6.00: Introduction to Computer Science and Programming

Problem Set 12: Simulating Virus Population Dynamics

Handed out: Thursday, November 20, 2008
Due: Friday, December 5, 2008

Introduction

In this problem set, you will design and implement a stochastic simulation of virus population dynamics.

There are medications for the treatment of infection by viruses; however, viruses may become resistant to one drug, sometimes to multiple drugs due to mutations. Despite not having gone to medical school (or maybe because of this), you can still decide on a good drug treatment regimen by observing how the virus population responds to the introduction of different drugs. We have been unable to reserve a bio lab for 6.00, so you will have to simulate the virus population dynamics with Python and reach conclusions based on the simulation results.

For this problem set, you should submit both your code, **ps12.py**, and a writeup in pdf format, **writeup.pdf** (you may find this online pdf converter useful).

Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

Collaboration

You may work with other students. However, each student should write up and hand in his or her assignment separately. Be sure to indicate with whom you have worked. For further details, please review the collaboration policy as stated in the syllabus.

Getting Started

Download and save this file.

- ps12.py: The code template for the simulation

Background: Viruses, Drug Treatments, and Computational Models

Viruses such as HIV and Influenza represent a significant challenge to modern medicine. One of the reasons that they are so difficult to treat is because of their ability to evolve.

As you may know from introductory biology classes, the traits of an organism are determined by its genetic code. When organisms reproduce, their offspring will inherit genetic information from their parent. This genetic information will be modified, either due to mixing of the two parents' genetic information, or through errors in the genome replication process, thus introducing diversity into a population.

Viruses are no exception and carry and propogate their own genetic information. Two characteristics of viruses make them particularly difficult to treat. The first is that their replication mechanism often lacks the error checking mechanisms that is present in more complex organisms. Secondly, viruses replicate extremely quickly, orders of magnitude faster than humans. Thus, while we may be used to thinking of evolution as a process which occurs over long time scales, populations of viruses can undergo substantial evolutionary changes within a single patient over the course of treatment.

These two characteristics allow a virus population to quickly acquire genetic resistance to therapies over the course of a treatment. In this problem set, we will make use of simulations to explore the effect of introducing drugs on the virus population and determine how best to address these treatment challenges within a simplified model.

Computational modeling has played an important role in the study of viruses such as HIV (for example, see this paper, by Time Magazine's Man of the Year, David Ho). In this problem set, we will implement a highly simplified stochastic model of virus population dynamics in vivo. Many details have been swept under the rug (host cells are not explicitly modeled and the size of the population is several orders of magnitude less than the size of actual virus populations). Nevertheless, our model exhibits biologically relevant characteristics and will give you a chance to analyze and interpret simulation data.

Problem 1: Implementing a Simple Simulation (No Drug Treatments)

We start with a trivial model of the virus population - the patient does not take any drugs and the viruses do not acquire resistance to drugs. We simply model the virus population in a patient as if it were left untreated.

At every time step of the simulation, each virus particle has a fixed probability of being cleared (eliminated from the patient's body). If the virus particle is not cleared, it is considered for reproduction. Unlike the clearance probability, which is constant, the probability of a virus particle reproducing is a function of the virus population. With a larger virus population, there are fewer resources in the patient's body to facilitate reproduction, and the probability of reproduction will be lower. One way to think of this limitation is to consider that virus particles need to make use of a the patient's cells to reproduce, they cannot reproduce on their own. As the virus population increases, there will be fewer available host cells for viruses to utilize for reproduction.

To implement this model, you will need to fill in the `SimpleVirus` class, which maintains the state of a single virus particle, and the `SimplePatient` class, which maintains the state of a virus population associated with a patient. The `update()` method in the `SimplePatient` class is the "inner loop" of the simulation. It modifies the state of the virus population for a single time step and returns the total virus population at the end of the time step.

`update()` should first decide which virus particles are cleared and which survive by making use of the `doesClear()` method of each `SimpleVirus` instance and update the collection of `SimpleVirus` instances accordingly. `update()` should then call the `reproduce()` method for each virus particle. Based on the population density, `reproduce()` should either return a new instance of `SimpleVirus` representing the offspring of the virus particle, or raise a `NoChildException` indicating that the virus particle does not reproduce during the current time step. The `update()` method should update the attributes of the patient appropriately under either of these conditions. After iterating through all the virus particles, the `update()` method returns the number of virus particles in the patient at the end of the time step.

The `reproduce()` method in `SimpleVirus` should produce an offspring by returning a new instance of `SimpleVirus` with probability:

```
self.maxBirthProb * ( 1 - popDensity)
```

`self.maxBirthProb` is the birth rate under optimal conditions (the virus population is negligible relative to the available host cells). `popDensity` is defined as the ratio of the current virus population to the maximum virus population for a patient and should be calculated in the `update()` method of the `SimplePatient` class.

HINT: Be very careful about mutating a variable while iterating over its elements. Either avoid doing this entirely (consider introducing additional "helper" variables), or make absolutely sure that the resulting values in the data structure are correct (for example, carefully think about the order in which you delete variables from a list).

Note that the mapping between time steps and actual time will vary depending on the type of virus being considered, but for this problem set, think of a time step as a simulated hour of time.

See the template for detailed specifications for each of the methods in these classes.

Fill in the implementation for the `__init__()`, `doesClear()`, and `reproduce()` methods of the `SimpleVirus` class according to the specifications. Use `random.random()` for generating random numbers to ensure that your results are consistent with ours.

Fill in the implementations for the `__init__()`, `getTotalPop()`, and `update()` methods of the `SimplePatient` class.

You will test your implementation in problem 2.

```
class SimpleVirus(object):
    """
    Representation of a simple virus (does not model drug effects/resistance).
    """

    def __init__(self, maxBirthProb, clearProb):
        """
        Initialize a SimpleVirus instance, saves all parameters as attributes
        of the instance.

        maxBirthProb: Maximum reproduction probability (a float between 0-1)

        clearProb: Maximum clearance probability (a float between 0-1).
        """
        # TODO

    def doesClear(self):
        """
        Stochastically determines whether this virus is cleared from the
        patient's body at a time step.

        returns: Using a random number generator (random.random()), this method
        returns True with probability self.clearProb and otherwise returns
        False.
        """
        # TODO

    def reproduce(self, popDensity):
        """
        Stochastically determines whether this virus particle reproduces at a
        time step. Called by the update() method in the SimplePatient and
        Patient classes. The virus particle reproduces with probability
        self.maxBirthProb * (1 - popDensity).

        If this virus particle reproduces, then reproduce() creates and returns
        the instance of the offspring SimpleVirus (which has the same
        maxBirthProb and clearProb values as its parent).

        popDensity: the population density (a float), defined as the current
        virus population divided by the maximum population.

        returns: a new instance of the SimpleVirus class representing the
        offspring of this virus particle. The child should have the same
        maxBirthProb and clearProb values as this virus. Raises a
        NoChildException if this virus particle does not reproduce.
        """
        # TODO

class SimplePatient(object):
    """
    Representation of a simplified patient. The patient does not take any drugs
    and his/her virus populations have no drug resistance.
    """

    def __init__(self, viruses, maxPop):
        """
        Initialization function, saves the viruses and maxPop parameters as
        attributes.

        viruses: the list representing the virus population (a list of
        SimpleVirus instances)

        maxPop: the  maximum virus population for this patient (an integer)
        """
        # TODO

    def getTotalPop(self):
        """
        Gets the current total virus population.

        returns: The total virus population (an integer)
        """
        # TODO

    def update(self):
        """
        Update the state of the virus population in this patient for a single
        time step. update() should execute the following steps in this order:

        - Determine whether each virus particle survives and updates the list
          of virus particles accordingly.

        - The current population density is calculated. This population density
          value is used until the next call to update()

        - Determine whether each virus particle should reproduce and add
          offspring virus particles to the list of viruses in this patient.

        returns: the total virus population at the end of the update (an
        integer)
        """
        # TODO
```

Problem 2: Running and Analyzing a Simple Simulation (No Drug Treatments)

You should start by understanding the population dynamics before introducing any drug. Fill in the function `problem2()`. This method should instantiate a `SimplePatient` and repeatedly call the `update()` method to simulate changes in the virus population over time. Save the population values over the course of the simulation and use pylab to plot the virus population as a function of time. Be sure to title and label your plot.

`SimplePatient` should be instantiated with the following parameters:

- `viruses`, a list of 100 `SimpleVirus` instances
- `maxPop`, Maximum Sustainable Virus Population = 1000

Each `SimpleVirus` instance in the `viruses` list should be initialized with the following parameters:

- `maxBirthProb`, Maximum Reproduction Probability for a Virus Particle = 0.1
- `clearProb`, Maximum Clearance Probability for a Virus Particle = 0.05

Fill in the function problem2() which instantiates a patient, simulates changes to the virus population for 300 time steps (i.e. 300 calls to `update()`), and plots the virus population as a function of time. Run the simulation multiple times and pick a representative plot to include in your writeup. Don’t forget to include axes labels and a title on your plot.

