Final Project: Virus

Introduction

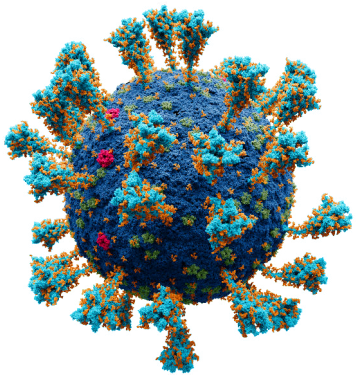
Imagine a time in the future when a new virus, deadlier than COVID-19, is ravaging the country. You're a health-care professional working on the island of Natucket, off the coast of Massachusetts. So far, no one on Natucket has contracted the virus, but you're a good computer programmer and your boss 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 1 on page 4). 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 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 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. copied?

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.

14?

The program should assume that vaccinated people are perfectly protected, and have no chance of becoming infected[1].

[1] 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 1. 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 appendix at the end of this document.

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 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 1: 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 1). 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[2].

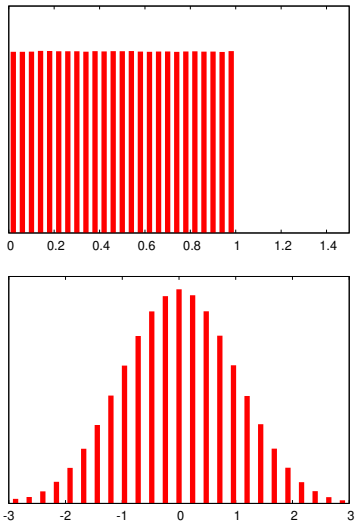


Figure 1: Histograms of random numbers in a uniform distribution (top) from `rand01()` and a normal distribution (bottom) from `normal()`.

[2] See the values in Table 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[3]; `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:

[3] 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[4]. (Remember that `status[person]` is the number of days an infected 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.

[4] See the bottom graph in Figure 1.

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[5]. 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[6].

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

[6] 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[7]. Use it like this:

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

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

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:

   used pointer not copy

   ```
   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:

   collect on the fly?

   ```
   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 2 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 2: 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[8] global CFR for COVID-19 is about 2% and for seasonal flu in the US it's about 0.2%.

[8] 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 lc scol title "Susceptible", \
 "" using 1:3 with lines lw 5 lc rcol title "Recovered", \
 "" using 1:4 with lines lw 5 lc vcol title "Vaccinated", \
 "" using 1:5 with lines lw 5 lc icol title "Infected",\
 "" using 1:6 with lines lw 5 lc dcol title "Dead"
```

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

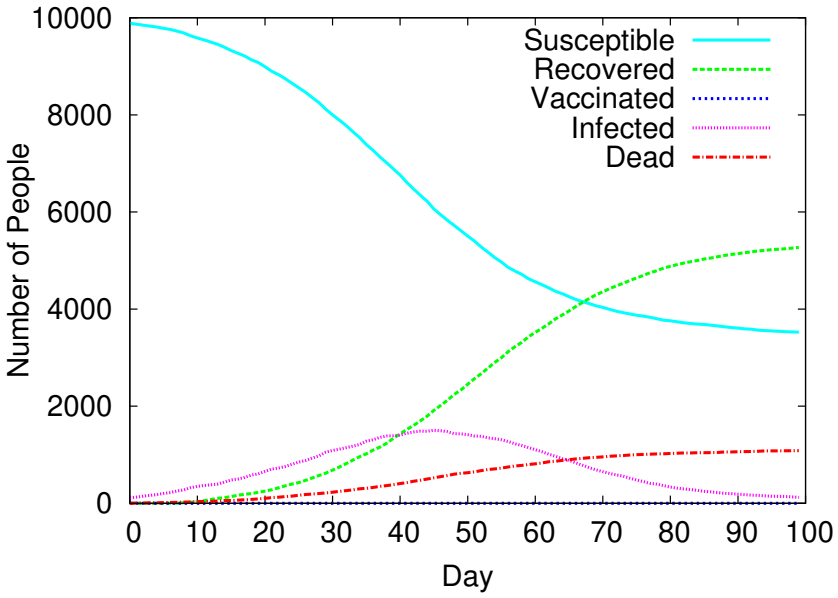


Figure 3: Results of a simulation with `vprob=0` and `tprob=0.015`.

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[9], and should further reduce the number of deaths. Some typical results are shown in Figure 4.

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 4: 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.)

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

Figures 5 and 6 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`**.

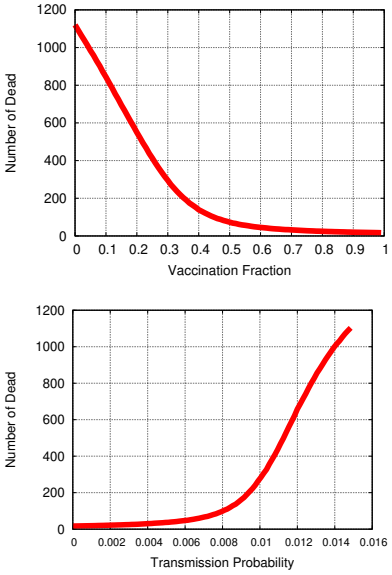


Figure 5: The top graph shows the number of deaths after 100 days for various values of `vprob` when `tprob` is 1.5%. The bottom graph shows number of deaths for various values of `tprob` when `vprob` is 0%.

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.)

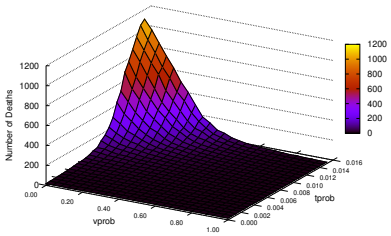


Figure 6: Effect of changing `vprob` and `tprob` on the number of deaths after 100 days.

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[10].

[10] 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
```

It should tell you something similar to Figure 7, 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.

```
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 7: Typical output from `analyze.cpp` when `vprob=0.0` and `tprob=0.015`.

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 3 below. Notice that it defines a 50-element array, `bin`, to hold the histogram data.

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

Program 3: 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 8 on page 12.

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[11]:

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

[11] 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[12]. Each line of the file will contain six integer values: `trial`, `nsusceptible`, `nrecovered`, `nvaccinated`, `ninfected`, and `ndead`.

[12] 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[13], 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.

[13] 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.

[14] 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[14]:

```
./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 8, 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.

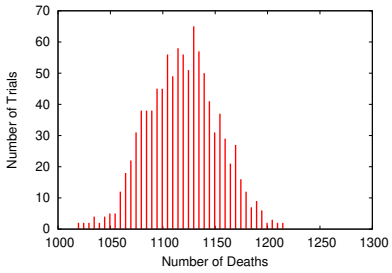


Figure 8: Histogram of number of deaths after 100 days, from 1,000 trials with `vprob=0` and `tprob=0.015`.

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[15]. As a step in this direction, in 2021 the CDC announced the formation of a new "Center for Forecasting and Outbreak Analytics"[16]. Efforts like this should provide fertile new ground for young programmers who want to make the world a better place.

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

Appendix: Two 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 1 on page 5 for an illustration of the output from the two functions.

Program 4: 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 5: 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;
  }

}
```