StefinRacho-Homework5

October 12, 2024

0.1 CS 4010: Homework 5

0.1.1 Due Monday, 10/14/2024

0.1.2 Create a copy of the code below in an iPython (Jupyter) notebook. Name the notebook FirstAndLastName_Homework5 and save it before you start working

0.1.3 To submit, export or print your notebook as a pdf, with all outputs visible. Upload both the pdf and a copy of your notebook (.ipynb) in Canvas.

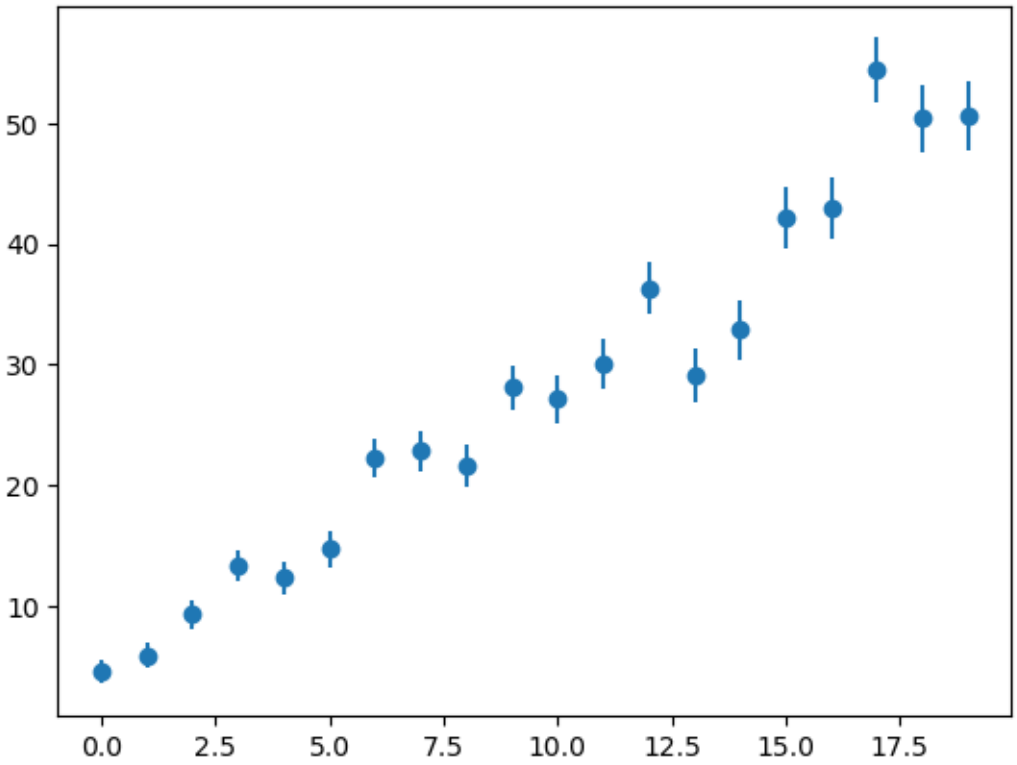
0.1.4 Curve Fitting

[204]:
```
# First, load any modules you need here.
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import colors as colors
import scipy.optimize as optimize
```

0.2 1) Line and Polynomial Fitting

The code below will generate some locatons, x1, data points, data1, and error on the data, sigma1.

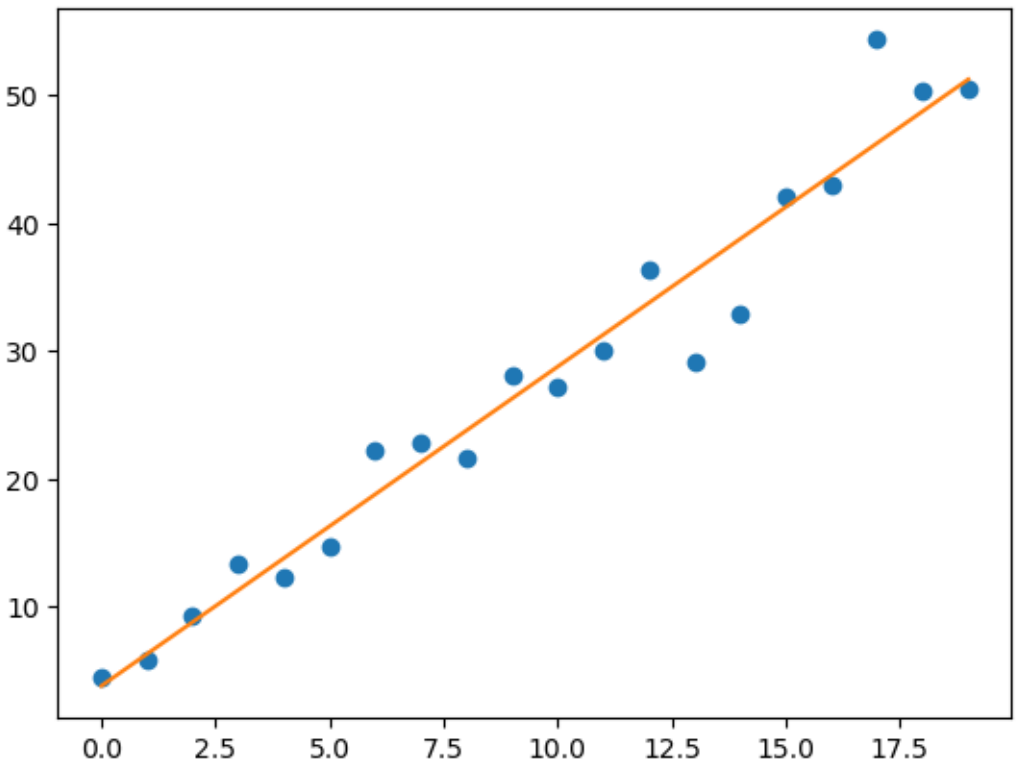
[207]:
```
n = 20

x1 = np.linspace(0,n-1,n)  # make x from 0 to n-1 for now

np.random.seed(42)     # This seeds a random number generator.  Using a fixed␣
 ↪seed makes out output reproducable.

data1 = 4. + 2.*x1 + 0.05*x1**2. + np.random.randn(n)*(x1*.3+1)

sigma1 = (x1*.1+1)

plt.errorbar(x1,data1,sigma1,marker='o',linestyle='')
plt.show()
```



Using this data, answer the following questions:

a) If you ignore the error and do a linear fit, what value of m and b do you get? What are the errors on each of these values?

[211]:
```
def f_linear(x,m,b):
    return m*x+b

fit_params, covariance = np.polyfit(x1, data1, 1, cov=True)

m, b = fit_params

errors = np.sqrt(np.diag(covariance))
m_error, b_error = errors

fit = f(x1, m, b)
plt.plot(x1, data1, "o")
plt.plot(x1, fit)
plt.show()
print('best fit m and b = ',m,b)
print(f"Error for m: {m_error}")
print(f"Error for b: {b_error}")
```

