Introduction to Computational Physics

Reading Days Edition!

Welcome back!

Remember that you can try out these programs using your account on Galileo. For instructions on how to use Galileo, see:

http://galileo.phys.virginia.edu/compfac/courses/comp_intro/connecting.html

Today we'll look at last week's practice problems, and work on a new challenge together.

Let's get started!

Last Week's Practice Problem #1:

```
int i;
int j;
int sum;
int nproblems;
int nright = 0;
srand(time(NULL));
for ( nproblems = 0; nproblems < 10; nproblems++ ) {
  i = 100.0 * rand()/(double)RAND_MAX;
  j = 100.0 * rand()/(double)RAND_MAX;
  printf("What is %d + %d ?: ",i,j);
  scanf("%d",&sum);
  if ( i+j == sum ) {
    printf("Right!\n");
    nright++;
  } else {
    printf("Nope. The sum is %d. Go back to school.\n", i+j);
  }
}
printf("Your score is %d / %d (%lf %%)\n",
       nright,
       nproblems,
       nright*100.0/nproblems );
```

I asked if you could modify the "math practice" program to make it keep score. Here's one way to do that:

Literal percent sign.

Note that I've omitted the "framework" of "int main ()", etc. around the program. You'll need to add it back in to get a working program.

Note that we've just added a counter, "nright", to keep track of how many problems the user answers correctly. Then at the end, the program prints out this information in a useful way. Note that we've used "%%" in the printf statement to cause it to print a literal "%". You need to do this because otherwise printf thinks you've mistyped %d, %lf or some other format specifier. By typing %% you're telling printf "Yes, I really do want you to print out a percent sign."

Last Week's Practice Problem #2:

I also asked if you could randomly choose addition or subtraction problems. Here's one way to do that:

```
int i;
int j;
int sum;
int nproblems;
double flip; // Result of "coin flip".
int answer;
srand(time(NULL));
for ( nproblems = 0; nproblems < 10; nproblems++ ) {
  i = 100.0 * rand()/(double)RAND_MAX;
  j = 100.0 * rand()/(double)RAND_MAX;
  flip = rand()/(double)RAND_MAX; // Flip coin.
  if ( flip < 0.5 ) {
          printf("What is %d + %d ?: ",i,j);
          answer = i+j;
  } else {
          printf("What is %d - %d ?: ",i,j);
          answer = i-j;
  }
  scanf("%d",&sum);
  if ( answer == sum ) {
    printf("Right!\n");
  } else {
    printf("Nope. The sum is %d. Go back to school.\n", answer);
  }
}
```

Note that I've omitted the “framework” of “int main ()”, etc. around the program. You'll need to add it back in to get a working program.

Here, I've used a random number to simulate a coin flip. The number “rand()/(double)RAND_MAX” is a random number between zero and one. In the “if” statement, I check to see if this number is less than or greater than 0.5, essentially assigning “heads” to one case and “tails” to the other. Can you see how we could generalize this for an unfair coin? Or for a 6-sided die?

Note that, since we don't know in advance whether the user will get an addition or a subtraction problem, we now have to store the problem's correct answer in a variable (“answer”) so we can compare it with what the user enters, rather than just calculating it on the fly.

Practice Problem: Zombie Plague!

Zombies are taking over the world! Quick, let's write a program to simulate the spread of the Zombie Plague!

Here are the rules:

- Zombism is infectious! When a zombie encounters a non-zombie, there's a good chance that the non-zombie will become a new zombie.
- Zombies travel slowly, so they don't encounter too many people each day, and the people they do encounter can run much faster than a zombie. Given these limitations, a zombie can only zombify three or fewer people per day.
- It all starts with one zombie....

Given these rules, can we write a program that will predict how many zombies there will be thirty days after the first zombie appears?

Let's work through this problem together.

As you read along, try to figure it out yourself before going on to the next page.

Taking a First Stab at it:

Here's some advice for getting started:

- Naturally, you're going to need some variables. I recommend these:

```
int zombies = 1; // Total number of zombies. Start with 1.
int newzombies; // Total number of new zombies zombified today.
int ndays = 30; // Number of days.
int i,z; //Some counters. Use as needed.
double maxencounter = 3; // Max. # of people encountered by zombie per day.
double encounters; // Actual number of people encountered by a zombie today.
double infected; // Number of people infected by a zombie today.
```

- Let's start out by assuming everyone the zombie encounters gets zombified. So, “infected” will always equal “encounters”.

- I recommend you print out the total number of zombies each day, from day zero to the end.

How would you go about writing the program? Give it a try before you look at the following pages.

The program's strategy should be something like this:

“Each day, for each zombie, find a random number of encounters (less than *maxencounters*). At the beginning of the day, set *newzombies* = 0. As you look at each zombie, add the number of people it zombifies to *newzombies*, until at the end of the day *newzombies* contains the total number of new zombies for that day. Add that to the total number of zombies, and print it out.”

One Possible Way to Do It:

Here's one way to write the program:

```
srand(time(NULL));
printf ("Starting with %d zombie(s)\n",zombies);
for ( i=0; i<ndays; i++ ) {
  newzombies = 0; // Reset number of new zombies for today.
  for ( z=0; z<zombies; z++ ) {
    // Number of people this zombie encounters today:
    encounters = maxencounter * ( rand()/(double)RAND_MAX );
    // Assume everyone encountered gets zombified:
    infected = encounters;
    // Add infected to number of zombies today:
    newzombies += infected + 0.5;
  }
  zombies += newzombies;

  printf ("Day %d: %d zombies\n", i,zombies);
}
```

This, combined with the fact that *newzombies* is an "int", rounds the number to the nearest integer. Can you understand why?

This uses the strategy described on the preceding page.

Note that I've left out the "#include" lines, the "int main () {", and the variable definitions (see preceding slide). You'll need to add these back in to have a working program.

What happens when you compile and run this program? You may be surprised by a couple of things:

- "Wow! The number of zombies goes up really fast!"

- "Waitaminnit! Why does it suddenly turn into a negative number?!!"

You'll also notice that the program goes slower and slower as you accumulate zombies.

Population Explosion:

If you ran the preceding program, you saw that the number of zombies gets really big really fast.

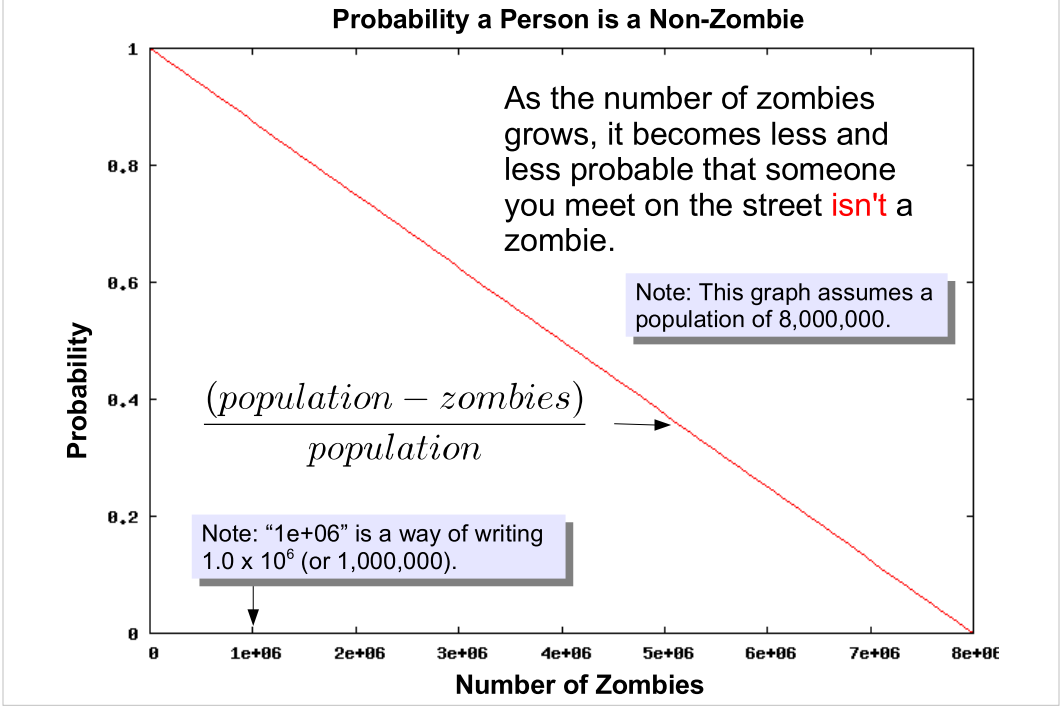
In fact, it gets so big that an “int” variable can't hold the number any more, and starts behaving oddly like a negative number.

On our machines, “int” variables can only hold numbers up to about 2 billion! There's nothing in our model that would ever stop the growth of the zombie population. Given long enough, it would approach infinity!

Maybe our zombie simulation is unrealistic. For example, once most people have been zombified, most zombies would be unable to find anyone to make into a new zombie.

Maybe we need to take this into account in our program. How could we do that? Think about it before looking at the next page.

Zombie Probability:



As more and more people are zombified, the probability that an encountered "person" isn't already a zombie falls lower and lower. We can estimate the probability as

(population – zombies)/(population)

This is equal to 1 when there are zero (or very few) zombies, and equal to 0 when all of the population has been zombified.

Using Probability:

When you flip a fair coin, the probability that it will land heads-up is 0.5 (50%).

In a C program, if we generate a random number between 0 and 1, there's a 50% probability that the number will be less than 0.5.

So, we can simulate a coin flip by looking to see if such a random number is less thanor greater than 0.5, as we did in the solution to one of last week's practice problems.

Similarly, the probability of rolling “6” (or any other given number) on a six-sided die is 1/6 (about 0.17).

We could simulate the roll of a die in a program by generating a random number. We could say that if the number is > 0.17 it's a “6”, if it's between 0.17 and 0.34 it's a “5”, and so on. (It doesn't matter which side of the die we assign to each range, as long as we do it the same way throughout the program.)

We can use this to simulate any kind of random event for which we know the probability.

In our zombie example, we're going to assume that the probability of meeting a non-zombie has a value given by:

(population – zombies)/population

which decreases as the number of zombies grows. On any given day of our simulation, we'll know how many zombies there are, and can calculate the probability. We can then “flip a coin” to decide whether a person is a zombie or not.

Putting on the Brakes:

Two new things:
- Don't exceed the total population
- Gets harder to find somebody to zombify

Some new variables:

```
int j;
double nonzombies; // Number encountered who are non-zombies.
double population = 8000000; // Population (VA = 8,000,000).
double flip;
```

Change the mid-section:

```
for ( i=0; i<ndays; i++ ) {
  newzombies = 0; // Reset number of new zombies for today.
  for ( z=0; z<zombies; z++ ) {
    // Number of people this zombie encounters today:
    encounters = maxencounter * ( rand()/(double)RAND_MAX );
    nonzombies = 0;
    for ( j=0; j<encounters; j++ ) {
            flip = rand()/(double)RAND_MAX;
            if ( flip < (population - zombies)/population ) {
              nonzombies++;
            }
    }
    infected = nonzombies; // Assume every non-zombie encountered gets zombified:
    newzombies += infected + 0.5;
  }
  zombies += newzombies;
  if ( zombies > population ) { // Can't exceed total population!
      zombies = population;
  }
  printf ("Day %d: %d zombies\n", i,zombies);
}
```

Flip a “coin” to see it this person is a non-zombie

Let's assume that Virginia closes its borders shortly after the zombie outbreak is discovered, so the maximum number of zombies is the total population of Virginia.

In the program above, every time a zombie encounters somebody we flip a virtual “coin” to decide whether the person has already been infected with zombism. This isn't a 50/50 coin though. Instead of comparing “flip” to 0.5, as we did in the solution to one of last week's practice problems, we compare it to the ever-decreasing probability that this person is a non-zombie.

We also put an explicit test into our program to make sure the number of zombies can never exceed the total population. We need this in case our program happens to generate a big batch of zombies just before we reach the population limit.

Try running the program. Do you see how it behaves differently?

That's still a lot of zombies, isn't it?!

Zombie Vaccine: A New Hope!

Until now, we've assumed that any non-zombie who meets a zombie automatically gets zombified.

What if that's not true? What if some people have a natural immunity, or if they've been given an anti-zombie vaccine?

Even if the vaccine isn't 100% effective, or if it's only given to a fraction of the population, it might still slow down the plague.

How might we modify our program to simulate the effect of partial immunity?

Let's assume that there's some non-zero probability that any non-zombie won't become zombified. Then we can use our virtual coin-flipping skills to decide whether an encounter results in infection.

We could either look at this as the probability that an encountered person is 100% immune to zombism (with some people being unvaccinated), or the probability that some partially-effective vaccine will protect a vaccinated individual. (Or some combination of the two scenarios.) We're not worrying about the details, we'd just like to see what happens if, somehow, some people are immune.

Accounting for Immunity:

To add the immunity effect to our program, we need to flip a "coin" again. This time, we check to see if each non-zombie we encounter is immune to zombism:

Here's the previous program:

Previous Program

```
nonzombies = 0;
for ( j=0; j<encounters; j++ ) {
        flip = rand()/(double)RAND_MAX;
        if ( flip < (population - zombies)/population ) {
              nonzombies++;
        }
}
```

To add immunity, we add another coin flip before counting our "converts":

```
double susceptibility = 0.1;
```

New Program

```
nonzombies = 0;
for ( j=0; j<encounters; j++ ) {
        flip = rand()/(double)RAND_MAX;
        if ( flip < (population - zombies)/population ) {
            // Is this person susceptible to zombism?
            flip = rand()/(double)RAND_MAX;
            if ( flip < susceptibility ) {
              nonzombies++;
            }
        }
}
```

The bottom example shows a new variable "susceptibility", which we've set to 0.1 (10%), meaning that only 10% of the population is susceptible to being zombified.

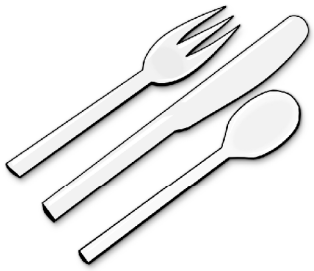
Try out this modification to the program, and try different values for susceptibility. Does it behave the way you expect? For example, this program should produce results similar to the preceding program if you set the susceptibility very high, but you should see almost no zombies if susceptibility is very low.

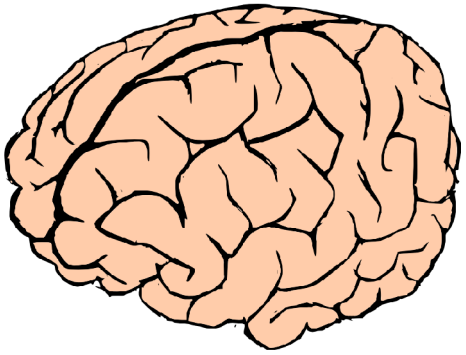
Going Forward:

What else could we do to improve our simulation? How could we make it more realistic?

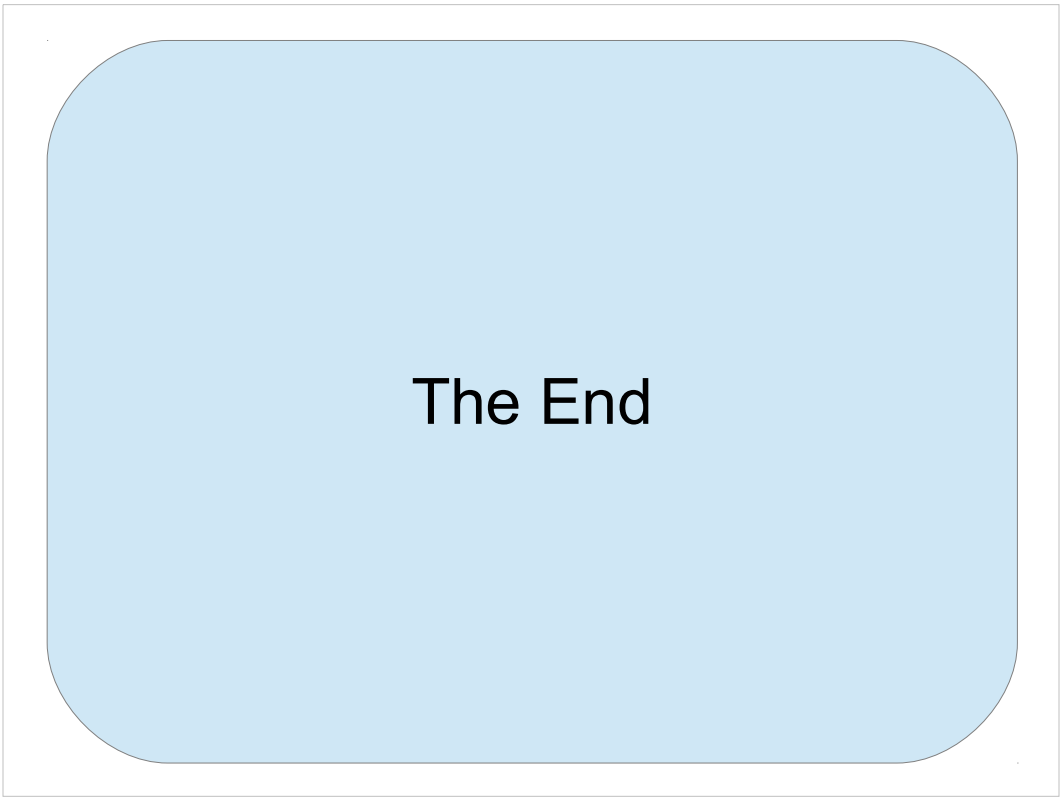
For one thing, we might think about the “life” span of the zombies. Do they live forever? Even if they don't suffer from natural death, do stalwart Virginians fight them off? How might we account for opposing forces of zombie creation and zombie destruction in our program?

You should be able to come up with lots of ways to improve on what we've done so far. Think about it! ...with your braaaaiiiiiins.





As I'm sure you've noticed, nothing we've done here is specific to zombism. The same general ideas are relevant to any contagious disease, or indeed to anything that's transmitted from one individual to another. (It could be applied to memes on the internet, for example.)



Thanks!