1. Objective

The objectives of this assignment are to a) exercise your Object Oriented Programming skills; b) model and solve a simple variant of a commonly occurring optimization problem using ideas from Discrete Event Simulation.

2. Problem

This is a continuation of Lab 8. The problem under consideration is the following:

A medical emergency room is going to be established in a particular neighborhood. It is expected that \( n \) people will be served by this emergency room. It is also supposed that the time between visits to the emergency room by a patient will be normally distributed with mean \( \mu_s \) and standard deviation \( \sigma_s \) (actually the time from previous discharge to current entry). The time a patient is paired with a doctor in the emergency room will be supposed to be normally distributed as well with parameters \( \mu_p \) and \( \sigma_p \). The average wait time of a patient is desired to be no greater than \( w \).

How many doctors should be hired to meet this requirement? Suppose a maximum of 15 doctors is allowed?

In Lab 8 it was determined that the following classes could be developed to solve this problem: queue, eventList, event, doctor, patient, eventManager, and receptionist. All but the last two should have been completed in Lab 8. In this lab we will develop the receptionist and eventManager classes and some code that will be used to gather statistics and plot the results.

3. Analysis

The event manager maintains the event list and performs repetitions of the following operations after the system is initialized:

1. Remove an event object, call it \( \text{evt} \), from the event list.
2. Set the epoch clock to the epoch obtained from \( \text{evt} \).
3. Send \( \text{evt} \) to the receptionist.
4. Get back a list of event objects from the receptionist.
5. Set the epochs of each event object returned by the receptionist. All event objects received from the receptionist contain time durations, and not epochs, because the epoch clock was not passed to the receptionist. For each of these event objects, the event manager computes its epoch by adding its time duration to the current epoch clock and then replaces the time duration with the computed epoch.
6. Insert all those event objects into the event list.
The system variables maintained by the receptionist are: \texttt{doc} of type \texttt{queue} (to hold doctors in the doctor’s pool); \texttt{pat} of type \texttt{queue} (to hold patients in the waiting area); the simulation parameter variables \texttt{sickMean}, \texttt{sickStdDev}, \texttt{pairMean}, \texttt{pairStdDev}, all are of type number (corresponding to $\mu_s$, $\sigma_s$, $\mu_p$, $\sigma_p$, respectively); and all variables associated with keeping statistics, namely \texttt{count} (the number of patients having waited for a doctor) and \texttt{totalWaitTime} (the total amount of time waited by all patients who have been paired with a doctor).

The receptionist is asked by the event manager to process information over many rounds. In each round, the receptionist is responsible for processing a single \texttt{event} object handed to it by the event manager and creating new \texttt{event} objects based on the results of processing. The receptionist places all new \texttt{event} objects in a list (of type \texttt{queue}) which is returned to the event manager. The function of the receptionist in each round depends on the type of event handled and includes one of the following:

1. **Event is of type 1**: That is, it has a doctor and no patient. This means the doctor is to be placed in the doctor’s pool. In this case, the receptionist performs an \texttt{insert} on \texttt{doc} and returns nothing. No new event is created. This only happens during initialization.

2. **Event is of type 2**: That is, it has a patient and no doctor. This means the patient has just become sick and is entering the emergency room. There are two subcases:
   
   (a) *If doc is not empty.* A doctor is removed from \texttt{doc} and paired with the patient in an \texttt{event} object that also stores a new time duration that is computed from the simulation parameters \texttt{pairMean} and \texttt{pairStdDev}. The \texttt{event} object is \texttt{inserted} into the list of \texttt{event} objects that is to be returned to the event manager. The total waiting time remains untouched. The count of patients having waited is increased by one.

   (b) *If doc is empty.* The \texttt{waitStartEpoch} variable of the patient is set to the current epoch. This value will be used when the patient is paired with a doctor to determine wait time. Then, the patient is \texttt{inserted} into \texttt{pat}.

3. **Event is of type 3**: That is, it has a doctor and a patient. This means a doctor who has been paired with a patient has just finished treatment. The receptionist places the doctor in \texttt{doc} and a new \texttt{event} object is created and contains the freed patient and a time duration which is computed from \texttt{sickMean} and \texttt{sickStdDev}. The \texttt{event} object is added to the list of \texttt{event} objects that will be returned to the event manager. If \texttt{pat} is not empty, a doctor is removed from \texttt{doc} and paired with a patient that is removed from \texttt{pat}. A new \texttt{event} object is created and contains the removed doctor and patient objects along with a time duration computed from \texttt{pairMean} and \texttt{pairStdDev}. The \texttt{event} object is added to the list of \texttt{event} objects that will be returned to the event manager. The current epoch is subtracted from the patient object’s stored \texttt{epoch} value to give the time the patient waited for a doctor. The \texttt{totalWaitTime} variable is increased by that amount and the \texttt{count} variable is increased by 1.

Except for initialization, after each round, the receptionist returns a list of events to the event manager.

The code that runs the simulation needs to perform the following tasks:

1. Get input parameters from the user.

2. Create a number of doctors according to the parameters and add them to the receptionist’s doctor’s pool (\texttt{doc}).
3. Create a number of patients (rather, soon to be patients) according to the parameters and add them to the receptionist’s waiting area queue (pat).

4. Start the simulation by invoking a method of the event manager.

5. Plot the results.

Steps 2.-4. will be repeated numerous times, each time adding one doctor to the pool initially. When data from all simulations is collected, the results will be plotted as a graph of waiting time versus number of doctors.

4. Coding

This section traces the thinking process in developing code for the remaining classes. It is natural to design the top level code first because it will help us to see how best to design the supporting classes. Doing so in this case we realize it is probably best to communicate to the event manager, but not the receptionist, directly from the top level. Therefore, we present considerations for top level development first, then the event manager, finally the receptionist.

4.1 Controlling the simulation

First we must determine what kind of input a user should be expected to provide. This is a simulation covering a large community of people. But it could be any number of people, according to the problem specification which uses $n$ to describe this number. So, the community size should be a parameter. The simulation will depend on assumptions about time to becoming sick and treatment time. These are given in the problem statement as parameters $\mu_s$, $\sigma_s$, $\mu_p$, $\sigma_p$. so these should be parameters too. Parameters such as these are easily entered as input from the console, for example as follows:

```python
npat = input('Number of patients..');
smean = input('Mean time to patient sickness..');
sstddev = input('Standard deviation of time to patient sickness..');
pmean = input('Mean time paired with a doctor..');
pstddev = input('Standard deviation of time paired with a doctor..');
```

The code will make several runs, each time with the number of doctors increased by 1. It is easy to record the result of each run as an array element. Therefore, at the outset of every run we can define

```python
result = [];
```

and the end of every run use something like

```python
result = [result runSim(X, iter)];
```

where runSim is a method of the class X belongs to and returns the result (average wait) of a simulation lasting iter epochs.

What object should X be? Well, it seems the most logical choice is the event manager because we said the event manager is going to manage the event list and therefore the simulation. We need to choose iter to be great enough to allow the simulation to reach steady state. We suppose $10\times npat$ is enough for now. Since the event manager needs to remember the simulation parameters we input, they will be passed as arguments to the constructor. Thus, the line:
em = eventManager(smean, sstddev, pmean, pstddev);

needs to appear before each run and runSim will have to be developed as a method of the event manager class.

The only thing remaining to worry about is the initialization of the doctor’s pool and the event list. These should be handled by the receptionist. We can create a receptionist here but it would have to be passed to the event manager to allow the close communication needed by both. Perhaps a better idea is to let the event manager create the receptionist so that the controlling code will only have to worry about one simulation object to communicate with. We choose to do it this way. Then, to perform the initialization we need to do something like this:

```matlab
for i=1:ndoc addDoctorToPool(em, doctor()); end
for i=1:npat addPatientSickEvent(em, patient()); end
```

where npat is the number of patients as input by the user, ndoc is the number of doctors and changes from run to run, and addDoctorToPool and addPatientSickEvent must be methods of the event manager class. Observe, since we do not care about names and addresses in this simulation, the constructors for patient and doctor objects contain no arguments.

All of the lines shown in this section can be arranged to control the simulation except that a `for ndoc=1:15` loop must be added to get a result array that will plot for differing numbers of doctors.

The plot itself can be handled with

```matlab
plot(result);
xlabel('Number of doctors');
ylabel('Average wait');
```

### 4.2 Event Manager

From considerations in Section 4.1 we know what the form of the constructor for this class needs to be, what variables it needs to keep track of, and we know the methods `addDoctorToPool`, `addPatientSickEvent`, and `runSim` must be implemented. Let’s start with the constructor. The first line must be:

```matlab
function em = eventManager(sickMean, sickStdDev, pairMean, pairStdDev);
```

and all the arguments must be saved in this class. In addition, an event list (which we call `events`) must be created using something like:

```matlab
em.events = eventList;
```

A clock variable, for keeping track of the current epoch, must also be created and initialized to 0. We said we would create a receptionist here and we can do so with a line such as:

```matlab
em.recep = receptionist(sickMean, sickStdDev, pairMean, pairStdDev);
```

We will consider `runSim` next because it is straightforward. The first line must be as follows from Section 4.1:
function result = runSim(em, n);

where n is the number of iterations. This function merely needs to remove an object evt from
the event list (the eventList class has a remove method which returns an event object), hand
evt to the receptionist (create a performFunction method of the receptionist class with argu-
ments that include evt), receive a list of events from the receptionist (this is what is returned by
performFunction), and insert those events into the event list (create an append method of the
eventList class that takes such a list as input and makes the insertions accordingly). You should
build a for i=1:n loop that performs these operations - it should be straightforward to do this.
The return value of runSim (which is result) can be obtained from totalWaitTime/count, both
values coming from the receptionist object created by the event manager.

Now consider addDoctorToPool. This method has a doctor object as argument and its action
results in the insertion of the doctor object into the doc queue of the receptionist. We can create
a type 1 event, and pass it to the receptionist who will do the insertion. We can write:

function addDoctorToPool(em, doctor);
    rec = em.recep;
    performFunction(rec, event(doctor, 0.0, 0.0));
    em.recep = rec;
    assignin('caller', inputname(1), em);
end

where performFunction, a method of the receptionist class, is yet to be developed. Do something
similar for addPatientSickEvent. In that case, the patient object must be inserted into the event
list. There is no need to use the receptionist to do this. Just create an event object with the
patient object and the sickMean and sickStdDev values, and insert into events.

4.3. Receptionist

Here is a reasonable constructor for the receptionist class:

function recep = receptionist(sickMean, sickStdDev, pairMean, pairStdDev);
    % Save simulation parameters
    recep.sickMean = sickMean;
    recep.sickStdDev = sickStdDev;
    recep.pairMean = pairMean;
    recep.pairStdDev = pairStdDev;
    % Create and initialize result variables
    recep.count = 0;          % Number of patients who have waited
    recep.totalWaitTime = 0.0; % Total time that patients have waited
    % Create and initialize local structures
    recep.doc = queue;        % The queue implementing the doctor’s pool
    recep.pat = queue;        % The queue implementing the patient’s waiting area
    recep = class(recep, ’receptionist’);
end

The one major method that needs to be developed is performFunction whose first line must look
like this:

function lst = performFunction(recep, evt);
where \( \text{evt} \) is an event. In the body of the method, the current \( \text{clock} \) value can be obtained from \( \text{evt} \) using the \text{getEpoch} method (see Lab 8). A local \text{queue} object must be created as well to store events generated by \text{performFunction} that must be passed to the event manager. There needs to be a sequence of \text{if-end} statements testing for event type of \( \text{evt} \) and performing the appropriate action according to Section 3.

5. Results

Output is expected to look like that shown in Figures 1 to 3. Thus, if the number of patients is 10, \( \mu_s = 10, \sigma_s = 1, \mu_p = 10, \sigma_p = 1 \), and the desired average wait time is 15, 3 doctors should suffice. If the number of patients is 100, \( \mu_s = 10, \sigma_s = 1, \mu_p = 10, \sigma_p = 1 \), and the desired average wait time is 100, 10 doctors should suffice. If the number of patients is 1000, \( \mu_s = 100, \sigma_s = 1, \mu_p = 10, \sigma_p = 1 \), and the desired average wait time is 2000, 5 doctors should suffice.

6. Submission

Submit sets of \( m \) files which implement classes \text{receptionist} and \text{eventManager} on or before June 1 using blackboard. Also submit \( m \) files of classes \text{doctor}, \text{patient}, \text{event}, \text{eventList}, and \text{queue}, which you have already submitted the previous week. New files should be \text{getWait.m}, \text{receptionist.m}, \text{showPatients.m}, \text{getNumberWaiters.m}, \text{performFunction.m}, \text{showDoctors.m} (\text{receptionist class}), and \text{display.m}, \text{getWait.m}, \text{showDoctors.m}, \text{addDoctorToPool.m}, \text{runSim.m}, \text{eventManager.m}, \text{nextEvent.m}, \text{showEventList.m}, \text{addPatientSickEvent.m}, \text{getNumberWaiters.m}, \text{showPatients.m} (\text{eventManager class}). See the course webpage at

\[ \text{http://gauss.ececs.uc.edu/Courses/HTML/E112.html} \]

for instructions.
Figure 1: Output of simulation with the following parameters: number of patients=10, $\mu_s = 10$, $\sigma_s = 1$, $\mu_p = 10$, $\sigma_p = 1$.

Figure 2: Output of simulation with the following parameters: number of patients=100, $\mu_s = 10$, $\sigma_s = 1$, $\mu_p = 10$, $\sigma_p = 1$.

Figure 3: Output of simulation with the following parameters: number of patients=1000, $\mu_s = 100$, $\sigma_s = 1$, $\mu_p = 10$, $\sigma_p = 1$. 