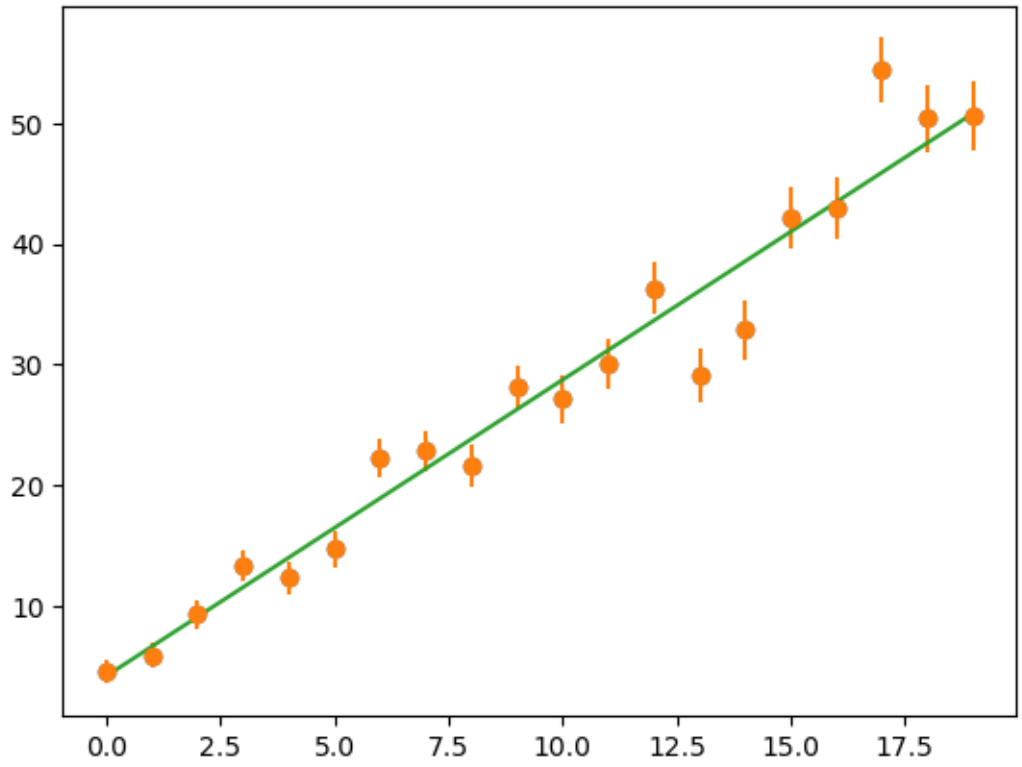



```
best fit m and b =  2.5007240820405356 3.7654633042768877
Error for m: 0.12926536468814956
Error for b: 1.4365331571993376
```

b) If you include the errors, what values of m and b do you get? What are the errors on these values? How close are these values to the ones used to create the data (m = 2, b = 4)? Are the fitted values within 1-sigma of the real values?

[214]:
```
sigma = data1*0.1
weight = 1/sigma

fit_params_linear, covariance_linear = np.polyfit(x1, data1, 1, w=1/sigma1,␣
 ↪cov=True)

m_linear, b_linear = fit_params_linear

errors_linear = np.sqrt(np.diag(covariance_linear))
m_error_linear, b_error_linear = errors_linear

fit_linear = f(x1, m_linear, b_linear)
plt.plot(x1, data1, "o")
plt.errorbar(x1,data1,sigma1,marker='o',linestyle='')
plt.plot(x1, fit_linear)
```

```
plt.show()

real_m_linear, real_b_linear = 2, 4
m_diff = abs(m - real_m_linear)
b_diff = abs(b - real_b_linear)

m_within_1sigma = m_diff <= m_error_linear
b_within_1sigma = b_diff <= b_error_linear

print('Best fit m and b (with errors included) =', m_linear, b_linear)
print(f"Error for m: {m_error_linear}")
print(f"Error for b: {b_error_linear}")
print(f"Is m within 1-sigma of real value (m = 2)? {m_within_1sigma}")
print(f"Is b within 1-sigma of real value (b = 4)? {b_within_1sigma}")
```

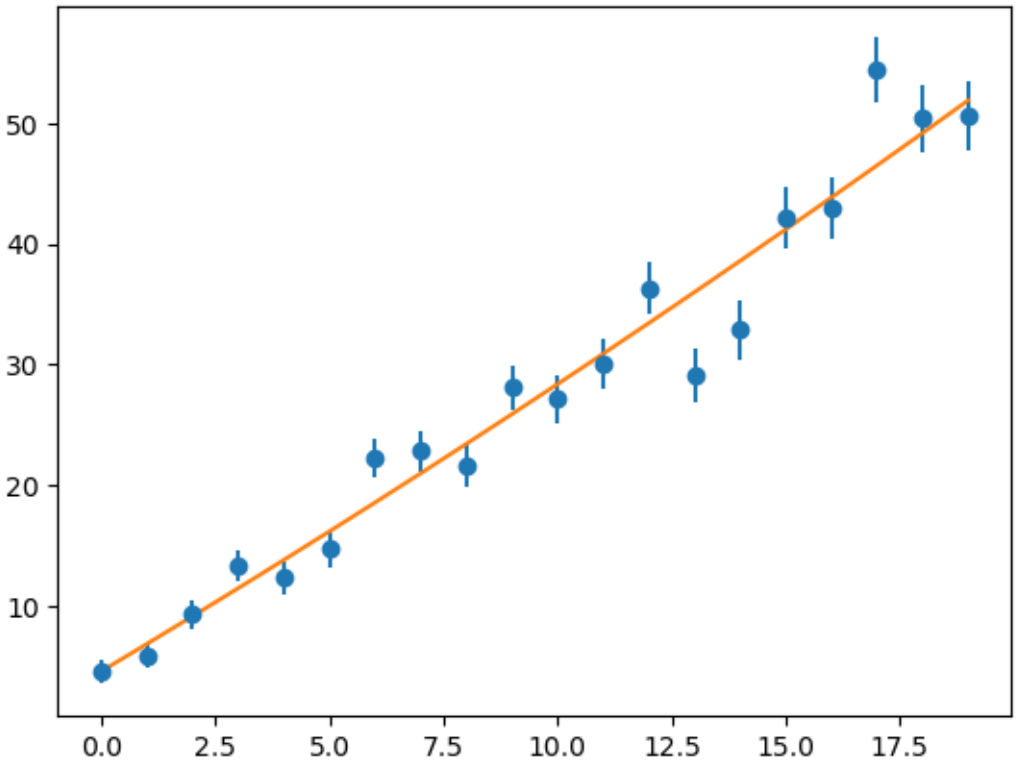



```
Best fit m and b (with errors included) = 2.4568167754083032 4.149959980820261
Error for m: 0.10653166993370844
Error for b: 0.8439974708440232
Is m within 1-sigma of real value (m = 2)? False
Is b within 1-sigma of real value (b = 4)? True
```

c) Notice that the formula used to make the data also has an x^2 term. If you fit the data with a degree-2 polynomial, what values of the 3 parameters do you get, and what is the error on

each parameter? Are these values consistent with the values used to make the data?

[217]:
```
def f_quadratic(x, a, b, c):
    return a * x**2 + b * x + c

fit_params_quad, covariance_quad = np.polyfit(x1, data1, 2, w=1/sigma1,␣
 ↪cov=True)

a_quad, b_quad, c_quad = fit_params_quad

errors_quad = np.sqrt(np.diag(covariance_quad))
a_error_quad, b_error_quad, c_error_quad = errors_quad

fit_quad = f_quadratic(x1, a_quad, b_quad, c_quad)
plt.errorbar(x1, data1, sigma1, marker='o', linestyle='')
plt.plot(x1, fit_quad)
plt.show()

print(f"Best fit parameters (quadratic): a = {a_quad}, b = {b_quad}, c =␣
 ↪{c_quad}")
print(f"Error for a: {a_error_quad}")
print(f"Error for b: {b_error_quad}")
print(f"Error for c: {c_error_quad}")

real_a, real_b, real_c = 0.05, 2, 4
print(f"Real values: a = {real_a}, b = {real_b}, c = {real_c}")
```




```
Best fit parameters (quadratic): a = 0.012250986360583546, b =
2.259603515593996, c = 4.543111869986923
Error for a: 0.021027129444512405
Error for b: 0.35546644593133053
Error for c: 1.093074049236864
Real values: a = 0.05, b = 2, c = 4
```

I would say that these values are consistent with the values used to make the data.

d) By using a degree-2 polynomial in part c, how much better is the fit than in part b)? You can determine this by comparing the values of χ^2. Is using the additional parameter reasonable when you compare the reduced χ^2?

