

Simulation Notes.

These notes describe some techniques useful for the stochastic simulation of engineering systems.

The use of a computer to conduct simulation experiments on a software model of some complex system is a useful technique. It allows us to perform studies of complicated systems whose analytic solution is not known, or perhaps known only approximately, and where it would be either impractical or impossible to experiment on the actual system. For example, simulation can be an important part of conducting design studies of proposed systems, in order to determine performance differences between different configurations.

The basic idea of simulation is to describe the interaction between the elements of the system by a software model. Once this software model has been generated, it can be driven by providing some input data describing the timing of the various events in the model. This timing information may be experimentally measured information, but for stochastic simulation is usually provided by a random number generator.

We see that simulation basically involves conducting experiments on the model of the system.

To illustrate a few concepts, let's consider a simple example – a single server queue with an infinite waiting room.

The actual software model for such a system is very simple. The possible events that occur in the system are:

1. The arrival of a new customer,

and

2. The departure of the customer at the head of the queue. (This coincides with the next customer in the queue beginning its service).

For our software model, the information that we need to know is simply the length of the queue at any instant. Let's call this variable q . An arrival to the system simply increments the queue length of the system. A C function to do this is:-

```
void arrival() /* a customer arrives */
{
    q = q+1;
}
```

Similarly, for a departure, we simply need to decrement the queue length:-

```
void departure() /* a customer departs */
{
    q = q-1;
}
```

We see that the software model is pretty simple!

The bit that we haven't described yet is the complicated bit: how we provide the timing information to the model. Fortunately, this bit is pretty much the same for all simulations, and so only needs to be done once.

Timing information falls into two subsets: the determination of the actual time intervals to be used, and the manipulation of these so that events are performed in the correct order. The actual time intervals to be used are determined from a pseudo-random number generator. Pseudo-random number generators are found as utilities on many computers, and usually return a number uniformly distributed on the unit interval $[0,1)$ (the end points may be inclusive or exclusive depending on the particular generator). We need to be able to convert these random variables into rv's drawn from the distributions we are interested in. This can be done using the transform method as follows:

Transform Method

Let X be the random variable we are interested in generating, and let $F(x)$ be its distribution function, ie

$$F(x) = \Pr\{X \leq x\}$$

Let $F(X) = Y$. Then Y is defined on the range between 0 and 1. We can show that if Y is a random variable uniformly distributed between 0 and 1, the variable X defined by

$$X = F^{-1}(Y)$$

has the cumulative distribution $F(X)$, ie

$$\Pr\{X \leq x\} = \Pr\{Y \leq F(x)\} = F(x)$$

For our simulation, we generate Y with the built-in pseudo-random number generator, and apply the inverse function F^{-1} to it to give us our desired random variable.

Example

Let's assume that the arrival process to the single server queue mentioned above is a Poisson Process with rate λ , so that the times between arrivals have a negative exponential distribution, with mean $1/\lambda$, ie $F(x) = 1 - e^{-\lambda x}$. The procedure outlined above says that we generate a uniform random variable Y , set $Y = 1 - e^{-\lambda X}$, so that

$X = F^{-1}(Y) = -\frac{\ln(1-Y)}{\lambda}$ is our desired variable. If `random()` is the function which returns the uniform random variable, the the following function will give us the required negative exponential.

```
float negexp(mean) /* returns negexp rv with mean 'mean' */
float mean;

{
  return(-log(random()) * mean)
}
```

Scheduling

Now, we need to consider how to schedule these events that we can now generate. First, let's just consider our Poisson Process. We could first generate all the arrival points of the Poisson Process and store them for later use in the simulation. However, this is not very efficient. A better idea is to use 'bootstrapping'. Here, we initially generate only the first arrival event. Then, when the simulation moves to that time, we generate the arrival time of the next arrival event before processing this current arrival event. This means that the arrival times of successive customers are only generated as they are needed. Suppose that we have a procedure `schedule(time_interval, event)` which schedules an event of type 'event' at a time 'time_interval' in the future. Then our previous function `arrival()`, needs to be modified as follows:

```
void arrival() /* a customer arrives */
{
    schedule(negexp(iat),1);
    q = q+1;
}
```

Here, `iat` is the mean inter-arrival time of customers, and we take event type '1' to signify that an arrival has been scheduled. Of course, we also need to schedule some departures from the queue, so if we take event type '2' to be a departure, and the mean service time required by a customer from the server to be negative-exponential with a mean of `service_time`, then the final version of this function becomes:

```
void arrival() /* a customer arrives */
{
    schedule(negexp(iat),1);
    q = q+1;
    if (q == 1)
        schedule(negexp(service_time), 2);
}
```

and the departure function becomes:

```
void departure() /* a customer departs */
{
    q = q-1;
    if (q > 0)
        schedule(negexp(service_time), 2);
}
```

We haven't yet described the function `schedule`. In this particular case of the single server queue, it is pretty simple, since we have a maximum of 1 arrival and 1 departure scheduled at any time, but let's consider a more general case. Let's assume that for some system we have arrivals (type '1') scheduled at times 224, 246, 558, and 1024, and departures (type '2') scheduled at times 237, 516, and 518. Then, at least conceptually, we need an array sorted into time order as follows:

time:	224	237	246	516	518	558	1024
event:	1	2	1	2	2	1	1

If we now need to schedule a new event, say an arrival at a time 136 units in the future, and the current time is, say, 200 units, then we need to schedule an event of type '1' at a time 336, ie we need to insert the pair (336,1) in the above array:

time:	224	237	246	336	516	518	558	1024
event:	1	2	1	1	2	2	1	1

To implement this as an array is inefficient, as it is better to use a linked list, but the idea is still the same.

