ABSTRACT

Title of Document:	EMERGENCY DEPARTMENT EFFICIENCY IN AN ACADEMIC HOSPITAL: A SIMULATION STUDY
	Katie Johnson, Daniel Kalowitz, Jay Kellegrew, Benjamin Kubic, Joseph Lim, John Silberholz, Alex Simpson, Emily Sze, Ekta Taneja, Edward Tao
Directed By:	Professor & France-Merrick Chair, Dr. Bruce L. Golden Decisions, Operations, and Information Technology R.H. Smith School of Business, University of Maryland

This study examined the effects of the resident education model on the efficiency of a teaching hospital emergency department. Patient data was collected from the University of Maryland Medical Center in Baltimore, MD. This data consisted of both patient information physically collected in the emergency department, as well as historical patient information accessed through the hospital's electronic databases. Simulation modeling was then used to analyze in a statistically significant manner the effects of the resident education model on patient throughput in the emergency department. We determined that the presence of residents in the ED improves patient throughput for both high-priority and low-priority patients. However, this improvement is higher for low-priority patients than for high-priority patients, which is a novel result. Future studies will entail determining how replacing residents with other types of personnel, such as nurse practitioners or other types of physicians, affects patient throughput.

EMERGENCY DEPARTMENT EFFICIENCY IN AN ACADEMIC HOSPITAL: A SIMULATION STUDY

Team HOPE: Team Hospital Optimal Productivity Enterprise

Authors: Katie Johnson, Daniel Kalowitz, Jay Kellegrew, Benjamin Kubic, Joseph Lim,
John Silberholz, Alex Simpson, Emily Sze, Ekta Taneja, Edward Tao

Thesis Submitted in fulfillment of the requirements of the Gemstone Program
University of Maryland, College Park
2010

Advisory Committee:
Dr. Bruce Golden, PhD, Mentor
Dr. Jon Mark Hirshon, M.D., Discussant
Mr. Michael Harrington, Discussant
Dr. Carter Price, PhD, Discussant
Mr. David Anderson, Discussant
Mr. Sean Barnes, Discussant
Mr. William Herring, Discussant

Acknowledgements

Team HOPE would like to thank a number of individuals and organizations for their contributions to our project; without their help, the successful completion of our study would not have been possible. Dr. Bruce Golden played a vital role in connecting us with the University of Maryland Medical Center (UMMC) where our research was conducted, and as our team mentor also provided us with valuable advice and guidance regarding our research topic. We would also like to acknowledge Mr. Mike Harrington, Dr. Jon Mark Hirshon, Ms. Gail Brandt, and the Emergency Department staff at UMMC that were kind enough to assist us in our data collection. Other individuals who contributed to our project include past team members Jon Anderson, Stanley He, Julie Markowitz, and Esther Yang, as well as our staff of UMD student volunteers who assisted in the data collection process. Finally, we would like to thank Peer Advantage Tutors and Lockheed Martin, whose financial assistance was greatly appreciated.

Table of Contents

List of Tables

List of Figures

1. Introduction

The cost of health care is a significant social concern in America today. Associated costs in the health care system have been on the rise for the past two decades. According to the Centers for Medicare and Medicaid Services (CMMS), in 2007, the total National Health Expenditure (NHE) reached $2.2 trillion, which equates to approximately $7,421 per person. This expenditure accounted for 16.2 percent of the Gross Domestic Product (GDP). The 2007 total was over three times as much as the $714 billion spent on health care in 1990 (CMMS, 2009).

These costs are expected to continue growing over the next decade. The CMMS projects a 6.1 percent growth in costs in 2008, resulting in total expenditures of $2.4 trillion; the average annual growth rate thereafter is estimated at 6.2 percent per year. From 2008-2018, annual growth in health spending is expected to exceed the economy's annual growth by 2.1 percentage points yearly. By the end of the projection period in 2018, health care spending is expected to constitute approximately one-fifth of U.S. GDP at $4.4 trillion. While there are many components of these costs, hospitals contributed the largest amount in 2007, costing consumers 31.7 percent of the total health-care expenditures (CMMS, 2009).

The Resident Model

There are approximately 300 million hospital visits across the nation each year (Brown, 2008). Of these, 119.2 million are to an emergency department (ED), which amounts to 227 people visiting an ED each minute. The average wait time in the nation's EDs in 2008 was four hours and three minutes, down two minutes from 2007. However, the national average has increased by 27 minutes since 2002, indicating an increasing trend in inefficiency (Press-Ganey, 2009).

The average person spends 185 minutes in the ED. If he or she is admitted to the hospital, the time spent increases to 265 minutes. The best 10 percent of hospitals have an average time spent in the ED of around 120 minutes, with 214 minutes for admitted patients. Though the hospitals' patient distributions may vary significantly, the difference in averages suggests a potential for improvement in patient length of stay (Advisory Board, 2008).

Longer ED wait times may result from a number of bottlenecks, such as lab and radiology test processing times, a dearth of inpatient beds available for ED patients, and lengthy triage systems, among other factors (Advisory Board, 2008). However, research hospitals have a unique system in place that plays an integral role in patient throughput and care: the resident teaching model. A medical student who graduates with a medical degree (M.D.) must complete three to seven years of medical practice as a resident physician under the guidance of a senior physician, called an attending physician. Residencies can be completed in general medicine or any specialty field within medicine. The purpose of the teaching system is to ensure that medical institutions produce physicians with the essential skills and experience necessary to provide quality patient care in a health care environment. Upon successful completion of residency and the United States Medical Licensing Examinations (USMLEs), a doctor is then considered an attending physician (Cooke, Irby, Sullivan, & Ludmerer, 2006).

An integral aspect of the teaching model is the inherent hierarchy that exists within every department staffed with residents. Before residency, third- and fourth-year medical students must rotate through departments for several weeks at a time. They are allowed to visit patients with senior physicians and residents, but have minimal direct influence on the care given to the patient. At the next level, residents visit patients, perform physical examinations, and are often responsible for determining which lab tests and medications should be ordered for a given patient.

Even though the first few medical students and residents who see the patient do not always directly contribute to the patient's treatment, graduate medical education (GME) guidelines mandate diverse exposure to ensure these doctors receive proper training towards the achievement of their medical degrees and licenses (AAMC, 2003). However, their every assessment must be given final approval by the floor's attending physician, introducing an element of inefficiency by lengthening the patient care process (Harvey, Shaar, Cave, Wallace, & Byrdon, 2008). Previous studies examining the teaching system have also demonstrated a multitude of one- to five-minute actions residents must perform which fragment their work processes (Dassinger, Eubanks, & Langham, 2008), and that residents exhibit poor time management when faced with increased patient densities (Shayne, Lin, Ufberg, Ankel, & Barringer,2009).

We chose to focus on the residency model of care as a significant area of inefficiency in research hospitals' emergency departments. Specifically, our team decided to create a simulation model of the ED at the University of Maryland Medical Center (UMMC) in Baltimore. Though numerous studies have been conducted individually on both the resident teaching system and on simulation models of various hospital departments, our study is unique in creating a model of the teaching system to test scenarios involving differing levels of resident care.

University of Maryland Medical Center (UMMC)

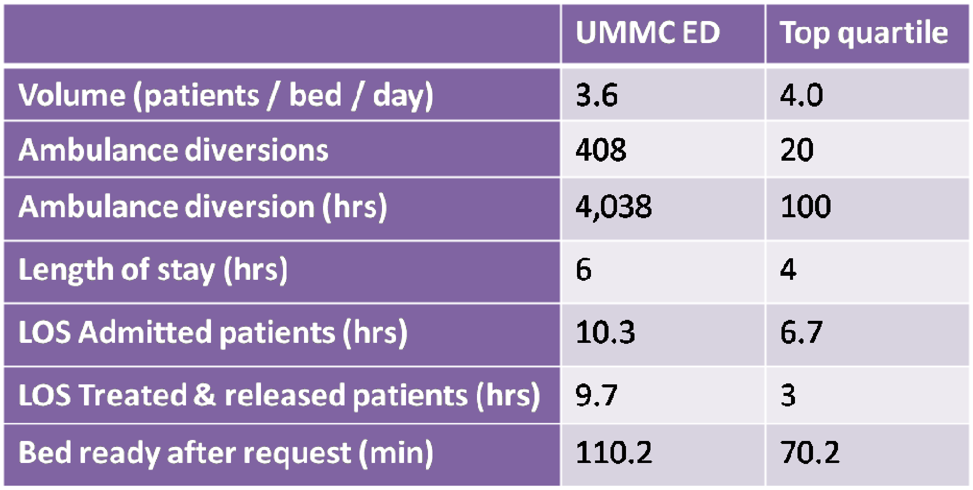
The University of Maryland Medical Center (UMMC) is a teaching hospital located in Baltimore, Maryland that incorporates the residency model across all of its departments. It is one of 18 hospitals located in the region, housing 800 beds staffed by 1,182 doctors. Since it is a teaching hospital, it also has 742 residents and fellows. Its emergency department has 55 beds, and with a 20 percent admission rate the ED sees approximately 46,000 adult patients each year (UMMC, 2007).

According to a 2006 report by the University HealthSystem Consortium (UHC), UMMC's emergency department falls well behind the top quartile of hospital EDs across numerous performance metrics. The UHC conducted a study to assist hospital organizations in identifying their strengths and weaknesses in achieving efficient patient flow and maximum capacity. Regarding ED volume, the hospital sees 3.6 patients per bed per day, compared to the top quartile's 4.0, a 10 percent deficiency (HealthSystem Consortium, 2006).

UMMC	UMMC ED
• 800 beds • 1,182 doctors • 742 residents & fellows	• 55 beds • 20% admission rate • 46,000 adult patients/year

Table 1: University of Maryland Medical Center Statistics (HealthSystem Consortium, 2006)

The UMMC'S ED performance measures were significantly short of the top quartile in a number of areas. The ED was on divert status for ambulances for 4,038 hours annually with 408 diversions occurring each year, compared to the top quartile's 100 hours and 20 diversions. The emergency department also falls behind in one of the most important metrics concerning patient care – wait time. The UMMC ED's overall length of stay (LOS) is six hours, compared to the optimal four hours. The ED LOS is 10.3 hours for admitted patients and 9.7 hours for treated and released patients, versus the optimal 6.7 hours and 3 hours, respectively. Fifteen percent of patients leave the UMMC ED without being seen, which is two percent more than the top quartile emergency departments. The UHC study rated the UMMC ED as "worse than target," defined as 51st to 89th percentile, or "substantially worse than target," defined as the bottom decile, in all but two performance measures, indicating a significant need for improvement (HealthSystem Consortium, 2006).



	UMMC ED	Top quartile
Volume (patients / bed / day)	3.6	4.0
Ambulance diversions	408	20
Ambulance diversion (hrs)	4,038	100
Length of stay (hrs)	6	4
LOS Admitted patients (hrs)	10.3	6.7
LOS Treated & released patients (hrs)	9.7	3
Bed ready after request (min)	110.2	70.2

Table 2: National ED Comparison Statistics (HealthSystem Consortium, 2006)

Focus of Research

Our research seeks to improve patient throughput in research hospital emergency departments. In particular, our team modeled the ED at the UMMC. Based on the results of computer simulations using that model, we will suggest novel improvements that could increase the efficiency and profitability not only of the UMMC but also of similar large-scale hospitals across the nation and globally. We are seeking to determine whether the resident model of emergency department care decreases overall ED throughput.

Our team collected data about the current operations of the UMMC's ED and created a computer simulation model to capture these operations. We modeled the attending and resident physicians' decision-making process using the data we collected, and modeled historical ED operations using historical database data. The model was then validated by comparing the simulation's outputs to actual department metrics. Once the model was validated, we ran experiments in the model by changing the types of doctors who examined certain patients as well as the order in which they examined the patients, then tested which ones demonstrated statistically significant improvements in patient throughput. We will propose the viable solutions to Dr. Jon Mark Hirshon, an Associate Professor in the Department of Emergency Medicine at the UMMC who has been a close collaborator of this study, to determine which solutions can be realistically implemented in the ED.

Our team first conducted a literature review on previous research investigating the residency model used in teaching hospitals, and on past simulation research done on various hospital departments. The team also visited the UMMC ED to gather observational data relevant to coding the simulation model, and to collect data relevant to the teaching system in place at the hospital's ED. The manually collected data was used in conjunction with data mined from the hospital's historical databases to code and verify a simulation model of the ED. The team then tested proposed solutions for improving the residency system through the simulation, and analyzed the simulation results to determine whether any of the suggested solutions produced statistically significant improvements in the ED's functionality.

2. Literature Review

Several resources in the hospital impact the functioning of the ED, and thus may impact any effects the resident education model potentially has on the functioning of the ED. Human interactions, including staffing and communication issues, as well as the physical area of the ED and its equipment, have implications in disrupting the maximal efficiency of the ED and its overall patient throughput. Given that the purpose of simulation modeling is to focus only on variables that are significant, we have to understand the effects of such resources and how they could potentially affect our model.