[221]:
```python
def chi_squared(y_observed, y_fitted, sigma):
    return np.sum(((y_observed - y_fitted) / sigma) ** 2)

def reduced_chi_squared(chi2, N, p):
    return chi2 / (N - p)

chi2_linear = chi_squared(data1, fit_linear, sigma1)
chi2_quadratic = chi_squared(data1, fit_quad, sigma1)

N = len(data1)
```

```
p_linear = 2
p_quadratic = 3

reduced_chi2_linear = reduced_chi_squared(chi2_linear, N, p_linear)
reduced_chi2_quadratic = reduced_chi_squared(chi2_quadratic, N, p_quadratic)

print(f"Chi-squared for linear fit: {chi2_linear}")
print(f"Reduced chi-squared for linear fit: {reduced_chi2_linear}")
print(f"Chi-squared for quadratic fit: {chi2_quadratic}")
print(f"Reduced chi-squared for quadratic fit: {reduced_chi2_quadratic}")
```

```
Chi-squared for linear fit: 40.82956426119699
Reduced chi-squared for linear fit: 2.2683091256220553
Chi-squared for quadratic fit: 40.03024286229497
Reduced chi-squared for quadratic fit: 2.3547201683702923
```

The quadratic fit barely improves the chi-squared compared to the linear fit, and the reduced chi-squared for the quadratic fit is also slightly worse. Thus, it's not reasonable to include the additional parameter.

0.3 2) Curve Fitting Failure

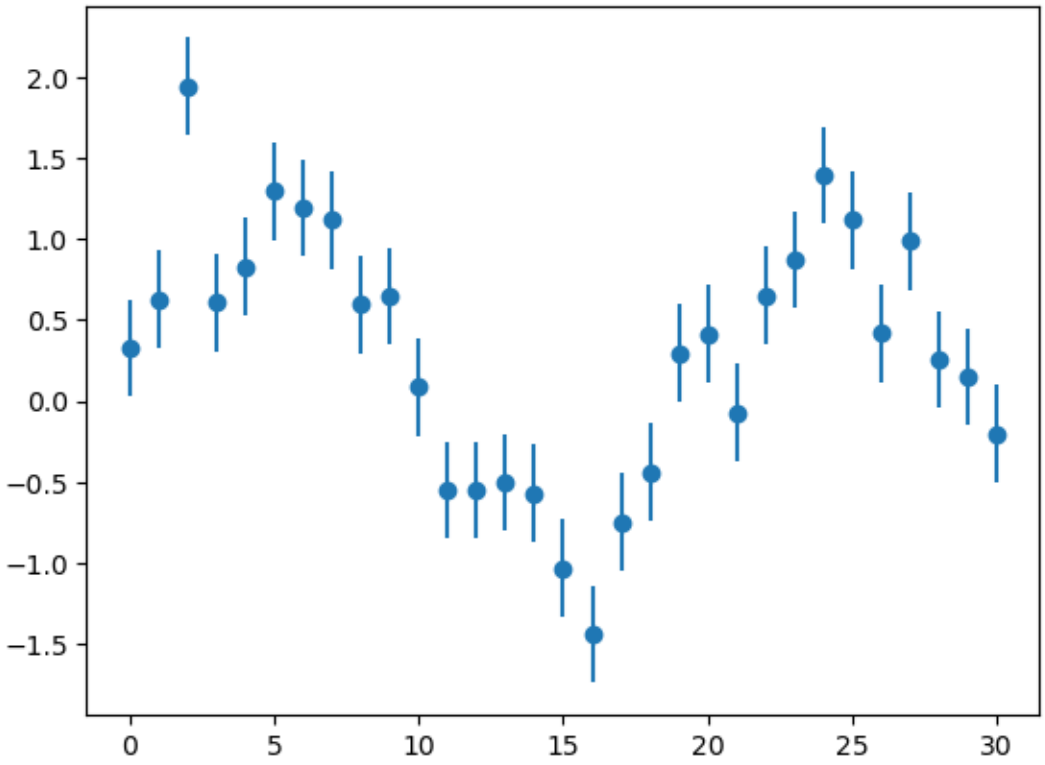
Below is code to generate some data we saw in class, for a sin function plus some noise. In class we saw a decent fit to this, using:

```
p=[1.,18.,0.]  # initial guess
bounds = [[0,0,0],[np.inf,np.inf,2*np.pi]]  # bounds on parameters
popt,pcov = optimize.curve_fit(sin_fit,x,y,p,sigma=sigma,bounds=bounds,absolute_sigma=True,meth
```

However, we can also make the fitting fail.

[225]:
```
# First, make some data:

n = 31  # number of points

x = np.linspace(0,n-1,n)  # make x from 0 to n-1 for now

np.random.seed(137)    # This seeds a random number generator.  Using a fixed␣
 ↪seed makes out output reproducable.


# Make a sin wave with some point.  Period is 20., amplutide is 1., phase = 0

y = np.sin(x/(n-1)*np.pi*3) + np.random.randn(n)*0.3    # Add some random noise␣
 ↪with sigma = 0.3

sigma = np.zeros(n)+0.3  # Set sigma to 0.3

plt.errorbar(x,y,sigma,marker='o',linestyle='')
```

```
plt.show()
```



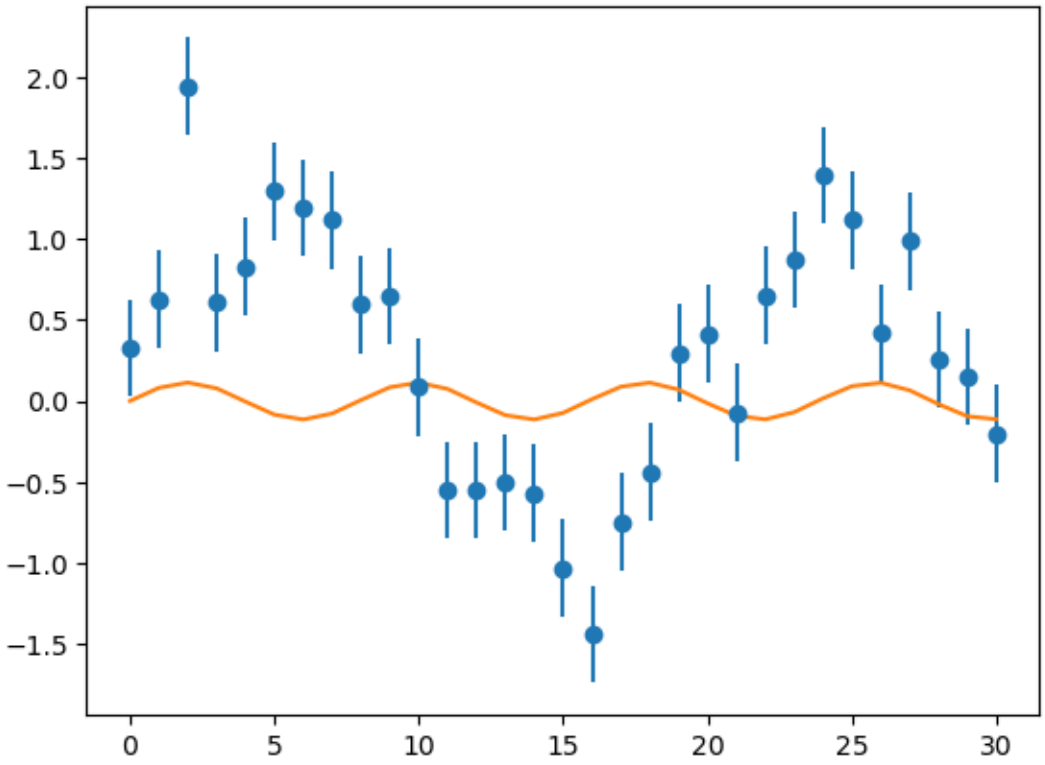
a) Try changing the initial guess to something else, say

p=[1.,8.,0.]

and rerun the fitting. What happens? Does this look like a good fit? Try changing your initial guess a few times to see roughly how close your guess has to be to get a good fit. Why does the fitting fail?