With an array like this, when we perform the simulation, we move the simulation clock forward until it reaches the first event in the event list, and then remove that first event from the event list, and perform the actions specified by it. We then continue on like this, stepping from event to event, according to the entries in the event list.

One point to note is that we are treating time here as a discrete quantity, rather than a continuous one. This is done since it generally makes the time keeping easier. In running the simulations, we should then choose our basic unit of time to be small enough so that this discretisation has no effect on the results. For example, if most of the time intervals that we are simulating are of the order of seconds, we should choose perhaps 1 millisecond as the basic time unit. However, if the simulation times were of the order of milliseconds, we would need a basic unit of maybe 1 microsecond.

I will provide a C program (called `ssq.c`) that puts these ideas together into a (very) basic simulation program. To compile it, you will need to link with the maths library which contains the routines for mathematical functions, eg `log(.)`. To do this, use `'cc ssq.c -lm'`. You should experiment with this program, running it for various numbers of customers. Note that the theoretical answer for the mean queue length seen by an arriving customer in this system is given by $\frac{h}{a-h}$ where h is the mean service time of a customer, and a is the mean time between arrivals of successive customers.

For your particular project, you should be able to reuse parts of this program directly — it is primarily the action blocks that need to be changed. Also note that this program assumes a random number generator `drand48(. .)`, which generates uniformly distributed real random numbers between 0 and 1. You will need to check the random number generator on the machine that you use, and may need to modify the program accordingly.

Initial Experiments:

1. On the computer that you intend to use for this project, investigate the random number generator. Use it to generate a long sequence of uniformly-distributed random numbers, and calculate the sample mean and sample variance of this sequence. Confirm that these agree with the values that you expect.
2. Generate a long sequence of negative-exponentially distributed random numbers, and calculate the sample mean and sample variance of this sequence. Confirm that these agree with the values that you expect.
3. To get a feel for the accuracy/time tradeoffs using simulation, you might begin by running my program for the following values (theoretical values are given in brackets):

service time = 1000

interarrival time =	2000 (1.00)	1500 (2.00)	1200 (5.00)
100 events			
1 000 events			
10 000 events			
100 000 events			

What conclusions can you draw?

D Everitt

```

/* program ssq.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

#define NEW(type) (type *) malloc(sizeof(type))
#define ARRIVAL 1
#define DEPARTURE 2

/* Event by event simulation of a single server queue with infinite waiting
   room */

int
    gmt,      /* absolute time */
    q,        /* number of customers in the system */
    narr,     /* number of arrivals */
    q_sum,    /* sum of queue lengths at arrival instants */
    iat,      /* mean interarrival time */
    service_time, /* mean holding time */
    total_events; /* number of events to be simulated */

long
    seed; /* seed for the random number generator */

typedef struct schedule_info *point;

typedef struct schedule_info{
    int time; /* Time that event occurs */
    int event_type; /* Type of event */
    point next; /* Pointer to next item in list */
} EVENTLIST;

point
    head,
    tail;

int act();
int negexp(int);
void arrival();
void departure();
void schedule(int, int);
void sim_init();

/*****
/
main(){

    sim_init();

    while (narr < total_events){
        switch (act()){
            case ARRIVAL:
                arrival();
                break;
            case DEPARTURE:
                departure();
                break;
            default:
                printf("error in act procedure\n");
                exit(1);
                break;
        }
    }
}

```

```

    } /* end switch */
} /* end while */
printf("The mean queue length seen by arriving customers is: %8.4f\n",
      ((float) q_sum) / narr);

} /* end main */
/*****
/
int negexp(int mean) /* returns a negexp rv with mean 'mean' */

{
return ( (int)(- log(drand48()) * mean + 0.5) );
}

/*****
/
void arrival() /* a customer arrives */

{
narr += 1; /* keep tally of number of arrivals */
q_sum += q;
schedule(negexp(iat), ARRIVAL); /* schedule the next arrival */
q += 1;
if (q == 1)
    schedule(negexp(service_time), DEPARTURE);
}

/*****
/
void departure() /* a customer departs */

{
q -= 1;
if (q > 0)
    schedule(negexp(service_time), DEPARTURE);
}

/*****
/

```

```

/*****
/
void schedule(int time_interval, int event) /* Schedules an event of type
*/
/* 'event' at time
'time_interval' in the future */

{
int
    event_time;

point
    x,
    t;

event_time = gmt + time_interval;
t = NEW(EVENTLIST);
for(x=head ; x->next->time<event_time && x->next!=tail ; x=x->next);
t->time = event_time;
t->event_type = event;
t->next = x->next;
x->next = t;
}
/*****
/
int act() /* find the next event and go to it */
{
int type;
point x;

gmt = head->next->time; /* step time forward to the next event */
type = head->next->event_type; /* Record type of this next event */
x = head->next; /* Delete event from linked list */
head->next = head->next->next;
free(x);
return type; /* return value is type of the next event */
}
/*****
/
void sim_init()
/* initialise the simulation */

{
printf("\nenter the mean interarrival time and the mean holding time\n");
scanf("%d%d", &iat, &service_time);
printf("enter the total number of customers to be simulated\n");
scanf("%d", &total_events);
printf("enter the seed\n");
scanf("%ld", &seed);
srand48(seed);

head = NEW(EVENTLIST);
tail = NEW(EVENTLIST);
head->next = tail;
tail->next = tail;

q = 0;
narr = 0;
q_sum = 0;
schedule(negexp(iat), ARRIVAL); /* schedule the first arrival */
}
/*****
/

```