Non-linear Regression

If your model function is *not* linear in its parameters, there is no general analytic solution to solve for the MLE parameters and their covariance matrix. We can still maximize the likelihood (or minimize the χ^2), but we must resort to other methods to find the MLE. One of the more commonly used routine is `scipy.optimize.curve_fit`.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

Here is an example of how to use it.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

Define function with a non-linear parameter (b):

$$y = ae^{-bx} + c$$

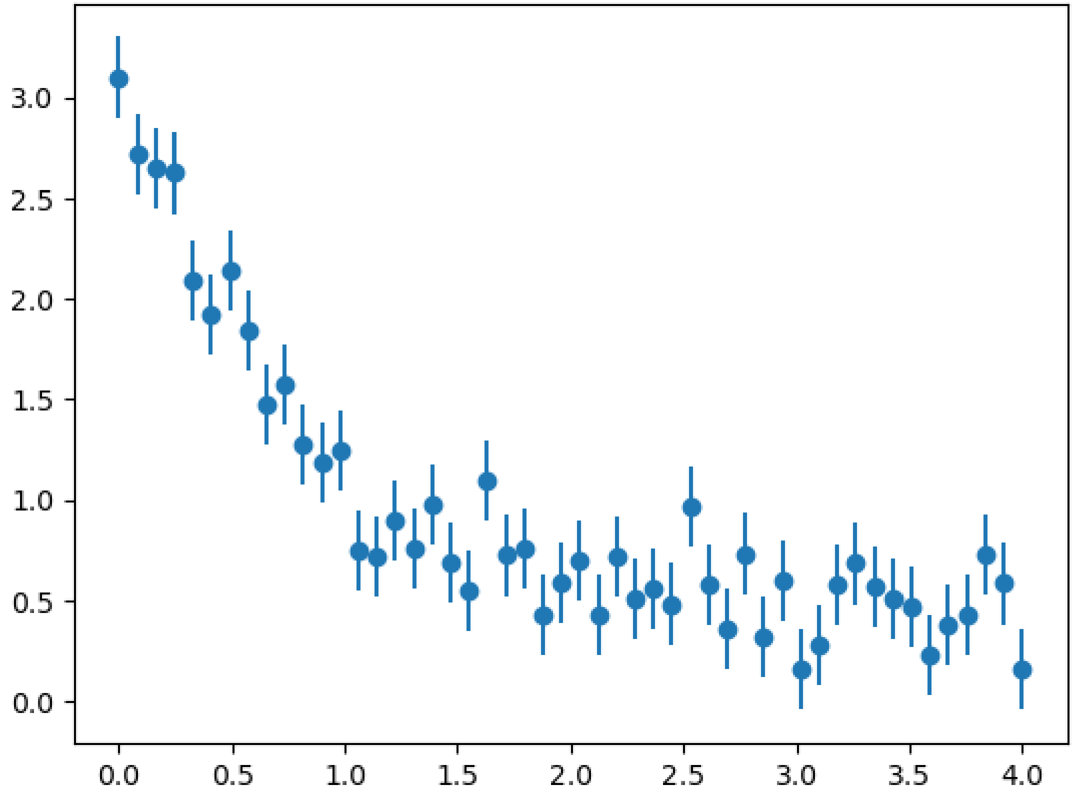
In [2]:
```
def func(x, a, b, c):
    return a * np.exp(-b * x) + c
```

Generate simulated dataset using the above function.

In [3]:
```
np.random.seed(42)                      # define random seed for repeatability
xdata = np.linspace(0, 4, 50)           # 50 points from x=[0,4]
y = func(xdata, 2.5, 1.3, 0.5)          # a=2.5, b=1.3, c=0.5
ysig   = 0.2                            # common y uncertainty
ydata  = np.random.normal(y, ysig)      # normal distribution N(y,ysig)
yerror = np.full_like(ydata, ysig)      # fill out yerror vector with ysig=0.2
```

In [4]:
```
plt.errorbar(xdata, ydata, yerror, fmt='o')
```

Out[4]:
```
<ErrorbarContainer object of 3 artists>
```



You can call `curve_fit` with the following arguments - 1) model function (`func` in this case), the x data, y data, and optionally error in y. The last keyword tells `curve_fit` that `yerror` is an absolute uncertainty. The output `popt` and `pcov` are the means and covariance matrix.