[228]:
```
def sin_fit(x,a,b,c):
    fit = a*np.sin(x*2*np.pi/b + c)
    return fit

p=[1.,8.,0.]  # initial guess
bounds = [[0,0,0],[np.inf,np.inf,2*np.pi]]  # bounds on parameters
popt,pcov = optimize.
 ↪curve_fit(sin_fit,x,y,p,sigma=sigma,bounds=bounds,absolute_sigma=True,method='dogbox')

fit = sin_fit(x,popt[0],popt[1],popt[2])

plt.errorbar(x,y,sigma,marker='o',linestyle='')
plt.plot(x,fit)
```

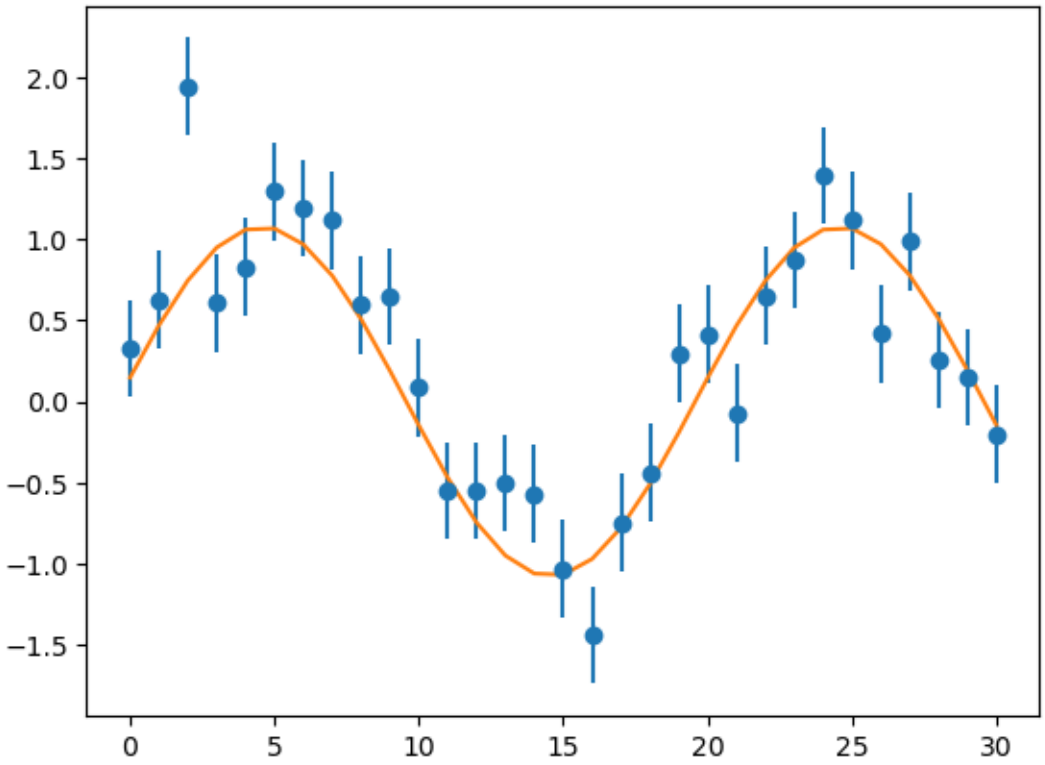
```
plt.show()
```



After rerunning with p=[1., 8., 0.,] the sin function does not properly fit the data. This does not look like a good fit. The fitting fails because the initial guess for the phase is too far off

b) How about if you try different methods for finding the best fit, i.e. 'lm', 'trf', 'dogbox'? (For 'lm', you'll need to take out the 'bounds' option.) Do some methods find a good fit over a wide range than others?

[232]:
```
p=[1.,18.,0.]  # initial guess
popt,pcov = optimize.
  ↪curve_fit(sin_fit,x,y,p,sigma=sigma,bounds=bounds,absolute_sigma=True,method='trf')

fit = sin_fit(x,popt[0],popt[1],popt[2])

plt.errorbar(x,y,sigma,marker='o',linestyle='')
plt.plot(x,fit)
plt.show()
```



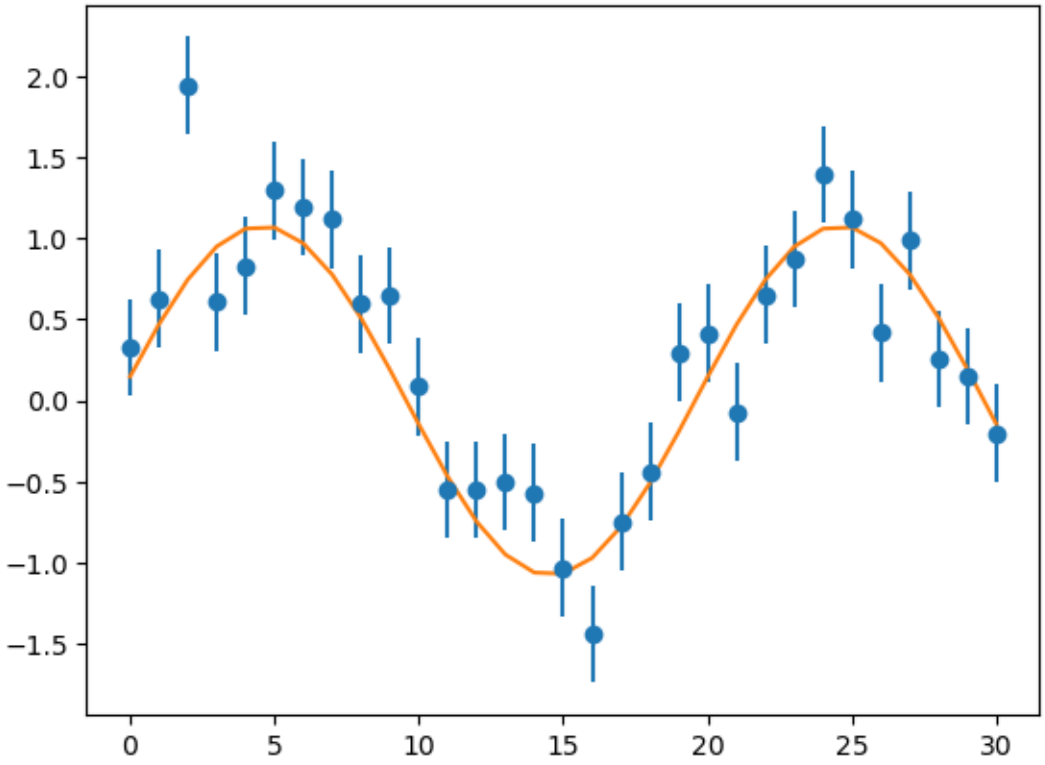
After some research, lm is best for smooth problems without bounds where a good initial guess is available. For trf, it's useful when you need to have bounds. And for dogbox, its useful for complex problems with bounds where the parameter space is complex, however this method is slower.

c) The optimize.minimize function has a wide range of methods available. First, write a new function that can be fed to optimize.minimize to find the best fit for this data. It should return the weighted χ^2 for the parameters passed to it.

[236]:
```
def chi_squared(params, x, y, sigma):
    a, b, c = params
    y_fitted = sin_fit(x, a, b, c)
    chi2 = np.sum(((y - y_fitted) / sigma) ** 2)
    return chi2

result = optimize.minimize(chi_squared, p, args=(x, y, sigma))

best_fit_params = result.x
print("Best-fit parameters (amplitude, period, phase):", best_fit_params)

fit = sin_fit(x, *best_fit_params)

plt.errorbar(x, y, sigma, marker='o', linestyle='')
plt.plot(x, fit)
```

```
plt.show()
```

```
Best-fit parameters (amplitude, period, phase): [ 1.07752476 20.00673177
0.13767186]
```