To add the plot to your writeup, click on the disk icon at the bottom of the figure window and save an image file. Use Microsoft Word (or your word processor of choice) to import the image file.

In your writeup, include the plot and answer this question: about how long does it take before the population stops growing?

```
def problem2():
    """
    Run the simulation and plot the graph for problem 2 (no drugs are used,
    viruses do not have any drug resistance).

    Instantiates a patient, runs a simulation for 300 timesteps, and plots the
    total virus population as a function of time.
    """
    # TODO
```

Problem 3: Implementing a Simulation With Drugs

In this problem, we consider the effects of both administering drugs to the patient and the ability of virus particle offspring to inherit or mutate genetic traits that confer drug resistance.

As the virus population reproduces, mutations will occur in the virus offspring, adding genetic diversity to the virus population. Some virus particles gain favorable mutations that confer resistance to drugs.

Drugs are given to the patient using the Patient class's `addPrescription()` method. What happens when a drug is introduced? The drugs we consider do not directly kill virus particles lacking resistance to the drug, but prevent those virus particles from reproducing (much like actual drugs used to treat HIV). Virus particles with resistance to the drug continue to reproduce normally.

In order to model this effect, we introduce a subclass of `SimpleVirus`, `ResistantVirus`. `ResistantVirus` maintains the state of a virus particle's drug resistances, and account for the inheritance of drug resistance traits to offspring.

We also need a representation for a patient which accounts for the use of drug treatments and manages a collection of `ResistantVirus` instances. For this we introduce the `Patient` class, which is a subclass of `SimplePatient`. `Patient` must make use of the new methods in `ResistantVirus()` and maintain the list of drugs that are administered to the patient.

See the template for detailed specifications for methods of these two classses.

Implement the `ResistantVirus` and `Patient` classes.

You will test your implementation in problem 4.

```
class ResistantVirus(SimpleVirus):
    """
    Representation of a virus which can have drug resistance.
    """

    def __init__(self, maxBirthProb, clearProb, resistances, mutProb):
        """
        Initialize a ResistantVirus instance, saves all parameters as attributes
        of the instance.

        maxBirthProb: Maximum reproduction probability (a float between 0-1)

        clearProb: Maximum clearance probability (a float between 0-1).

        resistances: A dictionary of drug names (strings) mapping to the state
        of this virus particle's resistance (either True or False) to each drug.
        e.g. {'guttagonol':False, 'grimpex',False}, means that this virus
        particle is resistant to neither guttagonol nor grimpex.

        mutProb: Mutation probability for this virus particle (a float). This is
        the probability of the offspring acquiring or losing resistance to a drug.
        """
        # TODO

    def getResistance(self, drug):
        """
        Get the state of this virus particle's resistance to a drug. This method
        is called by getResistPop() in Patient to determine how many virus
        particles have resistance to a drug.

        drug: the drug (a string).

        returns: True if this virus instance is resistant to the drug, False
        otherwise.
        """
        # TODO

    def reproduce(self, popDensity, activeDrugs):
        """
        Stochastically determines whether this virus particle reproduces at a
        time step. Called by the update() method in the Patient class.

        If the virus particle is not resistant to any drug in activeDrugs,
        then it does not reproduce. Otherwise, the virus particle reproduces
        with probability:

        self.maxBirthProb * (1 - popDensity).

        If this virus particle reproduces, then reproduce() creates and returns
        the instance of the offspring ResistantVirus (which has the same
        maxBirthProb and clearProb values as its parent).

        For each drug resistance trait of the virus (i.e. each key of
        self.resistances), the offspring has probability 1-mutProb of
        inheriting that resistance trait from the parent, and probability
        mutProb of switching that resistance trait in the offspring.

        For example, if a virus particle is resistant to guttagonol but not
        grimpex, and `self.mutProb` is 0.1, then there is a 10% chance that
        that the offspring will lose resistance to guttagonol and a 90%
        chance that the offspring will be resistant to guttagonol.
        There is also a 10% chance that the offspring will gain resistance to
        grimpex and a 90% chance that the offspring will not be resistant to
        grimpex.

        popDensity: the population density (a float), defined as the current
        virus population divided by the maximum population

        activeDrugs: a list of the drug names acting on this virus particle
        (a list of strings).

        returns: a new instance of the ResistantVirus class representing the
        offspring of this virus particle. The child should have the same
        maxBirthProb and clearProb values as this virus. Raises a
        NoChildException if this virus particle does not reproduce.
        """
        # TODO

class Patient(SimplePatient):
    """
    Representation of a patient. The patient is able to take drugs and his/her
    virus population can acquire resistance to the drugs he/she takes.
    """

    def __init__(self, viruses, maxPop):
        """
        Initialization function, saves the viruses and maxPop parameters as
        attributes. Also initializes the list of drugs being administered
        (which should initially include no drugs).

        viruses: the list representing the virus population (a list of
        SimpleVirus instances)

        maxPop: the  maximum virus population for this patient (an integer)
        """
        # TODO

    def addPrescription(self, newDrug):
        """
        Administer a drug to this patient. After a prescription is added, the
        drug acts on the virus population for all subsequent time steps. If the
        newDrug is already prescribed to this patient, the method has no effect.

        newDrug: The name of the drug to administer to the patient (a string).

        postcondition: list of drugs being administered to a patient is updated
        """
        # TODO

    def getPrescriptions(self):
        """
        Returns the drugs that are being administered to this patient.

        returns: The list of drug names (strings) being administered to this
        patient.
        """
        # TODO

    def getResistPop(self, drugResist):
        """
        Get the population of virus particles resistant to the drugs listed in
        drugResist.

        drugResist: Which drug resistances to include in the population (a list
        of strings - e.g. ['guttagonol'] or ['guttagonol', 'grimpex'])

        returns: the population of viruses (an integer) with resistances to all
        drugs in the drugResist list.
        """
        # TODO

    def update(self):
        """
        Update the state of the virus population in this patient for a single
        time step. update() should execute these actions in order:

        - Determine whether each virus particle survives and update the list of
          virus particles accordingly

        - The current population density is calculated. This population density
          value is used until the next call to update().

        - Determine whether each virus particle should reproduce and add
          offspring virus particles to the list of viruses in this patient.
          The listof drugs being administered should be accounted for in the
```

```
              determination of whether each virus particle reproduces.

            returns: the total virus population at the end of the update (an
            integer)
            """
            # TODO
```

