PHYS689
Homework 5
Will Wainwright
Repository: https://github.com/wjwainwright/PHYS689

Discussion

The procedure that I used involved generating data with a source function, applying noise to that data, and then using the normal method to fit the data. To generate the data, I used a generic polynomial function that can take any number of coefficients. For the most simple case, I used a linear model with $a_1 = 5$ and $a_0 = 7$. I generated random, Gaussian noise with a mean of zero and a standard deviation of one. I added this noise to the signal and then calculated the signal to noise ratio at the last point. With the target SNR being 3, I then scaled the noise so that the new SNR at the last point was 3, and added it to the signal. I used the same process to generate the data set for the second part of the homework, using a different function.

With the data set made and noise added, I first tested the fit using scipy's curve fit method. I then wrote my own normal method function which takes arrays of x values and the "measured" signal with noise at those points. The function also takes a list of uncertainties for the points, which in my case I took to be uniform. From my understanding of the normal method, I then had to generate the design matrix whose elements were essentially the terms of the equation I was trying to fit but without coefficients. While I could have generated a simple linear design matrix, I choose to adopt the extra challenge of making it work for a variable number of polynomial terms. While I have not attached any plots of the higher order terms being fit, I tested up to a 4th degree polynomial and got pretty good results from the fit. Once the design matrix is constructed, I just perform a few matrix operations as shown in lecture, and then return the fit parameters and variance matrix. The diagonal elements of the variance matrix are the individual variances of the fit parameters. Comparing the fit from my normal method to that of scipy, both fits return the same fit parameters, but my normal method reports a much lower error, which has me skeptical. I think I may have populated the noise matrix incorrectly, but I'm not sure how else I would do it.

For part two, in the case of a nonlinear function, I first experimented with exponential functions. I wanted to do something of the form of $a_i e^{x^i}$, but eventually I was restricted because the values were quickly overflowing. I ultimately settled on a function of the form $y = a_2x^2 + a_1x + a_0 + a_{-1}x^{-1} + a_{-2}x^{-2}$, with a variable number of terms being accepted, similar to my regular polynomial case. Creating the design matrix was the hardest part about using this function, as I had to map from a loop through the matrix to a loop from $-n$ to n. I also made some fancy strings for displaying the values as plot labels. One thing I noticed through both parts is that the spread on the data points, and therefore the goodness of fit, is highly dependent on whether the last data point is generated with a high or low random noise. Because the signal to noise is fixed by the last point, the scaling applied the the noise array as a whole depends on this dice roll.

Altogether, I had a lot of fun with this assignment, and learned some new things along the way, both about functions and fitting as well as about python. One thing I was unfortunately not able to accomplish was finding the probability to exceed for the given fits. I was able to calculate the χ^2 values for the fits, but none of the packages I could find would take a pre-calculated χ^2 and return a p-value. That being said, I'm still pleased with what I accomplished and impressed with how

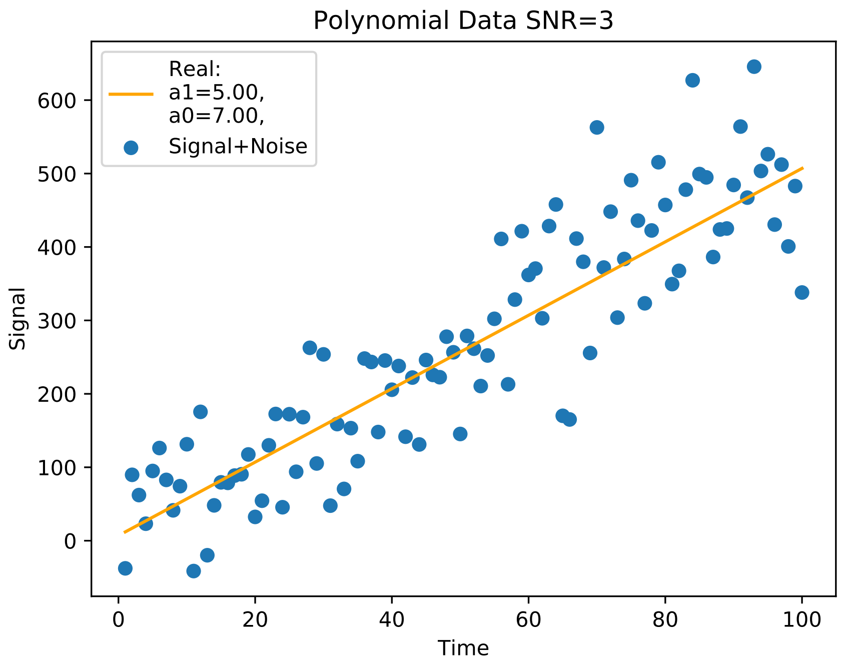

Figure 1: Polynomial, in this case linear with $a_1 = 5$ and $a_0 = 7$.

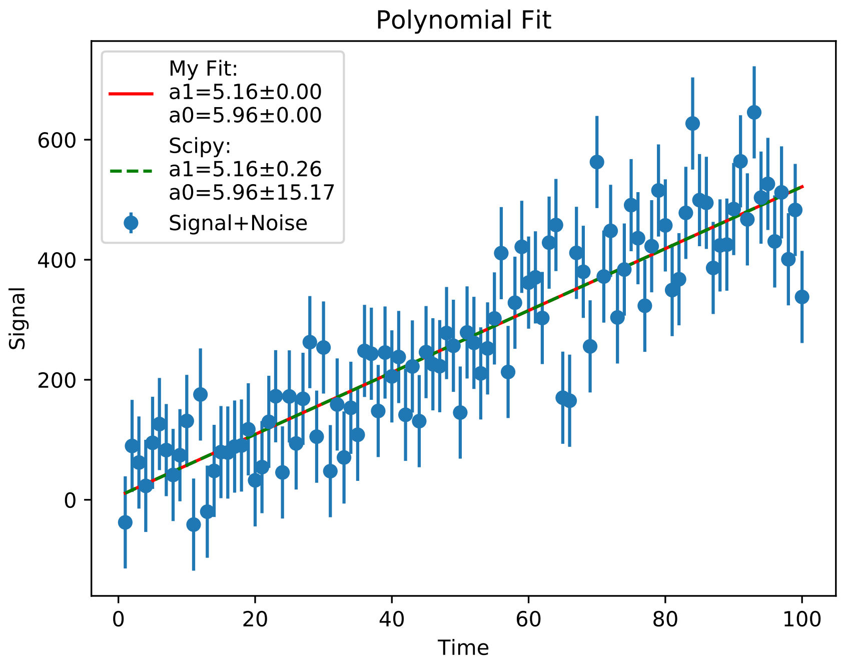

Figure 2: Polynomial fit comparing the normal method to scipy.

effective a few matrix operations are at fitting a data set. The hardest part seems to be effectively constructing the design matrix.

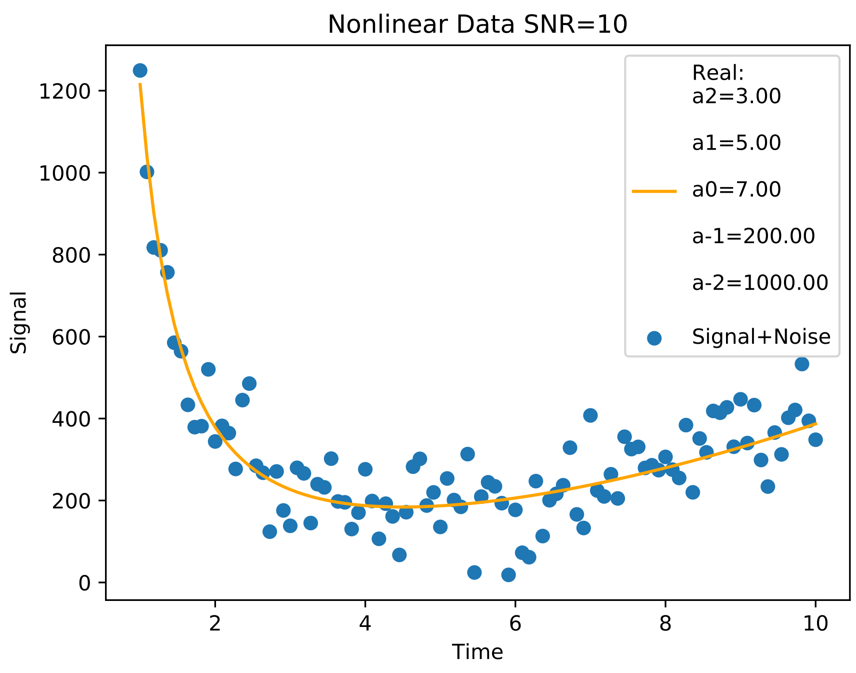

Figure 3: Nonlinear method following $y = a_2x^2 + a_1x + a_0 + a_{-1}x^{-1} + a_{-2}x^{-2}$.

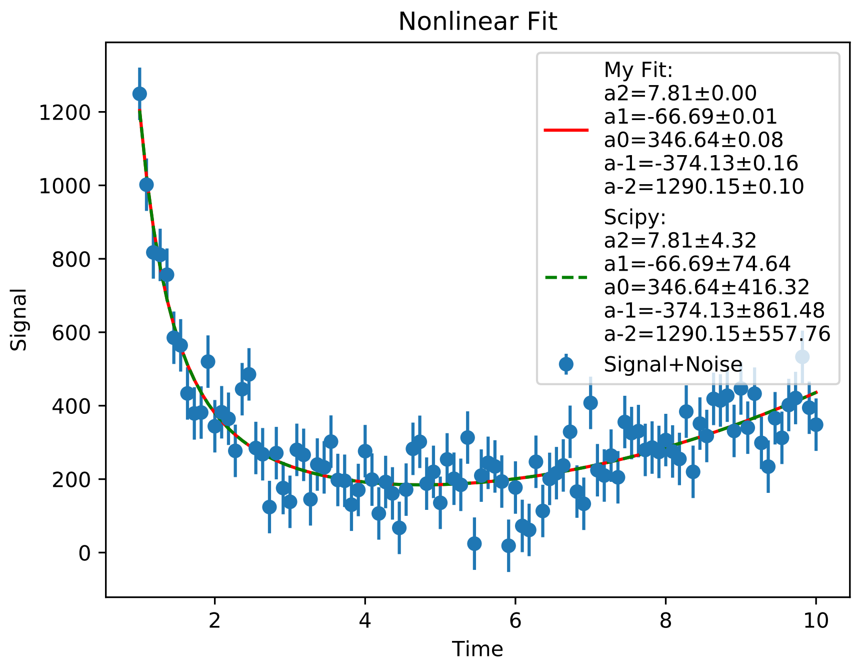

Figure 4: Nonlinear fit comparing the normal method to scipy.

```python
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as so


def normPoly(x,y,N,nparam):
    global noise
    noise = np.zeros((len(N),len(N)))
    for i in range(len(N)):
        noise[i,i] = N[i]**2

    A = np.zeros((len(x),nparam))
    for i in range(len(x)):
        for j in range(nparam):
            A[i,j] = x[i]**(nparam-1-j)

    ATA = (np.transpose(A) @ noise) @ A
    ATAinv = np.linalg.inv(ATA)
    ATb = np.dot((np.transpose(A) @ noise),y)
    params = np.dot(ATAinv,ATb)

    return params,ATAinv


def norm2(x,y,N,order):
    order = int(order)
    noise = np.zeros((len(N),len(N)))
    for i in range(len(N)):
        noise[i,i] = N[i]**2

    A = np.zeros((len(x),2*order+1))
    for i in range(len(x)):
        for j in range(-order,order+1):
            A[i,-1*j+order] = x[i]**j
    ATA = (np.transpose(A) @ noise) @ A
    ATAinv = np.linalg.inv(ATA)
    ATb = np.dot((np.transpose(A) @ noise),y)
    params = np.dot(ATAinv,ATb)

    return params,ATAinv






"""
Part 1
Generic polynomial function
y = a1*x + a0 for example
"""


#Defining variables
ai = [5,7]
```

```
nparams = len(ai)
n=100
stdev=1
noise = np.random.normal(0,stdev,n)



#Polynomial function
def f1(x,*params):
    l = []
    N = len(params)
    for i,a in enumerate(params):
        l.append(a*x**(N-i-1))
    return sum(l)

x = np.linspace(1,100,100)
y = [f1(a,*ai) for a in x]


#Fix the noise to have a SNR of 3 at the last point
ratio = abs(y[-1]/noise[-1])
print('Part 1: Linear Case')
print(f'Old SNR: {ratio}')
scaling = 3/ratio

noise = [a/scaling for a in noise]

#Signal + Noise
mixed = [a+b for a,b in zip(y,noise)]


print(f'New SNR: {abs(y[-1]/noise[-1])}')

uncertainty = [stdev/scaling]*n

#Fitting with my method and with scipy
canned,cvar = so.curve_fit(f1,x,mixed,p0=[1]*nparams,sigma=uncertainty)
coeff,var = normPoly(x,mixed,uncertainty,nparams)
fit = [f1(a,*coeff) for a in x]
cfit = [f1(a,*canned) for a in x]


fstr = "My Fit: "
cstr = "Scipy: "
rstr = "Real: "

#Label strings of variable lengths
for i in range(nparams-1,-1,-1):
    fstr += f"\na{i}={coeff[nparams-1-i]:.2f}±{np.sqrt(var[nparams-1-i,nparams-1-i]):.2f}"
    cstr += f"\na{i}={canned[nparams-1-i]:.2f}±{np.sqrt(cvar[nparams-1-i,nparams-1-i]):.2f
    rstr += f"\na{i}={ai[nparams-1-i]:.2f}, "



#Plot of the data
plt.figure()
plt.scatter(x,mixed,label='Signal+Noise')
```

```python
plt.plot(x,y,color='orange',label=rstr)
plt.title('Polynomial Data SNR=3')
plt.xlabel('Time')
plt.ylabel('Signal')
plt.legend(loc='upper left')
plt.savefig('polynomial.pdf')


#Plot of the fit
plt.figure()
plt.errorbar(x,mixed,uncertainty,fmt='o',label='Signal+Noise')
plt.plot(x,fit,color='red',label=fstr)
plt.plot(x,cfit,'--g',label=cstr)
plt.title('Polynomial Fit')
plt.xlabel('Time')
plt.ylabel('Signal')
plt.legend(loc='upper left')
plt.savefig('polynomialFit.pdf')

#Chi-squared
chi = []
for i in range(len(x)):
    chi.append( (mixed[i]-fit[i])**2/(2*uncertainty[i]**2) )
chi2 = sum(chi)
print(f"Chi-squared: {chi2}")



"""
Part 2
Nonlinear function
y = a2*x + a1 + a0/x for example
"""

ai = [3,5,7,200,1000]
order = int((len(ai)-1)/2)
n=100
stdev=1
noise = np.random.normal(0,stdev,n)

def f2(x,*params):
    l = []
    N = len(params)
    order = int((N-1)/2)
    for i in range(order,-1*order-1,-1):
        j = -1*i + order
        l.append(params[j]*x**i)
    return sum(l)


x = np.linspace(1,10,100)
y = [f2(a,*ai) for a in x]



#Fix the noise to have a SNR of 3 at the last point
ratio = abs(y[-1]/noise[-1])
```

```
print('Part 2: Nonlinear Case')
print(f'Old SNR: {ratio}')
scaling = 10/ratio

noise = [a/scaling for a in noise]

#Signal + Noise
mixed = [a+b for a,b in zip(y,noise)]


print(f'New SNR: {abs(y[-1]/noise[-1])}')

uncertainty = [stdev/scaling]*n

#Fitting with my method and with scipy
canned,cvar = so.curve_fit(f2,x,mixed,p0=[1]*len(ai),sigma=uncertainty)
coeff,var = norm2(x,mixed,uncertainty,order)
fit = [f2(a,*coeff) for a in x]
cfit = [f2(a,*canned) for a in x]


fstr = "My Fit: "
cstr = "Scipy: "
rstr = "Real: "

#Label strings of variable lengths
for i in range(int(order),-1*int(order)-1,-1):
    j = -1*i+order
    fstr += f"\na{i}={coeff[j]:.2f}±{np.sqrt(var[j,j]):.2f}"
    cstr += f"\na{i}={canned[j]:.2f}±{np.sqrt(cvar[j,j]):.2f}"
    rstr += f"\na{i}={ai[j]:.2f}\n "


#Plot of the data
plt.figure()
plt.scatter(x,mixed,label='Signal+Noise')
plt.plot(x,y,color='orange',label=rstr)
plt.title('Nonlinear Data SNR=10')
plt.xlabel('Time')
plt.ylabel('Signal')
plt.legend(loc='upper right')
plt.savefig('nonlinear.pdf')


#Plot of the fit
plt.figure()
plt.errorbar(x,mixed,uncertainty,fmt='o',label='Signal+Noise')
plt.plot(x,fit,color='red',label=fstr)
plt.plot(x,cfit,'--g',label=cstr)
plt.title('Nonlinear Fit')
plt.xlabel('Time')
plt.ylabel('Signal')
plt.legend(loc='upper right')
plt.savefig('nonlinearFit.pdf')


#Chi-squared
```

```python
chi = []
for i in range(len(x)):
    chi.append( (mixed[i]-fit[i])**2/(2*uncertainty[i]**2) )
chi2 = sum(chi)
print(f"Chi-squared: {chi2}")
```