Communication

Communication plays a crucial role in the decisions residents make in the ED. A 2006 study performed by Hanada, Fujiki, Nakakuni, & Sullivan (2006) concluded that inter-hospital communication is vital to success in the emergency department. Furthermore, communication failures in the ED are a prime cause of incidents concerning patient safety (Featherstone, Chalmers, & Smith, 2008). A patient's personal and medical information is in a continuous state of change and is constantly being exchanged between nurses and doctors through patient databases. Improved communication in the ED may increase efficiency and decrease the amount of time each patient spends in the ED (Clark, 2005). Efficiency in the ED relies on excellent verbal communication as well as technology-based communication.

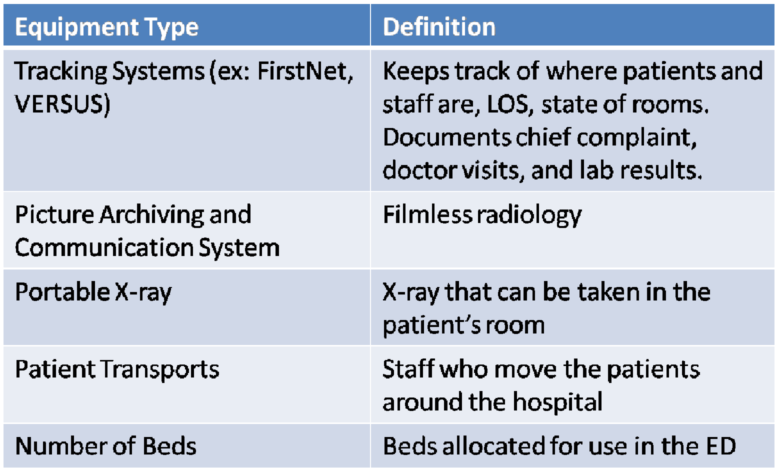
Verbal

Factors related to verbal communication that may hinder the productivity and efficiency of the ED include instances of miscommunication between doctors, nurses, and technicians. For example, better communication between transporters and medical care providers can shorten a patient's length of stay in the hospital. In order for patient transporters and ED cleaning staff to provide their services as quickly as possible, they must immediately be made aware of any requests for their services. The movement of patients requires the use of hospital resources, including its staff and beds, so making communication in intra-hospital patient transfers more efficient has been suggested for increased patient throughput. Inefficient communication is a major contributor to total transport process inefficiency. A study conducted by Hendrich and Lee found that 87.6 percent of the time taken during the transport process was inefficient and wasteful (2005). An effective system of communication between hospital units is essential to overall hospital operations, not just to the ED. In order to capture these effects, our study will record instances of verbal communication between staff personnel in order to see if we can highlight any possible inefficiency. Specifically, we will focus on conversations between staff members that have direct interactions with patients, such as doctors, nurses, technicians, and transports.

Physical Resources

Lab Tests

In addition to resources needed for communication, physical resources in the emergency department play a major role in efficient throughput. The amount and type of equipment present in the ED plays a crucial role in determining patient throughput. According to a survey conducted by Derlet and Richards of 210 hospitals across the United States, the main causes of overcrowding in EDs include space limitations, as well as radiology and consultation delays (2002). Especially important to our study is the time spent on lab testing and consults, both of which are significant sources contributing to patient length of stay. Our team will collect data specific to both of these areas to be used in the construction of a simulation that accurately models ED activity.



Equipment Type	Definition
Tracking Systems (ex: FirstNet, VERSUS)	Keeps track of where patients and staff are, LOS, state of rooms. Documents chief complaint, doctor visits, and lab results.
Picture Archiving and Communication System	Filmless radiology
Portable X-ray	X-ray that can be taken in the patient's room
Patient Transports	Staff who move the patients around the hospital
Number of Beds	Beds allocated for use in the ED

Table 3: Lab Equipment Glossary (Derlet & Richards, 2002)

Given that lab testing has been identified as a potential source of inefficiency, there has been extensive research done on how to improve this process, especially regarding radiology testing. One of the main systems used in radiology, picture archiving and communication systems (PACS), allows the completed radiology test to be easily sent to and stored on a computer instead of having to use films. This new system simplifies the process, resulting in a decrease in time spent by the radiology department on completing lab tests (Brunelle & Rawlinson, 2006). The UMMC ED has PACS in place, and during our data collection we will record when doctors in the ED access the system. New technologies have also allowed more tests to be conducted in the patients' rooms without the need for transportation to other departments within the hospital. One of these technologies is portable X-ray machines. Portable X-ray machines are used when the patient is too sick to be transported to radiology. While in theory this system should be more efficient than standard practices, there are significant bottlenecks in the sytem. Starting the x-ray process, ordering the exam, realizing the order, and leaving to complete the order are among the least efficient parts of the process. The examination portion of the test also creates a slight bottleneck. (Abujudeh, Vuong, & Baker, 2005). Because UMMC's ED uses portable x-ray machines, we will record the amount of time it takes for this process to be completed. After completing our simulation model, we will have the ability to compare the time spent using a portable x-ray machine versus sending the patient directly to the radiology lab to conduct the x-ray.

Bed Management

Besides having the necessary physical resources in the ED, proper organization and procedures are needed in order to facilitate the distribution of resources as well as to increase patient throughput. The concept of bed management, therefore, is essential in the optimization of patient throughput within an ED. The number of beds in an ED determines the number of patients that may be admitted, which in turn has a significant impact on how resources are distributed and used. By collecting data for a majority of the beds in UMMC's ED, we will be able to build a simulation that accurately models bed activity, such as average capacity and average number of beds open.

Research has shown the importance of bed usage in hospital efficiency and patient care. According to Roemer's Law, when more hospital beds are provided in a community at large, the more hospital care will be provided (Rohrer, 1988). However, these inpatient beds are not always readily available Accessibility of inpatient beds, therefore, has a profound impact on how many patients can be treated at any given hospital. Specifically to the ED, inpatient bed management is important to understand because it is inevitable that some percentage of patients treated in the ED will be admitted to inpatient wards. The efficiency of hospital emergency rooms suffers when bed overcrowding occurs (Kolb, Peck, Schoening, & Lee, 2008). The creation of discharge lounges, holding areas, or other regions that hold patients who are simply waiting to be discharged or to be transferred to another ward may help reduce bed overcrowding in the ED. Kolb et al. found that creating a series of buffer zones greatly increased overall patient throughput by more efficiently organizing waiting periods with a small amount of ED resources (2008). There were three primary types of buffer zones that were studied: a general holding area, an ED discharge lounge, and an observation unit. By using a combination of the aforementioned zones, triage-to-bed time was improved from approximately 12.1 percent to 21.7 percent (Kolb et al., 2008). Buffer zones can therefore be shown to decrease extended wait times as a result of ED bed overcrowding, but the effectiveness of buffer zones differs with the layout and policies of each hospital. Discharge lounges have also been found to be more beneficial in smaller clinical settings rather than large teaching hospitals. Instead of having a separate buffer area for patients during times of overcrowding, UMMC sets up temporary beds in ED hallways to accommodate more patients; we will be taking these into account when collecting data. By doing so, we must make sure that data from temporary beds is typical of regular ED operations.

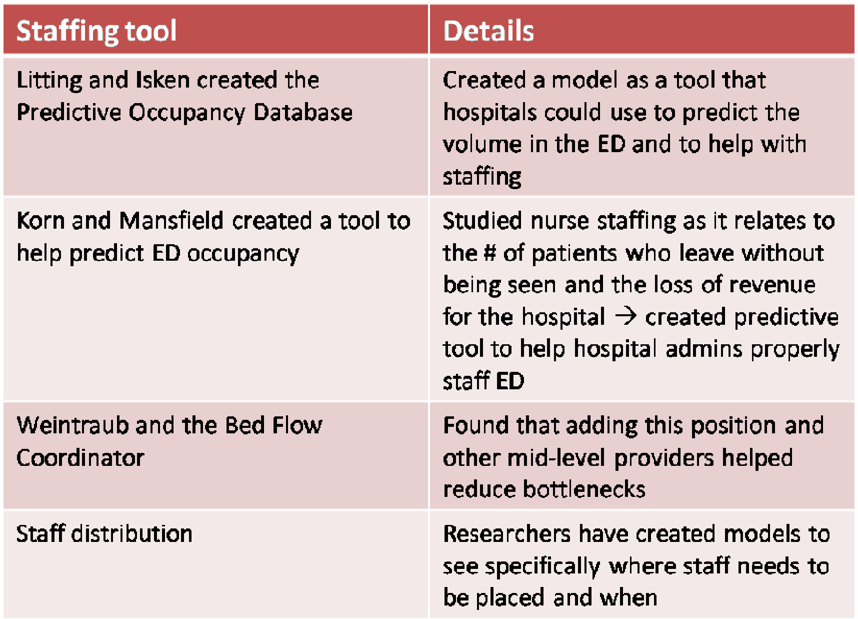
Scheduling

Patient Scheduling

The physical resources and the layout of the emergency department dictate the scheduling pattern necessary for optimal throughput. Effective scheduling within the ED is vital for maximizing patient throughput in this unit. The ED is the site of acute care in the hospital, and many times patients enter the hospital with a life-threatening condition which needs to be treated immediately. The rapid pace of activity in the ED makes scheduling difficult. The priority and triage numbers of patients need to be taken into account in order to ensure that the individuals who need the most urgent care are seen first, while still ensuring other patients are not waiting for extended periods to be seen. The priority and triage numbers are also known as severity score, which indicates the level of severity of a patient entering the ED. Scheduling is also important in dispensing medications and linen deliveries and allocating staff resources to care for ED patients. These variables coupled with the necessary protocols of a large teaching hospital complicate scheduling, and the effects of such variables must be understood in order to accurately model ED resources.

Operations research has been used as a tool to investigate better scheduling procedures. Green et al. used operations research to find the optimal scheduling pattern at a New York City hospital. Using a tool called the Lag Stationary Independent Period by Period (Lag SIPP), a tool that creates models of the patient care by breaking the day down into shifts, researchers were able to create a model for staffing levels at 2 hour increments. The models allow the researchers to see the number of care providers needed to satisfy the demand at the different increments throughout the day. The results from the models in this study were used to create new schedules in the ED. By creating a scheduling pattern based off the suggestions in the models, the hospital was able to significantly decrease the number of patients who left before being seen (Green, Soares, Giglio, & Green, 2006). Our study will use similar methods of simulation modeling to identify the effects of scheduling on ED operations, but will focus specifically on the resident model instead of dealing with all personnel.

Staff Scheduling



Staffing tool	Details
Litting and Isken created the Predictive Occupancy Database	Created a model as a tool that hospitals could use to predict the volume in the ED and to help with staffing
Korn and Mansfield created a tool to help predict ED occupancy	Studied nurse staffing as it relates to the # of patients who leave without being seen and the loss of revenue for the hospital → created predictive tool to help hospital admins properly staff ED
Weintraub and the Bed Flow Coordinator	Found that adding this position and other mid-level providers helped reduce bottlenecks
Staff distribution	Researchers have created models to see specifically where staff needs to be placed and when

Deciding which staff members in the ED are placed in the schedule is just as important, if not more so, than the scheduling method used. The number of staff members and their placement throughout the ED can have a large effect on overall ED efficiency. An influx of patients at any one time can place strain on hospital resources, most notably the staff caregivers. Therefore, it has been suggested that the addition or reorganization of staff members within the ED could create improvements in efficiency. Most implementations of new staffing protocols arise from analysis of current practices, and are suggested after sufficient data regarding past staffing performance has been collected.

Adequate staffing is necessary to eliminate job dissatisfaction and medical errors that result from shortages of hospital staff (Littig & Isken, 2007).

While staffing is an important consideration, it is often complicated because of the unpredictability of ED operations. Research has been conducted to attempt to create a model to predict the fluctuations so that an efficient staffing pattern can be maintained by the hospital. Littig and Isken created a simulation just for this purpose. They maintain that hospitals usually use several different methods, none of which provide a complete picture of future ED activity individually, to predict how many people are going seek medical treatment in the ED in the future. They studied a 226-bed, medium-sized community hospital to create a simulation. They considered three factors that contributed to changes in ED occupancy, including patients arriving, patients being admitted, and patients being discharged. From the Predictive Occupancy Database (POD) they were able to make predictions for the hospital staff to use. The POD is a model developed based on past actions in the ED including how the patients arrived at the ED, if patients are being transferred within the hospital, and when the patient leaves the hospital. Although our study focuses specifically on residents, it is still extremely important to understand how changes in other types of personnel can affect ED efficiency. Additionally, understanding past attempts to improve scheduling will be beneficial to our study. The effects of a past study that focused on staffing changes involving nurses, for example, could exhibit similarities to the effects of a staffing change involving residents in the ED.