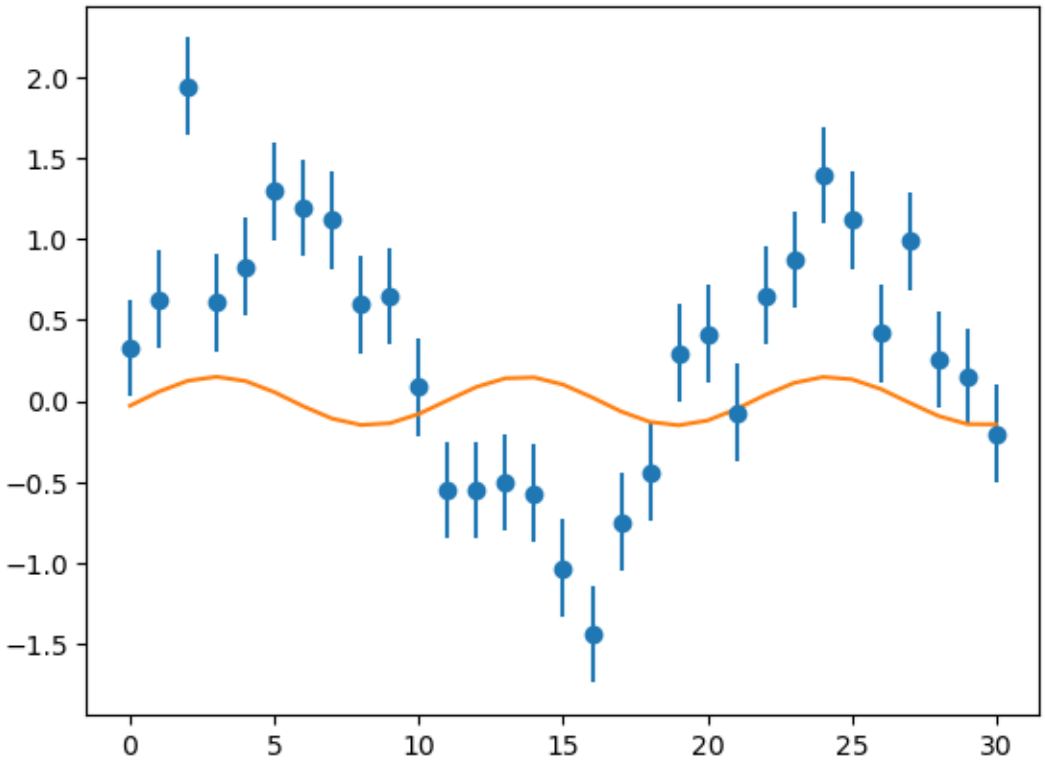


d) Now use optimize.minimize on the function you wrote in part c). Try out a few different methods with some 'bad' guesses. Do any methods produce a good fit even with a bad initial guess?

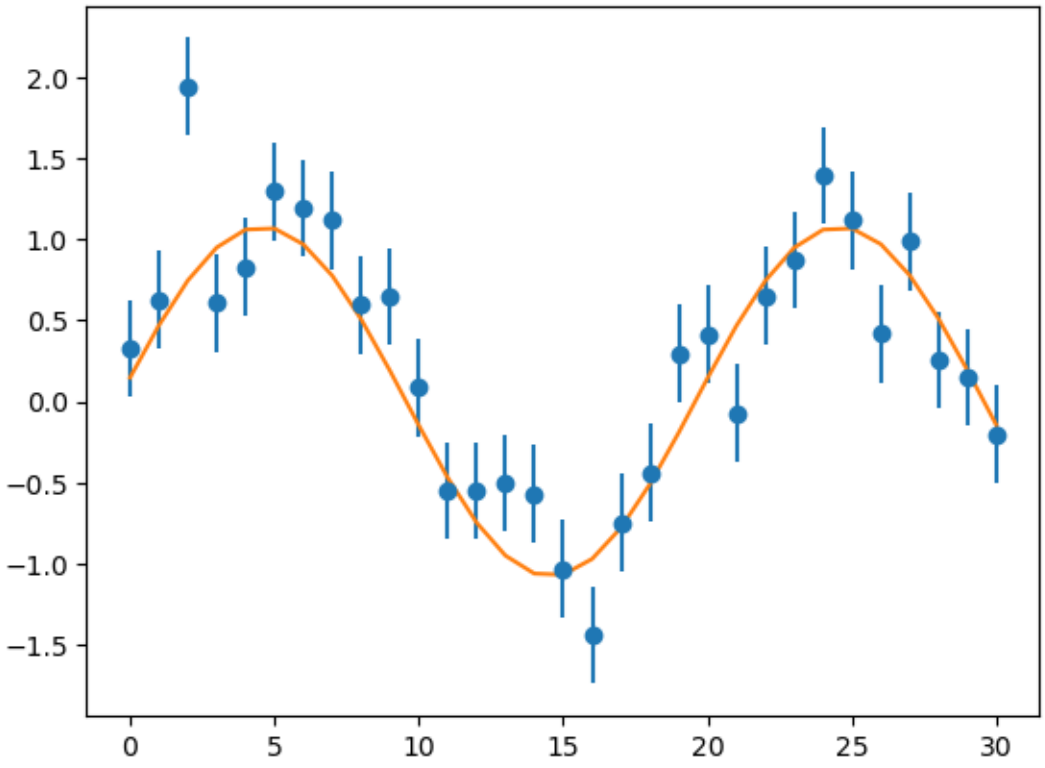
[239]:
```
bad_guess = [2., 12., 1.]

methods = ['Nelder-Mead', 'BFGS', 'Powell', 'CG']

for method in methods:
    print(f"\nUsing method: {method}")

    result = optimize.minimize(chi_squared, bad_guess, args=(x, y, sigma),␣
 ↪method=method)

    best_fit_params = result.x
    print("Best-fit parameters (amplitude, period, phase):", best_fit_params)

    fit = sin_fit(x, *best_fit_params)
```

```
    plt.errorbar(x, y, sigma, marker='o', linestyle='')
    plt.plot(x, fit, label=f"Fit using {method}")
    plt.show()
```

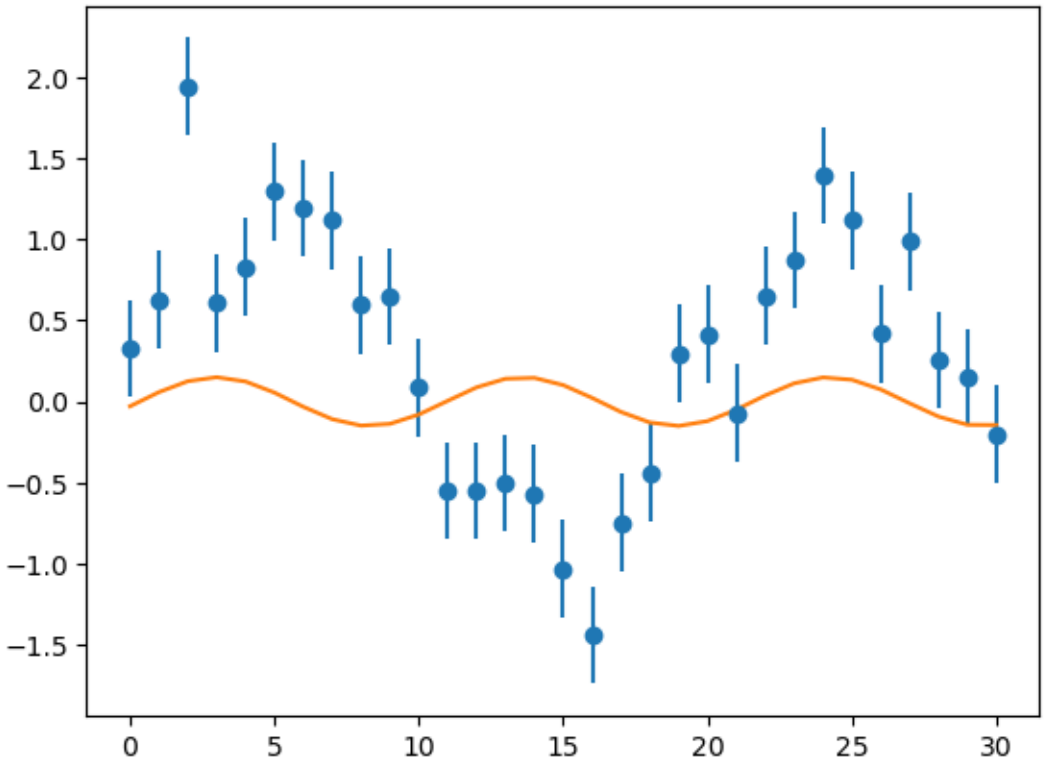
```
Using method: Nelder-Mead
Best-fit parameters (amplitude, period, phase): [ 0.14992181 10.62037996
-0.19222163]
```



