# Problem Set #2

Observational Techniques of Modern Astrophysics — PHYS 641

Fall 2018

Bridget Andersen

October 3, 2018

**The code script for each of these problems is located in "./prob0/prob0.py" where "0" is replaced by the problem number $(1, 2, 3, 4)$. Here we present any derivations, plots, and discussion necessary to answer the questions.**

## Problem 1

*As mentioned in class, using an SVD factorization of the matrix A when doing linear least-squared fits keeps one from squaring the condition number which happens if you explicitly form $A^T N^{-1} A$. As also mentioned briefly, using a QR decomposition does the same thing, but twice as fast. In case you havent met it before, the QR factorization is $A = QR$, where $Q$ is an orthogonal rectangular matrix (so $Q^T Q = \mathbb{1}$) and $R$ is a triangular matrix. What is the least squares solution for fit parameters when using QR to factor A? Write a code to fit polynomials using both the classical expression and the QR factorization, and show for some case that the QR fit behaves better than the classical fit. For the actual fit, feel free to set the noise matrix to the identity matrix (i.e. go ahead and skip it), but make sure your QR solution shows how you would use it.*

**Solution:** According to single value decomposition (SVD), we can decompose a matrix into three other matrices: $A = USV^T$ where $U$ is orthogonal and rectangular, $S$ is diagonal, and $V$ is orthogonal and square. As discussed in class, the classical least squares parameter solution $m$ is given by:

$$A^T N^{-1} A m = A^T N^{-1} d \implies m = \left(A^T N^{-1} A\right)^{-1} A^T N^{-1} d \tag{1}$$

If we substitute in the SVD decomposition for $A$, we find:

$$\begin{aligned} A^T N^{-1} A m = V S^T U^T N^{-1} U S V^T m = A^T N^{-1} d = V S^T U^T N^{-1} d \\ \implies V S^T U^T N^{-1} U S V^T m = V S^T U^T N^{-1} d \end{aligned}$$

For demonstrational purposes, we let $N = \mathbb{1}$ for now:

$$\begin{aligned} V S^T U^T U S V^T m = V S^T U^T d \implies V S^2 V^T m = V S U^T d \\ \implies m = (V S^2 V^T)^{-1} V S U^T d = \left(V S^{-2} V^T\right) V S U^T d \end{aligned}$$

where we used the fact that $U^T U = \mathbb{1}$ and $S^T = S$. Since $S$ is squared, if any of its entries happen to be small, then this would cause the resulting parameters $m$ to blow up and become very large. In this way, the $S^{-2}$ term is a source of instability in the fit. However, using the single value

decomposition, and assuming that $V$ and $S$ are square and invertible, we can cancel out a power of $S$ like so:

$$\cancel{V}\cancel{S^T}U^TN^{-1}USV^Tm = \cancel{V}\cancel{S^T}U^TN^{-1}d \implies m = \left(U^TN^{-1}USV^T\right)^{-1}U^TN^{-1}d$$

Now this equation for the least squares parameters using SVD decomposition is more stable than what we had before, since it only involves $S^1$.

Using an equivalent argument, we can also use QR decomposition to obtain a more stable fit than the classical equation 1. As mentioned in the problem description, according to QR decomposition, we can decompose a matrix into two other matrices: $A = QR$, where $Q$ is an orthogonal rectangular matrix and $R$ is a triangular matrix. If we substitute QR decomposition for $A$ in equation 1, we find:

$$\begin{aligned} A^TN^{-1}Am = R^TQ^TN^{-1}QRm = A^TN^{-1}d = R^TQ^TN^{-1}d \\ \implies R^TQ^TN^{-1}QRm = R^TQ^TN^{-1}d \end{aligned}$$

For demonstrational purposes, we let $N = \mathbb{1}$:

$$\begin{aligned} R^TQ^TQRm = R^TQ^Td \implies R^TRm = R^TQ^Td \\ \implies m = (R^TR)^{-1}R^TQ^Td \end{aligned}$$

where we used the fact that $Q^TQ = \mathbb{1}$. Now we are in the same scenario as before. If the values along the diagonal of $R$ are too small, then the $(R^TR)^{-1}$ term will cause the resulting parameters $m$ to blow up and become very large. However, assuming that $R$ is square and invertible, we can cancel out a power of $R$ like so:

$$\cancel{R^T}Q^TN^{-1}QRm = \cancel{R^T}Q^TN^{-1}d \implies \boxed{m = \left(Q^TN^{-1}QR\right)^{-1}Q^TN^{-1}d} \tag{2}$$

Same as before, this solution for the least squares parameters using QR decomposition is more stable than what we had before, since it only involves $R^1$.

To demonstrate the stability of this QR decomposition solution, we wrote a python script **("./prob1/prob1.py")** that fits a polynomial function using both the classical method described by equation 1 and the QR decomposition method described by equation 2. For the sake of simplicity, we allow our noise matrix to equal the identity ($N = \mathbb{1}$) for all fits. While we model our data after the 3rd order polynomial: $y = x^3 - 0.5x + 0.2$, we complete classical and QR fits using a large range of polynomial orders. For each fit, we save the RMS error and then plot it versus the polynomial order of the fit. The resulting plot is shown in Figure 1.

From this plot, it is clear that QR decomposition is much more stable than the classical solution. While the RMS error of the classical solution spikes wildly at higher orders, the RMS error of the QR solution remains relatively constant and low. In particular, the classical solution RMS error spikes significantly at the 40th order. To see a demonstration of this, look at Figure 2. At the lower 5th order, both fits behave and obtain good fits of the data. However, at the higher 40th order, the parameters of the classical fit blow up while the parameters of the QR fit continue to behave.

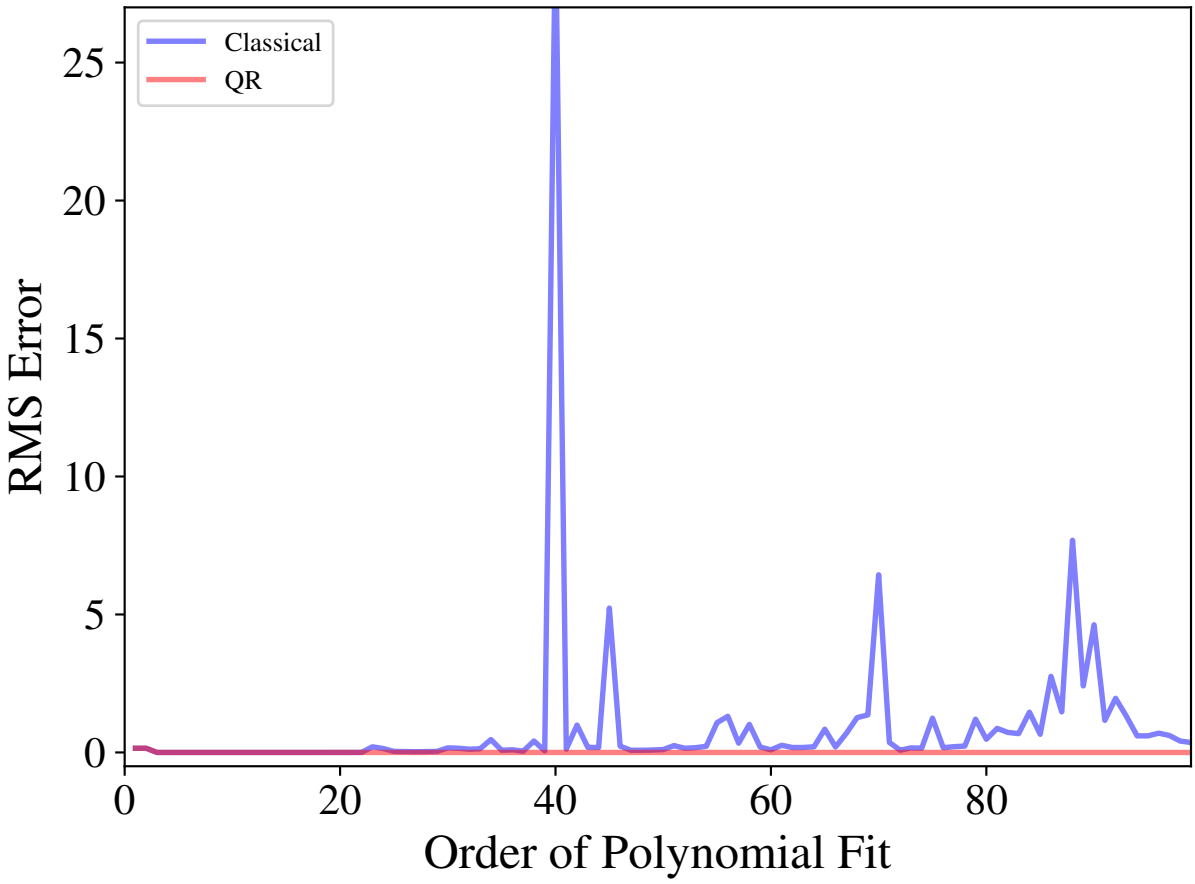



Figure 1: A plot of the fit RMS error versus the polynomial order of the fit. The red line represents fits using QR decomposition (equation 2), the blue line represents fits using the classical method (equation 1).

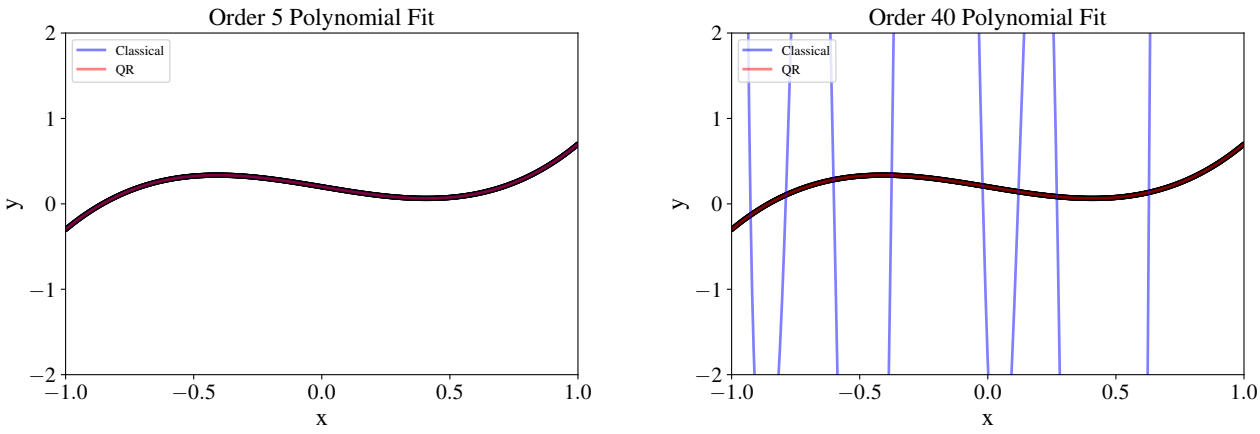



Figure 2: Plots of the classical and QR decomposition solutions for fit polynomial orders of 5 and 40.

## Problem 2

*Legendre polynomials are one useful class of polynomials that are much more stable to use when fitting data. Chebyshev polynomials are another. They are usually referred to as $T_n$ for the nth order Chebyshev polynomial, which is defined in the domain $(-1, 1)$ to be*

$$T_n = \cos(n \arccos(x))$$

*While its not obvious that would be a polynomial, it is. Similar to Legendre polynomials, Chebyshev polynomials can be generated with a recurrence relation*

$$T_{n+1} = 2xT_n - T_{n-1} \tag{3}$$

*with $T_0 = 1$ and $T_1 = x$.*

a) *Write a python code that fits a Chebyshev polynomial to $\exp(x)$ from $-1$ to $1$, with some fine sampling of $x$. Show that even as the polynomial order gets large, the fit stays stable.*
**Solution:** In this problem, we contrast using the classical method (described by equation 1) to fit $\exp(x)$ using both a simple polynomial basis $(1, x, x^2, x^3, ..., x^{\text{ord}})$ as well as the Chebyshev polynomials (described by equation 3). To do this, we created a python script **("./prob2/prob2.py")** that calculates fits using the simple and Chebyshev basis for a large range of fit polynomial orders. For the sake of simplicity, we allow our noise matrix to equal the identity ($N = \mathbb{1}$) for all fits. For each order, we calculate the RMS error for both methods. The resulting plot of RMS error versus polynomial fit order is shown in Figure 3. As is evident, the RMS error of the fits using the Chebyshev polynomials remains relatively low and stable as the order increases, while the RMS error of the fits using simple polynomials spikes wildly at higher orders.

b) *A brief inspection of the definition of the Chebyshev polyomials shows that their y values are bounded by $\pm 1$ (since they are the cosine of something). This property turns out to be very useful if we are trying to come up with a polynomial fit that has a small maximum error, rather than the smallest RMS error. A case where you might care about this is, say, coming up with numerical expressions for transcendental functions on a computer. Of course, you may also just want a polynomial fit that doesnt blow up at the edges (which the least-squares ones tend to do). Because the Chebyshev polynomials go between $-1$ and $1$, we know if we truncate a polynomial fit, the maximum error is bounded by the sum of the absolute value of the coefficients we truncated, since the worst case is those terms all lined up in-phase. Lets use this fact to find a "least-bad" fit to $\exp(x)$. First, fit say a $6$th order (so $7$ terms, including the constant) Chebyshev to $\exp(x)$ on $(-1, 1)$. What is the RMS error? What is the maximum error? Now fit a (much!) higher order Chebyshev fit to $\exp(x)$. If we truncate this fit, i.e. only use the first $7$ terms, what are the RMS error and the max error? How does the max error agree with what you would have predicted based on the terms you ignored? If you have done everything correctly, you should find that the RMS error goes up by about $15\%$, but the max error goes down by more than a factor of $2$.*
**Solution:** We do exactly as the problem suggests. To confirm that the maximum error is bounded by the sum of the absolute value of the coefficients truncated, we continue in the same python script to fit $\exp(x)$ using Chebyshev polynomials up to 6th order. For this fit we obtain RMS and maximum errors: $\sigma_{rms,6} = 1.99 \cdot 10^{-6}$ and $\sigma_{max,6} = 7.98 \cdot 10^{-6}$. Next, we complete a fit to $\exp(x)$ using Chebyshev polynomials up to 40th order. For this fit we obtain a much smaller RMS and maximum errors: $\sigma_{rms,40} = 1.71 \cdot 10^{-15}$ and $\sigma_{max,40} = 2.60 \cdot 10^{-14}$.

Then, when we truncate this fit to only the first 7 terms/parameters, we obtain RMS and maximum errors: $\sigma_{rms,trunc} = 2.26 \cdot 10^{-6}$ and $\sigma_{max,trunc} = 3.41 \cdot 10^{-6}$. By taking the sum of the absolute value of the rest of the parameters that we truncated, we get the predicted maximum error $\sigma_{max,pred} = 3.41 \cdot 10^{-6}$, which is consistent with $\sigma_{max,trunc}$. We also find that the relative difference between $\sigma_{rms,6}$ and $\sigma_{rms,trunc}$ is approximately 13% and that $\sigma_{max,6}/\sigma_{max,trunc} = 2.34$. This is all consistent with expectations.

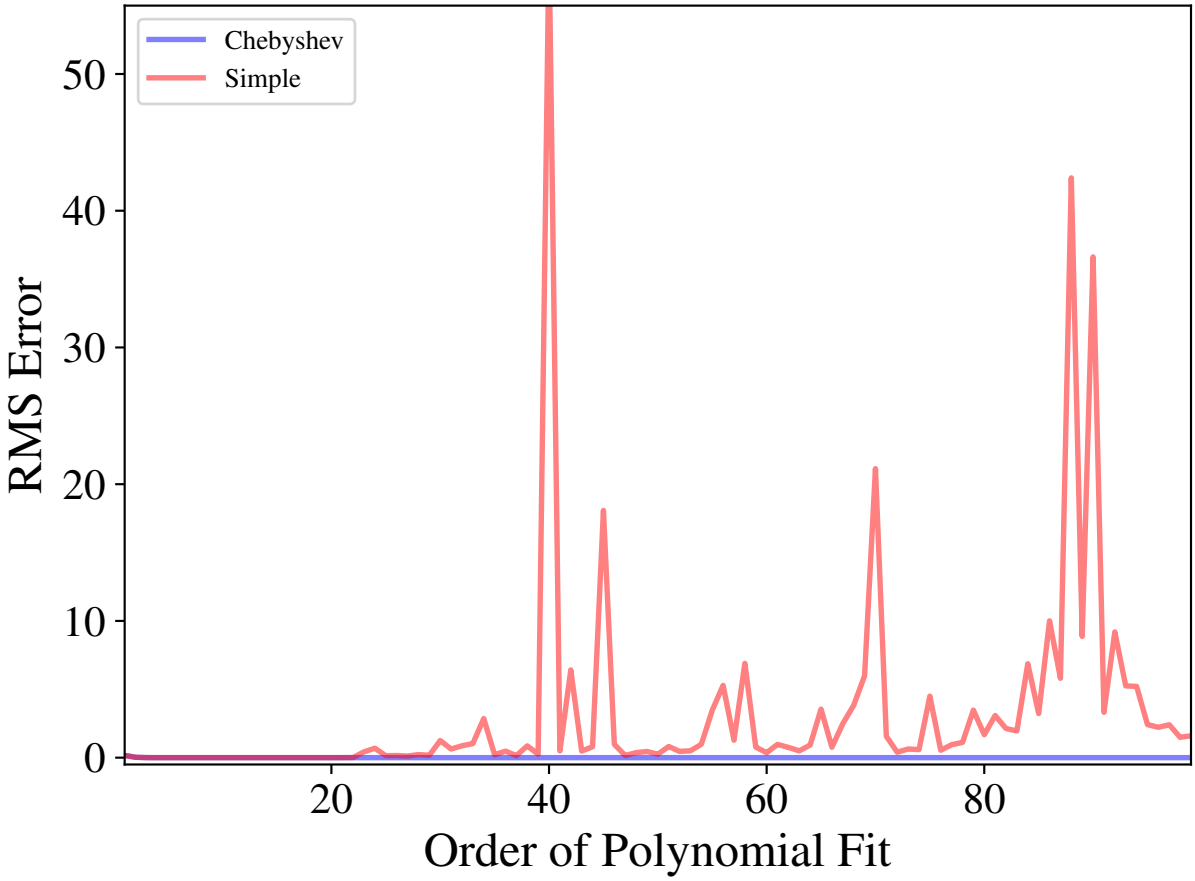



Figure 3: A plot of the fit RMS error versus the polynomial order of the fit. The red line represents fits using a simple polynomial bases, the blue line represents fits using a Chebyshev polynomial basis.

## Problem 3

*We saw in class that by introducing a rotation applied to both the data and the noise, we can go from uncorrelated to correlated noise while leaving $\chi^2$ unchanged. We can use the same trick to go from correlated back to uncorrelated noise. In particular, we can use this to generate realization of random noise with correlations in them. Write a python script that generates random correlated data by taking the eigenvalues/eigenvectors of a noise matrix, and show that if you average over many realizations, that $\langle nn^T \rangle$ converges to the noise matrix. One easy matrix to work with would be $N_{ij} = 1 + \delta_{i,j}$ , i.e. its 1 everywhere except 2 along the diagonal, but your code that does the*

*actual realization of the noise should be general. As an aside, if you have to do this in real life, you may want to look at the Cholesky decomposition. It will do the same thing while usually being much faster to calcualate than eigenvalues/eigenvectors, but is a bit touchier numerically.*

**Solution:** In this problem, we generate many data realizations $d$ of correlated noise according to the suggested noise matrix $N$. Then, by calculating the average $\langle nn^T \rangle$ over many realizations, we show that it converges to the noise matrix. We generate the correlated noise, we using two methods: Cholesky decomposition and eigenvalue decomposition.

First, as in class, we define $d = A(m)+n$. Then we have that $\chi^2 = n^T N^{-1} n$ where $N_{ij} = \langle n_i n_j \rangle$. Assuming that $N$ is symmetric and positive semi-definite, then Cholesky decomposition allows us to decompose the noise matrix like so: $N = LL^T$ where $L$ is triangular. Then we have that $N^{-1} = \left(LL^T\right)^{-1} = L^{-T}L^{-1}$ and so:

$$\chi^2 = n^T N^{-1} n = n^T L^{-T} L^{-1} n = \left(L^{-1}n\right)^T \mathbb{1} \left(L^{-1}n\right)$$
$$\implies \chi^2 = \tilde{n}^T \tilde{N}^{-1} \tilde{n} = \left(\tilde{d} - \tilde{A}(m)\right)^T \tilde{N}^{-1} \left(\tilde{d} - \tilde{A}(m)\right)$$

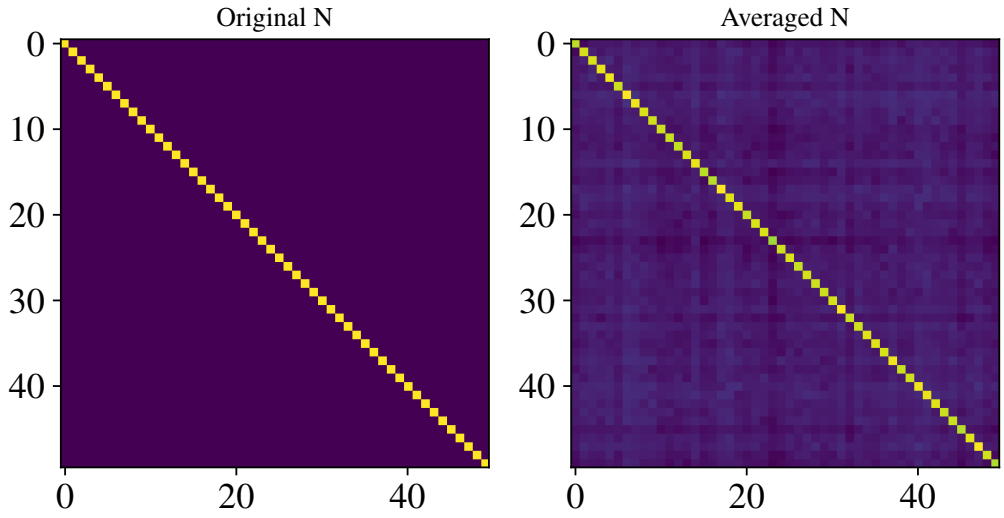
Here we have essentially changed bases $n \to \tilde{n} = L^{-1}n$. Therefore, since the noise matrix in the new basis is the identity $\tilde{N}^{-1} = \mathbb{1}$, $\tilde{n} = L^{-1}n$ must be Gaussian with a standard deviation of 1. So, to generate a realization of correlated noise $n$ according to the correlation matrix $N$, we simply need to calculate $n = L\tilde{n}$, where $\tilde{n}$ is random Gaussian noise with unit variance.

This sort of scheme also works for an eigenvalue decomposition, which allows us to decompose the noise matrix like so: $N = VEV^T$ where $V$ is a matrix whose columns are the eigenvectors of $N$ and $E$ is a diagonal matrix with nonzero elements equal to the corresponding eigenvalues of $N$ (note that $E^T = E$). Then, we can write the following:

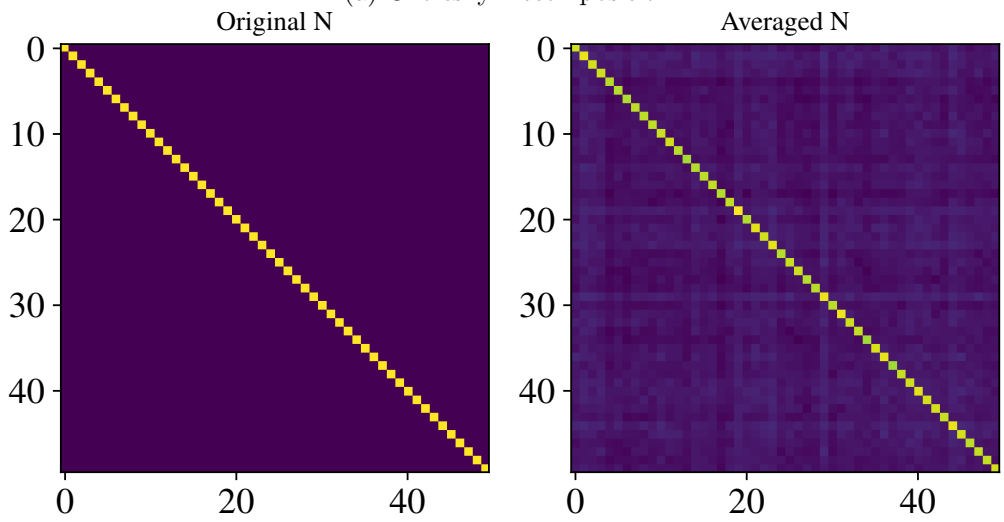
$$N = VEV^T = V\sqrt{E}\sqrt{E}V^T = \langle (V\sqrt{E}\rangle)\langle (V\sqrt{E}\rangle)^T = LL^T \tag{4}$$

Therefore, if we let $L = V\sqrt{E}$, then we can use the same scheme as described above to generate correlated noise realizations according to $N$.

We created a python script **("./prob3/prob3.py")** that generates 5000 noise realizations $n$ using both the Cholesky decomposition and eigenvalue decomposition methods. We then calculate $\langle nn^T \rangle$ for both methods to recover the noise matrix $N$. Representations of the true $N$ matrix and the recovered matrices for both methods are shown in Figure 4. For the Cholesky decomposition, the average error between elements in the true noise matrix and the recovered noise matrix is $\sigma_{avg,Chol} = 0.029$ out of an average true matrix value of 1.02. For the eigenvalue decomposition, the average error is roughly the same $\sigma_{avg,Chol} = 0.032$.




(a) Cholesky Decomposition




(b) Eigenvalue Decomposition

Figure 4: Plots of the recovered noise matrices according to $\langle nn^T \rangle$ for both the Cholesky decomposition and eigenvalue decomposition methods.

## Problem 4

*Now that we can generate fake correlated noise, lets use this in some actual fits. For this, well use a noise matrix that has two components: a diagonal term, plus correlated noise with a correlation length. In particular:*

$$N_{ij} = a \exp\left(\frac{(ij)^2}{2\sigma^2}\right) + (1a)\,\delta\,(ij) \tag{5}$$

*The variance of each point is always* 1*, but if* $a$ *is very small, the data are nearly uncorrelated, while if* $a$ *is very large (i.e. nearly* 1*), they are almost perfectly correlated. If* $\sigma$ *is large, the data points will be correlated over large distances, while if* $\sigma$ *is small, they will only be correlated with their near neighbors.*

a) *Lets say we have* 1000 *points (so* $x$ *goes from* 0 *to* 999*), and a Gaussian source with amplitude*

*of* 1 *and* $\sigma_{src} = 50$ *points in the center. Lets take the values of* $a$ *to be* $(0.1, 0.5, 0.9)$ *and the values of* $\sigma$ *to be* $(5, 50, 500)$*. Write a python script that generates the noise matrix for these values and uses it to report the error bar on the fit amplitude for each pair of* $a$ *and* $\sigma$*. As a sanity check, I get that the error for* $a = 0.5$ *and correlation length* $\sigma = 5$ *samples is* 0.276*.*

**Solution:** We have created a python script **("./prob4/prob4.py")** to accomplish exactly what this question suggests using the classical least squares solution described by equation 1. In this case, the "error bar" on the fit amplitude is obtained by subtracting the best fit model from the data and examining the covariance of what is left. After doing a bit of algebra, we can show that the error bar is then given by $\sqrt{\langle mm^T \rangle}$ where $\langle mm^T \rangle = \left(A^T N^{-1} A\right)^{-1}$ . For the sake of completeness, we can derive this like so:

$$\langle mm^T \rangle = \langle \left(A^T N^{-1} A\right)^{-1} A^T N^{-1} d \cdot d^T N^{-1} A \left(A^T N^{-1} A\right)^{-1} \rangle$$

where we have used the fact that both $N^T = N$ and so $\left(A^T N^{-1} A\right)^T = A^T N^{-1} A$. Now, since we have subtracted the best fit model off the data, we have that $d^2 = \sigma^2$ and so $d^T d = N$. Then we have:

$$\begin{aligned}
\langle mm^T \rangle &= \langle \left(A^T N^{-1} A\right)^{-1} A^T N^{-1} \cdot \cancel{N} \cdot \cancel{N^{-1}} A \left(A^T N^{-1} A\right)^{-1} \rangle \\
&= \langle \cancel{\left(A^T N^{-1} A\right)^{-1}} \cdot \cancel{A^T N^{-1} A} \cdot \left(A^T N^{-1} A\right)^{-1} \rangle = \langle \left(A^T N^{-1} A\right)^{-1} \rangle \\
&\implies \boxed{\langle mm^T \rangle = \left(A^T N^{-1} A\right)^{-1}}
\end{aligned}$$

After coding everything as suggested in the problem, our resulting error bars are shown in the table below and in the labels on the plots in Figure 5.

| | | $\sigma$ | | |
|---|---|---|---|---|
| | | **5** | **50** | **500** |
| | **0.1** | 0.156 | 0.338 | 0.128 |
| **a** | **0.5** | 0.276 | 0.714 | 0.101 |
| | **0.9** | 0.358 | 0.950 | 0.049 |

b) *Explain why the errors behave the way they do. Which set of parameters has the worst error bar? Which has the best? What sort of noise should you be most worried about? You might want to use your code from Problem 3 to generate noise realizations of these noise matrices, and plot the realizations with/without a signal added in.*

**Solution:** We do exactly as the problem suggests and use the framework described in Problem 3 to generate noise realizations of the noise matrix for each $(a, \sigma)$ pair and plot the realizations with and without a signal added in. The resulting plots are shown in Figure 5.

These plots help us to visualize the roles of $a$ and $\sigma$ in correlating the data. As stated in the question description, $a$ is a measure of how correlated the data is. If $a$ is very small, the data are nearly uncorrelated, while if $a$ is very large, they are almost perfectly correlated. As can be seen from the plots, as $a$ increases across each constant $\sigma$ value (moving vertically down the plots), the noise forms more structure. This indicates that neighboring data points are more correlated with each other.

On the other hand, $\sigma$ is the characteristic distance over which the correlation operates. If $\sigma$ is large, the data points will be correlated over large distances, while if $\sigma$ is small, they will only be correlated with their near neighbors. As can be seen from the plots, as $\sigma$ increases over each constant $a$ value (moving horizontally across the plots), the noise goes from oscillating

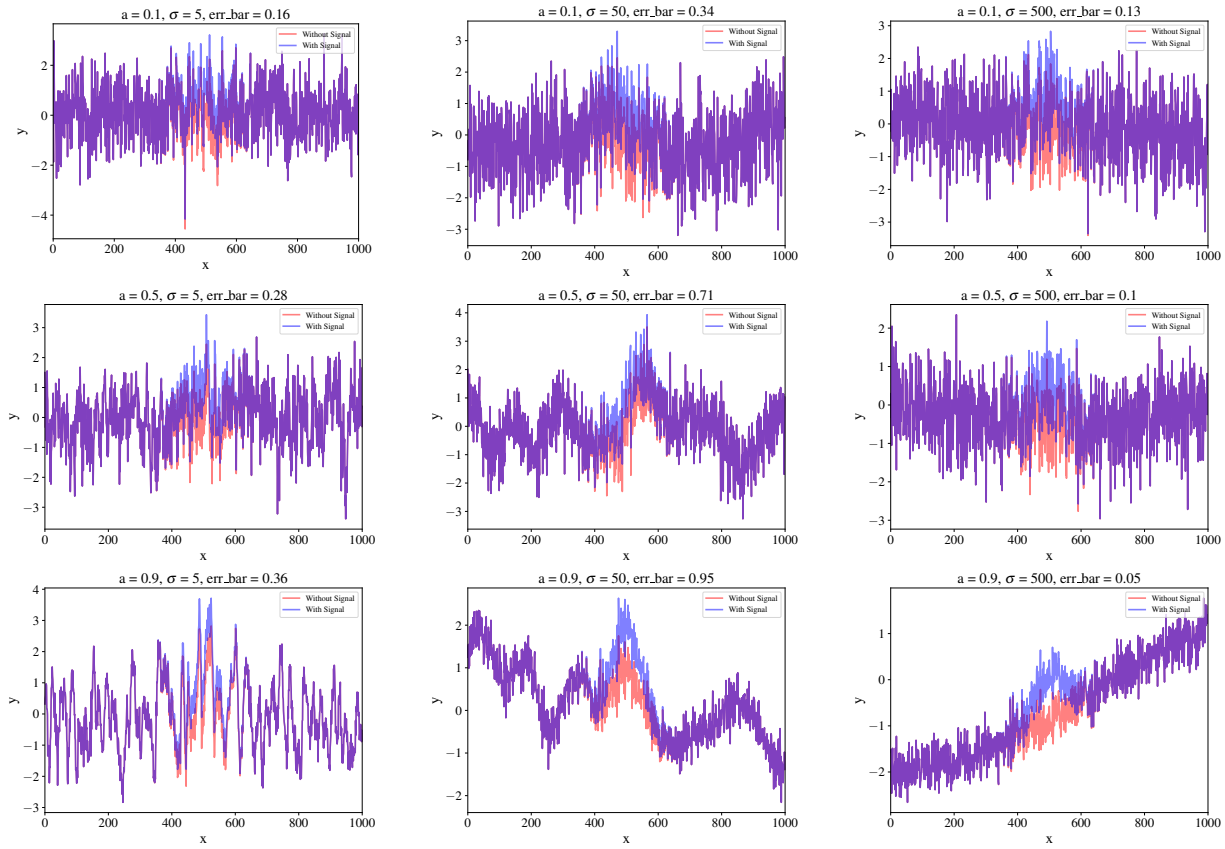



Figure 5: Plots of noise realizations for various values of $a$ and $\sigma$ in equation 5 (each plot is labelled with its corresponding values). The red line represents just the noise realization, the blue line represents the noise plus the gaussian signal that we are fitting.

very rapidly point-to-point at low $\sigma = 5$ to forming coherent peaks at moderate $\sigma = 50$ to forming global trends at $\sigma = 500$ (see, in particular, how the data basically linearly increases over the interval for the $a = 0.9$ and $\sigma = 500$ plot). This indicates that, as $\sigma$ increases, data points over larger and larger distances start to correlate with each other, displaying larger and larger structure.

The parameters that yield the worst error bar are $a = 0.9$ and $\sigma = 50$ with an error of 0.95. This is because there is a high amount of correlation, and the moderate characteristic distance size allows significant peak structures to form in the data that throw off the amplitude of the injected signal Gaussian. The parameters with the best error bar are $a = 0.9$ and $\sigma = 500$ with an error of 0.05. These parameters allow the data to form a single overall upward linear trend in the data over the interval such that the amplitude of the signal Gaussian is not significantly changed.

Based on all these observations, we must be most worried about correlated noise that operates on a moderate distance scale of about $50/1000 = 0.05 = 5\%$ of the interval.