Personnel Specific Scheduling

Patient transports play a major role in overall ED efficiency, since inefficient transports slow patient throughput. A major need for transports arises because more advanced testing is located far away from the ED (Hendrich & Lee, 2005). We will be recording transport actions while in the ED and will use the information collected in the construction of our simulation model.

Resident Education Model

While the above studies have researched and found solutions to several bottlenecks relating to communication, equipment, and staffing, one important area that our research will focus on is the resident education system. The resident education model creates a dual role for doctors in the emergency department, since a resident's role includes both treating patients and learning medicine. Thus, the resident care model can affect patient throughput because of the additional time spent instructing residents. In turn, overcrowding often has a negative impact on the education for the residents. Some research has been done to identify possible solutions to the current resident teaching method.

The time needed to teach residents in the ED has been shown to have a negative effect on throughput. In one study, researchers aimed to review ED patient waiting times, length of stay before an admission decision is made by the doctor, ED LOS, and utilization of various tools for the examination and treatment for periods of time where residents were on strike versus times of normal resident staffing patterns (Harvey et al., 2008). The main goal was to identify if the residents were interfering with the efficiency of the ED. The study was conducted at the 650-bed teaching hospital Waikato Hospital, located in New Zealand. After collecting data from a normal time period in the emergency department as well as during a period where residents were on strike, the researchers made several interesting discoveries. Some of the findings did not indicate a problem with the resident teaching model; for example, the researchers found that the emergency department had less bed availability during the strike period than during the non-strike period, there was no difference in the ED mortality rate, and there was no time difference between the strike period and the normal period for the patient time seen to deposition. The length of stay data and other timing data, however, indicated that during the period where residents were present in the ED, there was a major difference in throughput compared to when the residents were on strike. During the strike period, the doctors saw 0 percent of the lowest level triage patients within the recommended time frame compared to the non-strike period, when they saw 25 percent of the lowest level triage patients within the suggested time frame. For the more severe triage levels, however, the doctors saw a greater percentage of patients in the recommended time during the strike period than during the non-strike period. In addition, for all triage categories except the most life-threatening, patient LOS was reduced during the strike period (Harvey et al., 2008).

Just as the resident teaching model affects throughput, overcrowding in the ED has been shown to affect the education that residents receive. A greater patient density is beneficial for developing patient care and interpersonal skills. Overcrowding, however, causes residents to receive a poor education with regard to treating people effectively and in a timely manner. Overcrowding is also detrimental to teaching new doctors skills in professionalism. When temporary beds are set up in hallways due to overcrowding, the doctors have no choice but to discuss sensitive medical information with patients even if they are not in an ideally private setting (Shayne et al., 2009). Overcrowding interrupts the residents' education, as they must spend more time talking to doctors or answering pages instead of learning. Residents are not wasteful with their time; the problem arises because the majority of residents' tasks were performed in one- to five-minute increments and because the tasks were frequently interrupted by the need to answer pages or speak with other physicians. Researchers report that 57.8 percent of the residents' time was spent on patient care, 15 percent was spent on education, and only 1.5 percent of the time was wasted. They also discovered that the residents walked 13.87 feet for every minute they worked. The researchers called for a reevaluation of process and resident flow to eliminate dysfunctional and fragmented processes (Dassinger, Eubanks, Langham, 2008). The research on the impacts of overcrowding on residents' education is not comprehensive, however, and more studies with different measures are needed to better understand the effects of overcrowding on resident education (Shayne et al, 2009).

There have also been negative effects implicated with delays on raising funds for the teaching aspect of the resident model. Delays in throughput are negative for resident education because the fewer people treated, the less funding the teaching program receives. Residents can benefit from greater throughput and the generated revenue that comes with it. A more efficient use of time resulted in more exposure to patients for residents without sacrificing their education. Improved efficiency leads to more funding for the resident teaching system (Bush, Lao, Simmons, Goode, Cunningham, 2007).

A study by Atzema et al. proposed integrating new methods into the teaching model to improve the residents' education (2005). Atzema et al. found that checking in with the attending physician comprised 3 percent of the residents' total work time and that the process of being supervised by other doctors comprised 11 percent of their work time. The researchers hypothesized that these time limitations were a result of emergency department overcrowding. They suggested that the education of residents, because of the time wasted checking in with supervisors, was negatively impacted. They proposed several improvements that could better the education of residents, including optimizing the teacher-learner interaction, demonstrating a good teacher attitude, being a role model, actively involving the learner, tailoring teaching to the situation, providing feedback, tailoring teaching to the learner, establishing expectations, and using additional teaching resources (Atzema et al., 2005).

Identifying solutions to the education dilemma found in teaching hospitals is the focus of various studies. The use of physician extenders, staff members who complete excess service needs, may provide the balance necessary to maintain resident education without jeopardizing throughput and potential revenue (Bush et al., 2007). Changing the length of the residents' shifts can also make an impact on throughput. Residents are able to evaluate more patients per hour in the long run as they gain more training. Researchers evaluated 9- and 12-hour shifts of second-year residents, and found that the residents working 12-hour shifts evaluated 1.06 patients per hour while the residents working 9-hour shifts evaluated 1.15 patients per hour. The three-hour time difference resulted in 10 additional patients being seen by residents on the shorter shift. The scope of the study is limited, however, because it did not study patient turnaround time, the time it takes for a patient to be treated (Jeanmonod, Jeanmonod, & Ngiam, 2007).

Other research suggests a new system for teaching residents all together. One study proposed using human simulation to teach residents. Simulation allows residents to make mistakes and learn the necessary skills without affecting real patients. The simulation very closely models what residents would experience in real scenarios (McLaughlin, Doezema, & Sklar, 2002)**.** With the residents using the simulation, only the more experienced doctors would be treating patients, relieving some of the patient build-up. The goal of our study is to determine the impact of the resident education system on the ED by combining observational data with historical database data. The addition of both sets of data should provide new insight to this field.

Past Simulation Research

Studies have shown that using a simulation to identify changes to increase efficiency is the best way to test different scenarios in the emergency department. Mackay and Lee found that mathematical models were not flexible enough to fit the data collected in the hospitals (2005). They also found that the mathematical models they used were not able to accurately make predictions. Miller, Ferrin, and Messner went on to say that using a mathematical model became almost impossible when the scenarios became too complex (2004). Ruohonen also found it necessary to use simulation modeling in looking at the emergency department because of the complexity of the ED (2007). The ED is one of the most complicated systems to model because it consists of human and material resources, in addition to potential patients awaiting medical services. As it would not be possible to actually make changes in the ED to see if they would work without compromising patient health and safety, creating a simulation that can model the emergency department and allow for experimentation is the best tool for ED research. In medical research, three types of simulations are used to model processes. There are disease models to study medical treatments, operational models to study patient flow and how to alleviate bottlenecks, and strategic models to study the entire system (Stainsby, Taboada, &Luque, 2009). Our project will be using an operational model to study potential bottlenecks related to the residency model.

As demonstrated in our study, simulations enable the researcher to break down the steps of each staff member in the ED to analyze individual actions and how changes in specific areas can impact patient length of stay in the ED. The simulation also allows the researcher to see how different variables affect each other. Lu Wang, for example, used an agent based-simulation and found that radiology-related actions and doctor care had a major impact on each other, and thus that effective communication between the two was essential. The agent-based simulation was developed by breaking down the ED into specific functions and developing detailed tasks for each function (Wang, 2009).

Simulation modeling is also ideal for researching throughput because it allows the researcher to change inputs and analyze the effects of such changes (Ahmed & Alkhamis, 2009). Past research shows how simulations can impact various processes in hospitals. Komashie and Mousavi were able to create a simulation model that was used to test possible changes to the ED that could result in throughput improvements. The model tested scenarios that included adding staff, adding beds, and assessing delays due to admissions. They found that adding staff, beds, and a new clinic significantly reduced the waiting times (Komashie & Mousavi, 2005). In a study conducted by Ruohonen, the ability to change inputs in a simulation model was used effectively to test the redistribution of key resources. This solution, along with the utilization of different technologies and the implementation of a new plan for delivering patient care, resulted in a 40% reduction in the patients' length of stay (Ruohonen, 2007). Patient buffer concepts have also been successfully tested using a simulation model. In a study by Kolb, Peck, Schoening, and Lee, a simulation was created that tested multiple scenarios in which different buffer zones were included in the ED to be utilized in times of overcrowding. While Kolb et al. found that all of the buffer scenarios had some effect on hospital throughput, they found that a combination of buffers was the best scenario (2008).

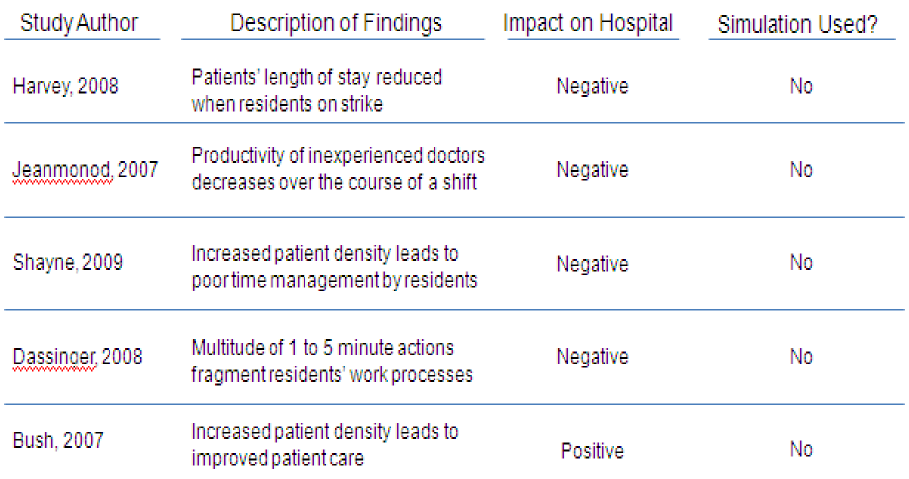
Simulation modeling has been used effectively to specifically test scenarios involving staffing changes. Ahmed and Alkhamis (2008) used simulations to evaluate how different staffing arrangements affected overcrowding in the emergency department. Ahmed and Alkhamis were able to create a scenario that could decrease the waiting time by 40 percent and increase throughput by 28 percent. Rossetti et al. also performed a simulation at the University of Virginia Medical Center to evaluate alternative staffing schedules for their attending physicians. They used the simulation software Arena 3.0 in their study, which provides guides, graphs and other tools based specifically on what the researcher is trying to model in the real world. In order to create the alternative scenarios, the researchers asked the hospital staff for suggestions and created timing variations of the current staffing pattern. The best scenario reduced the total patient stay in the ED by

40 hours a day by adding one attending to the schedule every day from 10 a.m. to 6 p.m. (Rossetti, Trzcinski, & Syverud, 1999).

Using a simulation model will be helpful to explore the less researched areas of concern in the emergency department. Since UMMC is a teaching hospital, the resident education system is a major factor in the operations of the ED. Much of the research on the ED has not focused on this area. We can assess the impact that the residents and the current teaching model have on the ED at UMMC.

Contribution to Literature

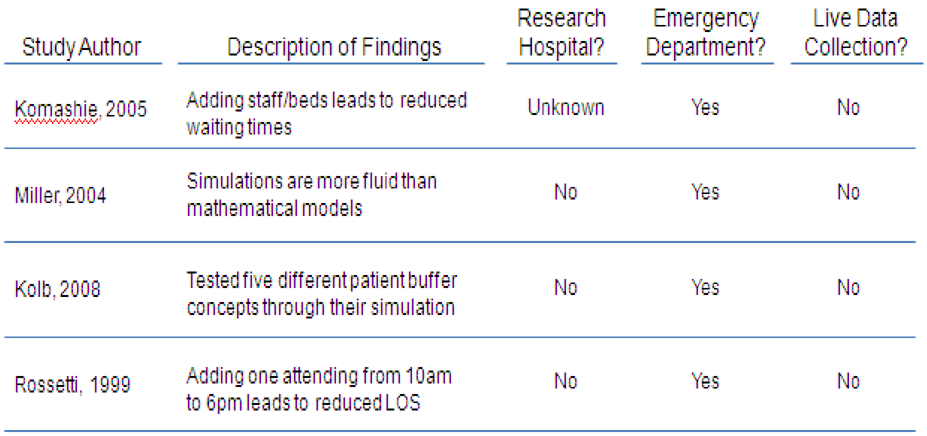
Our study occupies a unique niche in the literature previously discussed. Multiple studies researched inefficiencies in the ED caused by the resident education system. These studies found that, in general, the resident model has a negative impact on the efficiency of an ED. None of these studies, however, used a simulation model to test changes to the residency model in order to propose potential solutions to these inefficiencies. The use of a simulation model will allow us to not only identify bottlenecks due to the resident education system, but also to experiment with the actual ED studied and find statistically significant improvements to the department.



Study Author	Description of Findings	Impact on Hospital	Simulation Used?
Harvey, 2008	Patients' length of stay reduced when residents on strike	Negative	No
Jeanmonod, 2007	Productivity of inexperienced doctors decreases over the course of a shift	Negative	No
Shayne, 2009	Increased patient density leads to poor time management by residents	Negative	No
Dassinger, 2008	Multitude of 1 to 5 minute actions fragment residents' work processes	Negative	No
Bush, 2007	Increased patient density leads to improved patient care	Positive	No

Table 5: Comparison of Past Studies on the Resident Education Model

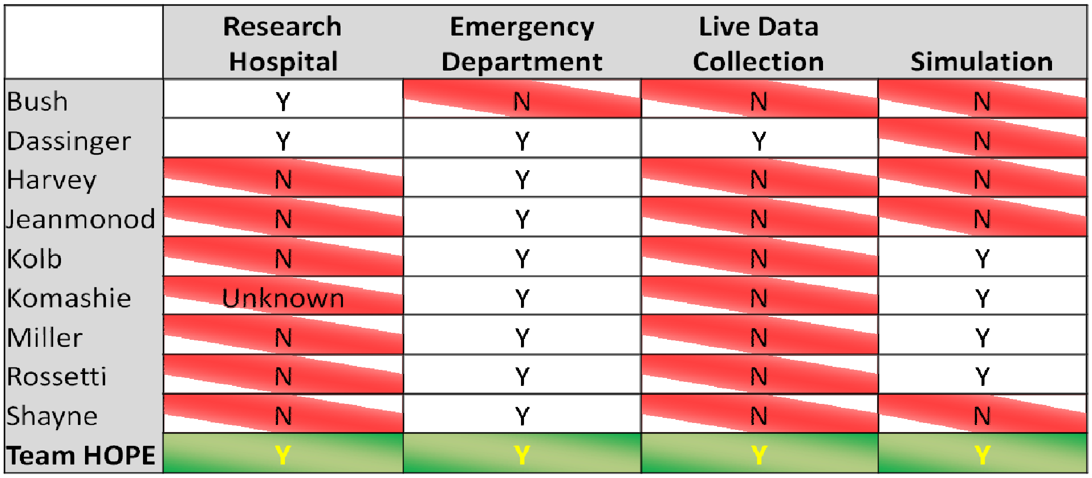
Those studies that did use a simulation model in an emergency department were generally not conducted at a research hospital, as ours was, and as such did not test the residency model. Although these studies could propose general improvements to the ED, our study is unique in that we can look for bottlenecks caused by the resident doctors themselves.



Study Author	Description of Findings	Research Hospital?	Emergency Department?	Live Data Collection?
Komashie, 2005	Adding staff/beds leads to reduced waiting times	Unknown	Yes	No
Miller, 2004	Simulations are more fluid than mathematical models	No	Yes	No
Kolb, 2008	Tested five different patient buffer concepts through their simulation	No	Yes	No
Rossetti, 1999	Adding one attending from 10am to 6pm leads to reduced LOS	No	Yes	No

Table 6: Comparison of Past Studies on Hospital Efficiency Using Simulation Modeling

The studies we found that used simulation modeling also relied entirely on historical data from the hospital database. Not only is this data often inaccurate and incomplete, but the data collected by the hospital does not allow for a study with significant granularity. The hospital database may have timestamps for when a patient was given a bed, when a doctor put in lab test orders, and when the patient was discharged; the live data collected in our study allows us to map not only the movement of doctors, patients, and nurses accurately, but it also allows us to build of a model of doctor preferences and how doctors make decisions on which patients to visit. Table 7 outlines where our study fits in with the current literature:



	Research Hospital	Emergency Department	Live Data Collection	Simulation
Bush	Y	N	N	N
Dassinger	Y	Y	Y	N
Harvey	N	Y	N	N
Jeanmonod	N	Y	N	N
Kolb	N	Y	N	Y
Komashie	Unknown	Y	N	Y
Miller	N	Y	N	Y
Rossetti	N	Y	N	Y
Shayne	N	Y	N	N
Team HOPE	Y	Y	Y	Y

Table 7: Research Study Comparison

Our study is the first to combine live and historical data to allow us to simulate the ED more accurately than previous research. Understanding how doctors decide to break up their time among various patients and patient-related activities enables us to experiment with changes in the residency model more effectively than any previous study.

3. Methodology

Simulation Modeling Overview

The first step to creating a simulation is defining the system that is going to be studied. A system is anything with related elements that interact with each other. It is important to consider both the factors within the system and the factors connecting the system to the outside community (Rubinstein, 1981). In order to research solutions to the problem that is being studied, the first thing that needs to be developed is a theoretical model of the system. In order to complete this task, it is necessary to understand the system through observations so that the operation of the system can be established. Each level of the system should be evaluated to determine how it is related to the next level (Rubinstein, 1981). Making a list of all of the essential components of the system and what needs to be studied in each component will help guide the modeling process (Rubinstein, 1981). For example, in the ED a list of components would include nurses and doctors. Within each component we would need to study attributes such as how many there are, what they do on each shift, and who they interact with. Before moving on to creating the model, it is important to establish that the conceptual model is accurate (Ruohonen, 2007). The experimental goal should also be established at this stage because when creating the simulation, it is essential that all of the necessary components needed for a specific manipulation are planned for. (Rubinstein, 1981).

Once the theoretical model is established and verified, the computer simulation can be created (Ruohonen, 2007). Using the language that best suits the system's needs, the simulation is created, and the state and interface variables and parameters for the model are developed (Rubinstein, 1981). Data from observations of the real system is needed for use in the model (Rubinstein, 1981). After creating the model the next step is to validate the model (Ruohonen, 2007). After the model has been validated against the real system, the experiments on the model can be run and the results can be reported (Ruohonen, 2007).

Components of the System

This study focuses on the emergency department at the University of Maryland Medical Center. Although the model being created is mapped out through simulation, it is important that the parameters in the model characterize important parts of a real-world system. Consequently, a thorough familiarity with the UMMC ED's physical layout, resources, and staff procedures are necessary to understanding and identifying potential areas of inefficiency in the typical patient throughput. An outline detailing the processes and resources needed for efficient patient throughput within the ED was compiled, using team members' observations as well as clarification about processes by staff members. The three stages critical in simulation modeling (input, throughput, output) were accompanied by real-life observations in order to ensure a valid model.

Input

When the patient arrives at the ED, they must first sign in at the Quick Registration (QuickReg) desk. At QuickReg, they are asked for their identifying information, whether they have visited UMMC before, and their chief complaint. After the patient is finished with the QuickReg, he or she sits in the waiting room until a triage nurse is available to see him or her.

During triage, the patient is taken to a small exam room directly attached to the waiting room. The triage nurse takes the patient's vital signs, asks them questions about their medical history, including any allergies, and assigns the patient a priority number. The priority number is given based on the triage nurse's evaluation of how urgently the patient needs to be treated. Priority numbers range from 1 to 5. A priority assignment of 1 means the patient should be seen immediately and is in extremely critical condition. A priority assignment of 5 means the patient is not in critical condition and could have visited his or her primary care physician instead of coming to the ED. Once the triage nurse finishes evaluating the patient, the patient returns to the waiting room until a bed becomes available. However, during triage, if the nurse finds it necessary to run certain tests before the patient is assigned a bed, the nurse can administer them in a room allocated for those purposes. The tests that nurses are permitted to administer during triage include electrocardiograms (EKGs), X-rays for pneumonia, and urine pregnancy tests. The nurse can also administer Tylenol for high fevers and a nebulizer for patients with asthma.

Furthermore, during triage, the patient's information is entered into the computer and a printed copy is taken to the ED so the ED charge nurse can see which patients are waiting for a bed. The charge nurse can then assign any available beds based on the waiting patients' priority number. Although charge nurses assign beds primarily by the priority number, they also take into consideration how long the patient has been waiting in the waiting room.

Throughput

Zoning

The ED at the UMMC features 21 normal exam rooms, with one bed in each room. There are also three Rapid Diagnostic Units (RDUs), which hold patients being treated for relatively simple conditions that don't require extensive tests and consultations, as well as three resuscitation rooms. Patients in the RDUs are under a 24-hour watch and will either be discharged or admitted upstairs upon the completion of their 24-hour watch period. Depending on their priority numbers, conditions, and length of time spent in the waiting room, patients are assigned to one of the ED rooms before treatment can commence. During the department's peak hours, when there is greater than average patient traffic, patients are often placed in hallway beds as well, a practice referred to as "parking."

The main hub of activity is the central service desk, where patients' medical clipboards are kept. The charge nurse is almost always stationed at this desk, while the other nurses and medical staff mainly move between the desk, the exam rooms, and the staff room. Across from the desk is a white board that keeps all staff members apprised of their roles and the patients they are in responsible for (See Figure 1).

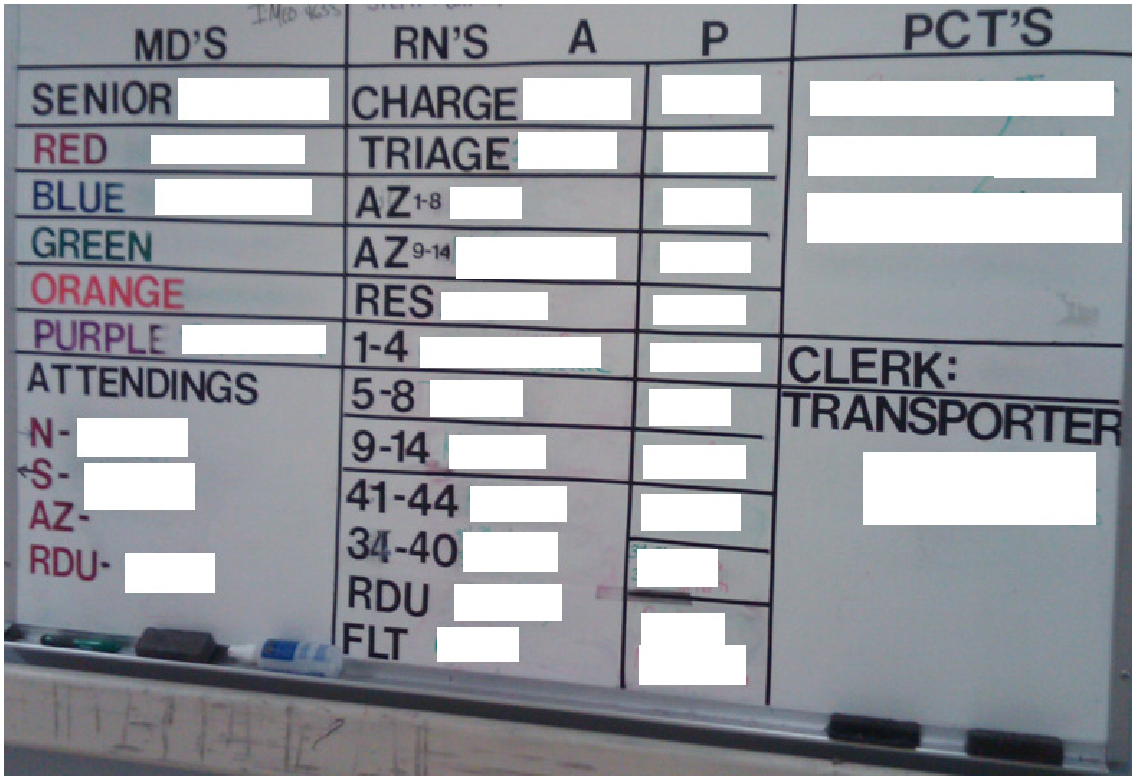


Figure 1: A photograph of scheduling whiteboard at the UMMC ED

The board is split into three main sections – one for the doctors, one for the nurses, and one for other staff members. The leftmost section, for the doctors, is split into two parts. The top half pertains to the residents and is labeled "MD's." A resident is assigned to each of the five colors in this section – red, blue, green, orange, and purple – and the senior resident on the floor is written under the "Senior" category. The lower half, labeled "Attendings," specifies the four attending physicians on duty during the current shift. Their locations are north (N), south (S), ambulatory zone (AZ), and RDU. The phone numbers written across the top of the board include any common numbers doctors and nurses may need to use to treat patients, such as pharmacy, the cardiology department for admissions or consultations, and other departments depending on the current patient body.

As patients are given beds, they are simultaneously assigned a red or blue color and are examined by the resident responsible for that color. The charge nurse arbitrarily assigns the red and blue colors to patients depending on the volume and acuity of patients each resident already has. The red and blue residents are stationed in the ED for the entirety of the current shift and answer to the shift's senior resident and attending. The green, orange, and purple MDs are swing residents who come in sporadically to the ED to provide assistance whenever they have time. They are usually 3rd year residents who come and pick whichever patients they want from the red and blue residents' workloads. These patients are then given the green, purple, or orange color. Green, purple, and orange residents answer to the attending physician only. At least one swing resident is usually present in the ED at all times. The residents are occasionally assisted by interns, the term for first-year residents, as well as 3rd and 4th year medical students carrying out their emergency department rotations.

Observing the actions of doctors and residents was particularly important in this project, as our model of the UMMC is focused on defining inefficiencies in the residency model. Nurses and other staff personnel in the ED were observed, but their actions were not as crucial to the development of the simulation model.

The middle section of the board pertains to the nurses. It is split into three columns: RNs (Resident Nurses), A (A.M. shift), and P (P.M. shift). The first column specifies either the nurses' roles or locations, and the other two columns contain the names of the nurses falling in those categories for each shift. For example, the first two rows define the charge nurse ("Charge") and the triage nurse ("Triage") for both shifts.

The next two rows split the ambulatory zone between two nurses ("AZ1-8" and "AZ9-14"). The "RES" nurses are responsible for the patients in the resuscitation rooms, while the "RDU" nurses are responsible for the patients in the RDU rooms. The next five rows split the other ED exam rooms into five categories (1-4, 5-8, 9-14, 34-40, and 41-44), with at least one nurse assigned to each group per shift. The last row, "FLT," lists the float nurse for each shift. "Parked" patients are assigned to one of these groups depending on their physical location within the ED. If any of the nurses need to take a break or has an overwhelming patient load, the float nurse is responsible for either covering for or assisting those nurses.

The last column on the board is divided between three types of supporting staff – the Patient Care Technicians (PCTs), the unit clerk, and the transporters. Three PCTs are present in the ED at any given time, split between three zones: 1) exam rooms 1-4, the RDUs, and the Pediatric Emergency Department (PED), which is located within the main ED; 2) exam rooms 5-8, the resuscitation rooms, and triage; and 3) exam rooms 37-40 and 41-44 and the ambulatory zones. The technicians are tasked with taking patients' vital signs, performing EKGs, drawing blood, assisting in patient transportation when necessary, assisting the patients with basic undertakings such as getting settled into their rooms, keeping the rooms stocked with materials, and other similar tasks. The unit clerk admits and registers those patients the charge nurse wants to admit to an ED bed, processes the paperwork for patients' inter-hospital transfers, orders all supplies for the ED, brings patients in from the waiting room, and answers phone calls, including those from family members, referring doctors, and other hospital personnel. The transporters move the patients throughout the hospital as necessary for intra-hospital transfers, lab and radiology tests, etc. During peak times, which are generally from 11am to 11pm and less frequently from 3am to 3pm, two to three transporters are kept in the ED. This is almost always the case on Mondays and Tuesdays, the ED's busiest days. At all other low volume times, at least one transporter stays in the ED. A nurse will only accompany a transporter if the patient needs constant monitoring; otherwise, if the primary transporter needs assistance with heavy equipment or a big bed, a second transporter or a PCT will provide assistance.

ED Personnel

Nurses, doctors, technicians and transporters comprise the staff of the emergency department that interacts directly with patient throughput. The staff moves around the ED to administer appropriate levels of patient care depending upon their own medical judgment, rather than in accordance to any rigid, uniform protocols. Understanding how the staff chooses to react to patient needs—how they prioritized the demands of several needy patients at once—was key to creating an accurate simulation model. This preliminary understanding of the fluid actions of staff was obtained via discussions with and simple shadowing of staff members. The formal observational study proceeded based upon this understanding.

Nurses

During the day, ten nurses work from 7AM until 7PM. To accommodate the peak flow at night, the number of nurses on shift increases to eleven or twelve, with these nurses working from 11PM until 11AM. There are exceptions to shift schedules, which allow for shift overlap and overtime. Up to 30 minutes are allowed for shift changes, wherein the outgoing nurse will report to the incoming nurse the condition of all the current patients in his or her designated area. On average, this reporting only takes 10 to 15 minutes. Nurses take breaks randomly, when they have a few free minutes.

Each nurse is responsible for monitoring four beds in one geographical zone; the department is split up into four zones. When the ED is particularly busy and all the rooms are full, nurses will be assigned additional patients, who are placed in the hallway beds within their zone. When the nurses first visit a patient, which may be before or after the doctor's first visit, they introduce themselves, assess the patient's condition, and discuss the patient's medical history in an effort to connect with the patient personally as well as a means of medical discovery. This step involves some duplication of questions from the Triage Chart. Without a doctor's order, the nurse may not administer any medication at this time, except for a nebulizer for asthma patients and Tylenol for patients for high fevers. The nurse may also order basic tests which include chest X-rays for suspected pneumonia, some blood work, and urine pregnancy tests, but nothing more. This is protocol dictated by the Medical Director of UMMC.

Following their initial visit, nurses make "rounds" informally, with the goal of checking in with each of their patients at least once every two hours. At this time, nurses reassess acuity levels for patients and update this information on the patient's clipboard and in the computer. When making rounds, nurses gauge the amount of time needed with each patient, ensuring that higher acuity patients receive the most attention. These decisions on which patients to serve first, and the amount of time required to service each patient, were incorporated into the simulation model in order to represent the actions a nurse may take in reality. The nurse's primary role is in the ED; they may leave the ED to help transport a patient who needs to be monitored while being moved, but all of their other tasks remain in the ED.

Additionally, a charge nurse is on staff at all times. He or she is responsible for the overall functionality of the ED, bed assignments, and the general welfare of all patients and staff. In rare cases of high bed demand, the charge nurse assumes responsibility for some patients. If so, their beds are located directly in front of the central desk, so that the charge nurse may maintain his or her regular duties. Other nurse titles include nurses in charge of the resuscitation rooms and a float nurse who roams the ED helping the other nurses with various tasks when necessary.

Midlevel Care Providers

Midlevel care providers include Nurse Practitioners. The Nurse Practitioner works only in the Ambulatory zone with the less severe patients. There are no midlevel care providers working in the main section of the ED, and therefore these personnel were not included in the simulation as a resource.

Doctors

While nurses are the most ubiquitous players within the ED, doctors play a significant role in determining the flow of patients. The residents work twelve-hour shifts while the attending physicians work eight-hour shifts, with two attending physicians and three to four residents working at all times. The shifts for attending physicians are 11PM-7AM, 7AM-3PM, and 3PM-11PM. The shifts for the residents are 7AM-7PM and 7PM-7AM.

Residents

Residents are an integral part of the ED. Since the UMMC is a teaching hospital, the residents continue their medical education in a practical setting. Residents perform the initial evaluations but the resident's supervisor, the attending physician, must make the final decisions in patient care. The residents are zoned off in the ED based on color categorization. The patients are each assigned a color when they are assigned a bed and each resident is in charge of a color. The most experienced residents are the senior residents. The red and blue categories are residents who report to the senior resident and the attending physician. The orange, purple, and green categories symbolize swing residents who are visiting the ED but do not typically work in the ED. The swing residents work primarily in the Ambulatory Zone and they pick up patients from the red and blue residents' workloads. The swing residents are typically 3rd year residents who report to the attending physician only.

Attending Physicians

The attending physician overlooks the residents and makes the final decisions regarding a patient's treatment. Attending physicians are located in the north section of the ED, in the south section of the ED, in the ambulatory zone, and in the Rapid Diagnostic Unit. The attending physician typically makes their evaluation after the residents have seen the patient. The attending is in charge of discussing the case with the residents and providing education. The attending physician has the final decision in a course of action for a patient and makes the decision to admit or discharge the patient. The pathway of patient care that the attending physician follows in the model needed to reflect the responsibilities of the attending physician as not only a care provider, but also as a teacher to the residents. The simulation is based on analyzing the resident care model, and the balance between teaching residents and caring for patients is critical. Therefore, a lot of time was spent shadowing the attending physician as well as the residents in order to try and understand the reasoning behind their actions.

Consults

When the doctor in the ED needs the opinion of a doctor in a specific department regarding the treatment of a patient, the ED unit clerk uses a pager to contact the consulting physician. There is a unit clerk available at all times. Usually, the consulting doctor comes to the ED to conduct an examination of the patient. When they arrive, they notify the clerk or a R.N. before seeing the patient. They can order lab tests or radiology tests by letting the nurses know what is needed. Sometimes the ED transporters transport the patient to the department where they will receive their consult. In this case, the patient is sent to the consult with a form explaining why they are in the ED and what they were diagnosed with. When the consulting doctor is finished with the exam, they fill out the patient's chart with a note for the ED doctor. The consulting doctor usually uses the paper charts to document their visit instead of using the electronic systems. If more follow-up work is needed, it is usually done through phone calls.

Technicians and Transporters

Technicians, or Certified Nursing Assistants, are also vital actors, with two to three working per eight-hour shift. One technician is stationed around the Urgent Care/Triage area and is responsible for the front rooms, while the other is charged with Acute Care and the backmost rooms. The technicians are able to transport patients, draw blood, provide personal care, answer hall lights, check vitals, start IVs, insert tracheotomy tubes and catheters and perform CPR, but cannot administer any medications. Technicians are directed by and shared among all nurses. Finally, there are transporters, who are assigned to moving non-critical patients around the hospital. For example, the transporter may take the patient to get an X-ray or to transfer the patient to their inpatient room.

Medicine and Test Orders

As part of the diagnosing and prioritizing process, doctors and nurses can also use results from laboratory tests and radiology examinations to help guide their decisions. Orders for lab tests and radiology must be input by doctors into FirstNET, the ED's electronic system, and are also noted on the patient's clipboard. In the case of a consulting physician ordering a test for an ED patient, the consult informs either a resident or a nurse, whoever is readily available, of any required lab or radiology orders. Nurses and technicians then draw the necessary labs, such as blood, urine, spinal fluid, body fluid, and taps; these activities are all performed within the room. Certain labs require special treatment. For example, spinal fluid requires special consent, a lumbar puncture, time-outs, and a certain body position for the patient. Assessing the Keppra level of a seizure patient is a lengthy process – after the labs are drawn and sent out, the results take another 2-3 days to arrive.

After a nurse or technician obtains the lab samples, they then send the samples to the lab for analysis. There are many labs located throughout the UMMC compound, each with different hours. The ED has its own lab located on the same floor, called the stat lab, which operates from 10:00AM to 2:00AM the next day. Because lab testing plays such a critical role in the staff's decision-making process, the ED ensures it always has at least one open lab facility to which it can send samples. During the interim period when the ED lab is closed, the ED can send its sealed samples to the main lab upstairs via a pneumatic chute. Results from labs upstairs are generally available within an hour.

When the ED lab is open, its technicians perform simple lab tests on blood, gas, chemicals, electrolytes, stool, serum, and urine, with the most common tests being pregnancy and cardiac tests. The lab technicians in this lab also grow cultures of bacterial samples taken from patients in order to determine whether they have an infection. In order to perform these tests, the ED lab is supplied with, but not limited to, one Bayer Clinitek 50 and two Stratus CS machines, as well as a centrifuge. These tests can be completed on average within 22 minutes with no batching or set order of testing. An ED test may take longer if a bacterial culture shows positive results for infection, in which case the sample is sent to the microbiology lab within the stat lab for further analysis. If the microbiology lab is unable to process the sample, it is sent upstairs through the chute for another lab to analyze.

In addition to the tests performed by the ED lab and its cohorts, the ED staff may also request radiology tests and electrocardiograms (EKGs) if necessary. For any type of radiology test, such as an X-ray or MRI, doctors enter an order into FirstNET, instantly sending a notification to a screen within the Radiology Department. These radiology tests are performed on a first-come, first-served basis, unless the doctor and patient arrive in the radiology room prepared to have the test performed immediately. The completion time of radiology tests varies, depending upon the weight and/or severity of the patient's health problem.

X-rays can be generated in four radiology centers located throughout the hospital: first floor, clinic/trauma area, shock trauma area, and ED Radiology. They may also be generated by a portable X-ray machine. This may be the preferred route if the patient already has a bed because it eliminates the need for a transporter. The location of an X-ray exam depends on the capacity of the machine, ease of testing, and the time of day, as machines have varying hours of operation.

The first floor X-ray room closes at 4:00PM and the clinic/trauma area closes at 5:00PM. The shock trauma X-ray machine is available twenty-four hours a day, and the ED Radiology operates from 4:30PM to the time the other radiology rooms open. If a portable X-ray machine is required, the machine is brought to the room. The machine operator does not follow a set schedule or batching system. Similarly, EKGs are performed in a patient's room if they have been assigned a bed already. Otherwise, they are done in the EKG room.

Based on test results and medical professionals' own physical examinations, doctors then may place medicine orders as required. Senior residents or attending physicians place medication orders into the system, after which the orders show up on the system screen. Simultaneously, a note is left on the patient's clipboard to inform the nurses of the order, after which the nurse checks the system to confirm the order.

Most medicines are stocked in the ED's pharmacy room; once a nurse checks the medicine orders, they retrieve the medication from the ED's stock. Approximately one in six patients requires medication not kept in the ED. If the medication is not readily available, the nurse right-clicks on the order through the EMAR system to send a medicine request to an upstairs pharmacy. There are four different pharmacies located in strategic places throughout the hospital, all of which are open 24 hours a day. When the pharmacy fills the order, the medication is sent to the ED via the pneumatic tube and a buzzer goes off in the ED to notify the ED staff.

After acquiring the medication, the nurses immediately administer it to the patient, after which the nurse makes a record in the electronic system. If a patient refuses to take the medication, the nurse selects the corresponding option and the system entry turns gray. Alternately, a star is added next to the entry if the patient reacted unexpectedly during administration or if the patient takes other medications in addition to or instead of the prescribed order.

Tests and medications administered to a patient were taken into account in the created simulation. If a patient has a more severe acuity, then it is more likely that they will need more medication and/or more medicine, so it was important to note which tests and medication orders corresponded to different patient severity levels.

Disposition

Once a patient has been thoroughly examined, with appropriate tests and medications having been administered to the attending physician's satisfaction, the attending physician determines what additional medical services, if any, the patient requires. At this stage, the attending physician decides whether to keep the patient in the ED for an additional period of time or to discharge the patient. The patient can either be discharged to a department within the main hospital if they are in need of further specialized care, or can be discharged to go home if they are deemed fit for release.

Output

Once the doctor decides where the patient is going to go from the ED, various steps occur based on the patient's destination. If the patient has been discharged to go home, the doctor signs the release forms, and the nurse helps the patient leave. Once the nurse is finished with their procedures, the patient simply leaves the hospital.

If the patient is admitted to an inpatient unit, the doctor in the inpatient ward is given a white card. The clerk sends the white card to the bed coordinators. The bed coordinators track the inpatient ward beds to find an opening. Once an opening is found, the patient is assigned the bed. Once the patient is ready to be moved from the ED, a transporter is called to move them. In certain cases, a nurse or doctor may also accompany the patient if medical attention is required en route to the ward.

If the patient is to be transferred to a different hospital, the steps are similar to when the patient is admitted to an inpatient unit. The difference is that the bed must be found at the hospital where the patient is being moved before the patient can leave the ED at UMMC. Once a bed is found, a private ambulance from the hospital the patient is being transferred to can be called to move the patient from UMMC.

In all cases, once the patient has been discharged and physically removed from the ED, the bed must be cleaned. There is a housekeeping staff assigned to the ED that is notified of a dirty bed through the ED intercom as well as the computer system. Once they clean the room, the bed’s status is updated in the system and a new patient can be assigned to the bed.

Precise observations on the roles, activities, and hours of the personnel, labs, and other components of the ED is essential to creating a valid model representative of the UMMC ED. Although observable information was collected about all staff member roles and other aspects of the patient throughput process, only several components (such as doctors, residents, and bed availability) were assessed through manual data collection.

Data Collection

The first step toward constructing an accurate simulation model of the emergency department (ED) was obtaining the data necessary to define the parameters and scope of the model. Once the resources and processes in the ED were identified, data was collected to represent each of the data points required by the model. This data encompassed every aspect of operation in the ED, including available equipment such as beds and laboratory machinery, medical personnel operating in the department, and detailed demographic and medical information on the patients coming through the ED. However, only several components were collected manually in order to maximize the relevant data to the residency model. The combined need for each of these data elements necessitated a two-pronged approach to the data collection – pre-existing data was obtained from the hospital’s historical databases, and in addition, the team collected certain data points manually over a period of time in the ED to supplement the database data.

Hospital Database Data Collection

Four months' worth of data, spanning from October 1, 2009 to January 31, 2010, provided the necessary volume to establish the basis of the model. This encompassed approximately 17,000 patient entries. The data set included the three weeks in January when the team manually collected data in the ED. This enabled us to link the patients present in the ED during manual data collection periods to their hospital database records.

No directly identifiable information was collected about patients. However, the data collected was indirectly identifiable because exact patient entry and exit times for the ED can be traced back to an individual patient. We received the historical database data from UMMC on a flash drive. It was saved in Microsoft Database Format. We converted this to a comma separated variable format, which could be more readily used by our scripts. The script "historical db insert" was run on this data, and a new database was created with the results. This script took care of a few parts of the data that needed cleaning. The script contained a few handlers that turned string values into more useful integer values. For example, one handler checked to see if the status of a lab test was completed. Any completed lab was assigned an integer value of 1, while all other entries received a 0. Some other handlers checked to make sure that items which were supposed to be integers (such as severity score) were in fact integers. Those places that were not integers were changed to null values. These handlers served as a way to ensure the data was in the correct format (datetime, integer, and string) and entered properly. It was a primary filter to ensure that all data was properly entered and that all incorrectly formatted data was treated as a null value. Table 8 shows all database data titles used, as well as their format and a brief description:

Title	Format	Description
Encntr_ID	Integer	A primary key to describe that patients entire visit to the ED
Lastname	String	The patient's last name
ARRIVALDATE	Datetime	The date and time that the patient arrived in the ED
TRAIGE_BEGIN	Datetime	The date and time that triage began
TRIAGE_END	Datetime	The date and time that triage ended
SEVERITY_SCORE	Integer	The number (from 1-5) assigned as the patient's severity after triage
NAWCDATE	Datetime	The date and time (if any) that the patient was offered a bed and did not respond
FIRSTBEDDATE	Datetime	The date and time that the patient is given a bed.
FIRSTBED	String	The name and number of the bed the patient is placed into
BEDLIST	Colon separated list	List of all the beds that the patient was placed in during his visit to the ED
PROVIDER	String	Name of the doctor assigned to the patient
PROVROLE	String converted to integer	The title of the doctor assigned to each patient, which was converted then to an integer denoting attending and residents.
PROV_DATE	Datetime	Date and time that a provider was assigned to the patient
DESC_TO_ADMIT	Datetime	Date and time that the doctors decided to admit the patient
DISCHARGEDATE	Datetime	The date and time that the patient is actually discharged from the ED

Table 8: UMMC Database Data Titles and Descriptions

In addition to these fields, we also created two more fields called numlab and numrad, which totaled the number of labs and the number of radiology tests from each patient. Although the historical database provided a wealth of information required for the simulation, there were admittedly several inconsistencies and "quirks" associated with the database.

Patient Database

In the patient database, there were no null values recorded for the discharge date, meaning a discharge date was recorded for every single patient. Given that the original query for the database was based on available discharge dates, this is not surprising. However, out of 16565 patients, 55 of these had discharge dates that were recorded as January 1, 1900. These dates were clearly not correct, and it is likely that they simply served as placeholders for various possible reasons, such as lack of knowledge of the actual discharge date. In the patient database used by Team HOPE, these placeholder dates were recorded as null.

The triage times also presented several oddities. There were 20 patients who lacked a triage begin time and 763 patients who lacked a triage end time. It is not immediately obvious why a patient would lack one of these times. It is possible that certain patients were rushed into the ED, and that in the midst of trying to quickly provide care to the patient, the task of recording the triage times was simply neglected. Furthermore, 9 of the patients had triage begin and triage end times that were equivalent, that is, the triage took no time. Again, it isn't clear why no time at all would elapse between the start and end of triage. The fact that such this happened for only a small number of patients suggests that it may simply be another clerical error.

There was also a small number of patients whose arrival dates occurred after their triage times or after they received their initial beds. 54 patients seemingly arrived after their triage begin times, while another 19 patients arrived after their triage end times. 13 patients arrived after receiving beds. The number of patients with inconsistencies associated with their arrival times was small, suggesting mere clerical error as the reason for these inconsistencies.

Some patients also had triage times that occurred after receiving a bed. 5 patients had triage begin times that occurred after they received a bed, while a much more significant number of patients – 1760 – had triage end times that occurred after they received a bed. Clerical error remains a possible reason for these inconsistencies, but the high number of patients with triage end times occurring after being offered beds suggests something less trivial. It is possible that these patients were brought into the ED in a bed, and that these bed times were subsequently recorded first before the patients were properly triaged.

There were also implications regarding patient response and the number of patients who received an initial bed. 1512 patients were classified as NAWC (no answer when called), meaning those patients did not respond when called for triage into the ED. Interestingly, 2640 patients never received an initial bed. This implies that some patients who were admitted to the ED were never offered beds; possibly they were simply treated in the hallways of the ED. Another 186 patients were classified as NAWC but also received beds in the ED. Perhaps these patients were called, did not initially answer, but then returned to the ED for care. Another 104 patients were classified as NAWC after receiving beds. The reason for this is not obvious. For those patients who did receive a bed, there was at least always a bed date and bed list.

A high number of patients – 2702 patients out of 16565 – seemingly never had a provider (e.g. an attending physician). Moreover, 361 of these patients received beds even though they never had a provider. It's unclear why a patient would not have a provider listed. There are a number of possible reasons: clerical errors, lack of need for a provider, patient care provided by a non-ED staff member, etc. All patients with providers did have provider dates recorded, which may be the time at which the provider officially started to provide care.

A significant number of patients – 4027 patients out of 16565 – had a null severity value. The high frequency of this occurrence suggests something less trivial than a clerical error. It is possible that these patients had a severity too extreme for the scale used by the ED. For instance, the patient may have had a severity that would be classified as something greater than 5 or less than 1.

Lab and Radiology Databases

The origin time and status time for all labs and radiology entries were nonnull. There were 44 instances when the status time occurred after the origin time in the lab database. A provider was always listed for all entries in both databases. Overall, the data for these two databases was internally consistent.

Title	Format	Description
Enctr_ID	Integer	A way to link back to the patient, which identifies the specific visit to the ED
Order_ID	Integer	The primary key for labs and radiology tests, it is the number that uniquely identifies the test
Order_Mnemonic	String	The name of the test ordered
ORIG_ORDER_DT_TM	Datetime	The date and time that the test order was placed into the computer
Order_status	String converted to integer	This is a status update of the lab telling whether it was completed or not, all completed labs were given a number 1 and all others were given a 0.
STATUS_DT_TM	Datetime	Date and time that the order_status was completed

Table 9: Lab and Radiology Database Data Titles and Descriptions

Manual Data Collection

While the historical database data was voluminous and informative, it did not achieve a level of detail necessary for a successful analysis of the effectiveness of the resident teaching model. The databases were useful for acquiring general patient and bed availability data, but in analyzing the residency model, the team needed specific timestamps accounting for the doctors' and nurses' activities as they pertained to patient care. By actively monitoring the medical personnel and patients in the ED for a period of time, the team was able to fill in these gaps in the database data, allowing for the creation of a more valid simulation model.

Logistics of Data Collection

The manual data collection was categorized into two parts – patient data collection and doctor data collection. The first step in the process was to determine the volume of manual data necessary, and accordingly set a schedule for data collection.

For the patient data collection, in order to maintain continuity with patients, we decided to station data collectors in the ED for two 72-hour periods on rotating shifts monitoring four specific bed regions: beds 1-4, 5-8, 9-14, and 41-44. These beds were chosen because they were regular patient beds and therefore likely to see the highest patient traffic in a given time period. The patient data was collected in two Wednesday-Friday sessions during two separate weeks in January 2010.

The doctor data collection was conducted over 10 six-hour shifts spread across six days – two Saturdays, one Monday, one Tuesday, and two Wednesdays. Eight of these shifts were connected into 12-hour periods to observe the transition between department attending physicians, who each work eight-hour shifts. Doctor data was collected over three weeks in January 2010, though the doctor data collection days did not overlap with patient data collection days.

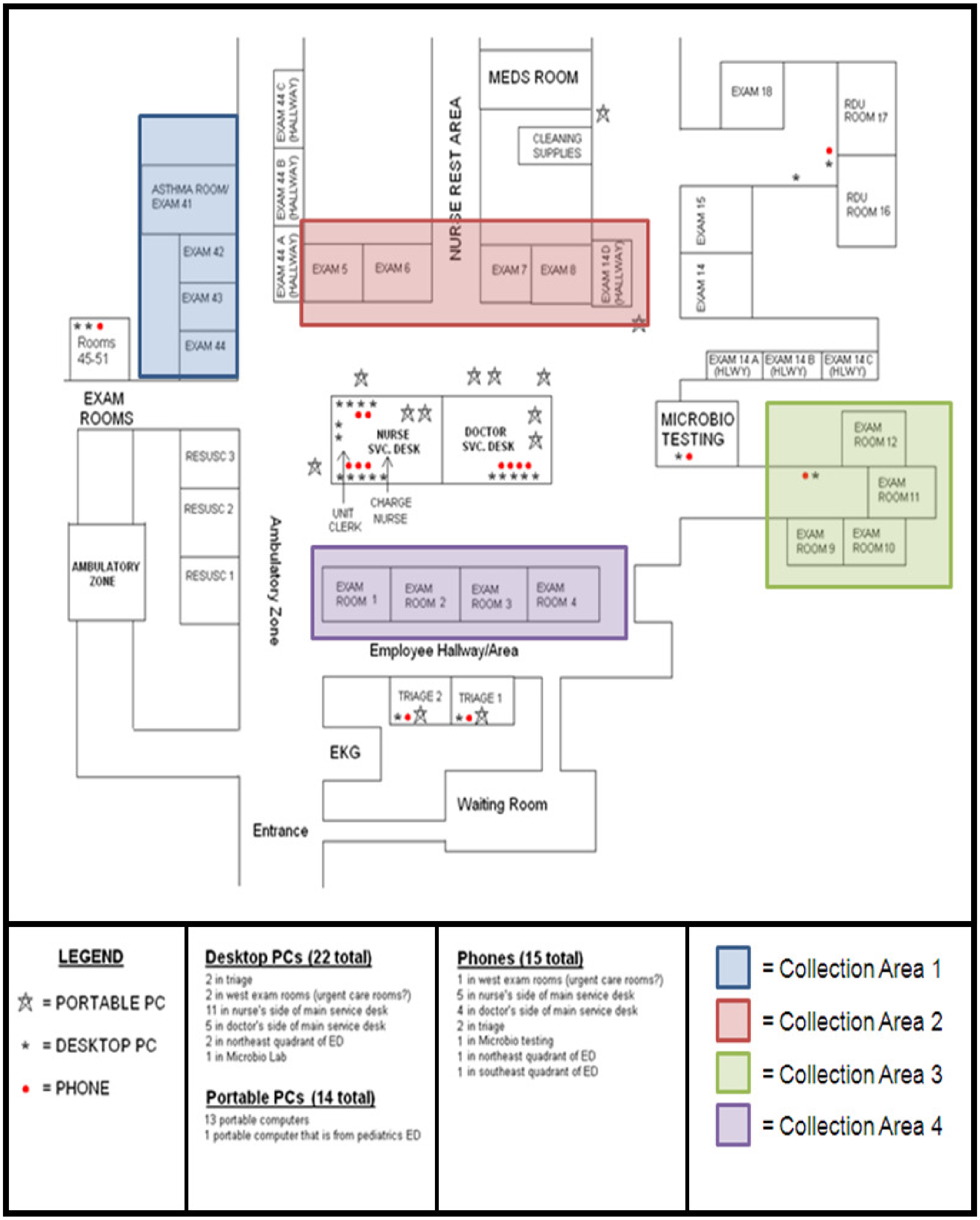
This specific data collection schedule was established to maintain a reliable spread of data. Timestamps were collected on different days of the week over several weeks. Major holidays were avoided to prevent skew in the data. The collection schedule was also reviewed by Michael Harrington and Dr. Hirshon.

For doctor data collection, two Wednesdays' worth of data were collected because residents attend mandatory conferences on Wednesday mornings. Therefore, on those days, the attending physicians perform all required tasks pertaining to patient care, instead of delegating tasks to the residents. This was especially critical for our simulation model, since we are specifically targeting the resident education model as an area of inefficiency. Collecting additional Wednesday data enabled us to more effectively compare the effect of having attending physicians perform certain tasks instead of residents, and vice versa.

Both doctor and patient shifts were staffed by four to six collectors each – two of these collectors were team members, while the other two to four were recruits hired by the team. The team advertised the data collector positions on several listservs on the University of Maryland, College Park, campus, and selected 16 responsible students with good academic records. The recruits were provided with detailed information packets and data collection sheets, and were required to attend a training session hosted by the team.

In addition, each recruit underwent HIPAA and CITI training and signed a waiver asserting they would maintain patient confidentiality.

The team's previous observations in the ED established a basic floor diagram that allowed us to plan where to station the data collectors. The 5-8 and 41-44 bed regions were monitored from the back left corner of the nurses' area, while the 1-4 and 9-14 regions were monitored from the front right corner of the doctors' area. These locations were chosen not only for convenience in monitoring the beds, but also to ensure the data collectors would not alter the flow of patients or hinder the work of the medical personnel in any way, thereby maintaining data validity. During doctor data collection, the collectors either followed the residents and attending physicians to different patients' rooms or observed the doctors at their service desk, again taking care to remain unobtrusive and only ask questions when the doctor had completed the task at hand. These proposed locations were presented to and affirmed by Gail Brandt, the Nursing and Patient Care Services Manager for the ED.



LEGEND

☆ = PORTABLE PC

* = DESKTOP PC

● = PHONE

Desktop PCs (22 total)

2 in triage
2 in west exam rooms (urgent care rooms?)
11 in nurse's side of main service desk
5 in doctor's side of main service desk
2 in northeast quadrant of ED
1 in Microbio Lab

Portable PCs (14 total)

13 portable computers
1 portable computer that is from pediatrics ED

Phones (15 total)

1 in west exam rooms (urgent care rooms?)
5 in nurse's side of main service desk
4 in doctor's side of main service desk
2 in triage
1 in Microbio testing
1 in northeast quadrant of ED
1 in southeast quadrant of ED

🟦 = Collection Area 1

🟥 = Collection Area 2

🟩 = Collection Area 3

🟪 = Collection Area 4

Figure 2: Map of UMMC Emergency Department

Our data collection periods began at either midnight or 7 a.m. Every day at 7 a.m., the nurses and doctors have their respective transition meetings to smooth the shift change. On days when the data collectors began at 7 a.m., they attended the transition meetings to identify the personnel by name and region of duty. For example, every nurse was assigned to a range of beds or designated as a swing nurse; similarly, the attending physicians were split between the northern and southern ends of the ED, while the residents were assigned different colors. Collectors who began their shifts at midnight had the advantage of lower patient and personnel volume, and were able to ask the charge nurse to point out the key medical personnel on the floor. During data collection shift changes, the departing collectors were responsible for identifying the personnel to their replacements.

For every action observed in the emergency department during doctor and patient data collection periods, the data collectors approached the doctors, nurses, technicians, and/or transporters involved in the action to request additional details. For example, the collectors inquired as to which lab tests were performed, what equipment was used, where the patient was being transported or discharged to, etc. While these observations were not assigned specific activity codes on the data collection sheets, they were noted in the comments section for later review. The activity codes will be detailed below, in the data points section.

Data points

While in the emergency department, the data collectors made note of specific data points detailed on the data sheets the team provided to them (see Appendix for blank and sample sheets). Patient and doctor data collection each necessitated different activity codes.

During doctor data collection, each data collector was assigned to a doctor. As noted in the throughput section of the methodology, at any given time, two attending physicians, one senior resident, and two interns, also known as first-year residents, staff the emergency department. There is generally also a third type of resident called a swing resident, who is a third- or fourth-year resident. At the top of each doctor data sheet, the doctor’s name and position was noted.

In any department, a doctor’s responsibilities encompass more than the patient’s physical examination. They are also required to fill out paperwork, enter medication and test orders into the computer, and consult with other doctors and nurses. Through the doctor data collection, we attempted to capture the entirety of a doctor’s responsibilities, thereby adding another layer of complexity to our simulation model.

When observing doctors, collectors noted time stamps and codes of one through nine for the following activities: visiting a patient for the first time, writing on a patient chart, using a computer (noting the program used), going on follow-up rounds to patients’ rooms, talking to fellow residents or attending physicians (noting the type of doctor), talking to nurses (noting the nurse’s region), using the phone, speaking to a unit clerk, and speaking to an EMT or other personnel. For each timestamp, the bed number in question was also noted. As mentioned in the logistics section, where the collector was unsure of the activity performed or the activity suggested additional details, they requested the information from the doctor.

By nature of level of experience, attending physicians and residents perform different tasks with varying frequency. For example, when residents are present in the ED, which is every day except Wednesday, they are almost exclusively responsible for entering medicine and lab test orders into the computer and for filling out discharge paperwork, though both acts require the approval of the attending physician. Meanwhile, the attending physicians spend a greater amount of time analyzing the lab results. Both residents and attending physicians spent a substantial amount of time discussing patients to test the residents' level of understanding – these discussions serve as informal teaching sessions and are critical in the resident teaching model.

On Wednesday mornings, however, when residents leave for mandatory conferences, the attending physicians assume responsibility for all patient paperwork and orders. In addition, this eliminates the teaching sessions with the residents, leading to a shorter decision-making interval. These factors could potentially result in shorter lengths of stay for patients. However, the effect of having only attending physicians in the ED on Wednesday mornings may be diminished by the lesser patient volumes on those mornings. By collecting data on different days of the week, we were able to capture these elements in the model.

During patient data collection, each data collector was assigned to one of the four previously mentioned bed regions, and noted the region at the top of the data sheet. In watching the rooms, the collector noted the activity that corresponded to every entry and exit of a person from each room. The collectors assigned time stamps and codes from one through nine for the following activities: attending physician visit (noting north or south), senior resident visit, other resident or intern visit (noting type), nurse visit (noting nurse region), transport (noting destination), consulting medical student visit (noting the ward s/he was visiting from), consulting physician visit (noting the ward s/he was visiting from), patient admission to bed, and patient discharge from bed. As data collection went forward, codes were also added for registration clerks, housekeepers, and patient care technicians out of necessity. For each time stamp and activity code, the bed number in question was also noted.

As mentioned in the logistics section, where the collector was unsure of the activity performed or the equipment used, they requested additional information from the nurse, doctor, technician, or transporter involved in the activity. This level of data collection assigned concrete time stamps to employees' actions and also allowed us to track the progress of individual patients in the ED.

In order to connect our manually collected patient and doctor data to the historical database data, we also had to record the financial identification (FID) numbers of the patients present in the ED during our data collection periods. A flat-screen monitor above the doctors' station displays the FID numbers for the patients in every occupied bed of the ED. Every hour, the team member in charge during that shift wrote down the FID numbers of all patients present in the ED at that time with the patient's corresponding bed number. This number enabled us to link our manually collected data to the database data, thereby allowing us to validate the data.

Additionally, to add another level of detail to our simulation model, we obtained personnel schedules from Mr. Harrington. The schedule for attending physicians specifies which attending is working which shift on a daily basis. The residents' and nurses' schedules are constantly rotating, however, thereby providing us with the different shifts worked by each type of personnel.

Processing the data

Following the data collection periods, the team members input the information from the data sheets into comma-separated values (CSV) files, which are used to digitally store data structured in a table of lists form, where each associated item in a group is separated by a comma from other items in that same set. We chose this file format because it is easily commutable between spreadsheet programs such as Excel, which are user-friendly for entering the data, and scripting languages such as Python, which was used to insert the data into a MySQL database.

Once the data was typed up and all the CSV files were uploaded to the team's file repository, we cleaned the entire data set. We wrote and implemented Python scripts designed to pinpoint errors in the data sheets, such as inconsistent formatting and missing time stamps, bed numbers, or action codes. For each faulty time stamp of data, we referred to the comments section of the data sheets to assign activity codes where possible. However, lines with missing time stamps and bed numbers were eliminated, since there was no reasonable technique for inferring the correct entry.

The data sheet comments were also used to refine the data and make it more robust by assigning new activity codes with a greater level of detail. For example, codes were introduced for bed cleaning after a patient room was vacated, or for interactions with police responders in certain cases.

After the data was refined, we had to validate it before we could incorporate it into the simulation model. In the first step of this two-part process, we matched patient bed entry and exit times from the patient data sheets to the changes in FID numbers for the same bed on the corresponding FID data sheets. We followed this procedure to ensure that when a data collector observed a patient permanently leaving a bed on a patient data sheet, this was also reflected and confirmed in the FID data copied from the flat-screen during the same time period, thereby checking for and correcting human error.

As established earlier, there were two states of data collection: following doctors and monitoring the events occurring on a subset of patients in the ED. The latter state required many data points to be obtained from the databases for each patient. From FirstNet, we obtained the priority number, bed number, timestamps and types of medications, labs, and radiology tests, as well as the timestamp of the patient exiting the ED bed. For the other phase concerning doctor-specific data collection, these data points were collected as needed to supplement our own data.

The second step toward data validation was to then link the FID numbers we collected in the ED in January to the historical data pulled from the hospital's databases. As an additional check, we compared the patients' length-of-stay noted in the database to the length-of-stay inferred from the patient data collected in person.

Once the manual data was connected to the database data using the relevant timestamps and patient FID numbers, the model was used to establish distributions comparing staff behavior in accordance with patient priority levels, etc. By quantifying the actions enacted by doctors, nurses, and technicians, we were able to implement a discrete event simulation model that enabled us to propose statistically probable chains of events for any given patient. For example, if a resident visited a patient and decided to order a radiology test, there would be an increased probability that the next person to enter the room would be either a technician with imaging equipment or a transporter to take the patient to radiology. Observing personnel and patients in the ED allowed us to build such a model.

Simulation Model Attributes

Patient Arrival

Patients in our simulation are created, or arrive at the hospital, according to an exponential distribution. The exponential distribution is the most widely used statistical function to mimic the real-life arrival rate of a person or object. An exponential distribution is used to determine the time between events in a Poisson Process, a mechanism in which events occur independent of each other and at a stable average rate.

