CSE 597 - Performance Modeling

- "Verification is determining that a simulation program performs as intended. Validation is concerned with determining whether the conceptual model is an accurate representation of the system. When a model and its results are accepted by the manager/client as being valid, we call the model credible."
- "A mistake often made by beginning modelers is to include an excessive amount of detail."
- "Do not have more detail in a model than is necessary to address the issues of interest, subject to the proviso that the model must have enough detail to be credible."
- "The level of model detail should be consistent with the type of data that are available."
- "If the number of factors for the study is large then use a 'coarse' simulation model or an analytic model to identify the important factors before developing a detailed simulation model."


Today’s Plan: Simulation

- Some slides we skipped last time
- Discrete Event Simulation
  - The Art of Modeling (by example)
  - Generating Random Samples
Simulation

- Types of simulation
  - static vs. dynamic
    - Does time play a role?
  - deterministic vs. stochastic
    - Is there randomness in the inputs or the operation of the model?
  - continuous vs. discrete
    - Do “state changes” take place continuously, or only instantaneously at certain points in time?

Static Simulation - Monte Carlo Methods

- Typically useful when we need to estimate the value of a function too complicated to deal with mathematically
- Make repeated random trials
- Each trial is independent of the others
- Each trial simply computes some value
- Average the outcomes to estimate the result
- More trials → “more accurate” answer
**Classical Example: Integration**

- Suppose we want to compute $I = \int_{a}^{b} g(x) \, dx$ for some function $g(x)$
- **Method 1**
  - Create a bounding box
  - Throw $N$ points in box at random: $(x_i, y_i) = (a+(b-a)U_2i, hU_{2i+1})$
  - $X_i = 1$ if point is below $g(x)$, 0 otherwise
  - Estimate $I$ as $\frac{1}{N} \sum_{i=1}^{N} X_i$

**Monte Carlo Integration II**

- Sample $g(x)$ at values of $x$ chosen randomly in $[a,b]$
- $X_i = g(a + (b - a)U_i)$
- Estimate $I$ as $(b - a) \frac{1}{N} \sum_{i=1}^{N} X_i$
- $E[X] = E[g(y)] = \int_{a}^{b} g(y)f(y) \, dy = \frac{1}{b-a} \int_{a}^{b} g(y) \, dy = \frac{1}{b-a} I$
More Typical Computer Science Example

- Computer system problems for which Monte Carlo is used are often combinatorial problems

- **Example:** What is the probability that three distinct link failures will disconnect a given network topology?

Deterministic Simulations

- Probability doesn’t play a role
- Typically trace driven evaluations of design proposals
  - memory cache design
  - processor design
  - distributed file system design
  - etc.
- **Why do this?**
The Art of Modeling

LAWREN HARRIS

"Lake Superior Hill XV", c. 1925

Art and Mathematics

“The Group of Seven was an influential force in Canadian art from around 1914 into the 1940s.

... Eleven men were closely associated with the Group of Seven.”
Discrete Event Simulation

- We'll look at issues through a specific example
- I'm going to use pseudo-code as a "convenient" way to
  * explain where various issues come up
  * show by example how some simulation issues translate into coding style
- Warnings:
  * I don't claim the examples run
  * pseudo-C++ (with Slovak notation)

Multiprocessor Scheduling Policies

Question: When a job arrives, how many processors should be allocated to it (given that allocation is non-preemptable)? in order to minimize average response time?
Why Is This a Reasonable Question?

How many processors should this job be given when the system is otherwise empty?

Danger of Too Many: New Arrivals
Danger of Too Few: No Arrivals

Now What?

