5-2008

Resetting a Degrading Single Server Queue

Frank Volny iv
Clemson University, fvolny@clemson.edu

Resetting a Degrading Single Server Queue

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
Frank Volny IV
May 2008

Accepted by:
Dr. Brian Dean, Committee Chair
Dr. Wayne Goddard
Dr. D. E. Stevenson

Abstract

Real life servers do not have IID service times - they slow down over time. This is usually dealt with by resetting or replacing the server. Often, however, it is not known how to determine when the best time to reset the server is. Resetting may be a costly process. We make an assumption that only some arrivals (we do not know which) are harmful. Then, we proceed to find an optimal policy for resetting a server with a given service parameterization by observing the service paremeter directly, service times, or waiting times (with and without arrival times) when the harmful arrivals are IID.

Next, we develop some methods for dealing with harmful arrivals that are not IID. Short term memory is used to cope with rapidly changing situations and is found to perform better when each observation contains less information.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Many results have been established for single server queues. Most of the time, however, these results are restricted to queues with IID service times. Often, this assumption is appropriate, but in many cases, one might find it more realistic to model fatigue, wear-and-tear, and outside influences in the service distribution - the server slows down. Indeed, many real world queues behave in this fashion, and one deals with it by replacing, rebooting, or resetting the server.

In this paper, we examine policies for resetting a server in such a queue by using Markov Decision Theory. We consider a single server whose service times stochastically increase over time as a result of processing harmful jobs. An agent, who is overlooking the system, must determine when to perform the resets according to some policy. We provide the agent with one of five levels of information. In descending order, they are: perfect information (knowledge of the service parameter), service times, interarrival times, waiting times, and no information. No information provides us only with the number of arrivals processed since the last reset. Each level also contains all the information contained in the levels below it. So, for example, knowledge of interarrival times includes knowledge of waiting times. Depending on how much it costs to provide each level of information, we may find that beyond a certain level, extra information provides no extra benefit.

1.2 Model Specifics

Throughout, we use standard queuing theory notation (see any of [1], [2], [8], or [10]). Let $\{S_n\}$ be a sequence of service times, where S_n is the n^{th} service time, since the last reset. Each S_n is distributed according to F_{θ_n}, $\theta_n \in \Theta \subseteq \mathbb{R}^+$ where $\theta_1 < \theta_2 \implies P\{S_1 < x\} > P\{S_2 < x\}$ for all $x > 0$. One might think of θ_n as taking values $0, 1, 2, \ldots$ and larger values of θ_n causing (probabilistically) larger service times. By resetting the server, θ_n is set equal to some fixed $\theta^- \in \Theta$. If the server is not reset after the n^{th} departure, then the $(n+1)^{th}$ service is distributed according to $F_{\theta_{n+1}}$. For a fixed[1] $\Delta\theta > 0$,

$$\theta_{n+1} = \begin{cases} \theta_n, & \text{when } (n+1)^{th} \text{ arrival is not harmful,} \\ \theta_n + \Delta\theta, & \text{when } (n+1)^{th} \text{ arrival is harmful.} \end{cases}$$

Between resets, $\{\theta_n\}$ forms a monotonically nondecreasing sequence, so $\{S_n\}$ forms a stochastically nondecreasing sequence of service times as well.

Further, if we denote $\{A_n\}$ as an IID sequence of interarrival times since the last reset, also independent of the service times, one can find the corresponding waiting times, $\{W_n\}$, with Lindley's equation:

$$W_1 = 0, \text{ and } W_{n+1} = (W_n + S_n - A_{n+1})^+ .$$

One can show that when $\theta_n < \theta_{n+1}$, we have $E[W_{n+1}] \geq E[W_n]$. So even though the service times are most relevant to understanding where the service parameter is, we can use Lindley's equation to glean some information about θ when the service times are not observed.

Harmful jobs are to be embedded into the arrival process, each selected with probability p. We will consider both the independent selection and correlated selection of harmful jobs. It is our hope that with enough correlation, or burstiness, one might find more information is useful. Except for the perfectly observable case, the knowledge of which jobs are harmful is not given to the agent. We assume that whenever a reset occurs, everyone currently in the line disappears. This is equivalent to subtracting out the contributing waiting time to as of yet unserviced customers. Depending on the policy in use (all but interarrival or waiting times provided), one can still reconstruct the original scenario.

[1] This is not required. It only simplifies the description.

With each service parameter value, the traffic intensity,

$$\rho_{\theta_n} = \frac{E_{\theta_n}[S_n]}{E[A_n]},$$

can be computed. From ρ, we can then compute an expected earnings per time unit the server can achieve for each parameter value. Under the assumption that it costs a certain amount just to keep the server running, we expect that beyond a certain parameter value, the queue begins to, not only lose potential profit, but lose more in expenses than it gains in revenue. Finally, we assume that resets also have associated costs - for if this was not the case, an optimal policy would be to simply reset with every arrival.

Harmful arrivals incur their damage the instant they begin service. Therefore their own service times follow the updated service parameter value. Alternate service distributions for harmful jobs may be a practical addition to the model (indeed, we included it in our computer code), but as we will see later, one of the probability mass functions we compare will not handle such a situation very well. Therefore alternate service times for harmful jobs will not be considered here.

We may simply tell the agent to perform a reset once it believes the actual parameter value moves beyond some threshold. This, however, provides the agent with a suboptimal policy. For if the agent finds any non-zero probability of being beyond this threshold, it will perform a reset. Depending on which service distribution is used, this may *always* be the case. If, on the other hand, the agent weights its belief in the occupancy of various service parameter states with how much the queue is expected to earn based on either action (to reset or not to reset), an optimal policy can be realized. Here, we define an optimal policy as one that acts to maximize expected earnings.

Finally, throughout this discussion, we consider fixed discrete arrival processes with a discrete interarrival distribution and fixed (except one parameter) discrete service distributions. These distributions can be extended to the continuous case with no major changes to the method.

1.3 Applications

Applications for such a model are endless as most of the servers in the real world act in this fashion. Human servers are the most obvious first example. Human servers get tired and work more slowly (one might consider all arrivals harmful in this regard). They often have a constant cost

per hour (i.e. minimum wage) no matter how quickly or slowly they are processing customers or whether or not they are idle. As they perform their services at a slower rate, customer turnaround slows and less revenue is received. The manager must determine when to put the server on break.

Computers and computer networks also provide a large class of examples. Computer networks may get clogged with worm activity, and such an increasing traffic intensity might trigger a virus sweep throughout a network. Computer programs may benefit from restarting themselves if memory leaks abound. In managed environments where memory leaks are less common, generational garbage collectors may still be able to use such a scheme when deciding to do a full collection.

Finally, although only discrete arrival processes are considered, we would like to provide an example of a continuous one. Imagine a biological cell that exists in some form of fluid flow. The flow contains a continuously varying concentration of some harmful chemical. When should the cell repair itself? Solving this problem involves a slight generalization of one of the main tools used here: Markov Decision Processes

1.4 Related Work

Queuing theory traditionally involves IID arrival processes and IID service processes. Depending on how general the arrival or service distributions are made to be, the analysis of these queuing models can become very intractable very quickly. Some research has been done on time varying service or arrival distributions, but it is on describing systems and not on maintaining them.

For example, Newell [7] and Rolski [9] allow a time-increasing arrival rate to find queue distributions and mean stationary queue size and delay. Green [4] makes the arrival distribution sinusoidal and tries to find an approximation for a stationary distribution. Gelenbe [5], Duda [3], and Yechiali [12] allow both the arrival and service distributions to vary in a very specific way and try to find steady state properties. Perhaps the most related paper is by Neuts [6] where both the arrival and service distributions change with time and the phase state, queue length, number served, and virtual waiting time are found. Neuts's phase state might correspond to the value of θ in our framework. However, no computations are done for consideration of any kind of action.

Indeed some of these other papers refer to periodically changing service distributions. Perhaps if it can be shown that under the policies outlined here, the queue can reach any kind of stationarity, some of the related work might apply to our queuing model.

1.5 Direction of this Research

For the purposes of comparison, we considered three probability measures. The first is a rigorously derived measure that is correct under the assumption of an IID embedding process of the harmful jobs. But since the embedding process will not always be IID, as we would like to show that more information might be useful in this case, we would like another option. Two other probability measures were be defined in the hope that they would perform well in varying environments.

Once we have a probability measure on the service parameter, the Bayesian framework applies. To solve this problem, it seems that all we need to do is integrate a loss function against our probability measure on the parameter value and reset when this surpasses some threshold - reset when

$$E[\ L(\theta) \mid \text{some observations}\ ] > \alpha^*.$$

The task of finding a loss function L and corresponding threshold α^* can be combined when we consider using a Markov Decision Process (MDP). After each service, we have a choice of resetting the queue ("reset") or not ("stay"). After a reset, the service parameter returns to θ^- with probability 1, but when staying, the parameter advances with probability p and remains with probability $q = 1 - p$ (figure 1.1). This is a natural, albeit simple, task for an MDP. When in each state (θ_n), an MDP tells us exactly which action is the optimal one. But we will not typically know θ_n. Partially Observable Markov Decision Processes (POMDP) allow us to combine a probability measure with an observable MDP to work under uncertainty.

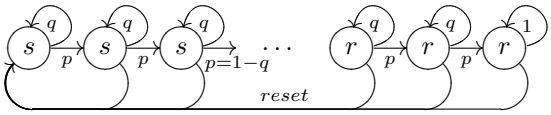

Figure 1.1: State space of the service parameter and available actions

Short term memory will be employed in the second and third probability measures in order to mitigate non-IID harmful arrivals. But how much forgetfulness should we use? After comparing the probability measures, we ran another simulation comparing many possible values of the short term memory parameter to see, at least for one example, what it should be set to.

1.6 Summary of Results

We considered 11 policies (the non-rigorous measures only handle three of the five levels of information) for both IID and bursty embedded arrival processes. A particular example was simulated and the results were exactly what one would expect. More information, does in fact, earn more of a profit.

The rigorously defined measure outperformed the others for the IID embedding process. But when the same policies were exposed to a bursty embedded arrival process, profits dropped off quicker (with the loss of information) than the IID case. Here, one of the other measures actually earned more than the rigorously defined measure.

Chapter 2

Probability Mass Functions

As described in the introduction, we proceeded to define 3 competing mass functions. The first is meant to deal with a harmful embedded arrival process that is distributed according to IID Bernoulli(p) random variables. That is, each arrival is considered harmful with probability p, independent of all others. The second is meant to better detect bursty arrivals where the embedded arrival process correlates the selection of jobs as harmful. The third will be a slight modification of the first in an attempt to get the best of both worlds.

For notational convenience, let I_n be the n^{th} observation. It may consist of the parameter value (as in perfect information), the n^{th} service time, interarrival time, waiting time, or simply n (as in the no information case). Let $\mathcal{G}_n = \sigma(I_n, I_{n-1}, \ldots, I_1)$ be all the information observed since the last reset.

2.1 The IID Probability Mass Function

The perfect information case is easily seen to be:

$$P\{\theta_n = \theta|\mathcal{G}_n\} = P\{\theta_n = \theta|\theta_n\} = \delta_{\{\theta_n\}}(\theta),$$

where δ is the Dirac delta. For the other 4 information levels, we begin with

$$\begin{aligned} P\{\theta_n = \theta | \mathcal{G}_n\} &= \frac{P\{\theta_n = \theta, I_n = i | \mathcal{G}_{n-1}\}}{P\{I_n = i | \mathcal{G}_{n-1}\}} \\ &= \frac{\sum_{\theta' \leq \theta} P\{\theta_n = \theta, I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}}{\sum_{\theta'} P\{I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}}, \end{aligned}$$

where we can compute $P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}$ recursively. In fact, it can be computed efficiently using dynamic programming, or in the case of our code, memoizing. For the case where service times are provided, we can compute the denominator. In all cases, the numerator is arrived at by the following.

$$P\{\theta_n = \theta, I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} = P\{I_n = i | \theta_n = \theta, \mathcal{G}_{n-1}\} P\{\theta_n = \theta | \theta_{n-1} = \theta'\}$$

Note that if we allowed alternate service distributions for harmful jobs, the right hand side would have $P\{I_n = I | \theta_n = \theta, \theta_{n-1} = \theta', \mathcal{G}_{n-1}\}$ as its first term, instead of dropping the θ_{n-1}. When service times are provided, this can be directly computed. For less information, we have two cases to consider. Lindley's equation tells us that $W_{n+1} = (W_n + S_n - A_{n+1})^+$. Because the waiting times cannot be negative, our cases are when $W_{n+1} > 0$ and when $W_{n+1} = 0$. For $W_{n+1} > 0$, $S_n = W_{n+1} - W_n + A_{n+1}$ and for the other case, $S_n \leq A_{n+1} - W_n$. Thus,

$$P\{I_n = i | \theta_n = \theta, \mathcal{G}_{n-1}\} = \begin{cases} P\{S_n = W_{n+1} - W_n + A_{n+1} | \theta_n = \theta, \mathcal{G}_{n-1}\}, & W_{n+1} > 0 \\ P\{S_n \leq W_{n+1} - W_n + A_{n+1} | \theta_n = \theta, \mathcal{G}_{n-1}\}, & W_{n+1} = 0 \end{cases}$$

When interarrival times are provided we can compute these. When all we have are the waiting times, we need to condition on the interarrival times as in

$$P\{W_n = w | \theta_n = \theta, \mathcal{G}_{n-1}\} = \begin{cases} \sum_{a>0} P\{S_n = W_{n+1} - W_n + a | \theta_n = \theta, \mathcal{G}_{n-1}\} P\{A_{n+1} = a\}, & W_{n+1} > 0 \\ \sum_{a>0} P\{S_n \leq W_{n+1} - W_n + a | \theta_n = \theta, \mathcal{G}_{n-1}\} P\{A_{n+1} = a\}, & W_{n+1} = 0, \end{cases}$$

where W_{n+1} denotes the $(n+1)^{th}$ waiting time. I_{n+1} would have worked, but in this case, we can be more specific since, at this level of information, I_n is a scalar. These both equal

$$E_A \left[S(W_{n+1} - W_n + A) \mid \theta_n = \theta, \mathcal{G}_{n-1} \right],$$

where $S(\cdot)$ is the probability mass function (pmf) of the service distribution when $W_{n+1} > 0$ and the cumulative mass function when $W_{n+1} = 0$. We mention this last step only because most of the conditionings of our simulation took this form.

We have to be careful here, this recursive equation actually has two base cases. The easier case is when $\mathcal{G}_0$ is provided. Since there have been no harmful arrivals, we know θ_0 is exactly where we reset it to, θ^-. The other base case is when there has only been one arrival and $\mathcal{G}_1$ is provided. The first two levels, when the parameter is known and when the service time is given, are calculated as before. The next two levels are similar to the bottom level, no information. This is due to the fact that the first arrival never has to wait (or had the waiting time adjusted to zero).

The case of no information is gotten with (recall, the probability of a harmful arrival, Bernoulli(p), is still known) an n-fold convolution, or simply a Binomial(n, p).

2.2 Short Term Memory / Local Linearization

Here, we would like to have the most recent observations count more than the less recent. We would also like a measure that knows relatively little about the embedded arrival process, for such a measure would cope with bursty and unplanned for situations better than if more favorable conditions were wrongfully assumed. The only assumptions we make are that small changes in θ produce small changes in the likelihood of an observation and that the service pmf is unimodal[1]. Neither assumption is strictly required since this *method* is far from rigorously derived.

To begin with, we propose a local linearization of the trajectory θ_n took just prior to the present. Such a linearization would be less and less helpful as we look farther back in time, so a short term memory component is crucial here. Set $\theta_j = max\{\theta_n - (n-j)\phi\Delta\theta,\ \theta^-\}$. Certainly, we cannot apply this pmf to the perfect information case since the linearization will fail to hold most of the time, resulting in zero probabilities. So we may think of $0 \leq \phi \leq 1$ as an estimate of the

[1]Both simplify maximization of the likelihood function.

probability that an arrival is harmful. Because of the importance of this estimator, later we use the MLE of ϕ, we cannot use this method for the no information case either.

Normally, one would compute the probability of the set of observations with

$$P\{\theta_j, j = 1, \ldots, n \mid I_j = i_j, j = 1, \ldots, n\} = \frac{1}{\alpha} \prod_{j=1}^{n} P_{\theta_j}\{I_j = i_j\},$$

where α is the normalizing constant from applying Bayes Theorem with a uniform prior.

Next, we apply some forgetfulness. Choosing our favorite constant $0 < \gamma \leq 1$ (ours is .8), we simply exponentiate the pmfs with this constant as in

$$P\{\ \underline{\theta} \mid \underline{I}\ \} = \frac{1}{\alpha} \prod_{j=1}^{n} P_{\theta_j}\{I_j = i_j\}^{(\gamma^{(n-j)})}. \tag{2.1}$$

Alpha is now a different, but still normalizing, constant. This has the effect of drawing the older observations toward a uniform distribution. In other words, we care about the older observations less with each new observation. But if any of the observations are impossible or highly improbable (as generated by the local linearization), then the overall probability *is still going to be close to zero.*

To see that this is at least a somewhat reasonable way to proceed, let us examine the likelihood function followed by the log-likelihood.

$$\mathcal{L}(\underline{I}, \theta_n, \phi) = \prod_{j=1}^{n} P_{\theta_j}\{I_j = i_j\}^{(\gamma^{(n-j)})}$$

$$l(\underline{I}, \theta_n, \phi) = \sum_{j=1}^{n} \gamma^{(n-j)} \ln\ P_{\theta_j}\{I_j = i_j\}$$

The log-likelihood function has the form of an exponential moving average. If $\gamma = \frac{1}{2}$, then the most recent observation will influence the log-likelihood approximately as much as all the other observations combined. So the same weighting will apply to the likelihood function, just on some kind of logarithmic scale. This seems very reasonable, but log-likelihoods do not have much to do with finding probabilities. So, one might be tempted to remove the logarithm and keep it as a mixture of distributions as below. But we argue that doing so is not practical.

$$P\{I_j = i_j, j = 1, \ldots, n | \theta_j, j = 1, \ldots, n\} = \frac{1-\gamma}{1-\gamma^n} \sum_{j=1}^{n} \gamma^{(n-j)}\ P_{\theta_j}\{I_j = i_j\}$$

This form has the advantage of making the same intuitive sense to us as the log-likelihood function did. But this form does have the huge disadvantage that impossible (or highly unlikely) θ_i's created by the local linearization will barely move this average, especially when γ is large or when the unlikely observations happened a long time ago. Mixtures are for one observation probabilistically following one of many distributions. Here we wish to simultaneously consider many observations.

Returning to (2.1), the likelihood part can be rewritten as

$$\begin{aligned}
& \mathcal{L}(I_j = i_j, \theta_n, \phi, j = 1, \ldots, n) \\
= & \prod_{j=1}^{n} P_{\theta_j}\{I_j = i_j\}^{(\gamma^{(n-j)})} \\
= & P_{\theta_n}\{I_n = i_n\} \left[\prod_{j=1}^{n-1} P_{\theta_j}\{I_j = i_j\}^{(\gamma^{(n-1-j)})} \right]^{\gamma} \\
= & P_{\theta_n}\{I_n = i_n\} \left[\mathcal{L}(I_j = i_j, max\{\theta_n - \phi\Delta\theta,\ \theta^-\}, \phi, j = 1, \ldots, n-1) \right]^{\gamma},
\end{aligned}$$

giving us a recursive form that makes it easier to see how to define our third probability measure. We will comment more on this pmf later when its relationship to the model is discussed.

2.3 A Hybrid PMF

Earlier, for the IID measure, we started with

$$\begin{aligned}
P\{\theta_n = \theta | \mathcal{G}_n\} &= \frac{P\{\theta_n = \theta, I_n = i | \mathcal{G}_{n-1}\}}{P\{I_n = i | \mathcal{G}_{n-1}\}} \\
&= \frac{\sum_{\theta' \leq \theta} P\{\theta_n = \theta, I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}}{\sum_{\theta'} P\{I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}}.
\end{aligned}$$

Built into the equation further in its derivation was the term $P\{\theta_n = \theta | \theta_{n-1} = \theta'\}$. Under the IID assumption, we simply take this to be $1 - p$ when $\theta = \theta'$ and p when $\theta = \theta' + \Delta\theta$. Furthermore, this is calculated independent of all others. Just like the no information case, this probability builds in the fact that bursty arrivals usually do not happen, or do happen with probability p^r where r is the length of the burst. Despite the presence of the extra information, this derivation still has some bias as to where it thinks theta should be.

So, just like we did for the short term memory equation, we would like to make all pasts

more likely. Simply rewrite the above equation as

$$\begin{aligned} P\{\theta_n = \theta | \mathcal{G}_n\} &= \frac{P\{\theta_n = \theta, I_n = i | \mathcal{G}_{n-1}\}}{P\{I_n = i | \mathcal{G}_{n-1}\}} \\ &= \frac{\sum_{\theta' \leq \theta} P\{\theta_n = \theta, I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}^{\gamma}}{\sum_{\theta'} P\{I_n = i | \theta_{n-1} = \theta', \mathcal{G}_{n-1}\} P\{\theta_{n-1} = \theta' | \mathcal{G}_{n-1}\}^{\gamma}}, \end{aligned}$$

by exponentiating the past with γ. For $\gamma < 1$, this differs from what was derived earlier. As such, we call it another *method*. But in light of the reasoning provided for the short-term memory pmf, this seems like a reasonable addition. Unfortunately, this heuristic reasoning gives little insight as to what we should set γ to.

Chapter 3

Optimal Policies

Given a sequence of observations, a probability measure, and a set of costs to set up a Partially Observable Markov Decision Process, we can then make an optimal decision at each step of the way. In this section, we consider how this is done in more detail. Also, one of the probability measures provide the model with more information than just probabilities, so the model's relationship with the POMDP is discussed as well.

3.1 Markov Decision Processes

As mentioned earlier, simply telling the agent to reset when the parameter reaches some threshold will produce a suboptimal policy. Various costs must be factored into the agent's decision. MDPs provide us with a very nice way of doing this.

Borrowing notation from [11], when we are at state θ', the probability of transitioning to state θ after executing action $a \in$\{stay, reset\} is $T(\theta', a, \theta)$. $R(\theta)$ is the amount we earn or lose by occupying state θ (during one service) and $U^{\pi}(\theta)$, our expected utility of state θ, is how much we expect to make in the future according to policy π after transitioning to state θ. So

$$\pi^*(\theta') = argmax_a \sum_{\theta \geq \theta'} T(\theta', a, \theta) U^{\pi^*}(\theta)$$

is our optimal policy, where,

$$U^{\pi^*}(\theta') = R(\theta') + \gamma \; max_a \sum_{\theta \geq \theta'} T(\theta', a, \theta) U^{\pi^*}(\theta). \tag{3.1}$$

Our choice of $0 < \gamma \leq 1$ is important in that it affects how far we see into the future in determining how much we expect to make. If the situation permits a positive profit, then $\gamma = 1$ will result in an infinite expected earnings. If γ is too small, we may actually defer a reset because its a better decision in the short term. For our simulation, we used $\gamma = .99$. A greedy, but suboptimal, policy can be arrived at using this exact reasoning. Do not reset as long as we expect to make more money than the cost of a reset. This short-sighted policy ignores just how much faster we can turn a profit after a reset.

Solving for (3.1) requires solving a system of $n = \#$(parameter values) non-linear equations of the same form as (3.1) and n unknowns. These are known as the Bellman Equations. Making a guess at the solution and iteratively updating the values forms a contraction. This Value Iteration, as it is called in [11] is how we solved them in our code.

Depending on the model we are using (and this will be discussed shortly), we may or may not know $T(\theta_{n-1}, stay, \theta_n)$ *a priori*. But when it is known, $T(\theta_{n-1}, stay, \theta_{n-1} + \Delta\theta)$ is simply the probability the next arrival is harmful.

MDPs tells us exactly what to do when we know which state we are in, but this only applies to the perfect information case (1.1 is labeled with "s" and "r" for stay and reset with each state). For the other information levels, we have a tool that allows us to work with MDPs in an uncertain environment.

3.2 Partially Observable Markov Decision Processes

The only difference in the partially observable case is that we do not know exactly where we are. Solving a POMDP in general is PSPACE-hard [11], but since our scenario is so extremely simple - a linear state space with only two available actions, we can do much better. Our policy becomes:

$$\pi^*(\mathcal{G}_n) = argmax_a \sum_{\theta'} P\{\theta_{n-1} = \theta' | \mathcal{G}_n\} \sum_{\theta \geq \theta'} T(\theta', a, \theta) U^{\pi^*}(\theta).$$

Finally, calculating $\{U^{\pi^*}(\theta)\}$ is deterministic and described above.

3.3 Differences in Models

For the IID and hybrid mass functions, everything works as expected. The pmfs give a probability vector on the state space for θ_n and the POMDP tells us whether to stay or reset. But both of these distributions know the embedded arrival process. They assume it is a sequence of IID Bernoulli(p) random variables. So, in solving the POMDP, $T(\theta', stay, \theta' + \Delta\theta)$ is known to be p. Without this information, as in the local linearization pmf, we must find it for ourselves. We need an estimator for p.

Returning to our likelihood function, $\mathcal{L}(\underline{I}, \theta_n, \phi)$, ϕ gives us an estimate of how fast θ has been advancing recently on average. If $X \sim$ Bernoulli(p), then $E[X] = p$ is how fast θ is advancing on average. Let us use $\hat{p} =$ the MLE of ϕ.

$$\hat{p} = argmax_{\phi} \sum_{\theta} \mathcal{L}(\underline{I}, \theta, \phi) \tag{3.2}$$

Now, we can use (3.2) to solve the Bellman Equations (3.1)[1][2].

This approximation of p does cause a few problems. The easiest to see is when only the first arrival has been processed. Here we would need $\phi = \theta_1 - \theta^-$. But when ϕ is sufficiently close to zero, that tells the Bellman Equations that staying means we are going to be in that state for a while. If there is positive mass on any state but θ^-, the POMDP realizes that if we reset now, we will stay in θ^- for a long time. So for ϕ sufficiently small, the POMDP chooses to reset, even if there is only been one arrival and $\theta_1 \leq \theta^- + \Delta\theta$. A quick way to bound phi away from zero for the first arrival, at least in the computer code, was to set (again, only for the first arrival)

$$\mathcal{L}(I_1, \theta_1, \phi) = P_{\theta_1}\{I_1\} P_{(\theta^- + \phi\Delta\theta)}\{I_1\}.$$

This gave a nice likelihood landscape, as in Figure 3.1, and considered both parameters θ_1 and ϕ equally.

But even for much larger n, this method suffered from the same problem. If ϕ dropped sufficiently low, the POMDP would decide that if it did a reset, θ would stay at θ^- for a while. So it would always do a reset in this case. Some possible fixes might be to set $\hat{p}$ to some kind of moving

[1] Returning to the subject of alternate harmful distributions, one might consider using a mixture of distributions to handle this case. p tells us the probability the service followed the harmful distribution.

[2] There is no reason to allow only one alternate harmful distribution. Jobs may be harmful for different reasons, or some jobs may cause more damage than others (they cause θ to advance by many $\Delta\theta$'s, for instance).

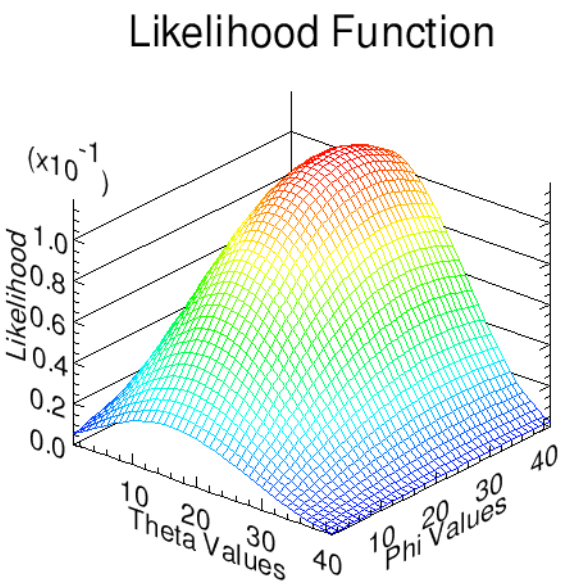

Figure 3.1: Likelihood function with one service time provided

average of the MLEs of ϕ. Or, perhaps, we need to keep more long term data to produce a long term average for $\hat{p}$. This is basically what the IID measures do. On the other hand, we want the POMDP to be more likely to reset when ϕ gets large, as it indicates that something is changing. This already happens. The undesirable part is when ϕ is too small. So the POMDP likes to stay when ϕ is in some kind of "sweet spot". One possible solution is to simply bound ϕ from below by some kind of long-term average. We simply bounded $\hat{p}$ from below by $\frac{p}{2}$.

Another difference is that pdfs and cdfs of service times are always evaluated with one of the parameter values from the parameter space $\{\theta_i = \theta^- + i\Delta\theta\}$ in the IID measures. We relax this requirement in the local linearization scheme since the linearization produces almost all of the θ_i's outside this discrete parameter space. We do not have any notion of which recent jobs were harmful, only how many were harmful. With a $\phi = .3$, say, we would estimate that every third, or so, arrival is harmful, but we do not know which arrivals did the harm. As a result, it is just not practical to model alternate service distributions for harmful arrivals. This is why we prefer service distributions whose output changes relatively little with small perturbations in θ. Also, a multimodal service distribution would create some kind of a lattice in the likelihood landscape where maximization would be difficult.

Finally, all three measures were written in recursive forms. The first and third can utilize dynamic programming approaches for efficient computation. But the local-linearization measure has a continuous parameter, namely $0 \leq \phi \leq 1$. Discretizing ϕ will allow for dynamic programming, but memory requirements may begin to grow pretty rapidly at that point. For our simulation, we discretized ϕ into only 10 points. The local linearization is only going to help us for the most recent few observations, and beyond that, we are reducing its effect anyways.

Chapter 4

Results from Simulation

The entire model (and 11 policies) described above was implemented in the form of a simulation in LISP. The entire project took about 1000 lines of code (1 KLOC) and stored data into a database for later analysis. One specific example was constructed consisting of reset costs, earnings per parameter value, and embedded harmful arrivals. Two embedded harmful arrival processes were used, IID Bernoulli and an identically distributed discrete Markov Modulated Poisson Process (MMPP). All eleven policies were run with each type of embedded arrivals, and results were compared.

4.1 Parameterization and Other Choices

The service parameter values took values of .1, .125, .15, . . . , .875, .9. The reset costs were -1 for the lower half of parameter values, and then decreased by $\frac{1}{5}$ beyond that. The interarrival distribution was a $A_n \sim$ Geometric($\frac{1}{4}$), although the arrival process need not be memoryless, and the service distribution was $S_n \sim$ Binomial(6,θ_n). The cost of running the server was 2 per time unit. Each customer processed gives the system an earnings of 10. The amount we earn or lose in state θ_i was found with

$$R(\theta_i) = \begin{cases} \frac{10}{E[A]} - 2, & \text{when } \rho_{\theta_i} < 1 \\ \frac{10}{E_{\theta_i}[S]} - 2, & \text{when } \rho_{\theta_i} \geq 1. \end{cases}$$

This parameterization of the $R(\theta_i)$'s calculates earnings strictly from revenues and costs. While this might be appropriate in some situations (as in when customers have many such queues to choose

from), it might not be realistic in many models. One might wish to penalize the server for longer waiting times or higher variance of waiting times. In such a situation, we would like to set the $\{R(\theta_i)\}$ sequence to something that decreases faster as ρ approaches unity.

Finally, the embedded arrival process selected jobs as harmful with probability $\frac{1}{4}$. But for the bursty version, they were identically distributed (as the MMPP was started in stationarity). The process alternated between two states: bursty and lull. In the bursty state, jobs were selected as harmful with probability $\frac{3}{4}$ and the MMPP entered the lull state with probability $\frac{1}{2}$ with each arrival. In the lull state, jobs were selected with probability $\frac{1}{8}$ and the lull state was exited with probability $\frac{1}{8}$. So in stationarity, the MMPP spent one-fifth of arrivals in the bursty state and four-fifths in the lull state. This tended to make bursty arrivals as in (NIL NIL NIL NIL NIL NIL NIL T T T NIL NIL NIL NIL NIL NIL NIL NIL) where T's were harmful and NIL's were not.

4.2 Simulation Details and Data Collection

Lindley's equation gave the simulation its time steps. At the completion of each service time, the policies were used to determine whether or not to perform a reset before the next service was started. Whenever a reset occurred, and line was cleared, and all relevant information was stored. Among it was the total amount earned during the last cycle (since the last reset), the length of the cycle, the number of harmful arrivals seen during the last cycle (although, the policy was not necessarily aware of these), waiting times, interarrival times, and service times.

The key question we would like to see answered is: how much can we earn by using each policy? After all the data were collected, an average earning (per service) was computed for each cycle. These gave us IID observations of how much policies actually earn. Finally, these average earnings (after control variates) were averaged for our final answer: how much the policies earn per service.

Two variance reduction techniques were employed. Common random numbers were used, although, the policies rarely reset at the same time, so individual cycles did not necessarily operate under comparable conditions. The common random numbers were stored in a database to facilitate parallelization, so all results are deterministically repeatable. Also, two control variates were used on the number of harmful arrivals seen in each cycle (more harmful arrivals would cause any policy

policy description	num cycles	average earned	percent of perfect
IID: Perfect Information	783	0.47492	100.0000
IID: Service Times	684	0.47212	99.41009
IID: Arrival Times	665	0.47133	99.24315
IID: Waiting Times	650	0.47071	99.11310
IID: No Information	657	0.46747	98.43045
LocalLin: Service Times	537	0.46281	97.44881
LocalLin: Arrival Times	542	0.46188	97.25247
LocalLin: Waiting Times	531	0.45403	95.60004
Hybrid: Service Times	717	0.47210	99.40511
Hybrid: Arrival Times	724	0.47182	99.34599
Hybrid: Waiting Times	840	0.47079	99.12860

Table 4.1: Performance of the eleven policies when given IID embedded arrivals

to earn less) and the length of a cycle (since we earn a positive amount unless we are resetting).

4.3 Results and Discussion

With IID embedded harmful arrivals, the IID measure performed the best, as expected. Also as expected, as less information was provided, the corresponding policy performed worse. For each policy, table 4.1 provides a summary. It lists the policy, the number of cycles, the average of averages the policy earned (its score), and finally, its score as a percent of perfect information's score. Note that a server earns $\frac{1}{2}$ in the very first service after a reset in our example, so the average earned should be just less than $\frac{1}{2}$. The local linearization performed much worse than the no information case, so (with one exception), it will not be mentioned again. Also, the hybrid measure actually performed better than the IID measure, particularly when interarrival and waiting times were provided. We assume this was to help account for the variance in the arrival process.

Next, we have the same thing with bursty lines in table 4.2. Note that now the IID measures no longer get their assumption of independence. The performance levels degrading faster (with respect to perfect information) with the loss of information. We see the same trend with the hybrid measures as we did for the IID interarrivals. It is interesting to note that with bursty harmful arrivals, the local linearization actually performed better than the previous case (oddly, the variance went down as well).

Finally, included in table 4.3 is the sample standard deviations of all these estimators, so that we have everything we need to determine whether a particular level of information is worth its cost (for this example). A simple Z-test will tell us whether or not the information gives enough

policy description	num cycles	average earned	percent of perfect
IID: Perfect Information	774	0.47470	100.0000
IID: Service Times	674	0.47197	99.42338
IID: Arrival Times	657	0.47102	99.22496
IID: Waiting Times	641	0.47003	99.01445
IID: No Information	657	0.46499	97.95313
LocalLin: Service Times	535	0.46325	97.58643
LocalLin: Arrival Times	536	0.46281	97.49496
LocalLin: Waiting Times	530	0.45475	95.79644
Hybrid: Service Times	707	0.47184	99.39770
Hybrid: Arrival Times	713	0.47149	99.32247
Hybrid: Waiting Times	832	0.47011	99.03299

Table 4.2: Performance of the eleven policies when given bursty embedded arrivals

policy description	StdDev for IID	StdDev for Bursty
IID: Perfect Information	0.00635	0.00654
IID: Service Times	0.00457	0.00492
IID: Arrival Times	0.00442	0.00519
IID: Waiting Times	0.00464	0.00566
IID: No Information	0.00979	0.01774
LocalLin: Service Times	0.00693	0.00647
LocalLin: Arrival Times	0.01099	0.00706
LocalLin: Waiting Times	0.01545	0.01564
Hybrid: Service Times	0.00586	0.00618
Hybrid: Arrival Times	0.00666	0.00648
Hybrid: Waiting Times	0.00783	0.00801

Table 4.3: Standard deviations of the estimators

benefit.

4.4 Choosing Gamma

Now that we have evidence that short term memory can be beneficial in rapidly changing scenarios, we would like to determine how to set γ. While we suspect that this is a difficult question in general, working with the example described above we can examine how changing γ changes the results. In the simulations above, we used $\gamma = 0.8$. But another batch of simulations were run with γ varying between 0.1 and 0.99 when applied to the "service times provided" policy (least computationally intensive among the four lower levels of information). The results of this simulation provides, for this example, that $\gamma \approx 0.71$ is the best value. Other values of $0.1 \leq \gamma \leq 0.99$ are plotted below, smoothed (Figure 4.1) and unsmoothed (Figure 4.2).

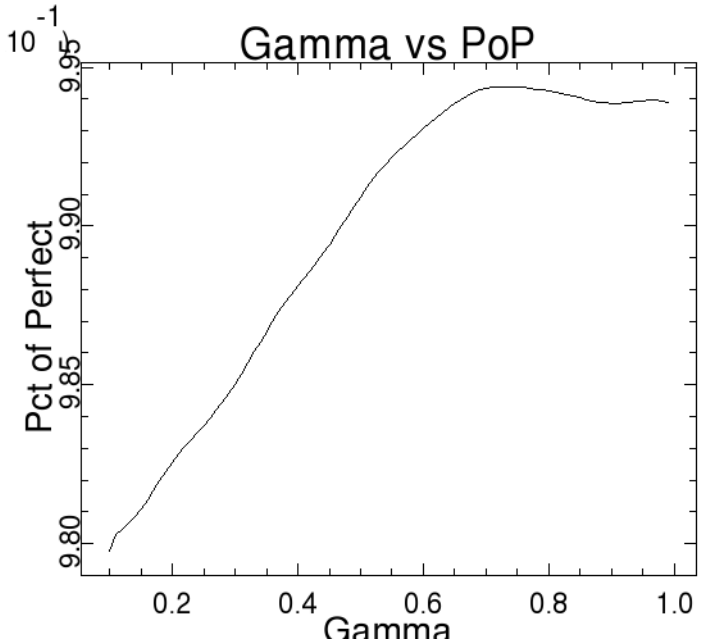

Figure 4.1: Gamma vs Percent of Perfect, after Gaussian smoothing

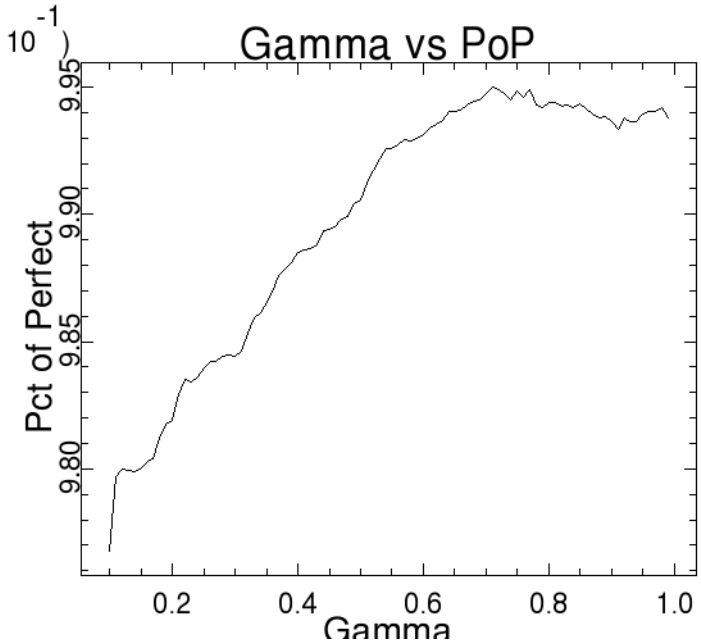

Figure 4.2: Gamma vs Percent of Perfect

Chapter 5

Conclusions and Future Directions

In conclusion, extra information improves the performance of the agent performing the resets. Whether such an improvement is worth the information's cost is really dependent upon the situation. With bursty embedded harmful arrivals, the performance of the agent degraded even more quickly as information was removed. Here, one might more easily justify extra information. Also, short term memory helped to mitigate the effect of burstiness (for some of the levels of information). We provided the existence of an example where short term memory is quite beneficial.

Throughout, we have assumed a linear state space in θ. Such an assumption is not necessary however. All the results contained herein still apply if θ follows are more general Markov state space. Our methods are directly applicable. But as the state space becomes more complicated, the POMDP becomes less feasible to use in practice.

Many unanswered questions remain. We would like to remove the assumption that the line is cleared with each reset so that we can get estimates for $E[W|\text{policy}]$ and $Var(W|\text{policy})$, and in particular, a tail distribution on the waiting time for each policy. Also, it might be worth finding an optimal way of setting γ, the short term memory exponent in the hybrid pmf, as a function of some measure of the autocorrelation of harmful arrivals (assuming such information is observable).

Appendix A

Computer Simulation

```
;;;; main.lisp
;;;; This file loads all the other files
;; these two (require)'s assume ASDF is loaded (or organic as in SBCL)

(require 'cl-plplot)
(require 'clsql)

(defvar *sql-reader* nil)
(unless *sql-reader*
  (clsql:enable-sql-reader-syntax)
  (setf *sql-reader* t))

;(declaim (optimize (speed 3)))

(defmacro lf (fn)
  `(load (compile-file ,fn)))

(lf "memo.lisp")
(lf "distn.lisp")
(lf "database.lisp")
(lf "mdp.lisp")
(lf "accounting.lisp")
(lf "utilities.lisp")
(lf "const-model.lisp")
(lf "var-model.lisp")
(lf "model-construction.lisp")
(lf "process.lisp")
(lf "stats.lisp")
```

```
;;;; memo.lisp
;; this file contains functions for memoizing

(defparameter *hashes* nil)

(defun clear-memo-hashes ()
  (dolist (x *hashes*)
    (clrhash x)))

(defun memo (fn undo-fn)
  (let ((cache (make-hash-table :test #'equal)))
    (push cache *hashes*)
    #'(lambda (&rest args)
(if (eq (car args) 'undo) (apply undo-fn cache (cdr args))
    (multiple-value-bind (val win) (gethash args cache)
      (if win
  val
  (setf (gethash args cache)
(apply fn args))))))))

(defun memoize (fn undo-fn)
  (setf (symbol-function fn) (memo (symbol-function fn) undo-fn)))

(defmacro defun-memo (fn undo-fn args &body body)
  `(memoize (defun ,fn ,args . ,body) ,undo-fn))
```

```
;;;; distn.lisp
;; This file includes a lot more distribution types than needed in
;; this project

(defun between (l x u) (and (<= l x) (<= x u)))

(defclass distn () ())

(defgeneric mean (distn) (:documentation "Compute the mean of this
distribution"))
(defgeneric 2ndmoment (distn) (:documentation "Compute the second moment"))
(defgeneric variance (distn) (:documentation "Compute the variance"))
(defgeneric pdf (distn x) (:documentation "Compute the pdf of distn at x"))
(defgeneric pdf-given (distn x &rest given) (:documentation "give conditional
distribution, given a list of info"))
(defgeneric cdf (distn x) (:documentation "Compute the cdf of distn at x"))
(defgeneric cdf-range (distn x y) ( :documentation "Compute the cdf of
a range: (= F(y) - F(x))" ))
(defmethod  cdf-range ((var distn) x y)
  (if (<= y x) 0
      (- (cdf var y) (cdf var x))))
(defgeneric cdf-given (distn x &rest given) (:documentation "Compute
the cdf given extra information"))
(defgeneric sample (distn &optional u) (:documentation "Generate a
value distributed according to distn (give u for
common-random-numbers)"))
(defgeneric integrate (distn fn) (:documentation "Integrate/Take
expectation over function fn (ex: fn=identity=>E[X]"))
(defgeneric integrate-given (distn fn &rest
given) (:documentation "Integrate given extra information"))
(defgeneric adjust-p-from-mean (distn mean) (:documentation "Adjust a single
parameter so its mean is mean"))
(defgeneric set-single-parameter-p (distn p) (:documentation "Adjust a single
parameter.  Intent: as p increases, the rv stochastically increases."))
(defgeneric to-string (distn) (:documentation "This outputs a string
to reconstruct the object as (apply #'make-instance ~A)"))

(defclass cont-distn (distn)
  ((h :initarg :h :initform .001 :documentation "h = width of quadrature
  intervals")))
(defclass disc-distn (distn)
  ((h :initarg :h :initform 1 :documentation "h = spaces if applicable")))
(defclass compact-support (distn)
  ((lower :initarg :lower :initform 0 :documentation "Lower endpoint")
   (upper :initarg :upper :initform 1 :documentation "Upper endpoint")))
(defclass unbounded-on-right (distn)
  ((lower :initarg :lower :initform 0 :documentation "Lower endpoint")
```

```
   (tol :initarg :tol :initform .001 :documentation "Tolerance for the
   distribution")))
(defclass unbounded-support (distn)
  ((tol :initarg :tol :initform .001 :documentation "Tolerance for the
  distribution")))

(defclass discrete-compact (disc-distn compact-support) ())
(defclass discrete-list (discrete-compact)
  ((atoms :initarg :atoms :initform '(0) :documentation "A list of atoms")))
(defclass discrete-unbounded-on-right (disc-distn unbounded-on-right) ())
(defclass cont-compact (cont-distn compact-support) ())
(defclass cont-unbounded-on-right (cont-distn unbounded-on-right) ())
(defclass cont-unbounded-symmetric (cont-distn unbounded-support) ())

;(defmethod set-single-parameter-p ((var disc-distn) p)
;  (when (not (eq (type-of var) 'disc-uniform))
;    (setf (slot-value var 'p) (min p 1)))) ;; note, we're not allowing p>1
(defun exp-val (var fn &rest given) ;; uniform interface
  (if given (apply #'integrate-given var fn given)
      (funcall #'integrate var fn)))
(defmethod integrate ((var discrete-compact) fn)
  (let ((h (slot-value var 'h))(lower (slot-value var
    'lower))(upper (slot-value var 'upper)))
    (do* ((i lower (+ i h))(prob (pdf var (the number i))
 (pdf var (the number i)))(final-answer
   0 (+ final-answer this-answer))
  (this-val (the number (funcall fn (the number i))) (the
         number (funcall fn (the number i)))) (this-answer (*
 prob this-val) (* prob this-val)))
 ((> i upper) final-answer))))
(defmethod integrate ((var discrete-list) fn)
  (let ((atoms (slot-value var 'atoms)))
    (do ((answer 0 (+ answer (* (funcall fn (car atoms)) (pdf var (car
    atoms)))))(atoms atoms (cdr atoms)))
((null atoms) answer))))
(defmethod integrate-given ((var discrete-list) fn &rest given)
  (let ((atoms (slot-value var 'atoms)))
    (do ((answer 0 (+ answer (* (the number (funcall fn (the
            number (car atoms))))
   (the number (apply #'pdf-given var (the
   number (car atoms)) given)))))(atoms atoms (cdr atoms)))
((null atoms) answer))))
(defmethod integrate ((var discrete-unbounded-on-right) fn)
  (let ((h (slot-value var 'h))(lower (slot-value var
  'lower))(tol (slot-value var 'tol)))
    (do* ((i lower (+ i h))(prob (pdf var i) (pdf var i))(cdf (- 1
    prob) (- cdf prob))
```

```
  (this-val (the number (funcall fn (the number i))) (the
  number (funcall fn (the number i)))) (answer (* prob
  this-val) (+ answer (* prob this-val))))
 ((< cdf tol) answer))))
(defmethod integrate ((var cont-compact) fn)
  (let* ((h (slot-value var 'h))(lower (slot-value var 'lower))
 (upper (slot-value var 'upper))(h-over-2 (/ h 2)))
    (do* ((i lower j)(j (+ i h) (+ j h))(prob (cdf-range var i
    j) (cdf-range var i j))(final-answer 0 (+ this-answer
    final-answer))
  (this-val (funcall fn (+ i h-over-2)) (funcall fn (+ i
  h-over-2)))(this-answer (* prob this-val) (* prob
  this-val)))
 ((> j upper) (+ final-answer (* (cdf-range var i
 upper) (funcall fn (/ (+ i upper) 2))))))))
(defmethod integrate ((var cont-unbounded-on-right) fn)
  (let* ((h (slot-value var 'h))(lower (slot-value var
  'lower))(h-over-2 (/ h 2))(tol (slot-value var 'tol)))
    (do* ((i lower j)(j (+ i h)(+ j h))(prob (cdf-range var i
    j)(cdf-range var i j))(cdf (- 1 prob)(- cdf prob))
  (this-val (funcall fn (+ i h-over-2))(funcall fn (+ i
  h-over-2)))(answer (* prob this-val) (+ answer (* prob
  this-val))))
 ((< cdf tol) answer))))
(defmacro symm-helper (pdf mean offset multiplier fn)
  `(* ,multiplier ,pdf (+ (funcall ,fn (+ ,mean ,offset)) (funcall
  ,fn (- ,mean ,offset)))))
(defmethod integrate ((var cont-unbounded-symmetric) fn)
  (let* ((h (slot-value var 'h))(h-over-2 (/ h 2))(mean (mean
  var))(tol (slot-value var 'tol)))
    (do* ((offset2 h-over-2 (+ offset2 h))(offset4 h (+ offset4
    h))(answer (* 4 (pdf var mean) (funcall fn mean)) (+ answer
    this-part))
  (pdf2 (pdf var (+ mean offset2)) (pdf var (+ mean
  offset2))) (pdf4 (pdf var (+ mean offset4)) (pdf var (+ mean
  offset4)))
  (part2 (symm-helper pdf2 mean offset2 2 fn) (symm-helper
  pdf2 mean offset2 2 fn))
  (part4 (symm-helper pdf4 mean offset4 4 fn) (symm-helper
  pdf4 mean offset4 4 fn))
  (this-part (+ part2 part4) (+ part2 part4))
  (cdf (- 1 (* 4 h-over-2 1/3 (pdf var mean)) (* 8 h-over-2
  1/3 pdf4) (* 4 h-over-2 1/3 pdf2))
       (- cdf (* 8 h-over-2 1/3 pdf4) (* 4 h-over-2 1/3
       pdf2))))
 ((< cdf tol) (* h-over-2 (/ 3) (+ answer (* pdf4 part4) (* (/
 2) pdf2 part2)))))))
```

```
;;;; here are some of the continuous distributions

;;; CONTINUOUS UNIFORM
(defclass uniform (cont-compact) ())
(defmethod mean ((var uniform))
  (/ (+ (slot-value var 'lower) (slot-value var 'upper)) 2))
(defmethod pdf ((var uniform) x)
  (let* ((upper (slot-value var 'upper)) (lower (slot-value var
  'lower))(width (- upper lower)))
    (if (between lower x upper) (/ width) 0)))
(defmethod cdf ((var uniform) x)
  (let* ((upper (slot-value var 'upper)) (lower (slot-value var
  'lower))(width (- upper lower)))
    (/ (- (min x upper) lower) width)))
(defmethod sample ((var uniform) &optional (u (random 1.0)))
  (let* ((lower (slot-value var 'lower))(width (- (slot-value var
  'upper) lower)))
    (+ (* width u) lower)))
(defmethod to-string ((var uniform))
  (with-slots (h lower upper) var
    (format nil "(uniform :h ~A :lower ~A :upper ~A)" h lower upper)))


;;; EXPONENTIAL
(defclass exponential (cont-unbounded-on-right)
  ((mu
    :initarg :mu :initform 1 :documentation "Mu = 1 / mean")))
(defmethod mean ((var exponential)) (/ (slot-value var 'mu)))
(defmethod pdf ((var exponential) x)
  (let ((mu (slot-value var 'mu)))
    (if (< x 0) 0
(exp (* mu (- mu) x)))))
(defmethod cdf ((var exponential) x)
  (let ((mu (slot-value var 'mu)))
    (- 1 (exp (* (- mu) (max x 0))))))
(defmethod sample ((var exponential) &optional (u (random 1.0)))
  (let ((mu (slot-value var 'mu)))
    (/ (log u) (- mu))))
(defmethod to-string ((var exponential))
  (with-slots (h lower tol mu) var
    (format nil "(exponential :h ~A :lower ~A :tol ~A mu: ~A)" h lower
    tol mu)))

;;; NORMAL
(defclass normal (cont-unbounded-symmetric)
  ((mu :initarg :mu :initform 0 :documentation "Mu = population mean")
```

```
   (sigma :initarg :sigma :initform 1 :documentation "Sigma = variance
   > 0")))
(defmethod mean ((var normal)) (slot-value var 'mu))
(defmethod variance ((var normal)) (slot-value var 'sigma))
(defmethod pdf ((var normal) x)
  (let* ((mu (slot-value var 'mu))(sigma (slot-value var
  'sigma))(factor (/ 1 sigma (sqrt (* 2 pi)))))
    (* factor (exp (/ (expt (/ (- mu x) sigma) 2) -2)))))
(defmethod cdf ((var normal) x)
  (integrate var (lambda (y) (if (<= y x) 1 0))))
(defmethod cdf-range ((var normal) x y)
  (if (>= x y) 0
      (integrate var (lambda (z) (if (between x z y) 1 0)))))
(defmethod to-string ((var normal))
  (with-slots (h tol mu sigma) var
    (format nil "(normal :h ~A :tol ~A :mu ~A :sigma ~A)" h tol mu
    sigma)))

(defmacro tween01 (p)
  `(max 0 (min 1 ,p)))

;;;; here are some of the discrete distributions

;;; BERNOULLI
(defclass bernoulli (discrete-compact)
  ((p :initarg :p :initform 1/2 :documentation "p = P(success)")
   (h :initarg :h :initform 1 :documentation "h = 1, always")))
(defmethod mean ((var bernoulli))
  (slot-value var 'p))
(defmethod pdf ((var bernoulli) x)
  (cond ((= x 0) (- 1 (slot-value var 'p)))
 ((= x 1) (slot-value var 'p))
 (t 0)))
(defmethod cdf ((var bernoulli) x)
  (cond ((< x 0) 0) ((< x 1) (slot-value var 'p)) (t 1)))
(defmethod sample ((var bernoulli) &optional (u (random 1.0)))
  (let ((p (slot-value var 'p)))
    (if (< u p) 1 0)))
(defmethod adjust-p-from-mean ((var bernoulli) mean)
  (with-slots (p) var
    (setf p (tween01 mean))))
(defmethod set-single-parameter-p ((var bernoulli) param)
  (with-slots (p) var
    (setf p (tween01 param))))
(defmethod to-string ((var bernoulli))
  (with-slots (p h lower upper) var
    (format nil "(bernoulli :p ~A :h ~A :lower ~A :upper ~A)" p h
```

```
    lower upper)))


;;; BINOMIAL
(defclass binomial (discrete-compact)
  ((p :initarg :p :initform 1/2 :documentation "p = P(success)")
   (n :initarg :n :initform 5 :documentation "n = number of trials")
   (upper :initarg :upper :initform (error "Must set upper =
   n") :documentation "upper = n = number of trials")
   (h :initarg :h :initform 1 :documentation "h = 1 always")))
(defun factorial (n)
  (if (< n 2) 1
      (* n (factorial (1- n)))))
(defun choose (n r)
  (/ (factorial n) (factorial r) (factorial (- n r))))
(defmethod mean ((var binomial))
  (* (slot-value var 'n) (slot-value var 'p)))
(defmethod pdf ((var binomial) x)
  (let* ((p (slot-value var 'p))(q (- 1 p))(n (slot-value var 'n)))
    (if (not (between 0 x n)) 0
(* (choose n x) (expt p x) (expt q (- n x))))))
(defmethod cdf ((var binomial) x)
  (let* ((p (slot-value var 'p))(q (- 1 p))(n (slot-value var
  'n))(answer 0))
    (dotimes (i (1+ x)) (incf answer (* (choose n i) (expt p i) (expt
    q (- n i))))) answer))
(defmethod sample ((var binomial) &optional (u (random 1.0)))
  (do ((i 0 (1+ i))(u u (- u (pdf var i))))
      ((< u 0) (1- i))))
(defmethod adjust-p-from-mean ((var binomial) mean)
  (with-slots (p n) var
    (setf p (/ (max 0 (min n mean)) n))))
(defmethod set-single-parameter-p ((var binomial) param)
  (with-slots (p) var
    (setf p (tween01 param))))
(defmethod to-string ((var binomial))
  (with-slots (h lower upper p n) var
    (format nil "(binomial :h ~A :lower ~A :upper ~A :p ~A :n ~A)" h
    lower upper p n)))


;;; GEOMETRIC
(defclass geometric (discrete-unbounded-on-right)
  ((p :accessor p      ;; notice the accessor function, call (p var)
      ;; to get or set p
    :initarg :p :initform 1/2 :documentation "p = P(success)")
   (h :initform 1 :documentation "h = 1 always")))
```

```
(defmethod mean ((var geometric))
  (/ (slot-value var 'p)))
(defmethod pdf ((var geometric) x)
  (if (< x 1) 0
    (let* ((p (slot-value var 'p)) (q (- 1 p)))
      (* (expt q (1- x)) p))))
(defmethod cdf ((var geometric) x)
  (if (< x 1) 0
      (let* ((p (slot-value var 'p))(q (- 1 p))(answer 0))
(dotimes (i x) (incf answer (expt q i)))
(* answer p))))
(defmethod sample ((var geometric) &optional (u (random 1.0)))
  (do* ((i 0 (1+ i))(answer (pdf var 0) (incf answer (pdf var i))))
       ((> answer u) i)))
(defmethod adjust-p-from-mean ((var geometric) mean)
  (with-slots (p) var
    (setf p (/ (max 0 mean)))))
(defmethod set-single-parameter-p ((var geometric) param)
  (with-slots (p) var
    (setf p (- 1 (tween01 param)))))
(defmethod to-string ((var geometric))
  (with-slots (p h lower tol) var
    (format nil "(geometric :p ~A :h ~A :lower ~A :tol ~A)" p h lower tol)))


;;; TRUNCATED GEOMETRIC
(defclass truncated-geometric (discrete-compact) ;;; this one has mass
 ;;; at zero!!!
  ((p :accessor p :initarg :p :initform 1/2 :documentation "p = P(success)")
   (h :initform 1 :documentation "h = 1 always")
   (lower :initform 0 :initarg :lower :documentation "support over
   [lower,upper]")
   (upper :initform 3 :initarg :upper :documentation "support over
   [lower,upper]")))
(defmethod mean ((var truncated-geometric)) ;;;; THESE NEED WORK!!!!!!!
  (with-slots (lower upper) var
    (do ((i lower (1+ i))(answer 0 (+ answer (* i (pdf var i)))))
((> i upper) answer))))
(defmethod pdf ((var truncated-geometric) x)
  (with-slots (p upper) var
    (if (or (< x 0) (> x upper)) 0
(/ (* p (expt (- 1 p) x)) (- 1 (expt (- 1 p) (1+ upper)))))))
(defmethod cdf ((var truncated-geometric) x)
  (do ((i 0 (1+ i))(answer 0 (+ answer (pdf var i))))
      ((> i x) answer)))
(defmethod sample ((var truncated-geometric) &optional (u (random 1.0)))
  (with-slots (upper) var
```

```
    (do ((i 0 (1+ i))(u u (- u (pdf var i))))
((or (< u 0) (> i upper)) (1- i)))))
(defmethod adjust-p-from-mean ((var truncated-geometric) mean)
  (bisect01 #'(lambda (p) (set-single-parameter-p var p) (mean var)) mean))
(defmethod set-single-parameter-p ((var truncated-geometric) param)
  (with-slots (p) var
    (setf p (- 1 (tween01 param)))))
(defun bisect01 (mono-up-fn target &optional (left 0) (right 1) (tol .001))
  "Bisect this interval trying to make the fn return target, used in
adjust-p-from-mean"
  (let ((left-val (funcall mono-up-fn left))
(middle (/ (+ left right) 2))
(middle-val (funcall mono-up-fn (/ (+ left right) 2))))
    (cond ((< (- right left) tol) middle)
  ((< (* (- left-val target) (- middle-val target))
  0) (bisect01 mono-up-fn target left middle tol))
  (t (bisect01 mono-up-fn target middle right tol)))))
(defmethod to-string ((var truncated-geometric))
  (with-slots (p h lower upper) var
    (format nil "(truncated-geometric :p ~A :h ~A :lower ~A :upper
    ~A)" p h lower upper)))


;;; NEGATIVE BINOMIAL
(defclass negative-binomial (geometric)
  ((r :initarg :r :initform (error "Must enter :r") :documentation "r
  = required number of successes")
   (lower :initarg :lower :initform (error "Must
   enter :lower") :documentation "lower = r = must be the same!")))
(defmethod mean ((var negative-binomial))
  (/ (slot-value var 'r) (slot-value var 'p)))
(defmethod pdf ((var negative-binomial) x)
  (let* ((p (slot-value var 'p))(q (- 1 p))(r (slot-value var 'r)))
    (if (< x r) 0
(* (choose (1- x) (1- r)) (expt p r) (expt q (- x r))))))
(defmethod cdf ((var negative-binomial) x)
  (let* ((p (slot-value var 'p))(q (- 1 p))(r (slot-value var 'r)))
    (if (< x r) 0
(do ((n r (1+ n))(answer 0 (+ answer (* (choose (1- n) (1-
r)) (expt p r) (expt q (- n r))))))
    ((> n x) answer)))))
(defmethod sample ((var negative-binomial) &optional (u (random 1.0)))
  (let ((r (slot-value var 'r)))
    (do* ((i r (1+ i))(answer (pdf var r) (incf answer (pdf var i))))
 ((>= answer u) i))))
(defmethod adjust-p-from-mean ((var negative-binomial) mean)
  (with-slots (p r) var
```

```
    (setf p (/ r (+ (max mean r) r)))))
(defmethod set-single-parameter-p ((var negative-binomial) param)
  (with-slots (p) var
    (setf p (- 1 (tween01 param)))))
(defmethod to-string ((var negative-binomial))
  (with-slots (h lower tol r) var
    (format nil "(negative-binomial :h ~A :lower ~A :tol ~A :r ~A)" h
    lower tol r)))




;;; DISCRETE UNIFORM
(defclass disc-uniform (discrete-compact)
  ((lower :initarg :lower :initform 0 :documentation "Lower endpoint
  inclusive")
   (upper :initarg :upper :initform 1 :documentation "Upper endpoint
   inclusive")
   (h :initarg :h :initform 1 :documentation "h = Space between the
   atoms")))
(defmethod mean ((var disc-uniform))
  (/ (+ (slot-value var 'lower) (slot-value var 'upper)) 2))
(defun disc-uniform-helper-get-idx (var x)
  (let* ((lower (slot-value var 'lower))(upper (slot-value var
  'upper))(h (slot-value var 'h))
 (width (- upper lower))(n (+ (/ width h) 1))(idx (1+ (/ (- x
 lower) h))))
    (values idx n)))
(defmethod pdf ((var disc-uniform) x)
  (multiple-value-bind (idx n) (disc-uniform-helper-get-idx var x)
    (cond ((> idx n) 0)
  ((integerp idx) (/ n))
  (t 0))))
(defmethod cdf ((var disc-uniform) x)
  (multiple-value-bind (idx n) (disc-uniform-helper-get-idx var x)
    (cond ((> idx n) 1)
  ((< idx 1) 0)
  (t (/ (floor idx) n)))))
(defmethod sample ((var disc-uniform) &optional (u (random 1.0)))
  (let ((n (slot-value var 'n)) (lower (slot-value var
  'lower)) (upper (slot-value var 'upper)))
    (let* ((width (- upper lower)) (num (floor (* (1+ n) u))))
      (+ lower (* width (/ num (1- n)))))))
(defmethod to-string ((var disc-uniform))
  (with-slots (h lower upper) var
    (format nil "(disc-uniform :h ~A :lower ~A :upper ~A)" h lower upper)))
```

```
(defclass list-of-outcomes (discrete-list)
  ((outcomes :initarg :outcomes :initform '(0 1) :documentation "List
  of possible outcomes")
   (probabilities :initarg :probabilities :initform '(.5
   .5) :documentation "List of corresponding probabilities")))
(defmethod  pdf ((var list-of-outcomes) x)
  (let ((pos (position x (slot-value var 'outcomes))))
    (if pos (nth pos (slot-value var 'probabilities)) 0)))
(defmethod sample ((var list-of-outcomes) &optional (u (random 1.0)))
  (do* ((outs (slot-value var 'outcomes) (cdr outs))
(probs (slot-value var 'probabilities) (cdr probs))
(u (- u (car probs)) (- u (car probs))))
       ((or (null probs) (>= 0 u)) (car outs))))
```

```
;;;; database.lisp
;; This file handles most if not all of the database interation

(defvar *db* (clsql:connect
'("localhost" "queuing" "lisp" "lisp")' :database-type :mysql))

(defun connect ()
  (setf *db* (clsql:connect
  '("localhost" "queuing" "lisp" "lisp")' :database-type :mysql)))

(defun disconnect ()
  (clsql:disconnect) (setf *db* nil))

(defparameter *use-db* t)

(defun last-insert-id ()
  (if *use-db*
      (let ((response (clsql:query "select last_insert_id()")))
(caar response))
      (random 100)))

(defun qs (s)
  (format nil "~A" s))

(defun serialize-scenario-pt1 (range resets utilities transition
transition-model)
  (if *use-db*
      (progn
(unless *db* (error "We're not connected to the db?"))
(clsql:insert-records :into [model] :attributes '(range resets
utilities transition transition_model)
      :values `(,(qs range) ,(qs resets) ,(qs
      utilities) ,(qs transition)
      ,(qs (to-string transition-model))))
(last-insert-id))
      (random 100)))

(defun serialize-scenario-pt2 (modelid arrival service)
  (declare (ignorable modelid))
  (when *use-db*
      (clsql:update-records [MODEL] :attributes '(arrival_distn
      service_distn) :values `(,(qs (to-string arrival))
      ,(qs (to-string service))))))

(defun serialize-policy (model filter modelid)
  (when *use-db*
    (with-slots (policy-id short-term desc) model
```

```
      (unless *db* (error "We're not connected to the db?"))
      (clsql:insert-records :into [policy] :attributes '(modelid
      short_term filter description)
    :values `(,modelid ,short-term ,filter ,desc))
      (setf (slot-value model 'policy-id) (last-insert-id)))))

(defun serialize-output (model variable data n)
  (if *use-db*
    (with-slots (policy-id) model
      (clsql:insert-records :into [output] :attributes '(policyid n
      variable data)
    :values `(,policy-id ,n ,(format
    nil "'~A'" variable) ,(qs data))))
    (with-slots (desc) model
      (format t "~A - output: n: ~A for ~A : ~A~%" desc n variable data))))

(defun get-model-id (model)
  (if *use-db*
      (let ((policy-id (slot-value model 'policy-id)))
(caar (clsql:query (format nil "select modelid from POLICY
where id = ~A" policy-id))))
      (random 100)))

;-------------------------- DATA PREP -----------------------------

(defun grab (policyid n var)
  (let* ((str (format nil "select data from OUTPUT where policyid = ~A
  and n = ~A and variable like '%~A%'" policyid n var))
 (result (clsql:query str)))
    (read-from-string (caar result))))

(defmacro prep-vars ((policyid n) &body body)
  `(let ((earn (grab ,policyid ,n 'earn)))
    (when (and earn (cdr earn))
      (let ((length (length earn))
    (harm (sample-stddev (interarrivals (grab ,policyid ,n 'harm))))
    (wait (sample-stddev (grab ,policyid ,n 'wait)))
    (service (sample-stddev (grab ,policyid ,n 'service)))
    (arrival (sample-stddev (grab ,policyid ,n 'arrival) 4))
    (earn (sample-mean earn)))
(declare (ignorable length harm wait service arrival earn))
,@body))))

(defun interarrivals (seq)
  (let ((answer (copy-list '(0))))
    (dolist (x (reverse seq))
      (incf (car answer))
```

```
      (when x (push 0 answer)))
    (when (= 0 (car answer)) (pop answer))
    answer))

(defun fill-analysis1 (from-policyid to-policyid embed)
  (dolist (n (clsql:query (format nil "select distinct n from OUTPUT
  where policyid = ~A order by n" from-policyid)))
    (prep-vars (from-policyid n)
      (clsql:insert-records :into [analysis1] :attributes '(embed
      policyid avg_earn s_harmful s_wait s_service s_arrival length)
    :values (list embed to-policyid earn harm
    wait service arrival length)))))

(defun get-var (policy n var)
  (caar (clsql:query (format nil "select data from OUTPUT where
  policyid = ~A and n = ~A and variable like '%~A%'" policy n var))))


;-------------- This seems to be the start of the data prep ----------------

(defun fill-data (from-policy to-policy embed &optional (count 1))
  (let ((from-policy from-policy)(to-policy to-policy))
    (dotimes (i count)
      (dolist (n (clsql:query (format nil "select distinct n from
      OUTPUT where policyid = ~A order by n" from-policy)))
(let ((earn (get-var from-policy n 'earn))
      (harm (get-var from-policy n 'harm))
      (reset (get-var from-policy n 'reset))
      (service (get-var from-policy n 'service))
      (arrival (get-var from-policy n 'arrival))
      (waiting (get-var from-policy n 'wait)))
  (clsql:insert-records :into [data] :attributes '(policy
  embed earn harm reset service arrival waiting)
:values (list to-policy embed earn
harm reset service arrival waiting))))
      (incf from-policy) (incf to-policy)))
  (set-lengths-in-data) (set-harm-counts) (set-earn-counts)
  (dotimes (i count) (var-redux (+ i to-policy) embed)))

;(defun process-a-list-for-filling-data (list embed)
;  (dolist (x list)
;    (let ((from (car x))(to (second x)))
;      (format t "Calling (fill-data ~A ~A ~A)~%" from to embed)
;      (fill-data from to embed))))

(defun set-lengths-in-data ()
  (dolist (x (clsql:query "select id,reset from DATA where length < 0"))
```

```
    (let ((n (length (read-from-string (second x)))) (id (first x)))
      (clsql:update-records [data] :attributes '(length) :values (list
      n) :where [= [id] id])))))
;  (do ((x (clsql:query "select reset from DATA where length < 0 limit
;      1")(clsql:query "select reset from DATA where length < 0 limit
;      1"))) ((null x)) (let ((n (length (read-from-string (caar
;      x))))) (clsql:update-records [data] :attributes '(length)
;      :values (list n) :where [and [= [reset] (car x)] [< [length]
;      0]] ))))

(defun set-harm-counts ()
  (dolist (x (clsql:query "select id,harm from DATA where harm_count < 0"))
    (let ((id (first x)) (harm (read-from-string (second x))))
      (clsql:update-records [data] :attributes
      '(harm_count) :values (list (reduce #'+ (mapcar #'t->1
      harm))) :where [= [id] id]))))

(defun set-earn-counts ()
  (dolist (x (clsql:query "select id,earn from DATA where earn_count < 0"))
    (let ((id (first x)) (earn (read-from-string (second x))))
      (clsql:update-records [data] :attributes
      '(earn_count) :values (list (reduce #'+ earn)) :where [= [id]
      id]))))




;----------------------------- access to DATA table ----------------------

(defun access-data (policy embed column)
  (let ((command '(clsql:select))
(attributes nil)
(from (list :from [data]))
(where (list :where [and [= [policy] policy] [= [embed] embed]])))
    (dolist (x column) (push (clsql:sql-expression :attribute x) attributes))
    (eval (append command (nreverse attributes) from where))))

(defun access-description (policy embed)
  (caar (clsql:select [description] :from [description] :where [and [=
  [embed] embed] [= [policy] policy]])))

;----------------------------- DATA for PLOTTING ------------------------

;(defun get-max (column) (let ((answer 0)) (dolist (policyid
;  (clsql:query (format nil "select distinct policyid from
;  ANALYSIS1"))) (let ((data (clsql:query (format nil "select ~A from
;  ANALYSIS1 where policyid = ~A order by ~A" column policyid
```

```
;  column)))) (do ((top (cdr data) (cdr top))(bottom data (cdr
;  bottom))) ((null top)) (when (< answer (- (caar top) (caar
;  bottom))) (setf answer (- (caar top) (caar bottom))) (when (>
;  answer 2) (format t "Got another max with policy id ~A~%"
;  policyid)))))) answer))

(defun get-data (policy column &optional (embed 1) (model 1))
  (clsql:query (format nil "select ~A,avg_earn from ANALYSIS1 where
       policyid = ~A and embed = ~A and model = ~A
       order by ~A" column policy embed model
       column)))

(defun gauss-blur (x-point data &optional (sigma .25))
  (let ((numerator 0) (denominator 0) (coeff (/ 1 (sqrt (* 2 pi)) sigma)))
    (dolist (datum data)
      (let ((x (car datum))(y (cadr datum)))
(let ((weight (* coeff (exp (* -1/2 (sq (/ (- x-point x) sigma)))))))
  (incf numerator (* weight y))
  (incf denominator weight))))
    (/ numerator denominator)))

(defun construct-array (policyid column &optional (n 1000))
  (let ((data (get-data policyid column))(answer nil))
    (let ((min (apply #'min (mapcar #'car data)))(max (apply
    #'max (mapcar #'car data))))
      (let ((h (/ (- max min) n)))
(do ((i min (+ i h)))
    ((> i max))
  (push (list i (gauss-blur i data)) answer))))
    (nreverse answer)))

(defun construct-histogram (column &optional (bins 100) (n 1000) (embed 1))
  (let ((answer (clsql:query (format nil "select
  min(~A),max(~A),count(*) from ANALYSIS1 where embed = ~A" column
  column embed))))
    (let ((min (caar answer)) (max (cadar
    answer)) (tally (make-array (list bins) :initial-element
    0)) (count (third (car answer))))
      (let ((width (+ (/ (- max min) bins) 1e-6)))
(dolist (x (clsql:query (format nil "select ~A from ANALYSIS1
where embed = ~A" column embed)))
  (incf (aref tally (floor (/ (- (car x) min) width))) (/ count)))
(let ((list nil)(answer-array (make-array (list
n)))(x-array (make-array (list n))))
  (dotimes (i bins) (push (list (+ min (* i width)) (aref
  tally i)) list))
  (setf list (nreverse list))
```

```
  (let ((h (/ (- max min) n)))
    (dotimes (i n)
      (setf (aref x-array i) (+ min (* i h)))
      (setf (aref answer-array i) (gauss-blur (+ min (* i h))
      list width)))
    (list x-array answer-array)))))))

(defun to-array (list)
  (let ((answer (make-array (list (length list)))))
    (do ((i 0 (1+ i))(list list (cdr list)))
((null list) answer)
      (setf (aref answer i) (car list)))))

;------------------------------ LINE table --------------------------------

(defun construct-line-model (n model &optional (line-gen-type *iid-line*))
  (let ((arrival-process (slot-value line-gen-type 'arrival-distn)))
    (dotimes (i n)
      (let ((arrival (sample arrival-process))
    (service-u (random 1.0)))
(multiple-value-bind (harmful?
harmful-count) (arrival-harmful? line-gen-type)
  (declare (ignorable harmful?))
  (clsql:insert-records :into [line] :attributes '(id model
  arrival harmful service_u)
:values `(,i ,model ,arrival
,harmful-count ,service-u)))))))

(defun get-input-to-process-arrival (model n)
  (clsql:query (format nil "select arrival, service_u, harmful from
  LINE where model = ~a order by id limit ~A" model n)))

;---------------------------- Control Variates ---------------------------

(defun setup-ctrl-var (policy embed from-x from-y to)
  (let ((data (clsql:query (format nil "select id,~A,~A from DATA
  where model = 5 and policy = ~A and embed = ~A" from-x from-y policy
  embed))))
    (let* ((x-seq (mapcar #'second data))(y-seq (mapcar #'third data))
   (cov (sample-covariance x-seq y-seq))
   (var (sample-variance y-seq))
   (y-bar (sample-mean y-seq))
   (c-hat-star (if (> var 0) (- (/ cov var)) 0)))
      (dolist (x data)
(let ((id (first x)) (x-val (second x)) (y-val (third x)))
  (let ((ctrl (+ x-val (* c-hat-star (- y-val y-bar)))))
    (clsql:update-records [data] :attributes (list
```

```
    to) :values (list ctrl) :where [= id [id]])))))))

(defun var-redux (policy embed)
;  (clsql:update-records [data] :attributes '(avg_earn) :values (list
;  [/ [earn_count] [length]]) :where [and [= policy [policy]] [= embed
;  [embed]]])
  (setup-ctrl-var policy embed 'avg_earn 'harm_count 'ctrl1)
  (setup-ctrl-var policy embed 'ctrl1 'length 'ctrl2))

(defun redux-all ()
  (do ((i 212 (1+ i)))((> i 311))
    (dotimes (j 2)
      (when (> (caar (clsql:query (format nil "select count(*) from
      DATA where policy=~A and embed=~A and model=1" i (1+ j)))) 0)
(var-redux i (1+ j))))))
```

```
;;;; mdp.lisp
;; this file handles most of the MDP logic


(defclass cost-model ()
  ((prob-measure :initarg :prob-measure :initform (error "Must specify
  a probability measure on state space") :documentation "probability
  measure")
   (prob :initarg :prob :initform nil :documentation "This is the
   stored probability measure on each state from our observations")
   (filter :initarg :filter :initform (error "must spec a
   filter") :documentation "function that filters the data")
   (service-distn :initarg :service :initform (make-instance
   'geometric) :documentation "This is the service distribution")
   (harmful-distn :initarg :harmful :initform
   nil :documentation "These are the alternate service distributions
   for the harmful jobs")
   (arrival-distn :initarg :arrival :initform (make-instance
   'geometric :p 1/3) :documentation "This is the arrival
   distribution")
   (param-values :initarg :param-values :initform (error "Must
   specify :param-values") :documentation "Statespace for the
   parameter (Vector)")
   (reset-costs :initarg :reset-costs :initform (error "Must
   specify :reset-costs") :documentation "Cost to reset from each
   parameter (Vector)")
   (utilities :initarg :utilities :initform (error "Must
   specify :utilities") :documentation "Utility of being in each
   parameter state (Vector)")
   (exp-util :initform nil :documentation "The expected utility of
   each state calculated from policy")
   (short-term :initarg :short-term :initform 1 :documentation "This
   is a parameter representing the short term memory.  1 means
   remember all")
   (policy :initform nil :documentation "The deterministic policy")
   (accounting :initform (construct-account) :documentation "This
   class accounts for the policy's performance")
   (my-line :initform nil :documentation "This is my line")
   (policy-id :initform 0 :documentation "This is the primary key in
   the database")
   (last-wait :initform 0 :documentation "This keeps the last wait time")
   (last-service :initform 0 :documentation "This keeps the last
   service time")
   (desc :documentation "This is the description of my model")))

(defclass cost-model-const-transitions (cost-model)
  ((transition :initarg :transition :initform (error "Must specify
```

```
  transition probs") :documentation "Fixed Transition
  Probabilities (Vector)")
   (state-rv :initarg :state-rv :initform nil :documentation "This is
   the representation of the SS as a RV for integration purposes.")))


(defclass cost-model-estimate-transitions (cost-model)
  ((transition :initarg :transition :initform '(.75
  .25) :documentation "Calculated Transition Probabilities (Vector)")
   (transition-model :initarg :transition-model :initform (make-instance
   'binomial :upper 3 :n 3) :documentation "The distribution to model
   transitions with")
   (phi-disc-width :initarg :phi-disc-width :initform (/
   100) :documentation "This is the width of intervals for our phi
   discritization")
   (phi-grid :initarg :phi-grid :initform nil :documentation "Here's
   where we cache our results from calling likelihood")
   (alpha-cache :documentation "This is the normalizing constant - the
   sum of all elements of phi-grid")
   (for-line :initform nil :documentation "This is the line we're
   caching results for")
   (mmpp :documentation "This is the Markov Moduled Poisson Process
   state.")))

(defgeneric set-distributions (model arrival service &rest
rest) (:documentation "For construction, add the distributions as
parameters"))
(defgeneric get-transitions (model &rest
rest) (:documentation "Returns the fixed or current transition
vector"))
;(defgeneric next-arrival (model &optional arrival service-u harmful)
;  (:documentation "Generates either an iid or bursty line depending
;  on model"))
(defgeneric arrival-harmful? (model) (:documentation "This
tells (depending on the model) whether or not this arrival is
harmful"))

(defmethod get-transitions ((model cost-model-const-transitions) &rest
rest)
  (declare (ignorable rest)) (slot-value model 'transition))

(defmethod get-transitions ((model cost-model-estimate-transitions)
&rest rest)
  (with-slots (transition transition-model for-line) model
    (when rest
      (unless (apply #'equal for-line rest) (apply
      #'fill-likelihood-table model rest))
```

```
      (let ((phi-hat (apply #'mle-of-phi model rest)))
         ;; for the bernoulli case, this is a lot of extra work
(adjust-p-from-mean transition-model phi-hat)
(calc-transition model)))
    transition))

(defun clear-state (model)
  (with-slots (my-line) model
    (setf my-line nil)
    (clear-accounting model)))


(defun iterate-exp-util (exp-utils util-nodes reset-nodes answers
model first-exp-util discount)
  (when exp-utils
    (with-slots (transition) model
      (let ((stay 0)(tmp-exp-utils exp-utils)(reset-util (+ (*
      discount (+ (* discount first-exp-util) (car reset-nodes))) (car
      util-nodes))))
(dolist (x transition)
  (incf stay (* x (car tmp-exp-utils)))
  (when (cdr tmp-exp-utils) (pop tmp-exp-utils)))
(setf stay (* stay discount))
(incf stay (car util-nodes))
(if (< stay reset-util) (setf (car answers) 'reset (car
exp-utils) reset-util)
    (setf (car answers) 'stay (car exp-utils) stay)))
      (iterate-exp-util (cdr exp-utils) (cdr util-nodes) (cdr
      reset-nodes) (cdr answers) model first-exp-util discount))))

(defun utility-norm (one two)
  (do ((x one (cdr x)) (y two (cdr y)) (max 0 (max max (abs (- (car
  x) (car y))))))
      ((or (null x) (null y)) max)))

(defun solve-bellman (model &optional (n 100000) (discount .99) (tol .001))
  (with-slots (exp-util utilities reset-costs policy) model
    (let (old-exp-util)
      (dotimes (i n)
(setf old-exp-util (copy-list exp-util))
(iterate-exp-util exp-util utilities reset-costs policy
model (car exp-util) discount)
(when (< (utility-norm exp-util old-exp-util) tol) (return)))
      policy)))

(defun set-probability (cost-model &rest rest)
  (with-slots (param-values prob-measure prob) cost-model
```

```
    (do ((theta param-values (cdr theta))(probly prob (cdr probly)))
((or (null theta) (null probly)) prob)
      (setf (car probly) (apply prob-measure (car theta) cost-model rest)))))

(defun select-po-action (cost-model &optional (discount .99))
  (with-slots (transition prob exp-util reset-costs) cost-model
    (let ((stay 0)(reset (* discount (car exp-util))))
      (do ((resets reset-costs (cdr resets))(prob prob (cdr prob)))
  ((or (null resets) (null prob)))
(incf reset (* (car prob) (car resets))))
      (do ((exp-util exp-util (cdr exp-util))(prob prob (cdr prob)))
  ((or (null exp-util) (null prob)))
(let ((for-all-s-prime 0)(exp-util exp-util))
  (dolist (x transition)
    (incf for-all-s-prime (* x (car exp-util)))
    (when (cdr exp-util) (pop exp-util)))
  (incf stay (* (car prob) for-all-s-prime))))
      (if (> stay reset) 'stay 'reset))))


(defun decide-stay-or-reset (cost-model info)
  "I'm the decider: takes a line and returns (line action).  This is
meant as input to do-accounting"
  (with-slots (filter) cost-model
    (let ((filtered-info (mapcar filter info)))
      (get-transitions cost-model filtered-info)
      (set-probability cost-model filtered-info)
      (solve-bellman cost-model)
      (list info (select-po-action cost-model)))))

(defun process-arrival (model arrival service-u harmfulness)
  (apply #'do-accounting model (decide-stay-or-reset
  model (next-arrival model arrival service-u harmfulness))))
```

```
;;;; accounting.lisp
;; this file handles most of the accounting for simulation results

;; all the lists of lists are
    ;; outer lists each contain information from one reset to another
    ;; inner lists are the individual-per-service data

(defclass policy-accounting ()
  ((n :initform 0 :documentation "This is the total number of resets
  performed to date")
   (earn-per-cycle :initform '(nil) :documentation "List of lists of
   earnings")
   (harmful-jobs :initform '(nil) :documentation "List of lists.  Each
   t is harmful while nil is not")
   (wait-times :initform '(nil) :documentation "List of lists of wait
   times")
   (service-times :initform '(nil) :documentation "List of lists of
   service times")
   (arrival-times :initform '(nil) :documentation "List of lists of
   arrival times")
   (reset-periods :initform '(nil) :documentation "List of lists of
   resets.  Each t is a reset while a nil is not")))

(defparameter *accounting-fields* '(earn-per-cycle harmful-jobs
wait-times service-times arrival-times reset-periods))

(defun construct-account ()
  (let ((answer (make-instance 'policy-accounting))(fill-with (list nil)))
    (dolist (x *accounting-fields*) (setf (slot-value answer
    x) (copy-list fill-with))) answer))

;(defun count-harmfuls (line)
;  (let ((answer 0)(last 1/10))
;    (dolist (x line)
;      (let ((this (car x)))
; (when (/= this last) (incf answer))
; (setf last this)))
;    answer))

(defun do-accounting (model line action)
  "This is the function the model needs to call AFTER an arrival."
  (with-slots (accounting my-line) model
    (unless line (error "Why are we accounting before anyone has shown
    up?"))
    (handle-earnings accounting model line action)
    (handle-counting accounting model line action)
    (setf my-line (if (eq action 'reset) nil line))))
```

```
(defun handle-earnings (account model line action) ;; this updates
   ;; earn-per-cycle
  (declare (ignorable account line action))
  (with-slots (param-values reset-costs utilities) model
    (let* ((theta (caar line))(pos (position theta param-values)))
      (unless pos (error "Ok, our line generation gave a bogus theta
      value"))
      (let ((cost/earn (if (eq action 'stay) (nth pos utilities) (nth
      pos reset-costs))))
(with-slots (earn-per-cycle) account
  (push cost/earn (car earn-per-cycle)))))))

   ;;;; handle-counting gets called AFTER handle-earnings
(defun handle-counting (account model line action) ;; this updates n,
   ;; harmful-jobs,
   ;; wait-times,
   ;; service-times,
   ;; reset-periods
  (with-slots (param-values last-wait last-service) model
    (let* ((first-param-value (car param-values)) (harmful? (if (cdr
    line) (/= (caar line) (caadr line)) (/= (caar line)
    first-param-value))))
      (with-slots (n harmful-jobs wait-times service-times
      arrival-times reset-periods) account
(push harmful? (car harmful-jobs))
(push (car (cdddar line)) (car wait-times))
(push (cadar line) (car service-times))
(push (caddar line) (car arrival-times))
(push (eq action 'reset) (car reset-periods))
(when (eq action 'reset)
  (setf last-wait (caar wait-times) last-service (caar
  service-times))
  (incf n)
  (dolist (x  *accounting-fields*) ; (push nil (slot-value
   ; account x)))
    (serialize-output model x (car (slot-value account x)) n)
    (setf (car (slot-value account x)) nil))  ;; clear it from
      ;; memory
  (funcall (slot-value model 'prob-measure) 'undo model line)
  (clear-memo-hashes)
  (gc :gen 6))))))

(defmacro clear-list (list)
  `(setf ,list (copy-list '(nil))))

(defun clear-accounting (model)
```

```
(with-slots (accounting my-line) model
  (with-slots (n earn-per-cycle harmful-jobs wait-times
  service-times reset-periods arrival-times) accounting
    (setf n 0)
    (clear-list earn-per-cycle)
    (setf my-line nil)
    (clear-list harmful-jobs)
    (clear-list wait-times)
    (clear-list service-times)
    (clear-list reset-periods)
    (clear-list arrival-times))))
```

```
;;;; utilities.lisp
;; this file contains misc utilities


(defmacro service-xdf (xdf service-xdf)
  `(defun ,service-xdf (model x theta-n &optional (theta-n-1 theta-n))
    (if (< theta-n theta-n-1) 0
(with-slots (param-values harmful-distn service-distn) model
  (set-single-parameter-p service-distn (float theta-n))
  (,xdf service-distn x)))))
        ;;; These next two functions now have parameter list (model x
        ;;; theta-n theta-n-1)
(service-xdf pdf service-pdf)  ;; this creates a function that chooses
       ;; the proper service pdf
(service-xdf cdf service-cdf)  ;; this creates a function that chooses
       ;; the proper service cdf

(defun service-chooser (var x theta-n &optional (theta-n-1 theta-n))
  (with-slots (service-distn) var (set-single-parameter-p
  service-distn (float theta-n)))
  (let ((val (+ (- (cadar x) (cadadr x)) (caar x))))
    (if (> (cadar x) 0) (service-pdf var val theta-n theta-n-1)
(service-cdf var val theta-n theta-n-1))))

(defun convolver (var x theta-n &optional (theta-n-1 theta-n))
  (with-slots (arrival-distn service-distn) var
    (set-single-parameter-p service-distn (float theta-n))
    (let ((service-xdf (if (> (caar x) 0) #'service-pdf
    #'service-cdf))(val (- (caar x) (caadr x))))
      (exp-val arrival-distn #'(lambda (dot) (funcall service-xdf
      var (+ val dot) theta-n theta-n-1))))))

(defgeneric prob-after-one-arrival (model theta-n
theta-n-1) (:documentation "Handles the second base case
accordingly"))

(defmethod prob-after-one-arrival ((model
cost-model-const-transitions) theta-n theta-n-1)
  (with-slots (transition param-values) model
    (let ((idx-n (position theta-n param-values))
  (idx-n-1 (position theta-n-1 param-values)))
      (unless (and idx-n idx-n-1) (error "prob-after-one-arrival can't
      find find theta-n or theta-n-1"))
      (if (>= (- idx-n idx-n-1) (length transition)) 0
  (nth (- idx-n idx-n-1) transition)))))

(defmethod prob-after-one-arrival ((model
```

```
  cost-model-estimate-transitions) theta-n theta-n-1) 1) ;;
  multiplicative identity

(defun-memo unified-service-distn #'list (var x theta-n
&optional (theta-n-1 theta-n))
  (unless x (error "You handed unified-service-distn an empty line"))
  (let ((n (length (car x))))
    (cond ((= n 4) (if (= theta-n (caar x)) 1 0))
  ((= n 3) (service-pdf var (caar x) theta-n theta-n-1))
  ((null (cdr x)) (prob-after-one-arrival var theta-n theta-n-1))
  ((= n 2) (service-chooser var x theta-n theta-n-1))
  ((= n 1) (convolver var x theta-n theta-n-1))
  (t (error "OOps - unified-service-distn")))))

(defun construct-2queue ()
  (let ((lead 0)(follow 0))
    (declare (ignorable lead follow))
    #'(lambda (x)
      (cond ((not (symbolp x)) (shiftf follow lead x))
    ((eq x 'difference) (- lead follow))
    ((eq x 'last) lead)
    (t (pprint "Can't help ya, invalid argument to
    construct-2queue"))))))
```

```
;;;; const-model.lisp
;; this file contains a lot of the non-linearized (first and third)
;; model stuff

(defun construct-const-graph (range resets utilities filter
&optional (transition '(.75 .25))(measure #'iid-pm) )
  (let ((model (make-instance
  'cost-model-const-transitions :param-values range :reset-costs
  resets :utilities utilities
      :prob-measure measure :filter
      filter :transition transition :harmful
      nil)))
    (with-slots (prob exp-util policy) model
      (unless (= (length range) (length utilities) (length
      resets)) (error "range, resets, utilities don't have same
      length"))
      (setf prob (make-list (length range) :initial-element 0))
      (setf (slot-value model 'state-rv) (make-instance
      'state-distn :atoms range))
      (setf exp-util (make-list (length range) :initial-element 0))
      (setf policy (make-list (length range) :initial-element 'stay)))
    model))

(defmethod set-distributions ((model cost-model-const-transitions)
arrival service &rest other-service)
  (with-slots (transition harmful-distn service-distn arrival-distn) model
    (when (>= (length other-service) (length transition)) (error "You
    just supplied more service distns than transitions for them."))
    (push service other-service)
    (setf arrival-distn arrival)
    (do ((trans transition (cdr trans))(distn other-service (if (cdr
    distn) (cdr distn) distn))(answer nil (cons (car distn) answer)))
((null trans) (setf harmful-distn (nreverse answer))))
    (setf service-distn (car harmful-distn))))

;;;; This class represents the RV of the SS.  Integration is already
;;;; defined in distn.lisp
(defclass state-distn (discrete-list) ())
(defmethod pdf-given ((var state-distn) x &rest given)
  (apply #'iid-pm x (car given) (cdr given)))
(defmethod cdf-given ((var state-distn) x &rest given)
  (with-slots (param-values) var
    (do ((p param-values (cdr p)) (answer 0 (+ answer (apply
    #'iid-pm (car p) var given))))
((> (car p) x) answer))))
(defparameter *disc-list-exponent* 1)
(defmethod integrate-given ((var discrete-list) fn &rest given)
```

```
  (let ((atoms (slot-value var 'atoms)))
    (do ((atoms atoms (cdr atoms))(answer 0 (+ answer (* (funcall
    fn (car atoms)) (expt (apply #'pdf-given var (car atoms) given)
    *disc-list-exponent*)))))
((null atoms) answer))))

(defun p-theta-theta (theta-n theta-n-1 model)
  (if (< theta-n theta-n-1) 0
      (with-slots (transition param-values) model
(let ((n (position theta-n param-values)) (n-1 (position
theta-n-1 param-values)))
  (if (or (not (and n n-1)) (<= (length transition) (- n n-1))) 0
      (nth (- n n-1) transition))))))

(defun iid-pm-handle-none (theta-n model)
  (with-slots (param-values) model
    (if (= (car param-values) theta-n) 1 0)))

(defun iid-pm-handle-one (theta-n model line)
  (with-slots (transition harmful-distn param-values) model
    (let ((n (length (car line)))(pos (position theta-n
    param-values))(theta-n-1 (car param-values)))
      (cond ((not pos) 0)  ;; can't transition to something not
   ;; allowed?
    ((= 4 n) (if (= theta-n (caar line)) 1 0)) ;; have exact
       ;; knowledge
    ((not (nth pos transition)) 0) ;; almost-surely can't
   ;; transition here
    ((< n 3) (nth pos transition)) ;; the transition vector
   ;; gives us this exact
   ;; answer
    ((= n 3) (let ((num (* (service-pdf model (caar line)
    theta-n theta-n-1)(p-theta-theta theta-n theta-n-1
    model))) (den 0))
       (dolist (theta param-values) (incf
       den (* (service-pdf model (caar line) theta
       theta-n-1)(p-theta-theta theta theta-n-1
       model))))
       (/ num den)))
    (t (error "iid-pm-handle-one something's wrong with our
    line construction"))))))

(defun num-func (theta-n x model)
  #'(lambda (theta-n-1) (* (unified-service-distn model x theta-n theta-n-1)
   (p-theta-theta theta-n theta-n-1 model))))
(defun den-func (theta-n x model)
  #'(lambda (theta-n-1) (unified-service-distn model x theta-n theta-n-1)))
```

```
(defun get-shortened-parameter-list (theta-n model
&optional (look-back (1- (length (slot-value model 'transition)))))
  (with-slots (param-values) model
    (let ((pos (max 0 (- (position theta-n param-values) look-back))))
      (do ((i 0 (1+ i))(answer nil (cons (car upper)
      answer))(upper (nthcdr pos param-values) (cdr upper)))
  ((> i look-back) (nreverse answer))))))

(defun iid-pm-handle-many (theta-n model line)
  (let ((*disc-list-exponent* (slot-value model 'short-term)))
    (if (= 4 (length (car line))) (if (= theta-n (caar line)) 1 0)
(with-slots (state-rv param-values) model
  (setf (slot-value state-rv
  'atoms) (get-shortened-parameter-list theta-n model))
  (let* ((num (exp-val state-rv (num-func theta-n line model)
  model (cdr line)))(den 0))
    (dolist (x param-values)
      (setf (slot-value state-rv
      'atoms) (get-shortened-parameter-list x model))
      (incf den (exp-val state-rv (num-func x line model)
      model (cdr line))))
    (if (= 0 den) 0 (/ num den)))))))

(defun non-zero-length (transition-vector)
  (let ((answer 0))
    (dolist (x transition-vector) (unless (= x 0) (incf answer)))
    answer))

(defun iid-pm-handle-no-info (theta-n model line)
  (with-slots (transition param-values) model
    (if (= 2 (non-zero-length transition))
(pdf (make-instance 'binomial :p (second
transition) :n (length line) :upper (length line)) (position
theta-n param-values))
(let* ((n (length param-values))(exponent (length
line))(matrix (make-transition-matrix transition n)))
  (aref (matrix-exp matrix exponent) 0 (min (1- n) (position
  theta-n param-values)))))))

(defun undo-iid-pm (hash model line) (declare (ignorable hash model line)))
;  (when line
;    (with-slots (param-values) model
;      (dolist (x param-values)
; (multiple-value-bind (val hit) (gethash (list x model line) hash)
;   (declare (ignorable val))
;   (when hit (remhash (list x model line) hash)))))
```

```
;    (undo-iid-pm hash model (cdr line))))

(defun-memo iid-pm #'undo-iid-pm (theta-n model line)
  (cond ((null line) (iid-pm-handle-none theta-n model))
((null (car line)) (iid-pm-handle-no-info theta-n model line))
((null (cdr line)) (iid-pm-handle-one theta-n model line))
(t (iid-pm-handle-many theta-n model line))))

(defmethod arrival-harmful? ((model cost-model-const-transitions))
  (with-slots (transition) model
    (do ((trans transition (cdr trans))(i 0 (1+ i))(u (random 1.0) (-
    u (car trans))))
((or (null trans) (< u 0)) (values (> i 1) (1- i))))))

(defun next-arrival (model &optional (arrival nil) (service-u (random
1.0)) (harmful nil harmful-p))
  (with-slots (transition arrival-distn harmful-distn param-values
  service-distn my-line last-wait last-service) model
    (let ((old-theta (if my-line (caar my-line) (car param-values))))
      (let ((new-theta (cond (harmful-p (next-theta model old-theta
      harmful))
     ((arrival-harmful? model) (progn (setf
     harmful 1) (next-theta model old-theta)))
     (t (progn (setf harmful 0) old-theta)))))
(let (service (arrival (if arrival arrival (sample
arrival-distn))) wait)
  (set-single-parameter-p service-distn new-theta)
  (setf service (sample (nth harmful harmful-distn) service-u))
;   (setf wait (max 0 (+ (if my-line (car (last (car my-line)))
;   last-wait) (if my-line (cadar my-line) last-service) (-
;   arrival))))
  (setf wait (if my-line (max 0 (+ (car (last (car
  my-line))) (cadar my-line) (- arrival))) 0))
  (cons (list new-theta service arrival wait) my-line))))))
```

```
;;;; var-model.lisp
;; this file contains the local linearization (second) model stuff

(defparameter *prior-transition* '(.75 .25))

(defun construct-var-graph (range resets utilities filter
&optional (short-term .8) (measure #'var-pm) (phi-disc-width 1/10))
  (let ((model (make-instance
  'cost-model-estimate-transitions :param-values range :reset-costs
  resets :utilities utilities
      :prob-measure measure :filter
      filter :transition
      *prior-transition* :transition-model
      (make-instance 'bernoulli)
      :short-term short-term :phi-disc-width
      phi-disc-width)))
  (with-slots (prob exp-util policy phi-grid) model
    (unless (= (length range) (length resets) (length
    utilities)) (error "Range, resets, utilities don't have
    the same length"))
    (setf prob (make-list (length range) :initial-element 0))
    (setf exp-util (make-list (length range) :initial-element 0))
    (setf policy (make-list (length range) :initial-element 'stay))
    (setf phi-grid (make-array (list (length range) (/
  phi-disc-width)) :initial-element 0)) model)))

(defun var-pm (theta-n model line)
  (unless (eq theta-n 'undo)
    (when (and line (= 4 (length (car line)))) (error "This method is
    not to be given full information!"))
    (with-slots (phi-grid alpha-cache for-line) model
      (unless (equal for-line line) (fill-likelihood-table model line))
      (let ((theta-idx (car (param2idx model theta-n
      0)))(phi-cap (array-dimension phi-grid 1)))
(let ((answer 0))
  (dotimes (i phi-cap) (incf answer (aref phi-grid theta-idx i)))
  (/ answer alpha-cache))))))

(defun mle-of-phi (model line)
  (with-slots (phi-grid param-values for-line) model
    (unless (equal line for-line) (fill-likelihood-table model line))
    (let ((max -1e10)(max-idx -1)(theta-cap (array-dimension phi-grid 0))
  (phi-cap (array-dimension phi-grid 1))(del-theta (- (cadr
  param-values) (car param-values))))
      (dotimes (i phi-cap)
(let ((sum 0))
  (dotimes (j theta-cap) (incf sum (aref phi-grid j i)))
```

```
  (when (> sum max) (setf max sum max-idx i))))
      (* del-theta (max (/ (- 1 (car *prior-transition*))
      2) (cadr (idx2param model 0 max-idx)))))))

(defmethod set-distributions ((model cost-model-estimate-transitions)
arrival service &rest transition-model)
  (with-slots (arrival-distn service-distn harmful-distn param-values)
  model
    (setf arrival-distn arrival)
    (setf service-distn service)
    (setf harmful-distn (make-list (length
    param-values) :initial-element service))
    (when transition-model (setf (slot-value model
    'transition-model) (car transition-model)))))

(defun calc-transition (model)
  (with-slots (transition-model transition) model
    (with-slots (lower upper) transition-model
      (do ((i lower (1+ i))(answer nil (cons (pdf transition-model i)
      answer)))
  ((> i upper) (setf transition (nreverse answer)))))))

(defun-memo likelihood #'undo-iid-pm (theta-n model line phi)
  (if (null line) (*) ;; that's the multiplicative identity
      (with-slots (short-term param-values) model
(let ((first-param (car param-values)) (del-theta (- (cadr
param-values)(car param-values))))
  (* (unified-service-distn model line
  theta-n) (expt (likelihood (max first-param (- theta-n (*
  del-theta phi))) model (cdr line) phi) short-term))))))

(defun likelihood-one-line (theta-n model line phi)
  (with-slots (param-values transition) model
    (setf transition *prior-transition*)
    (let ((n (length (car line)))(first-param (car
    param-values))(del-theta (- (cadr param-values) (car
    param-values))))
      (let* ((idx-theta-n (position theta-n
      param-values)) (theta-n-transition-prob (if idx-theta-n (nth
      idx-theta-n *prior-transition*) nil)))
(cond ((= 4 n) (error "Perfect into inside var-model"))
      ((= 3 n) (* (service-pdf model (caar line)
      theta-n) (service-pdf model (caar line) (+
      first-param (* phi del-theta)))))
      ((= first-param theta-n) (car *prior-transition*))
      (theta-n-transition-prob theta-n-transition-prob)
      (t 0))))))
```

```
(defun idx2param (model idx jdx)
  (with-slots (phi-disc-width param-values) model
    (list (nth idx param-values) (* jdx phi-disc-width))))

(defun param2idx (model theta phi)
  (with-slots (phi-disc-width param-values) model
    (let ((pos (position theta param-values)))
      (if pos (list pos (round (/ phi phi-disc-width)))
  (let ((closest 1e10))
    (dolist (sheta param-values) (when (< (abs (- sheta
    theta)) (abs (- theta closest))) (setf closest sheta)))
    (list (position closest param-values) (round (/ phi
    phi-disc-width))))))))

(defun fill-likelihood-table (model line)
  (let ((likelihood-fn (cond ((null line) (error "Var-est not meant
  for no information"))
     ((null (cdr line)) #'likelihood-one-line)
     (t #'likelihood))))
    (with-slots (phi-grid alpha-cache for-line) model
      (setf alpha-cache 0 for-line line)
      (let ((theta-cap (array-dimension phi-grid
      0)) (phi-cap (array-dimension phi-grid 1)))
(dotimes (i theta-cap)
  (dotimes (j phi-cap)
    (destructuring-bind (left right) (idx2param model i j)
    (let ((this-val (funcall likelihood-fn left model line right)))
      (incf alpha-cache this-val)
      (setf (aref phi-grid i j) this-val)))))))))

(defun plot-phi-grid (model &key (altitude 30) (azimuth 320) (filename nil))
  (with-slots (phi-grid) model
    (let* ((mesh (cl-plplot:new-3d-mesh nil nil
    phi-grid :contour-options :magnitude-contour))
   (window (cl-plplot:basic-3d-window :altitude
   altitude :azimuth azimuth :x-label "Theta
   Values" :y-label "Phi Values"
     :z-label "Likelihood" :title "Likelihood Function")))
      (cl-plplot:add-plot-to-window window mesh)
      (if filename (cl-plplot:render
      window "png" :filename (concatenate 'string filename ".png"))
  (cl-plplot:render window "xwin")))))

(defclass mmpp () ;; this class models the state of our bursty arrivals
  ((n :initform 2 :initarg :n :documentation "This is the number of
  states we have")
```

```
   (state :initform (if (< (random 10.0) 2) 0 1) :documentation "This
   is the state we're in from 0..n-1")
  (exp-durations :initform '(1/2
  1/8) :initarg :durations :documentation "This is the inverse of the
  exp-amount of time spent in each state")
   (rates :initform '(3/4 1/8) :initarg :rates :documentation "This is
   the rate of the point process given which state we're in")))

(defmacro wp (p &body body)
  `(when (< (random 1.0) ,p) . ,body))

(defmethod arrival-harmful? ((model cost-model-estimate-transitions))
  (with-slots (mmpp) model
    (with-slots (n state exp-durations rates) mmpp
      (wp (nth state exp-durations) (setf state (mod (1+ state) n)))
      (let ((harmful? (< (random 1.0) (nth state rates))))
(values harmful? (if harmful? 1 0))))))

(defun endow-model-with-mmpp (model)
    (setf (slot-value model 'mmpp) (make-instance 'mmpp)))

(defun find-rest (elt sequence &optional (test #'=))
  (do ((seq sequence (cdr seq)))
      ((or (null seq) (funcall test (car seq) elt)) seq)))

(defun next-theta (model theta &optional (times 1))
  (with-slots (param-values) model
    (let* ((old (find-rest theta param-values))(new old))
      (if (null old) (error "next-theta - Theta not found!!!")
  (dotimes (i times) (when (cdr new) (setf new (cdr new)))))
      (car new))))
```

```
;;;; model-construction.lisp
;; This file constructs one example for comparison of the policies

(defun construct-8-policies (range resets utilities transition
transition-model)
  (let ((answer nil)(short-term .8)(modelid (serialize-scenario-pt1
  range resets utilities transition transition-model))
(info-levels `((,#'identity "#'identity" "Everything")
       (,#'cddddr "#'cddddr" "Nothing")
       (,#'cdr "#'cdr" "Service Times")
       (,#'cddr "#'cddr" "Arrival
       Times") (,#'cdddr "#'cdddr" "Waiting Times"))))
    (declare (ignore short-term))
;    (dolist (x info-levels) ;; add iid-version with total-long-term
;      memory (push (construct-const-graph range resets utilities
;      (first x) transition) answer) (setf (slot-value (car answer)
;      'desc) (concatenate 'string "IID(1) - " (third x) " Provided"))
;      (serialize-policy (car answer) (second x) modelid)) (setf
;      *iid-line* (car answer)) (dolist (x (cddr info-levels)) ;; add
;      iid-version with short-term memory
    (let ((x (caddr info-levels)))
      (dotimes (shorty 100)
(push (construct-const-graph range resets utilities (first x)
transition) answer)
(setf (slot-value (car answer) 'desc) (concatenate
'string "IID(short) - " (third x) " Provided"))
(setf (slot-value (car answer) 'short-term) (/ shorty 100))
(serialize-policy (car answer) (second x) modelid)))
;    (dolist (x (cddr info-levels)) ;; the estimated method doesn't
;      cope with perfect (or no) information (let ((model
;      (construct-var-graph range resets utilities (first x)))) (setf
;      (slot-value model 'transition-model) transition-model) (setf
;      (slot-value model 'desc) (concatenate 'string "Est - " (third
;      x) " Provided")) (setf (slot-value model 'short-term)
;      short-term) (push model answer) (serialize-policy model (second
;      x) modelid))) (setf *var-line* (car answer))
;      (endow-model-with-mmpp *var-line*)
    answer))

   ;; sharing these distributions isn't a problem, they don't contain state.
(defun set-8-distributions (8policies arrival service)
  (dolist (x 8policies) (set-distributions x arrival service))
  (let ((model-id (get-model-id (car 8policies))))
    (serialize-scenario-pt2 model-id arrival service)))

(defun clear-8-policies (8policies)
  (dolist (x 8policies)
```

```
    (clear-state x)
    (format t "Clearing: ~A~%" (slot-value x 'desc))))

(defun linear-range (lower upper step)
  (let ((lower (rationalize lower)) (upper (rationalize
  upper)) (step (rationalize step)))
    (do ((val (if (> step 0) lower upper) (+ val step)) (answer
    nil (cons val answer)))
((if (> step 0) (> val upper) (< val lower)) (nreverse answer)))))

(defun const-adder (n)
  #'(lambda () n))
(defun linear-adder (a b)
  (decf a b)
  #'(lambda () (incf a b)))
(defun piecewise-adder (left after right)
  #'(lambda () (if (>= (decf after) 0) (funcall left) (funcall right))))
(defun times (this n)
  (let ((answer nil))
    (dotimes (i n) (push this answer))
    answer))

(defun simple-half-reset-costs (range)
  (let* ((n (length range))(adder (piecewise-adder (const-adder
  -1) (floor (/ n 2)) (linear-adder -1 -1/5))))
    (do ((i 1 (1+ i))(answer nil (cons (funcall adder) answer)))
((> i n) (nreverse answer)))))

(defun simple-utilities-based-on-rho (range arrival-distn
service-distn earn-per-cust cost-per-hour-busy
&optional (cost-per-hour-idle cost-per-hour-busy))
  (let* ((exp-arrival (mean arrival-distn)))
    (do ((answer nil)(param range (cdr param)))
((null param) (nreverse answer))
      (set-single-parameter-p service-distn (car param))
      (let* ((exp-service (mean service-distn))(rho (/ exp-service
      exp-arrival))(1-rho (- 1 rho)))
(if (>= exp-service exp-arrival)
    (push (- (/ earn-per-cust exp-service) cost-per-hour-busy)
    answer)
    (push (- (/ earn-per-cust exp-arrival) (* rho
    cost-per-hour-busy) (* 1-rho cost-per-hour-idle))
    answer))))))


(defparameter *range* (linear-range 1/10 9/10 1/40))
(defparameter *reset* (simple-half-reset-costs *range*))
```

```
(defparameter *arrival-distn* (make-instance 'geometric :p 1/4))
(defparameter *service-distn* (make-instance 'binomial :upper 6 :n 6))
(defparameter *utils* (simple-utilities-based-on-rho *range*
*arrival-distn* *service-distn* 10 2))
(defparameter *var-line* nil)
(defparameter *iid-line* nil)
(defparameter *8policies* nil)

(defun prep8 ()
  (setf *8policies* (construct-8-policies *range* *reset* *utils*
  '(.75 .25) (make-instance 'bernoulli)))
  (set-8-distributions *8policies* *arrival-distn* *service-distn*))
```

```
;;;; process.lisp
;; this file is the big loop of the simulation

(defun process-n-arrivals (n line-model-in-db &optional (policy-list
*8policies*))
  (clear-list-accounting policy-list)
  (let ((i 0))
    (dolist (line-input (get-input-to-process-arrival line-model-in-db n))
      (incf i)
      (dolist (x policy-list)
(apply #'process-arrival x line-input)))))

(defun clear-list-accounting (list &optional (key #'identity))
  (dolist (x list) (clear-accounting (funcall key x))))

(defparameter *to-do* nil)
(defparameter *done* nil)

(defun prep-batch ()
  (prep8)
  (setf *to-do* *8policies*)
  (setf *8policies* nil *done* nil))

(defun prep-next (n)
  (dolist (x *8policies*) (push x *done*))
  (setf *8policies* nil)
  (dotimes (i n) (when *to-do* (push (pop *to-do*) *8policies*))))

(defun do-it-all (n &optional (model 1))
  (prep8)
  (let ((list (nreverse *8policies*)))
    (dolist (x list)
      (setf *8policies* (list x))
      (format t "Beginning policy: ~A~%" (slot-value x 'desc))
      (process-n-arrivals n model))))
```

```
;;;; stats.lisp
;; this file contains a lot of misc functions for generating statistics
;; a lot of the plotting software calls happen in here

(defun summarize (model txt)
  (with-slots (accounting) model
    (with-slots (earn-per-cycle wait-times service-times arrival-times
    reset-periods) accounting
      (format t "~A~A~%" (make-array '(15) :element-type
      'character :initial-element #\-) txt)
      (format t " -- Earnings per cycle: ~A~%" earn-per-cycle)
      (format t " -- Waiting times : ~A~%"  wait-times)
      (format t " -- Service times : ~A~%" service-times))))

(defun sum8 ()
  (dolist (x *8policies*)
    (summarize x (slot-value x 'desc))))

;;---------------------------------------------

(defun sample-mean (seq)
  (let ((sum (apply #'+ seq)) (n (length seq)))
    (if (= n 0) 0
(/ sum n))))

(defun sq (x) (* x x))

(defun sample-variance (seq &optional (mean nil))
  (let ((sample-mean (if mean mean (sample-mean seq)))(n-1 (if
  mean (length seq) (1- (length seq)))))
    (if (= n-1 0) 0
(/ (apply #'+ (mapcar #'(lambda (x) (sq (- x sample-mean)))
seq)) n-1))))

(defun sample-stddev (seq &optional (mean nil))
  (if (and seq (cdr seq)) (sqrt (sample-variance seq mean)) 0))


(defun sample-covariance (seq1 seq2)
  (unless (= (length seq1) (length seq2)) (error "Must redu covariance
  - counts don't match"))
  (let ((mean1 (sample-mean seq1)) (mean2 (sample-mean
  seq2)) (n-1 (1- (length seq1))))
    (if (= n-1 0) 0
(/ (apply #'+ (mapcar #'* (mapcar #'(lambda (x) (- x mean1))
seq1) (mapcar #'(lambda (x) (- x mean2)) seq2))) n-1))))
```

```
(defun sample-correlation (seq1 seq2)
  (let ((cov (sample-covariance seq1 seq2))(std1 (sample-stddev
  seq1))(std2 (sample-stddev seq2)))
    (if (and cov std1 std2) (/ cov std1 std2) 0)))


    ;; ----------------------------------------

(defun count-t (seq)
  (let ((answer 0))
    (dolist (x seq) (when x (incf answer)))
    answer))

(defparameter *stat-action-list* nil)

(defmacro symbol->action (symbol action) `(push (cons ,symbol ,action)
*stat-action-list*))

(setf *stat-action-list* nil)
     ;; sum earnings within a cycle
(symbol->action 'earn-per-cycle #'(lambda (x) (apply #'+ x)))
(symbol->action 'earn-per-cycle #'sample-mean)
(symbol->action 'harmful-jobs #'count-t) ;; sum the t's in a list of
 ;; t's and nil's
(symbol->action 'wait-times #'sample-mean)
(symbol->action 'wait-times #'sample-stddev)
(symbol->action 'service-times #'sample-mean)
(symbol->action 'service-times #'sample-stddev)
(symbol->action 'arrival-times #'sample-mean)
(symbol->action 'arrival-times #'sample-stddev)
(symbol->action 'reset-periods #'length)

(defun get-cov-entry (model elt1 elt2)
  (with-slots (accounting) model
    (let ((seq1 (mapcar (cdr elt1) (slot-value accounting (car elt1))))
  (seq2 (mapcar (cdr elt2) (slot-value accounting (car elt2)))))
      (float (sample-correlation seq1 seq2)))))

(defun mk-cov-matrix (model)
  (declare (ignorable model))
  (let* ((n (length *stat-action-list*))(answer (make-array (list n
  n) :initial-element 0)))
    (do ((elt1 *stat-action-list* (cdr elt1))(i 0 (1+ i)))
((null elt1) answer)
      (do ((elt2 elt1 (cdr elt2))(j i (1+ j)))
  ((null elt2))
(setf (aref answer j i) (setf (aref answer i j) (get-cov-entry
```

```
model (car elt1) (car elt2))))))))

;(defun plot-cov-matrix (model &key (altitude 30) (azimuth 60)
;  (filename nil)) (let* ((cov-matrix (mk-cov-matrix model)) (c
;  (cl-plplot:new-surface-plot nil nil cov-matrix :line-color :blue))
;  (w (cl-plplot:basic-3d-window :altitude altitude :azimuth
;  azimuth))) (cl-plplot:add-plot-to-window w c)
;  (cl-plplot:add-text-label-to-window w (cl-plplot:new-text-label
;  (cl-plplot:new-text-item "Window Title") 0 0)) (if filename
;  (cl-plplot:render w "png" :filename filename) (cl-plplot:render w
;  "xwin"))))

(defun l2a (list)
  (let ((answer (make-array (list (length list)))))
    (do ((i 0 (1+ i))(list list (cdr list)))
((null list) answer)
      (setf (aref answer i) (car list)))))

;(defun plot-these (&rest rest) (let ((plots nil)(colors '(:red :blue
;  :grey :black :violet :orange :green :yellow :brown)) (w
;  (cl-plplot:basic-window))) (do ((p rest (cdr p))(color colors (cdr
;  color))(i 0 (1+ i))) ((null p)) (push (cl-plplot:new-x-y-plot (l2a
;  (caar p)) (l2a (cdar p)) :line-width 3 :color (car color)
;  :line-style i :x-error #(.3 .5 .9) :y-error #(.3 .5 .6)) plots))
;  (dolist (x plots) (cl-plplot:add-plot-to-window w x))
;  (cl-plplot:render w "xwin")))


;-------------------------- PLOTTING COMPARISONS ----------------------

(defun plot-earn-vs-column (policyid column &optional (filename nil))
  (let ((p (get-graph policyid column :color :blue))
(w (cl-plplot:basic-window)))
    (cl-plplot:add-plot-to-window w p)
    (cl-plplot:render w "xwin")
    (setf w (cl-plplot:basic-window))
    (cl-plplot:add-plot-to-window w p)
    (if filename (cl-plplot:render w "png" :filename filename)
(cl-plplot:render w "xwin"))))

(defun compare-plots (policyid column &optional (filename nil))
  (let* ((data (get-data policyid column))
 (smooth (construct-array policyid column))
 (p-d (cl-plplot:new-x-y-plot (to-array (mapcar #'car
 data)) (to-array (mapcar #'cadr data)) :symbol-size 1.0))
 (p-s (cl-plplot:new-x-y-plot (to-array (mapcar #'car
 smooth)) (to-array (mapcar #'cadr
```

```
 smooth)) :color :blue :line-style 4))
 (w (cl-plplot:basic-window)))
    (cl-plplot:add-plot-to-window w p-d)
    (cl-plplot:add-plot-to-window w p-s)
    (if filename (cl-plplot:render w "png" :filename filename)
(cl-plplot:render w "xwin"))))

(defun get-graph (policyid column &rest options)
  (let ((data (construct-array policyid column)))
    (apply #'get-graph-from-data data options)))

(defun get-graph-from-data (data &rest options)
  (apply #'cl-plplot:new-x-y-plot (to-array (mapcar #'car
  data)) (to-array (mapcar #'cadr data)):copy t options))

(defparameter *indep-vars* '(s_harmful s_service s_arrival s_wait))
(defparameter *colors* (list :black :red :green :blue :orange :purple))
;(defparameter *iid-measure-comp* '(815 816 34 35))
;(defparameter *iid-service-comp* '(815 817 798 801))
;(defparameter *iid-arrival-comp* '(815 796 799 802))
;(defparameter *iid-waiting-comp* '(815 797 800 803))
;(defparameter *var-service-comp* '(34 36 39 42))
;(defparameter *var-arrival-comp* '(34 37 40 43))
;(defparameter *var-waiting-comp* '(34 38 41 44))
;(defparameter *petes-question-var* '(34 36 37 38 35))

(defun 4policy-1indep (policy-list column)
  (let ((plots nil))
    (dolist (x policy-list)
      (push (construct-array x column) plots))
    (plot-and-save (nreverse plots))))

(defun 1policy-4indep (policyid &optional (indep-list *indep-vars*))
  (let ((plots nil))
    (dolist (x indep-list)
      (push (construct-array policyid x) plots))
    (plot-and-save (nreverse plots))))

(defun plot-and-save (plots)
  (cl-plplot:render (plots-from-data plots) "xwin")
  (format t "Would you like to save to a file?  Nill for No.  Surround
  with quotes!")
  (let ((filename (read)))
    (when (and filename (not (symbolp
    filename))) (cl-plplot:render (plots-from-data
    plots) "png" :filename filename))))
```

```
(defun plots-from-data (data)
  (let ((w (cl-plplot:basic-window)))
    (do ((i 1 (1+ i))(data data (cdr data))(colors *colors* (cdr colors)))
((null data) w)
      (cl-plplot:add-plot-to-window w (get-graph-from-data (car
      data) :color (car colors) :line-style i)))))

(defun plot-histogram (column &optional filename)
  (let* ((w (cl-plplot:basic-window))
 (x (apply #'cl-plplot:new-x-y-plot (construct-histogram column))))
    (cl-plplot:add-plot-to-window w x)
    (if filename (cl-plplot:render w "png" :filename filename)
(cl-plplot:render w "xwin"))))


;---------------------------------- TOTAL EARNINGS


(defun get-earnings-for-first-n (policy embed n &key (key
#'identity) (column 'earn))
  (let ((table (access-data policy embed `(length
  ,column)))(total-earned 0.0))
    (do ((i 0)(row table (cdr row)))
((or (null row) (>= i n)) (values total-earned i))
      (let ((length (first (car
      row)))(earn-list (read-from-string (second (car row)))))
(if (<= (+ i length) n)
    (progn (incf i length)
   (incf total-earned (reduce #'+ earn-list :key key)))
    (let* ((need (- n i))(reversed (nreverse earn-list)))
      (setf i n)
      (do ((j 0 (1+ j))(data reversed (cdr data)))
  ((= j need))
(incf total-earned (funcall key (car data)))))))))

(defun report-total-earnings (&key (n 10000) (max-embed 2) (key
#'identity) (column 'earn))
  (format t " ~15f  |   Description of the policy~%" "Total Earnings")
  (format t "----------|-------------------------------------------~%")
  (let ((answer nil))
    (dotimes (embed max-embed)
      (dotimes (policy 11)
(push (list (get-earnings-for-first-n (1+ policy) (1+ embed)
n :key key :column column) (access-description (1+ policy) (1+
embed))) answer)))
    (dolist (x (sort answer #'> :key #'first))
      (format t "~15f   |   ~a~%" (car x) (cdr x))))
```

```
  (format t "-----------|-----------------------------------------~%"))

;(defun t-to-1 (list)
;  (let ((answer nil))
;    (dolist (x list) (push (if x 1 0) answer))
;    (nreverse answer)))

(defun t->1 (elt)
  (if elt 1 0))

(defun avg-earn-given-harm-count (policy embed harm-count)
  (let ((earn-total 0.0) (num-arrivals 0))
    (dolist (x (clsql:query (format nil "select earn_count,length from
    DATA where model = 1 and
    harm_count = ~A and policy = ~A
    and embed=~A" harm-count policy
    embed)))
      (let ((earn-count (first x)) (length (second x)))
(incf num-arrivals length)
(incf earn-total earn-count)))
    (if (> num-arrivals 0) (/ earn-total num-arrivals) 0)))

(defun histogram-harm-count (embed)
  (clsql:query (format nil "select harm_count,count(*) from DATA where
  model=1 and embed = ~A group by harm_count order by harm_count"
  embed)))

(defun plot-histogram-harm-count (embed &key filename)
  (let ((data (histogram-harm-count embed))
(w (cl-plplot:basic-window)))
    (cl-plplot:add-plot-to-window w (cl-plplot:new-bar-graph ;(to-array
     ;(mapcar
     ;#'car
     ;data))
     nil (to-array (mapcar #'second
     data)) :fill-colors (vector :grey)))
    (if filename (cl-plplot:render w "png" :filename filename)
(cl-plplot:render w "xwin"))))

(defun construct-vector (policy embed)
  (let* ((max (1+ (caar (clsql:query "select max(harm_count) from DATA
  where model = 1")))(answer (make-array max)))
    (dotimes (i max)
      (setf (aref answer i) (avg-earn-given-harm-count policy embed i)))
    answer))

(defparameter *petes-question* '((1 1)(3 1)(4 1)(5 1)(2 1)))
```

```
(defparameter *colors* (list :black :red :green :blue :grey))

(defun get-list-4-plot-avg-earnings (policy/embed-list color-list)
  (do ((x policy/embed-list (cdr x))(colors color-list (cdr colors))
       (answer nil (cons (cl-plplot:new-bar-graph
       nil (construct-vector (first (car x)) (second (car
       x))) :fill-colors (vector (car colors))) answer)))
      ((or (null x) (null colors)) (nreverse answer))))

(defun plot-avg-earnings (policy/embed-list &optional filename)
  (let ((w (cl-plplot:basic-window))(plot-list (get-list-4-plot-avg-earnings
  policy/embed-list *colors*)))
    (dolist (x plot-list) (cl-plplot:add-plot-to-window w x))
    (dolist (x plot-list) (cl-plplot:bring-to-front w x))
    (cl-plplot:bring-to-front w (nth 3 plot-list))
    (if filename (cl-plplot:render w "png" :filename filename)
(cl-plplot:render w "xwin"))))

;---------------- again but using x-y plots --------------------

(defun construct-harm-vectors (policy embed &optional (n 1000))
  (let* ((result (clsql:query (format nil "select
  min(harm_count),max(harm_count) from DATA where policy = ~A and
  embed = ~A and model = 1" policy embed)))
 (min (max 12 (caar result))) (max (min 28 (cadar
 result))) (h (/ (- max min) (1- n)))(points nil))
    (do ((i min (1+ i)))((> i max))
      (push (list i (avg-earn-given-harm-count policy embed i)) points))
    (let ((x-vec (make-array n))(y-vec (make-array n)))
      (dotimes (i n)
(setf (aref x-vec i) (+ min (* i h)))
(setf (aref y-vec i) (gauss-blur (+ min (* i h)) points 1.5)))
      (values y-vec x-vec))))

(defun plot-smooth-averages (policy/embed-list &key (colors *colors*)
filename)
  (let ((w (cl-plplot:basic-window)))
    (do ((x policy/embed-list (cdr x))(colors colors (cdr colors)))
((or (null x) (null colors)))
      (multiple-value-bind (y-vec
      x-vec) (construct-harm-vectors (first (car x)) (second (car x)))
(cl-plplot:add-plot-to-window w (cl-plplot:new-x-y-plot x-vec
y-vec :color (car colors)))))
    (if filename (cl-plplot:render w "png" :filename filename)
(cl-plplot:render w "xwin"))))

;;;; This is the plot showing which gamma is the best
```

```
(defun examine-summary3 (&key (filename nil)(smooth nil) (radius 0.05))
  (let ((summary3 (clsql:query "select * from summary3")))
    (dolist (x summary3) (setf (car x) (read-from-string (car x))))
    (let ((smoothed-summary3 (copy-list summary3)))
      (when smooth (dolist (x smoothed-summary3)
     (setf (cadr x) (gauss-blur (car x) summary3
     radius))))
      (let ((w (cl-plplot:basic-window :x-label "Gamma" :y-label
       "Pct of Perfect" :title "Gamma vs PoP"))
    (g (cl-plplot:new-x-y-plot (to-array (mapcar #'car
    smoothed-summary3)) (to-array (mapcar #'cadr
    smoothed-summary3)))))
   (cl-plplot:add-plot-to-window w g)
      (if filename (cl-plplot:render w "png" :filename filename)
  (cl-plplot:render w "xwin"))))))
```

Bibliography

[1] P. Brémaud. *Markov Chains, Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer, 1999.

[2] D. Cox and W. Smith. *Queues*. Spottiswoode, Ballantyne & Co. Ltd., 1961.

[3] A. Duda. Diffusion approximations for time-dependent queuing systems. *IEEE Journal on Selected Areas in Communications*, SAC-4(6):905–918, Sept. 1986.

[4] L. Green et al. Some effects of nonstationarity on multiserver markovian queueing systems. *Operations Research*, 39(3):502–511, May - Jun. 1991.

[5] E. Gelenbe and C. Rosenberg. Queues with slowly varying arrival and service processes. *Management Science*, 36(8):928–937, Aug. 1990.

[6] M. Neuts. A queue subject to extraneous phase changes. *Advances in Applied Probability*, 3(1):78–119, Spring 1971.

[7] G. Newell. Queues with time-dependent arrival rates i: The transition through saturation. *Journal of Applied Probability*, 5(2):436–451, Aug. 1968.

[8] S. Resnick. *Adventures in Stochastic Processes*. Birkhäuser, 2005.

[9] T. Rolski. Upper bounds for single server queues with doubly stochastic poisson arrivals. *Mathematics of Operations Research*, 11(3):442–450, Aug. 1986.

[10] S. Ross. *Stochastic Processes*. John Wiley and Sons, Inc., 1996.

[11] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Inc, 2003.

[12] U. Yechiali and P. Naor. Queuing problems with heterogeneous arrivals and service. *Operations Research*, 19(3):722–734, May - Jun. 1971.