In [5]:
```
ahat, covmat = curve_fit(func, xdata, ydata,
                         sigma=yerror, absolute_sigma=True)
```

In [6]:
```
print("[a,b,c] = ", ahat)   # best-fit values
```

```
[a,b,c] =  [2.73144648 1.38015943 0.43816933]
```

In [7]:
```
print(covmat)   # and the covariance matrix
```

```
[[ 0.0146974   0.00609427 -0.00066339]
 [ 0.00609427  0.01667886  0.00456777]
 [-0.00066339  0.00456777  0.00248719]]
```

In [8]:
```
# diagonal terms
print("a = %7.3f +/- %7.3f  (true a = 2.5)" % (ahat[0], np.sqrt(covmat[0,0])))
print("b = %7.3f +/- %7.3f  (true b = 1.3)" % (ahat[1], np.sqrt(covmat[1,1])))
print("c = %7.3f +/- %7.3f  (true c = 0.5)" % (ahat[2], np.sqrt(covmat[2,2])))
```

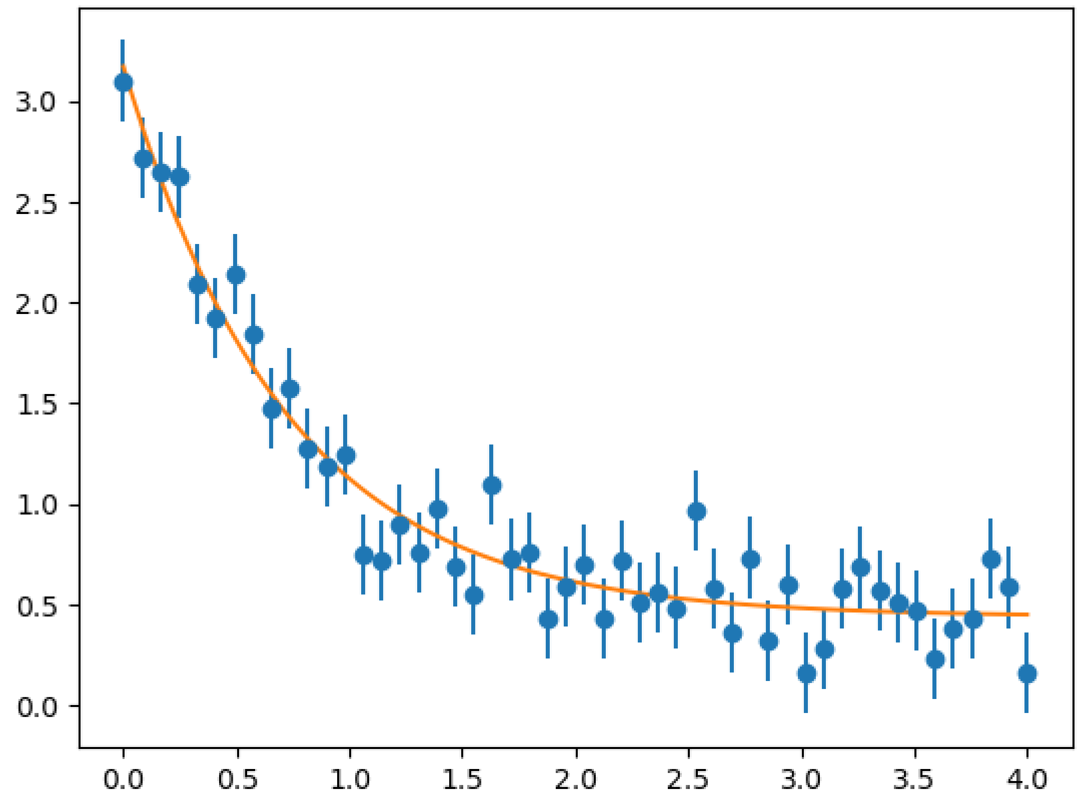
```
a =   2.731 +/-   0.121  (true a = 2.5)
b =   1.380 +/-   0.129  (true b = 1.3)
c =   0.438 +/-   0.050  (true c = 0.5)
```

Let's overplot the data and the best-fit model.

