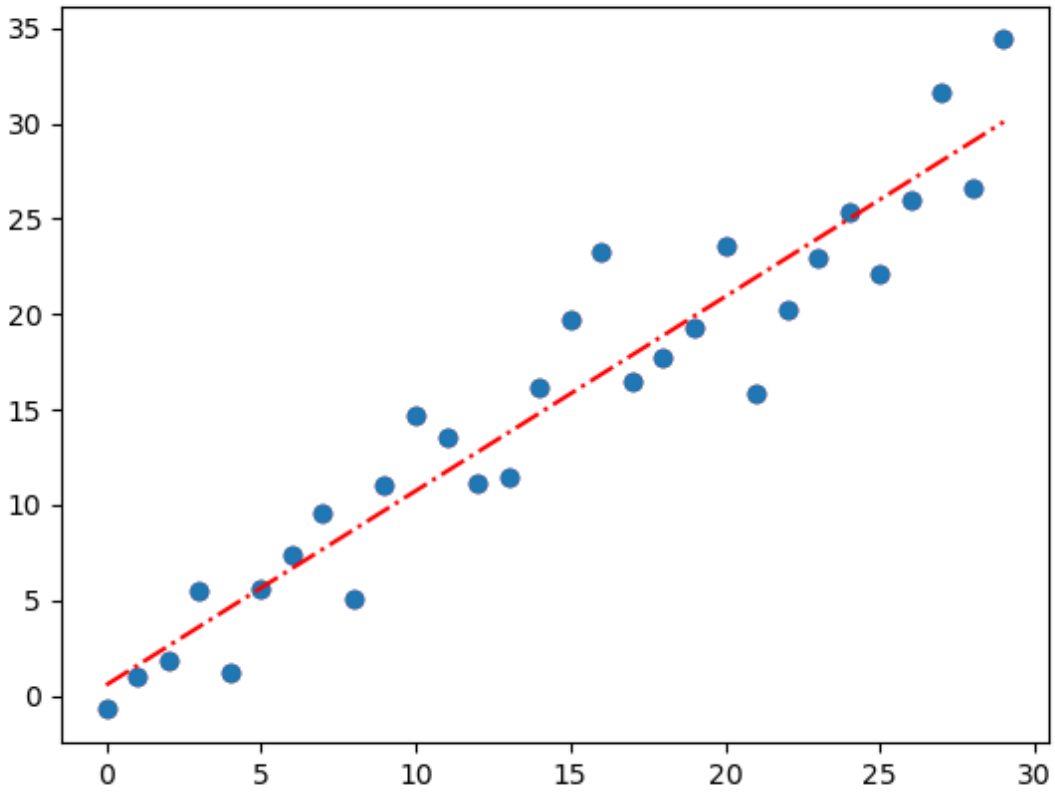
NumPy also provides fitting capabilities through the `np.polyfit` function (prior to v1.4; I'm more familiar with this) and `np.polynomial` function (newer version).

The major difference between these two is that `curve_fit` requires you to supply a functional fit; `np.polyfit` is just that (a polynomial of some specified degree where you solve for the coefficients). In this sense, I guess that `curve_fit` is more powerful.

We will demonstrate a simple polynomial fit to data using both libraries then a more complex one fitting a functional form with SciPy.

First, the simple case

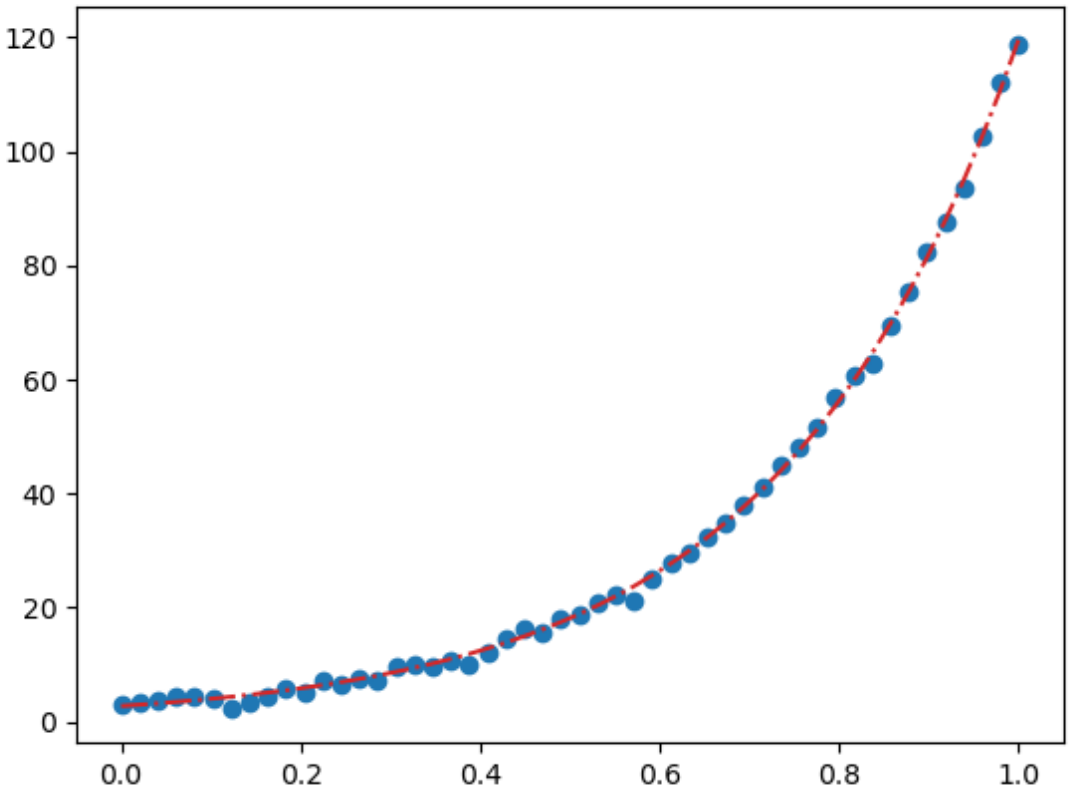
```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

xarray=np.arange(30)
yarray=np.arange(30)+3*np.random.randn(30)

#numpy
result=np.polyfit(xarray,yarray,1)
print(result)
#array([1.01815066, 0.54987548])

#scipy
def func(x,a,b):
 return a*x+b

param, param_cov = curve_fit(func, xarray, yarray)
print(param)
#array([1.01815066, 0.54987548])

#the fit with SciPy
plt.plot(xarray,ans,'-.',color='red')
#or with NumPy
#plt.plot(xarray,result[0]*xarray+result[1],'-.',color='red')
plt.plot(xarray, yarray, 'o', color ='tab:blue') #the data
```




Obviously, we just had a simple linear plot with some noise added and not surprisingly that is what both NumPy and SciPy found.

Here's one that is more complex, and where we need SciPy to do the work

```
#generate fake data
xarray = np.linspace(0, 1, num = 50)
yarray = 2.7 * np.exp(3.8 * xarray) + np.random.normal(size = 50) #some e
xponential with noise

def test(xarray, a, b):
    return a*np.exp(b*xarray)

param,param_cov = curve_fit(test,xarray,yarray)

print(param)

plt.plot(xarray,test(xarray,param[0],param[1]),'-.',color='tab:red')
plt.scatter(xarray,yarray)
```




Note here that the assumed functional form is key. For example, if you are going to model your data as a simple cosine function then the function you port into `curve_fit` should look like:

```
ef test(xarray, a, b):
    return a*np.cos(b*x)
```