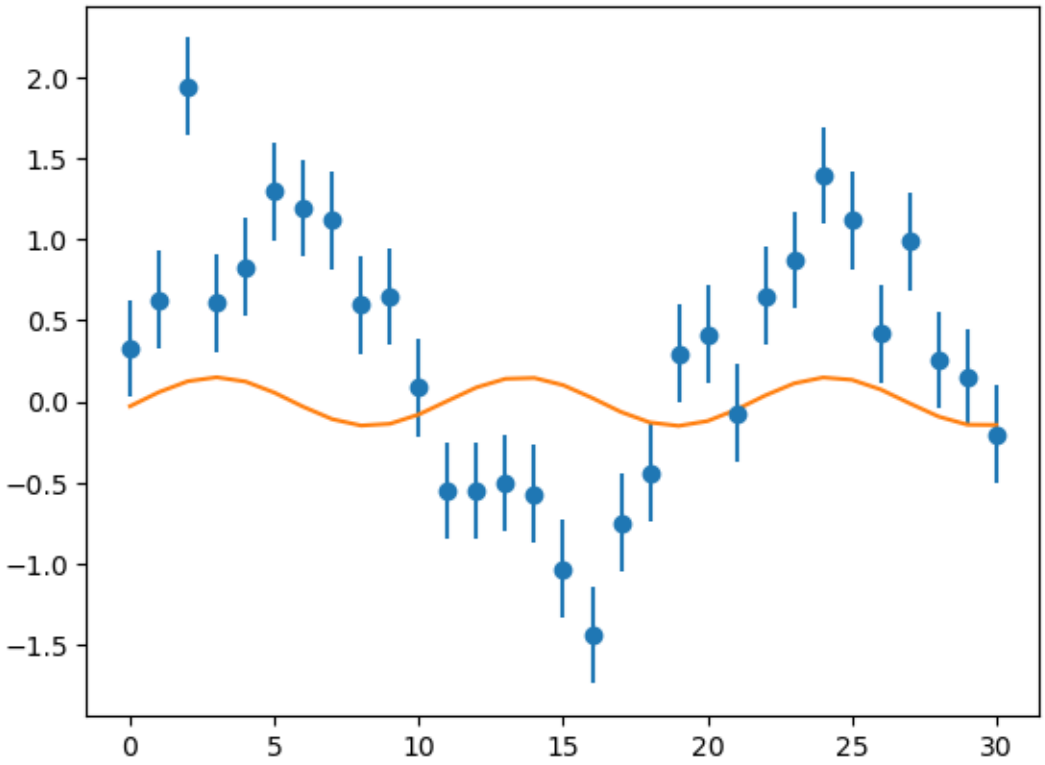

```
Using method: BFGS
Best-fit parameters (amplitude, period, phase): [  1.07752479 -20.00673195
34.41984728]
```




```
Using method: Powell
Best-fit parameters (amplitude, period, phase): [ 0.1502659  10.61960402
-0.19373888]
```




```
Using method: CG
Best-fit parameters (amplitude, period, phase): [ 0.14991649 10.6203737
-0.19221699]
```



The BFGS method produces a somewhat good fit, even with a bad inital guess.

0.4 3) Curve Fitting

You have some substance you suspect is radioactive. With a Geiger counter, you record the following counts in 1 second intervals spaced 1 minute apart:

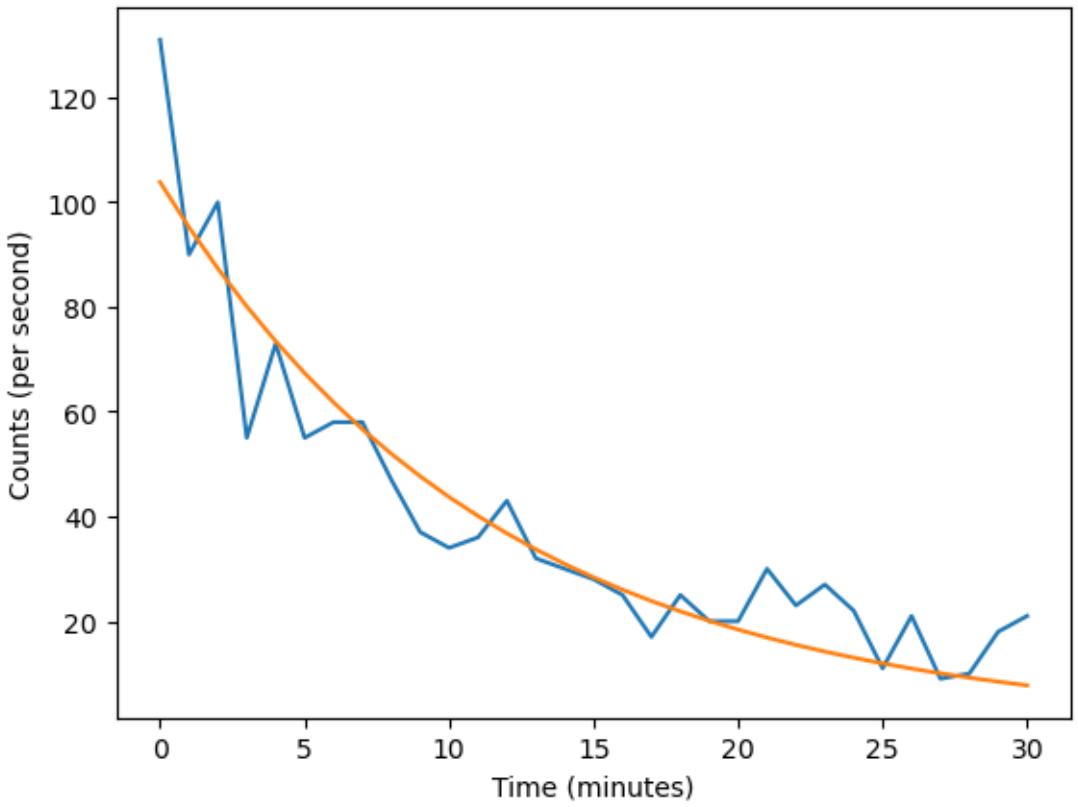
```
time (minutes):
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23

counts (per second):
[131  90 100  55  73  55  58  58  47  37  34  36  43  32  30  28  25  17  25  20  20  30  23  2
```

a) Fit a reasonable function (An expoential decay) to this data, using curve_fit. Assume the error (sigma) on each measurement is 1 count) What is the decay constant of your substance? What are the errors on this parameter?

[244]:
```
time = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,␣
 ↪18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
counts = np.array([131, 90, 100, 55, 73, 55, 58, 58, 47, 37, 34, 36, 43, 32,␣
 ↪30, 28, 25, 17, 25, 20, 20, 30, 23, 27, 22, 11, 21, 9, 10, 18, 21])
plt.plot(time, counts)
```

```python
def exp_dec(t, A, k):
    return A * np.exp(-k * t)

sigma_ones = np.ones(len(counts))
initial_guesses = [130, 0.1]
params_ones, covariance_ones = optimize.curve_fit(exp_dec, time, counts,␣
 ↪p0=initial_guesses, sigma=sigma_ones)

A_ones, k_ones = params_ones
errors_ones = np.sqrt(np.diag(covariance_ones))
A_error_ones, k_error_ones = errors_ones

print(f"Decay constant (k): {k_ones:.4f} ± {k_error_ones:.4f}")
print(f"A: {A_ones:.4f} ± {A_error_ones:.4f}")

plt.plot(time, exp_dec(time, *params_ones))
plt.xlabel('Time (minutes)')
plt.ylabel('Counts (per second)')
plt.show()
```

