7. Statistics

Figure 7.1: *Der Sommer*, by Abel Grimmer (565-1630).

Source: Wikimedia Commons

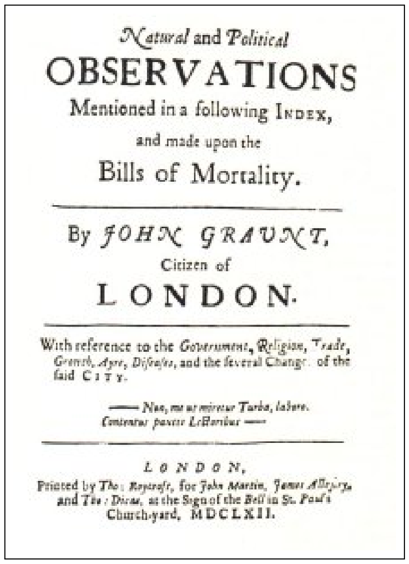
7.1. Introduction

In the 17th century, English authors John Graunt and William Petty began writing about a new science called "Political Arithmetic", which tried to understand social, economic, and public health problems through the collection and analysis of numerical data. In the 18th century, authors such as Germany's Gottfried Achenwall began writing about another new field of study called "Statistik" which aimed at discovering the general principles by which a state could be successfully run.

Statistik soon began using the techniques of Political Arithmetic. The success of a state might depend on the amount of wheat or milk it produces, or the number of skilled craftsman. A spreading plague might be detected by systematically collecting data about deaths. These studies were the beginning of what we call "statistics" today.

The modern science of statistics attempts to see inaccessible underlying truths by sampling the superficial things that are visible to us. By surveying a limited number of households, we arrive at an estimate of the total number of families living in poverty. By observing a few thousand particle decays, we estimate the probability that such decays will happen. In the language of Antoine de St. Exupery's *Little Prince*, statistics tries to see the elephant that lies hidden inside the boa (see Figure 7.3).

The available data is often incomplete, and shows us only a blurry outline of what's underneath, so statistics also tries to measure the uncertainty in its estimates. These measures of uncertainty help us judge how much we should trust our statistical conclusions.



Natural and *Political*
OBSERVATIONS
Mentioned in a following INDEX,
and made upon the
Bills of Mortality.

By *JOHN GRAUNT*,
Citizen of
LONDON.

With reference to the *Government*, *Religion*, *Trade*, *Growth*, *Ayre*, *Diſeaſes*, and the ſeveral Changes of the ſaid CITY.

—— *Non, me ut miretur Turba, laboro.*
Contentus paucis Lectoribus ——

LONDON,
Printed by *Tho: Roycroft*, for *John Martin*, *James Allestry*, and *Tho: Dicas*, at the Sign of the *Bell* in *St. Paul's* Church-yard, MDCLXII.

Figure 7.2: John Graunt's *Observations on the Bills of Mortality* (1662) studied mortality data in an effort to understand the spread of Bubonic Plague.

Source: Wikimedia Commons

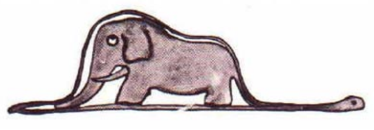


Figure 7.3: A boa who's swallowed an elephant, from Antoine de St. Exupery's *The Little Prince*.

7.2. Summarizing Data with Histograms

It can be hard to see the patterns in a bunch of raw numbers, but a graph often makes the data snap into focus. In this section, we'll look at a new kind of graph called a "histogram". The histogram was introduced in 1891 by Karl Pearson, one of the founders of modern statistics. It summarizes an arbitrarily large amount of data by reducing it to a smaller, fixed, number of data points that represent how often certain values appear in the original data.

Figure 7.4: British mathematician Karl Pearson (1857-1936).
Source: Wikimedia Commons

Let's look at an example. Particle physicists often use "scintillation detectors" to measure the energy of subatomic particles. A "scintillator" is a material such as Thallium-doped Sodium Iodide which produces a flash of light when an energetic particle passes through it. By measuring this flash of light, we can find out how much energy is deposited as a particle passes through. More light means more energy.

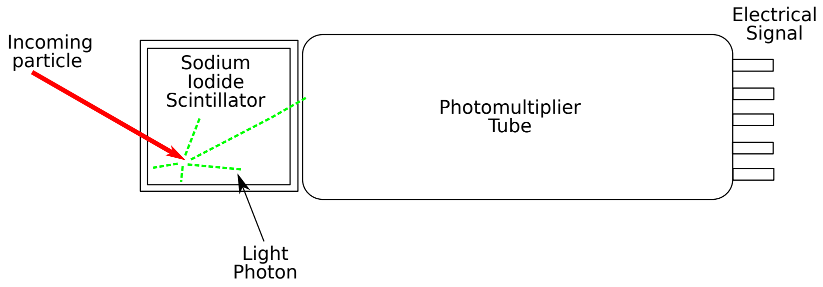


Figure 7.5: A scintillation detector produces a flash of light whenever an energetic particle passes through it. The amount of light is proportional to the energy that the particle deposits in the detector. The flash of light is converted into an electrical signal by a "photomultiplier tube", and the electrical signal is measured and recorded.

The output of such a detector is just a bunch of numbers, each of which corresponds to the energy deposited by a detected particle.[1] These energies are measured in "electron Volts" (eV), and a million electron volts is called an MeV. The data we collect might look like Figure 7.6.

[1] The size of the electrical signals coming out of the detector is proportional to the energy. For our example, we'll just assume that we can read the energy values directly.

```
15.130490
16.942571
16.627112
10.780935
14.569799
15.192141
 6.489004
12.386759
17.793823
 4.181682
19.381618
...
```

Figure 7.6: Some data from our detector, representing energies measured in MeV. It's hard to make sense of a stream of numbers.

It's hard to see patterns in a stream of numbers like this, but let's imagine that we've looked at the data and noticed that all of the numbers lie between 0 and 20 MeV. It would be interesting to know how the numbers are distributed in this range. Are they spread uniformly? Do they bunch up in some places?

If we were rather bad at programming but good with tools, we might construct a set of bins like those in Figure 7.7 to satisfy our curiosity. Each bin represents a 4 Mev-wide range of energies. Whenever we see a particle with an energy in that range, we could drop a marble into the corresponding bin. After going through all of the data we could look at our bins and easily see which energies were the most common, because they'd contain the most marbles.

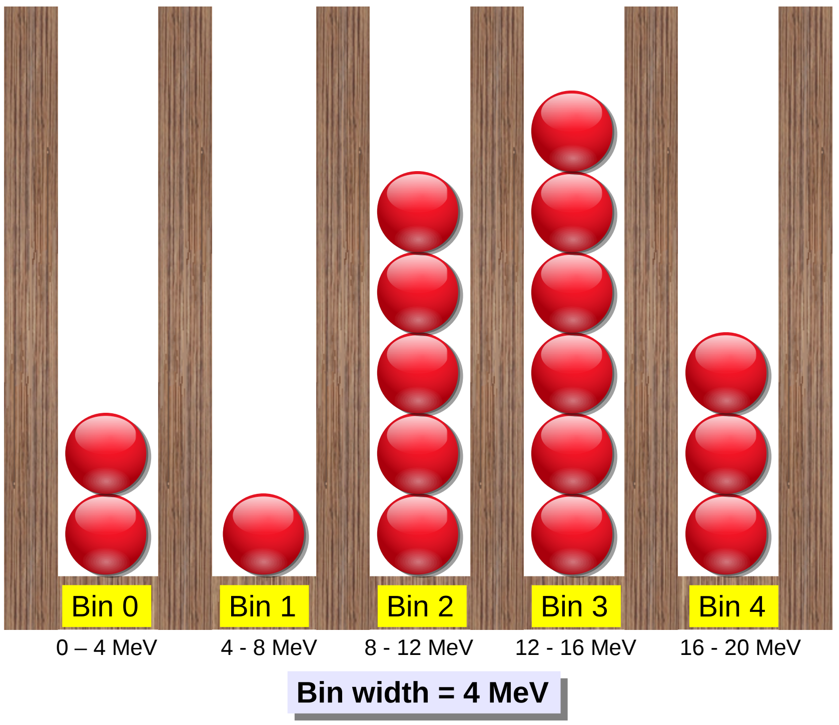


Figure 7.7: Binning the detector data produces a histogram.

The pattern of high and low marble stacks that we've produced is called a histogram. It tells us how frequently a measurement falls within a given range. For this reason, histograms are sometimes called "frequency plots".

If we wanted to save our histogram (maybe we want to re-use the lumber for another project?) we could just write down the number of marbles in each bin. But if a histogram is just equivalent to a list of numbers, that means we could use an array in a C program to store it.

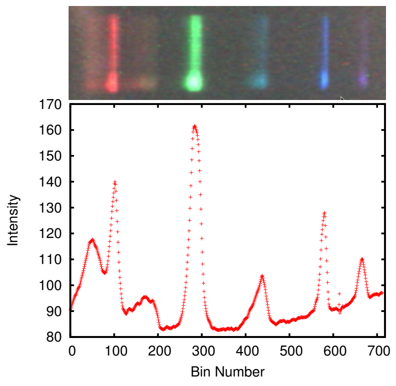


Figure 7.8: A histogram can also represent a spectrum. The most intense places on this fluorescent light spectrum are just those where photons are most frequent. In the graph, we've marked only the top of each of 700 "columns of marbles".

Spectrum taken by Finian Wright, using a DIY spectrometer.

Program 7.1 reads energies from a file and produces a histogram, represented by an array of bin counts. The program reads a list of numbers from the file `energy.dat`. The numbers represent energies from a scintillation counter, ranging between approximately 0 and 50 MeV. For each number, the program adds a virtual marble to one of 50 bins. The bins are the elements of the array named `bin`.

To find out which bin to put the marble into, the program divides each energy value by the bin width, and rounds the result down to the nearest integer. The result is the bin number. For example, take a look at Figure 7.7 again. In this figure, an energy of 9 MeV would go into bin number 2, since the bin width is 4 MeV, and $9/4 = 2.25$.

In Program 7.1, for simplicity, we've made the bin width 1 MeV, so we can just look at the bin number to see the approximate energy it represents.

Program 7.1: hist.cpp

```
#include <stdio.h>
int main () {
  int i, binno, overunderflow = 0;
  double energy, binwidth = 1.0;
  int bin[50];
  FILE *input;

  for ( i=0; i<50; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &energy ) == 1 ) {
    binno = energy/binwidth; // Find which bin.

    // Is it too small or too big?
    if ( binno < 0 || binno >= 50 ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }

    bin[binno]++; // Add a marble to this bin.
  }
  fclose(input);

  for ( i=0; i<50; i++ ) {
    printf ("%d %d\n", i, bin[i]);
  }
  printf ("# Saw %d over/underflows\n", overunderflow);
}
```

Read lines from file.

At the end of the program, it prints out each bin number and the number of virtual marbles that bin contains.

As we saw in Chapter 6, it's important to check our array indices to make sure we're not going past the end of the array. What if the file `energy.dat` contains some unexpected energies that would fall into bins beyond the last element of our `bin` array? What if a negative number somehow found its way into the file? We'd want to know about these things, but we wouldn't want our program to crash.

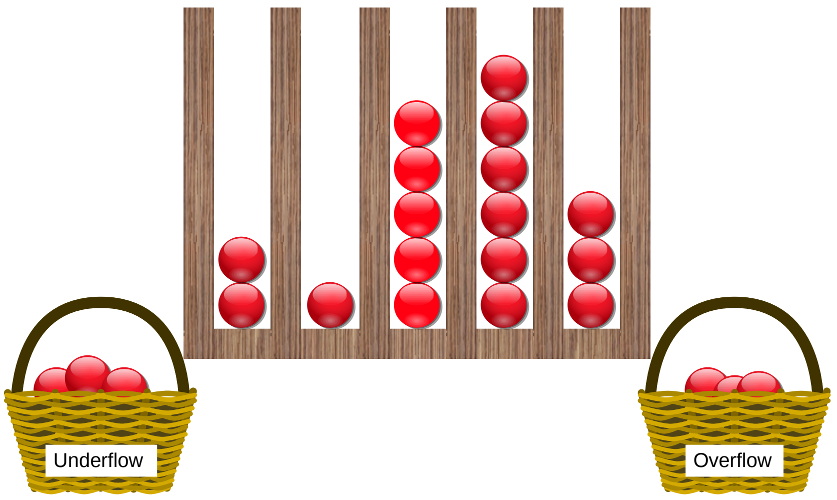


Figure 7.9: In Program 7.1, `overunderflow` counts the number of overflows and underflows.

To record these unexpected values, Program 7.1 has a variable called `overunderflow` that counts the number of overflows (energies that are too low) and overflows (energies that are too high). The program checks the energy with an "`if`" statement like this:

```
if ( binno < 0 || binno >= 50 )
```

The condition in the "`if`" statement checks to see if either of two conditions are true by using the "`or`" operator, `||`. (We say `>= 50` because the highest bin number is 49.)

If an overflow or underflow is found, the program increments the value of `overunderflow` and then immediately skips to the next energy value in `energy.dat`. It accomplishes this by using a "`continue`" statement. In Chapter 4 we saw that it was possible to stop a loop by using a `break` statement. The `continue` statement is similar, except that instead of stopping the loop, it causes the program to skip the rest of the current trip through the loop and immediately start the next trip.

When the program finishes, it prints out the number of overflows and underflows that were seen. Notice that it prints a hash symbol (#) in front of the message about over/underflows. This is so the message won't confuse *gnuplot* if we want to plot the results. *Gnuplot* ignores any lines that begin with `#`.

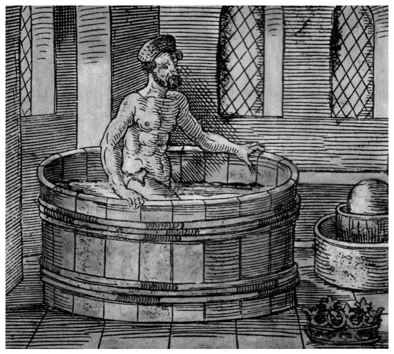


Figure 7.10: Legend has it that the Greek philosopher Archimedes proved the value of noticing overflows. He'd been given the task of measuring the density of a crown to determine whether it was made of pure gold. This required measuring the crown's volume, but he couldn't figure out how to do that. Getting into his bath one day, he noticed that his body displaced an equal volume of water, and it was easy to measure the volume of water. He jumped from the tub, shouted "*Eureka!*", meaning "I've found it!" and ran naked through the streets of Syracuse.
Source: Wikimedia Commons

Exercise 36: Making a Histogram

For this exercise you'll need a copy of the data file `energy.dat`. You can find instructions for obtaining it in Appendix C.2 on page 622. Take a look inside this file using *nano*. You should see a single column of numbers, representing simulated energy measurements of 100,000 particles.

```
35.130490
36.942571
36.627112
40.780935
34.569799
35.192141
36.489004
32.386759
...
```

Figure 7.11: Some of the data in the file `energy.dat`.

Try graphing this file by starting *gnuplot* and typing:

```
plot "energy.dat"
```

The result should look something like Figure 7.12.

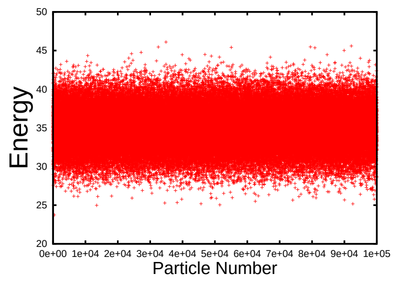


Figure 7.12: The data in `energy.dat`, plotted with *gnuplot*.

Exit from *gnuplot* and then create, compile and run Program 7.1. The program's output should be two columns of numbers (a bin number and the number of "virtual marbles" in that bin), followed by a message about overflows and underflows. By looking at the columns of numbers, you should already be able to see a pattern emerging.

Now run the program again, redirecting its output into a file, like this:

```
./hist > hist.dat
```

Start *gnuplot* and plot the data by using the command:

```
plot "hist.dat" with impulses
```

"`with impulses`" causes *gnuplot* to draw a vertical line for each point. The result should look something like Figure 7.13. Where do most of the energy values lie?

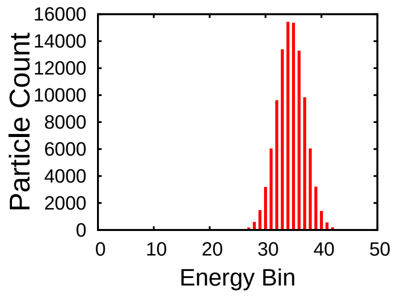


Figure 7.13: The output of Program 7.1, plotted with *gnuplot*.

Even though the data file we're analyzing (`energy.dat`) contains 100,000 lines, the output of Program 7.1 is just two 50-line columns. We could give Program 7.1 a million times more data to analyze, and the program's output would still be only fifty lines, although the numbers on those lines would be larger. This is one reason histograms are useful: they can summarize large data sets very efficiently. In the exercise above, the program turns 100,000 numbers into a 50-number summary.

7.3. Resolution and Range of Histograms

We could improve Program 7.1 by making a few changes that allow us to adjust the *resolution* of the histogram (the width of its bins) and its *range* (the lowest and highest energy values it can display). Let's also make the program more general, so it's clear we can use it for other kinds of data besides energy values.

Controlling the Resolution of a Histogram:

In Program 7.1 we set the bin width to 1 MeV for convenience, so we could see the energy values by just looking at the bin number. Bin number 35 corresponded to 35 MeV. What if we wanted a finer- or coarser-grained histogram, though? We might want a bin width of 0.5 MeV or 2 MeV, for example. In that case, we might want the program to print the *energy value* of each bin instead of the bin number.

But do we want to print the energy at the left side of the bin, the right side, or the middle? These are all different. Let's just print all of them, and then we can decide which value we want to use when we graph the data.

We can make this happen by modifying just a few lines of our program. Instead of saying this:

```
printf ("%d %d\n", i, bin[i]);
```

we can say this:

```
elow = binwidth*i;
emid = binwidth*(0.5+i);
ehi = binwidth*(i+1);
printf ("%lf %lf %lf %d\n", elow, emid, ehi, bin[i]);
```

The first three lines calculate the energy value at the left, center, and right of the bin (to get the center, we add 0.5 to the bin number). Then, instead of printing the bin number, we print all three energy values. This will mean that our output has four columns: the three energy values and the number of "marbles" in the corresponding histogram bin.

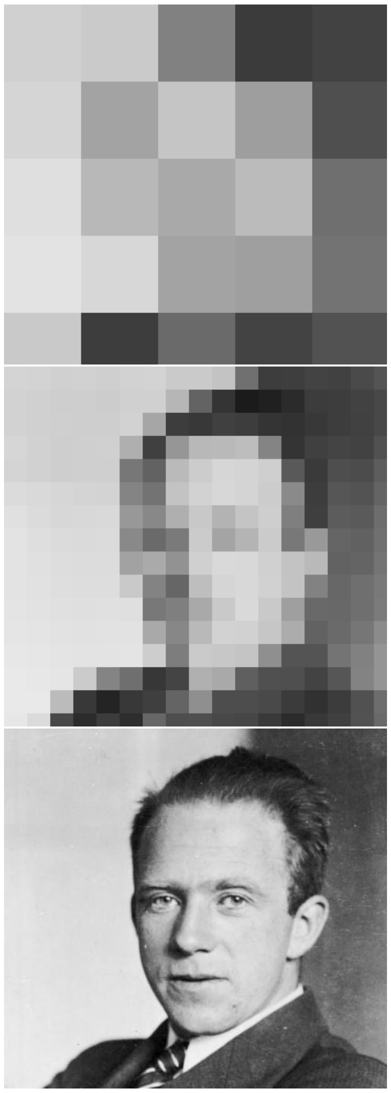


Figure 7.14: Finer-grained resolution sometimes shows us features of our data that are invisible at lower resolutions. (Photo of Werner Heisenberg.)
Source: Wikimedia Commons

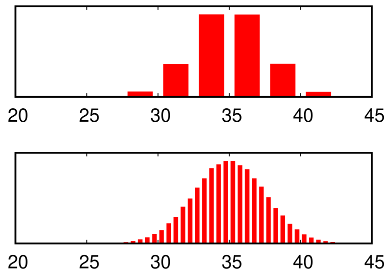


Figure 7.15: A low-resolution histogram (top) with 10 bins, and a high-resolution histogram (bottom) with 50 bins, both showing the same set of data.

Controlling the Range of a Histogram:

Program 7.1 also assumes that the energy range we're interested in starts at zero. Sometimes this won't be the case. Maybe we want to focus on the range between 30 and 40 MeV, for example. Or, if we're measuring something other than energy, we might even have negative values. Maybe we're measuring distance, and we want to look at values between -10 meters and 10 meters, where zero is the origin of our coordinate system.

To accommodate that we'll need to make a few more changes to our program. First, let's define the lower bound of our energy range with a new variable:

```
double emin = 20.0; //MeV.
```

Here we've set it to 20 MeV, but we could set it to whatever we want. Now we'll need to use this value when we calculate the bin number (binno) and when we calculate the energy of each bin at the end of the program. Our new calculation of binno would look like this:

```
binno = (energy-emin)/binwidth;
```

Instead of just energy, we're using energy-emin to determine which bin we should use. When energy is equal to emin, the bin number is zero. At the end of the program, when calculating the left, center, and right energy values of the bin we can say:

```
elow = emin + binwidth*i;
emid = emin + binwidth*(0.5+i);
ehi = emin + binwidth*(i+1);
```

We've added emin because the lowest bins correspond to that energy.

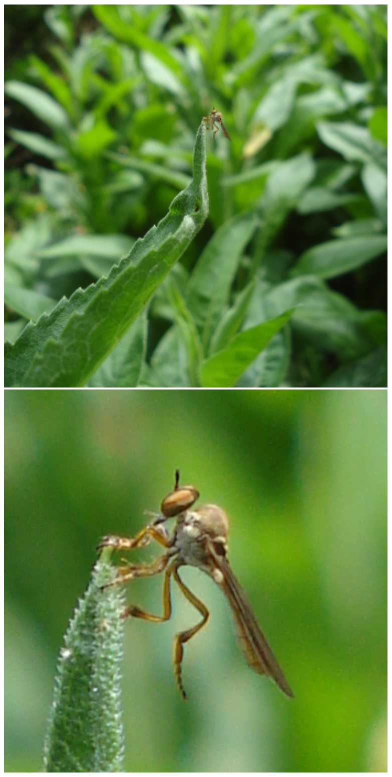


Figure 7.16: Two images with the same resolution (both are 348×348), but the bottom image zooms in on a small region near the center of the upper image. If we have a fixed number of histogram bins, we should try not to waste them on regions where there's no interesting data. (Image of a "gnat ogre" – a robber fly of the genus *Holcocephala* – taken by the author.)

Calculating `binwidth` *Instead of Specifying It:*

It's often convenient to specify the limits of a histogram's range and the number of bins, and then let the program calculate the value of binwidth. We might, for example, want 100 bins covering the range from 20 MeV to 45 MeV. That would tell us that each bin has a width of $(45-20)/100 = 0.25$ MeV.

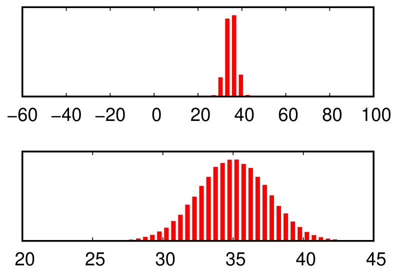


Figure 7.17: A histogram with a large range (top), and a small range (bottom). Both show the same set of data using the same number of bins.

We'll need to rearrange a few things to make that happen. Let's start by adding a new variable to specify the upper end of our range:

```
double emax = 45.0; //MeV.
```

Now let's define a variable that specifies the number of bins, to make it easy to adjust this value later:

```
const int nbins = 50;
int bin[nbins];
```

As we mentioned in Chapter 6, the word `const` tells the C compiler that this value should never change. (See Page 186.) Next, we need to add a line to calculate the value of `binwidth`:

```
binwidth = (emax-emin)/nbins;
```

Finally, we need to replace `50` with `nbins` wherever the program has previously assumed there were 50 bins.

Putting It All Together:

Okay, now let's see what the finished program looks like after we've made all of these changes. Notice that Program 7.2 uses `x`, `xmin` and `xmax` in place of `energy`, `emin` and `emax`, since we can use this program for any kind of data. There are also some new `printf` statements at the bottom of the program that remind the user about the program's settings.

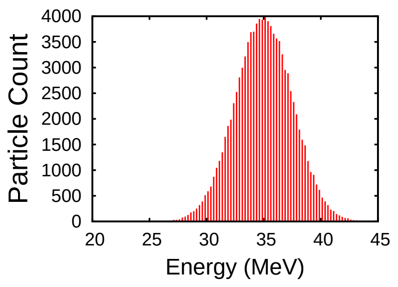


Figure 7.18: Output of Program 7.2, plotted with the *gnuplot* comand `plot "hist.dat" using 2:4 with impulses`.

Program 7.2: hist.cpp, Improved

```
#include <stdio.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 20.0;
  double xmax = 45.0;
  const int nbins = 100;
  int bin[nbins];
  FILE *input;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &x ) == 1 ) {
    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
}
```

7.4. Two-Dimensional Histograms

Imagine that you're a school principal whose students have just finished taking reading and math tests. You could make a histogram of all the reading scores or all the math scores, but you'd like to see how reading scores and math scores are related to each other. Do students with high math scores also have high reading scores, or do students excel in only one area? What can we do? Let's stroll down the hall and talk to the Shop teacher. He's a clever guy. Maybe he'll have a suggestion.

You begin by telling him about the wooden bin you constructed for sorting marbles in the preceding section. He thinks about the problem for a moment, then says, "Well, all you need to do is make a crate that lets you sort marbles out in two directions: one direction for reading scores and the other for math. Give me a few minutes and I'll make one for you." Sure enough, after a few minutes of sawing and hammering, he's produced a crate like the one shown in Figure 7.19.

"Great!" you say. "Each marble represents a student. I just need to drop the marble into the bin that corresponds to that student's reading and math scores. In the end, the number of marbles in a bin will tell me how many students had that particular combination of reading and math scores."

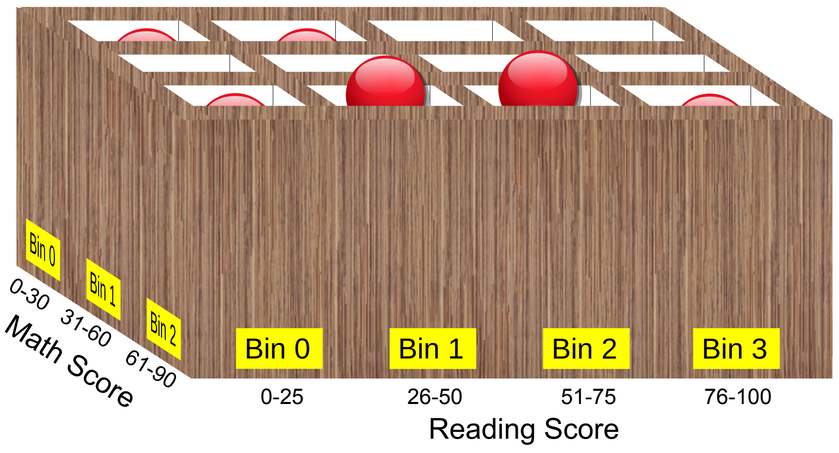


Figure 7.19: Binning marbles in two directions produces a two-dimensional histogram. In this example, math scores range from zero to 90 and reading scores range from zero to 100. We've divided the math scores into bins with a width of 30, and the reading scores into bins with a width of 25.

Our crate full of marbles can be thought of as a two-dimensional histogram[2]. As with the one-dimensional version we saw in the preceding section, we can save our histogram by just writing down the number of marbles in each bin. In Program 7.1 we used a one-dimensional array (`bin[50]`) to hold the values in our one-dimensional energy histogram. For a two-dimensional histogram, we'll need a *two-dimensional* array. We might store the number of marbles in each bin of Figure 7.19 in a 3×4 array of integers, defined like this: `int bin[3][4];`

[2] Two-dimensional histograms are sometimes called "bivariate" histograms, because they show data from two variables (reading score and math score in this example).

Take a look again at Program 7.1 (`hist.cpp`). If we wanted to modify this program so that it makes a two-dimensional histogram, we'd need to change `bin` into a 2-d array, and we'd need to modify the way we fill this array.

For example, assume we have a data file that has two numbers on each line: a math score and a reading score. Instead of the single bin number (`binno`) that we calculate in Program 7.1, we now need to calculate two bin numbers, one for math and one for reading. We might do that like this:

```
mbin = math/mbinwidth;
rbin = reading/rbinwidth;

if ( rbin < 0 || rbin >= nrbins ||
     mbin < 0 || mbin >= nmbins ) {
  overunderflow++;
  continue; // Skip this value and jump to the next.
}

bin[mbin][rbin]++; // Increment the appropriate bin.
```

where `reading` and `math` are the reading and math scores, `mbin` and `rbin` are the calculated bin numbers for math and reading, `mbinwidth` and `rbinwidth` are the widths of the math and reading bins, and `nmbins` and `nrbins` are the number of math and reading bins.

Figure 7.20 shows two ways of representing a 2-dimensional histogram of reading and math scores. Here the reading and math scores both range between zero and 100, and we've split each range into ten bins. In the top picture, we use a vertical bar to represent the height of each bin's stack of marbles. In the bottom picture we look down on the top of these stacks, and we've color-coded each stack to indicate its height.

Two-dimensional histograms are useful when we want to see how two measured quantities interact with each other. In Figure 7.20, we can easily see that students with high math scores also tend to have high reading scores. This wouldn't be obvious if we just looked at the numbers, or graphed math or reading scores by themselves.

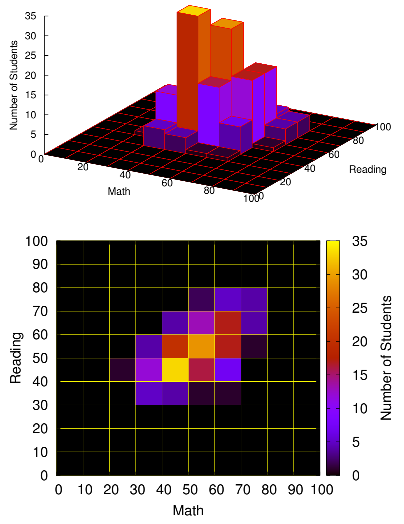


Figure 7.20: Two ways we might represent the data in a two-dimensional histogram.

7.5. Finding the Mean

Looking at the one-dimensional histogram in Figure 7.13 we can see that the energies tend to cluster around approximately 35 MeV, but they trail off to the left and right in a bell-shaped curve. If all of the particles actually had the same energy, and all of their energy was deposited in the detector, we might expect all of the numbers in `energy.dat` to be exactly the same. In practice, though, our measurements will always have some random variation no matter how careful we are. This is partly because of imperfections in our instruments, but there may also be physical limits to the precision of our measurements

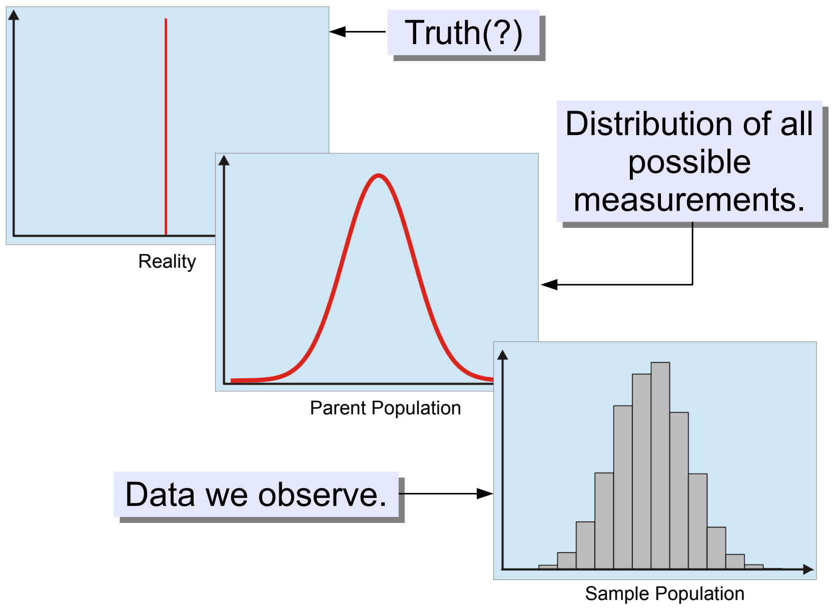


Figure 7.21: We are always at two removes from the "underlying truths" that we're trying to measure. Statisticians call the right-hand graph the "sample population", and the middle graph the "parent population", from which the sample is drawn at random.

If we made an infinite number of measurements, we might see that they're spread out like the middle graph in Figure 7.21. In reality, we make a finite number of measurements that are just a small sample of all of the possible measurements, like the right-hand graph. If we only take a few measurements, it's not too unlikely that all of them may happen to lie on the left or right side of the true value. As we make more measurements, our data will begin to look more and more like the middle graph.[3]

[3] In statistics, this is called "The Law of Large Numbers".

Once we've taken enough measurements to approximate the middle graph, what's our best guess for the true value in the left-hand graph? Some of our measurements are higher than the true value and some are lower, but we expect that the true value lies somewhere between the extremes, at some "average" value.

In everyday speech, we use the word "average" to mean "typical". The "average guy" is a typical person. How do we measure this, though? How can we *objectively* decide what's "typical"?

Figure 7.22: *The Tempting Cake*, by Albert Rosenboom

Source: *Wikimedia Commons*

In science, we often use a quantity called the "arithmetic mean" (often just called the "mean") to represent what's "average" or "typical". You've probably used this before. The mean of a set of values is the sum of all the values, divided by the number of values. Mathematically, we could write it like this:

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{7.1}$$

where N is the number of values, X_i are the values themselves, and $\overline{X}$ is the mean.

If we slice a cake into several pieces, the *mean size* of a piece is the sum of the size of all the pieces (which is just the total size of the cake), divided by the number of pieces. The *mean* is the size that each piece would have if the cake were sliced up into perfectly equal parts.

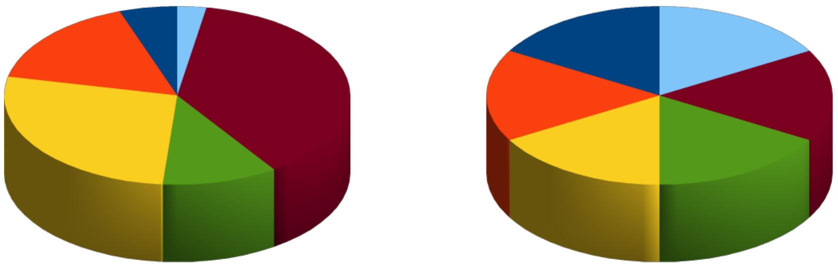


Figure 7.23: On the left, an unfairly sliced cake. On the right, a cake sliced into equal pieces. The size of each right-hand slice is equal to the *mean size* of the left-hand slices.

We often assume that the mean value of our measurements is the best guess at the true, underlying value that we're trying to measure. If we make enough measurements, we expect that the mean value will approximate the mean value of all possible measurements, and we expect that the mean of all possible measurements will approximate the true, underlying value, which may never be directly accessible to us.

Program 7.3 reads the energy values from `energy.dat` and finds their arithmetic mean. In the program, the variable named `sum` is intially set equal to zero. Each time a new number is read, it's added to `sum`. After reading all of the numbers, the program calculates the mean by dividing the sum by the number of energy values.

Program 7.3: mean.cpp

```
#include <stdio.h>
int main () {
  double energy;
  double sum = 0.0;
  int nvalues = 0;
  double mean;
  FILE *input;

  input = fopen("energy.dat","r");
  while ( fscanf( input, "%lf", &energy ) == 1 ) {
    sum += energy;
    nvalues++;
  }

  mean = sum/nvalues;

  printf ("Number of values is: %d\n", nvalues );
  printf ("Mean value is: %lf\n", mean );

  fclose (input);
}
```

We could also modify our histogram program (Program 7.1) so that it tells us the mean energy. Program 7.4 is a new version of `hist.cpp` that adds up the energy values as they're read, and prints out the mean when it's done. Again, we put a # on the front, so *gnuplot* will ignore this line.

Notice that we want to include all of the energy values, even the underflows and overflows. We want the arithmetic mean of *all* values.

Exercise 37: You Big Meanie!

Modify your earlier `hist.cpp` program so that it looks like Program 7.4. Compile and run it. Does the value given by the program look consistent with what you saw when you plotted a histogram of the data (Figure 7.13)?

Figure 7.24: In the 1968 Beatles movie *The Yellow Submarine*, the Blue Meanies *hated* music.

Source: *unigami, at Deviant Art*

Program 7.4: hist.cpp, Version 2

This version of the program prints the average energy value. Changes from Program 7.1 are shown in bold.

```
#include <stdio.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 50.0;
  double sum = 0.0;
  int nvalues = 0;
  const int nbins = 50;
  int bin[nbins];
  FILE *input;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &x ) == 1 ) {

    sum += x;          Add each value to the sum.
    nvalues++;         Count the number of values.

    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Nvalues = %d\n", nvalues );
}
```

7.6. Standard Deviation

Figure 7.13 shows that the energy values in `energy.dat` tend to bunch up in one spot, forming a peak. If you were describing this shape to someone, you could start by telling them that "the mean energy value is 35 MeV". This says where the peak is, but it doesn't tell them anything about how wide it is. How can we measure the width of a peak like this?

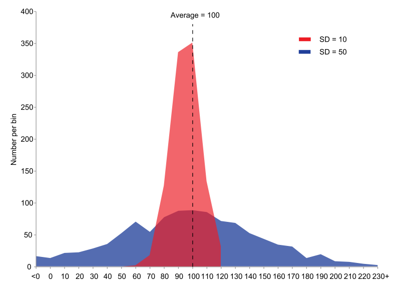


Figure 7.25: A comparison of histograms made from two samples, one with a small standard deviation and one with a large standard deviation. Both samples have the same mean value and contain the same number of data points.
Source: Wikimedia Commons

If the peak is wide, we might expect that a lot of data points would be far from the mean value. In the terms used in Equation 7.1, we might think about going through all of the points and adding up the values of $X_i - \overline{X}$. Unfortunately, we'd find that this sum is always zero, since some points are to the left of the mean and some to the right. It's possible to prove mathematically that the sum of all of these positive and negative distances will always add up to zero.

What we really want is just the distance from the mean, without worrying about whether it's positive or negative. Since the square of a real number is always positive, we might think about adding up the squares of the $X_i - \overline{X}$ values. Statisticians define a quantity called the "sample variance" that does just this. It's defined this way[4]:

[4] Why do we divide by $N-1$ instead of N? A simple explanation is that the variance is undefined if you have only one data point.

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \overline{X})^2 \tag{7.2}$$

where s^2 is the variance. For the example we've been working on, the units of the variance would be MeV^2 (energy squared). The square root of the variance is called the "standard deviation".[5] In our example, this has units of MeV, and it can be used to describe the width of the peak in Figure 7.13. The standard deviation tells us the "typical" distance between a data point and the mean value.

[5] This is another term that was introduced in the 1890s by Karl Pearson.

Figure 7.26 shows some data along with its arithmetic mean ($\overline{X}$) and standard deviation (s). The data we observe is just a sample of all the possible values we might see if we did an infinite number of measurements. Our data is called the "sample" and the collection of all possible values is called the "parent". Underneath it all, like the elephant inside the boa, is the true value that we're trying to estimate.

There's a practical problem with using Equation 7.2 in a computer program, though. Since it uses $\overline{X}$ (the mean value of the energy), we'd have to first loop through all of the energy values to calculate their mean, and then loop through them all again to calculate the variance.

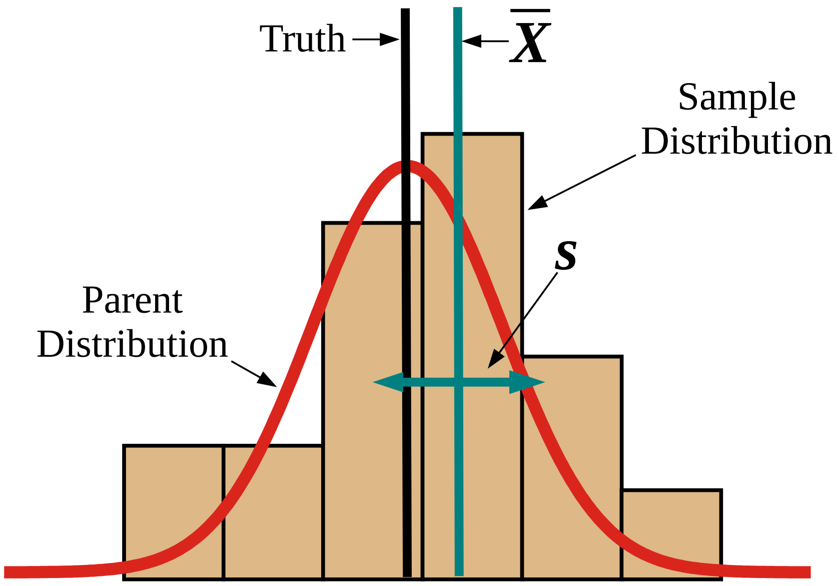


Figure 7.26: Sample distribution, parent distribution (the set of all possible measurements), and true value. $\overline{X}$ is the mean of the sample, and s is its standard deviation.

Fortunately, clever mathematicians have provided us with a shortcut to make things easier. It turns out that Equation 7.2 can be rewritten like this:

$$s^2 = \frac{1}{N-1}\left[\sum_{i=1}^{N} X_i^2 - \frac{1}{N}\left(\sum_{i=1}^{N} X_i\right)^2\right] \tag{7.3}$$

The right-hand sum in Equation 7.3 is the same one we're already using in Program 7.4. To find the variance we also need the left-hand sum, which is the sum of the *squares* of the values. Our program just needs to do one loop, and keep two sums: the sum of the values and the sum of their squares.

That's what Program 7.5 does with our `energy.dat` data. The program includes `math.h` at the top, since it uses the `sqrt` and `pow` functions. We've also added a new variable `sum2` to store the sum of the squares, from Equation 7.3. At the end of the program, we calculate the standard deviation and print it out.

Program 7.5: stddev.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  double energy;
  double mean;
  double stddev;
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  FILE *input;

  input = fopen("energy.dat","r");
  while ( fscanf( input, "%lf", &energy ) == 1 ) {
    sum += energy;
    sum2 += pow( energy, 2 );
    nvalues++;
  }

  mean = sum/nvalues;
  stddev = sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) );

  printf ("Number of values is: %d\n", nvalues );
  printf ("Mean value is: %lf\n", mean );
  printf ("Std. Dev is: %lf\n", stddev );

  fclose (input);
}
```

This program is an improved version of `mean.cpp` (Program 7.3) that prints out the standard deviation of the energy values. Changes from Program 7.3 are shown in bold.

We can apply the same technique to our ever-improving `hist.cpp` program, giving it the ability to print out the standard deviation as well as the mean value. That's what we do in Program 7.6.

Exercise 38: Finding the Standard Deviation

Create, compile, and run Program 7.6, a new version of `hist.cpp` that now prints the standard deviation. How large is this value in comparison with the width of the peak in Figure 7.13?

Program 7.6: hist.cpp, Version 3

This is an updated version Program 7.4. Changes from Program 7.4 are shown in bold.

```
#include <stdio.h>
#include <math.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 50.0;
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  const int nbins = 50;
  int bin[nbins];
  FILE *input;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &x ) == 1 ) {

    sum += x;
    sum2 += pow( x, 2 );
    nvalues++;

    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Std. dev. is %lf\n",
          sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) ) );
  printf ("# Nvalues = %d\n", nvalues );
}
```

Needed for `sqrt` and `pow`.

Add square of each value to `sum2`.

7.7. The "Normal" or "Gaussian" Distribution

The peak in Figure 7.13 is a bell-shaped curve. Curves like this occur very frequently in data. In fact, they occur so frequently that this shape is called the "Normal Curve". The German mathematician Carl Friedrich Gauss (1777-1855) was perhaps the first to appreciate the significance of it, so it's sometimes called a "Gaussian Curve".

Figure 7.27: Gauss is pictured on this German banknote. If you look closely you'll see a small picture of the Normal curve at the left.

Source: Wikimedia Commons

The ubiquity of this curve was a source of amazement to early statisticians, who saw it popping up everywhere: astronomical data, actuarial tables, agricultural data.

Why does this curve appear so often? Because of the "Central Limit Theorem", which says that any linear sum of random variables tends toward a Normal distribution, no matter what the distribution of the individual variables looks like.[6]

[6] Note that this means you can construct a pretty good Normal distribution just by adding together sufficiently many numbers pulled from any random distribution. For example, roll six dice and add their numbers together. Keep doing this and recording the sum each time. A histogram of the sums will look very similar to the Normal distribution.

The Central Limit Theorem is so important that it's called the "second fundamental theorem of probability". (The first is the Law of Large Numbers.)

The Normal curve can be expressed mathematically by the following equation:

$$P(x) = Ae^{-\frac{(x-\overline{x})^2}{2s^2}} \tag{7.4}$$

The curve reaches its maximum at $\overline{x}$, the mean value of x. The curve's width is controlled by s, the standard deviation. The height of the curve at its maximum is A.

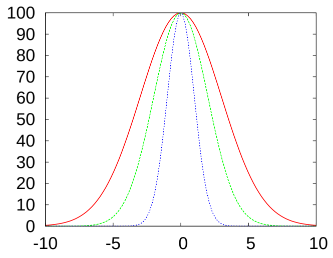


Figure 7.28: Three Normal curves with standard deviations of 3 (the widest), 2 and 1.

If we look at data that's bunched together in a Normal distribution, the standard deviation of the data gives us some quantitative information about the way the data is distributed. We know, for example, that about 68% of Normally-distributed data lies within one standard deviation away from the mean value. (See Figure 7.30.)

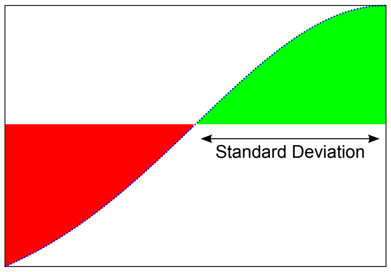


Figure 7.29: The standard deviation of a Normal curve is the horizontal distance from the midline to one of the points where the curvature changes from positive to negative.

If Program 7.6 tells us that the standard deviation of our energy data is 2.5 MeV and the mean is 35 MeV, that implies that 68% of our energy values fall between 32.5 MeV and 37.5 MeV. If we were telling someone about our measurements, we might say that the energy value we observed was 35 ± 2.5 MeV.

We also know that about 95% of the data lie within 2 standard deviations from the mean, and about 99.7% of the data are within 3 standard deviations.

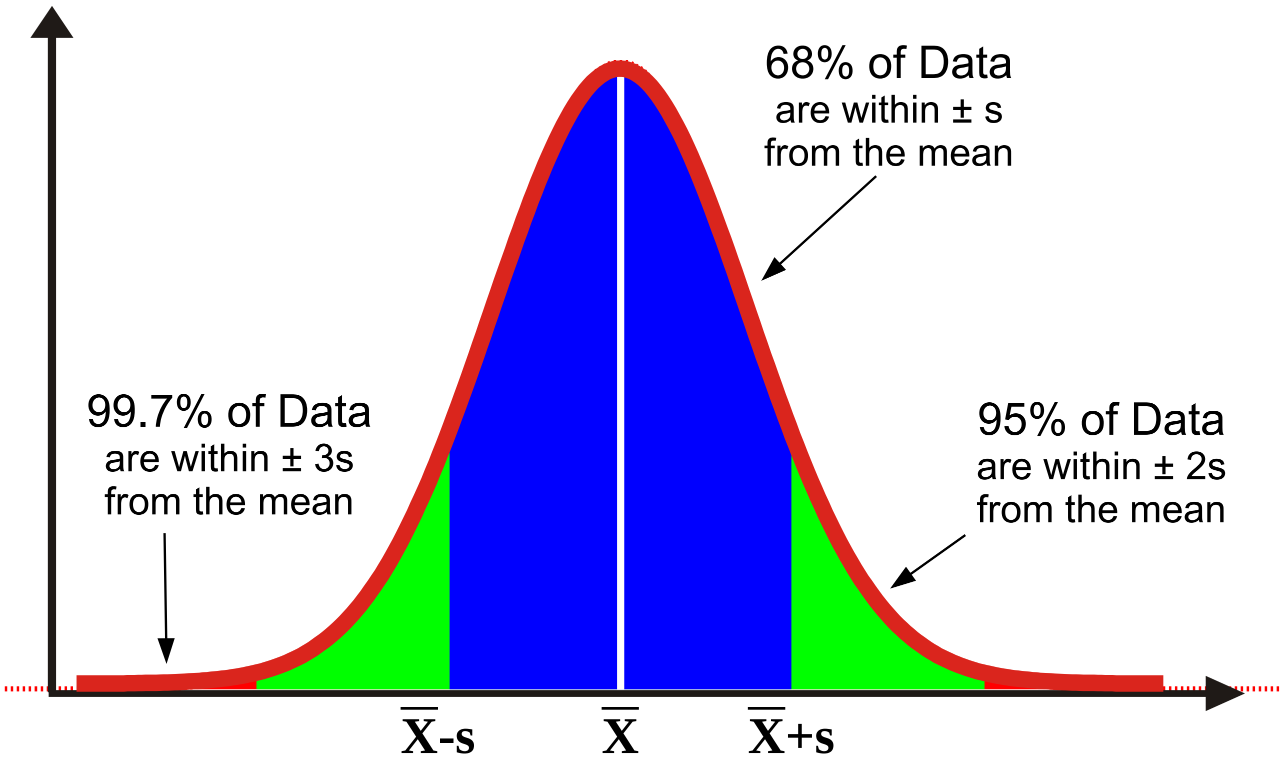


Figure 7.30: If data are distributed Normally, 68% of the values fall within one standard deviation from the mean. 95% of values are within two standard deviations, and 99.7% are within three standard deviations.

If you look at a Normal curve, you can find its standard deviation by locating the places where the curvature changes from positive (concave up) to negative (concave down). Mathematically, these points (called "points of inflection") are where the 2nd derivative of the function is zero. The standard deviation is the horizontal distance from the mean to either of these two points. (See Figure 7.29.)

Exercise 39: It's Only Fitting

We've seen that *gnuplot* can plot data, but it can also plot functions. Several functions, like sin(x), cos(x), and exp(x) are built into *gnuplot*, but you can also define your own functions. Try starting up *gnuplot* and typing the following:

```
p(x) = a*exp(-0.5*(x-m)**2/s**2)
s=2.5
m=35
a=10000
plot "hist.dat" with impulses, p(x)
```

The first line defines a new function `p(x)` that's just the

Normal curve given in Equation 7.4 above. The next three lines set the parameters: s is the standard deviation, m is the mean ($\overline{X}$), and a is the height of the peak.

The last line plots your histogram data from the file hist.dat and overlays a Normal curve on top of it. You can see that the shapes are similar, but the curve doesn't exactly match the data.

We could try adjusting the values of s, m, and a by hand to make the curve fit better, but *gnuplot* can do this for us automatically.

Type the following *gnuplot* commands:

```
fit p(x) "hist.dat" via s,m,a
replot
```

The first command tells gnuplot to adjust the parameters s, m, and a to make p(x) match the data in hist.dat. When it's done, it prints out a lot of information including the new values of the parameters. The second command tells gnuplot to re-do our last graph, which will now draw p(x) using the new parameters. Does it fit better now?

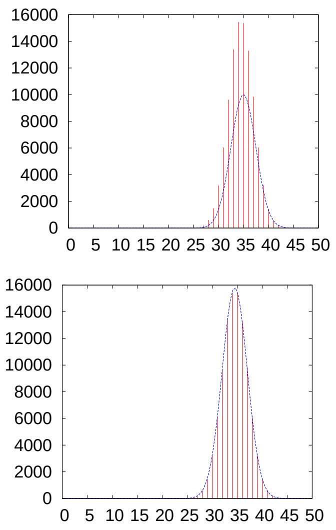


Figure 7.31: A Normal curve superimposed on our hist.dat data. The top graph shows a curve that doesn't quite match. The bottom graph shows the curve after we've asked *gnuplot* to adjust the parameters for the best fit.

But what about...?

In the data we've been looking at, each data point is some distance, d (positive or negative) from the mean value. The sample standard deviation, s, tells us how far a "typical" data point strays from the mean, but there are other ways we could choose to quantify a "typical" deviation. For example, we could look at the average absolute value of d.

The standard deviation has some nice properties, though. In particular, it has a natural relationship to the Normal distribution. As we saw above, 2s is the distance between the "points of inflection" (the places where the curvature goes from positive to negative) of the Normal distribution.

More importantly, statisticians tell us that the sample standard deviation is usually the best estimate of the standard deviation of the infinitely many data points we could possibly collect (the "parent population").

7.8. Exploring The Central Limit Theorem

In Chapter 2 we learned how to simulate rolling dice. For example, Program 2.4 generates a random number between 1 and 6, just like rolling a 6-sided die. Program 7.7, below, is an updated version that rolls a 6-sided die 1,000 times. If we used *gnuplot* to plot this program's output, we would see something like Figure 7.32.

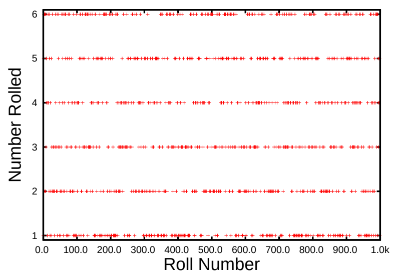


Figure 7.32: The output of `singledie.cpp` plotted using the *gnuplot* command `plot "singledie.dat"`

Notice that we see about the same number of rolls landing on each number, which is what we'd expect from a fair die (or a good random-number generator!). If we made a histogram of the values obtained from rolling a single 6-sided die, it might look like Figure 7.33. As you can see, each value has an equal probability of turning up.

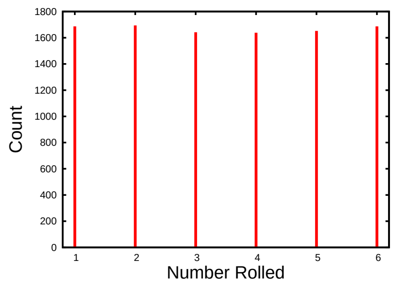


Figure 7.33: A histogram of the values obtained by rolling a single 6-sided die 1,000 times.

Program 7.7: singledie.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int roll;
  int min = 1;
  int max = 6;
  int nvals;
  int i;
  double x;

  nvals = max - min + 1;
  srand(time(NULL));

  for ( roll=0; roll<1000; roll++ ) {**
      x = rand()/(1.0 + RAND_MAX);
      i = min + (int)(nvals*x );
    printf( "%d\n", i );
  }
}
```

Some dice games require you to roll two or more dice at once, and add up their numbers. Let's modify Program 7.7 so that it rolls twelve dice at once, instead of just rolling one die. We'll need to add an extra loop and a couple of variables to do that. The result is Program 7.8.

Program 7.8: multidice.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int roll, die;
  int min = 1;
  int max = 6;
  int nvals;
  int i, sum;
  double x;

  nvals = max - min + 1;
  srand(time(NULL));

  for ( roll=0; roll<1000; roll++ ) {
    sum = 0;
    for ( die=0; die<12; die++ ) {
      x = rand()/(1.0 + RAND_MAX);
      i = min + (int)(nvals*x );
      sum += i;
    }
    printf ( "%d\n", sum );
  }
}
```

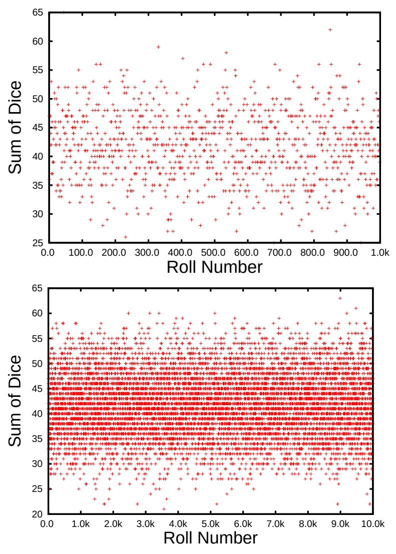


Figure 7.34: The upper figure shows the output of Program 7.8 plotted using *gnuplot*. The bottom figure shows what it would look like if we increased the number of rolls from 1,000 to 10,000.

If we plotted the output of Program 7.8 we'd see something like the upper graph in Figure 7.34. Notice that now the values aren't spread evenly any more. When we roll twelve dice and add them up, their sum is most likely to be somewhere around 42. This is even more apparent in the bottom graph of Figure 7.34, where we've increased the number of rolls to 10,000.

To get a better sense of the distrubution of the values, let's make a histogram of them. We can do that by combining Program 7.8 with Program 7.2. The result is Program 7.9 below. (Notice that we've set the number of dice rolls to 10,000 now.) If we ran this program and plotted its output using the *gnuplot* command

```
plot "dicehist.dat" using 1:4 with impulses"
```

we'd see something like Figure 7.35.

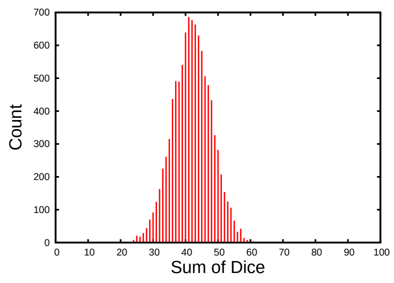


Figure 7.35: A histogram of our dice roll sums, created by Program 7.9, using the following *gnuplot* command:
`plot "dicehist.dat" using 1:4 with impulses`

Program 7.9: dicehist.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int roll, die;
  int min = 1;
  int max = 6;
  int nvals;
  int i, sum;
  double x;
  const int nbins = 100;
  int bin[nbins];
  int binno, overunderflow = 0;
  double xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 100.0;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  nvals = max - min + 1;
  srand(time(NULL));

  for ( roll=0; roll<10000; roll++ ) {
    sum = 0;
    for ( die=0; die<12; die++ ) {
      x = rand()/(1.0 + RAND_MAX);
      i = min + (int)(nvals*x );
      sum += i;
    }
    binno = (sum-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
}
```

Figure 7.35 shows that a value of 42 appears almost 700 times when we sum up our twelve dice. The farther away from 42 we get, the less likely we are to see a given sum. The distribution of values looks like a Gaussian or Normal distribution, as described in Section 7.7. As we noted in that section, this effect is known as the "Central Limit Theorem". It tells us that the sum of several random variables tends to take on a Normal distribution.

Even though the distribution of numbers we get from each die is flat, as shown in Figures 7.32 and 7.33, the sum of these numbers approaches a Normal distribution (see Figures 7.34 and 7.35).

The fact that our observed values are centered around 42 makes sense too. Each 6-sided die gives a value between 1 and 6, so the average value we should get from a single roll of a die is $(1+6)/2 = 3.5$. That means that the average value for the sum of twelve dice should be $12 \times 3.5 = 42$.

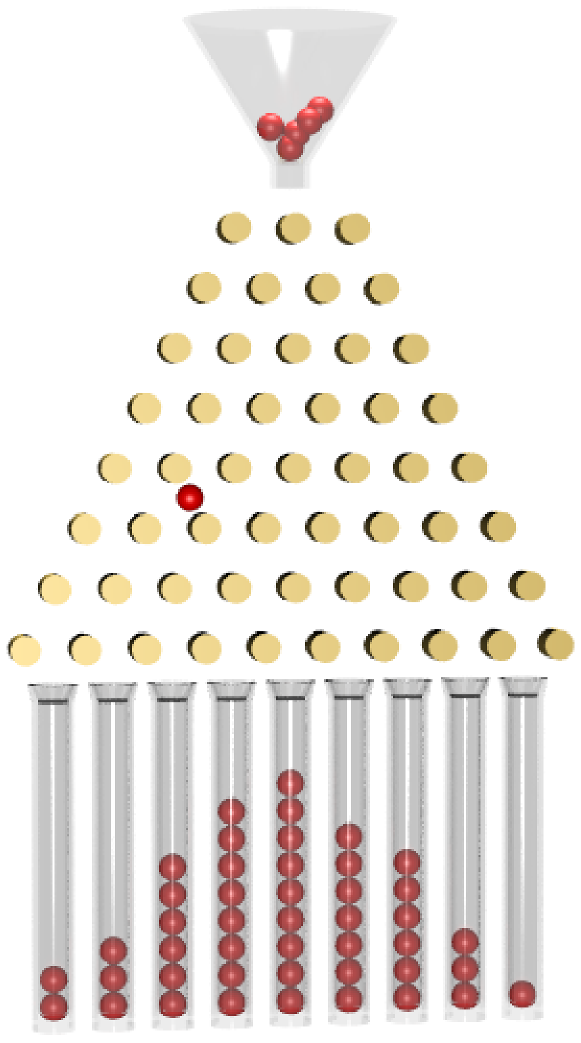


Figure 7.36: Beans bounce off of pegs as they roll down a "Galton Board". At the bottom they fall into bins, like histogram bins. The sum of all the random left and right bounces experienced by the beans results in an approximately Normal distribution.

Source: Wikimedia Commons

7.9. Analyzing Multi-Column Data

Statistics began as a study of demographic data (numerical data about populations), so let's take a look at some "people data" before we finish. The US constitution mandates that a census be taken every ten years, and the task of collecting and analyzing data falls on the US Census Bureau.

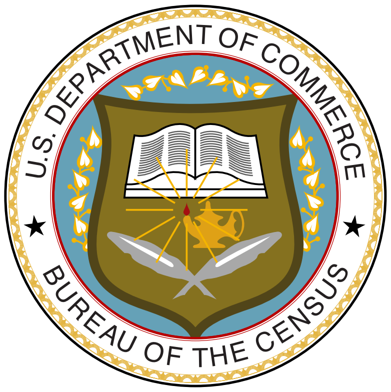


Figure 7.37: The US Census Bureau is charged with conducting a decennial census.

Source: Wikimedia Commons

Census takers collect a lot of data for each household they visit. They might record the number of children, the number of bedrooms in the house, the amount paid monthly in rent, and so forth. We might store the data for each household in a row, with a column for each quantity that was recorded. The result would look something like this:

```
0        1        3     10700        2        2        0
0        1        4      7800        2       40        0
0        1        3     64200        2      130        0
0        1        3        -1     2400      210        0
0        0        1        -1        2       10      780
0        1        3     44600        2       90     1905
...
```

In the following sections, we'll be constructing a program that can read a data file from the US Census Bureau that contains information about 1,285,588 households. The file has seven columns of integers for each household. Each column represents a different measurement:

Column	Description
0	Number of related children in household
1	Lot size
2	Number of bedrooms
3	Family income
4	Annual fuel cost
5	Monthly gas cost
6	Monthly rent

Figure 7.38: Taking the census could be a dangerous job. Consider the plight of a census taker asked to survey these denizen's of an 1890 New York "Bandit's Roost". This picture was taken by Jacob Riis, who prowled New York's tenements accompanied by then-Police-Commissioner Theodore Roosevelt, documenting "How the Other Half Lives" (the title of Riis's best-known book).

Source: Wikimedia Commons

The file we've been analyzing, `energy.dat`, contains only one column of data. Only one measurement (the amount of energy deposited) was recorded for each particle that passed through the detector. The census taker, on the other hand, takes several measurements for each family. Let's look at how we might modify our earlier programs to allow them to read such multi-column data.

One way to do it would be to replace our single variable (`energy`, in the earlier programs) with an array. The number of elements in the array will need to match the number of columns in the data file.

Program 7.10 uses this strategy to analyze data from a seven-column data file. In order to read each row, it loops through the seven elements of the array `data`. The new variable `field` specifies which column of the data we want to analyze, and the new program gives the variable `x` the value of `data[field]`. (Program 7.10 sets `field` to 0, but it could be set to any value from 0 to 6.) The new program also changes the name of the data file from `energy.dat` to `census.dat`.

Because Program 7.10 uses a "`for`" loop to read multiple items from each line, we can no longer use a simple `break` when we reach the end of the file, as we did when reading `energy.dat`. Remember from Chapter 4 that the `break` statement only stops the loop it's in. If we used a `break` inside the "`for`" loop of Program 7.10 when we get to the end of the file, the `break` would only stop the "`for`" loop. It wouldn't stop the outer, enclosing "`while`" loop, so the program would keep trying (and failing) to read lines forever.

There are several ways we could handle this. One of them is to use C's "`goto`" statement. A `goto` statement jumps immediately to another location in your program. You might think that this could be a highly dangerous thing to do, and you'd be right. There's a superstition among programmers that says `goto` should never be used, but experts agree[7] that `goto` is sometimes the best solution in one specific case: when your program needs to break out of nested loops like the ones we have in Program 7.10.

[7] See the "exception" under ES.76 in the *CPP Core Guidelines*: https://github.com/isocpp/CppCoreGuidelines.

Notice the line in Program 7.10 that just says "`done:;`". This is called a "label". A label can be any word, followed by a colon[8], on a line by itself. Labels don't do anything. They just mark a spot in your program. Think of them as bookmarks. When we say `goto done;` we're telling the program to jump to the label named "`done`". When Program 7.10 gets to the end of the file it's reading, the `goto` statement jumps out of the nested loops and continues below the `done:;` label.

[8] Notice that this is a colon, not a semicolon. In the examples in this book we'll also put a semicolon after the label, just as we do with other C statements.

Used in this way, `goto` statements can be a safe and efficient way to break out of nested loops. If you think of `goto` as a kind of "super-`break`" it's quite unlikely that you'll be eaten by a velociraptor[9]... but remain vigilant.

[9] See https://xkcd.com/292.

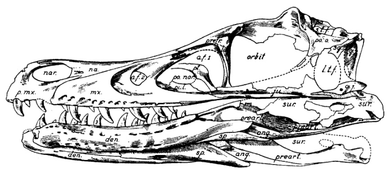


Figure 7.39: Skull of *Velociraptor mongoliensis*.
Source: Wikimedia Commons

Program 7.10: census.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 50.0;
  const int nbins = 50;
  int bin[nbins];
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  FILE *input;
  int field=0; // Select column 0 from data.
  double data[7]; // Add "data" array.

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "census.dat", "r" );
  while ( 1 ) {
               for ( i=0; i<7; i++ ) {
                if ( fscanf( input, "%lf", &data[i] ) != 1 ) {
                  goto done;
                }
               }

              x = data[field]; // Choose which column.

              sum += x;
              sum2 += pow( x, 2 );
              nvalues++;

              binno = (x-xmin)/binwidth;
              if ( binno < 0 || binno >= nbins ) {
                overunderflow++;
                continue;
              }
              bin[binno]++;
  }
 done:;
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Field number %d\n", field);
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Std. dev. is %lf\n",
          sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) ) );
  printf ("# Nvalues = %d\n", nvalues );
}
```

Get 7 items from each line

Read lines from file

Jump out of nested `for` and `while` loops when we reach the end of the file

7.10. Filtering Data

Census takers can't always collect all measurements from every household. Sometimes a measurement just doesn't apply. What's the monthly rent on a house that's not being rented? What's the annual household income for an unoccupied house? Our data sets will sometimes contain special values that indicate "Not Applicable". We might not want to include these values in our averages, or show them on our histograms. We could think of this a "filtering" our data.

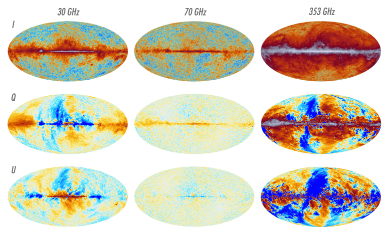


Figure 7.40: The *planck* spacecraft examined the microwave radition leftover from the Big Bang. The figure above shows analyses of *planck's* data with several different filters applied.
Source: *Planck Mission, European Space Agency*

In the census data we're going to look at, these special values are indicated by zeros or negative numbers. By making a couple of changes, we can cause our program to ignore such values. First, we want to look for special values whenever we read a line from our data file. When we find one, we want to skip that line and just go on to the next. We can accomplish this by adding the following section before the "`sum +=`" in Program 7.10:

```
if ( x <= 0 ) {
  continue; // Ignore zeros and negatives.
}
```

We'll probably want to know how many values were ignored (or, equivalently, how many *weren't*). It would be a good idea to add a line like the following at the end of the program, along with the other numbers we print out:

```
printf ("# Saw %d data values\n", nvalues);
```

The variable `nvalues` tells us how many data points we really analyzed, not counting those we filtered out.

We can modify our data analysis program to filter our data any way we like. We might even look at the other columns on each line when deciding whether or not to use the data on that line. For example, maybe we're interested in the number of children per household, but only want to look at families paying more than $500 per month in rent.

7.11. Setting Analysis Parameters

Program 7.10 explicitly chooses a particular column to analyze by setting the `field` variable. It would be nice if the program asked us which column we wanted to use. We can easily add a section somewhere before our `while` loop to do this:

```
printf ( "Pick a column [0-6]: " );
scanf ( "%d", &field );
```

If we pick a different column, we might also want to use a different bin width. (This is the width of the bins into which we drop our "virtual marbles" while making a histogram.) A bin width of 1 is fine if we're looking at the number of children per household, but we might want a width of 10,000 if we're looking at annual household income. An income difference of $1 isn't very interesting, but $10,000 would be. We could add another section to our program for setting the bin width:

```
printf ( "Enter bin width: " );
scanf ( "%lf", &binwidth );
```

If we specify `binwidth`, we can calculate the value of `xmax` (the maximum value we're interested in) like this:

```
xmax = binwidth*nbins + xmin;
```

Let's leave the lower end of our range (`xmin`) at zero, since the data in each colum of our data set includes some small values.

We could add any number of similar sections to the beginning of a data analysis program, to allow us to set any parameters we need. Maybe we want to analyze only data for households with annual incomes in a given range (say, between $20,000 and $30,000). In that case, the program could ask for `minincome` and `maxincome`, and use those variables when filtering the data.

Figure 7.41: Some members of the author's family, *circa* 1939.

7.12. Using stderr

If our program asks the user for parameters, we introduce another complication: some of the program's output (the request to "Enter bin width", for example) needs to go to the computer's display, so the user can see it, but other output (the histogram data) needs to be written into a file so we can plot it with *gnuplot*. If we just type `./census > output.dat` then the user won't see the requests for entering parameters, and the program will just sit forever waiting for them.

There are several ways to solve this problem. For one, we could use `fprintf` to write the histogram into a file instead of sending it to the display, as we saw in Chapter 5.

Let's look at another way of doing it, though. As we saw in Chapter 5, we can open a file with `fopen` like this:

```
FILE *output;
output = fopen("output.dat","w");
```

The variable `output` is a "file handle" that we can use later with `fprintf`. We can open as many files as we want, and choose which file handle to use when we want to print something into one of them.

It turns out that three file handles are automatically created whenever you run your program. These are named `stdout`, `stderr`, and `stdin`. The `stdout` file handle doesn't point to a real file. Instead, it points to your display. The `printf` statement uses this file handle whenever it prints something. The statement `printf("Hi!");` is just equivalent to `fprintf(stdout,"Hi!");`.

When you type a command like "`./census > output.dat`" the computer disconnects `stdout` from your display and connects it to the file `output.dat` instead. This makes the output of any `printf` statements go into the file instead of to your screen.

The `stdin` file handle points to your keyboard. The statement `scanf("%d", &i);` is the same as `fscanf(stdin,"%d", &i);`.

The third predefined file handle, `stderr`, also points to your display, but it's intended to be used for errors and warnings. Imagine, for example, that you've typed "`./census > output.dat`" but your program crashes with a Segmentation Fault error. The error message should be sent to your display, not to the file. Error messages like this are sent to `stderr`, which is still connected to your display.

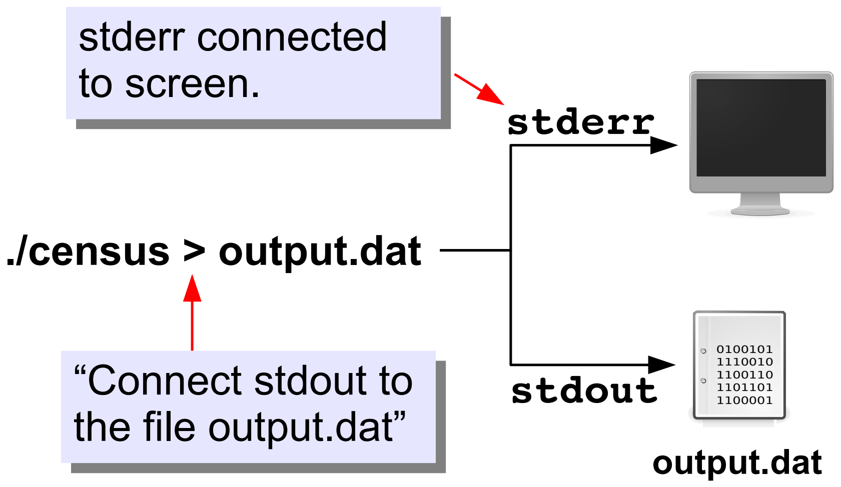


Figure 7.42: The predefined file handles `stdout` and `stderr` both start out pointing at your display, but they can be redirected elsewhere.

We can use `stderr` for our own purposes, too. We want our "Enter bin width" message to go to the display even if we've redirected the program's output into a file. All we need to do is send those messages to `stderr` instead of `stdout`. We can do that by modifying a couple of `printf` statements:

```
fprintf ( stderr, "Pick a column [0-6]: " );
scanf ( "%d", &field );

fprintf ( stderr, "Enter bin width: " );
scanf ( "%lf", &binwidth );
```

Instead of `printf`, we use `fprintf` to send these messages to `stderr`.

But what about...?

Are there other ways we could split the program's output between display and file? Why yes, I'm glad you asked!

One way involves the third predefined file handle, `stdin`. This normally points to your keyboard, and it's used by `scanf` whenever it reads some input. However, just like `stdout`, you can disconnect `stdin` from the keyboard and connect it to a file instead. If you did that, you could cause your program to read stored answers from a file, rather than having to type them in at the keyboard. Figure 7.43 shows how to do this, using the "<" symbol on the command line. If we did it this way, the program would expect to find two numbers in the file `input.dat`: the column number we want to analyze, and the bin width.

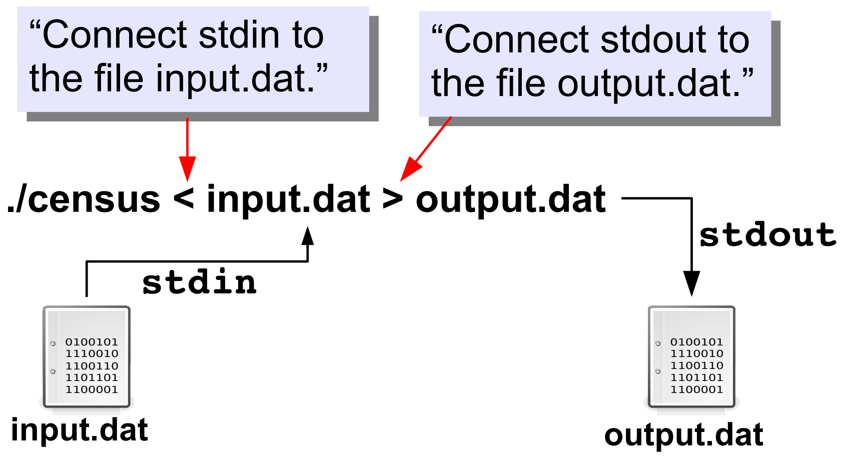

“Connect stdin to the file input.dat.”
“Connect stdout to the file output.dat.”
./census < input.dat > output.dat
stdout
stdin
0100101 1110010 1100110 1101101 1100001
input.dat
0100101 1110010 1100110 1101101 1100001
output.dat

Figure 7.43: We could redirect both `stdout` and stdin if we wanted to. The “<” on the command line means “Read input from this file”, just as the “>” means “Write output to this file”. If the program asks us some questions, we can save our answers in the file `input.dat`. The program will read them from there, instead of waiting for us to type them.

7.13. Improved Analysis Program

Program 7.11 is an improved analysis program that incorporates all of the improvements we've talked about in the preceding sections. When we run the program it asks us which column (0 through 6) we want to analyze, then it asks us what bin width we want to use. The histogram data is sent to the display, unless we redirect it to a file.

Figure 7.44 shows some results from the program. To plot the income graph, for example, we did this:

```
./census > income.dat
```

and then answered the questions:

```
Pick a column [0-6]: 3
Enter bin width: 10000
```

The output file was graphed with *gnuplot* as with our earlier histograms.

Notice that the data here aren't bunched into Normal distributions like the energy data. In the energy case, we were making many measurements of the same value (the energy of some kind of particle striking our detector). The only variations in these measurements were due to random factors.

The census data, on the other hand, is inherently different from one household to another. The distribution of values could give us some real information about people's lives. Nonetheless, we can still calculate the mean values of things like income, and calculate the standard deviation of our data sample. The standard deviation still tells us something

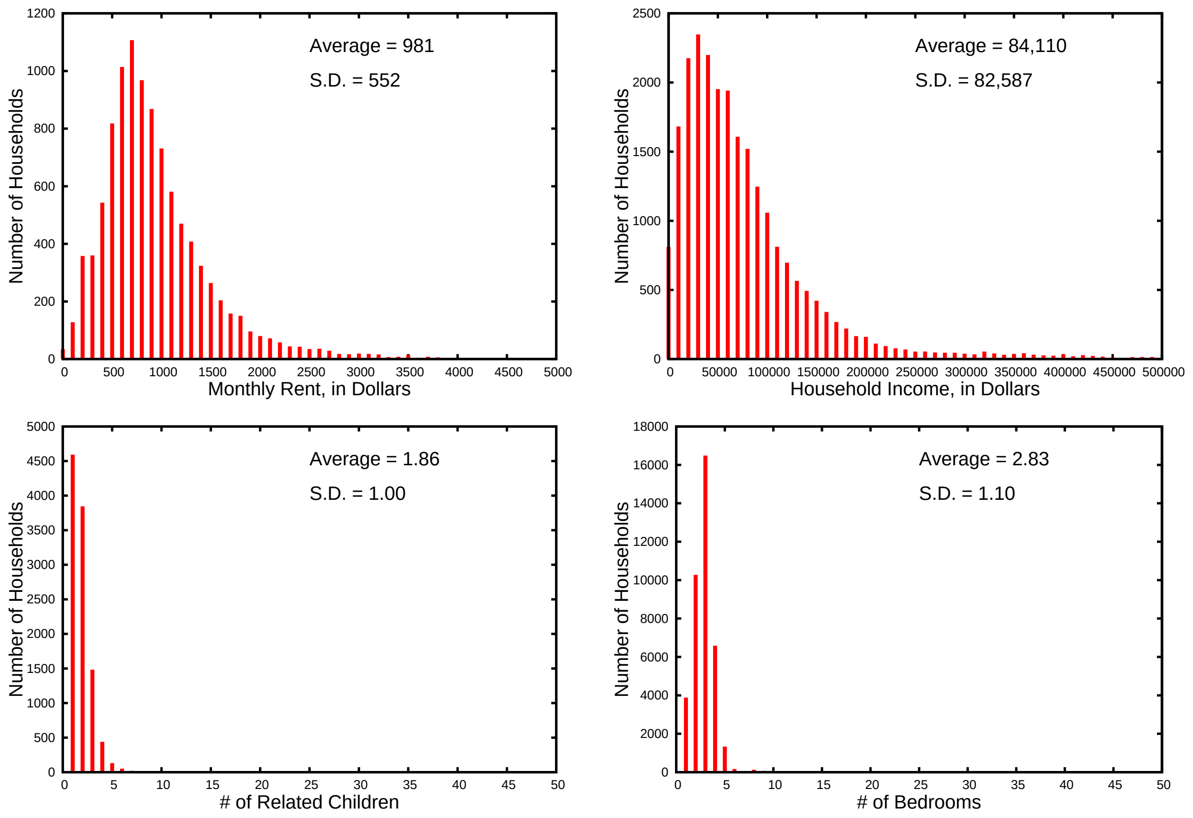


Figure 7.44: Some results from Program 7.11, plotted with *gnuplot*. Bin width was set to 10,000 for the income graph, 100 for the rent graph, and 1 for the bedrooms and children graphs.

about the width of the distribution, as it did with the energy data, even though the income distribution is far from Normally-distributed.

Exercise 40: Little Pink Houses

For this exercise you'll need a copy of the file `census.dat`. You'll find instructions for obtaining it in Appendix C.3 on page 624.

First, examine `census.dat` with *gnuplot*. Start *gnuplot* and type the command:

```
plot "census.dat" using 4
```

gnuplot numbers columns starting with 1, so this should display a graph of household income similar to Figure 7.45. Note the bar of negative values representing special cases that our analysis program will ignore.

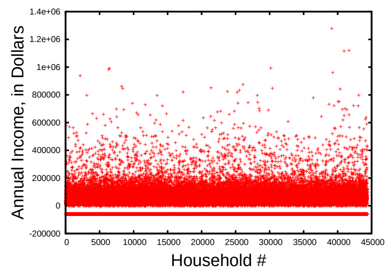


Figure 7.45: The output of the *gnuplot* command `plot "census.dat" using 4`.

Now exit from *gnuplot* and compile Program 7.11 (the new `census.cpp`, on Page 254). Run the program like this:

```
./census > income.dat
```

Select the income by choosing column number 3 (the program starts numbering the columns with 0). Use a bin width of ten thousand.

Now start up *gnuplot* again and ask it to plot the results of your analysis:

```
plot "income.dat" using 1:4 with boxes
```

By saying "`using 1:4`" we tell gnuplot to use column 1 (the smallest value in each bin) as the value on the x axis, and column 4 (the number of "virtual marbles" in each bin) as the y value. The graph shows us how many households are in each income range.

If you have time, try plotting other columns from the `census.dat` file and analyzing them.

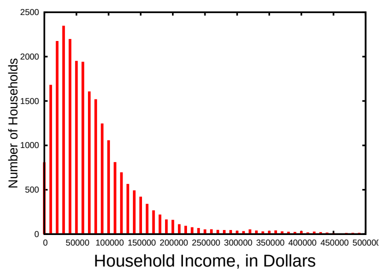


Figure 7.46: The income histogram produced by our analysis program.

Program 7.11: census.cpp, Version 2

```
#include <stdio.h>
#include <math.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0;
  double xmax;
  const int nbins = 50;
  int bin[nbins];
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  FILE *input;
  int field=0;
  double data[7]; // Add "data" array.

  fprintf ( stderr, "Pick a column [0-6]: " );
  scanf ( "%d", &field );

  fprintf ( stderr, "Enter binwidth: " );
  scanf ( "%lf", &binwidth );

  xmax = binwidth*nbins + xmin; // Calculate xmax from xmin and binwidth.

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "census.dat", "r" );
  while ( 1 ) {
    for ( i=0; i<7; i++ ) {
      if ( fscanf( input, "%lf", &data[i] ) != 1 ) {
        goto done;
      }
    }

    x = data[field]; // Choose which column.

    if ( x <= 0 ) {
      continue; // Ignore zeros and negatives, since they're special.
    }

    sum += x;
    sum2 += pow( x, 2 );
    nvalues++;

    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
 done:;
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Field number %d\n", field);
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Std. dev. is %lf\n",
          sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) ) );
  printf ("# Nvalues = %d\n", nvalues );
}
```

7.14. Conclusion

In this chapter we've looked at some basic techniques for doing statistical analysis of data with computer programs. Histograms and calculations of the mean and standard deviation are primary tools for data analysis in the sciences.

The details can vary greatly, but the outline of most data analysis programs will look much like Figure 7.47. We've discussed each of these steps as we developed and improved our census analysis program.

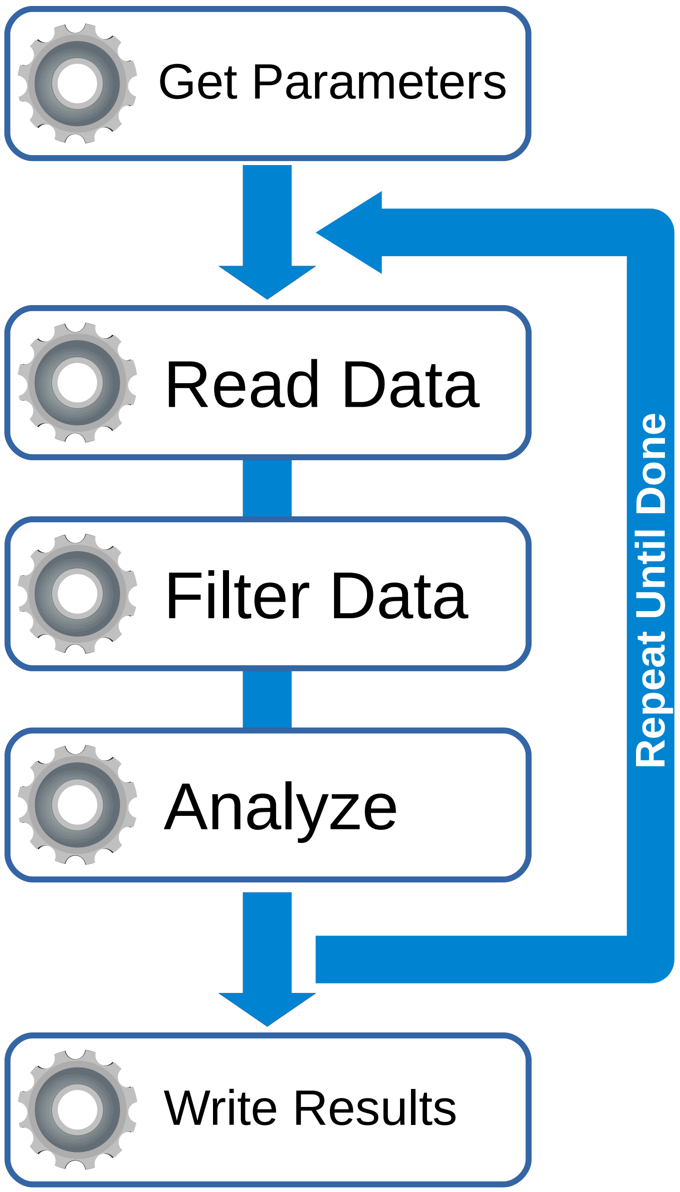


Figure 7.47: The figure above shows an outline of a typical data analysis program.

Practice Problems

1. Write a small program named `listmean.cpp` that finds the mean value of a list of numbers. Start out with an array of numbers, like this:

   ```
   double x[10] = {0,1,2,3,4,5,6,7,8,9};
   ```

   Use a "`for`" loop to go through the elements of the array, adding them up. At the end of the program, print out the mean value of these numbers.

2. The "mean" that we've talked about in this chapter is the "arithmetic mean". There are other kinds of mean value that we could calculate. One of them is called the "geometric mean". To find the geometric mean of a set of numbers, multiply them together and take the n-th root of their product, where n is how many numbers are in the set. For example, if we have the numbers 4, 5, and 6, their geometric mean would be:

   $$\sqrt[3]{4\times5\times6} \quad \text{or, alternatively} \quad (4\times5\times6)^{1/3}$$

   Write a program named `geomean.cpp` that calculates the geometric mean of these nine numbers:

   ```
   double x[9] = {1,2,3,4,5,6,7,8,9};
   ```

   Hints:

   - You can use the `pow` function to find the n-th root. For example, the 4th root of 38 would be `pow( 38, 1.0/4 )`. Note that it's important to say `1.0/4` instead of `1/4`, because the latter would tell the computer that you wanted to trim the decimal places off of the result.
   - When summing up a bunch of numbers we start with `sum = 0.0` and add each number by saying `sum += x`. When multiplying a bunch of numbers, you might start by saying `product = 1.0`, then multiply by each number by saying `product *= x`.

3. When you roll two six-sided dice the sum of their values can be any number between 2 and 12. We know that some combinations are more unlikely than others. For example, you're less likely to roll a 12 than a 7. Let's see if we can quantify how likely each possible combination is.

   Write a program named `2dice.cpp` that rolls two dice 1,000,000 times and counts how many times each possible sum occurs. Each time we roll, we'll get two random numbers between 1 and 6. If you have an `int` variable named `die1` for the first die, you can generate a random value for it like this:

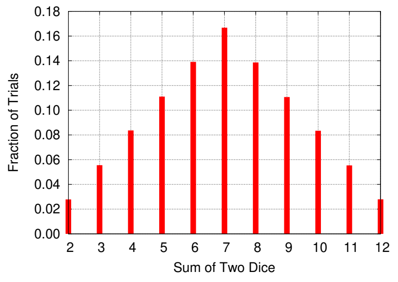


Figure 7.48: How often do you roll a particular sum? It looks like 7 comes up most often – about 17% of the time, and 2 and 12 are least common – about 3% each.

```
die1 = 1 + 6.0*rand()/(1.0+RAND_MAX);
```

To record the results of the rolls, make a histogram using a 13-element integer array like this:

```
int bin[13];
```

Make sure you set all of the elements of this array to zero at the beginning of the program. Use the elements of the array to record how many times you saw a particular sum. For example, `bin[7]` will hold the number of times you rolled a 7. (Notice that we won't use `bin[0]` or `bin[1]` since we can never get a sum of zero or one, but that's OK. We'll just ignore those bins.)

After you've done all the rolls, your program should have a loop that prints the contents of elements 2 through 12 of `bin`. The format should be:

```
printf( "%d %d %lf\n", i, bin[i], bin[i]/1000000.0 );
```

That will print the bin index, the contents of that bin, and the fraction of our 1,000,000 rolls that ended up with that sum. If you run the program like "`./2dice > 2dice.dat`" and then plot the output using *gnuplot* you should see something like Figure 7.48.

4. Using Program 7.5 as a starting point, create a program called `calcstats.cpp` that prompts the user to enter numbers, one at a time, and then prints out the mean value and standard deviation of the numbers entered.Make sure the program can accept numbers that have decimal places.

   You'll need to think about how the user can let the program know that he/she is finished entering numbers. If you only allow positive numbers, you could ask the user to enter "`-1`" to stop the program, but then you'd be unable to enter -1 as a number! There's a better way to do it. Remember that `scanf` and `fscanf` return a value that tells you how many numbers were successfully read. That means that if you use `scanf` in a "`while`" loop just like we used `fscanf` in Program 7.1, the loop will stop if the user enters anything that's not a number[10]. Use this trick in your program. **Hint:** You won't need to open or close any files. Just use `scanf` in the way we've been using `fscanf` in programs like Program 7.1.

My first calculator was one of these. It was a present from my Dad in the early 1970s. It was also the first calculator Texas Instruments made. Being a kid who struggled with long division, I loved this device!
Source: Wikimedia Commons

5. Imagine an inebriated person standing beside a lamppost. He wants to get home, so he starts walking, but each time he takes a step it's in a different, random, direction. How far away from the lamppost will he be, on average, after 100 steps?

   This is a well-known problem in mathematics called "the drunkard's

[10] The usual convention is to ask the user to press Ctrl-D when they're done entering data. That's a special character that represents the "End Of File" (EOF).

walk". As you can see from Figure 7.49, the distance travelled by the drunkard can vary a lot from one trial to the next. If he walked in a straight line, he'd end up 100 steps away from the lamppost, but most of these random paths leave him much closer.

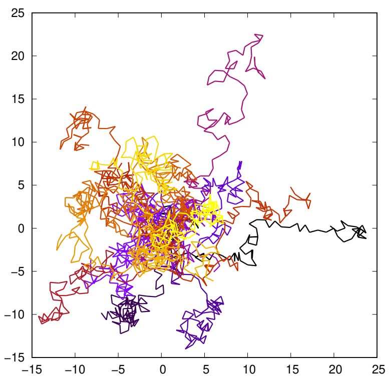


Figure 7.49: The paths of 20 drunken people, each shown in a different color. The lamppost is at the origin. The distance units are "steps", which we assume to be of equal length. Each person has taken 100 steps.

Write a program named `randomwalk.cpp` that simulates 1,000 of these 100-step paths and prints out the average final distance from the lamppost. (Measure all distances in "steps", which we assume to be of equal length.) Make sure you use `srand(time(NULL))` to choose a different "seed" for the random number generator each time you run your program.

Here are a few hints to help you:

- You'll need a pair of nested loops: An outer loop for each path, and an inner one for each step.
- Keep track of the person's position with a couple of variables, `x` and `y`. Remember to set them both back to zero at the beginning of each path.
- Every time the person takes a step, generate a random angle like this:

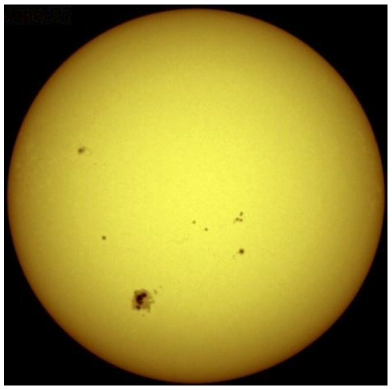
  ```
  angle = 2.0*M_PI*rand()/(1.0+RAND_MAX);
  ```

  then add `cos(angle)` to `x` and `sin(angle)` to `y` to get the person's new position.
- At the end of each path, calculate the final distance from the origin like this:

  ```
  distance = sqrt( x*x + y*y );
  ```

  and add that to a sum of all of the distances, for use later when you compute the mean distance.
- To check your work: your program should find that the average final distance is about 8.86 steps. This is 0.886 × the square root of the number of steps.

This kind of random motion is common in nature, making the drunkard's walk an important problem in science. In physics, for example, it describes the random motions of molecules in a gas, or the motion of impurities jumping across a surface. In chemistry it describes the shapes of polymers. In economics, random walks can even explain some of the variation in stock prices.



Photons generated in the center of the sun follow a "drunkard's walk" path as they make their way to the sun's surface. This twisty path can include trillions of steps and take as much as a million years to complete.

Source: *Wikimedia Commons*

6. Modify Program 7.11 so that it asks the user for two new parameters:

`maxincome` and `minincome` (maximum and minimum income) as described on Page 248. Use these in the filter section of the program (the section where we currently check to see if `x` is less than or equal to zero). Skip the current row of data if the following is true:

```
data[3] < minincome || data[3] > maxincome
```

7. Write a program named `liststats.cpp` that reads a list of numbers, one number per line, from a file named `liststats.dat` and prints how many numbers were read, their mean, and their standard deviation.

   You can test your program with data generated by the three small programs at the end of this chapter: `urand`, `nrand`, and `prand`. For example, to use `nrand` to create a `liststats.dat` file, you could type:

   ```
   ./nrand > liststats.dat
   ```

   You should find that the numbers generated by `nrand` have a mean of about zero, and a standard deviation of around 1. The numbers generated by `prand` should have a mean of about 3 and a standard deviation of about $\sqrt{3}$.

8. Write a program named `arraystats.cpp` that defines an array of numbers like this:

   ```
   const int nvalues = 10;
   double x[nvalues] = {0,1,2,3,4,5,6,7,8,9};
   ```

   Have the program calculate and print the mean and the standard deviation of these numbers.

9. Write a program named `bufferstats.cpp` that defines an array of numbers like this:

   ```
   const int n = 1000;
   double x[n];
   ```

   Have the program open a file named `bufferstats.dat` that will contain one number per line. Use a "`for`" loop to read the first 1,000 numbers from the file, and put them into the 1,000 elements of the array `x`. Close the file, then have a second "`for`" loop that calculates the mean and standard deviation of the numbers. Print these values at the end of the program, in a friendly easy-to-read format.

You can test your program with data generated by the three small programs at the end of this chapter: `urand`, `nrand`, and `prand`. For example, to use `nrand` to create a `bufferstats.dat` file, you could type:

```
./nrand > bufferstats.dat
```

You should find that the numbers generated by `nrand` have a mean of about zero, and a standard deviation of around 1. The numbers generated by `prand` should have a mean of about 3 and a standard deviation of about $\sqrt{3}$.

10. The following program tests how fast your computer can create files. The program repeatedly opens a file ("jittertest.dat"), writes into it, then closes it. As it's doing this it keeps track of how long each open/write/close cycle takes (in microseconds).

Program 7.12: jitter.cpp

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
long epoch;
void startclock(){
  struct timeval t;
  gettimeofday(&t, NULL);
  epoch = t.tv_sec * (int)1e6 + t.tv_usec;
}
int microtime(){
  struct timeval t;
  gettimeofday(&t, NULL);
  return( (int)(t.tv_sec * (int)1e6 +
        t.tv_usec - epoch) );
}

int main () {
  int i;
  int tstart, delay;
  FILE * output;

  startclock();

  for ( i=0; i<1000; i++ ) {
    tstart = microtime();
    output = fopen( "jittertest.dat", "w" );
    fprintf( output, "Testing...\n" );
    fclose( output );
    delay = microtime() - tstart;
    printf ( "%d\n", delay );
  }
}
```

The top part of the program (everything above `int main()`) is just some magic that lets us measure time to microsecond accuracy. Some of this will become clear in Chapters 9 and 12, but for now, don't worry about how it works.

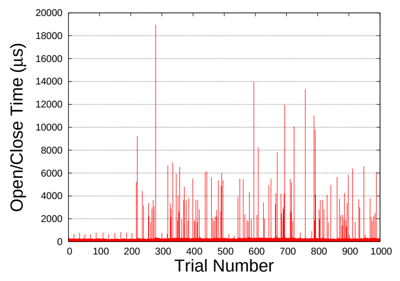


Figure 7.50: If you graphed the numbers from the `jitter` program, they might look like this. As you can see, sometimes an open/close takes a lot longer than usual.

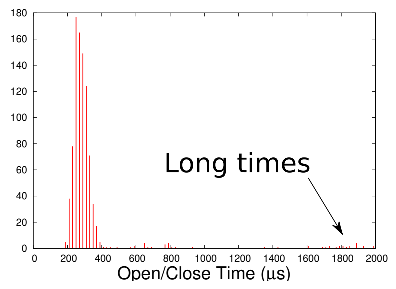


Figure 7.51: If you made a histogram from the numbers, it might look like this. Note that most of the data are clustered around 300 microseconds here, but there are some measurements that go all the way up to thousands of microseconds. (This graph throws away anything bigger than 2,000 μs.)

The program's "`for`" loop opens, writes, and closes a file 1,000 times. Before opening the file, the program saves the current time (in microseconds) in the variable `tstart`. After the file is closed, the program looks at the new time and calculates how long it took to open, write, and close the file. This time (again in microseconds) is stored in the variable named `delay` and printed with `printf`.

Copy this program, compile it and run it. You should see a string of mostly 3-digit numbers. Now modify the program so that it calculates the mean and standard deviation of `delay` and prints those values at the end of the program.

The mean value will tell you how long, on average, it takes your computer to open a file, write a little text into it, and close the file.

11. Imagine that you're a scientist doing an experiment. You measure the velocity of a dropped ball at several times during its fall. Being a good scientist, you'll probably repeat the experiment several times. Your measurements will be a little different every time because of random things like jiggling your experimental equipment or air currents in the room. By looking at these differences you can get an idea of the uncertainty in your measurements.

t	v	s
0.00	0.00	0.5
1.00	14.88	3.1
2.00	23.08	4.4
3.00	18.04	5.4
4.00	32.78	6.2
5.00	54.27	7.0
6.00	59.86	7.6
7.00	58.04	8.2
8.00	79.84	8.8
9.00	110.69	9.3

Figure 7.52: This table shows the ball's average velocity (v) at several different times during its fall, along with the standard deviation (s) of the velocity at that time.

The data you collect might look like Figure 7.52. If you graph your data, it might look like Figure 7.53. The bars above and below each point represent the uncertainties in the measurements of the velocity. Their length is the standard deviation of all the velocity measurements made at that point.

It looks like the data points might approximately lie on a straight line, like the one drawn in Figure 7.53, but we'd like to have some kind of numerical value that tells us how well these points match the line.

One way to measure the goodness of such a match is a quantity called *chi-squared* (χ^2). Chi-squared measures how well a set of data matches a model (in this case, a particular straight line). For our velocity data, we could calculate chi-squared like this:

$$\chi^2 = \sum_{i=1}^{N} \frac{(v_i - v_{line})^2}{s_i^2} \tag{7.5}$$

where N is the number of data points, v_i and s_i are the measured velocity and standard deviation values, and v_{line} is the velocity predicted by the straight line. The χ^2 value compares the differences

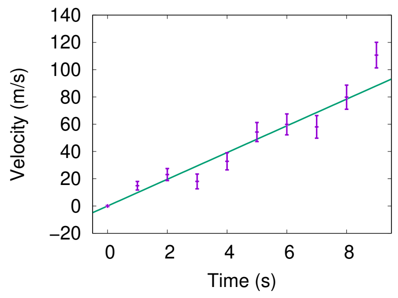


Figure 7.53: A graph of velocity *versus* time measurements, with error bars.

between measured and predicted values with our experimental uncertainties.

Since the sum in Equation 7.5 adds up the squares of a bunch of numbers, it gets bigger as we add more and more data points. To make it easier to judge whether a chi-squared value represents a good match between data and model, we often use a related quantity called the *reduced chi-squared*. It's defined like this[11]:

$$\chi^2_{red} = \frac{\chi^2}{N} \tag{7.6}$$

If a model is a good match to a set of data, the reduced chi-squared should be close to 1.

[11] If we plan on searching for the model that best matches some data, we'll usually want to divide by a number slightly smaller than N (we'd use N-2 for finding the best straight line, for example), but that's beyond the scope of what we can describe here.

Copy the three columns of numbers from Figure 7.52 into a file named `velocity.dat`, then write a program name `chi.cpp` that reads data from this file and compares it to a straight line by calculating χ^2. The program should start proceed like this:

- Use a variable named `chisq` to hold the value of X^2. Set this to zero initially.
- Use a variable named `n` to count the number of data points. Set this to zero initially.
- Start by asking the user for the slope and y-intercept of the line they want to test.
- Open `velicity.dat` for reading.
- Use a `while` loop to read values for `t`, `v`, and `s` from each line of the file.
- Each time you read a line, use the `t` value and the slope and y-intercept to calculate the velocity value predicted by the straight line:

  ```
  vline = yint + slope*t;
  ```
- Add $(v_i - v_{line})^2/s_i^2$ to the value of χ^2.
- Add one to the value of `n`.
- Keep doing this until you've read all of the data from the file.
- After reading all of the data, calculate the reduced chi-squared (χ^2_{red}).
- Print the values of χ^2 and χ^2_{red}.

Test your program by entering a slope of 9.8 and a y-intercept of zero. The reduced chi-squared should be on the order of 1, showing that this is a reasonable fit to the data. If you try a slope of 5 or 15, for example, you should get a much larger value for χ^2_{red}.

12. The mean tells us where the center of our data lies, and the standard deviation tells us how spread out the data are, but these aren't the only quantitative measurements we can derive from a list of numbers. Another thing we can measure is the *skew* of the numbers. This tells us how symmetrical they are about the mean. Data like those shown in Figures 7.54 and 7.55 have a small skew, because the numbers are symmetrically distributed on either side of the mean. Data like those shown in Figure 7.56 have a large skew, because the distribution is asymmetric: it's *skewed* to one side.

    The skew of a set of N numbers X_i is defined as:

    $$g = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{X_i - \overline{X}}{s}\right)^3 \tag{7.7}$$

    where $\overline{X}$ is the mean value and s is the standard deviation. Notice that we first need to calculate $\overline{X}$ and s before we can start calculating the skew. This means that any program we write to calculate the skew will need to look at all of the numbers twice: once to calculate the mean and standard deviation, and then again to calculate the skew.

    Skew is a little more slippery than mean and standard deviation. Small random variations in the data can cause it to appear to be skewed, even though it isn't really. One way to gauge the significance of a skew value is to compare it to the expected random variation of the skew of a normal distribution, which is

    $$\sigma_{ns} = \sqrt{15/N} \tag{7.8}$$

    If the skew value you calculate is several times larger than this, then it's probably significant. If it's smaller, then it's probably just random wiggling.

    Write a program named `skew.cpp` that does the following:

    - Open, for reading, a file named `skew-data.dat` containing a single column of numbers with decimal places.
    - Read the data from the file and calculate the mean and standard deviation of the numbers.
    - Now close the file.

- Re-open the data file for reading. This will cause your program to start back at the beginning of the file.
- Read the data from the file and use Equation 7.7 to calculate the skew of the numbers. **Hint:** Use a variable named `sum3` to accumulate the sum of the cubes shown in this equation, then after adding up all of these, divide `sum3` by N to find the skew.
- At the end, print the number of values that were in the file, the mean, the standard deviation, and the skew (g in Equation 7.7 above). Also print σ_{ns}, defined in Equation 7.8 above, and print the ratio g/σ_{ns}.

In order to test your program you'll need a data file for it to read. At the end of this chapter you'll find three small programs that can be used to generate different kinds of data sets. They're named `urand`, `nrand`, and `prand`. You can use one of them to make a `skew-data.dat` file for your program to read. For example:

```
./prand > skew-data.dat
```

If you run your `skew.cpp` program using this data set, it should print something like this:

```
N is 100000
Mean is 2.998120
Stddev is 1.731314
Skew is 0.580567
Compare to 0.012247
Ratio is 47.403123
```

The ratio at the end is g/σ_{ns}, which is much larger than 1, showing that this set of data has a significant skew.

Some Programs for Generating Lists of Random Numbers

On the following page you'll see three programs that will generate output that you can use to test the programs you write in response to this chapter's practice problems. The three programs are:

- `urand.cpp` (Program 7.13), which generates 100,000 random numbers uniformly distributed between zero and one. This is just a flat distribution, where any number in that range is equally likely.
- `nrand.cpp` (Program 7.14), which generates 100,000 random numbers in a *normal* or *gaussian* distribution centered at zero. These are numbers that are most likely to be near zero, but sometimes have higher or lower values, with decreasing probability as the number gets farther from zero.
- `prand.cpp` (Program 7.15), which generates 100,000 random numbers in a *Poisson* distribution. In this case, the numbers are all greater than or equal to zero, and are most likely to be equal to 3, but might be smaller or larger than 3. Smaller numbers are less likely as they approach zero, and larger numbers are less likely as they get farther from 3. This distribution isn't symmetric about the mean.

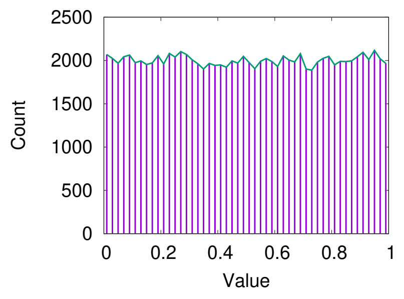


Figure 7.54: A histogram of the random numbers created by `urand.cpp`.

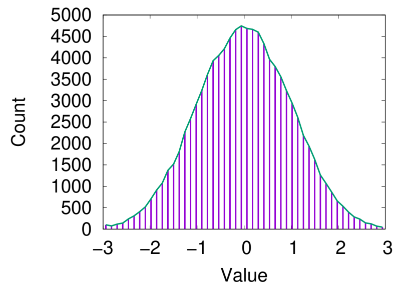


Figure 7.55: A histogram of the random numbers created by `nrand.cpp`.

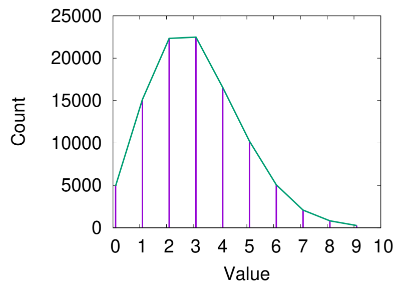


Figure 7.56: A histogram of the random numbers created by `prand.cpp`.

Program 7.13: urand.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int i;
  srand(time(NULL));
  for ( i=0; i<100000; i++ ) {
    printf( "%lf\n", rand()/(1.0+RAND_MAX) );
  }
}
```

Program 7.14: nrand.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
int main () {
  int i;
  int iroll;
  int nroll = 12;
  double sum;
  srand(time(NULL));
  for ( i=0; i<100000; i++ ) {
    sum = 0;
    for ( iroll=0; iroll<nroll; iroll++ ) {
      sum += rand()/(1.0+RAND_MAX);
    }
    printf ( "%lf\n", sum-6.0 );
  }
}
```

Program 7.15: prand.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
int main () {
  int i;
  int x;
  double u, p, s;
  double mu = 3;
  srand(time(NULL));
  for ( i=0; i<100000; i++ ) {
    x = 0;
    p = exp( -mu );
    s = p;
    u = rand()/(1.0+RAND_MAX);
    while ( u > s ) {
      x++;
      p = p*mu/x;
      s += p;
    }
    printf ( "%d\n", x );
  }
}
```