- In fact, our posed question is still at least too imprecise to be useful:
  When a job arrives, how many processors should be allocated to it in order to minimize average response time?
- Why? The model can't answer this general a question.
- Instead, we need to pose a question like:
  How does my new fancy scheduling policy compare wrt average response time to some (specific) existing policy?
- To do this, we need to choose a new policy and a baseline to compare against
Policies to Compare

- Let $N_k$ be the number of processors job $k$ wants at maximum, $P$ be the total number of processors in the system, and $F$ the number of processors currently free.
- Pure FCFS
  * Allocate $\min(F, N_k)$
- Diminishing
  * Allocate $\min(F, P \times N_k / \sum_{j \text{ in system}} N_j)$
- Random
  * Allocate $U(\min(1, F), F)$

The Model

- Jobs arrive into the system and are queued. Jobs are scheduled onto the processors.
- Jobs have inputs $N_k$, $T(1)$, $B_k$.
- Processors have no inputs.
- System inputs include arrival rates, $P$, and scheduling discipline.
- Processor inputs are not specified (N/A).
Parallel Speedup

- Parallel jobs typically have overheads that grow with the number of processors used.
  * E.g., a job that takes 1000 seconds on one processor may take 400 seconds on five.

\[
S(p) = \frac{T(1)}{T(p)}
\]

We'll use
\[
S(p) = \frac{(1+B_k)p}{B_k+p}
\]

Simulator Implementation

```cpp
class Simulator {
    float clock;
    FEL eventList;
    Monitor myMonitor;
    Simulator();
    int EventLoop();
    int EventHandler(Event* e);
    RNStream arvRN(seed);
    RNStream jobRN(seed);
    ArvStrm arrivals(arvRN);
    JobQueue jobQueue(jobRN);
    ProcPool procs;
    FCFSSch scheduler;
};
```

```cpp
class Job {
    RNStream& myRN;
    int maxProcs;
    float seqTime;
    float speedupParm;
    float arrivalTime;
    int numAlloced;
    Job(RNStream& r);
    int EventHandler(Event* e);
};
```

Warning: There are plenty of other classes...
**Simulator Initialization**

Simulator::Simulator() {
    Event* stopEvent;
    clock = 0.0;  // yep
    stopEvent = new Event(stoptime, this, STOP);  // “this” will register this object as the handler
    // STOP is the event type
    eventList.ScheduleEvent(stopEvent);
}

**Issue:** When do we stop the simulation?

**The Event Loop**

Int Simulator::Eventloop() {
    Event* nextEvent;
    int rcode;
    for (;;) {  // do forever loop
        nextEvent = eventList.GetNextEvent();  // returns earliest list in the FEL
        rcode = nextEvent->owner->EventHandler(nextEvent);  // invoke event handler
        if (rcode != NORMAL) break;
    }
    return rcode;
}
The Simulator’s Event Handler

```cpp
int Simulator::EventHandler(Event* e) {
    switch (e->type) {
    case STOP: myMonitor.Done(); // give monitor a chance to report final information
        return DONE;
        break;
    default: return ERROR;
        break;
    }
}
```

Arrival Process Class

```cpp
ArrvStrm::ArvStrm(RNStream& rn) {  // Initialization routine
    Event* arrivalEvent;
    myRN = rn; // just save my random number stream for future use
    arrivalEvent = new Event(0.0, this, ARRIVAL); // the 0.0 arrival time is arbitrary…
    eventList.Schedule(arrivalEvent);
}
```

```cpp
int ArvStrm::EventHandler(Event* e) {
    if (e->type != ARRIVAL) return ERROR;
    new Job(myRN); // the Job class initializer does some things…
    float timeIncrement = generate next inter-arrival time (or read from log);
    e->eventTime = clock + timeIncrement;
    eventList.Schedule(arrivalEvent); // note that we reuse this Event object over and over
    return NORMAL;
}
```
**Job Class**

```c
Job::Job(RNStream& rm) {  // Initialization routine
    maxProcs = generate random maximum processor demand;
    seqTime = generate random service time;
    speedupParm = generate random parameter "B" of speedup function approximation;
    arrivalTime = clock;  // need this to compute output performance measures
    jobQueue.InsertJob(this);    // queue for service
    scheduler.Go();
}
```

```c
int Job:EventHandler (Event* e) {
    switch (e->type) {
    case DONE: // scheduled by the job when it is allocated processors
        myMonitor.JobDone(this); // invoke monitor routine
        break;
    case ...
    }
    return NORMAL;
}
```

**Monitor Class**

```c
Monitor::Monitor() {  // Initialization routine
    totalResponseTime = 0.0;
    doneCount = 0;
}
```

```c
void Monitor::JobDone (Job* job) {
    totalResponseTime = totalResponseTime + (clock - job->arrivalTime);
    doneCount = doneCount + 1;
}
```

```c
void Monitor::Done() {
    write_to_output("Finished %d jobs at simulation time %f in
", doneCount, clock);
    write_to_output("Average response time = %f in
", totalResponseTime / doneCount);
    write_to_output("Ok, the average response time was 104.633879. I wonder what the
average processor allocation was…");
}
```
Issues / Observations

- “Gee, it took a really long time to run that simulation…”
- How do we choose distributions for the randomly chosen parameters (e.g., inter-arrival time and job compute time)?
- How do we generate random samples from the distributions we choose?
- How long do we run the simulation?
- “I compute that average service time is 5.28 under FCFS and 5.19 under my discipline. Now what?”

1. “Gee, that was slow”

- The “event loop” is simulation model independent
- All models do a lot of FEL operations
  * It can be worthwhile to implement a decent data structure
- The FEL is basically a priority queue
  * dequeue smallest timestamp event
  * enqueue arbitrary events
  * (worry about ordering of simultaneous events?)
- People have used unsorted lists, sorted lists, heaps, splay trees...
The Calendar Queue

- A data structure for FELs in simulations
  (Article by Randy Brown, Communications of the ACM, Oct. 1988.)
  (Formal analysis by Erickson, Ladner, and LaMarca,

Brown suggests dynamically adjusted number of buckets and bucket width as simulation proceeds
(Doubling/halving #buckets leads to an average of no more than 3 moves of each event in the worst case).

Bucket width should be “about average separation between events”

Empirical result: calendar queue has
enqueue+dequeue time costs that are constant in the FEL queue length (vs. O(n) for linked list and O(logn) for splay trees)
2. How Do You Choose Distributions?

- There are a few options
  - (A) Don’t use a distribution, use a trace
  - (B) Find a “classical distribution” that “matches well”
    (E.g., fit a uniform, or a truncated normal, or…)
  - (C) Use an “empirical distribution”
  - (D) Appeal to some special properties in some circumstances

(C) Empirical Distribution

- Suppose you have $N$ samples, $X_1$ to $X_{N-1}$
  - E.g., $N$ inter-arrival times
- Sort the samples from smallest to largest and call the sorted values $Y_1$ to $Y_{N-1}$
- Choose a (continuous) distribution $f(x)$ such that:
  - $f(x) = 0$ if $x \leq Y_1$ or $x \geq Y_{N-1}$
  - $f(x) = \frac{1}{((N-1) \cdot (Y_k - Y_{k-1}))}$ for $Y_{k-1} < x \leq Y_k$
- (We’re assigning $1/(N-1)$ of the total probability to lie between each two adjacent values observed.)
Example

- Suppose the samples are 2, 8, 2.5, 3.3, and 1

(D) Special Properties

- Arrival Processes

If each of the N input streams has independent, identically distributed inter-arrival times, and the N streams are independent of each other, the the output process approaches Poisson as \( N \to \infty \).

This means inter-arrival times of the output process approach an exponential distribution.

Note: the inter-arrival time distributions of the streams can be entirely different from each other.
Other Distributions: The (Dread) Maximum Entropy Argument

- Entropy originally proposed by Shannon as a measure of information/uncertainty
- Basic idea: suppose you're waiting for a message from some sender, and the message is a binary string. How much information does each bit you receive give you about the string? (Or, equivalently, how much uncertainty about the message does each bit remove?)

Max Entropy

- The answer depends on the probabilities distribution of the zeros and ones
  - As an extreme, if the distribution is P[0]=0 and P[1]=1, then each bit gives you no new information
  - Intuitively, the distribution P[0]=0.5 and P[1]=0.5 is the one for which each new bit is “most valuable”
- Shannon (in 1948) proposed a measure of this, called entropy, for arbitrary distributions
  \[ S_n(p) = \sum_{i=1}^{n} p_i \ln p_i \]
Max Entropy

\[ S_n(\bar{p}) = -\sum_{i=1}^{n} p_i \ln p_i \]

\[ S_n((0,1)) = -(0 \ln 0 + 1 \ln 1) = 0 \]

\[ S_n((0.5,0.5)) = -(0.5 \ln 0.5 + 0.5 \ln 0.5) = 0.69315 \]

Max Entropy

- Given known values \(a_1\ldots a_m\) for a set of \(m<n-1\) expected values
  \[ a_r = \sum_{i=1}^{n} g_r(x_i)p_i \]

The known values do not determine a specific distribution (i.e., values \(p_1\ldots p_n\)). There are an infinite number of consistent solutions.

- Jayne’s principle of maximum entropy (1957)
  Out of all the distributions consistent with the known values \(a_1\ldots a_m\), choose the one that has maximum entropy
Maximum Entropy Distributions

- If the distribution is over \([a,b]\) and there are no other constraints, chose the Uniform over \([a,b]\)
- If the domain is \([a,b]\) and the mean \(\mu\) is known, chose the truncated exponential
  \[ f(x) = ce^{-kx} \]
  \[ c \int_a^b e^{-kx} \, dx = 1 \quad \text{and} \quad c \int_a^b xe^{-kx} \, dx = \mu \]

Maximum Entropy Distributions

- If the distribution is over \([0,\infty)\) and there are no other constraints, there is no maximum entropy distribution
- If the distribution is over \([0,\infty)\) and the mean \(\mu\) is known, chose the exponential
  \[ f(x) = \frac{1}{\mu} e^{-x/\mu} \]
Maximum Entropy Distributions

- If the distribution is over \((-\infty, \infty)\) and either no other constraint is given or only the mean is known, there is no maximum entropy distribution.
- If the distribution is over \((-\infty, \infty)\) and the mean \(\mu\) and variance \(\sigma^2\) are known, the maximum entropy distribution is the normal (the “bell shaped distribution”).

3. How Do We Generate Samples?

- Let’s start with what seems like the easiest case, the Uniform distribution over \([0,1]\).
- First of all, we don’t want real randomness (because we want repeatability in our experiments).
- We’ll look at the most successful pseudo-random number generator, the congruential generator.
The Congruential Generator for U(0,1)

- (Park and Miller, Random Number Generators: Good Ones are Hard to Find, Communications of the ACM, October 1988.)
- The basic idea is to generate a sequence of integers in the interval \([1,m-1]\) that have random characteristics (and to convert this to (0,1) by dividing by \(m\)) using
  \[ z_{n+1} = az_n \mod m \]
  for constants \(a\) and \(m\)
- It turns out this is possible, assuming you’re careful how you pick \(a\) and \(m\)

The Congruential Generator

- To be most useful, the generator should be full period, that is, generate all \(m-2\) integers possible before any repeats
- Full period is assured if
  * \(m\) is a (large) prime
  * \(a\) is a primitive root of \(m\) (\(a^n \mod m \neq 1\), \(n=1,2,\ldots,m-2\))
- This doesn’t assure randomness, though. (More on that later)
The Congruential Generator

- The authors suggest the following specific choices:
  - \( m = 2^{32} - 1 \)
  - \( a = 16807 \)
- They then point out that the straightforward implementation can fail (because of overflow on 32-bit arithmetic)
- An only slightly more complicated implementation is given
- (They also survey the then existing implementations and discussions of the generator in texts.)

Testing the Generator

- Want uniform and independent samples
- Uniform means what, exactly?
  - If we generate \( N \) samples and sort them into \( d \) bins, about fraction \( 1/d \) of them should appear in each bin
  - Of course, if \( N \) is small, it’s not too disconcerting that the fractions don’t work out quite right
    - \( d=2, N=10, C_1=3, C_2=7 \)
    - \( d=2, N=1000, C_1=300, C_2=700 \)
- We’ll get back to this notion (testing the distribution)
- We’ll start with what is really an independence test
Testing the Generator

- Independence: remember the strict definition
  \[ f(U_{i+1} | U_i) = f(U_{i+1}) \]
- Informal technique: plot the (x,y) points defined by (U_i, U_{i+1}) and eyeball the results
- What should we see?
  - Points should be uniformly scattered in the unit square
- Remember, the main point of this is testing for independence

Example Using Excel’s `rand()`
What Does This Tell Us?

- Results are subjective
- Unix rand(): \( f(z) = (129z + 907633385) \mod 2^{32} \)
  - “The difference is that rand(3) produces a much less random sequence - in fact, the low dozen bits generated by rand go through a cyclic pattern.”
- Note that the scatter plot can be used (to test independence) with any distribution. What you expect to see, though, is complicated.

The Normal Distribution
Hypothesis Testing

- We're switching now to testing the distribution, ignoring questions of independence
- 1. Hypothesize that the data comes from some specific distribution
- 2. Examine the data and (more or less) compute the probability that the hypothesized distribution would have produced that result
- 3. If that probability is below some acceptance threshold, reject the hypothesized distribution at that "acceptance level"

The Chi-Squared Test (for Uniformity)

- Partition the unit interval into k subintervals of equal length: (0,1/k), (1/k,2/k), …, ((k-1)/k,1)
- Of our N total observations, on average N/k should fall in each subinterval
- Let n_j be the number of observations falling in the j^th subinterval
- Compute the “Chi-squared statistic”
\[ \chi^2 = \frac{k}{N} \sum_{j=1}^{k} \left( n_j - \frac{N}{k} \right) \]
Chi-Squared Test

It is known that if the hypothesized distribution is correct, $\chi^2$ approaches a known distribution, the “Chi-squared distribution with $k-1$ degrees of freedom”

Like all the hypothesis tests we'll see, this one at most lets us reject the hypothesis. It says nothing to indicate that the hypothesis is true. What's the difference?

Extending the Chi-Squared

- Group the samples into $d$-dimensional vectors: $(U_1, U_2, \ldots, U_d), (U_{d+1}, \ldots, U_{2d}), \ldots$
- Now do the same thing as before, except we're placing points into a $d$-dimensional unit volume, and our buckets are defined by dividing each dimension into $k$ equal subintervals

$$
\chi^2(d) = \frac{k^d}{N} \sum_{j_1=1}^{k} \sum_{j_2=1}^{k} \cdots \sum_{j_d=1}^{k} \left( \frac{n_{j_1j_2\cdots j_d}N}{k^d} - \frac{N}{k^d} \right)
$$

$k^{d-1}$ df
Generating Samples from an Arbitrary Distribution

- Let’s presume we have a good U(0,1) generator
- We can use it to generate samples from an arbitrary distribution
- First the general discrete case, then the general continuous case, then one special case

Generating Discrete Samples

- Suppose we have the discrete distribution
  \( p_0 = 0.2, p_1 = 0.3, p_2 = 0.5, \) and \( p_3 = 0.1 \)
- To generate samples from it

\[
\begin{align*}
\text{Generate } U_i &= 0.63251 \\
\text{j=0;} \\
\text{while (CDF[j]<sample)} \\
\text{j++;}
\end{align*}
\]

CDF

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

return j;
Inverse Transformation Method

\[ X_i = F^{-1}(U_i) \]

Generating Samples from Continuous Distributions

Nothing changes!

\[ X_i = F^{-1}(U_i) \]
Generating Exponentials

General Inverse Transformation Method:
\[ U_i = F(X_i) \rightarrow F^{-1}(U_i) = X_i \]

For the exponential:
\[ U_i = 1 - e^{-X_i} \]
\[ \ln(U_i - 1) = -X_i / \mu \]
\[ X_i = -\mu \ln U_i \]

Verifying This Works

• What fraction of (an infinite number of) samples should fall in the range \([a,b]\), 0 < a < b?

\[ \int_{a}^{b} \frac{1}{\mu} e^{-x/\mu} \, dx = \left[ 1 - e^{-x/\mu} \right]_{a}^{b} = F(b) - F(a) \]
One Last Thing

- To apply this technique you must be able to determine the function $F^{-1}$.
- One relatively common case where that’s not possible is the normal distribution (the “bell shaped curve”)
- Approaches for generating:
  - use tables of values precomputed at particular points (and interpolate in between)
  - use a special technique (the “polar method”) that involves square roots of log functions of $U(0,1)$ samples