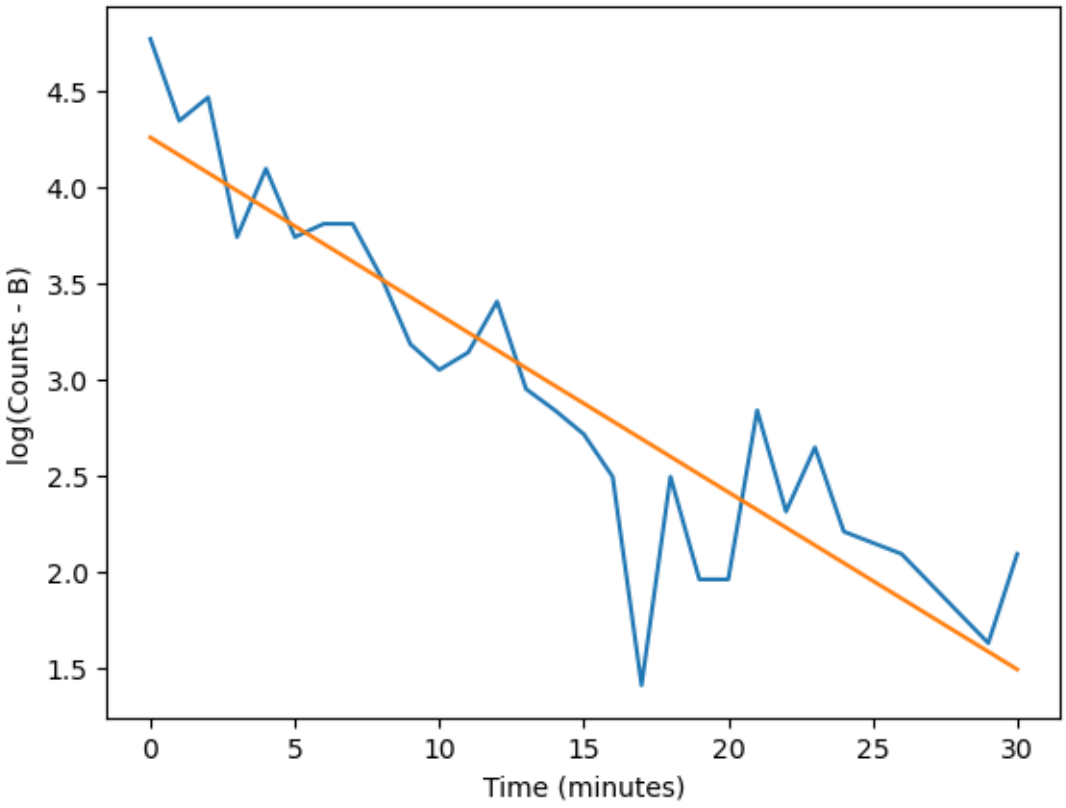
```
Decay constant (k): 0.0865 ± 0.0072
A: 103.8196 ± 5.5580
```



b) You should also be able to fit this data by making a change of coordinates, and then fitting a line to the new data. Do this. What decay constant do you get, and what are the errors? Is this consistent with your fit from part a)?

[247]:
```
def linear_func(t, m, c):
    return m * t + c

B = params[2]
adjusted_counts = counts - B
adjusted_counts[adjusted_counts <= 0] = np.nan
log_counts = np.log(adjusted_counts)

valid_indices = ~np.isnan(log_counts)
time_valid = time[valid_indices]
log_counts_valid = log_counts[valid_indices]

linear_params, linear_covariance = optimize.curve_fit(linear_func, time_valid,␣
 ↪log_counts_valid)

m, c = linear_params
m_error, c_error = np.sqrt(np.diag(linear_covariance))

k_linear = -m
k_linear_error = m_error

print(f"Decay constant from linear fit (k): {k_linear:.4f} ± {k_linear_error:.
 ↪4f}")

plt.plot(time_valid, log_counts_valid)
plt.plot(time_valid, linear_func(time_valid, *linear_params))
plt.xlabel('Time (minutes)')
plt.ylabel('log(Counts - B)')
plt.show()
```

```
Decay constant from linear fit (k): 0.0923 ± 0.0088
```



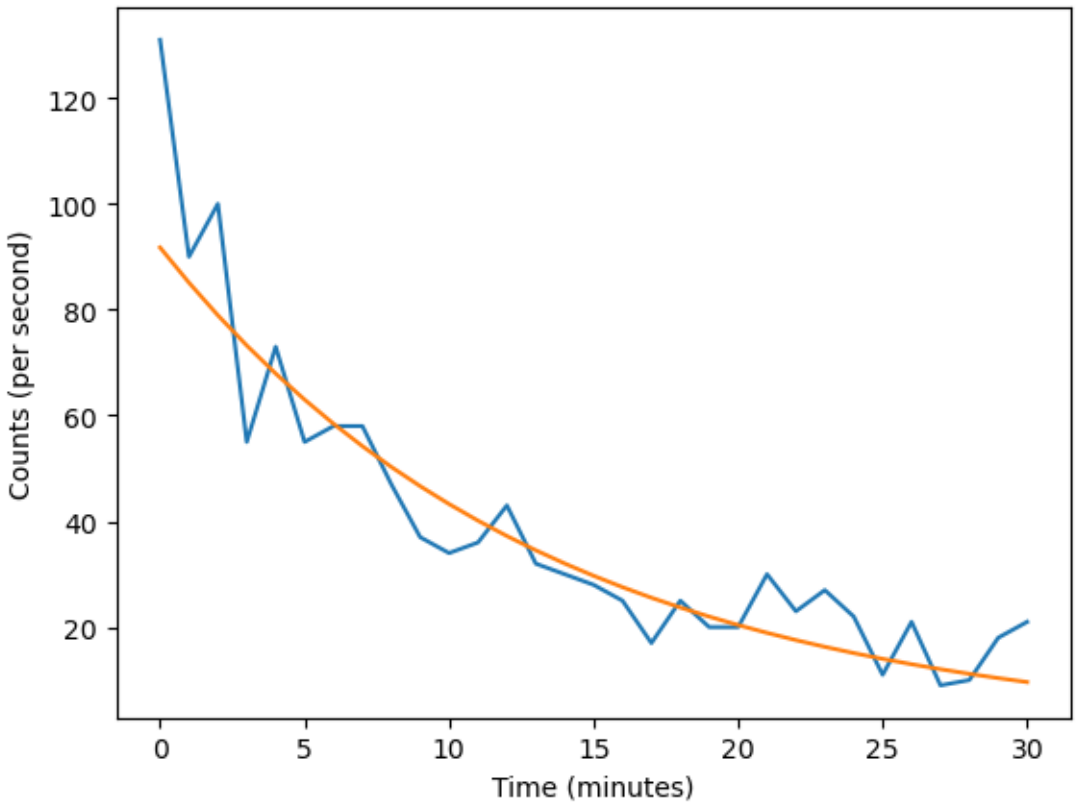
I would say this is consistent with the fit from part a.

c) Really, the count data will have poisson errors. Assuming the counts were measured in a single 1-second time interval, the error (sigma) will be the square root of the number of counts. Using this, try using curve_fit again and see what you get for the decay constant ant error on this parameter.

[251]:
```
plt.plot(time, counts)
sigma_sqrt = np.sqrt(counts)

params_sqrt, covariance_sqrt = optimize.curve_fit(exp_dec, time, counts,␣
 ↪p0=initial_guesses, sigma=sigma_sqrt)

A_sqrt, k_sqrt = params_sqrt
errors_sqrt = np.sqrt(np.diag(covariance_sqrt))
A_error_sqrt, k_error_sqrt = errors_sqrt

print(f"Decay constant (k): {k_sqrt:.4f} ± {k_error_sqrt:.4f}")
print(f"A: {A_sqrt:.4f} ± {A_error_sqrt:.4f}")

plt.plot(time, exp_dec(time, *params_sqrt))
plt.xlabel('Time (minutes)')
```

```
plt.ylabel('Counts (per second)')
plt.show()
```

```
Decay constant (k): 0.0752 ± 0.0058
A: 91.7488 ± 6.4403
```

