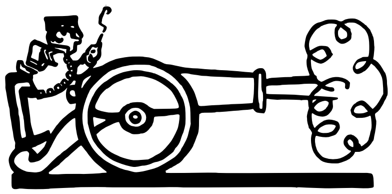
A. Some Challenging Projects

The following pages contain some projects that will challenge you to write programs using the skills you've learned in this book. Give them a try!

Project 1: Cannonball Run

Introduction: The Visitor

Imagine that you're an artilleryman in Napoleon's army. Your job is to fire a cannon, and to drop cannonballs as close as possible to a given target. You take your job seriously, and spend a lot of time thinking about the factors that limit your cannon's accuracy.



Ignoring effects of the wind and rain (which you can't control), you know that if the cannon always fired cannonballs at the same speed and angle, they'd always hit the same spot. But in reality, the speed and angle aren't always the same. Damp gunpowder or badly-formed, ill-fitting cannonballs change the speed, and the cannon doesn't stay in exactly the same position from one shot to the next, tilting a little up or down, or side to side.

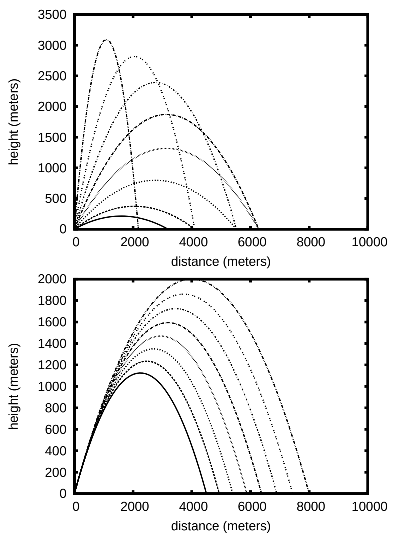


Figure A.1: The figures above illustrate how a cannonball's final position depends on the upward angle at which it's fired (top) and its initial velocity (bottom).

If you could fix even one of these problems you'd deserve a medal! But, sadly, it would take years of experimentation and tons of gunpowder to develop a new cannon design. If only there were some way to accurately simulate a real cannon with something smaller, like the toy cannons that tin soldiers use.

As you're standing beside your cannon, musing about this, a mighty concussion knocks you off your feet! An attack! But no. Rolling onto your stomach and peering through the settling dust you see, not a cannonball's crater, but an oddly-dressed man. He's lying on the ground, waving his hands in the air. "I've done it!", he shouts, "I've done it! I'm the first man to travel back in time!"

Over the course of the next hour you find out that this man has come from the 21st Century, and that the technology of his time is almost magical. The time-traveller has brought with him an object the size of a book which, when unfolded, can display moving images and even play music! The traveller calls it a "computer". This device is the solution to your problem! It can instantly simulate thousands of cannon shots!

Program 1: Simulating the Cannon

This project will require you to write three programs. The first of them will be named "`simulate.cpp`", and it will simulate a cannon. The program will allow the user to specify a speed and vertical angle for the cannonballs, but will add some random "jitter" to these values to simulate the cannon's imperfections. It will also add some random jitter to the side-to-side direction in which the cannon is pointing. The program's output will be a file containing the x, y coordinates at which each simulated cannonball lands[1].

[1] Note that in all of the following we'll ignore the effects of air resistance.

The program should accept all of its parameters on the command line, as described in Section 9.15 of Chapter 9. The usage should be:

```
./simulate nshots vinit theta outfile
```

where:

- `nshots` is the number of cannonballs to fire.
- `vinit` is the ideal initial velocity of the cannonballs (before adding any jitter).
- `theta` is the ideal angle between the cannon and the ground (before adding jitter), expressed in degrees. An angle of zero means the cannon is horizontal, and an angle of 90 means the cannon is pointing straight up into the air. (See Figure A.3.)
- `outfile` is the name of a file into which the program will write the x and y coordinates at which each cannonball lands. Assume that the cannon points along the x axis, but cannonballs may veer by some small random angle, β, to the right or left. (See Figure A.4.)

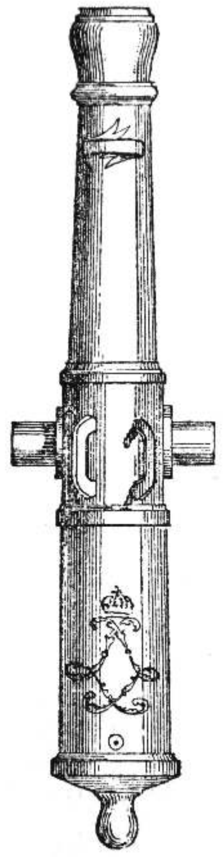


Figure A.2: Canon de 16 Gribeauval.
Source: Wikimedia Commons

If the user doesn't supply enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else.

After running the program, the output file should contain two columns of numbers with a space between them. The first column is x and the second column is y.

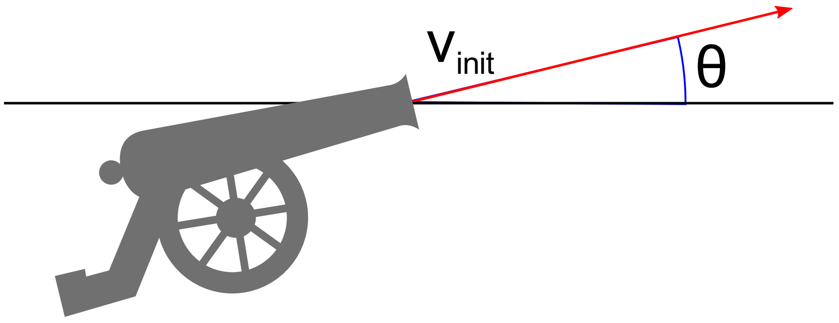


Figure A.3: **Side** view of the cannon's upward angle, θ.

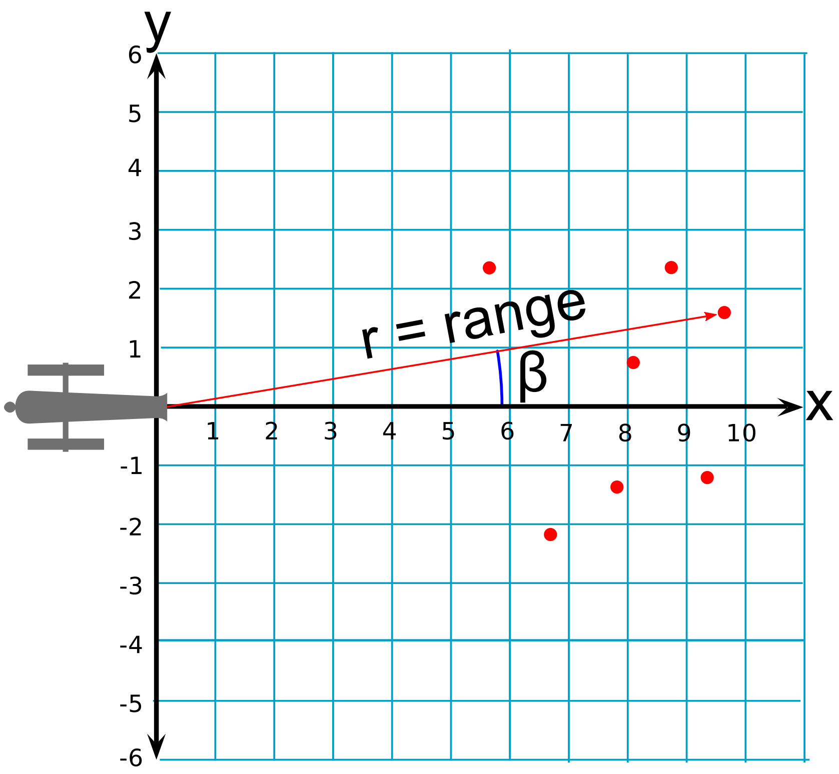


Figure A.4: **Overhead** view of a cannonballs side-to-side angle, β, its range, r, and the landing positions of some cannonballs.

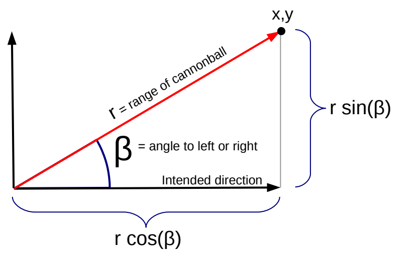


Figure A.5: Finding the x and y coordinates of a cannonball, given its range and the horizontal angle β.

To get you started, the helpful time-traveller has already written much of the program for you (see Program A.1). All you need to do is complete the program by filling in `main` and adding a `help` function that prints out a friendly usage message when the user doesn't supply enough arguments on the command line. As you can see, you'll be using several functions that have appeared in Chapters 9 and 11. These are at the top of Program A.1.

Your program should determine the landing positions of the cannonballs as follows:

1. Convert the upward angle (`theta`) into radians, since C's trigonometry functions use radians instead of degrees. You can use the function `to_radians` to do this. (This function is taken from Chapter 9, Section 9.8.)

2. Open the output file for writing. (See examples like Program 5.3 in Chapter 5.) Note that the name of the file will be in the command-line argument `argv[4]`.

3. Now loop through all of the cannon shots, using a `for` loop.

4. Each time the cannon shoots, set the cannonball's initial velocity and upward angle to the values of `vinit` and `theta`, plus some random "jitter". To do this, use the function named `normal` (taken from Section 11.4 of Chapter 11). The `normal` function generates numbers that tend to be close to zero, but sometimes have other values. (See Figure A.6.)

   - For each shot your program makes, set the cannonball's initial velocity to `vinit + 0.1*vinit*normal()`. This will give a value that tends to be within +/- 10% of the "ideal" velocity, `vinit`.
   - Set each cannonball's upward angle to `theta + 0.01*normal()`. This will give a value that tends to be close the "ideal" angle, `theta`, but has some small random variation.

5. Now that you have the cannonball's velocity and upward angle, use the `range` function (taken from Chapter 9, Section 9.8) to calculate its range. This function takes the cannonball's initial velocity and its upward angle, and returns the cannonball's "range" (the horizontal distance from the launch point to the landing point). (See Figure A.4.)

6. To determine the cannonball's landing position you'll also need to know β, the angle by which its path deviates to the right or left. (See Figure A.4.) Use the `normal` function for this by setting β equal to `0.01*normal()`. This will give you a random, small angle.

7. Now that you have the cannonball's range and the angle β, you calculate the x and y coordinates of its landing spot. See Figure A.5.

8. Finally, write the x and y coordinates into the output file. (See examples like Program 5.3 in Chapter 5 if you don't remember how to do this.)

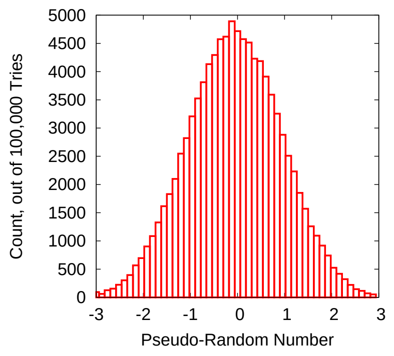


Figure A.6: The `normal` function generates pseudo-random numbers that are most likely to be near zero, with smaller probabilities for other values. This figure shows 100,000 pseudo-random numbers generated by `normal`.

Once you've written and compiled your program, run it like this to produce an output file to use with your next program:

```
./simulate 10000 250 45 simulate.dat
```

Program A.1: simulate.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }

  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;

  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }

  return ( sum - 6.0 );
}

double g = 9.81; // Acceleration of gravity.

double to_radians ( double degrees ) {
  return ( 2.0 * M_PI * degrees / 360.0 );
}

double time_of_flight ( double v0, double angle ) {
  double t;
  double vy0;
  vy0 = v0 * sin(angle);
  t = 2.0 * vy0 / g;
  return ( t );
}

double range ( double v0, double angle ) {
  double d;
  d = v0 * cos(angle) * time_of_flight( v0, angle );
  return ( d );
}

int main (int argc, char *argv[]) {

        //
        // Insert program here!
        //
}
```

Program 2: Analyzing the Results

Your second program will be named `analyze.cpp`. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first program, `analyze` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze filename
```

where `filename` is the name of a data file produced by your `simulate.cpp` program.

The output of the `analyze` program should look like this:

```
Average x = 6428.287617
Std. dev. of x = 1286.944844
Min x = 2568.660526
Max x = 13046.427659
Average y = -0.611109
Std. dev. of y = 65.978704
Min y = -284.001774
Max y = 313.589122
```

showing the average values of x and y, the standard deviations of x and y, and the minium and maximum values of x and y.

The helpful time-traveller has come to your aid again, and written some of the program for you (see Program A.2). You'll just need to fill in `main` and write a `help` function.

To analyze the data, the program should proceed as follows:

1. First, open the data file for reading. See Program 7.5 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation.

2. The time-traveller has kindly provided you with an easy way to find minimum and maximum values, using the two functions `findmin` and `findmax`. Each time you read a new value of x, for example, just say `xmax = findmax(x,xmax,n)`. This will update the value of `xmax` if necessary. When you're done reading all of the data, `xmax` will contain the largest value of x.

Run your program to analyze the `simulate.dat` file you produced earlier. Check to make sure its results look realistic. (Compare them to the sample output above.)

Program A.2: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
double findmax ( double x, double oldvalue, int n ) {
  if ( n == 0) {
    oldvalue = x;
  } else {
    if ( x > oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
double findmin ( double x, double oldvalue, int n ) {
  if ( n == 0 ) {
    oldvalue = x;
  } else {
    if ( x < oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
int main ( int argc, char *argv[] ) {

        //
        // Insert program here!
        //

}
```

Program 3: Making Pictures

Your final program will be called `visualize.cpp` and it will let you make pictures like the ones shown in Figure A.8. These figures show the distribution of landing positions of 10,000 simulated cannonballs.

The figures represent 2-dimensional histograms. We talked about histograms in Chapter 7, but we didn't say much about 2-dimensional ones. Because of that, our friendly time-traveller has written almost all of this program for you. (See Program A.3.)

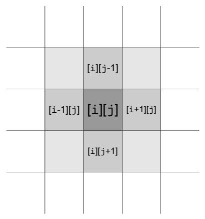


Figure A.7: Each element of `grid` records the number of cannonballs that landed within a particular section of the battlefield.

This program uses a 2-dimensional, `nbins` × `nbins` array named `grid`. Each element of the array represents an area of the battlefield. The number stored in each element is the number of cannonballs that landed in that area.

Like the preceding programs, this one will expect parameters on its command line. Its usage will be:

```
./visualize xmin xmax ymin ymax infile outfile
```

where `xmin`, `xmax`, `ymin`, and `ymax` specify the limits of rectangular area of the battlefield. `infile` is the name of a data file produced by your `simulate` program. `outfile` is the name of a file into which the current program will write its results.

Two key parts of the program have been left for you to fill in. First, near the top of `main`, you need to set all of the elements of `grid` to zero. To do this, you'll need two nested "`for`" loops. Inside the loops, set each element, `grid[xbin][ybin]`, to zero.

Second, near the end of `main`, you need to open the output file for writing and write your results into it. (You'll again need two nested "`for`" loops to do this.)

The file should have three columns, `x`, `y`, and `grid[xbin][ybin]`, where `x` and `y` are the coordinates of the center of the grid element. Use `x=xmin+xbinwidth*(0.5+xbin)`, and `y` similarly, for the center position of each grid element.

There should also be a blank line after every `nbins` rows. See the end of Section 6.12 for an explanation of this blank line, and the last part of Program 6.8 for an example showing how to create it.

After writing and compiling the program, try it out. Use your `analyze` program to find good values for `xmin`, `xmax`, `ymin`, and `ymax`. Use these values and your newest program to process the data in `simulate.dat` and create a new file, `visualize.dat`, that can be plotted with *gnuplot*:

```
./visualize 2569 13046 -284 314 simulate.dat visualize.dat
```

Try plotting your results with *gnuplot*. To produce the top graph in Figure A.8, give *gnuplot* the following command:

```
plot "visualize.dat" with image
```

To produce the bottom graph in Figure A.8, use this *gnuplot* command:

```
splot "visualize.dat" with image, "" with histeps
```

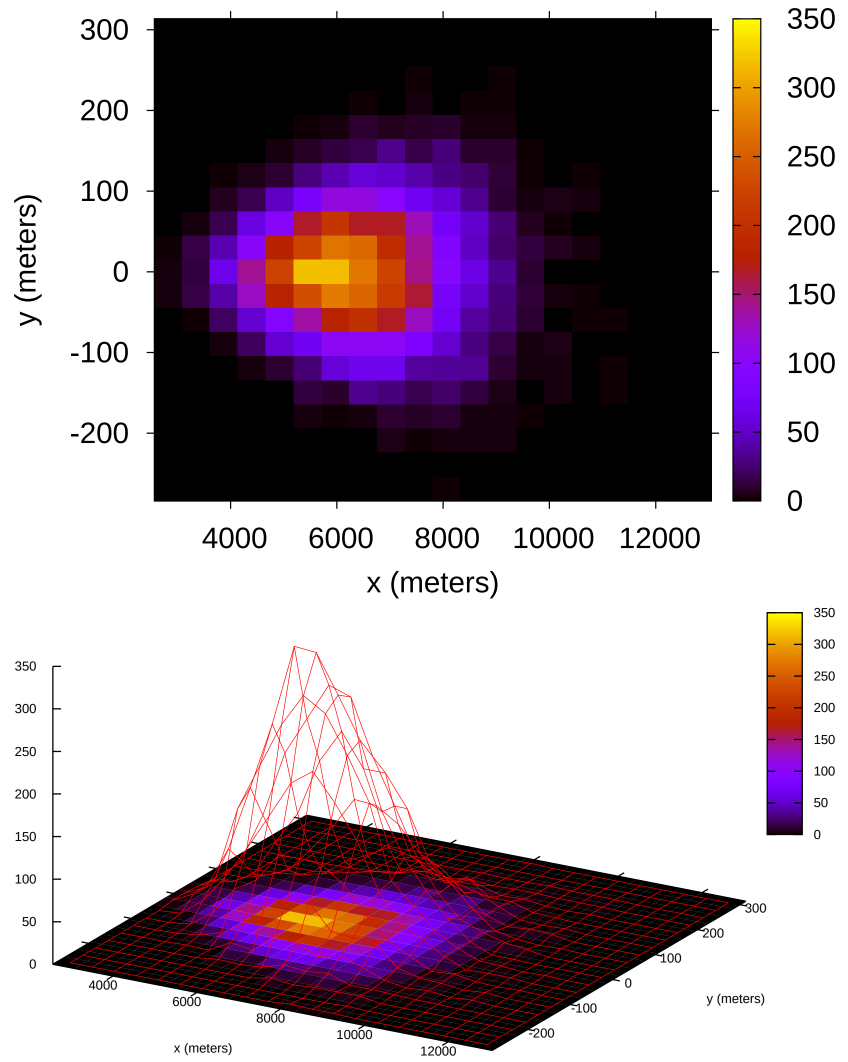


Figure A.8: Two views of the distribution of cannonball landing positions. The color scale shows how many cannonballs (out of 10,000) landed in each grid element.

Program A.3: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>

void help() {
  printf ("Usage: ./visualize xmin xmax ymin ymax input.dat output.dat\n");
}

int main ( int argc, char *argv[] ) {
  const int nbins = 20;
  int grid[nbins][nbins];
  double x, y;
  double xmin, xmax;
  double ymin, ymax;
  double xbinwidth, ybinwidth;
  FILE *output;
  FILE *input;
  int xbin, ybin;

  if ( argc != 7 ) {
    help();
    exit(1);
  }

  // Insert code here to reset all bins to zero.

  xmin = atof(argv[1]);
  xmax = atof(argv[2]);
  ymin = atof(argv[3]);
  ymax = atof(argv[4]);

  xbinwidth = (xmax - xmin)/(double)nbins;
  ybinwidth = (ymax - ymin)/(double)nbins;

  input = fopen ( argv[5],"r" );

  while ( fscanf(input, "%lf %lf", &x, &y) != EOF ) {
    xbin = (x-xmin)/xbinwidth;
    ybin = (y-ymin)/ybinwidth;
    if ( xbin >= 0 && ybin >= 0 && xbin < nbins && ybin < nbins ) {
      grid[xbin][ybin]++;
    }
  }

  fclose ( input );

  // Insert code here to open the output file and write
  // the contents of "grid" into it.
}
```

Last Words

As your friend from the future fades away in a cloud of sparkles, you stand there savoring your brief glimpse of the future. "If only we had such technology today," you sigh, as you hear your commander shout the order to begin breaking camp.

Figure A.9: Wellington at Waterloo.
Source: Wikimedia Commons

While you prepare to march into Russia during the Spring of 1812, far away in England a mathematician named Charles Babbage is looking at mathematical tables, like the ones used by artillerymen for aiming their cannons, and thinking about how these tables could be generated automatically, by machinery instead of humans.

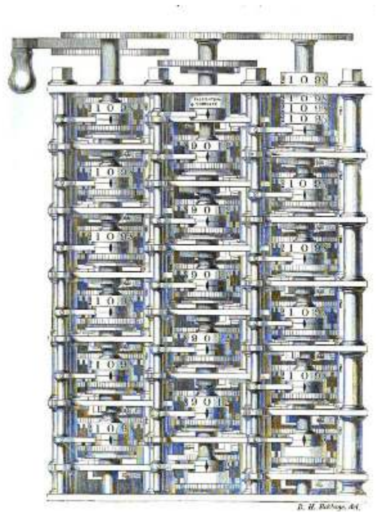


Figure A.10: Part of Babbage's "Difference Engine".
Source: Wikimedia Commons

After Napoleon's defeat at Waterloo in 1815, Babbage exchanges ideas with other mathematicians, English and French, and in 1822 he begins work on the series of computing machines that will become the ancestors of all modern computers.

Figure A.11: The Emperor Napoleon (left), and Babbage's brain (right).
Source: Wikimedia Commons 1, 2

Project 2: Diffusion Confusion

Introduction: Randomly-Bouncing Molecules

Imagine that you're in a large room full of perfectly still air. At the opposite end of the room is a just-opened bottle of perfume. The volatile molecules from the perfume have started to wander out into the room, bouncing off of molecules in the air. How long would it take these molecules to bounce their way across the room to your nose?

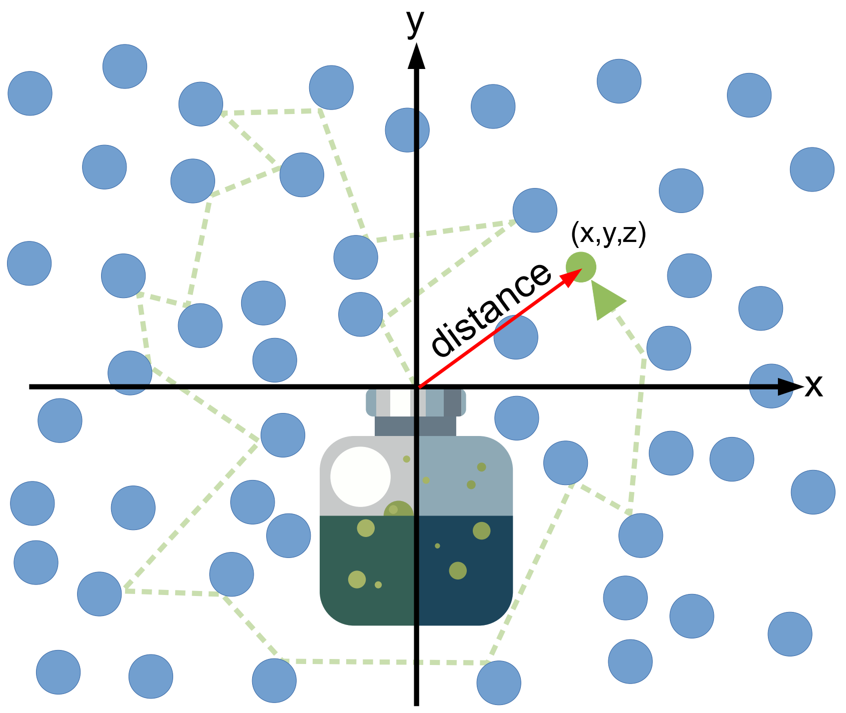


Figure A.12: A molecule leaves the perfume bottle, then bounces around among the air molecules for a while, ending up at a position (x, y, z) some distance from where it started.

A typical speed for a molecule in air is about 1,000 miles per hour, but our perfume molecules don't travel in a straight line. Figure A.12 shows a typical perfume molecule's path. Since it bounces around at random, it tends to linger near the bottle for a long time. The process by which molecules spread out by bouncing around this way is called "diffusion".

In this project you will write three programs: `simulate.cpp`, `analyze.cpp`, and `visualize.cpp`. The first will simulate the paths of perfume molecules through air, the second will analyze the simulated data, and the third will help us visualize one of the results.

Program 1: Simulating the Paths of Molecules

Your first program will be named `simulate.cpp`. It will track the random movement of some number of perfume molecules as they undergo some number of collisions. The program will write the final position of each molecule, and how long it took the molecule to get there, into an output file.

The perfume molecule's path is an example of a random walk, and this program will be very similar to Practice Problem 5 in Chapter 7. One difference is that the new program tracks a random path in three dimensions instead of two, so you'll need to keep track of the molecule's x, y, and z coordinates. Another difference is that we won't assume that each step of the path has the same length, as we did in the earlier program. This time, we'll let the step length vary a little. Each step in the molecule's random path will be the distance from one collision to the next. Finally, the new program won't bother with keeping track of sums or averages.

The program should accept all of its parameters on the command line, as described in Section 9.15 of Chapter 9. The usage should be:

```
./simulate nparticles ncollisions output.dat
```

where:

- `nparticles` is the number of perfume molecules we want to simulate.
- `ncollisions` is the number of collisions each molecule will experience.
- `output.dat` is the name of a file into which the program's results will be written.

If the user doesn't supply enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else. See Section 9.15 of Chapter 9 for an example showing how to do this.

After running the program, its output file should contain four columns of numbers: The x,y, and z coordinates where the molecule ended up, and the time it took to get there. We'll measure time in microseconds (1 microsecond = 10^{-6} seconds) and distances in microns (1 micron = 10^{-6} meters).

Each time a perfume molecule collides with an air molecule, we'll need to generate a new random direction for it, and a new random distance to the next collision. In 3-dimensional space, we can describe a particle's direction with two angles, θ (theta) and φ (phi) (see Figure A.13):

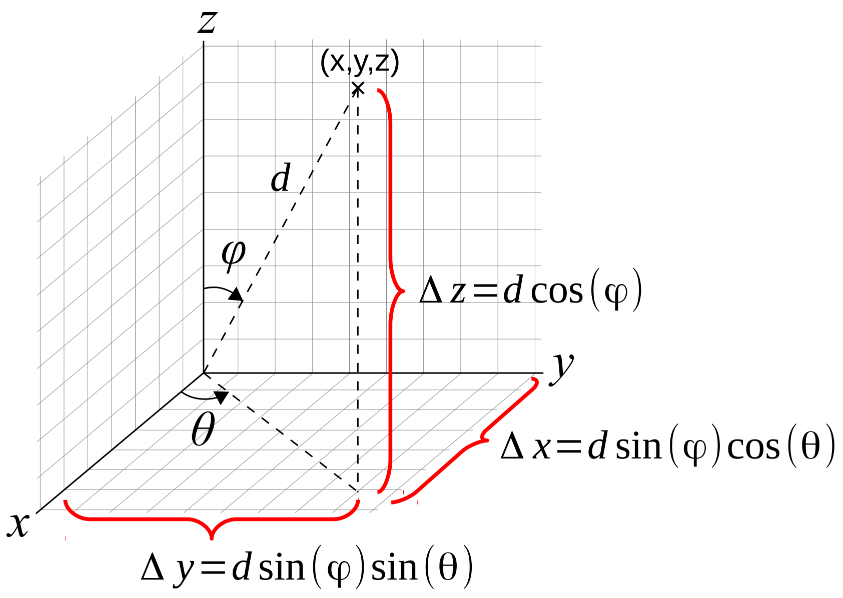


Figure A.13: After a collision, the molecule's new direction is given by two angles, θ and φ. The distance to the next collision is d.

- The angle θ can point in any direction away from the Z axis. It can have any value between zero and 2π radians (360°).
- The angle φ can have any value between straight up (zero) and straight down (π radians, or 180°).

The distance, d, will vary around some average value called the "mean free path", which we'll assume to be 0.14 microns. Each time we generate a value for d we'll do so by adding a little bit of random "jitter" to this distance.

To get you started, I've already written some of the program for you (see Program A.4). All you need to do is complete the program by filling in `main`. As you can see, you'll be using two functions that have appeared in Chapter 11. These are at the top of Program A.4. You'll also see that I've defined the values of the mean free path (`meanpath`) and the speed of the molecules (`speed`), which we assume to be 500 microns/microsecond.

Figure A.14: Trading card for Hoyt's German Cologne, circa 1900.

Source: Wikimedia Commons

To track the molecules, your program should do the following:

1. Open the output file for writing[2]. The output file name will be given by `argv[3]`, so you can say something like "`output = fopen(argv[3],"w");`".

2. You'll need a pair of nested `for` loops: An outer loop for each molecule, and an inner one for each collision[3].

3. Keep track of the molecule's position with three variables, `xpos`, `ypos`, and `zpos`. Keep track of the time elapsed with a variable named `t`. Remember to set all of these back to zero whenever you begin tracking a new molecule.

4. Every time the molecule collides, do the following:

   (a) Generate two random angles like this:

   ```
   theta = 2.0*M_PI*rand01();
   phi = M_PI*rand01();
   ```

   (b) Generate a random distance like this:

   ```
   d = meanpath * ( 1.0 + 0.1*normal() );
   ```

   where `normal` is a function shown in Program A.4 below.

   (c) Add Δx, Δy, and Δz (as shown in Figure A.13) to the values of `xpos`, `ypos`, and `zpos`, respectively, to get the molecule's new position[4].

   (d) Update `t` by adding `d/speed` to it. This is the time it will take the molecule to travel the distance `d`.

5. Use the trick described in Section 4.4 of Chapter 4 to print out progress reports as your program is running. After every 10 molecules, print a message like this on the screen: `Working on molecule 10...` (or 20, or 30, and so on). It's OK if the program prints "`Working on molecule 0`" when it starts.

[2] For a reminder about how to write output into files, see examples like Program 5.3 in Chapter 5.

[3] This is similar to what we did in Program 2.7 in Chapter 2.

[4] If you're not familiar with the symbols in Figure A.13, remember that θ is `theta` and φ is `phi`. These are the random angles you generated in step (a) above.

6. After tracking the molecule through `ncollisions` collisions, write `xpos`, `ypos`, `zpos`, and `t` into the program's output file[5]. These should be written as four numbers separated by single spaces, with a `\n` at the end of the line.

[5] See examples like Program 5.3 in Chapter 5.

Once you've written and compiled your program, run it like this to produce an output file to use with your next program:

```
./simulate 1000 16000 simulate-16000.dat
```

This should produce an output file (`simulate-16000.dat`) containing the final positions and times for 1,000 perfume molecules after each of them bounces 16,000 times.

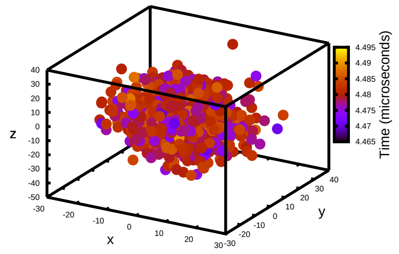


Figure A.15: You can check your first program's results by plotting them with *gnuplot*. This figure shows what you should see if you type:
`splot "simulate-16000.dat" with points palette pointsize 3 pointtype 7`
It shows the final x, y, and z positions of the molecules, color-coded by how long it took them to get there.

Program A.4: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;

  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

int main ( int argc, char *argv[] ) {

  double meanpath = 0.14; // Microns per collision
  double speed = 500; // Microns per microsecond

        //
        // Insert program here!
        //
}
```

Program 2: Analyzing the Results

Your second program will be named `analyze.cpp`. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first program, `analyze` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze input.dat
```

where `input.dat` is the name of a data file produced by your `simulate.cpp` program.

The output of the `analyze` program should look like this:

```
Average distance = 16.292850 microns
Std. dev. of distance = 6.987062 microns
Min distance = 0.684207 microns
Max distance = 45.581858 microns
Average time = 4.480129 microseconds
Std. dev. of time = 0.003552 microseconds
Min time = 4.469744 microseconds
Max time = 4.491567 microseconds
Diffusion Coefficient is 0.29626 cm^2/s
```

where `distance` is the final distance of a molecule from the origin, which is given by

$$distance = \sqrt{x^2 + y^2 + z^2}$$

and `time` is the amount of time the molecule took to get there, which is just the fourth column in your data file.

The "Diffusion Coefficient" is a way of measuring how fast molecules diffuse through the air. It's usually given in units of cm^2/s. If your program calls the average distance `davg` and the average time `tavg`, you can calculate the diffusion coefficient like this:

```
dcm = davg/1.0e4;
tseconds = tavg/1.0e6;
dcoeff = dcm*dcm/2.0/tseconds;
```

where `dcm` is the distance converted to centimeters and `tseconds` is the time converted to seconds. `dcoeff` is the Diffusion Coefficient. It should end up having a value of around 0.3 cm^2/s if your programs are working properly.

Figure A.16: Broken glass perfume amphora from Ephesus, 2nd century CE.
Source: Wikimedia Commons

Again, I've already written some of the program for you (see Program A.5). You'll just need to fill in `main`.

To analyze the data, the program should proceed as follows:

1. First, open the data file for reading. See Program 7.5 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation.

2. At the top of Program A.5 below I've provided you with an easy way to find minimum and maximum values, using the two functions `findmin` and `findmax`. Each time you read a new value of t, for example, just say `tmax = findmax(t,tmax,n)`, where `n` is the number of molecules you've processed so far. This will update the value of `tmax` if necessary. When you're done reading all of the data, `tmax` will contain the largest value of t. **Note:** It's important that `n` be equal to zero the first time you use these functions.

3. After reading all of the data from the input file, calculate the Diffusion Coefficient (as shown above) and print all of the results.

Figure A.17: "The Perfume Maker", by Rudolf Ernst.

Source: Wikimedia Commons

Program A.5: analyze.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
double findmax ( double x, double oldvalue, int n ) {
  if ( n == 0) {
    oldvalue = x;
  } else {
    if ( x > oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
double findmin ( double x, double oldvalue, int n ) {
  if ( n == 0 ) {
    oldvalue = x;
  } else {
    if ( x < oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
int main ( int argc, char *argv[] ) {

        //
        // Insert program here!
        //

}
```

Program 3: Visualizing the Distance

Your final program will be called `visualize.cpp` and it will let you make pictures like the one shown in Figure A.18. This figure shows the distribution of final distances of 1,000 perfume molecules after 16,000 collisions.

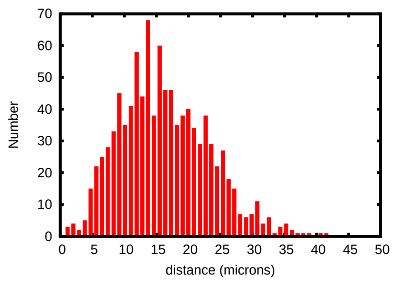


Figure A.18: Distribution of the final positions of 1,000 perfume molecules after each has experienced 16,000 collisions.

This figure is a histogram, like the ones we discussed in Chapter 7. Your third program will be similar to Program 7.1 in that chapter. Again, to get you started, I've written part of the program for you (see Program A.6 below). Notice that I've defined a 50-element array, `bin`, to hold the histogram data.

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize dmin dmax input.dat output.dat
```

where `dmin` and `dmax` are the minimum and maximum distances (as determined by your `analyze` program) `input.dat` is the name of a file produced by your `simulate` program, and `output.dat` is a file into which your new program will write the histogram data.

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a distance instead of a bin number (see below for instructions about converting bin number to distance). The second column will be the number of molecules in that bin.

To make the histogram, the program should proceed as follows:

1. First, determine the `binwidth`, like this:

   ```
   binwidth = (dmax-dmin)/nbins;
   ```

2. Next, use a `while` loop to read data from the input file. Each line of the file will contain four values: x, y, z, and t.

3. Every time you read a line, determine the distance from $distance = \sqrt{x^2 + y^2 + z^2}$.

4. Determine which bin this distance belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does.

5. After processing all of the input data, write the histogram data into

the output file. For each bin of the histogram, write two numbers separated by a single space: the distance represented by that bin, and the number of molecules that fell within it. The distance can be calculated from the bin number, like this:

```
distance = dmin + binwidth*(0.5+i);
```

where `i` is the bin number.

6. Finally, at the bottom of the output file, write a line beginning with a `#` that tells how many overflows or underflows were seen.

Run your program like this to make a histogram of the data you produced earlier:

```
./visualize  0.684207 45.581858 simulate-16000.dat visualize-16000.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize-16000.dat" with impulses lw 5
```

The result should look like Figure A.18.

Program A.6: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {

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

        //
        // Insert program here!
        //

}
```

Results

What do your results tell you? If you were to run your `simulate` program two more times, like this:

```
./simulate 1000 1000 simulate-1000.dat
./simulate 1000 4000 simulate-4000.dat
```

and then use your `analyze` program to analyze each of these files and your `simulate-1600.dat` file, you might notice a pattern. Every time you increase the number of collisions by a factor of four, the average distance increases by a factor of two. This fact is reflected in the definition of the Diffusion Coefficient, which tells us that the time it takes molecules to travel a given distance by diffusion is:

$$t = \frac{d^2}{2D}$$

where t is the time, d is the distance, and D is the diffusion coefficient.

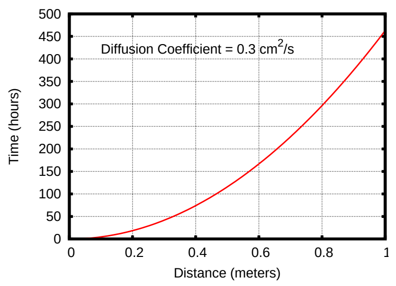


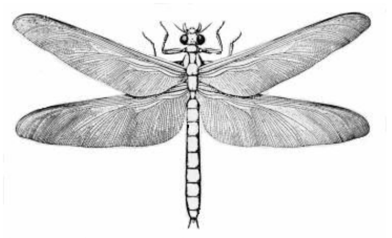
Figure A.19: How long would it take our perfume molecules to diffuse across a room? A long time!

If we plotted time versus distance, we'd get a graph like Figure A.19. As you can see from the graph, it would take hundreds of hours for our perfume molecules to travel even one meter. Diffusion is apparently very slow! Scents usually reach our nose by riding on air currents, rather than through diffusion.

Why is diffusion so slow? From Chemistry class we know that a small amount of air (say, a ballon full) contains on the order of 10^{23} molecules. That's a lot of obstacles to bounce off of. Even though our perfume molecule might be traveling at 1,000 miles per hour, it collides with air molecules billions of times per second, and each collision sends it off in another random direction.

The low speed of diffusion explains why we have lungs, and why there aren't any human-sized insects. Breathing moves oxygen by two mechanisms: *diffusion* and *advection*. When we breath, air is drawn into our lungs by advection (the bulk motion of a fluid) and it brings oxygen molecules along with it. When the air gets down into our lungs, oxygen molecules then diffuse through the thin walls of blood vessels. This is a very short distance, so diffusion can do the job relatively quickly. The blood then carries the oxygen all through our body (advection again).

Insects don't have lungs. Their bodies contain hollow tubes called *tracheae* that open to the outside world. Oxygen molecules wander into these tubes by diffusion, and then wander through the tubes until they reach cells inside the insect's body. This is a slow process, but since insects are small, the distances are short. If insects were human-sized, they couldn't get oxygen quickly enough through this mechanism.



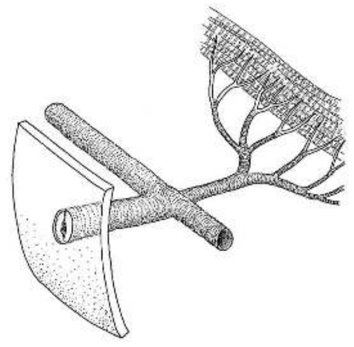


Figure A.20: In the Carboniferous period Earth's oxygen levels were much higher than they are today. This allowed giant inects like the dragonfly *Meganeura* (top) to survive, even without lungs. *Meganeura* had a two-foot wingspan! The bottom illustration shows tracheae inside an insect's body.

Source: *Wikimedia Commons* *and* *D.G. Mackean*

Project 3: Proton Power

Introduction: Particle Beam Therapy

We all know that radiation can cause cancer, but radiation can also be used to fight cancer. One example of this is particle beam cancer therapy, in which a beam of charged particles (usually protons or pions) is shot into a tumor with the goal of destroying it.

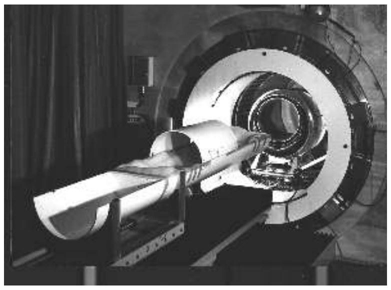


Figure A.21: An apparatus used for pion-beam radiation therapy at the Paul Scherrer Institut. The patient lies in the semicircular cradle, which is inserted into the apparatus behind during treatment.

As particles from such a beam travel through the body, they gradually lose energy and eventually come to rest. As it turns out, much of the particle's energy is lost close to the point at which it stops. This makes such beams well-suited for killing tumors without doing too much damage to the other tissues they pass through on the way to the tumor, or tissues beyond the tumor.

Particles with higher energies will travel farther into the body. By adjusting the energy of the particles, we can cause them to stop at a chosen depth (ideally, inside a tumor).

At moderate energies, a beam of particles traveling through a body loses energy mostly through interactions with electrons. Although it's possible that some of the particles will bump into an atomic nucleus, that doesn't happen very often. Since protons are 2,000 times heavier than electrons, beams of these particles tend to travel in a straight line, knocking puny electrons aside as they go.

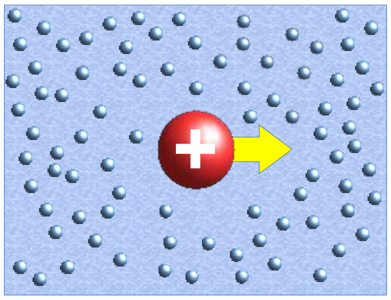


Figure A.22: A proton (shown with a plus sign because of its positive charge) is much larger than the electrons it knocks aside while travelling through the body.

Figure A.23 shows how much energy protons deposit as they travel through the body. The four curves show what happens when you use protons of four different starting energies, ranging from 50 MeV to 125 MeV. The energy deposited damages the body's tissues. The goal is to destroy the tumor without doing too much damage to healthy tissue.

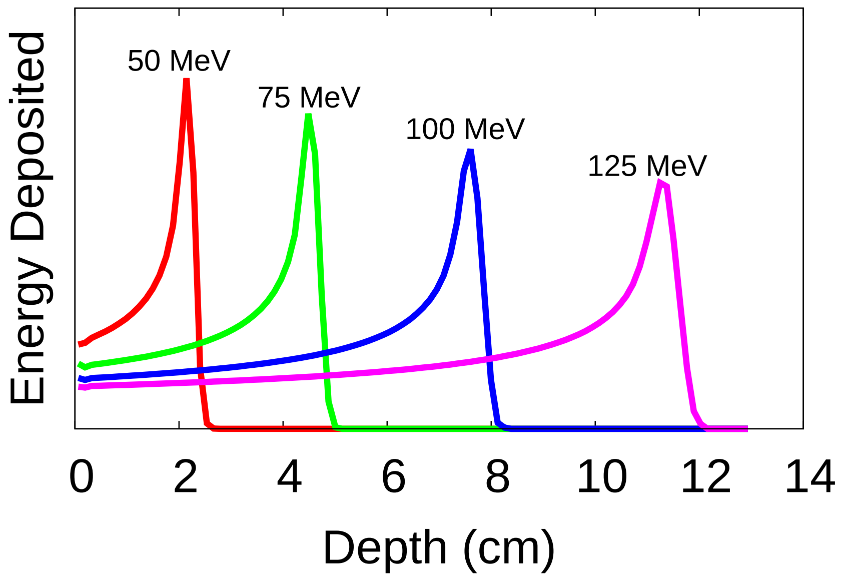


Figure A.23: Energy deposited at various depths by incoming protons having energies of 50, 75, 100, or 125 MeV. As you can see, more energetic protons penetrate to greater depths. Also notice that most of a proton's energy is deposited near its stopping point.

The Assignment

Imagine you're a doctor working at a radiation therapy facility. You have at your command a beam of protons. You can aim the beam precisely, and control its energy.

You're preparing for a visit by a patient with a 2-centimeter-thick tumor buried 8 centimeters deep in her body (see Figure A.24). You need to determine what energy the protons should have in order to deposit most of their energy in the region of the tumor.

A physicist colleague has given you a formula to calculate the energy lost by a particle while going through a thin slice of material. The formula has a form like this[6]:

[6] The actual equation is called the Bethe-Bloch formula.

$$\Delta E = \Delta x \cdot f(E, proton\ properties, material\ properties)$$

where ΔE is the amount of energy the particle loses, Δx is the thickness of the slice, and f is some function that depends on E (the energy at the beginning of the slice) as well as the constant properties of the particle (like charge and mass) and properties of the material (like density).

Unfortunately, your physicist friend tells you that eight centimeters is too big to call a "thin slice". But that's OK, she says. Just treat the eight centimeters as though it was a stack of thinner slices, as shown in Figure A.25. Each time the proton passes through one of the slices, it loses some amount of energy, ΔE. This lost energy damages the tissue in that slice. The proton then enters the next slice with its energy reduced by the amount ΔE.

Your assignment is to write three programs: **`simulate.cpp`**, **`visualize.cpp`**, and **`analyze.cpp`**. The first will simulate the passage of protons through the patient's body, the second will help visualize these results, and the third will help choose the right proton energy.

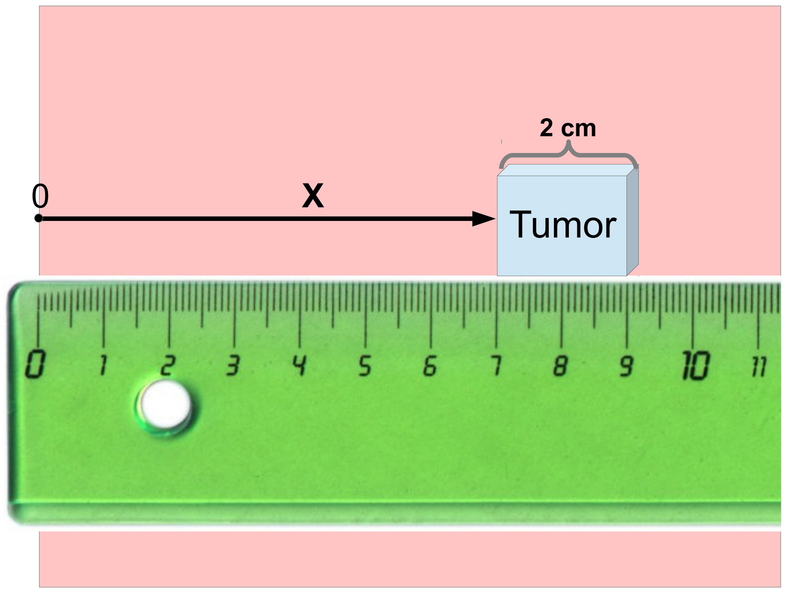


Figure A.24: Our patient's tumor is 2 cm thick, and is centered at a depth of 8 cm. Here "x" represents the depth below the patient's skin.

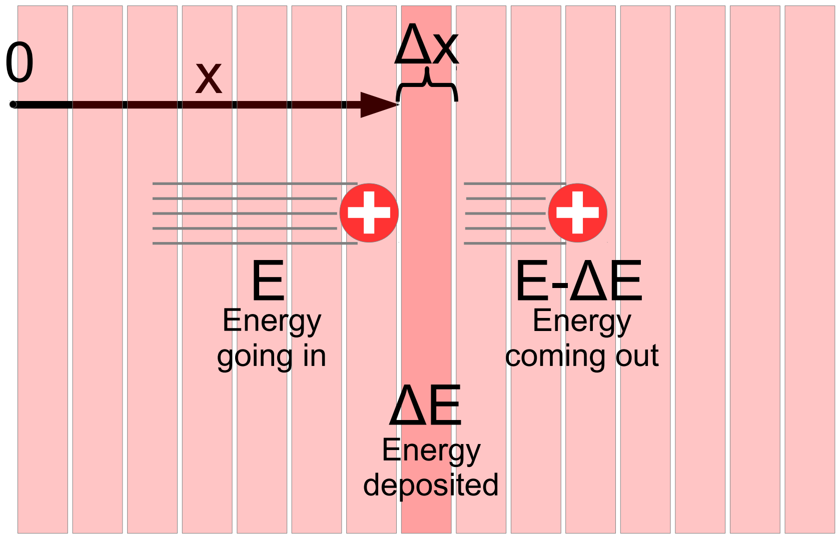


Figure A.25: We can look at the patient's body as a series of thin slices through which the proton must pass. Each time the proton passes through one of the slices, it loses some amount of energy, ΔE. This lost energy damages the tissue in that slice.

Program 1: Simulating Protons

Your first job will be to write a program named **simulate.cpp** that keeps track of the energy that protons lose as they travel through such a stack of thin slices. Each slice will have a thickness of 0.01 cm. Assume each proton travels in a straight line, starting at `x = 0` and progresses along the `x` axis until it runs out of energy. Each time a proton passes through a slice, the program should write the proton's position, energy loss, and remaining energy into an output file.

Figure A.26: The international symbol for ionizing radiation, which was first used at Berkeley Radiation Laboratory in 1946.

Source: Wikimedia Commons

Your physicist friend has kindly provided you with the beginning of a program, but she's too busy to finish it. The part she's written for you is shown in Program A.7. Near the top of the program are some numbers you'll need. The program assumes that humans are just made out of water, since they mostly are.

She's also written some useful functions in a header file named **dedx.h**, which is shown below as Program A.8. The biggest function in it is named `dEdx`, and it does most of the work of calculating how much energy a proton loses while going through one of the slices. Notice that `simulate.cpp` has an `include` statement near the top that fetches `dedx.h`.

Program A.7: simulate.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "dedx.h"

int main (int argc, char *argv[]) {
  double pmass = 938.27;     // MeV, Proton mass.
  double pcharge = 1.0;      // Proton charge.
  double rho = 1.0;          // Density, g/cm^2, for water.
  double amass = 18.01;      // Atomic mass, AMU, for water.
  double anum = 10.0;        // Atomic number, Z, for water.
  double activation = 75.0; // Activation energy, eV, for water.

  double dx = 0.01;          // Slice thickness, cm.

  int nprotons;
  double estart, energy;
  double x, de;
  FILE *output;

  // Sorry! got to run to a faculty meeting. You'll
  // have to insert the rest of the program here.

}
```

To complete the program, you'll need to do the following:

1. First, copy Program A.8 (`dedx.h`) into a file named `dedx.h` and save it. Then create a file named `simulate.cpp` and start by putting the contents of Program A.7 into it. This will be the program that does your proton simulation.
2. Your program should accept three arguments on the command line.[7] When you're done writing your program, you should be able to run it like this:

   ```
   ./simulate nprotons estart output
   ```

   where:

   - `nprotons` is the number of protons you want to simulate.
   - `estart` is the starting energy of the protons.
   - `output` is the name of an output file into which the program will write its results.

   If the user doesn't supply enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[8].

   Since `nprotons` is an integer, you'll need to use the `atoi` function to convert this command-line argument[9]. For `estart` you'll need to use `atof`, since this number can contain decimal places. The output file name won't need any conversion, since it's already a character string. You can just use that argument directly, like this:

   ```
           output = fopen( argv[3], "w" );
   ```

3. Your program will need a pair of nested loops: An outer "`for`" loop that generates protons, one a at a time, and an inner "`do-while`" loop that tracks each proton through the slices until the proton loses all of its energy.

4. Each time the program starts tracking a new proton it should set the proton's initial position and energy. To be more realistic, the program should add some "wiggle" to these values. In the real world, the particles in a proton beam don't all have exactly the same energy, and they won't necessarily enter the body at exactly the same spot (the patient might move a little, for example). Use the function named "`normal`" (defined in `dedx.h`) for this. Here's how to do it:

   ```
   energy = estart + 0.01*estart*normal();
   ```

[7] We learned how to use command-line arguments in Sections 9.15 and 9.16 of Chapter 9.

[8] See Section 9.16 of Chapter 9 for information about how to do this.

[9] See Problem 7 (`add.cpp`) at the end of Chapter 9.

Figure A.27: A "wind" of charged particles, including many protons, blows outward from the Sun. It interacts with the earth's magnetic field to produce the aurora.
Source: Wikimedia Commons

```
x = 0 + 0.1*normal();
```

This sets the proton's initial energy to `estart` $\pm$ 1% and the starting position to zero cm $\pm$ 1 mm.

5. Each time a proton goes through a slice of tissue, your program should do the following:

   (a) Calculate the amount of energy the proton deposits in the slice (we'll call that "`de`"). Our physicist friend has given us the function named `dEdx` to help us calculate this.

   ```
   de = dx * dEdx(energy, pmass, pcharge, rho, amass, anum, activation);
   ```

   (b) Calculate the proton's new energy:

   ```
   energy = energy - de;
   ```

   (c) Update the proton's position:

   ```
   x = x + dx;
   ```

6. Every time we change the values of `x`, `de`, and `energy`, the program should write those values into the output file specified on the command line[10]. These should be written as three numbers, separated by single spaces, with a `\n` at the end of the line.

7. We can't know in advance how many slices a proton will travel through before its energy is all gone. We just have to look at the energy after each slice, and see if it's still greater than zero[11].

   Near the end of the proton's path, because of the approximations we're making, the `dEdx` function might tell us that the proton loses no energy, even though it has some energy left. That means you also need to check the value of `de` at the end of your "`do-while`" loop:

   ```
   } while ( energy > 0 && de > 0 );
   ```

Once you've written and compiled your program, run it like this to produce an output file to use with your next programs:

```
./simulate 1000 100 100-mev.dat
```

This should produce an output file named `100-mev.dat` containing information about the energy deposited by each proton, in each slice of the patient's body.

[10] See examples like Program 5.3 in Chapter 5.

[11] This is similar to the `baselpi.cpp` program you wrote for Problem 6 in Chapter 4. In that program, we kept calculating smaller and smaller terms, until we got to one that was less than some limit. That program used a "`do-while`" loop, and we can use one of those here, too.

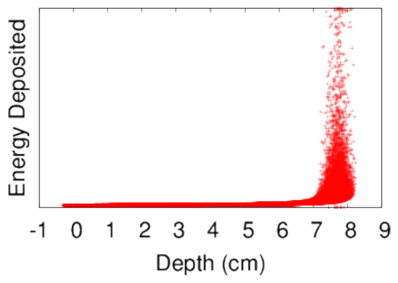


Figure A.28: You can check your first program's results by plotting them with *gnuplot*. This figure shows what you should see if you type:
`plot "100-mev.dat" using 1:2`
It shows the energy deposited in each slice by each proton.

The file `dedx.h`, below, contains a function named `dEdx` for calculating the energy lost (ΔE) in a slice of matter with thickness Δx. This file also contains two random-number functions that we've used before. `rand01` generates random numbers uniformly distributed between zero and one, and `normal` generates random numbers in a Gaussian or "normal" distribution[12].

[12] You can read about both of these in Chapter 11.

Program A.8: dedx.h

```
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int i, nroll = 12;
  double sum = 0;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

// Returns dE/dx, in MeV * cm^2/g (see units of "constant", below.)
double dEdx (double T, double pmass, double pcharge,
             double rho, double a, double z, double activation) {
  const double constant = 0.1535; // MeV cm^2/g
  const double me = 0.5110034; //MeV/c^2, Electron mass.
  double E, p, beta, gamma, wmax, excite;
  double term1, term2, term3, bbdedx;

  E = T + pmass;
  p = sqrt(T*T + 2.0*pmass*T);
  beta = sqrt(p*p/E/E);
  gamma = 1.0/sqrt(1.0-beta*beta);
  wmax = 2.0*me*beta*beta/(1.0-beta*beta); // MeV
  excite = activation/1.0e6 ; // Convert to MeV.

  term1 = constant*rho*z*pcharge*pcharge/a/(beta*beta);
  term2 = log(2.0*me*gamma*gamma*beta*beta*wmax/excite/excite);
  term3 = 2.0*beta*beta;

  bbdedx = term1*(term2-term3);
  if ( bbdedx < 0.0 ) {
        bbdedx = 0.0;
  }

  // Add 10% gaussian noise:
  bbdedx += 0.1*sqrt(bbdedx)*normal();
  return (bbdedx);
}
```

Program 2: Visualizing the Results

Your next program will be named **`visualize.cpp`** and it will help us see how much total energy our beam of protons has deposited at various points along its path. To do this, you'll make what's called a "weighted histogram".

In Chapter 7 we learned about histograms, which are graphs that tell us which values in our data occur most frequently. We imagined dropping marbles into bins to count how many times we'd seen a data value within a particular range.

Think about a similar situation: Imagine you're the owner of a restaurant, and you're concerned about wasting water. You've noticed that sometimes full glasses of water are left on tables after the diners have left. You suspect that some of your wait staff are filling glasses too often. To investigate, you get five large beakers, one for each of your waitpersons. Whenever diners leave, you dump their leftover water into the beaker representing that table's waitperson.

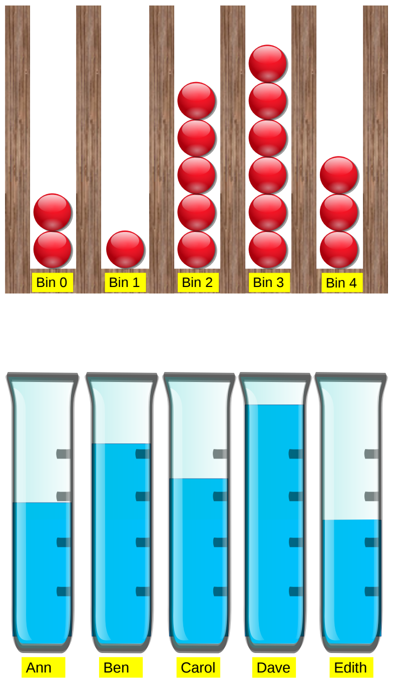


Figure A.29: The top histogram just counts things. The bottom histogram gives each thing a different weight. A weighted histogram doesn't just add 1 for each thing. Instead, it adds some "weight", given by a property that we're interested in (how much water is in a glass, for example). These weights generally won't be integers.

As you can see in Figure A.29, this is almost the same as the histograms we've made before, but instead of putting an integer number of marbles into each bin, we're pouring some (non-integer) amount of water into a beaker. We can think of this as a "weighted" histogram. Instead of just counting each glass as "1 glass", and adding "1" to our histogram, we're giving the glasses different weights, depending on how much water they contain, and adding that weight to the histogram.

In your `visualize.cpp` program, you'll make a weighted histogram that shows how much total energy our proton beam deposited at various points along its path. Your program will be very similar to Program 7.1 in Chapter 7.

Again, to get you started, your physicist friend has taken a break from her busy schedule and written part of the program for you (see Program A.9 below). Notice that she's defined a 100-element array, `hist`, to hold the histogram data. Also notice that this is an array of `double` values instead of integers, since we're making a weighted histogram.

Program A.9: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  const int nbins = 100;
  double hist[nbins];
  FILE *input;
  FILE *output;

  // Gotta go give a lecture. You'll have to
  // write the rest of the program.
}
```

Like the preceding program, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters[13]. Its usage will be:

[13] See Sections 9.15 & 9.16 of Chapter 9.

```
./visualize xmin xmax input output
```

where `xmin` and `xmax` are the minimum and maximum depth we're interested in, in centimeters, `input` is the name of a file produced by your `simulate` program, and `output` is the name of a file into which your program will write the histogram data.

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a depth instead of a bin number (see below for instructions about converting bin number to distance). The second column will be the total energy deposited at that depth, in MeV.

To make the histogram, the program should proceed as follows:

1. Make sure the program sets all of the bins to zero at the beginning.
2. Determine the `binwidth`, like this:

   ```
   binwidth = (xmax-xmin)/nbins;
   ```

3. Next, use a `while` loop to read data from the input file. Each line of the file will contain three values: `x`, `de`, and `energy`.
4. Determine which bin this x value belongs in, as Program 7.1 does.
5. Be sure to keep a count of the number of over/underflows, as Program 7.1 does.

Figure A.30: A painting by Gretchen Andrew, from her series "Malignant Epithelial Ovarian Cancer", which aims to "humanize the experience of having cancer".

6. If it's not an over- or underflow, add the value of `de` to this bin. (Note that this is different from Program 7.1, which just adds 1 to the bin.)

7. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the depth represented by that bin, and the total amount of energy deposited within it. The depth can be calculated from the bin number, like this:

   ```
   depth = xmin + binwidth*(0.5+i);
   ```
   where `i` is the bin number.

8. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Run your program like this to make a histogram of the data you produced earlier:

```
./visualize  0 10 100-mev.dat hist100.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "hist100.dat" with lines
```

The result should look like Figure A.31.

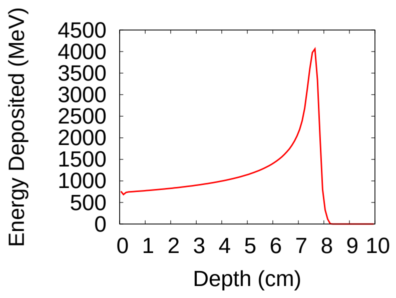


Figure A.31: The total energy deposited at each depth by a 1,000 100-MeV protons.

Program 3: Analyzing the Data

Your last program will be called **`analyze.cpp`**. It will read data produced by your first program and determine how much total energy was deposited in the patient's body, and how much energy was deposited in the tumor.

Like the preceding programs, this one should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze input tcenter tsize
```

Where "`input`" is the name of a data file produced by your `simulate` program, "`tcenter`" is the depth of the center of the tumor, in cm, and "`tsize`" is the size of the tumor, in cm.

Program A.10: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {

        // Ack! My lab is on fire (again)!
        // You're on your own here!

}
```

Once again, your physicist friend has written the first part of the program for you, as shown in Program A.10. She didn't have time for much, but you shouldn't have any trouble completing it. Here's how to do it:

1. First, make sure you define two `double` variables to keep track of the total amount of energy and the amount of energy deposited in the tumor. Make sure both of these are set to zero initially.
2. Next, you'll need to find the depth at which the tumor begins, and the depth at which it ends. These can be found from `tcenter` and `tsize`, like this:

   ```
   xmin = tcenter - tsize/2.0;
   xmax = tcenter + tsize/2.0;
   ```
3. Use a `while` loop to read data from the input file. Each line of the file will contain three values: `x`, `de`, and `energy`.
4. Each time you read a `de` value, add it to the total energy.
5. If `x` is between `xmin` and `xmax`, also add `de` to the amount of energy deposited in the tumor.
6. After reading all of the data, print your results in a nice way that tells the user the total energy and the energy in the tumor. Also tell the user what *fraction* of the total energy was deposited in the tumor, expressed as a percentage. Note that you can tell `printf` to print a percent sign by writing %%.

After you've written your program, run it like this:

```
./analyze 100-mev.dat 8 2
```

This tells the program to read the data for 100 Mev protons that you produced with your `simulate` program, and look at the amount of energy that would end up in a two-centimeter-thick tumor located at a depth of eight centimeters. The program's output should look something like this:

Figure A.32: A Russian "Proton" rocket.
Source: Wikimedia Commons

Figure A.33: The BBC Micro"Proton" computer.
Source: Wikimedia Commons

Figure A.34: A 2016 "Proton Persona" automobile.
Source: Wikimedia Commons

```
Total energy deposited: 102252.422287 MeV
Energy deposited in tumor: 28645.976102 MeV
Fraction deposited in tumor: 28.014961%
```

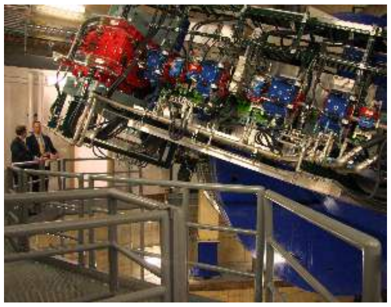


Figure A.35: Proton therapy is a valuable treatment for some types of cancer. It's becoming more widely used, with over 100 treatment centers online now or in planning. Shown above are a facility in Orsay, France (top) and the Mayo Clinic in the US (bottom). The cost, while still significant, is coming down. The ability to minimize radiation damage to surrounding tissues makes it particularly appealing in pediatric cases, where collateral radiation damage can have long-term effects on development.

Results

Using the tools you've written you could find the proton energy that best suits your patient's needs. For example, you could simulate protons of several energies using your `simulate` program:

```
./simulate 1000  50  50-mev.dat
./simulate 1000  75  75-mev.dat
./simulate 1000 100 100-mev.dat
./simulate 1000 125 125-mev.dat
```

then take a look at the energy distribution created by each energy:

```
./visualize 0 10  50-mev.dat hist50.dat
./visualize 0 10  75-mev.dat hist75.dat
./visualize 0 10 100-mev.dat hist100.dat
./visualize 0 10 125-mev.dat hist125.dat
```

You'd see distributions like those shown in Figure A.23 in the introduction. Each distribution has a distinct peak, called the "Bragg peak", near the end of the proton's path. If you saw that one of these peaks lies in the region of the tumor, you might use your `analyze` program to see what fraction of the energy would go into the tumor, like this:

```
./analyze 100-mev.dat 8 2
```

Congratulations, Doctor! You've helped a patient along the road to recovery.

If you're interested in learning more about proton beam therapy, you can find information here:

- Proton Therapy, from Wikipedia.
- The evolution of proton beam therapy: Current and future status, from the NIH's National Center for Biotechnology Information.
- The physics of proton therapy, by Wayne D Newhauser and Rui Zhang.

Project 4: Population Explosion

Introduction

Imagine that a derelict boat washes up on the shore of an uninhabited island. Aboard the boat is a crew of ten rats, all grateful to be on dry land again. Finding plenty of food and water on the island, the happy rats settle down and begin raising families[14].

Boat (1922-1928), Adriano de Sousa Lopes.
Source: Wikimedia Commons

[14] This is reminiscent of the famous radio drama Three Skeleton Key, first broadcast in 1949. If you want to hear a scary story, you can listen to it here: mp3 at archive.org

We might wonder how rapidly our rat population grows in their new island home. Common brown Norway rats are known to have a very high reproductive rate of 0.015 offspring per day. In a perfect environment, we might expect their population to grow over time like this:

$$N(t) = N_0 e^{0.015t}$$

where $N(t)$ is the population after t days, given an initial population of N_0. If we graphed the population over a few years, we'd see something like Figure A.36.

This predicts a rat population of 6 trillion trillion after 10 years! Clearly that's unrealistic. Although there are a lot of rats in the world, their total population is probably only a few billion[15].

Floating from place to place like this (a phenomenon called "rafting") is one way organisms colonize new territories. About 50 million years ago the first lemurs floated on wind-swept debris across the Mozambique Channel from the African mainland to the island of Madagascar. In 1995, a dozen iguanas floating on trees uprooted by a hurricane colonized the previously iguana-fee Caribbean island of Anguilla.
Sources: Wikimedia Commons and Wikimedia Commons.

[15] *See https://www.worldatlas.com/articles/how-many-rats-are-there-in-the-world.html*

The problem is that our estimate assumes that birth and death rates will stay the same as the population grows. Observations of the natural world show that this isn't really the case. For example, populations typically share a limited amount of food and other resources. As the population grows, food is harder to find and some individuals die of starvation. Malnutrition also throttles population growth by reducing birth rates. Typically death rates increase and birth rates decline as populations grow. Taking these effects into account, a more realistic graph of our rat population might look like Figure A.37.

This graph shows the population initially increasing, but then levelling off at some constant value. This value (called the *carrying capacity* of the environment) is the population at which the birth rate is equal to the death rate. When these rates are equal, the population no longer increases. The S-shaped curve of this graph is called a *logistic curve* and is typical of the growth of a population colonizing a new, initially resource-rich, environment.

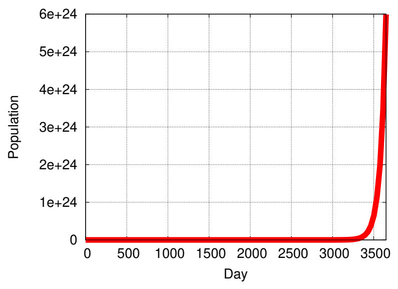


Figure A.36: Rat population given by the equation $N(t) = N_0 e^{0.015t}$.

The Assignment

Now consider a post-apocalyptic scenario where a group of 100 humans is stranded on an island. The island is a pleasant place where the plants and animals could easily provide food and shelter for a population of 1,000 humans. Resigned to their fate, the humans settle down and begin making the best of a bad situation. Ultimately, they have children who grow up knowing no home but the island. These children have grandchildren, and so on down the generations.

Your task in this project is to write three programs that simulate, visualize, and analyze the growth of such a population.

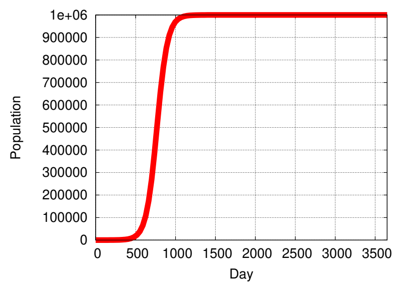


Figure A.37: Rat population with limited resources.

In order to write a program to model the population's growth, we'll need to know how birth and death rates change as the population increases. The shape of the functions governing birth and death rates will vary from one species to another, and will generally depend on many environmental factors. For the purpose of our simulation, though, let's assume that these rates depend solely on the amount of food available per individual. When food is plentiful, the birth rate is high and the death rate is low. In times of famine, the birth rate is low, and the death rate is high.

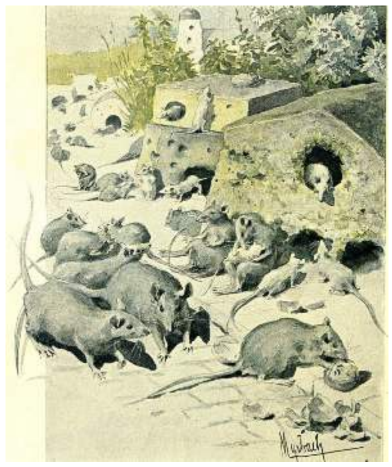


Illustration from Jules Verne's story La Famille Raton, written in 1886.

Source: Wikimedia Commons

We'll assume that we're told the total food-producing capacity of the environment, in terms of the number of individuals that can be fully fed. To find each person's share of this bounty (his or her *ration*), we can just divide the total amount of food by the number of people. Birth and death rates will be functions that depend on this ration.

Figure A.38 shows the shapes of the two functions we'll use. These functions give the annual probability of dying or having offspring when the ration has various values. When the ration is 1, everybody is well fed: the annual probability of having offspring is at its maximum, and the probability of dying is at some minimum value due purely to accident, disease, or old age. As the ration approaches zero, the probability of dying approaches 1 (100%) and the probability of giving birth trails off to some tiny value. We'll assume that if the ration is greater than 1, the birth and death rates stay constant at the same values they had when the ration was 1. (We'll ignore any possible ill-effects of overeating!)

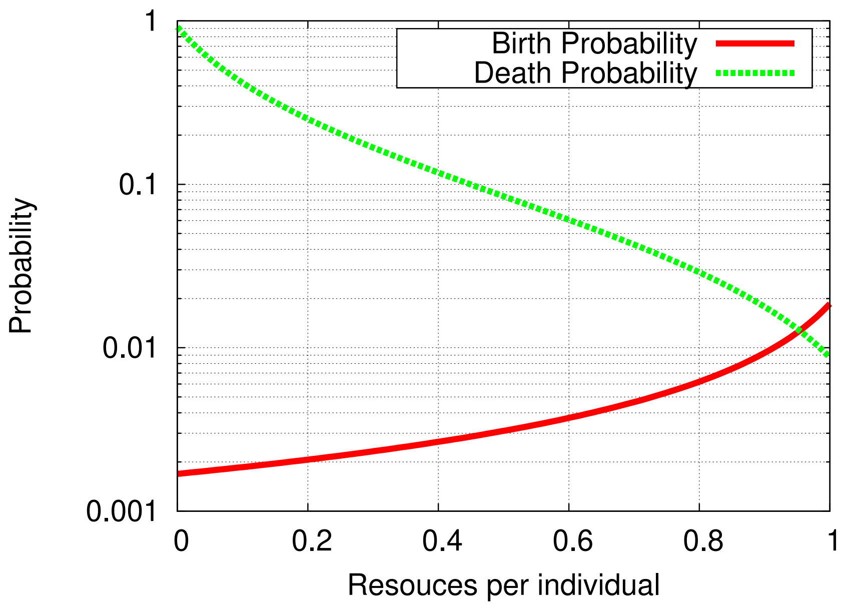


Figure A.38: Annual probability of birth or death as a function of ration.

The birth probability function we've graphed looks like this:

$$b(r) = \begin{cases} \dfrac{b_{max}}{10(1-r)+1} & \text{if } r \le 1 \\ b_{max} & \text{if } r > 1 \end{cases} \tag{A.1}$$

and the death probability function looks like this:

$$d(r) = \begin{cases} d_{min} + \dfrac{1}{10r+1} - 0.09 & \text{if } r \le 1 \\ d_{min} & \text{if } r > 1 \end{cases} \tag{A.2}$$

where r is the ration, b_{max} is the maximum probability per year of having offspring, and d_{min} is the minimum probability per year of dying.

Now let's get programming! You'll be writing three programs: **`simulate.cpp`**, **`visualize.cpp`**, and **`analyze.cpp`**. The first will simulate the population's growth, the second will help visualize the results, and the third will do some statistical analysis on them.

Program 1: Simulating Population Growth

Your first job will be to write a program named **`simulate.cpp`** that simulates the growth of the population over some number of years and writes its results into a file.

Théodore Géricault's *The Raft of the Medusa* (1818-1819).
Source: Wikimedia Commons

Our simulation program's strategy will be this: We'll give the program an initial population, the total amount of food, the values of b_{max} and d_{min}, and tell it how many years to simulate. The program will then loop through the years, one at a time. For each year it will loop through all of the individuals in the population. For each person, the program will check to see whether the person has offspring during that year and whether the person dies during that year, using the $b(r)$ and $d(r)$ functions in Equations A.1 and A.2 above. If the person dies, the population will be reduced by one. If the person has offspring, the population will increase by one[16].

[16] for simplicity, we're assuming one child per person per year, at most.

The program should accept all of its parameters on the command line, as described in Section 9.15 of Chapter 9. The usage should be:

```
./simulate population food bmax dmin nyears outfile
```

where:

- `population` is the initial population.
- `food` is the total amount of food the island can produce, in terms of the number of people who can be well-fed.
- `bmax` is b_{max} from Equation A.1 above.
- `dmin` is d_{min} from Equation A.2 above.
- `nyears` is the number of years to simulate.
- `outfile` is the name of a data file into which the program will write its results.

To get you started, I've already written some of the program for you (see Program A.11). All you need to do is complete the program by filling in `main` and the two functions `birthprob` and `deathprob`. Notice that I've added a handy function named `rand01` near the top of the program[17]. It can be used to generate a random number between zero and one.

[17] This function is described in Section 11.4 in Chapter 11.

Program A.11: simulate.cpp

```
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double birthprob ( double bmax, double ration ) {
        // Insert function here.
}

double deathprob ( double dmin, double ration ) {
        // Insert function here.
}

int main ( int argc, char *argv[] ) {
  double population;
  double popgrowth;
  int nyears;
  int year;
  int individual;
  double food;
  double ration;
  double bmax, dmin;
  double bprob, dprob;
  FILE *output;

  // Insert program here.
}
```

To complete the program, you'll need to add code to do the following:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[18].

2. Convert the command-line arguments into the variables `population`, `food`, `bmax`, `dmin`, and `nyears` by using the **`atoi`** and **`atof`** functions. The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

[18] See Section 9.16 of Chapter 9 for an example of how to do this.

```
output = fopen( argv[6], "w" );
```

Crowded Boardwalk, Atlantic City, New Jersey (1910).
Source: Wikimedia Commons

3. Your program will need a pair of nested "`for`" loops: An outer loop that goes through all the years, and an inner loop that goes through all of the individuals in the population and, for each one, checks to see whether that person died or had offspring during the current year.

   The outer loop might start like this:

   ```
   for ( year=0; year<nyears; year++ ) {
   ```

   and the inner loop might start like this:

   ```
   for ( individual=0; individual<population; individual++ ) {
   ```

4. At the beginning of each year the program will need to do a few things:

   - Find the `ration` by dividing `food` by `population`
   - Find the probability of having offspring, which we'll call `bprob`, by using the `birthprob` function defined at the top of the program (we'll describe this and the `deathprob` function below).
   - Find the probability of dying, which we'll call `dprob`, by using the `deathprob` function defined at the top of the program.
   - Set `popgrowth` to zero. We'll use this variable to keep track of how much the population grows during the current year. (If there are more deaths than births, this number might be negative, but that's OK.)

5. Inside the inner loop we'll do some things for each individual who's currently in the population:

   - Check to see if that person had offspring during the year. We do this by using the `rand01` function to give us a random number between zero and one, and then checking to see if that number is less than `bprob`. If it is, then we add 1 to `popgrowth`, indicating that a person has been added to the population this year.
   - Similarly, we check to see if the person died this year. We do this by looking to see if `rand01` gives us a number less than `dprob`. If it does, then we subtract 1 from `popgrowth`, indicating that a person has been removed from the population. (Remember that it's OK for `popgrowth` to be negative.)

6. At the end of each year, we add `popgrowth` to `population` to get

the new value for `population`, and we write data about this year into our output file. The values of `year`, `population`, `popgrowth`, `bprob`, `dprob`, and `ration` should be written to the file, in that order, separated by spaces[19].

[19] See Chapter 5 for information about writing data into a file.

7. The last step in completing the program is to write the two functions `birthprob` and `deathprob`. The `birthprob` function takes the value of `bmax` and `ration` and uses the relationship shown in Equation A.1 to calculate the birth probability. Similarly, `deathprob` uses `dmin` and `ration` to calculate the death probability, as given by Equation A.2. Note that you'll need an `if/else` statement in each of these functions, to deal with the cases when `ration` is less than one or greater than one.[20]

Edoardo Matania, *Die geschlossene Bank* (1870s).
Source: Wikimedia Commons

[20] See Chapter 3 for information about writing `if/else` statements, and Chapter 9 for information about writing functions.

After you've completed your program, compile it and run it three times, with these arguments:

```
./simulate 2000 1000 0.0182 0.0077 1000 hipop.dat
./simulate  500 1000 0.0182 0.0077 1000 medpop.dat
./simulate  100 1000 0.0182 0.0077 1000 lopop.dat
```

The three simulations are the same except for the starting population. In the first one, the initial population is higher than the amount of food available in the environment (2,000 people, but only food enough for 1,000). The second simulation has an initial population of 500, with the same amount of food, and the third simulation shows what happens when the initial population is only 100. The values used for `bmax` and `dmin` are actual current worldwide average values for birth and death rates in human populations[21]. Each of the simulations tracks the population growth over a period of 1,000 years.

[21] CIA World Factbook, estimated values for 2018.

You can plot your results by giving *gnuplot* the command:

```
plot [0:300] "hipop.dat" with lines, "medpop.dat" with lines, "lopop.dat" with lines
```

which shows just the first 300 years. The result should look something like Figure A.39. Notice that, in all cases, the population eventually settles down to a stable level that's slightly greater than 1,000 individuals.

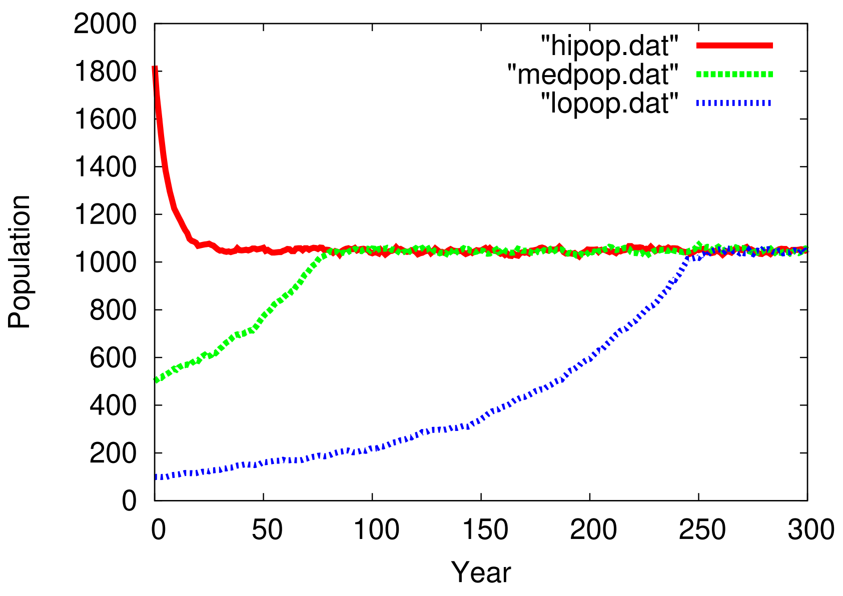


Figure A.39: Population growth when there is sufficient food for 1,000 people, starting with populations of 100, 500, and 2,000 people.

Program 2: Visualizing the Stable Population

So now we know that the island's population always tends toward a particular value, but what is that value exactly? Let's start to investigate this by writing a program to visualize the data from our simulations in a different way. The program will be called **`visualize.cpp`** and it will let you make graphs like the one shown in Figure A.40. This graph shows population on the horizontal axis, divided into 50 bins. The vertical axis shows how many years had a population within each bin.

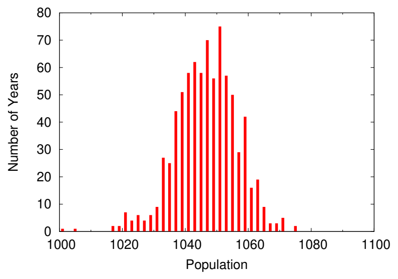


Figure A.40: Histogram of population values from `lopop.dat`.

This figure is a histogram, like the ones we discussed in Chapter 7. Your program will be similar to Program 7.1 in that chapter. Again, to get you started, I've written part of the program for you (see Program A.12 below). Notice that I've defined a 50-element array, `bin`, to hold the histogram data.

Program A.12: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {
  const int nbins=50;
  int bin[nbins];
  double binwidth;
  int binno;
  int overunderflow=0;
  int i;
  FILE *input;
  FILE *output;

  // Insert program here.

}
```

"Cynicus", *The last car for Miramar* (c. 1910).
Source: Wikimedia Commons

Like the preceding program, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize popmin popmax inputfile outputfile
```

where `popmin` and `popmax` are the minimum and maximum population you want to include in your histogram, `inputfile` is the name of a file produced by your `simulate.cpp` program, and `outputfile` is a file into which your new program will write the histogram data.

The input and output files can be opened like this[22]:

```
input  = fopen(argv[3],"r");
output = fopen(argv[4],"w");
```

[22] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a population value instead of a bin number (see below for instructions about converting bin number to population). The second column will be the number of years in that bin.

To make the histogram, the program should proceed as follows:

1. First, determine the `binwidth`, like this:

   ```
   binwidth = (popmax-popmin)/nbins;
   ```
2. Next, use a `while` loop to read data from the input file[23]. Each line of the file will contain six values: `year`, `population`, `popgrowth`, `bprob`, `dprob`, and `ration`. The first value is an integer, and the other five are `double`s.

[23] See Chapter 5 for information about reading data from files.

3. Determine which bin this population value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the range of our histogram is `popmin` to `popmax`, the bin number will be:

   ```
   binno = (population-popmin)/binwidth;
   ```

4. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the population value represented by that bin, and the value of `bin[i]`. The population value can be calculated from the bin number, like this:

   ```
   population = popmin + binwidth*(0.5+i);
   ```

where i is the bin number.

5. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Run your program like this to make a histogram of the data you produced earlier. Start out by looking at population values between 0 and 1,100:

```
./visualize 0 1100 lopop.dat visualize.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize.dat" with impulses lw 5
```

The graph should look like Figure A.41. Now let's zoom in on the region around a population of 1,000 by running your visualize program again, this time setting popmin to 1,000 and popmax to 1,100:

```
./visualize 1000 1100 lopop.dat visualize.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize.dat" with impulses lw 5
```

The result should look like Figure A.40 at the beginning of this section.

As you can see, the population values cluster around 1,040 or so, slightly above the 1,000 individuals that can be fully fed. Think for a minute about what this means: We're finding that the population tends to settle in at a level where there's not quite enough food to go around. This raises the death rate and lowers the birth rate until the two rates are equal. In your final program you'll find an exact value for this equilibrium population.

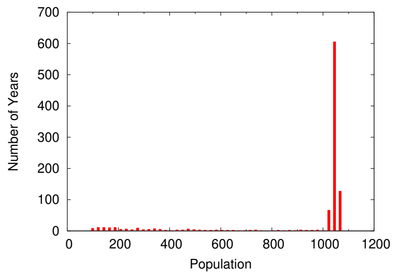


Figure A.41: A histogram of population values from lopop.dat in the range 0 to 1,100.

Program 3: Finding the Mean and Standard Deviation

Your third program will be named **analyze.cpp**. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first two programs, analyze should accept all of its parameters on the command line, and give users a helpful message if they

Cualquier parada de cualquier línea de cualquier tranvía de Madrid
—¡Y sin ver una pareja de guardias!
—¿Cómo quiere usted verla, si hace tres horas que no viene ningún tranvía?

Manuel Tovar Siles, *"Any stop of any line of any tramway of Madrid"* (1920).
Source: *Wikimedia Commons*

don't give it the right number of arguments. The usage should be:

```
./analyze popmin popmax inputfile
```

where `popmin` and `popmax` delimit the range of population values you're interested in, as they do in your preceding program, and `inputfile` is the name of a data file produced by your `simulate.cpp` program.

The output of the `analyze` program should look like this:

```
Mean population = 1046.245636
Std. dev. = 9.590271
```

Again, I've written some of the program for you (see Program A.13). You'll just need to fill in `main`.

Program A.13: analyze.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  int year;
  double population, popgrowth;
  int popmin, popmax;
  double dprob, bprob, ration;
  double sum=0;
  double sum2=0;
  double mean, stddev;
  int nvalues=0;
  FILE *input;

  // Insert program here.

}
```

To analyze the data, the program should proceed as follows:

1. First, open the data file for reading. See Program 7.5 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation.
2. Like your `visualize.cpp` program, this new program should use a `while` loop to read data from the input file. Each line of the file will contain six values: `year`, `population`, `popgrowth`, `bprob`, `dprob`, and `ration`.
3. Unlike Program 7.5, our new program will need to check to see whether a population value is between `popmin` and `popmax` before adding it to `sum` and `sum2`.

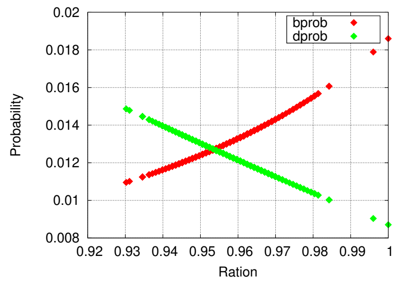


Figure A.42: `bprob` and `dprob` versus `ration`, from our `lopop.dat` simulation.

If you run your program like this:

```
./analyze 1000 1100 lopop.dat
```

you should see that the mean population value is about[24] 1,046, which corresponds to a ration of $1,000/1,046$ or about 95.6%. If we plot our simulation's birth and death probabilities versus ration, using *gnuplot* commands like this:

[24] The value you see will vary, because the simulation uses random numbers

```
set xrange [0.92:1]
set yrange [0.008:0.02]
plot "lopop.dat" using 6:4, "" using 6:5
```

(column 6 of our output file is `ration`, column 4 is `bprob` and column 5 is `dprob`) we would see something like Figure A.42. This confirms that birth probability and death probability are equal when the ration is around 95.6%, the ration where our analysis shows that our population is stable.

Thomas Malthus (left) and Charles Darwin.

Conclusion

In 1798, English scholar Thomas Robert Malthus wrote *An Essay on the Principle of Population*, in which he observed that English populations were growing more rapidly than the increase in agricultural production. Malthus anticipated the phenomenon we've explored in this project: Populations tend to grow to the point where resources are no longer sufficient for everyone, causing death rates to increase and birth rates to decline until the population stabilizes. Malthus's ideas about competition for scarce resources were an inspiration for Charles Darwin's theory of evolution by natural selection.

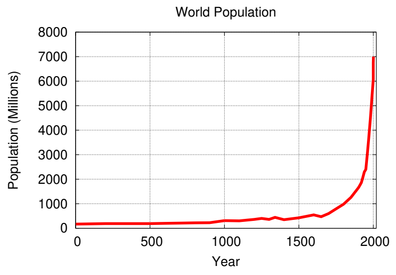


Figure A.43: World population growth.
Source: Wikimedia Commons

Such plateaus in population have occurred many times in human history, but have typically been temporary and limited to a geographic region. In Malthus's time, for example, England was heading for a shortage of food, while Russia had an overabundance of agricultural capacity. Malthus expected these shortages to last only until new agricultural land had been developed, or until improvements in agriculture increased the yield of existing land.

Arnold Böcklin, *The Isle of the Dead*, third version (1883) and *The Isle of the Living* (1888).
Source: Wikimedia Commons and Wikimedia Commons

Globally, the human race has shown no slowing of its exponential growth rate (see Figure A.43). So far, development of new land and improvements in agricultural science have, on average, kept ahead of population growth, but humans also depend on fresh water, shelter, and other limited resources. Some people estimate[25] that the global

[25] *See https://en.wikipedia.org/wiki/Planetary_boundaries.*

population already exceeds the Earth's carrying capacity[26].

What will happen to our islanders? Will they find a clever way to avoid a "Malthusian crisis?" Let's wish them luck, and the same for the inhabitants of this island Earth.

[26] Apparently we're not running short of physical space. John Brunner's novel Stand on Zanzibar notes that 7 billion people (the current population of Earth) could fit on the island of Zanzibar – if they stood shoulder to shoulder!

Project 5: Yard Sale!

Introduction

Every August a 630-mile-long yard sale stretches from Michigan to Alabama along US Highway 127. It's called the "World's Longest Yard Sale". Thousands of people visit it. In the early 21^{st} century Economists began to realize that yard sales like this provide a good model for the whole world's economy. By simulating the interactions between buyers and sellers at such a sale, we can make predictions about wealth distribution that match data observed in the real world. The trick is to assume that the economy is made up of many, many one-to-one interactions where a buyer and a seller exchange some wealth.

Source: Wikimedia Commons

Economists gauge a person's wealth by looking not just at how much money you have, but also the value of the goods you own. Imagine that I'm a vendor at the yard sale and you're a shopper. If you pay me five dollars for a toaster, an economist would traditionally have said that there was no net change in either person's wealth: I have your five dollars, but you now have a toaster worth five dollars.

Source: Wikimedia Commons

But is it really? What if, when you get home, you find that the toaster doesn't work. Then you really have a toaster worth less than five dollars, but I still have your money. We could say that you've lost some wealth by giving me five dollars and getting something worth less than that, and I've gained some wealth by getting five dollars in exchange for a worthless toaster. In that case, wealth has flowed from you to me, making you poorer and me richer.

This happened because you mis-judged the value of the toaster. Traditionally, economists have assumed that shoppers are good at judging the value of things, and economic models have used this assumption to make predictions about the economy. But recently economists have become interested in models that take into account the fact that buyers and sellers often make mistakes about the value of things. A seller might sell a "worthless" painting for five dollars, only to find later that it's a valuable Picasso, or a buyer might pay a lot of money for a "Rolex" watch only to find that it's a cheap knock-off.

Anirban Chakraborti, who first proposed the "yard sale" model of economics in 2002.

The mistakes we make are usually small, but we probably always make some small error when we assign a value to something we buy or sell. The effect of this is that wealth flows around the economy, with some people becoming more wealthy than others. If everyone had an equal chance of gaining or losing an equal amount because of these mistakes, we might assume that, on average, they don't matter, and that any inequalities of wealth would even out over time. But the yard sale models that Economists have developed, and which match real-world economic data, make an additional assumption: They assume that the biggest possible mistake in each transaction is the total wealth of the *poorest* person involved in the transaction. (A person with only one dollar can't buy the five-dollar toaster, no matter whether the toaster is broken or not.) By doing this we're ignoring people who win the lottery or accidentally sell a Picasso for five dollars, but it turns out that those situations are rare and don't have much effect on the economy as a whole[27].

[27] These models also assume that wealth of any kind can be exchanged. I can pay you five dollars for that toaster, or I can trade you a record player for it. My wealth includes both the money I have and the value of the items I own.

In this project we're going to write three programs that investigate such a yard-sale model of the economy. The first program (**`simulate.cpp`**) will simulate lots of interactions between buyers and sellers. The second (**`visualize.cpp`**) will visualize the distribution of wealth after some time has passed. The third (**`analyze.cpp`**) will analyze the data and boil it down to a single number that measures how evenly wealth is distributed. Let's get started!

Program 1: Buyers and Sellers

Our first program will be named **`simulate.cpp`**, and it should start out like Program A.14 below. The program will simulate many random transactions between pairs of people, and track the wealth flowing from person to person. We'll assume everybody starts out with the same amount of wealth.

How the Program Works

The program should accept three parameters on the command line: The initial wealth of each person, the number of transactions we want

Program A.14: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char* argv[] ) {
  const int N = 10000;  // Number of people.
  double wealth[N];     // Wealth of each person.
  double wstart;        // Starting wealth of each person.
  double mistake;       // Size of a mistake.
  double flip;          // A random number, used for deciding who made the mistake.
  double ntransactions; // Number of transactions.
  int alice, bob;       // The two people involved in a transaction.
  int poor;             // which of the two people is poorer.
  int i;
  FILE *output;

  srand( time(NULL) );  // Set the seed of the random number generator.

  // Put the rest of the program here!

}
```

to simulate, and the name of a file we want to write our results into. For example:

```
./simulate 100 2e+5 output.dat
```

The first number is the initial wealth of each person, the second is how many transactions we want to simulate[28], and the final argument is the name of the output file we want to create.

[28] This number is in C-style scientific notation. In this example, we've used `2e+5` which is 2×10^5, or 200,000. See Section 4.3 of Chapter 4.

The program should assume that this is a very big yard sale, with 10,000 people swapping money and goods. That's the population of a small town or a rural county. To keep track of how much wealth each person has, it should use an array with 10,000 elements. The wealth of person number `i` will be `wealth[i]`. A person's wealth will generally be a number with decimal places, so `wealth` will need to be an array of `double`s.

We'll start each person out with the same amount of wealth. Let's call it `wstart`. After setting the initial wealths, the program should enter a loop that simulates some number of random transactions. For each transaction, we'll pick two people at random. Let's call them `alice` and `bob`, and their wealths will be `wealth[alice]` and `wealth[bob]`.

After we've picked two people, we need to see which one is poorer by comparing their wealths. Let's have another variable, `poor`, and say that if Alice is poorer, `poor=alice` and if Bob is poorer, `poor=bob`.

Now assume that somebody makes a mistake in the transaction. Remember that we're limiting the size of the mistake to the wealth of the poorer person, so at most the mistake will be `wealth[poor]`. Let's say that the size of the mistake is a random number between zero and one, multiplied by `wealth[poor]`.

Then we "flip a coin" to decide which person, Alice or Bob, benefits from this mistake. We do this by generating a random number between zero and one. If this number is greater than 0.5 Alice wins, otherwise Bob wins. If Alice wins, the amount of the mistake is added to her wealth and subtracted from Bob's wealth. If Bob wins, the mistake is added to his wealth and subtracted from Alice's.

Aaah, wealth! (Portuguese actor António Silva portraying a wealthy man)
Source: Wikimedia Commons

After the program has done the requested number of transactions, it should write the final wealth of each person into the file specified on the command line. The output file should have two columns separated by a space: person number and the wealth of that person.

How to Write the Program

To get you started, Program A.14 shows part of `simulate.cpp`. It includes all of the variables you'll need. You just need to write the middle part, where all the work gets done. To complete the program, you'll need to add code to do the following:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[29].

[29] See Section 9.16 of Chapter 9 for an example of how to do this.

2. Convert the command-line arguments into the variables `wstart` and `ntransactions` by using the **`atof`** function[30].The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

```
output = fopen( argv[3], "w" );
```

[30] Notice that we've chosen to make `ntransactions` a `double`, even though it will always have some integer value. That's because we'll be using large numbers of transactions, and it's convenient to write things like `1e+7` instead of `10000000`, so we don't have to carefully count zeros. C only lets you use scientific notation with `double`s.

3. Next you'll need a "`for`" loop to set the initial wealth of each person to `wstart`.

4. Then you'll need a second "`for`" loop that goes through `ntransactions`

transactions. During each transaction the program will need to do several things:

(a) Pick two random people to be Alice and Bob for this transaction. You might do something like this:

```
alice = rand()/(1.0+RAND_MAX) * N;
bob   = rand()/(1.0+RAND_MAX) * N;
```

Notice that this generates a random number between zero and (almost) one, and then multiplies it by `N`, the total number of people [31].

[31] On rare occasions, at random, it will turn out that "Alice" and "Bob" are the same person, but we won't worry about that. It happens rarely, and it won't affect the results.

(b) Then we need to use an "`if`" statement to check which person has the smaller wealth. Set the variable `poor` to equal either `alice` or `bob`, as appropriate.

(c) Next the program needs to determine a random size for the mistake that's made in this transaction. Remember that it should be an amount between zero and `wealth[poor]`. One way to do this is:

```
mistake = wealth[poor]*rand()/(1.0+RAND_MAX);
```

(d) As the last thing in the loop the program should "flip a coin" to see whether Alice or Bob gets the benefit of the mistake. To do this, generate a random number between zero and one, and then use an "`if`" statement to see if it's greater than 0.5. If it is, then Alice wins. Transfer `mistake` amount of wealth from the loser to the winner.

5. After the loop is done, the program should write its results into a file[32]. This should be done with a third "`for`" loop. For each person, there should be one line in the file with two numbers separated by a space. For person "`i`" the numbers should be `i` and `wealth[i]`.

[32] See examples like Program 5.3 in Chapter 5.

Running the Program

After you've created the program, run it several times to make some output files that you'll use with the next two programs. Try these commands:

```
./simulate 100 0 simulate-0.dat
./simulate 100 1e+4 simulate-10K.dat
./simulate 100 1e+6 simulate-1M.dat
./simulate 100 1e+9 simulate-1G.dat
```

Those commands will create four output files representing a starting wealth of $100 for each person, and simulating 0 transactions, 10 thousand transactions, then 1 million and 1 billion transactions. If you look inside any of these files with *nano* you should see two columns of numbers. The first column will be the person number (an integer) and the second column will be that person's wealth (a number with decimal places) after the specified number of transactions. You can graph the results with *gnuplot* if you like, using *gnuplot* commands like:

```
plot "simulate-1M.dat" with impulses
```

You should see graphs like the ones in Figure A.44.

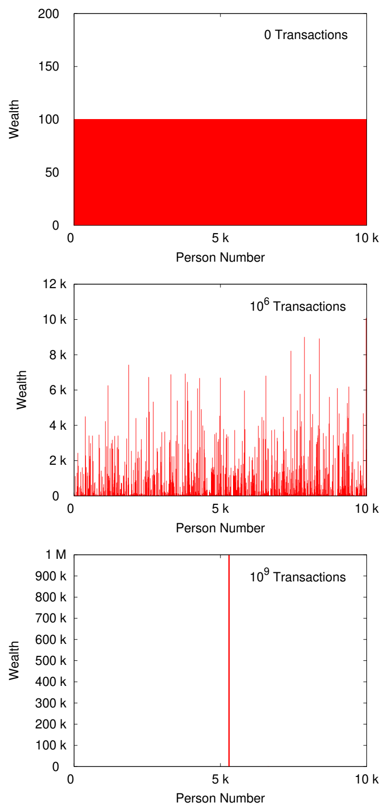


Figure A.44: The top graph shows the distribution of wealth after 0 transactions. Everybody has the same amount of money ($100). The middle graph shows the situation after 1 million transactions. Now some people have a lot more wealth than others. The bottom graph show the situation after 1 billion transactions. Now one random, lucky person has *all* of the money, and everyone else has nothing!

Look at what happens as the number of transactions increases. At zero transactions everybody has the same amount of money. After a million transactions wealth has spread around, and some people have thousands of dollars. This isn't too surprising. But after a billion transactions we find that one lucky person has *all* of the money, and nobody else has any! If you run this billion-transaction simulation several times, you'll find that one person always ends up with all the money, but it will be a different person each time.

That's something that economists have found to be an inescapable property of the yard sale model: If you let it run long enough one person inevitably ends up with all the wealth.

Program 2: Visualizing at the results

Let's take a closer look at how our simulation distributes wealth. To investigate this, we might make a graph that shows wealth across the bottom, divided into equal-sized ranges like $0-$25, $25-$50, $50-$75, and so on. On the vertical axis we could show how many people have a wealth in each range. We learned in Chapter 7 that a graph like this is called a histogram.

The next program you'll write is named **visualize.cpp** and it will make histograms of the simulated wealth data created by your first program. The new program will be similar to Program 7.1 in Chapter 7. It should start out like Program A.15 below.

Program A.15: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {
  const int nbins=100;
  int bin[nbins];      // How many people are in each wealth range.
  double binsize;      // Width of wealth ranges.
  int n;               // "Person number".
  double wealth;       // Wealth of that person.
  double maxwealth;    // Maximum wealth we want to graph.
  int binno;           // Bin number for a person, based on person's wealth.
  int overunderflow=0; // How many people were outside the range of the graph?
  int i;
  FILE *input;
  FILE *output;

  // Insert program here.

}
```

How the Program Works

Like the preceding program, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize maxwealth input.dat output.dat
```

where `maxwealth` is the maximum wealth you want to include in your histogram, `input.dat` is the name of a file produced by your `simulate.cpp` program, and `output.dat` is a file into which your new program will write the histogram data.

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a wealth value instead of a bin number (see below for instructions about converting bin number to wealth). The second column will be the number of people who have that amount of wealth.

Postcard: "Youth poverty at the beginning of the 20th century in Europe."
Source: Wikimedia Commons

How to Write the Program

To make the histogram, the program should proceed as follows:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the

program should print out a friendly usage message and then stop without trying to do anything else.

2. Convert the first command-line argument into the variable `maxwealth` by using the **`atof`** function. The other two command-line arguments (the input and output file names) don't need to be converted. The input and output files can be opened like this[33]:

```
input  = fopen(argv[2],"r");
output = fopen(argv[3],"w");
```

[33] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

3. Next, determine the `binwidth`, like this:

```
binwidth = maxwealth/nbins;
```

4. Use a "`for`" loop to set all the elements of `bin` to zero.

5. Now use a `while` loop to read data from the input file[34]. Each line of the file will contain two values: A person number and that person's wealth. The first value is an integer, and second is a `double`.

[34] See Chapter 5 for information about reading data from files. In particular, look at Program 5.4.

6. Determine which bin each person's wealth value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the size of each bin is `binwidth`, the bin number will be:

```
binno = wealth/binwidth;
```

7. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the first number is the wealth value represented by that bin, and the second is the value of `bin[i]`. The wealth value can be calculated from the bin number, like this:

```
wealth = binwidth*(0.5+i);
```

where `i` is the bin number. This will give you the wealth at the midpoint of that bin's wealth range.

8. Finally, at the bottom of the output file, write a line beginning with a `#` that tells how many overflows or underflows were seen.

After you've written the program, run it a few times like this to create histograms from the files you created previously, limiting the graph to a maximum wealth of $2,500:

```
./visualize 2500 simulate-0.dat visualize-0.dat
./visualize 2500 simulate-10K.dat visualize-10K.dat
./visualize 2500 simulate-1M.dat visualize-1M.dat
./visualize 2500 simulate-1G.dat visualize-1G.dat
```

You can use *gnuplot* to view the histograms by giving it commands like:

```
set log y
set yrange [0.1:]
plot "visualize-10K.dat" with impulses
```

This will draw a vertical line for each wealth range, with the height of the line indicating the number of people who have a wealth in that range. The first command makes the Y-axis logarithmic. If we didn't do this, we wouldn't be able to the bins that only have a few people in them. Your graphs should look something like the ones shown in Figure A.46.

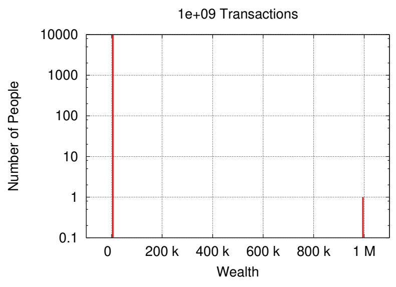


Figure A.45: Wealth distribution after 1 billion transactions, showing wealth up to $1 million.

You can see that the data in the last graphs is starting to run off the right-hand edge of the graph. The total amount of money in our population is $1 million ($100 per person × 10,000 people). Let's graph the data from our longest simulation using this as `maxwealth`. To do that, run your `visualize` program again, like this:

```
./visualize 1000000 simulate-1G.dat visualize-long-1G.dat
```

This extends the wealth scale out to $1,000,000. If you graph the new file with *gnuplot* (again using a logarithmic Y-axis) you should see something like Figure A.45.

This is another way of seeing that only one person ends up with all of the money. The short spike on the right-hand side represents the one person who now has 1 million dollars. The tall spike on the left-hand side is everyone else, with zero dollars[35].

[35] Sometimes after a billion transactions you'll find that there are still two people who have some money. After more transactions, though, one of them always ends up with all of the money.

Program 3: Quantifying Wealth Inequality

Our simulated economy produces severe wealth inequality, but how does it compare to real-life economies? How can we measure the amount of wealth inequality? One way of quantifying it is called the "Gini Coefficient[36]".

[36] See https://en.wikipedia.org/wiki/Gini_coefficient.

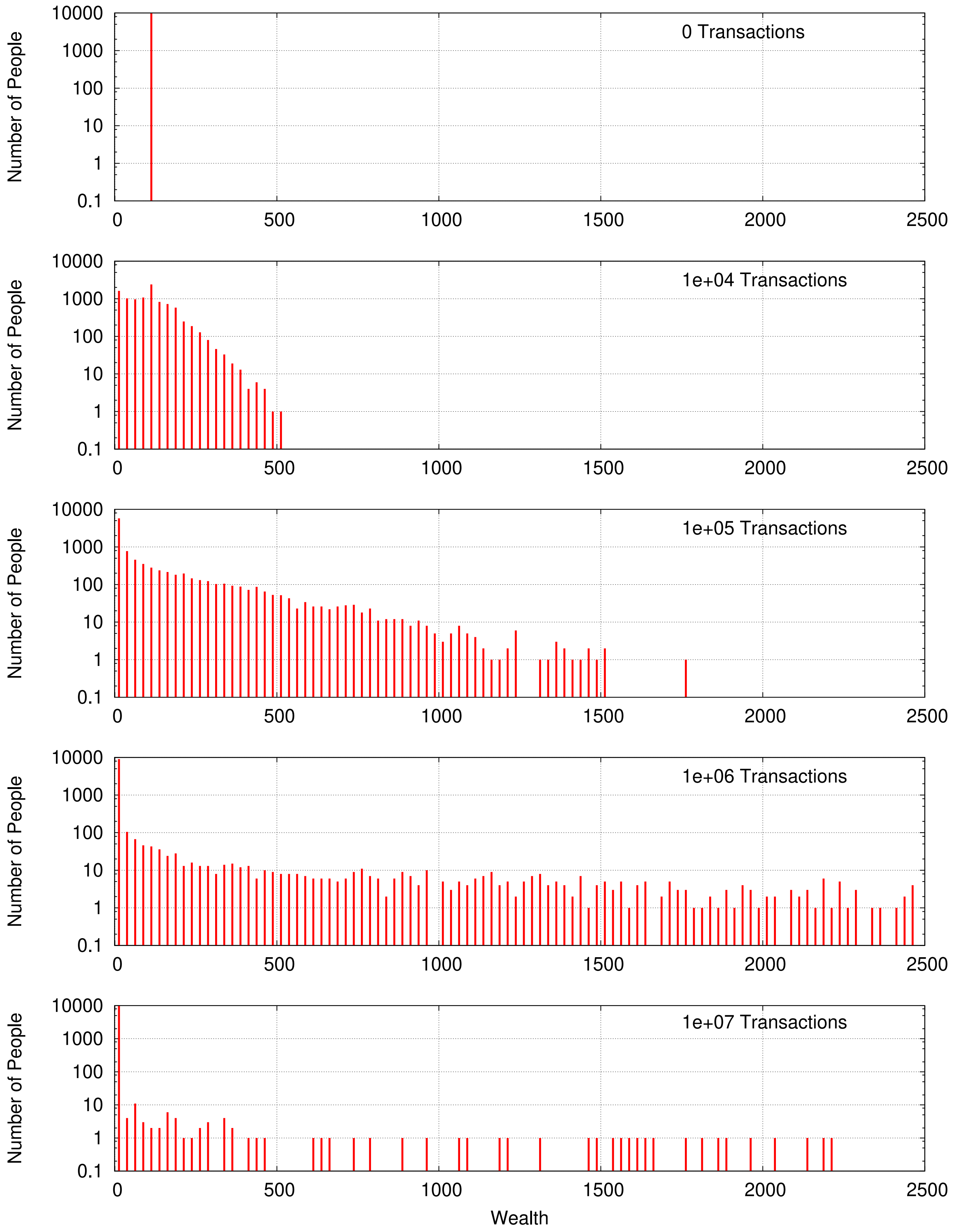


Figure A.46: Histograms of wealth after different numbers of transactions.

The Gini Coefficient starts by measuring the average difference in wealth between any two individuals in the population. (It ignores the sign of this difference by taking the absolute value.) Then it divides the result by the total amount of wealth in the population. A Gini Coefficient of zero corresponds to an economy where everybody has the same amount of wealth. A value of one corresponds to an economy where a single person has all the wealth, and everyone else has nothing. Real-life economies fall somewhere between these two extremes.

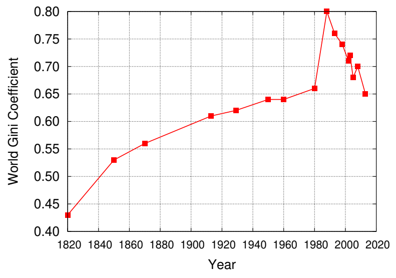


Figure A.47: Estimated world-wide Gini Coefficient, by year. See Milanovic and World Bank in the "Further Reading" section below.

Researchers at the World Bank have estimated values for the world-wide Gini Coefficient for various years, beginning with 1820 (see Figure A.47). The value seems to have risen to a peak of about 0.8 in the 1980s and then begun a downward trend. The current value is about 0.65[37]. Your third program, **`analyze.cpp`**, will read the data produced by your simulation and calculate the Gini Coefficient for your simulated economy.

[37] Note that some writers refer to the "Gini Index", which is just 100 times the Gini Coefficient. That means the current world-wide Gini Index is about 65.

How the Program Works

Like the first two programs, this one should accept arguments on the command line. In this case, there will be just one argument: the name of a data file produced by your `simulate.cpp` program. For example, you should be able to run your latest program like this:

```
./analyze simulate-10K.dat
```

"Children sleeping in Mulberry Street" (detail), by Jacob Riis (1890).
Source: Wikimedia Commons

Your program should start by reading the data from the data file and putting it back into a 10,000-element array called `wealth`, just like the array you used in your first program.

Next your program will need to add up the total wealth of all of the people. You'll need this later for calculating the Gini Coefficient.

After that, you'll need to go through each pair of people in the population, find the difference in their income, and add its absolute value to a sum. You should do this with two nested "`for`" loops. Once the wealth differences have all been added up, you can use that sum and the total wealth to calculate the Gini Coefficient. Mathematically, the Gini Coefficient is defined as:

$$\texttt{gini} = \frac{\sum_{\texttt{i}}\sum_{\texttt{j}} |\,\texttt{wealth[i]} - \texttt{wealth[j]}\,|}{2\texttt{N}\sum_{\texttt{i}}\texttt{wealth[i]}}$$

Program A.16 below shows how your program should start. It contains all the variables you'll need. You just need to fill in the rest of the program.

Program A.16: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char* argv[] ) {
  const int N = 10000;  // Number of people.
  double wealth[N];     // Wealth of each person.
  double sum = 0;       // Sum of all the wealth.
  double sumdiff = 0;   // Sum of wealth differences between pairs of people.
  double gini;          // Gini coeffficient.
  int n;                // ``Person number''
  int i,j;
  FILE *input;

  // Add the rest of the program here.
}
```

How to Write the Program

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else.
2. The only command-line argument (the input file name) doesn't need to be converted. You can just use it directly, like this:

   ```
   input = fopen( argv[1], "r" );
   ```
3. Next you'll need a "for" loop that repeats 10,000 times (the value of N in the program) and reads one line out of the input file each time. The input file has two columns of data: the person number and that person's wealth. That means you should have a statement like this for reading a line from the input file:

   ```
   fscanf( input, "%d %lf", &n, &wealth[i] );
   ```
   As you read each wealth value, add it to the value of sum. This will give you the sum of all the wealth in the population, which you'll need later for calculating the Gini Coefficient.
4. Now the program needs to find the difference in wealth between each pair of people in the population. To do this you'll need a pair of nested "for" loops. Use the fabs function to get the absolute value of the wealth difference, and then add it to sumdiff like this:

   ```
   sumdiff += fabs( wealth[i] - wealth[j] );
   ```

Note that this will actually count each pair of people twice. For example, if `i` is 20 and `j` is 30, the sum will include both `wealth[20] - wealth[30]` and `wealth[30] - wealth[20]`. We'll take care of this later by dividing `sumdiff` by 2 when doing the Gini Coefficient calculation.

5. Finally, the program just needs to calculate the Gini Coefficient and print it out. The Gini Coefficient will be equal to `sumdiff/( 2.0*N*sum )`.

The Gini Coefficient calculated by your program will be a value between zero and one. If you run it with your simulation data for 10,000 transactions, like this:

```
./analyze simulate-10K.dat
```

you should see a Gini Coefficient of about 0.37. If you run it with the simulation data for 1 billion transactions, the value should be much higher, almost 1.0. Figure A.48 shows how the Gini Coefficient varies with the number of transactions. As you can see, it approaches a value of one for large numbers of transactions, meaning that only a few people end up with all of the wealth.

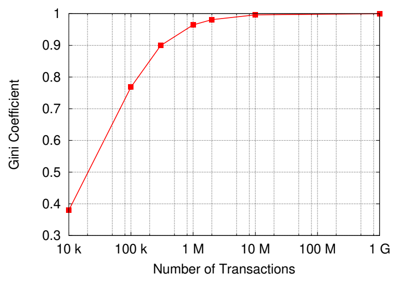


Figure A.48: Gini Coefficient calculated for various numbers of transactions using our yard sale simulation.

Conclusion

So what does this model of economics tell us about the real world? Although there is great inequality of wealth (for example, five billionaires now have more wealth than the poorest half of humanity combined), it seems unrealistic that one person would end up with all of it.

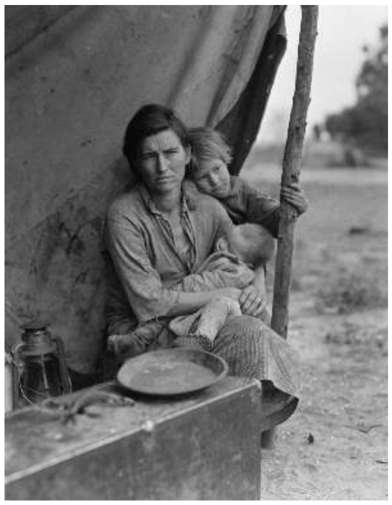
The yard sale model seems pretty simple. It just makes two assumptions: pairs of people exchange wealth, and poor people can't spend more money than they have. Why does it make a prediction that's so different from what we see in the world around us? Clearly there's some factor that we're leaving out of our model.

It might seem that everybody at the yard sale has an equal opportunity to gain wealth, and at first they do. Initially wealth is distributed evenly among all of them, with perfect symmetry. But this initial symmetry is spontaneously broken as soon as some people become a little poorer than others. Poorer people in the model are always at an economic disadvantage because poverty limits the size of the economic risks they can take. This creates a tendency for the rich to get richer and the poor to get poorer, causing the yard sale model to inevitably collapse into oligarchy.

Lou Hoover, First Lady of the United States, with her sons.

Why doesn't this happen in the real world? Mathematician Bruce Boghosian at Tufts University and his economist colleagues have shown that by transferring a small fraction of wealth from rich people to poor people after each transaction, the yard sale model's wealth distribution can be stabilized. In the real world, this corresponds to the wealth redistribution that's done by taxes and social programs.

With this one small change, Boghosian has found the modified yard sale model can match recent European and U.S. wealth distribution patterns to within 2%. By making two more tweaks, allowing people to go into debt and accounting for advantages that wealthier people have in business transactions, the model can match U.S. data over a span of several decades with an accuracy of a fraction of a percent.


Migrant Mother, Photo by Dorothea Lange.
Source: Wikimedia Commons

Boghosian also points to economies where social programs have broken down, like Armenia after the fall of the Soviet Union. In those cases, the economy really does devolve into oligarchy, with all of the wealth being held by a few people after an initial struggle for resources, just as our unmodified yard sale model would predict.

In a 2019 *Scientific American* article Boghosian said

> "We find it noteworthy that the best-fitting model for empirical wealth distribution discovered so far is one that would be completely unstable without redistribution rather than one based on a supposed equilibrium of market forces. In fact, these mathematical models demonstrate that far from wealth trickling down to the poor, the natural inclination of wealth is to flow upward, so that the 'natural' wealth distribution in a free-market economy is one of complete oligarchy. It is only redistribution that sets limits on inequality."

Further Reading

- *"The Mathematics of Inequality"*, https://now.tufts.edu/articles/mathematics-inequality.
- Bruce M. Boghosian, *"Is Inequality Inevitable?"* (originally published under the title *"The Inescapable Casino"*), Scientific American 321, 5, 70-77 (November 2019).
- Anirban Chakraborti, *"Distributions of money in model markets of economy"*, https://arxiv.org/abs/cond-mat/0205221.
- Branko Milanovic, *"Global Inequality and the Global Inequality Extraction Ratio"*, http://documents1.worldbank.org/curated/en/389721468330911675/pdf/WPS5044.pdf.
- World Bank, *"Poverty and Prosperity 2016 / Taking on Inequality"*, https://openknowledge.worldbank.org/bitstream/handle/10986/25078/9781464809583.pdf.

Project 6: Virus

Introduction

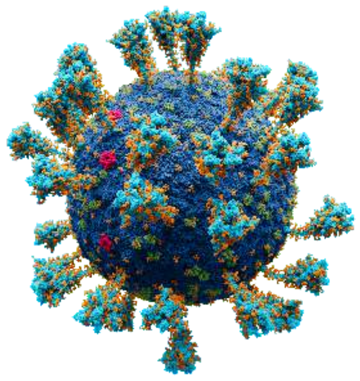
Imagine a time in the future when a new virus, deadlier than COVID-19, is ravaging the country. You're a health-care professional working on the island of Natucket, off the coast of Massachusetts. So far, no one on Natucket has contracted the virus, but you're a good computer programmer and your boss has asked you to simulate the potential effect of a virus outbreak on the island's residents.

Nantucket, Massachusetts.
Source: Wikimedia Commons

Nantucket has a full-time population of a little over 10,000. During the tourist season it normally swells to 50,000, but concerns over the virus and restrictions on travel have kept visitors away this year. Nonetheless, if the disease reaches Nantucket the number of patients could easily exceed the twenty beds available in the island's only hospital.

Fortunately, a very effective vaccine has been developed. One thing your boss has asked you to look into is how different percentages of vaccinated people would affect the impact of the virus.

You also know that measures like mask-wearing and social distancing can slow the spread of the virus by reducing the probabilty that it will be transmitted from one person to another. This is the second thing your boss has asked you to investigate: How does transmission probability influence the spread of the disease?



The SARS-CoV-2 virus, which causes COVID-19.
Source: Wikimedia Commons

It's late at night now, and you've started writing three programs that will help answer these questions. Go get some sleep and finish them tomorrow!

Program 1: Simulating The Progress of the Disease

Your first program is called **`simulate.cpp`** (see Program A.17 on page 572). It tracks the health of 10,000 people (the same as Nantucket's population) for 100 days. Each person's health status is represented by one of the numbers shown in Table A.1. The program uses a 10,000-element array of integers to keep track of the status of each person. The program assumes that 100 people (1%) are infected on the first day.

Status	Number	Symbol
Susceptible	0	`is_susceptible`
Recovered	-1	`is_recovered`
Vaccinated	-2	`is_vaccinated`
Dead	-3	`is_dead`
Infected	Any positive number: # of days infected	

Table A.1: Numerical status indicators for each person in the simulation. The last column shows symbols (defined in `virus.h` – see below) that we can use instead of numbers if we like. Notice that any positive number indicates the number of days that an infected person has been sick.

Each day, the program simulates random interactions between people. These interactions cause some susceptible people to get sick. Each day, sick people have some chance of getting well or dying. At the end of each day, the program counts how many people are in each of these states and writes that data into a file.

The program should accept three arguments on the command line: The fraction of people who are vaccinated (`vprob`), the probability of catching the disease from an infected person (`tprob`), and the name of the output file.

By running the program with various values of `vprob` and `tprob` you can investigate the effects of vaccination and masking or social distancing on the spread of the disease, as you boss asked you to do.

Vaccination and mask-wearing can slow the spread of some diseases.
Sources: Wikimedia Commons and Wikimedia Commons

How the Program Works

Your program actually has *two* 10,000-element arrays: one to hold each person's current health status, and the other to hold the status they'll have during the next day of the simulation. At the end of each day, the data from the "next day" array should be copied into the "current status" array.

The program should loop through all 100 days, and on each day it should loop through all 10,000 people, determining each person's new status. The new status will be determined by two functions you need to write.

To decide whether a susceptible person has been infected, you should write a function named **`catch_or_not`** that simulates a random number of encounters between the person and other people in the population. When an infected person is encountered, the function flips a virtual coin to see if the susceptible person catches the infection.

To decide whether an infected person gets well, dies, or stays sick, you should write another function named **`die_or_not`**. If the person survives more than about 14 days, this function determines that the person has recovered and is no longer sick or susceptible to the infection.

The program should assume that vaccinated people are perfectly protected, and have no chance of becoming infected[38].

[38] This is obviously not the case for a real vaccine, but it makes our program simpler. The Pfizer vaccine is 95% effective at preventing hospitalization due to Covid-19, so 100% effectiveness for a vaccine is not an unreasonable approximation for purposes of our simulation.

How to Write the Program

Your previous day's work is shown in Program A.17. You'll just need to finish the two functions at the top, and finish the middle of the main program. Follow the instructions below to complete the program. Note that the program also uses two header files that you wrote the previous day (Wow, you were really productive!), named `random.h` and `virus.h`. You'll find these in the "Header Files" section at the end of this project.

The program contains two arrays named `status` and `newstatus`. These are used to keep track of each person's current status and the new status they'll have on the next day of the simulation.

Start by writing the `catch_or_not` and `die_or_not` functions.

Writing `catch_or_not`:
This function will simulate a random number of interactions between a susceptible person and other people in the population and return an integer that's the new status for the person (one of the numbers in Table A.1). This function takes three arguments: `tprob`, the transmission probability; `npeople`, the number of people in the population; and `status`, the array of current status information. To write this function, follow these steps (and define any variables you need for them):

The island of Nantucket, once famous for whaling. Herman Melville said "Two thirds of this terraqueous globe are the Nantucketer's. For the sea is his; he owns it, as Emperors own empires".
Source: Wikimedia Commons

1. You're first going to need to generate a random number of people that this person is exposed to during the day. The header file `random.h` contains a function that will be handy for this, named `rand01`. It generates random numbers that are uniformly dis-

Program A.17: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "random.h"
#include "virus.h"

int catch_or_not ( double tprob, int npeople, int status[] ) {

  // Insert function here!

}

int die_or_not ( double dprob, int sick_days, int person, int npeople, int status[] ) {

  // Insert function here!

}

int main ( int argc, char *argv[] ) {
  const int npeople=1e+4;
  int status[npeople];
  int newstatus[npeople];
  int day,person;
  double vprob; // Vaccination probability.
  double tprob; // Transmission probability
  double dprob = 0.015; // Death probability per day
  int ndays = 100;
  int initial_infections = 0.01*npeople;
  int sick_days = 14;
  FILE *output;

  // Insert the rest of the program here!

  printf ( "Population: %d\n", npeople );
  printf ( "Vaccination Probability: %lf\n", vprob );
  printf ( "Transmission Probability: %lf\n", tprob );
  printf ( "Initial Infections: %d\n", initial_infections );
  printf ( "Simulation Period: %d days\n", ndays );
  printf ( "Number of Recovered: %d\n", nrecovered );
  printf ( "Number of Dead: %d\n", ndead );
  printf ( "Case Fatality Rate: %lf\n", (double)ndead/(nrecovered+ndead) );
}
```

tributed between zero and one (see Figure A.49). Use the following statement to generate a random number of people met that day:

```
nexposures = 20*rand01();
```

2. Then, you'll need a "`for`" statement that loops through `nexposures` people. Each time around the loop you'll need to do a couple of things:
   (a) pick a random other person from the list of people, like this:

   ```
   otherperson = rand01() * npeople
   ```

   (b) Use an "`if`" statement to see if the other person is infected (in other words, if `status[otherperson]` is greater than zero). If the other person is infected, check to see if the person we're tracking catches the disease.
   We do this by using the `rand01` function to give us a random number between zero and one, and then checking to see if that number is less than `tprob` (the "transmission probability"). If it is, the `catch_or_not` function should return a value of 1, meaning that the person is now on his first day of infection.

   If none of the encounters result in infection, `catch_or_not` should return a value of `is_susceptible`, meaning that the person isn't infected, but is still susceptible[39].

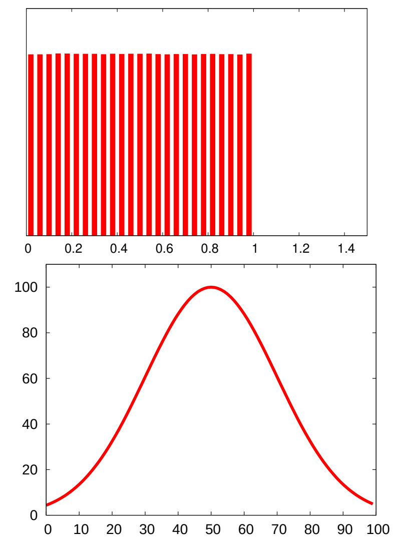


Figure A.49: Histograms of random numbers in a uniform distribution (top) from `rand01()` and a normal distribution (bottom) from `normal()`.

[39] See the values in Table A.1.

Writing `die_or_not`:
This function checks to see if an infected person stays sick, dies, or gets better. It takes five arguments: `dprob`, the probability of dying each day; `sick_days`, the typical number of days someone is sick[40]; `person`, this person's index in the `status` array; `npeople`, the total number of people; and the `status` array itself. Here's how to write this function:

[40] Notice that `simulate.cpp` already defines `dprob=0.015` and `sick_days=14`. You don't need to change these.

1. Start by using the `rand01` function to give a number between zero and one, and then checking to see if that number is less than `dprob`. If it is, that means that the person died, so the function should return a value of `is_dead`.
2. If the person doesn't die, we need to check to see if she recovers or remains sick. To do this, check to see if the number of days the person has been sick is greater than `sick_days`, plus or minus a small random amount (everyone isn't sick for exactly the same amount of time). Do that like this:

```
    if ( status[person] > sick_days + 3.0*normal() ) {
```

   This uses the `normal` function, which generates random numbers that tend to be around zero, but are sometimes bigger or smaller[41]. (Remember that `status[person]` is the number of days an in-

[41] See the bottom graph in Figure A.49.

fected person has been sick.) If this condition is true, the function should return `is_recovered`. Otherwise it should return `status[person]+1`, indicating that the person will stay infected for another day.

Writing `main`:
To complete the program, you'll need to add some lines to `main` to do the following:

1. Check to make sure the user has supplied enough command-line arguments[42]. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[43].

[42] Syntax should be:
`./simulate vprob tprob file`

[43] See Section 9.16 of Chapter 9 for an example of how to do this.

2. Convert the command-line arguments into the variables `vprob` and `tprob` by using the **`atof`** function. The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

```
output = fopen( argv[3], "w" );
```

3. Next, calculate the number of vaccinated people, using `vprob` like this:

```
  nvaccinated = vprob*npeople;
```

4. Now your program is ready to set the initial status of each person. The `virus.h` file contains a function named `initialize_status` to help you do that[44]. Use it like this:

```
  initialize_status( npeople, initial_infections, nvaccinated, status );
```

[44] Notice that `simulate.cpp` already defines `initial_infections` to be `0.01*npeople`. You don't need to change this.

5. Your program will need a pair of nested "`for`" loops: An outer loop that goes through all the days, and an inner loop that goes through all of the individuals in the population and, for each one, checks to see whether that person gets sick, gets well, dies, or stays the same.

   The outer loop might start like this:

```
for ( day=0; day<ndays; day++ ) {
```

   and the inner loop might start like this:

```
for ( person=0; person<npeople; person++ ) {
```

6. Inside the inner loop we'll determine the person's new status for

the next day (`newstatus[person]`). We'll need an "`if`" statement that has three branches that do different things depending on the person's current status:

- If the person is susceptible (in other words, if `status[person] == is_susceptible`) then:

  ```
  newstatus[person] = catch_or_not( tprob, npeople, status );
  ```
- If the person is infected (that is, if `status[person] > 0`) then:

  ```
  newstatus[person] = die_or_not( dprob, sick_days, person, npeople, status );
  ```
- Otherwise, we'll assume the person just stays the same the next day, and set `newstatus[person] = status[person]`.

7. At the end of each day, the program should update the status of all the people by copying all of the elements of `newstatus` to `status`. The `virus.h` header file includes a function named `update_status` that will do that for us. Add the following line to your program to use it:

   ```
   update_status( npeople, status, newstatus );
   ```

8. The `update_status` function will also count how many people are in each state and put those numbers into the variables `nsusceptible`, `nrecovered`, `nvaccinated`, `ninfected`, and `ndead`. Use this `fprintf` line to write the those values and the current day number into the output file at the end of each day:

   ```
   fprintf ( output, "%d %d %d %d %d %d\n",
             day, nsusceptible, nrecovered, nvaccinated, ninfected, ndead );
   ```

After you've completed your program, compile it and run it like this:

```
./simulate 0.0 0.015 simulate-0.0-0.015.dat
```

That will run the simulation with `vprob` equal to zero (nobody vaccinated) and `tprob` equal to 0.015 (1.5% chance of catching the disease from an infected person). The program should print something like Figure A.50 on the screen when it's done.

```
Population: 10000
Vaccination Probability: 0.000000
Transmission Probability: 0.015000
Initial Infections: 100
Simulation Period: 100 days
Number of Recovered: 5271
Number of Dead: 1084
Case Fatality Rate: 0.170574
```

Figure A.50: Typical output from `simulate.cpp` with `vprob=0.0` and `tprob=0.015`.

As you can see, under these conditions we might expect over a thousand people to die during this 100-day period. The program also prints a "Case Fatality Rate" (CFR) that tells us the likelihood of dying if you catch the disease. The CFR for this disease is about 17%. For comparison, the reported[45] global CFR for COVID-19 is about 2% and for seasonal flu in the US it's about 0.2%.

[45] See ourworldindata.org for some information about the difficulty of interpreting real-world CFR numbers.

You could graph the data in the output file using this *gnuplot* command:

```
plot "simulate-0.0-0.015.dat" using 1:2 with lines lw 5 title "Susceptible", \
 "" using 1:3 with lines lw 5 title "Recovered", \
 "" using 1:4 with lines lw 5 title "Vaccinated", \
 "" using 1:5 with lines lw 5 title "Infected",\
 "" using 1:6 with lines lw 5 title "Dead"
```

If you did, you should see a graph like the following:

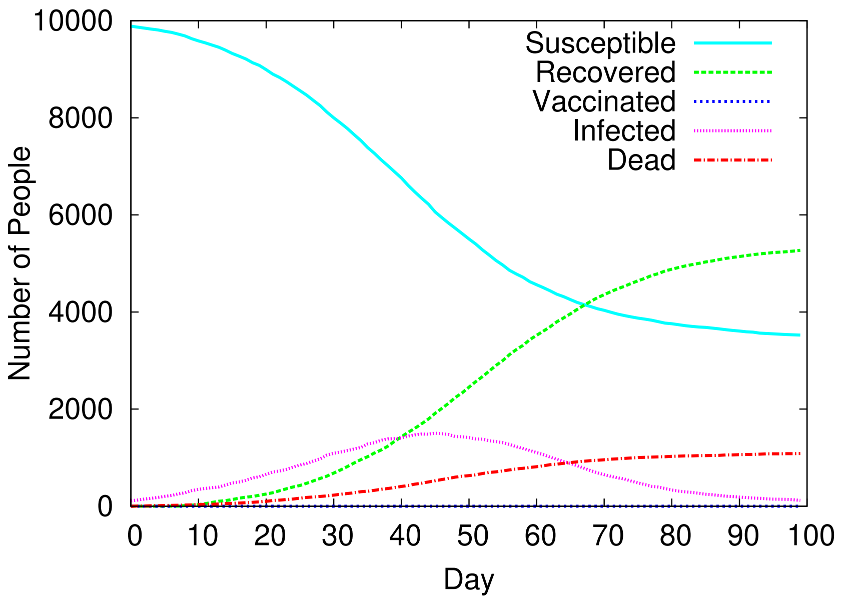


Figure A.51: Results of a simulation with `vprob=0` and `tprob=0.015`.

vprob	tprob	Deaths
0.0	0.015	1205
0.1	0.015	937
0.6	0.015	38
0.0	0.007	71
0.1	0.007	44
0.6	0.007	21

Figure A.52: Some typical values for number of deaths, given various values for `vprob` and `tprob`. (Your program's results will vary slightly from these because it uses random numbers.)

Try running the program with `vprob` set to 0.6 (60% of people vaccinated). You should see that the number of deaths is a lot smaller. Then try running it with `vprob=0.6` and `tprob=0.007` (reducing `tprob` by about half from the previous value). This simulates the effect of masking and social distancing, which reduce the transmission probability[46], and should further reduce the number of deaths. Some typical results are shown in Figure A.52.

[46] See "Why Masks Work BETTER Than You'd Think": https://www.youtube.com/watch?v=Y47t9qLc9I4.

Figures A.53 and A.54 on the next page show how you might expect the number of deaths to change as you change `vprob` and `tprob`. The graphs show that vaccinating even half the people would have a dramatic effect, and similar results could be obtained by cutting the transmission probability in half.

Program 2: How Much Do Results Vary?

When we run our simulation program it gives us numbers that tell us how many people were infected, recovered, died, and so forth, but the program works by simulating random interactions between people. It's possible that the program could sometimes just "roll the dice" in a really unlikely way, and give us results that are unrealistic.

How much random variation is there in the results our program produces? One way to get a handle on this would be by running the program many times with the same set of parameters, and then looking at how much the results vary.

We don't have to do that by hand, though. As we've seen, computers are very good at doing the same thing over and over again very quickly. We can just get the computer to run our simulation many times for us. That's what you'll do for your second program, which will be called **`analyze.cpp`**.

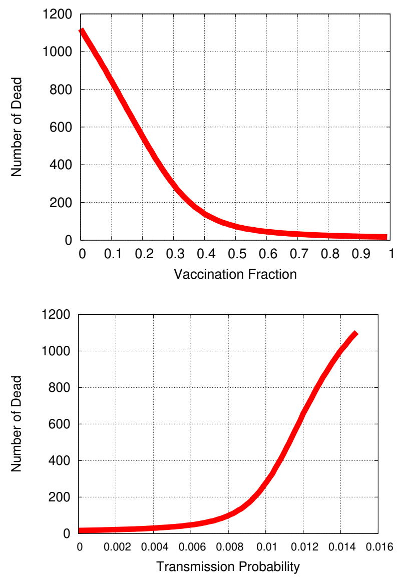


Figure A.53: The top graph shows the number of deaths after 100 days for various values of `vprob` when `tprob` is 1.5%. The bottom graph shows number of deaths for various values of `tprob` when `vprob` is 0%.

How the Program Works

This new program will start out as a copy of `simulate.cpp`. Instead of just simulating 100 days once, though, the new program will do the same simulation 1,000 times. At the end, it will tell the user the average number of deaths after 100 days, and the standard deviation of this number. The program will also write the results of each trial into an output file for later analysis. (We'll use this file with the third program you write.)

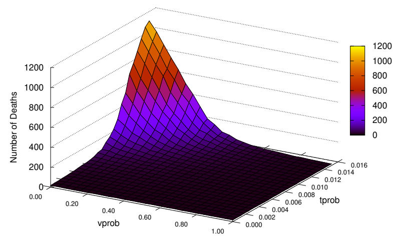


Figure A.54: Effect of changing `vprob` and `tprob` on the number of deaths after 100 days.

How to Write the Program

Follow these steps to write `analyze.cpp`:

1. Start by copying `simulate.cpp`:

   ```
   cp simulate.cpp analyze.cpp
   ```

2. Then edit `analyze.cpp` and add a few new variables to `main`:

   ```
   double sum=0;
   double sum2=0;
   int trial;
   int ntrials=1000;
   ```

   We'll use these for looping through 1,000 trials and calculating the average and standard deviation of the number of deaths.

3. Next, you'll need to add a "`for`" loop around most of the rest of `main`. The "`for`" loop should start before the `initialize_status` line (so that the status of all the people gets reset at the beginning of each trial), and it should begin like this:

   ```
   for ( trial=0; trial<ntrials; trial++ ) {
   ```
4. This program will take a while to run, so immediately under the beginning of the new "`for`" loop you should include some lines to print progress messages. Do this by printing a message like "Processing trial 20" whenever `trial` is a multiple of ten[47].

[47] See Section 4.4 of Chapter 4 for information about how to do this.

5. Remove the existing `fprintf` statement, since we don't want this program to write results at the end of every day. (We'll add a new `fprintf` statement in an other place – see below – to write the results at the end of each trial.)

6. At the end of each trial, do a couple of things:

   - Add `ndead` to `sum`, and add `ndead*ndead` to `sum2`. We'll use these sums later to calculate the mean and standard deviation after we've done all the trials.
   - Write the results from this trial into the output file. Use a line like this:

   ```
   fprintf ( output, "%d %d %d %d %d %d\n",
             trial, nsusceptible, nrecovered, nvaccinated, ninfected, ndead );
   ```

7. At the very end of the program, instead of printing the "Number of Recovered", "Number of dead", and "Case Fatality Rate" instead print:

   (a) The number of trials
   (b) The average number of deaths at the end of each trial
   (c) The standard deviation of this number

Compile `analyze.cpp` and try running it like this:

```
./analyze 0.0 0.015 analyze-0.0-0.015.dat
```

```
Population: 10000
Vaccination Probability: 0.000000
Transmission Probability: 0.015000
Initial Infections: 100
Simulation Period: 100 days
Number of Trials: 1000
Average Number of Dead: 1117.258000
Standard Deviation: 34.126149
```

Figure A.55: Typical output from `analyze.cpp` when `vprob=0.0` and `tprob=0.015`.

It should tell you something similar to Figure A.55, showing that, if nobody is vaccinated and people have a 1.5% chance of catching the virus from an infected person, the expected number of deaths after 100 days is about 1117 ±34, so between 1083 and 1151.

If you set `vprob=0.6` and `tprob=0.007` you should see that the number of deaths after 100 days drops to about 20 ±5.

Lambs, Nantucket (1874), by Eastman Johnson, National Gallery of Art.
Source: Wikimedia Commons

Program 3: Visualizing the Variation

While `analyze.cpp` is running it writes the results of each trial into an output file. Your next job is to write a new program named **`visualize.cpp`** that will read that file and produce a histogram. Your program will be similar to program 7.1 in Chapter 7. The data it will histogram is the number of deaths. It should start out like Program A.19 below. Notice that it defines a 50-element array, `bin`, to hold the histogram data.

Program A.19: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {
  const int nbins=50;
  int bin[nbins];
  double dmin, dmax;
  double binsize;
  int trial;
  int nsusceptible, nrecovered, nvaccinated, ninfected, ndead;
  int binno;
  int overunderflow=0;
  int i;
  FILE *input;
  FILE *output;

  // Insert Program Here.
}
```

How the Program Works

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize dmin dmax inputfile outputfile
```

where `dmin` and `dmax` are the minimum and maximum number of deaths that can be recorded in the histogram; `inputfile` is the name of a file that was produced by `analyze.cpp`; and `outputfile` is the name of a file into which the new program will write the histogram data. When plotted, the data should look like Figure A.56 on page 580.

How to Write the Program

To make the histogram, the program should proceed as follows:

1. Check the number of command-line arguments, and use `atof` to

set the values of `dmin` and `dmax`. The input and output files can be opened like this[48]:

```
input  = fopen(argv[3],"r");
output = fopen(argv[4],"w");
```

[48] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

2. Then, determine the `binsize`, like this:

```
binsize = (dmax-dmin)/nbins;
```

3. Next, use a `while` loop to read data from the input file[49]. Each line of the file will contain six integer values: `trial`, `nsusceptible`, `nrecovered`, `nvaccinated`, `ninfected`, and `ndead`.

[49] See Chapter 5 for information about reading data from files.

4. Determine which bin this `ndead` value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the range of our histogram is `dmin` to `dmax`, the bin number will be:

```
binno = (ndead-dmin)/binsize;
```

5. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the `ndead` value represented by that bin[50], and the value of `bin[i]`. The `ndead` value can be calculated from the bin number, like this:

```
ndead = dmin + binsize*(0.5+i);
```

where `i` is the bin number.

[50] Note that this is different from Program 7.1, where we just printed the bin number as the first column in the output file.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

[51] Note that I've chosen values of `dmin` and `dmax` that are separated by a multiple of 50 (the number of bins). Doing this will make your graphs look better.

Compile your program and try running it like this, using the output file you created above[51]:

```
./visualize 1000 1250 analyze-0.0-0.015.dat visualize-0.0-0.015.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize-0.0-0.015.dat" with impulses lw 5
```

The result should look like Figure A.56, which seems to be in agreement with our previous program's prediction that the number of deaths with `vprob=0` and `tprob=0.015` would typically lie somewhere between 1083 and 1151.

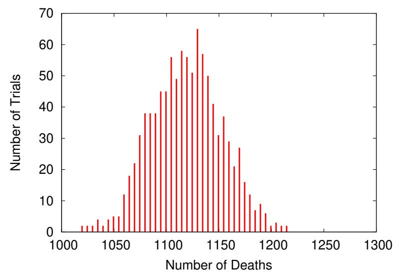


Figure A.56: Histogram of number of deaths after 100 days, from 1,000 trials with `vprob=0` and `tprob=0.015`.

Conclusion

Congratulations! The programs you've written will help you and your boss know what to expect if this disease reaches Nantucket. It appears that if you can get 60% of the people vaccinated and use mask-wearing and social distancing to reduce transmission by half, the number of deaths can be reduced from over a thousand to only twenty or so. This is still tragic, but the lives of a thousand people could be saved.

An inflatable sea serpent on a Nantucket beach (1937).
Source: Wikimedia Commons

The programs you've written are very simple compared to the sophisticated models that organizations like the CDC use for forecasting the spread of a disease, but they still give you some insight in to how a pandemic works and a couple of the factors that influence its progress. You might think about how you could improve on the programs you've written. Here are some things to consider:

The first surfboard on Nantucket (1932).
Source: Wikimedia Commons

- Rather than assuming a perfect vaccine, what if the vaccine only reduced your probability of getting infected? What if it reduced your probability of dying?
- What if the simulation allowed you to put a limit on the size of gatherings by changing how `nexposures` is calculated?
- What if infected people were quarantined after they showed symptoms?
- What if the simulation increased the probability of death after the number of sick people exceeded the number of available hospital beds?
- What if conditions change during the course of the outbreak? `tprob` might be higher in cold weather, for example, or people might be getting vaccinated while the disease is spreading.
- What about the effect of social networks? People tend to interact with a network of friends, relatives, and co-workers rather than people chosen completely at random.

Because of the increasing global population and the growth of global travel and commerce, forecasting disease outbreaks is more important than ever before. Sara Del Valle, a mathematical epidemiologist at Los Alamos National Laboratory, argues that we need to make disease forecasting as accurate and reliable as weather forecasting[52]. As a step in this direction, in 2021 the CDC announced the formation of a new "Center for Forecasting and Outbreak Analytics"[53]. Efforts like this should provide fertile new ground for young programmers who want to make the world a better place.

[52] Sara Del Valle, "We need to forecast epidemics like we forecast the weather".
[53] CDC press release

Header Files

You'll need the following two header files in order to write your programs.

The first file, `random.h`, contains two functions for generating random numbers. See Figure A.49 on page 573 for an illustration of the output from the two functions.

Program A.20: random.h

```
// Random number between zero and one
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}
// Normal distribution:
double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}
```

The second file, `virus.h`, contains some global variable definitions and two functions: `update_status` and `initialize_status`.

The `initialize_status` function takes care of setting up the initial health status of all 10,000 people. Depending on the values of `initial_infections` and `nvaccinated`, some number of people are set to the infected or vaccinated states. Everybody else is set to the susceptible state.

The `update_status` function takes care of copying information from `newstatus` to `status` and also counts how many people have each health status.

Program A.21: virus.h

```
// Global variables:

// Counters:
int nsusceptible, ninfected, nrecovered, nvaccinated, ndead;

// Status indicators:
// Any positive value indicates user has been infected for n days.
// Negative values indicate:
const int is_susceptible = 0;
const int is_recovered = -1;
const int is_vaccinated = -2;
const int is_dead = -3;

// Update status of each person and count them
void update_status( int npeople, int status[], int newstatus[] ) {
  int i;

  // Reset counters in preparation for counting:
  nsusceptible = 0;
  ninfected = 0;
  nrecovered = 0;
  ndead = 0;

  for ( i=0; i<npeople; i++ ) {
    status[i] = newstatus[i];
    if ( status[i] == is_susceptible ) {
      nsusceptible++;
    } else if ( status[i] == is_recovered ) {
      nrecovered++;
    } else if ( status[i] == is_dead ) {
      ndead++;
    } else if ( status[i] > 0 ) {
      ninfected++;
    }
  }
}
void initialize_status ( int npeople,
                         int initial_infections, int nvaccinated, int status[] ) {
  int i;
  int vmax;

  // Initial infections:
  for ( i=0; i<initial_infections; i++ ) {
    status[i] = 1;
  }

  // Vaccinations:
  // Don't exceed total number of people!
  vmax = initial_infections+nvaccinated;
  if ( vmax > npeople ) {
    vmax = npeople;
  }
  for ( i=initial_infections; i<vmax; i++ ) {
    status[i] = is_vaccinated;
  }

  // Everybody else is susceptible:
  for ( i=vmax; i<npeople; i++ ) {
    status[i] = is_susceptible;
  }

}
```

Project 7: Crowd Control

Introduction

Imagine you're in charge of security for a stadium. The stadium hosts large sports events and concerts. It holds 10,000 people. Your job is to move these people safely into the stadium during the 90 minutes before an event begins.

To enter the stadium, attendees need to pass through a security screening process that includes bag checks and metal detectors. To help get people into the stadium quickly, there are 20 different security check stations distributed around the stadium's perimeter.

In this project we'll be writing three programs that simulate the process of passing spectators into such a stadium, and analyze and visualize the results of our simulation. With our programs we'll try to answer questions that would come up in the real world:

- How long does it take to get everybody into the stadium?
- How long do people wait in line?
- How long do the lines get?

Our simulation will have two adjustable parameters: The rate at which people arrive, and the average time needed to do a security scan on a person. You'll be writing three programs, named `simulate.cpp`, `analyze.cpp`, and `visualize.cpp`.

The Roman Colosseum (top photo) could hold 50,000 to 80,000 spectators. The University of Virginia's Scott Stadium (middle photo) accommodates 60,000. Currently, the largest stadium in the world is Narendra Modi Stadium (bottom photo) in India, which holds 132,000 people.
Source: Wikimedia Commons (top), University of Virginia (middle), Gujarat Cricket Association (bottom)

Program 1: Simulating the Lines

Your first program will be called `simulate.cpp`. It will simulate the process of moving 10,000 people into a stadium, using 20 entrance lines. To answer the questions listed in the introduction we'll need to keep track of a few things for each line:

- When did a person join the line?
- Who's next in line?
- How many people are currently in line?

In programming, we often refer to a line like this as a "queue", and I'll use those words interchangeably in the following descriptions.

Our program will keep track of the necessary information by using arrays. To start with, imagine that we only had one queue leading to one scanning station. We could have an array that stored the time at which each person entered the line, as shown in Figure A.57.

In the top diagram we see that there are six people in line. The first person (element 0 of the array) entered the line 49 seconds after the gates opened. The other people entered at successively later times. A variable named `qstart` keeps track of who is at the front of the line (in this case, person 0). The variable `qend` keeps track of the next free spot in line (element number 6 of the array, in this case). The number of people in line at this point is just `qend - qstart`.

When we want to add a person to the line, we put the person's arrival time into element `qend`, and then add one to the value of `qend`, as shown in the middle diagram. There are now 7 people in line, which is equal to the new value of `qend - qstart`.

When the person at the front of the line gets passed through the scanning station, we add 1 to `qstart`, as shown in the bottom diagram. Now `qend - qstart` is 6 again, the number of people still in line.

If we have more than one line, we'll need to keep track of `qstart` and `qend` for each of them. We can do this by making `qstart` and `qend` arrays. If we have 20 lines, numbered 0 through 20, we could have `qstart[0]` be the value of `qstart` for line number 0, `qstart[1]` for line number 1, and so forth.

For keeping track of everybody's arrival time, let's use a 2-dimensional array named `q`. In our program, we'll define `q`, `qstart`, and `qend` like this[54]:

```
int q[ nqueue ][ npeople ];
int qstart[ nqueue ] = {0};
int qend[ nqueue ] = {0};
```

where `nqueue` is the number of queues ("lines"), and `npeople` is the number of people in our simulation. `npeople` is the maximum possible number of people who can be standing in line, even if we

In 2005, UVa's Scott Stadium hosted a Rolling Stones concert.
Source: Wikimedia Commons

Bono and Adam Clayton during a U2 concert at Scott Stadium on October 1, 2009.
Source: Wikimedia Commons

In 2017, Dave Matthews and many other musicians played a "Concert for Charlottesville" in Scott Stadium, in response to the violent events in August of that year.
Source: Wikimedia Commons

[54] Notice that we set all the values of `qstart` and `qend` to zero initially.

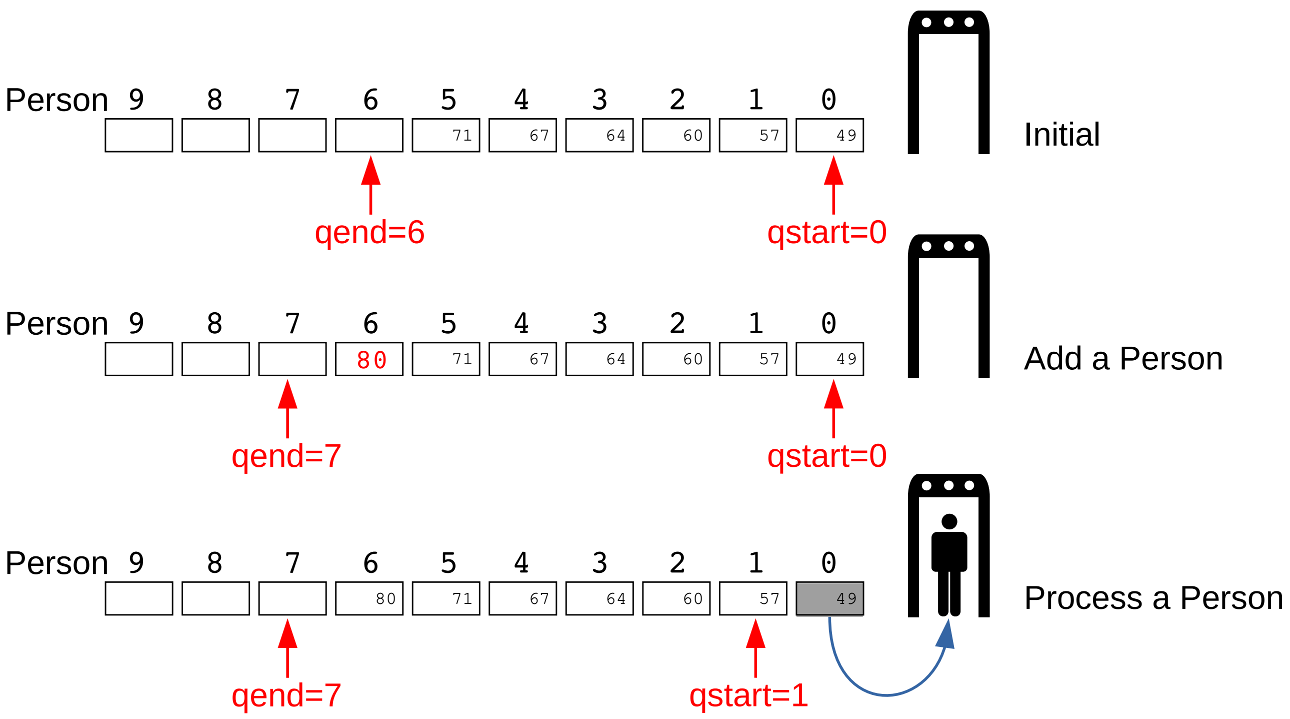


Figure A.57: This diagram shows how our program will add and remove people from a queue.

only had one line. Figure A.58 shows what the situation will now look like with multiple queues[55]. Here we show a situation in which the first three people in queue 4 have been processed through the scanning station. The number of people left in queue 4 is `qend[4]-qstart[4]` = 6. If we wanted to know when person number 5 entered this queue, we could look at `q[4][5]` and see that he arrived at 569 seconds after the gates opened.

[55] For clarity, this only shows five of the twenty queues.

How to Write the Program

Writing the simulation program will take a little work, but fortunately you have an assistant who's already done some of it for you. Program A.22 shows what he's written so far. You'll just need to fill in the missing pieces (marked with "ADD").

Notice that the program has a variable named `t` that keeps track of the elapsed time. It starts out with `t = 0`. At the bottom of the program's `do-while` loop, 1 second is added to `t` every time the program goes around the loop. Each trip around the loop represents the passage of 1 second of time. During that time, newly-arrived people can join the queues, and people at the front of the queues can be scanned and passed into the stadium.

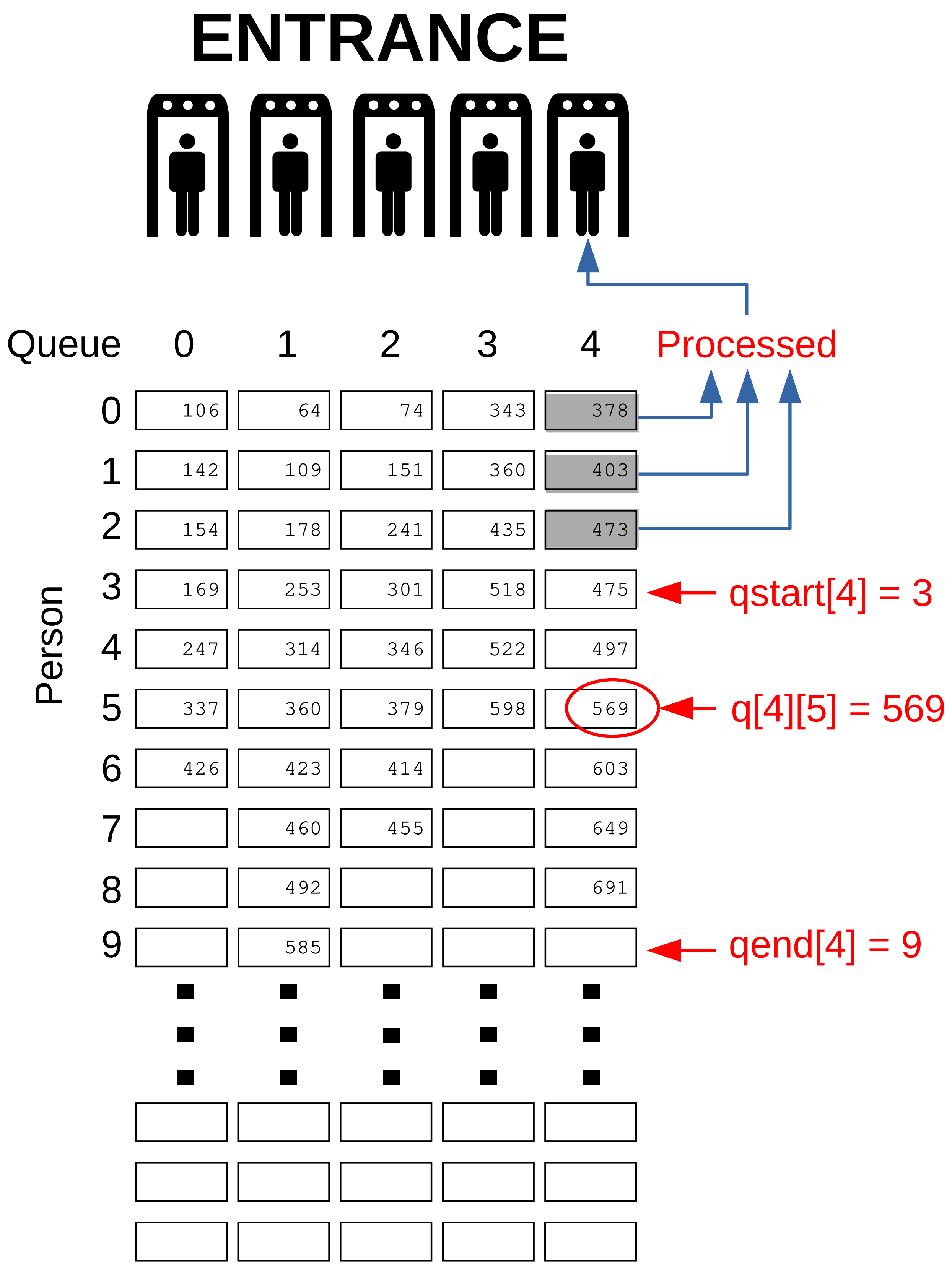


Figure A.58: A diagram illustrating our program's strategy for simulating the queues.

Program A.22: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char *argv[] ) {
  const int npeople = 10000;
  const int nqueue = 20;
  int q[nqueue][npeople];
  int qstart[nqueue] = {0};
  int qend[nqueue] = {0};
  int currentq = 0;
  int i;
  int t = 0;
  int nscanned = 0;
  int nqueued = 0;
  int max_arrivals_per_second;
  int arrivals;
  double scan_delay;
  double flip;
  FILE *output;

  // ADD: Check number of command-line arguments.
  // ADD: Set max_arrivals_per_second and scan_delay.
  // ADD: Open output file

  srand(time(NULL));

  do {

    // ADD: Find number of new arrivals:
    // ADD: Add new arrivals to the back of the lines:

    // Process people at the front of the lines:
    for ( i=0; i<nqueue; i++ ) {
      if ( qend[i] - qstart[i] > 0 ) { // Is there anyone in the queue?
        flip = rand()/(1.0+RAND_MAX);
        if ( flip < 1.0/scan_delay ) { // Is the scanner ready for a new person?

          // ADD: Write data to output file.
          // ADD: Update qstart.
          // ADD: Count number of people scanned so far.

        }
      }
    }

    t++;
  } while ( nscanned != npeople );

  printf( "Elapsed time %d seconds (%lf hours)\n", t, t/3600.0);

  fclose( output );
}
```

The program uses the variable nqueued to keep track of how many people have arrived so far, and it uses nscanned to keep track of how many people have been passed into the stadium. The variable npeople is set to 10,000, the total number of attendees.

The program should be written so that you can run it with command-line arguments like this:

```
./simulate 5 10 simulate-5-10.dat
```

where the first number sets the value of the variable max_arrivals_per_second. This will control the rate at which people arrive. The second number sets scan_delay[56]. This controls the average time (in seconds) required to do a security scan on a person. The last thing on the command line is the name of a file into which the program will write its output.

[56] Notice that max_arrivals_per_second and scan_delay are defined as an int and a double, respectively, near the top of Program A.22.

Here's a list of the things you'll need to add to the program:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print a friendly usage message and then stop without trying to do anything else[57].

[57] See Section 9.16 of Chapter 9 for an example of how to do this.

2. Convert the command-line arguments into the variables max_arrivals_per_second and scan_delay by using the atoi and atof functions[58]. The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

   ```
   output = fopen( argv[3], "w" );
   ```

[58] Notice that max_arrivals_per_second is an int and scan_delay is a double.

3. Inside the do-while loop you'll first need to determine how many new arrivals there are during this second. We'll add these people to the total number of people who have arrived (nqueued) and put their arrival time into slots in the queues.

   When you run the program you'll specify max_arrivals_per_second on the command line. We'll assume that the number of new arrivals during each trip around the loop will be some random number less than this. We can start by saying:

   ```
   arrivals = max_arrivals_per_second*(rand()/(1.0+RAND_MAX));
   ```

   but we need to make sure that nqueued never gets bigger than npeople (the total number of attendees). To prevent that, we need to add something like this next:

```
if ( nqueued + arrivals > npeople ) {
  arrivals = npeople - nqueued;
}
```

Now `arrivals` will be zero when `nqueued` gets as big as `npeople`.

4. Now that we know how many people arrived, we need to add these people, one by one, to the queues. To do that we'll need a "`for`" loop that starts like "`for (i=0;i<arrivals;i++)`". Inside the loop we'll need to do a few things.

   We'll need to decide which queue we'll put this person into. There are several different reasonable ways to do this, but let's use something called "round-robin" queueing. In that way of doing it, we add the first person to queue 0, the second person to queue 1, and so on until we get to the last queue, and then we start back at queue 0 again. To keep track of which queue is the current one, we'll use the variable `currentq` which initially has the value 0.

   We'll also need to know where to put this person into the current queue. The first open slot of this queue is given by `qend[ currentq ]`. With all that in mind, here's what we need to do for each person who's just arrived:

   (a) Add each of the new arrivals to the current queue by putting the current time (`t`) into `q[ currentq ][ qend[currentq] ]`.
   (b) Then we need to add 1 to `qend[currentq]`, so that it points to the next unfilled slot (See Figure A.57.)
   (c) Next we need to figure out which queue we'll be putting the next person into. We can do that like this:

   ```
   currentq = (currentq+1)%nqueue;
   ```

   What we're doing here is adding 1 to `currentq`, but making sure that we go back to zero after the last queue. The `%` is the modulo operator, which gives the remainder after division. It makes our calculation work like the numbers on a clock face: When we've gone through all of the numbers, we start back at zero.
   (d) Finally, we need to add 1 to `nqueued` to indicate that one more person has successfully arrived.

5. Now that we've added any new arrivals, we need to take care of the people at the front of the lines. The number of queues

A happy crowd at a football game in Scott Stadium.

A game in progress (UVa versus Wake Forest, 2007).
Source: Wikimedia Commons

Scott Stadium covered in snow.

is given by `nqueue` (this is set to 20 at the top of the program). We need to process the people standing at the front of each of these queues, so we'll need a "`for`" loop that starts out like this: "`for (i=0;i<nqueue;i++)`". Inside this loop we'll need to do the following for each queue[59]:

[59] Steps *a* through *c* have already been written by your assistant. You'll need to add the things in step *d*.

(a) First, let's see if anyone is waiting in this line. (Maybe the line is empty right now!) The number of people waiting in one of the queues is `qend[i] - qstart[i]`. If this is greater than zero, then someone is waiting to get into the stadium, and we should go on to the next steps.

(b) Next, we'll need to see if the gate attendant has finished scanning this person. We'll decide by "flipping a coin". We do this by generating a random number between zero and one[60]. The program should put this number into the variable named "`flip`", for use in the next step.

[60] See Section 2.6 of Chapter 2 for an example of this.

(c) The average number of seconds it takes for a security scan is given by `scan_delay`, so we're going to assume that the probability of finishing a scan during one of our 1-second-long trips through the `do-while` loop is `1.0/scan_delay`. To decide whether this person is ready to pass into the stadium, check to see if `flip` is less than `1.0/scan_delay`. If it is, then proceed to the next step.

(d) At this point, we know that someone is waiting in line and that the gate attendant is ready to pass them into the stadium. We just have three small things to do in order to finish processing this person:

   i. Print the information about this person into the output file. For each person processed we want to print the following 4 things[61]:

   [61] See Program 5.3 in Chapter 5 for an example of writing numbers into a file.

   - The queue number, `i`,
   - The current time, `t`,
   - The amount of time the person has been waiting, `t-q[i][qstart[i]]`,
   - The number of people currently in this line, `qend[i]-qstart[i]`

   Each of those things is an integer. They should be printed into the output file in the order above, with spaces between them and with a "`\n`" at the end of the line.

   ii. We need to add 1 to `qstart[i]` (See the bottom of Figure A.57).

   iii. The variable `nscanned` will keep track of how many people have passed into the stadium. It's initially set to zero at the top of the program. Now we need to add 1 to it, to indicate that we've finished processing another person. Our program's `do-while` loop will stop when `nscanned` is equal to `npeople`.

Running the Program

After you've finished writing and compiling the program try running it a few times, like this:

```
./simulate 5 5 simulate-5-5.dat
./simulate 5 10 simulate-5-10.dat
./simulate 5 30 simulate-5-30.dat
```

Arrivals/sec	Delay (sec)	Time (hours)
5	5	1.36
5	10	1.49
5	30	4.43
30	5	0.76
30	10	1.51
30	30	4.54

Figure A.59: This table shows the total amount of time it would take to get all 10,000 people into the stadium, given a few different values of `max_arrivals_per_second` and `scan_delay`.

Those commands tell the program to simulate 5 people/sec arriving, with a scan delay of 5, 10, and 30 seconds. Notice the total elapsed time that the program prints in each case. Figure A.59 shows the time required to move everybody into the stadium for a few different choices of `max_arrivals_per_second` and `scan_delay`.

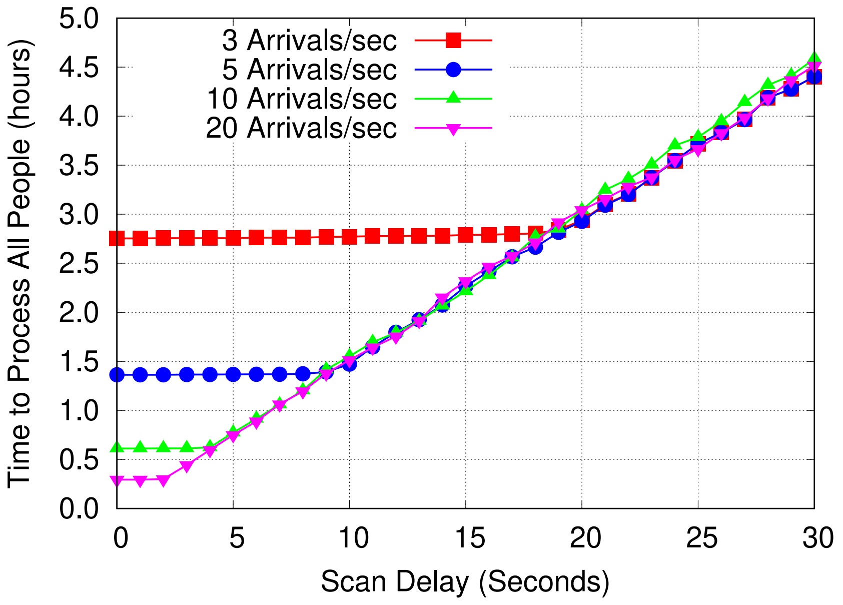


Figure A.60: The graph at left shows the time required to get all the people into the stadium as a function of `scan_delay`, for several different values of `max_arrivals_per_second`.

If you did this for many different values of these two parameters you could make a graph like Figure A.60. The vertical axis shows how long it takes to get all the people into the stadium, and the horizontal axis shows `scan_delay`. Different kinds of lines show different values of `max_arrivals_per_second`.

Clearly, changing the `scan_delay` can make a big difference in the way the queues behave. For any given value of `max_arrivals_per_second` there seems to be a critical value of `scan_delay`, beyond which it starts taking longer and longer to get everybody into the stadium. For 5 arrivals/second, this critical delay is about 10 seconds.

We could use graphs like Figure A.60 or those shown in Figure A.61 to examine our data, but it's useful to have a quantitative summary of our results. That's what we'll do in our second program.

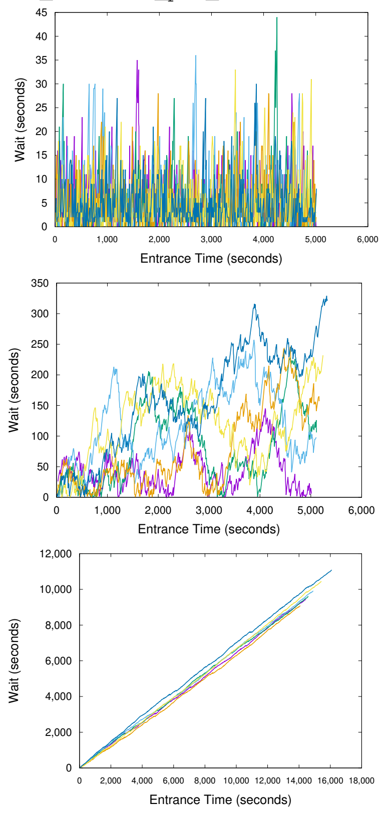


Figure A.61: These figures show how long people waited in line, versus when they finally entered the stadium. Different colors represent different queues (six queues are shown here). In all figures, the `max_arrivals_per_second` was set to 5. For the top figure, the `scan_delay` was 5 seconds. For the middle figure, it was 10 seconds. For the bottom figure, it was 30 seconds.

Program 2: Analyzing the Simulation Data

Your second program will be named `analyze.cpp`. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first program, `analyze.cpp` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze filename
```

where `filename` is the name of a data file produced by your `simulate.cpp` program.

Once again, your assistant has already started writing the program for you. You can see his work in Program A.23.

Program A.23: analyze.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char *argv[] ) {
  FILE *input;
  int queue, t, wait, length;
  int maxwait, maxlength, maxtime;
  double waitsum=0, waitsum2=0, mean, stddev;
  int npeople=0;
  int initialized=0;

  // ADD: Write the rest of the program here.

  printf ("Analyzed %d people.\n", npeople );
  printf ("Maximum wait was %d seconds (%lf minutes)\n", maxwait, maxwait/60.0 );
  printf ("Average wait was %lf seconds (%lf minutes)\n", mean, mean/60.0 );
  printf ("Standard deviation was %lf seconds (%lf minutes)\n", stddev, stddev/60.0 );
  printf ("Maximum queue length was %d people\n", maxlength);
  printf ("Time to fill stadium was %d seconds (%lf hours)\n",
         maxtime, maxtime/60.0/60.0 );
}
```

How to Write the Program

Here's a list of things you'll need to add to the program:

1. As you did in the first program, check to make sure the user has supplied the correct number of command-line arguments, and exit with a friendly message if the number isn't right.

2. The only command-line argument for this program is the file name, which doesn't need to be converted in any way. It can just be used directly like this:

```
input = fopen( argv[1], "r" );
```

3. Now you'll need a "`while`" loop to read the file you've opened[62]. Our program will read four values from each line of the data file:

   - **`queue`**, The queue from which this person entered the stadium.
   - **`t`**, The time at which the person entered the stadium.
   - **`wait`**, How long the person waited in line.
   - **`length`**, How many people were in line at that time.

   All of these values are integers, and variables to hold them are already in Program A.23.

[62] See Program 7.4 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation as you read data from the input file.)

4. Inside the loop, you'll need to do a few things:

   (a) Add up the `wait` values you read from the file. Program A.23 already has a variable named `waitsum` for this. You'll need this sum later to calculate the mean value of `wait`.
   (b) You'll also need to find the sum of the *squares* of the `wait` values. Program A.23 already has a variable named `waitsum2` for this. This sum will be used later for finding standard deviation of `wait`.
   (c) You'll also need to count the number of lines you've read from the file. You'll need this later, along with the sum and sum of squares, to calculate means and standard deviations.
   (d) Lastly, you'll need to keep track of the maximum values of `wait`, `length`, and `t`. Refer to Program 5.4 in Chapter 5 for an example of how to do this. In our analyze.cpp program, we'll need something like this for each thing we want to find the maximum of:

   ```
   if ( initialized == 0 || wait > maxwait ) {
     maxwait = wait;
   }
   ```

   so there should be three "`if`" statements like this (one for `wait`, one for `length`, and one for `t`). After all three "`if`" statements are done, we'll need to tell the program that we now have initial guesses at the maximum values, so we should put in a line that says:

   ```
   initialized = 1;
   ```

An early picture of UVa's marching band, taken by Rufus W. Holsinger (1866-1930).
Source: Wikimedia Commons

A 1919 football game, photographed by Holsinger.

5. After the "`while`" loop is done, we just need to calculate the mean and standard deviation of `wait`. The `printf` statements that are already in Program A.23 will then print out the results.

If you use the program to analyze the file `simulate-5-10.dat` that you created earlier, you should see output like this:

```
Analyzed 10000 people.
Maximum wait was 376 seconds (6.266667 minutes)
Average wait was 104.385000 seconds (1.739750 minutes)
Standard deviation was 80.031982 seconds (1.333866 minutes)
Maximum queue length was 40 people
Time to fill stadium was 5330 seconds (1.480556 hours)
```

Program 3: Visualizing the Wait Times

Your next job is to write a new program named `visualize.cpp` that will read the files produced by `simulate.cpp` and produce a histogram of the wait times. Your program will be similar to program 7.1 in Chapter 7. Your assistant has started writing it for you, as you can see in Program A.24 below. Notice that it defines a 50-element array, `bin`, to hold the histogram data.

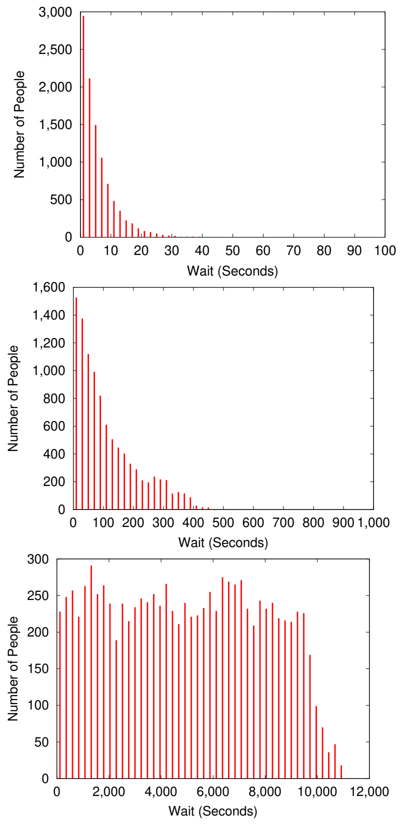


Figure A.62: Histograms of wait times for 5 arrivals/sec, and delays of 5 (top), 10 (middle), and 30 (bottom) seconds.

Program A.24: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  const int nbins=50;
  int bin[nbins] = {0};
  double min, max;
  double binsize;
  int binno;
  int overunderflow=0;
  int i;
  int t, queue, wait, length;
  FILE *input;
  FILE *output;

  // ADD: Write the rest of the program here.

}
```

How to Write the Program

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize min max inputfile outputfile
```

where min and max are the minimum and maximum wait times that can be recorded in the histogram; inputfile is the name of a file that was produced by analyze.cpp; and outputfile is the name of a file into which the new program will write the histogram data. When plotted, the data should look like Figure A.62 on page 596.

To make the histogram, the program should proceed as follows:

1. Check the number of command-line arguments, and use atof to set the values of min and max. The input and output files can be opened like this[63]:

   ```
   input  = fopen(argv[3],"r");
   output = fopen(argv[4],"w");
   ```

[63] Notice that we open one file for reading (with "r") and the other for writing (with "w").

2. Then, determine the binsize, like this:

   ```
   binsize = (max-min)/nbins;
   ```

3. Next, use a "while" loop to read data from the input file[64]. Your "while" loop should start out exactly like the loop in your analyze.cpp program, since it will be reading the same files.

[64] See Chapter 5 for information about reading data from files.

4. Inside the loop, determine which bin each wait value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the range of our histogram is min to max, the bin number will be:

   ```
   binno = (wait-min)/binsize;
   ```

5. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the wait value represented by that bin[65], and the value of bin[i]. The wait value can be calculated from the bin number, like this:

   ```
   wait = min + binsize*(0.5+i);
   ```

   where i is the bin number.

[65] Note that this is different from Program 7.1, where we just printed the bin number as the first column in the output file.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Compile your program and try running it like this, using one of the data files created by your simulate.cpp program:

```
./visualize 0 1000 simulate-5-10.dat visualize-5-10.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize-5-10.dat" with impulses lw 5
```

Your results should look like the graphs in Figure A.62.

Final Thoughts

Congratulations! The simulation, analysis, and visualization programs you've written are powerful tools that will help you keep the spectators happy. Apparently, by controlling the amount of time it takes to do a security scan, and the rate at which people arrive, we can assure that everybody gets into the stadium quickly, nobody has to wait in line too long, and the lengths of the lines aren't too great.

For example, by running

```
./analyze simulate-5-10.dat
```

we can see that, if we limit the arrival rate to 5 people per second, and if our security scan takes 10 seconds on average, we can get everybody into the stadium in 1.5 hours, with people only standing in line for a couple of minutes, and never having more than about 40 people in line at a time.

It's interesting that our results show a remarkable change in the behavior of the queues when we change from a small scan delay to a large one. Take a look again at Figure A.60 and Figure A.61 on page 593.

For small delays, people flow quickly into the stadium. Sometimes there's nobody in line at all. The wait times vary up and down randomly. For long delays, the wait times are long and predictable, falling on a straight line. What we're seeing here is a transition between what we'd call "turbulent flow" and "laminar flow" in fluid dynamics. In the small-delay case there's not much "friction". The crowd behaves like a low-viscosity fluid, such as water. In the long-delay case, the flow is dominated by the "friction" due to the scanning process, and becomes slow and steady, like pouring syrup.[66]

Source: *Wikimedia Commons* *(top)*, *Wikimedia Commons* *(bottom)*

[66] If we did a little math, we could show that the transition from turbulence to laminar flow happens when:

$$\texttt{scan\_delay} = \frac{2 * \texttt{nqueue}}{\texttt{max\_arrivals\_per\_second} - 1}$$

This is the point where people start piling up in the queues because they aren't being scanned fast enough.

Project 8: Auto-Compose

Introduction

Imagine you have a friend who's a songwriter. Your friend has run out of ideas for new melodies, so she asks if you can use your programming skills to give her some inspiration. Let's use a little musical mathematics to generate some random tunes.

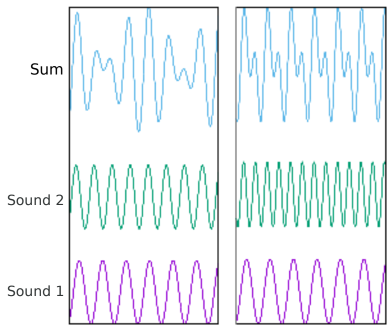


Figure A.63: If we pick two sound frequencies at random, they'll probably combine into something jangly and unpleasant, as shown on the left. If one frequency is twice the other, though, as shown on the right, the two sounds form a nice repeating pattern that's noticed by our ears.

The human mind is wired for finding patterns. We often think of visual patterns, but our ears can also find patterns in sound. For example, as shown in Figure A.63, when a sound has a frequency that's twice that of another sound, the two sounds seem harmonious to us. Our mind knows that they're somehow related.

For millenia, musicians have used this principle to organize sound into *octaves*, where an octave is a range that starts at some base frequency and goes up to twice that starting frequency. In Western music, an octave is typically divided into twelve sections, each starting with a different frequency, or "tone". This twelve-tone system is what we'll be working with in this project.

Twelve Tones

Figure A.64 shows a set of twelve tones that starts with a base frequency of 440 Hz[67]. Each of the tones is obtained by multiplying the preceding term by a constant factor, which we'll call by the Greek letter ρ. If we want to get to twice the original frequency after twelve steps, that means that ρ has to be equal to $\sqrt[12]{2}$ (the twelfth root of two)[68]. If we number the tones starting with zero, then tone number `itone` will have a frequency of $base \times \rho^{itone}$, where `base` is the frequency of tone number zero.

[67] This is the frequency of the A above middle C on a piano.

[68] This is called the 12-tone even-tempered system of tuning.

Notice in Figure A.64 that some of the tones are approximately nice fractions times the base frequency. Tone number 7, for example, is about $\frac{3}{2}$ times the base, and tone 5 is about $\frac{4}{3}$ times the base. Just like pairs of frequencies where one is twice the other, our ears also like to hear sounds that are in small-integer ratios like $\frac{3}{2}$ and $\frac{4}{3}$. This system for picking twelve tones is appealing to musicians because it produces many tones that are approximately small-integer multiples of the base frequency.

itone	Frequency	Ratio (ρ^{itone})	Fraction
0	440.00	1.00	$\sim$ 1/1
1	466.16	1.06	
2	493.88	1.12	$\sim$ 9/8
3	523.25	1.19	
4	554.37	1.26	$\sim$ 5/4
5	587.33	1.33	$\sim$ 4/3
6	622.25	1.41	
7	659.26	1.50	$\sim$ 3/2
8	698.46	1.59	
9	739.99	1.68	$\sim$ 5/3
10	783.99	1.78	
11	830.61	1.89	$\sim$ 15/8
0	880.00	2.00	$\sim$ 2/1

Figure A.64: Twelve tones, starting with a base frequency of 440 Hz. The table also shows a thirteenth tone, which is just twice the base frequency and is the beginning of the next octave. We'll say that this is tone number 0 again, but in a higher octave.

The twelve tones in Figure A.64 cover just one octave (plus the first note of the next octave), but we can keep multiplying by ρ to get higher frequencies, or we could start dividing the base frequency by ρ to get lower frequencies, as shown in Figure A.65. Each time we get to a factor of two we've covered another octave, and we're back to a tone that sounds similar to the base frequency. We'll say that the octave that begins with the base frequency is `octave` number 0. In this project, we'll say that frequencies that differ by exactly an octave (or multiple octaves) have the same tone number (`itone`), but different `octave` numbers.

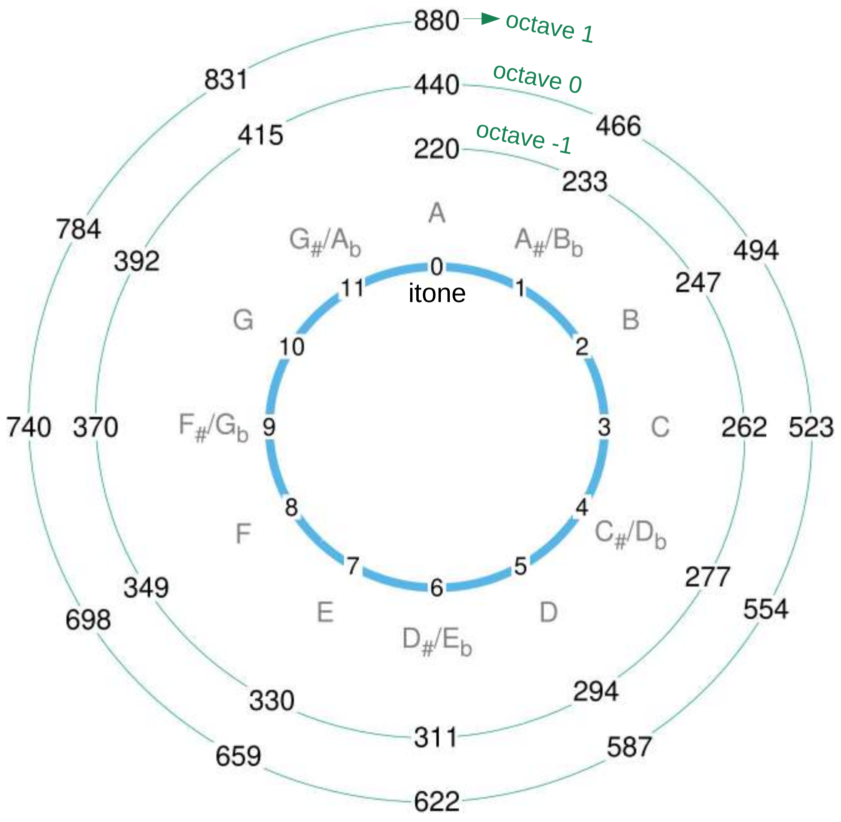


Figure A.65: A spiral of frequencies, starting with a base frequency of 440 Hz and going up and down from there. The frequencies double each time we go around the spiral.

If we choose 440 Hz as our base frequency, the tones in `octave` 0 correspond to the piano keys shown in Figure A.65, where `itone` 0 in `octave` 0 is A above middle C. Figure A.66 shows how these would look on a piano keyboard. The gray letters in Figure A.65 show the standard names for all the tones on a piano. We could choose any base frequency we want, and that would generate a different set of twelve tones with the same ratios between them.

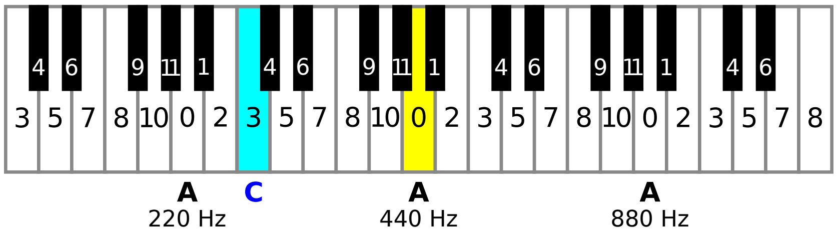


Figure A.66: The center of a piano keyboard, showing middle C and several A keys in different octaves. If we use 440 Hz as the base frequency for our tones, then the A above middle C is `itone` 0 in `octave` 0. The numbers on the keys show the value of `itone` for that key.
Based on a figure from Wikimedia Commons.

Scales and Intervals

Composers don't necessarily use all twelve tones when writing music. They choose a set of tones to make a *scale*. If they do use all twelve of the tones, we call that a *chromatic* scale. We can start with any tone we want. If we started with `itone` 0 on Figure A.65 and went clockwise around the circle picking every tone until we got back to 0, we'd have a "chromatic scale in A". The ratio, or *interval*, between each pair of neighboring notes would be ρ, which we'll call one *step*. Our starting point (`itone` 0 in this case) is called the *tonic* of the scale.

But often composers don't use all of the tones. Look at Figure A.68. In this case, there's an interval of 2 steps between some notes, and an interval of 1 step between others. If we start with `itone` 2 (a B on a piano keyboard) and go from one circled tone to another until we get all the way back to 2, we'll have a "B-major scale" with 7 notes. A *major* scale is any one that has intervals between notes in this pattern.

The type of a scale (chromatic, major, pentatonic, harmonic minor, ...there are a lot!) is determined by the list of intervals between its notes. Figure A.67 shows intervals for several common scales.

Let's use these musical principles to write some programs.

Intervals for a Few Scales

chromatic:	1 1 1 1 1 1 1 1 1 1 1 1
major:	2 2 1 2 2 2 1
pentatonic:	2 2 3 2 3
harmonic minor:	2 1 2 2 1 3 1
major blues:	2 1 1 3 2 3
minor blues:	3 2 1 1 3 2
jazz minor:	2 1 2 2 2 2 1
minor pentatonic:	3 2 2 3 2
ambassel:	1 4 2 1 4
anchihoye:	1 4 1 4 2
bati:	4 2 1 4 1
tezita:	2 1 4 1 4
wenz:	4 1 4 2 1

Figure A.67: List of intervals for several different scales. Notice that for each of these the intervals add up to 12, since that's the total number of steps all the way around the circle of tones. The last five scales are Ethiopian 5-note scales called *kignits*.

Program 1: Simulating a Composer

The first program you'll write is called `simulate.cpp`. It will generate 12 tones starting with a base frequency of 440 Hz. Then it will pick a set of those tones to make a scale, based on a starting tone and a given set of intervals. It will then do a random walk up and down this scale

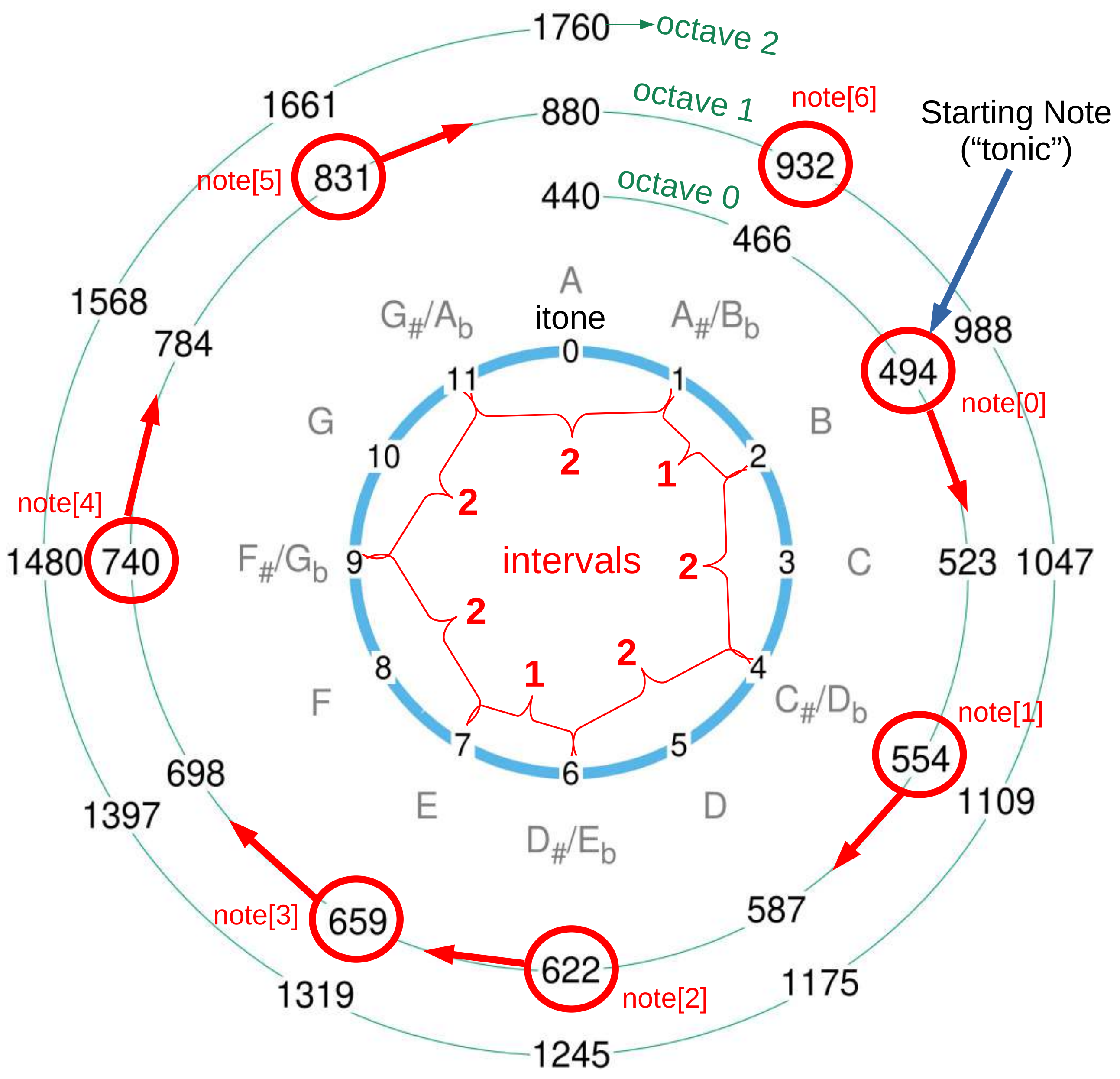


Figure A.68: A B-major scale. It starts with `itone` 2, and steps around the circle in intervals of 2, 2, 1, 2, 2, 2, and 1 steps.

to generate a melody, with some notes skipped at random. Finally, it will write the notes into an output file that can be converted into a playable music file.

Dave Brubeck at the piano.
Source: Wikimedia Commons.

Program A.25 shows part of what this program should look like. Your job is to fill in the missing part, where it says "Insert the rest of the program here!". The program will make use of three functions in the header file named `music.h`, which you can find in the appendix to this project.

The program should be written so that you can run it with command-line arguments like this:

```
./simulate 3 280 0.1 60 major-interval.dat simulate-major-60.dat
```

where the list of command line arguments is:

1. `tonic`: The `itone` number that should be used to start the scale.
2. `npm`: The number of notes per minute that should be played.
3. `restprob`: The probability that the melody will skip a note.
4. `tmax`: How long the melody will last, in seconds.
5. `intervalfile`: A file containing a list of intervals between notes of the scale.
6. `outputfile`: The file that the program's output will be written into.

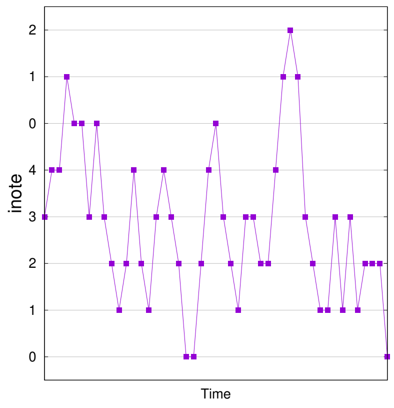


Figure A.69: If we used a 5-note scale, this is what wandering up and down the scale might look like. Here we start at note number 3, then repeatedly move by one or two notes up or down from the note we're currently on.

How to Write the Program

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print a friendly usage message and then stop without trying to do anything else[69].

2. Convert the command-line arguments into the variables `tonic`, `npm`, `restprob`, and `tmax` by using the `atoi` and `atof` functions[70]. The last two command-line arguments (the interval file and output file) don't need to be converted. You can just use them directly, like this[71]:

   ```
   intervalfile = fopen( argv[5], "r" );
   outputfile = fopen( argv[6], "w" );
   ```

3. All of our notes will be of the same length. Calculate the length, in seconds, of a note by dividing 60 seconds by `npm` (notes per minute). Put the result into the variable named `duration`.

4. Next, use a `for` loop to calculate the frequencies of the 12 tones and store those in the array named `tone`. For each value of `itone`, `tone[itone]` should be equal to `base*pow(rho,itone)`, where

[69] See Section 9.16 of Chapter 9 for an example of how to do this.

[70] Notice that `tonic` and `npm` are `int` and `restprob` and `tmax` are `double`.

[71] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

Program A.25: simulate.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "music.h"

int main ( int argc, char *argv[] ) {

  const int ntones = 12;
  double base = 440;
  double tone[ntones];
  double rho = pow( 2, 1.0/ntones );

  int tonic;
  double note[ntones];
  int nnotes=0;
  int steps;
  int octave = 0;
  double npm;
  double duration;
  double restprob;

  double frequency;
  double minfreq = 200;
  double maxfreq = 1300;

  double t = 0;
  double tmax;

  int position, itone, inote, imove;

  FILE *intervalfile;
  FILE *outputfile;


        // Insert the rest of the program here!

}
```

`base` and `rho` are already defined for you in Program A.25.

5. We're going to pick the notes of our scale by starting with the tone we chose as the `tonic` and then walking clockwise around the spiral path, as shown in Figure A.68. We'll use the variable named `position` to keep track of where we are on the spiral. At this point in the program, set `position = tonic` to indicate that this is our starting position..

6. Now use a `while` loop to read intervals from the interval file you opened earlier[72]. Each line of the file contains one number, the number of tone steps between two notes on our scale[73]. We'll store the frequencies of the tones of our scale in the array named `note`. The variable `ntones` tells us how many tones are available[74] and we'll use the variable `nnotes` (which is initialized to zero) to count how many notes are in our scale.

   The `while` loop should read numbers from the interval file into the variable named `steps`. Inside the loop we'll need to do several things:

   (a) First, find the current values of `itone` and `octave` by looking at `position`:

   ```
         itone = position % ntones;
         octave = position / ntones;
   ```

   As we saw in Chapter 4, the `%` operator gives the remainder after division. Also, since both `position` and `ntones` are `int`, doing `position/ntones` tells us how many whole multiples of `ntones` are in `position`.

   (b) Next, use these to find the frequency of this note:

   ```
   note[nnotes] = tone[ itone ]*pow(2,octave);
   ```

   (c) Finally, add `steps` to the value of `position`, and add one to the value of `nnotes`.

   Keep doing this until you've read all of the numbers from the interval file.

7. Next, we need to pick a random note from our scale to start our melody. The header file `music.h` contains a function named `randminmax` that will help us do that. It generates a random number between given minimum and maximum values. We'll use the variable `position` again. Start by setting `position` to a random note on our scale:

   ```
   position = randminmax( 0, nnotes-1 );
   ```

[72] See Program 5.4 in Chapter 5 for an example of reading numbers from a file.

[73] See Figure A.68.

[74] This is set to 12 at the top of our program. If we wanted to use a musical system with a different number of tones, we could change this number to something else.

8. Now we're ready to generate our melody. We'll use the variable `t` to keep track of the elapsed time in our melody, and the variable `frequency` to keep track of the frequency of the current note.

   To prevent damaging our ears or generating notes too low-pitched to hear, the program has two variables named `maxfreq` and `minfreq`. If our notes wander above `maxfreq` we'll reduce it by one octave, cutting the frequency in half. If they wander below `minfreq` we'll increase it by one octave, doubling the frequency.

   With all that in mind, create a `while` loop that keeps repeating for as long as `t` is less than `tmax`.

   Here are the things that need to be done inside the loop:

   (a) First, find the values of `inote` and `octave` based on `position`. We did something similar in the previous while loop, but this time it's trickier because we'll sometimes be dealing with negative numbers. Our path can wander up or down from our starting point, so sometimes `position` will be negative (see Figure A.70). The `emod` function is defined in `music.h`. Unlike the mathematical modulo operator, most computer languages use a modulo operator that gives a negative answer for `n%m` when `n` is negative. The `emod` function does a modulo operation that always gives a positive answer[75], which is what we want because `inote` should always be positive.

   To find `octave` we want to divide `position` by `nnotes` and round down. If `position` were always positive, we could just say `position/nnotes`, since both variables are `int`. But for negative values of `position` this would round *up*, making `-1/7` equal to zero, for example. To always round down, we can use the `floor` function, which is one of C's standard math functions.

   So, with all that in mind here's how we'll find the values for `inote` and `octave`:

```
inote = emod(position,nnotes);
octave = floor( (double)position/nnotes );
```

   (b) Next, check to see if we're supposed to skip this note. We do that by generating a random number between zero and one, using the `rand01` function from `music.h`, and checking whether it's less than `restprob`. If it is, we set `frequency` to 0.

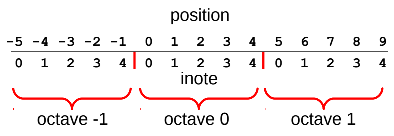


Figure A.70: Here are three octaves of an `nnotes` = 5 scale. Notice that if `position` is 6, then `inote` is 1 and `octave` is 1. This is the same thing we would have gotten by saying `inote = position % nnotes` and `octave = position/nnotes`. But if `position` is -2, `inote` is 3 and `octave` is -1. Because of the negative `position`, the `%` and `/` operators by themselves won't give us the right values for `inote` and `octave` in this case.

75 This is called a Euclidean modulo, which is the modulo operator usually used in mathematics.

Ahmad Jamal at the piano.

Source: Wikimedia Commons.

Otherwise, we set `frequency` to

```
note[inote] * pow( 2, octave )
```

and check to make sure the frequency isn't too low or too high. If `freqeuency` is above `maxfreq`, reduce `position` by `nnotes` and divide `frequency` by 2. If it's below `minfreq`, increase `position` by `nnotes` and double `frequency`.

(c) Now we're ready to write this note into the output file. We'll write two numbers for each note: The duration of the note, in seconds (given by the variable `duration`) and the frequency of the note (given by `frequency`). They should be written as two numbers separated by a space, and followed by a line break[76].

(d) Next we need to decide how far and in which direction we'll walk to get to the next note. We'll move up or down the scale at random by one or two notes. To decide how many notes to move, we can use the `randminmax` function again, like this:

```
imove = randminmax( -2, 2 );
```

(e) Then add `imove` to `position`.

(f) The last thing in the `while` loop should be a line that adds this note's duration to the total elapsed time:

```
t += duration;
```

Shirley Horn at the piano.
Source: Wikimedia Commons.

[76] See Program 5.3 in Chapter 5 for an example of writing numbers into a file.

Running the Program

After you've finished writing and compiling the program you'll need some interval files to try it out with. If you have a nice instructor, they might give you these files. Otherwise, you can copy them from the appendix to this project.

When you have the interval files, try running your program a few times with different ones, like this:

```
./simulate 0 280 0.1 60 major-interval.dat simulate-major-60.dat
./simulate 0 280 0.1 60 harmonic-minor-interval.dat simulate-harmonic-minor-60.dat
./simulate 0 280 0.1 60 pentatonic-interval.dat simulate-pentatonic-60.dat
./simulate 0 280 0.1 60 chromatic-interval.dat simulate-chromatic-60.dat
```

You can get a sense of what the melodies in the output files are like by plotting them with *gnuplot*. For example:

```
  plot "simulate-major-60.dat" using 2 with lines
```

Figure A.71 shows what you might see.

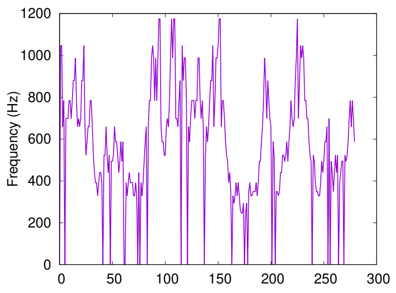


Figure A.71: An output file from the `simulate` program, graphed with *gnuplot*.

But of course what we'd really like to do is *hear* the melody. For that, you can use the `makewav` program in the appendix. You can run it like this:

```
./makewav simulate-major-60.dat major-60.wav
```

This will make a `.wav` file that your computer should be able to play as sound. Try listening to melodies generated with different scales and different starting tones.

Program 2: Analyzing a Melody

Now that you've made some melodies, it would be nice to check your first program's output to make sure it's doing what you expect. One way to do this would be to look at the ratios between successive notes. These should always be multiples of ρ, the ratio between the twelve tones we started with.

Fats Waller at the piano.

Source: Wikimedia Commons.

The second program you'll write is called `analyze.cpp`. It will read the files created by your first program and make a list of the number of steps between each pair of successive notes. Program A.26 shows how the program should start. You'll just need to fill in the missing part.

If frequency f_2 is higher than f_1 by n steps, then the following should be true:

$$\frac{f_2}{f_1} = \rho^n \quad \text{(A.3)}$$

To find n we can take the log of both sides:

$$log(\frac{f_2}{f_1}) = n \times log(\rho) \quad \text{(A.4)}$$

and we can rearrange this to find n:

$$n = \frac{log(\frac{f_2}{f_1})}{log(\rho)} \quad \text{(A.5)}$$

You'll find that if f_2 is *smaller* than f_1 by n steps, it will just change the sign of n. So, if we look at the absolute value of n it will tell us how many steps up or down we need to go to get from one note to the other.

If our `simulate` program is working correctly, all the values of n should be whole numbers.

Program A.26: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
int main ( int argc, char *argv[] ) {
  const int ntones = 12;
  double duration, frequency;
  double oldfreq = 0;
  double ratio;
  double rho = pow( 2, 1.0/ntones );
  double steps;
  FILE *input;
  FILE *output;

        // Insert the rest of the program here!
}
```

Thelonious Monk at the piano.
Source: Wikimedia Commons.

The program should be written so that you can run it with command-line arguments like this:

```
./analyze input output
```

where `input` is the name of one of the files produced by your `simulate` program, and `output` is the name of the file that this new program will create.

How to Write the Program

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print a friendly usage message and then stop without trying to do anything else[77].

[77] See Section 9.16 of Chapter 9 for an example of how to do this.

2. For this program, the two command-line arguments don't need to be converted in any way. You can just use them directly, like this:

```
   input = fopen( argv[1], "r" );
   output = fopen( argv[2], "w" );
```

3. Now use a `while` loop to read lines from the input file you opened earlier[78]. Each line of the file contains two numbers with decimal places: the duration of a note, and the frequency of the note. Put those numbers into the variables named `duration` and `frequency`.

[78] See Program 5.4 in Chapter 5 for an example of reading numbers from a file.

4. Inside the `while` loop you'll need to do several things:

   (a) Some of the notes in the data file will have a frequency of zero. These are the notes we skipped. We want to also skip them in this second program, so you'll need to check to see if `frequency ==`

`0`. If it is, then use a `continue` statement to go back to the top of the loop and get the next line from the file.

(b) Our goal is to find ratios of pairs of notes, so we need to have two notes to compare. To do that, we'll store the frequency of the note we previously read in the variable named `oldfreq`. Then we can look at the ratio of `frequency` to `oldfreq`. You'll notice that `oldfreq` has an initial value of zero, so we'll use this as an indicator that we haven't yet read two frequencies. We're also uninterested in cases where the frequency hasn't changed. With all of that in mind, we'll now need an `if` statement that starts like this:

```
if ( frequency != oldfreq && oldfreq != 0 ) {
```

and inside this `if` statement we'll need to do the following:

i. Find the ratio of `frequency` to `oldfreq` and put it into the variable named `ratio`.

ii. Calculate the absolute value of n (the number of steps) from Equation A.5 above, like this:

```
steps = fabs( log(ratio)/log(rho) );
```

iii. Print the number of steps into the output file, followed by a line break.

(c) At the bottom of the `while` loop, save the current frequency in the variable `oldfreq` by saying:

```
oldfreq = frequency;
```

Count Basie at the piano.
Source: Wikimedia Commons.

After you've written and compiled the program, try running it with some of the output files from your first program. For example:

```
./analyze simulate-chromatic-60.dat analyze-chromatic-60.dat
./analyze simulate-major-60.dat analyze-major-60.dat
./analyze simulate-pentatonic-60.dat analyze-pentatonic-60.dat
```

If you graph the output files with *gnuplot* you should see graphs like Figure A.72. As you can see, there are only whole numbers of steps between the notes.

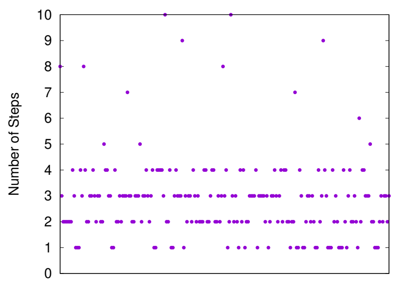


Figure A.72: Output from the `analyze` program, plotted with *gnuplot*.

Program 3: Counting Notes

The output from the `analyze` program tells us that there are integer numbers of steps between notes, but it doesn't tell us how often we take 1 step, 2 steps, 3 steps, and so forth. To find that out, we can make a histogram. That's what your third program will do.

This third program will be called `visualize.cpp` and it will read output files created by the `analyze` program and make histograms from the data in them. Program A.27 shows how the program should start. You just need to fill in the missing part.

Bill Evans at the piano.
Source: Wikimedia Commons.

Program A.27: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  const int nbins=100;
  int bin[nbins] = {0};
  double min, max;
  double binsize;
  double steps;
  int binno;
  int overunderflow=0;
  int i;
  FILE *input;
  FILE *output;


        // Insert the rest of the program here!


}
```

How to Write the Program

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize min max input output
```

where `min` and `max` are the minimum and maximum number of steps that can be recorded in the histogram; `input` is the name of a file that was produced by `analyze.cpp`; and `output` is the name of a file into which the new program will write the histogram data. When plotted, the data should look like those shown Figure A.73 on page 613.

To make the histogram, the program should proceed as follows:

1. Check the number of command-line arguments, and use `atof` to set the values of `min` and `max`. The input and output files can be opened like this:

```
input  = fopen(argv[3],"r");
output = fopen(argv[4],"w");
```

2. Then, determine the `binsize`, like this:

```
binsize = (max-min)/nbins;
```

3. Next, use a `while` loop to read data from the input file[79]. Your `while` loop should read one number from the input file each time it goes around. It should stick the number into the variable named `steps`.

Abdullah Ibrahim at the piano.
Source: Wikimedia Commons.

[79] See Chapter 5 for information about reading data from files.

4. Inside the loop, determine which bin each `steps` value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does[80]. Since the range of our histogram is `min` to `max`, the bin number will be:

```
binno = (steps-min)/binsize;
```

[80] See Section 7.1 of Chapter 7 for information about making histograms.

5. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the `steps` value represented by that bin[81], and the value of `bin[i]`. The `steps` value can be calculated from the bin number, like this:

```
steps = min + binsize*(0.5+i);
```

where `i` is the bin number.

[81] Note that this is different from Program 7.1, where we just printed the bin number as the first column in the output file.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Compile the program and run it like this, using some data files created by your `analyze` program:

```
./visualize 0 13 analyze-chromatic-60.dat visualize-chromatic-60.dat
./visualize 0 13 analyze-major-60.dat visualize-major-60.dat
./visualize 0 13 analyze-pentatonic-60.dat visualize-pentatonic-60.dat
```

The output files can be graphed with *gnuplot* commands like this:

```
plot "visualize-chromatic-60.dat" with impulses
```

This should produce graphs like those shown in Figure A.73.

It's possible to identify the scale a melody uses by looking at these histograms. For example, the pentatonic scale has no intervals of 1 step[82], so there are no `step` values of 1 in the pentatonic histogram.

[82] see Figure A.67 on page 601

On the other hand, all of the intervals of the chromatic scale are 1 step, so if our melody goes up or down the scale by 1 or 2 notes, we'd expect to only see intervals of 1 or 2 steps in the chromatic histogram. This is mostly the case, but there are also a few values in the histogram's 3 column. This is because our simulation program sometimes intentionally skipped a note, which could leave a larger than expected gap between the notes we hear. Also, if you look closely you might see some data in the 10 or 11 column. These happened when the simulation program was at note 0 or 1 and stepped down from there, going to a high note in a lower octave.

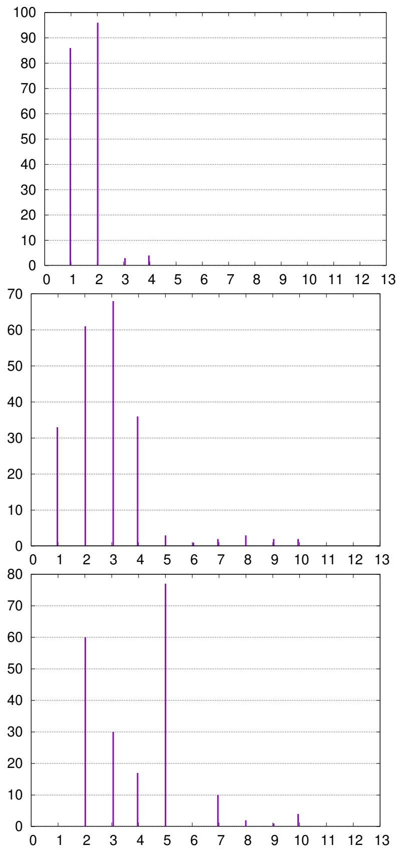


Figure A.73: Histograms made with the `visualize` program. The top graph shows the result for a melody produced using a chromatic scale. The middle graph is a major scale. The bottom graph is a pentatonic scale.

Conclusion

Hopefully you've created some melodies to inspire your songwriter friend.

There are many enhancements you could add to your `simulate` program. For example, you could take a short series of notes and repeat it, maybe with modifications like raising or lowering the last note every other time it's repeated, or raising or lowering all the notes by an octave. Also, instead of only going up or down by one or two notes, you could sometimes, at random, leap to a really high or low note.

You might think about how `simulate.cpp` could be modified so to create melodies that have more than one note playing at the same time. You'd need to think about which notes sound good together, and maybe do some more research into the mathematics of chords. You'd also need to modify `makewav.cpp`, if you wanted to hear the melodies.

For more information about the mathematics of music, I highly recommend the series of videos that Norman Wildberger has been doing recently on his Insights into Mathematics Youtube channel.

Hank Jones at the piano.

Source: Wikimedia Commons.

Appendix

- **The `music.h` header file**
  Here's the `music.h` header file that you'll need for writing `simulate.cpp`:

Program A.28: music.h

```
// Random number between 0 and 1:
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

// Random number between min and max:
double randminmax( int min, int max ) {
  return ( min + (int)( (max-min+1)*rand01() ) );
}

// Euclidean modulus (always positive):
int emod ( int n, int modulus ) {
  return (   (n%modulus + modulus)%modulus );
}
```

- **Some Interval Files**
  Here are the contents of three files that give the interval sequence for a major scale, a harmonic minor scale, and a pentatonic scale. As you can see, they're just lists of the numbers of steps between notes on each scale. You can create other files using the intervals listed in Figure A.67.

major-interval.dat
```
2
2
1
2
2
2
1
```

harmonic-minor-interval.dat
```
2
1
2
2
1
3
1
```

pentatonic-interval.dat
```
2
2
3
2
3
```

You can also create a new scale by taking the top interval from the major scale and moving it to the bottom. This is called the *Dorian* mode. By continuing to move the top interval to the bottom, you can produce other scales. The next one is *Phrygian*, then *Lydian*, *Mixolydian*, *Aeolian*, and *Locrian*. If you do it one more time, you're back to where you started, which is called the *Ionian* mode.

- **The `makewav` program**
  Here's the `makewav` program that you can use to convert the output from your `simulate` program into playable `.wav` files. In order to compile it, you'll also need to fetch two other files:

  ```
  https://www3.nd.edu/~dthain/courses/cse20211/fall2013/wavfile/wavfile.c
  https://www3.nd.edu/~dthain/courses/cse20211/fall2013/wavfile/wavfile.h
  ```

  You can use *wget* or *curl* to download these. After you have them and the `makewav.cpp` file below, you can compile the `makewav` program by typing:

  ```
  g++ -Wall -O -o makewav makewav.cpp wavfile.c
  ```

Program A.29: makewav.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "wavfile.h"

double safefreq ( double frequency ) {
  double minfreq = 200;
  double maxfreq = 1300;
  if ( frequency > maxfreq || frequency < minfreq ) {
    return(0);
  } else {
    return(frequency);
  }
}

int main( int argc, char *argv[] ) {
  double duration, frequency;

  double t = 0;
  double tsample = 1.0/WAVFILE_SAMPLES_PER_SECOND;
  int nsamples;

  short amplitude;
  int volume = 8000;

  int i;

  double oldphase = 0, phase;

  FILE *input;
  FILE *output;


  input = fopen( argv[1], "r" );
  if(!input) {
```

```
    printf("couldn't open %s for writing: %s",argv[1],strerror(errno));
    exit(1);
  }


  output = wavfile_open( argv[2] );
  if(!output) {
    printf("couldn't open %s for writing: %s",argv[2],strerror(errno));
    exit(1);
  }

  while ( fscanf( input, "%lf %lf", &duration, &frequency ) != EOF ) {
    frequency = safefreq( frequency );

    nsamples = duration/tsample;
    phase = oldphase - frequency*t*2*M_PI;
    for ( i=0; i<nsamples; i++ ) {
      amplitude = volume*sin(frequency*t*2*M_PI + phase);
      oldphase = frequency*t*2*M_PI + phase;
      wavfile_write(output,&amplitude,1);
      t += tsample;
    }
  }

  wavfile_close(output);
  fclose(input);
}
```