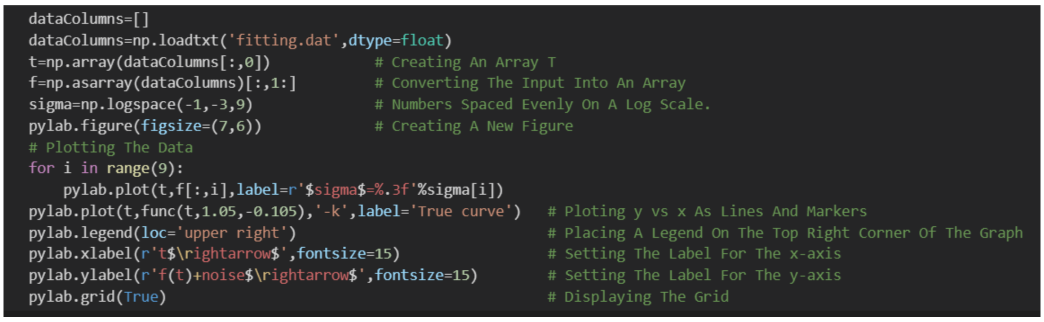
EE2703 : Applied Programming Lab
Assignment-1

Nihal Gajjala
EE19B044

February 2021

Extracting And Visualizing The Data

Extracting the data obtained by running the python script *generate.py* whose output data was written into the file *fitting.dat*.



```
dataColumns=[]
dataColumns=np.loadtxt('fitting.dat',dtype=float)
t=np.array(dataColumns[:,0])            # Creating An Array T
f=np.asarray(dataColumns)[:,1:]         # Converting The Input Into An Array
sigma=np.logspace(-1,-3,9)              # Numbers Spaced Evenly On A Log Scale.
pylab.figure(figsize=(7,6))             # Creating A New Figure
# Plotting The Data
for i in range(9):
    pylab.plot(t,f[:,i],label=r'$sigma$=%.3f'%sigma[i])
pylab.plot(t,func(t,1.05,-0.105),'-k',label='True curve')   # Ploting y vs x As Lines And Markers
pylab.legend(loc='upper right')                             # Placing A Legend On The Top Right Corner Of The Graph
pylab.xlabel(r't$\rightarrow$',fontsize=15)                 # Setting The Label For The x-axis
pylab.ylabel(r'f(t)+noise$\rightarrow$',fontsize=15)        # Setting The Label For The y-axis
pylab.grid(True)                                            # Displaying The Grid
```

Figure 1: generate.py

The data obtained has 10 columns. The first column is time and the remaining nine columns riddled with different amount of noise, with standard deviation uniformly sampled on a logarithmic scale.

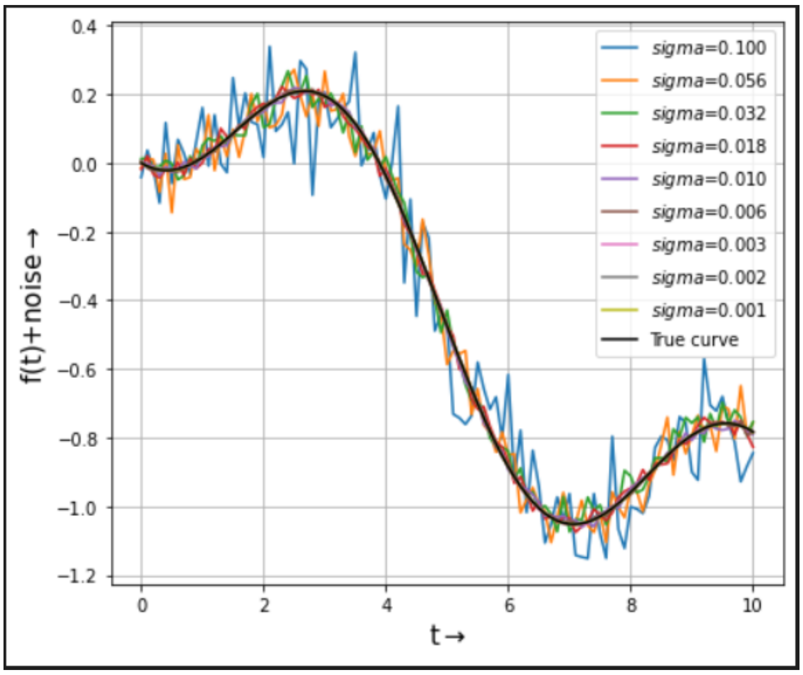


Figure 2: Signal With Noise

Functions

The data columns correspond to the function

$$f(t) = 1.05J_2(t) - 0.105t + n(t) \tag{1}$$

with different amounts of noise added. The noise is given to be normally distributed, i.e., its probability distribution is given by

$$Pr(n(t)|\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\frac{-n(t)^2}{2\sigma^2} \tag{2}$$

where sigma = logspace(-1,-3,9)

Errorbar

Errorbar is a convenient way of visualizing the uncertainty in the measurement. The errorbar for the first data column ($\sigma = 0.1$) is plotted in the below figure.

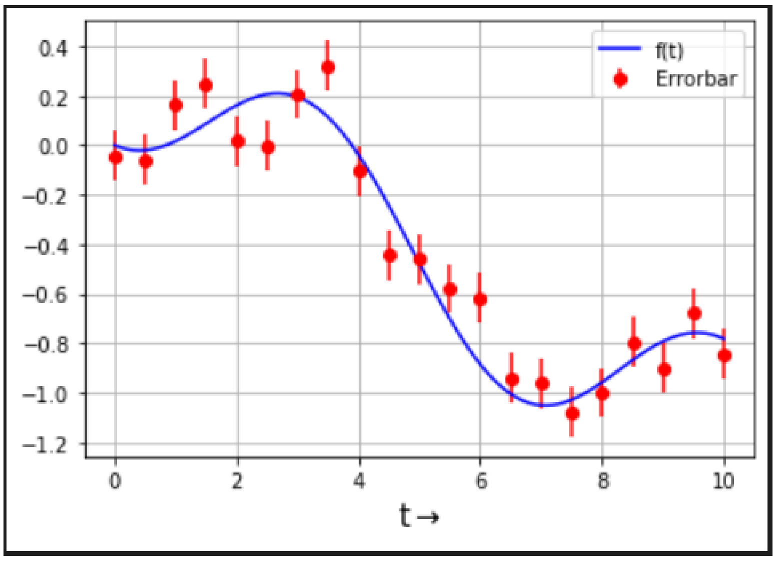
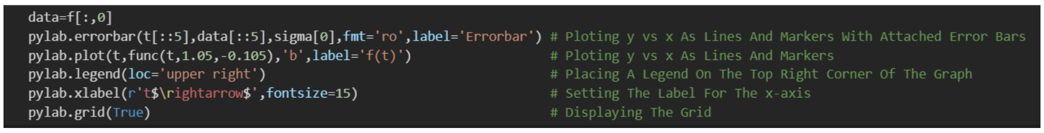


Figure 3: Errorbar

Every 5th data point is plotted to make the data readable.



```
data=f[:,0]
pylab.errorbar(t[::5],data[::5],sigma[0],fmt='ro',label='Errorbar') # Ploting y vs x As Lines And Markers With Attached Error Bars
pylab.plot(t,func(t,1.05,-0.105),'b',label='f(t)')                  # Ploting y vs x As Lines And Markers
pylab.legend(loc='upper right')                                     # Placing A Legend On The Top Right Corner Of The Graph
pylab.xlabel(r't$\rightarrow$',fontsize=15)                         # Setting The Label For The x-axis
pylab.grid(True)                                                    # Displaying The Grid
```

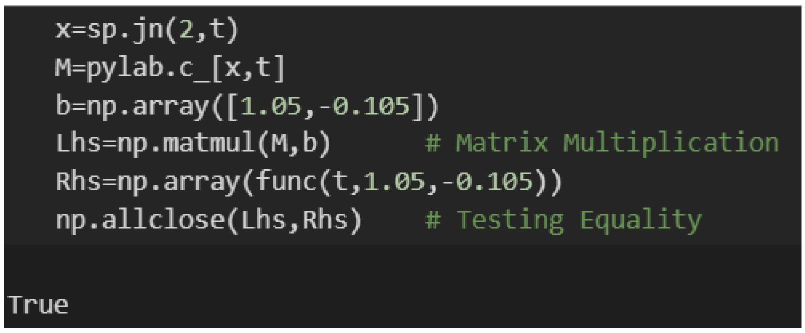
Figure 4: Python Code For Errorbar

Each point in the data column can be seen to be varying within a σ width of the true value (except in extreme cases).

Matrices

Comparing the results obtained from matrix multiplication and user-defined function.

$$g(t, A, B) = \begin{pmatrix} J_2(t_1) & t_1 \\ \dots & \dots \\ J_2(t_m) & t_m \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = M.p \tag{3}$$



```
x=sp.jn(2,t)
M=pylab.c_[x,t]
b=np.array([1.05,-0.105])
Lhs=np.matmul(M,b)       # Matrix Multiplication
Rhs=np.array(func(t,1.05,-0.105))
np.allclose(Lhs,Rhs)     # Testing Equality

True
```

Figure 5: Matrix Check

Error Computation

The errors in each data column is computed and stored in an array *error*. The error is given by

$$\epsilon_{i,j} = \frac{1}{101} \sum_{k=0}^{101} (f_k - g(t_k, A_i, B_j)^2 \tag{4}$$

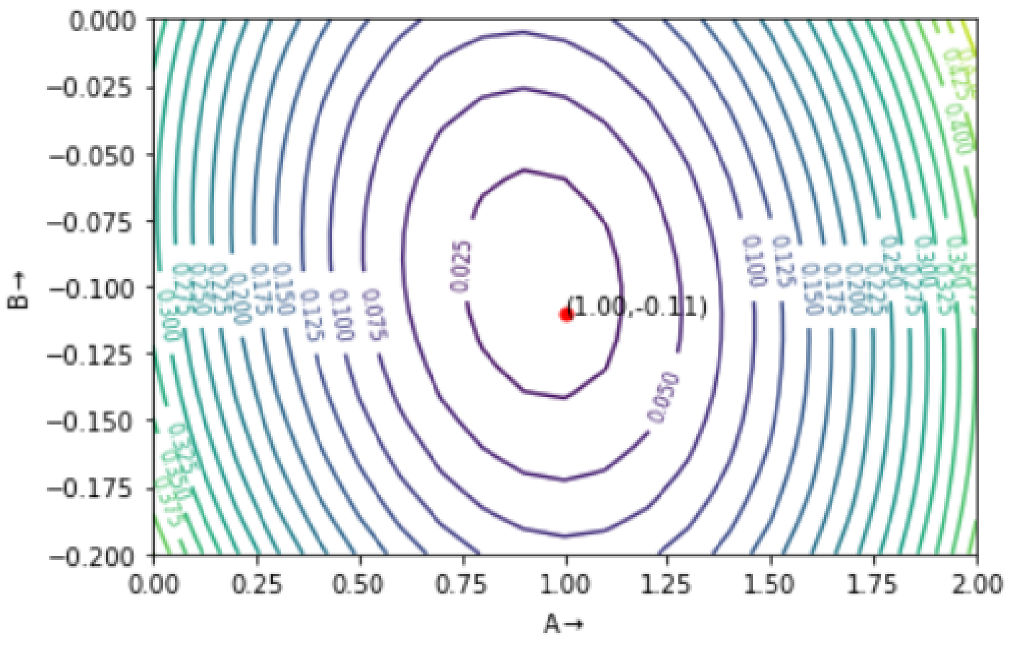
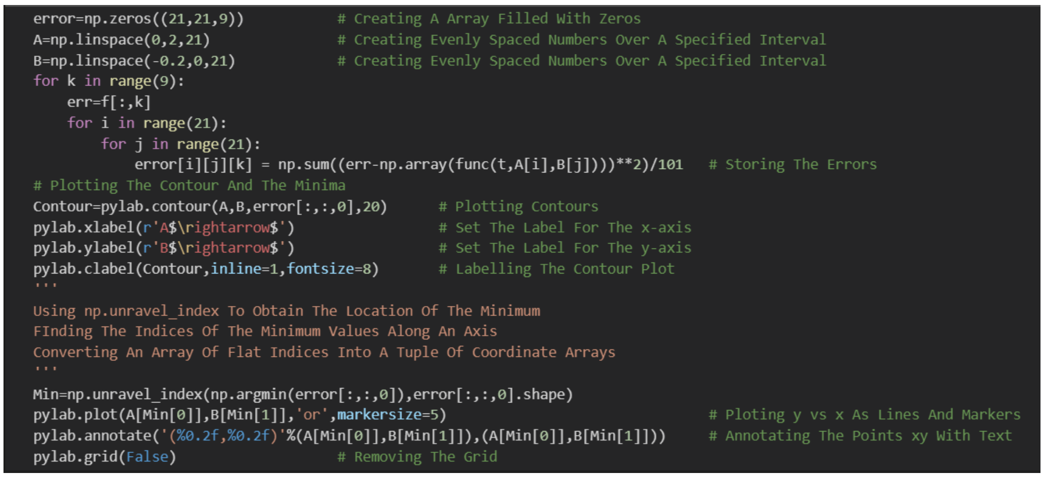


Figure 6: Contour Plot

The above graph represents the plot of the mean squared error for the first data column. The red dot represents the minima which is used to determine the best approximate of A and B. The above graph has only one minima at 1.1 and -0.1 i.e A=1.1 and B=-0.1

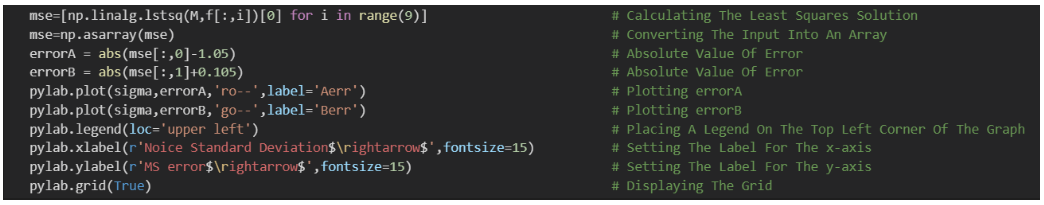


```
error=np.zeros((21,21,9))          # Creating A Array Filled With Zeros
A=np.linspace(0,2,21)              # Creating Evenly Spaced Numbers Over A Specified Interval
B=np.linspace(-0.2,0,21)           # Creating Evenly Spaced Numbers Over A Specified Interval
for k in range(9):
    err=f[:,k]
    for i in range(21):
        for j in range(21):
            error[i][j][k] = np.sum((err-np.array(func(t,A[i],B[j])))**2)/101   # Storing The Errors
# Plotting The Contour And The Minima
Contour=pylab.contour(A,B,error[:,:,0],20)      # Plotting Contours
pylab.xlabel(r'A$\rightarrow$')                 # Set The Label For The x-axis
pylab.ylabel(r'B$\rightarrow$')                 # Set The Label For The y-axis
pylab.clabel(Contour,inline=1,fontsize=8)       # Labelling The Contour Plot
'''
Using np.unravel_index To Obtain The Location Of The Minimum
FInding The Indices Of The Minimum Values Along An Axis
Converting An Array Of Flat Indices Into A Tuple Of Coordinate Arrays
'''
Min=np.unravel_index(np.argmin(error[:,:,0]),error[:,:,0].shape)
pylab.plot(A[Min[0]],B[Min[1]],'or',markersize=5)                      # Ploting y vs x As Lines And Markers
pylab.annotate('(%0.2f,%0.2f)'%(A[Min[0]],B[Min[1]]),(A[Min[0]],B[Min[1]]))   # Annotating The Points xy With Text
pylab.grid(False)                  # Removing The Grid
```

Figure 7: Python Code Contour Plot

Error Estimate

The errors in A and B are calculated using least squares



```
mse=[np.linalg.lstsq(M,f[:,i])[0] for i in range(9)]                # Calculating The Least Squares Solution
mse=np.asarray(mse)                                                  # Converting The Input Into An Array
errorA = abs(mse[:,0]-1.05)                                          # Absolute Value Of Error
errorB = abs(mse[:,1]+0.105)                                         # Absolute Value Of Error
pylab.plot(sigma,errorA,'ro--',label='Aerr')                         # Plotting errorA
pylab.plot(sigma,errorB,'go--',label='Berr')                         # Plotting errorB
pylab.legend(loc='upper left')                                       # Placing A Legend On The Top Left Corner Of The Graph
pylab.xlabel(r'Noice Standard Deviation$\rightarrow$',fontsize=15)   # Setting The Label For The x-axis
pylab.ylabel(r'MS error$\rightarrow$',fontsize=15)                   # Setting The Label For The y-axis
pylab.grid(True)                                                     # Displaying The Grid
```

Figure 8: Python Code For MS error vs Noise Standard deviation

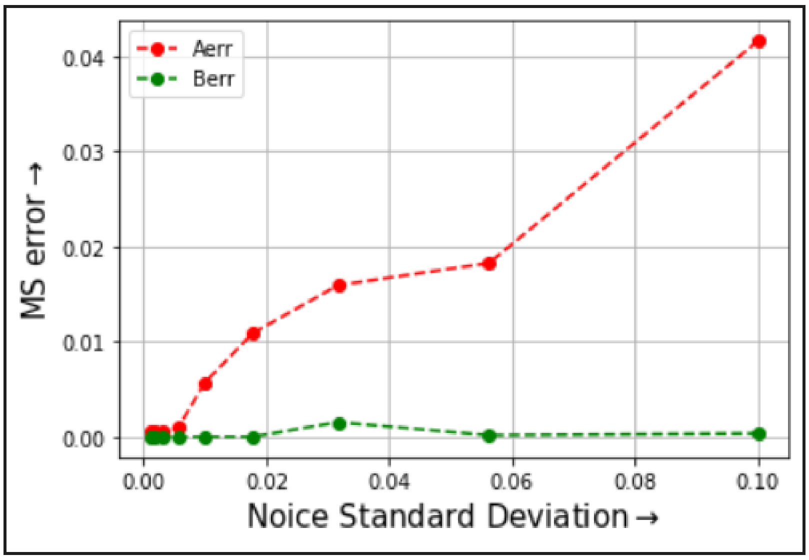
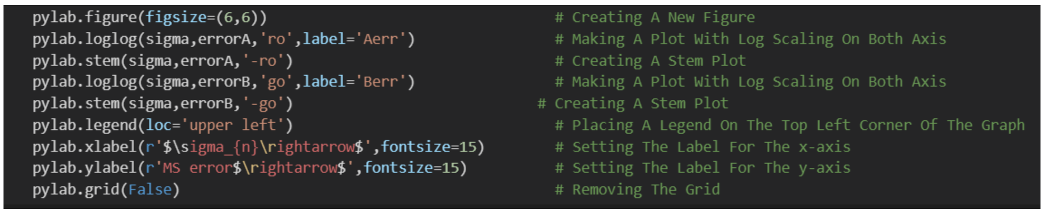


Figure 9: MS error vs Noise Standard deviation

The error estimate is non-linear. Plotting the same graph in logarithmic scale gives a comparatively linear plot compared to the above graph.



```
pylab.figure(figsize=(6,6))                                  # Creating A New Figure
pylab.loglog(sigma,errorA,'ro',label='Aerr')                 # Making A Plot With Log Scaling On Both Axis
pylab.stem(sigma,errorA,'-ro')                               # Creating A Stem Plot
pylab.loglog(sigma,errorB,'go',label='Berr')                 # Making A Plot With Log Scaling On Both Axis
pylab.stem(sigma,errorB,'-go')                             # Creating A Stem Plot
pylab.legend(loc='upper left')                               # Placing A Legend On The Top Left Corner Of The Graph
pylab.xlabel(r'$\sigma_{n}\rightarrow$',fontsize=15)         # Setting The Label For The x-axis
pylab.ylabel(r'MS error$\rightarrow$',fontsize=15)           # Setting The Label For The y-axis
pylab.grid(False)                                            # Removing The Grid
```

Figure 10: Python Code For MS error vs σ_n

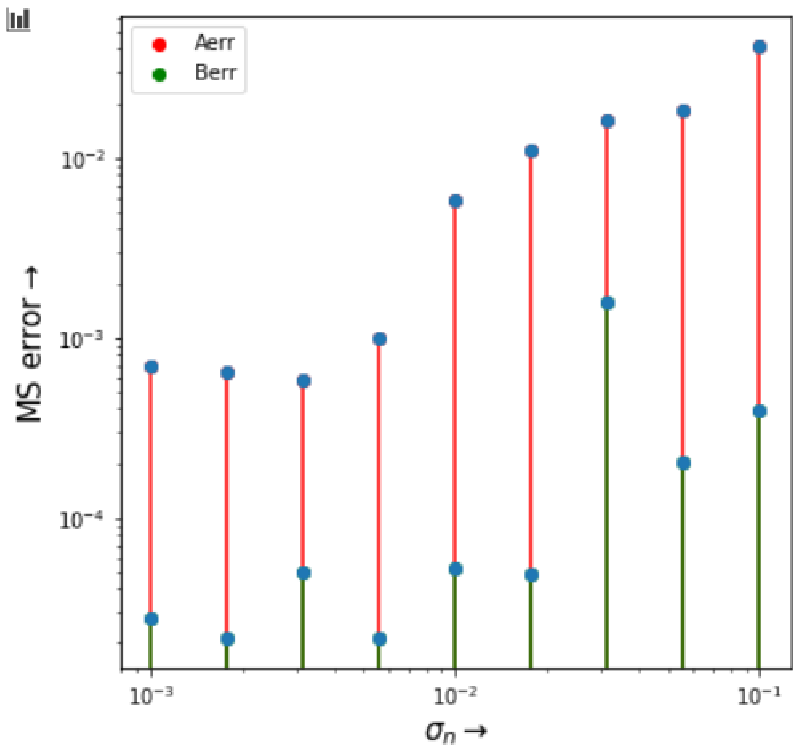


Figure 11: MS error vs σ_n

Conclusion

The noisy data was extracted and the best approximate was determined by minimizing the mean squared error. It was observed that the error was approximately linear with respect to σ_n in the logarithmic scale.