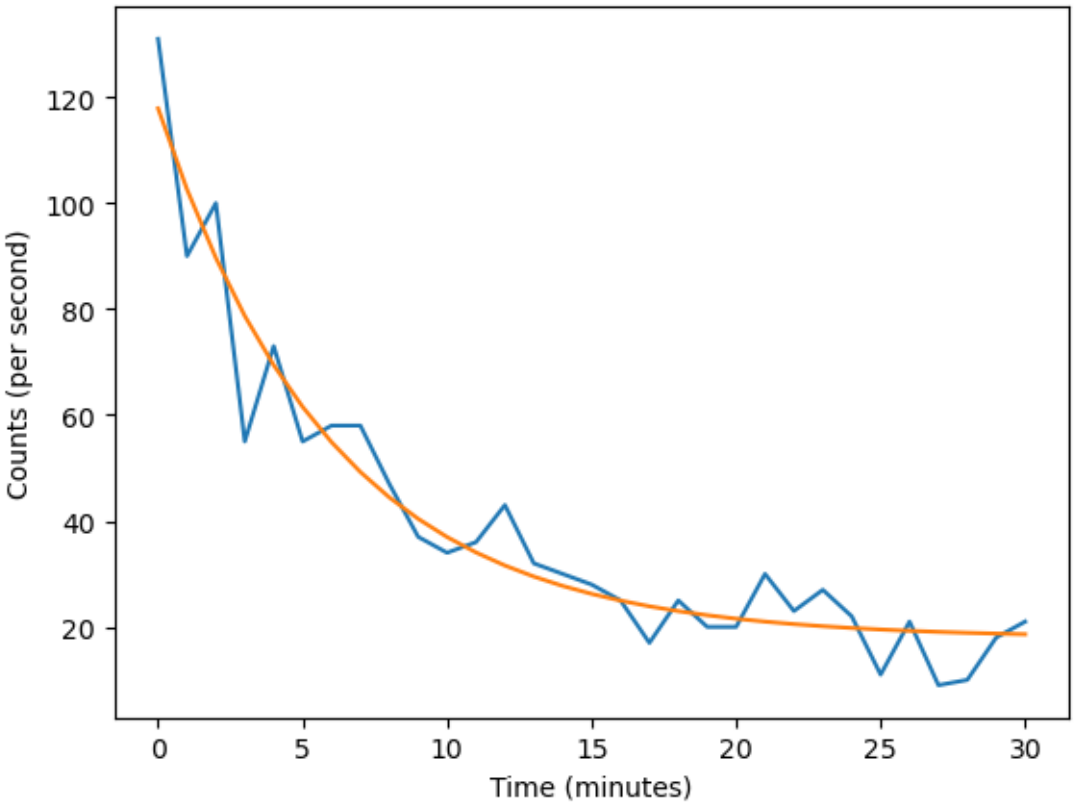


d) To make you fit better, keep in mind there are probably some background counts as well (from cosmic rays, electronic noise in your instrument, etc.). Try fitting again using and exponential decay plus a constant (3 parameters) with and without assuming Poisson noise. How do these results compare to the values in part a) and c) ?

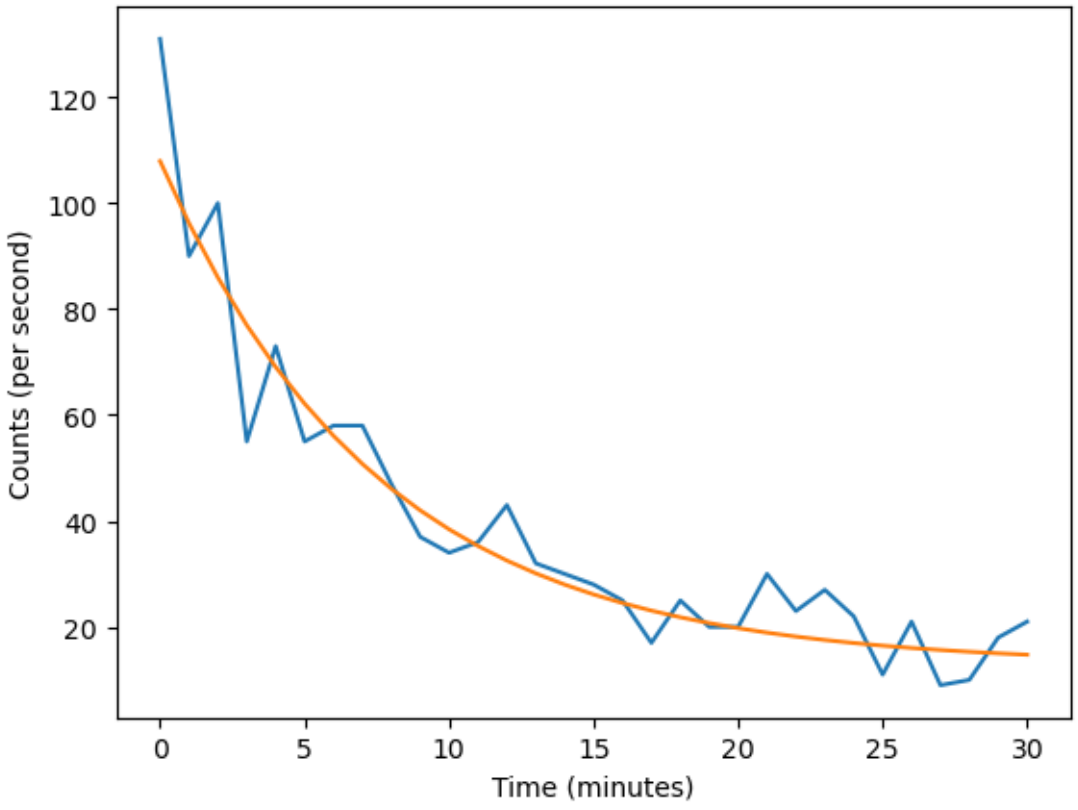
[254]:
```
plt.plot(time, counts)

def exp_dec(t, A, k, B):
    return A * np.exp(-k * t) + B

sigma_ones = np.ones(len(counts))
initial_guesses = [130, 0.1, 10]
params_ones, covariance_ones = optimize.curve_fit(exp_dec, time, counts,␣
 ↪p0=initial_guesses, sigma=sigma_ones)

A_ones, k_ones, b_ones = params_ones
```

```
errors_ones = np.sqrt(np.diag(covariance_ones))
A_error_ones, k_error_ones, B_error_ones = errors_ones

print(f"Decay constant (k): {k_ones:.4f} ± {k_error_ones:.4f}")
print(f"A: {A_ones:.4f} ± {A_error_ones:.4f}")
print(f"B: {B:.4f} ± {B_error_ones:.4f}")

plt.plot(time, exp_dec(time, *params_ones))
plt.xlabel('Time (minutes)')
plt.ylabel('Counts (per second)')
plt.show()

sigma_sqrt = np.sqrt(counts)

params_sqrt, covariance_sqrt = optimize.curve_fit(exp_dec, time, counts,␣
 ↪p0=initial_guesses, sigma=sigma_sqrt)

A_sqrt, k_sqrt, B_sqrt = params_sqrt
errors_sqrt = np.sqrt(np.diag(covariance_sqrt))
A_error_sqrt, k_error_sqrt, B_error_sqrt = errors_sqrt

print(f"Decay constant (k): {k_sqrt:.4f} ± {k_error_sqrt:.4f}")
print(f"A: {A_sqrt:.4f} ± {A_error_sqrt:.4f}")
print(f"B: {B:.4f} ± {B_error_sqrt:.4f}")

plt.plot(time, counts)
plt.plot(time, exp_dec(time, *params_sqrt))
plt.xlabel('Time (minutes)')
plt.ylabel('Counts (per second)')
plt.show()
```

```
Decay constant (k): 0.1658 ± 0.0212
A: 99.9448 ± 5.6422
B: 12.9173 ± 2.7383
```




```
Decay constant (k): 0.1315 ± 0.0201
A: 95.0064 ± 7.6301
B: 12.9173 ± 2.7007
```



Including a background constant and assuming Poisson noise gives a higher decay constant and larger uncertainty compared to using a constant error, thus it's a better model for radioactive decay.

e) Extra cradit: THe error on eahc point isn't really just the square root of the number of counts, it will be the square root of the count there should be for the fit at that point. Fit the data with a function that used sigma(x) = np.sqrt(np.abs(fit(x))). Also include the background, like in part d). You'll probably have to use something like optimize.minimize rather than curve_fit. How do these results compare to part d?

[ ]: