Tutorial problems for Lectures 4 and 5. Due Wednesday September 26th.

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 $\mathrm{A}^T\mathrm{N}^{-1}\mathrm{A}$. As also mentioned briefly, using a QR decomposition does the same thing, but twice as fast. In case you haven't met it before, the QR factorization is $\mathrm{A} = \mathrm{QR}$, where Q is an orthogonal rectangular matrix (so $\mathrm{Q}^T\mathrm{Q} = \mathrm{I}$) 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.

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 n^{th} order Chebyshev polynomial, which is defined in the domain (-1,1) to be

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

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

$$T_{n+1} = 2xT_n - T_{n-1}$$

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

Part 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.

Part b) A brief inspection of the definition of the Chebyshev polyomials shows that their y values are bounded by +/-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 doesn't 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. Let's use this fact to find a "least-bad" fit to exp(x). First, fit say a 6th 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

LS fit with 7 terms gives smallest RMS error (look at max)
Then do it with 200 and find RMS and max err
Worth potting to look at

Sum of coeffs you haven't used. compared to ones you do use

I'm getting a huge increase in both errors

RMS error of what exactly? np.std(pred-true) ?

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.

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 χ^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 $< dd^T >$ converges to the noise matrix. One easy matrix to work with would be $N_{ij} = 1 + \delta_{i,j}$, *i.e.* it's one 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.

Problem 4: Now that we can generate fake correlated noise, let's use this in some actual fits. For this, we'll use a noise matrix that has two components: a diagonal term, plus correlated noise with a correlation length. In particular:

$$N_{ij} = a\exp(-\frac{(i-j)^2}{2\sigma^2}) + (1-a)\delta(i-j)$$

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 one), they are almost perfectly correlated. If σ is large, the data points will be correlated over large distances, while if σ is small, they will only be correlated with their near neighbors.

part a) Let's say we have 1000 points (so x goes from 0 to 999), and a Gaussian source with amplitude of 1 and σ_{src}=50 points in the center. Let's take the values of a to be (0.1,0.5,0.9) and the values of σ 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 σ. As a sanity check, I get that the error for $a = 0.5$ and correlation length $\sigma = 5$ samples is 0.276.

part b) Explain why the errors behave the way they do (comments in your code will suffice, no need to kill trees). 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.