In [9]:
```
plt.errorbar(xdata, ydata, yerror, fmt='o')
xgrid = np.linspace(0.0, 4.0, 100)
```

```
plt.plot(xgrid, func(xgrid, *ahat))
```

Out[9]: [<matplotlib.lines.Line2D at 0x10db86fe0>]



Next, a slightly more complicated model

$$y = a + be^{-\frac{1}{2}\left(\frac{x-c}{d}\right)^2}$$

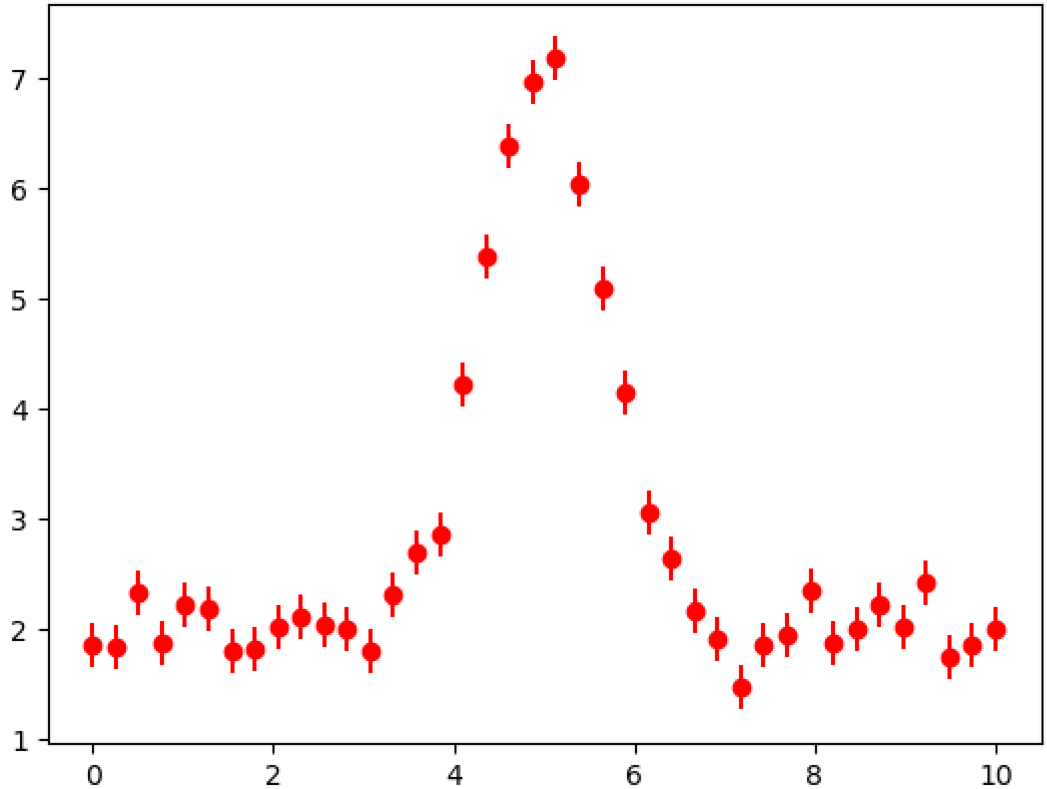
A constant plus a gaussian function.

In [10]:
```
def func2(x, a, b, c, d):
    return a + b * np.exp(-0.5*((x-c)/d)**2)
```

In [11]:
```
np.random.seed(1729)
xdata = np.linspace(0, 10, 40)
y = func2(xdata, 2.0, 5.0, 5.0, 0.7)   # a=2.0, b=5.0, c=5.0, d=0.7
ysig   = 0.2
ydata  = np.random.normal(y, ysig)
yerror = np.full_like(ydata, ysig)
```

In [12]:
```
plt.errorbar(xdata, ydata, yerror, fmt='ro')
```

Out[12]: <ErrorbarContainer object of 3 artists>



In [13]:
```
ahat, covmat = curve_fit(func2, xdata, ydata,
                         sigma=yerror, absolute_sigma=True)
```

In [14]:
```
print(ahat)
print(covmat)
```

```
[ 3.19794001 -1.47693057  1.39869991 -1.16487622]
[[ 0.00180206 -0.00142581  0.00010613 -0.00173541]
 [-0.00142581  0.00917379 -0.00111718 -0.00408403]
 [ 0.00010613 -0.00111718  0.00824811  0.00277496]
 [-0.00173541 -0.00408403  0.00277496  0.01199062]]
```

Best-fit values are $a = 3.2, b = -1.5, c = 1.4, d = -1.7$, which is not close to what we put in. What is going on? Whatever starting point `curve_fit` is using for the initial guess is bad, and fails to find the global mininum. This is very common problem with essentially all non-linear fitting routines; it is not easy to automatically find the global χ^2 mininum. `curve_fit` can take `bounds`, which helps guide the fit.

In [15]:
```
ahat, covmat = curve_fit(func2, xdata, ydata,
                         sigma=yerror, absolute_sigma=True,
                         bounds=([1, 4, 4, 0.3],[3, 6, 6, 1.0]))
```

In [16]:
```
print(ahat)
print(covmat)
```

```
[1.98239261 5.13116883 4.98366601 0.66570498]
[[ 1.52702045e-03 -1.07976650e-03 -5.38782124e-13 -2.80172408e-04]
 [-1.07976650e-03  1.38020781e-02  1.29563474e-10 -9.29615596e-04]
 [-5.38782124e-13  1.29563474e-10  2.92617059e-04 -1.67308106e-11]
 [-2.80172408e-04 -9.29615596e-04 -1.67308106e-11  3.44022118e-04]]
```

In [17]:
```
print("a = %7.3f +/- %7.3f  (true a = 2.0)" % (ahat[0], np.sqrt(covmat[0,0])))
print("b = %7.3f +/- %7.3f  (true b = 5.0)" % (ahat[1], np.sqrt(covmat[1,1])))
print("c = %7.3f +/- %7.3f  (true c = 5.0)" % (ahat[2], np.sqrt(covmat[2,2])))
print("d = %7.3f +/- %7.3f  (true c = 0.7)" % (ahat[3], np.sqrt(covmat[3,3])))
```

```
a =   1.982 +/-   0.039  (true a = 2.0)
b =   5.131 +/-   0.117  (true b = 5.0)
c =   4.984 +/-   0.017  (true c = 5.0)
d =   0.666 +/-   0.019  (true c = 0.7)
```

These are consistent with the true values.

In [ ]:

Another popular, but much more complicated (and also very flexible), non-linear curve fitter used by natural scientists is `lmfit`:

https://lmfit.github.io/lmfit-py/

`statsmodel` is also popular amongst the social scientists:

https://www.statsmodels.org/stable/index.html

Both are massive modules with extensive documentation.

In [ ]:

In [ ]:

Runtime comparisons

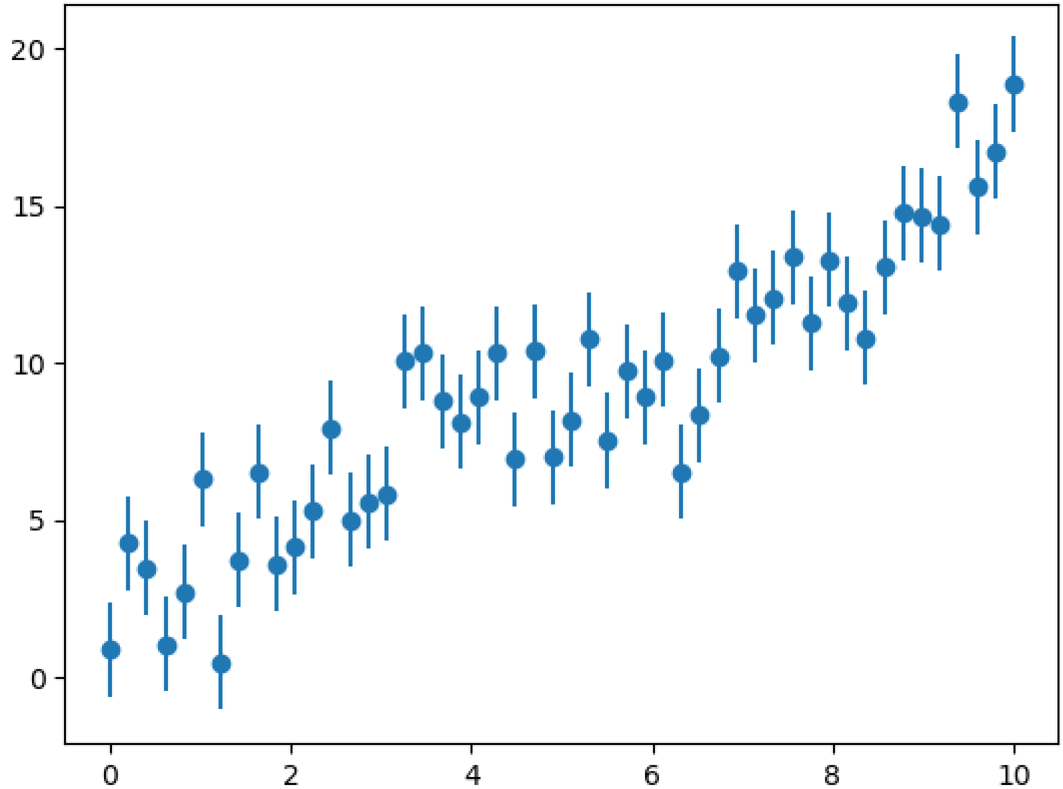
Using matrix algebra to solve for the parameters (assuming that your model is linear!) is almost always faster than using non-linear fitting modules. This is especially true when there are many fitting parameters.

In [18]:
```
def func3(x, a, b):
    return a + b*x
```

In [19]:
```
np.random.seed(123)                     # define random seed for repeatability
xdata = np.linspace(0, 10, 50)          # 50 points from x=[0,10]
y = func3(xdata, 2.5, 1.3)              # a=2.5, b=1.3, c=0.5
ysig   = 1.5                            # common y uncertainty of ysig=0.2
ydata  = np.random.normal(y, ysig)      # normal distribution N(y,ysig)
yerror = np.full_like(ydata, ysig)      # fill out yerror vector with ysig=0.2
```

In [20]:
```
plt.errorbar(xdata, ydata, yerror, fmt='o')
```

Out[20]:
```
<ErrorbarContainer object of 3 artists>
```



In [21]:
```
def matrix_fit(xdata, D, yerror):
    G1 = np.ones_like(xdata)
    G2 = xdata
    G = np.vstack([G1, G2]).T
    E = np.diag(yerror*yerror)
    Einv = np.linalg.inv(E)
    covmat = np.linalg.inv(np.dot(G.T, np.dot(Einv, G)))
    ahat   = np.dot(covmat, np.dot(G.T, np.dot(Einv, D)))
    return ahat, covmat
```

In [22]:
```
%timeit ahat, covmat = curve_fit(func3, xdata, ydata, sigma=yerror, absolute_s:
```

```
209 µs ± 1.23 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [23]:
```
%timeit ahat, covmat = matrix_fit(xdata, ydata, yerror)
```

```
97.6 µs ± 3.18 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

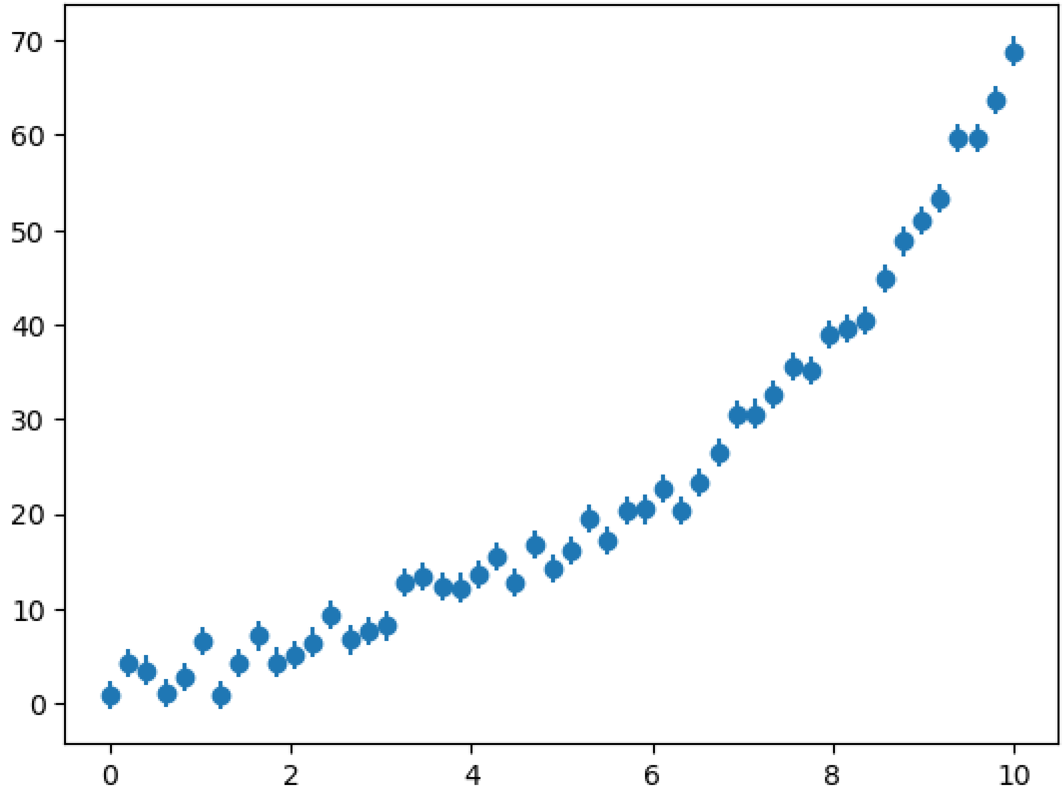
Let's experiment with a model with more parameters, but still linear:

In [24]:
```
def func4(x, a, b, c, d, e):
    return a + b*x + c*x*x + d*x*x*x + e*x*x*x*x
```

In [25]:
```
np.random.seed(123)                     # define random seed for repeatability
xdata = np.linspace(0, 10, 50)          # 50 points from x=[0,10]
y = func4(xdata, 2.5, 1.3, 0.2, 0.01, 0.002)            # a=2.5, b=1.3, c=0.5
ysig   = 1.5                            # common y uncertainty of ysig=0.2
ydata  = np.random.normal(y, ysig)      # normal distribution N(y,ysig)
yerror = np.full_like(ydata, ysig)      # fill out yerror vector with ysig=0.2
```

In [26]:
```
plt.errorbar(xdata, ydata, yerror, fmt='o')
```

Out[26]: <ErrorbarContainer object of 3 artists>



In [27]:
```
def matrix_fit(xdata, D, yerror):
    G1 = np.ones_like(xdata)
    G2 = xdata
    G = np.vstack([G1, G2]).T
    E = np.diag(yerror*yerror)
    Einv = np.linalg.inv(E)
    covmat = np.linalg.inv(np.dot(G.T, np.dot(Einv, G)))
    ahat   = np.dot(covmat, np.dot(G.T, np.dot(Einv, D)))
    return ahat, covmat
```

In [28]:
```
%timeit ahat, covmat = curve_fit(func4, xdata, ydata, sigma=yerror, absolute_s:
```

340 µs ± 4.77 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

In [29]:
```
%timeit ahat, covmat = matrix_fit(xdata, ydata, yerror)
```

98.2 µs ± 3.08 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

In [ ]: