Copy the question files and suggested program templates from /home/5156/assignments/kernel, for example using the following commands:

```
cd

start at your home directory

cp -r /home/5156/assignments/kernel .
copy the directory to all levels

cd kernel

begin working in this directory
```

When you have completed everything, the `make` command should compile and demonstrate your programs. Feel free to modify the `Makefile` if you wish. When you are done, issue the `make submit` command to tell the instructor you have finished. Do not change anything in the directory after running `make submit`.

Last week we wrote some utilities for making and displaying histograms of data. A histogram might be useful in a stochastic measurement as a rough estimate of the probability distribution, i.e., the probability of getting a measurement between \( x_0 \) and \( x_0 + \Delta \) (bin 0), the probability between \( x_0 + \Delta \) and \( x_0 + 2\Delta \) (bin 1), etc. The histogram can be thought of as a rather blocky, stepwise continuous, approximation to the true probability distribution. As an alternative, we propose now to interpolate a continuous estimate of that distribution.

![Figure 1](image)

Figure 1 plots as vertical bars the 50 (artificial) measurements stored in the file `/home/5156/data/stochastic.dat`. If we believe there is one “true” value for the measured quantity, we could characterize the data by their mean, standard deviation, skewness, and (in principle) kurtosis and further cumulants (see *Numerical Recipes*, second C edition, chapter 14). If we're sure that the random fluctuations about the “true” value should be Gaussian, the mean \( \mu \) and standard deviation \( \sigma \) define the probability distribution completely. The solid bell-shaped curve is the normalized Gaussian

\[
\frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)
\]

using the best inferred values for \( \mu \) and \( \sigma \).
1. We have supplied a two-line awk script, mean, that calculates the mean of a list of numbers. Copy it into dist, and modify the script to calculate both the mean and the standard deviation of the data in /home/5156/data/stochastic.dat as given by the square root of Numerical Recipes equation 14.1.7. For the purposes of this exercise, you needn’t implement the two-pass algorithm, equation 14.1.8, unless you wish: the roundoff won’t be very severe. When you have completed this, the command make 1 should run dist on the given file, printing out the mean and standard deviation. The modified awk script should still be only two lines long, so this is a quick exercise.

Armed with an estimate of the probability distribution of your real data, you could create further stochastic data following the same distribution. You might test the hypothesis that the original data were Gaussian distributed by comparing experimentally-observed behavior with that of computer-generated numbers. Numerical Recipes devotes a chapter to random-number generation. We may have an opportunity to use it later in the course for Monte-Carlo integration.

We have so far imagined that the quantity underlying our data is a single Gaussian. What, however, if there are two peaks, or three, or if it’s really a Lorentzian? We can fit each hypothesis, and Numerical Recipes suggests a number of tests and algorithms. The general idea is to make a model with some number of parameters and then find the set of parameters (not necessarily unique) that best fits the data using some criterion such as the least-square deviation. For the sum of two Gaussians you would have five parameters: the mean and standard deviation of each of the Gaussians and the ratio of their weights.

Unless you have a very large set of data, this can be a dangerous game. With \( n \) data points, you can invent a model with \( n \) parameters that fits the data exactly, but that doesn’t mean that the model is right. I don’t know where the adage originated, but it has been observed that “with enough parameters, you can fit an elephant.”

Sometimes a non-parametric estimate of a probability distribution is appropriate. If all we want is to display the bars of figure 1 as a smooth function without imposing our own prejudice as to its form, we can replace each of the spikes with a smooth function of limited extent, then add all the smooth functions. Figure 2 (a and b) shows a blowup of part of the data from figure 1 with a narrower and a wider function; figure 2c shows the whole distribution for a function wider still. The non-parametric smoothing shows some shoulders that may or may not be real.  

\[ \text{Fig. 2} \]

\[ \text{1} \] In fact, the data were generated from a Gaussian distribution with a mean of 10 and a standard deviation of 2, so the shoulders aren’t real. The estimated mean of 10.1121 falls well inside one “standard deviation of the mean” from the underlying “true” value.
In kernel.c we use an inverted parabola cut off where it hits the \( x \) axis (see figure 2a), although you may instead choose a Gaussian with cutoffs or any similar curve. The choice and width of the smoothing function can be thought of as parameters, but they do not bias the form of our estimated probability distribution. Of course, as the smoothing function gets broader, it widens the estimate. As Numerical Recipes comments in introducing section 14.8, one should generally use the original data, rather than a smoothed curve, in fitting a parametric model.

I have found kernel smoothing a useful complement to histograms in exploring the level-spacing statistics of electrons in a wire;\(^2\) I have also used it as a low-pass filter to reduce the spikiness of noisy data.

2. The program kernel.c smooths an unordered set of data points, such as in /home/5156/data/stochastic.dat, writing the axis-style two-column files used to create figure 2. It takes at least one command-line argument, the standard deviation of the smoothing function; an additional optional command-line argument tells how many evenly-spaced ordinates to print (the default is 250). For example, figure 2a can be viewed with the command

```
kernel .01 4000 < /home/5156/data/stochastic.dat | axis -x 12.25 13.5 | xplot
```

Your assignment is to fill in the missing parts of kernel.c (although as always you are free to rewrite it in the language and style of your choice). When you are done, the command make 2 defined in the Makefile will put the program through some tests.

notes on implementing kernel.c

Comments in and accompanying the source file describe the algorithms in greater detail. We again call the histread() function included last week to read the input into an array. Next—we defer to the next paragraph an optimization—for each of the evenly-spaced \( x \) values (outer loop) between the largest and smallest, we consider (inner loop) each input point \( x_i \) in the array, adding to the output value at \( x \) the value of the smoothing function \( K \) evaluated at \( x - x_i \).

The function \( K(x) \) is zero for \( |x| \geq a \), so only points \( x_i \) lying within a distance \( a \) of \( x \) contribute at \( x \). To avoid having to test every input point at every \( x \) point, we sort the input array into ascending numerical order: we suggest using the system-supplied qsort() function to sort the array in place.\(^3\) This lets us make binary searches of the input array to find the first and last point that will need to be evaluated. The way most people find the right page in a dictionary is by a modified binary search. Numerical Recipes discusses the algorithm on page 117 (second C edition); you needn’t implement the fancier hunt() idea on the following pages. To find an item in a sorted list of length \( N \) by binary search takes on average a time that grows with \( N \) only as \( \log_2 N \), whereas to find it the straightforward way, going through the array until it’s found, the average time is \( N/2 \). If you had a million things, and each step took one second, it would take you almost six days to find your item in an unsorted array, but if the array had already been sorted, a binary search would average only 20 seconds. We still have to sort the array, a step whose time scales as \( N \log N \), but we do that only once. If the number of \( x \) steps in the outer loop is comparable to \( N \), we win very big with the binary search.

\(^2\) See for example Figure 1 in Phys. Rev. B 69, 054403 (2004).

\(^3\) “In place” means without the need for a second array. If you would like to understand the algorithm used internally by qsort(), it is explained in section 4.10 of Kernighan and Ritchie (2nd ed) and section 8.2 of Numerical Recipes (2nd C ed). Compare Kernighan’s and Ritchie’s elegant recursive code to the cumbersome, difficult-to-read subroutine in Numerical Recipes. This is proof, if any were needed, that Press et al. never really learned the C language, instead just translating literally from the original FORTRAN. Their English descriptions of algorithms, however, are good.