Problem 4: Running and Analyzing a Simulation with a Drug

In this problem, we will use the implementation you filled-in for problem 3 to run a simulation. You will create a `Patient` instance with the following parameters, then run the simulation and answer several questions:

- `viruses`, a list of 100 `ResistantVirus` instances
- `maxPop`, Maximum Sustainable Virus Population = 1000

Each `ResistantVirus` instance in the `viruses` list should be initialized with the following parameters:

- `maxBirthProb`, Maximum Reproduction Probability for a Virus Particle = 0.1
- `clearProb`, Maximum Clearance Probability for a Virus Particle = 0.05
- `resistances`, The virus's genetic resistance to drugs in the experiment = {'guttagonol':False}
- `mutProb`, Probability of a mutation in a virus particle's offspring = 0.005

Run a simulation that consists of 150 time steps, followed by the addition of the drug, guttagonol, followed by another 150 time steps. As with problem 2, perform multiple trials and make sure that your results are repeatable and representative.

Plot the record of the total population and the population of guttagonol-resistant virus particles. What trends do you observe? Are the trends consistent with your intuition? Include the plot and your answers to these questions in the writeup.

```
    def problem4():
        """
        Runs simulations and plots graphs for problem 4.

        Instantiates a patient, runs a simulation for 150 timesteps, adds
        guttagonol, and runs the simulation for an additional 150 timesteps.

        total virus population vs. time  and guttagonol-resistant virus population
        vs. time are plotted
        """
        # TODO
```

Problem 5: The Effect of Delaying Treatment on Patient Outcome

In this problem, we explore the effect of delaying treatment on the ability of the drug to eradicate the virus population. You will need to run multiple simulations to observe trends in the distributions of patient outcomes.

Run the simulation for 300, 150, 75, and 0 time steps before administering guttagonol to the patient. Then run the simulation for an additional 150 time steps. Use the same initialization parameters for `ResistantVirus` and `Patient` as you did for Problem 4.

For each of the 4 conditions, repeat the experiment multiple times, while recording the final virus populations. Use pylab's hist() function to plot a histogram of the final virus populations under each condition. The x-axis of the histogram should be the final total virus population and the y-axis of the histogram should be the number of patients belonging to each histogram bin. You should decide the number of times you need to repeat each condition in order to obtain a reasonable distribution. Justify your decision in your writeup.

Include the four histograms in your writeup and answer the following questions: If you consider final virus particle counts of 0–50 to be cured (or in remission), what percentage of patients were cured (or in remission) at the end of the simulation? What is the relationship between the number of patients cured (or in remission) and the delay in treatment? Explain how this relationship arises from the model.

HINT: It may take some time to run enough trials to arrive at a distribution for each condition. Debug your code using a small number of trials. Once your code is debugged, use a larger number of trials and expect the simulation to take a few minutes. Use `print` statements to monitor the simulation's progress. The simulation should take about 3–6 minutes to run a reasonable number of trials.

```
def problem5():
    """
    Runs simulations and make histograms for problem 5.

    Runs multiple simulations to show the relationship between delayed treatment
    and patient outcome.

    Histograms of final total virus populations are displayed for delays of 300,
    150, 75, 0 timesteps (followed by an additional 150 timesteps of
    simulation).
    """
    # TODO
```

Problem 6: Designing a Treatment Plan with Two Drugs

One approach to addressing the problem of acquired drug resisstance is to use cocktails - administration of multiple drugs that act independently to attack the virus population.

In problems 6 and 7, we use two independently-acting drugs to treat the virus. We will use this model to decide the best way of administering the two drugs. Specifically, we examine the effect of a lag time between administering the first and second drugs on patient outcomes.

For problems 6–7, use the following parameters to initialize a `Patient`:

- `viruses`, a list of 100 `ResistantVirus` instances
- `maxPop`, Maximum Sustainable Virus Population = 1000

Each `ResistantVirus` instance in the `viruses` list should be initialized with the following parameters:

- `maxBirthProb`, Maximum Reproduction Probability for a Virus Particle = 0.1
- `clearProb`, Maximum Clearance Probability for a Virus Particle = 0.05
- `resistances`, The virus's genetic resistance to drugs in the experiment = {'guttagonol':False 'grimpex':False}
- `mutProb`, Probability of a mutation in a virus particle's offspring = 0.005

Run the simulation for 150 time steps before administering guttagonol to the patient. Then run the simulation for 300, 150, 75, and 0 time steps before administering a second drug, grimpex, to the patient. Finally, run the simulation for an additional 150 time steps.

For each of these 4 conditions, repeat the experiment 30 times, while recording the final virus populations. Use pylab's hist() function to plot a histogram of the final total virus populations under each condition.

Include the histogram in your writeup and answer the following: What percentage of patients were cured (or in remission) at the end of the simulation? What is the relationship between the number of patients cured (or in remission) and the time between administering the two drugs?

HINT: As with problem 5, the simulation wll take a few minutes to run. Use `print` statements to monitor the simulation's progress.

```
def problem6():
    """
    Runs simulations and make histograms for problem 6.

    Runs multiple simulations to show the relationship between administration
    of multiple drugs and patient outcome.

    Histograms of final total virus populations are displayed for lag times of
    150, 75, 0 timesteps between adding drugs (followed by an additional 150
    timesteps of simulation).
    """
    # TODO
```

Problem 7: Analysis of Virus Population Dynamics With Two Drugs

To better understand the relationship between patient outcome and the time between administering the drugs, we examine the virus population dynamics of two individual simulations from problem 6 in more detail.

Run a simulation for 150 time steps before administering guttagonol to the patient. Then run the simulation for an additional 300 time steps before administering a second drug, grimpex, to the patient. Then run the simulation for an additional 150 time steps. Use the same initialization parameters for `Patient` and `Resistantvirus` as you did for problem 6.

Run a second simulation for 150 time steps before simultaneously administering guttagonol and grimpex to the patient. Then run the simulation for an additional 150 time steps.

Make sure you run the simulation multiple times to ensure that you are analyzing results that are representative of the most common outcome.

For both of these simulations, plot the total population, the population of guttagonol-resistant virus, the population of grimpex-resistant virus, and the population of viruses that are resistant to both drugs as a function of time.

Explain why the relationship between the patient outcome and the time between administering the two drugs arises.

```
def problem7():
    """
    Run simulations and plot graphs examining the relationship between
    administration of multiple drugs and patient outcome.

    Plots of total and drug-resistant viruses vs. time are made for a
    simulation with a 300 time step delay between administering the 2 drugs and
    a simulations for which drugs are administered simultaneously.
    """
    # TODO
```

Problem 8: Patient Non-compliance

A very common problem is that a patient may not consistently take the drugs they are prescribed. They can sometimes forget or refuse to take their drugs. Describe in your writeup (do not write any code) how you would model such effects.

Hand-In Procedure

1. **Save.** Your modified code should be in a file called `ps12.py`. Your writeup should be called `writeup.pdf`.

2. **Time and collaboration info.** At the start of the file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 12
# Name: Jane Lee
# Collaborators: John Doe
# Time: 1:30

... your code goes here ...
```

3. **Sanity checks.** After you are done with the problem set, do these sanity checks:

    - Run the `ps12.py` and make sure it can be run without errors.