The exponential distribution is parameterized by the average patient arrival rate for each given day and time period. To determine these rates, we used the historical data collected from the hospital's database, as discussed in the Data Collection section. Our historical data, for patients who entered the ED between October 1, 2009 and January 31, 2010, was divided into one-hour time periods. The smoothed graph of hourly arrival rates at the emergency department for each day and time period are shown in Figure 3:

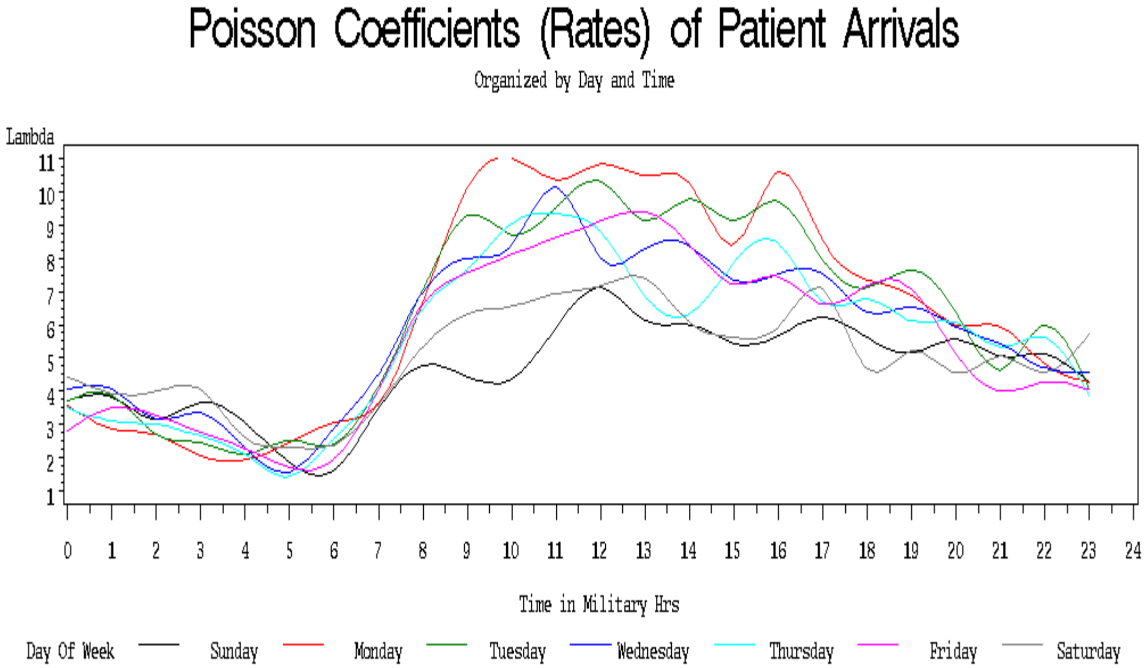


Figure 3: Rates of Patient Arrivals

Figure 3 shows some interesting trends in patient arrivals at the hospital emergency department. Although all the days have similar arrival rates between midnight and 7 AM at that hour the arrival rates diverge to different levels depending on the day of the week. The largest patient arrival rate occurs during the middle of the day on Mondays, while the weekend has the lowest arrival rates. Patient arrivals tend to drop off towards the end of the day.

Our simulation uses this distribution, with the appropriate coefficient for the exponential distribution drawn from Figure 3 depending on the hour of day being simulated, to randomly create new patients to enter the emergency department.

Patient Attributes

As soon as new patients are created by the simulation, they are immediately assigned the following attributes:

- Whether or not the patient will be sent to the ambulatory zone in the ED
- A severity score (1 through 5)
- Whether or not the patient will be admitted to an inpatient ward after his stay in the ED
- The amount of time it will take to triage that patient
- The number of lab tests the patient will have throughout his stay in the ED

Although in real life these attributes are not known until later in the patient's progress through the ED, by assigning these attributes to patients as they enter the department our simulation can more effectively predict the patients' paths through the ED. For example, patients who are admitted to inpatient wards tend to have longer stays in the ED. If our model did not assign this attribute when the patient first entered, then the patient's length of stay in the ED would not take into account this effect. The value of each attribute is assigned to a patient through based either on simple probability or on correlative distributions.

The simulation first determines whether the new patient will be sent to the Ambulatory Zone (AZ). This probability is determined based on the historical percentages of patients sent to that area. If the simulation assigns a patient to the ambulatory zone, then that patient essentially exits the simulation and is no longer considered for a bed. Our study did not cover the doctors in the AZ, and, as such, our simulation does not take them into account. Our simulation does account for the fact that the AZ is only open 2/3 of the time.

We assume that patients are selected to go to the Ambulatory Zone independent of how long other patients have been waiting. The triage nurse selects patients to go to the AZ purely based on how severe the patient is in order to fast-track the selected patient because that patient has an easily remedied ailment. Our simulation assumes that patients do not transfer between the AZ and beds in the actual ED.

After removing those patients who are sent to the AZ from the pool, the severity score number is assigned to the remaining patients purely based on the historical distribution of severity scores in the data we received from the hospital databases, as shown in Figure 4.

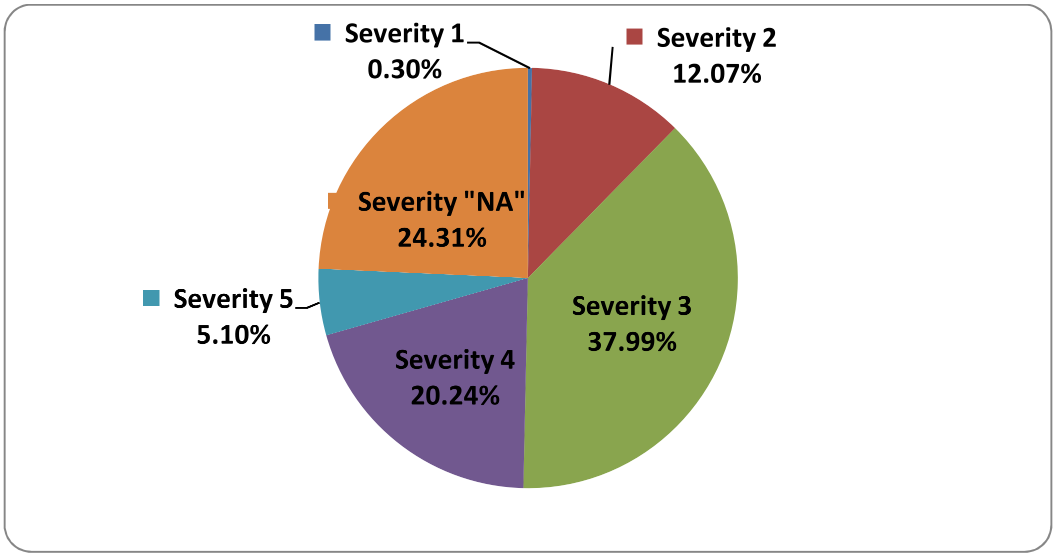


Figure 4: Severity of Incoming Patients to ED Waiting Room

To select a severity number for a new patient, one can imagine placing a spinner in the center of the chart in Figure 4 and spinning it. As such, a plurality of patients will be assigned a severity score of three, and the next greatest number of patient’s will be assigned severity score “NA.” About a quarter of the patients in the historical database did not have a severity score and were assigned “NA” instead. Based on the percentage of patients who are admitted to the AZ, around two-thirds of “NA” patients were high-priority patients and would normally be given a score of one or two, while the other third were very low priority patients.

For those patients that are not sent to the ambulatory zone, the simulation determines whether those individuals will be admitted into an inpatient ward based on their severity score. The process for determining this factor is similar to that of the previous attribute; historical data provides probabilities that patients of certain severity scores will be admitted to the hospital after their stay in the ED. We grouped the patients into categories of similar priorities – severity scores 1-2, 3, and 4-5 – and computed the probability of a patient being admitted into the hospital for each of those groups.

Both the amount of time it takes for a new patient to be triaged and the number of lab tests the patient will undergo during his stay are determined by a combination of the severity score of the patient and whether the patient will be admitted to an inpatient ward after he leaves the ED. Historical data was categorized into the various combinations of the previously mentioned attributes, such as the data for all patients with a severity score of three who were admitted to an inpatient ward after their stay in the ED. We then created histograms of this categorized data, and fitted well-known statistical distributions to the shapes of the graphs. The statistics of determining these curves are outlined in the Statistics section. Figure 5 and Figure 6 show some sample distributions for determining how long it takes to triage a patient with a severity score of “NA” and how many lab tests that patient receives:

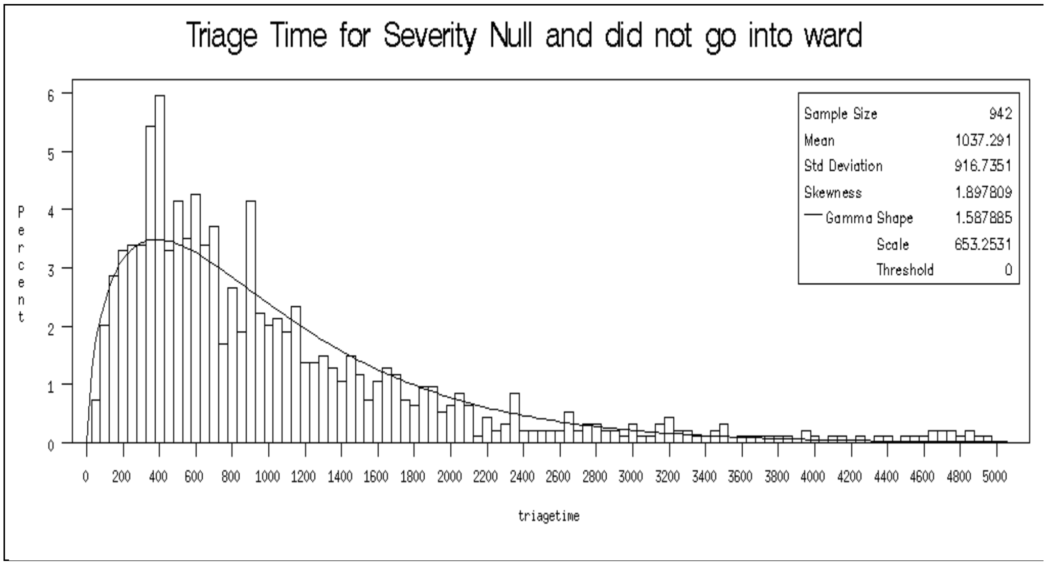


Figure 5: Sample Triage Time Statistics

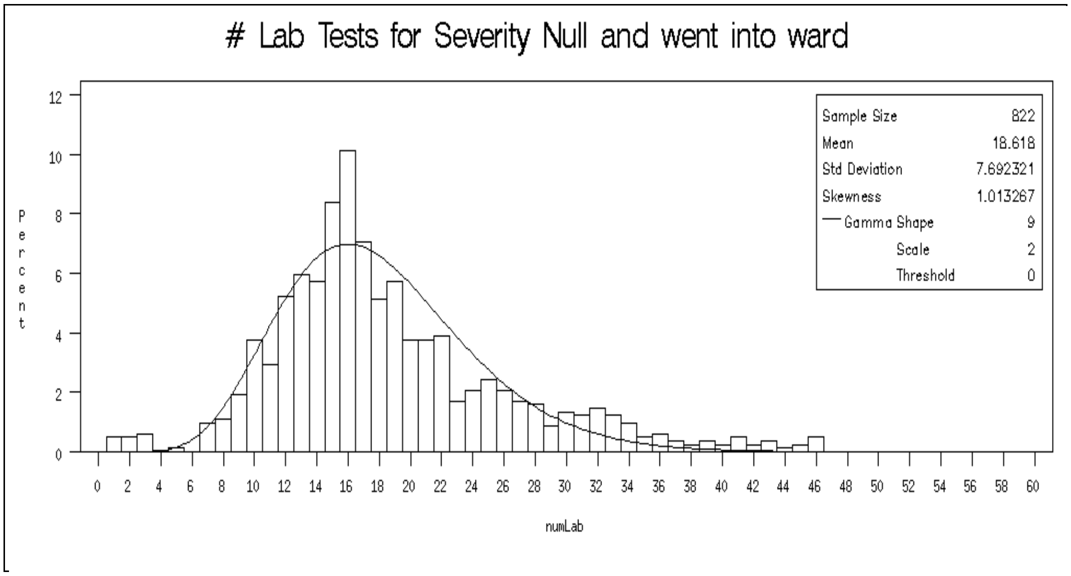


Figure 6: Sample Lab Test Statistics

These histograms were developed from the historical database data retrieved from the hospital. In Figure 5, each bar represents the percent of total patients classified as “NA” and who were not admitted to an inpatient ward who had a triage time of the number of seconds listed under the bar. Figure 6 refers to the percent of patients who received each number of lab tests. Both distributions are “left-skewed,” meaning that the bulk of the data is concentrated close to the 0 mark on the graph. Both graphs were fitted with a Gamma distribution that had its parameters altered to approximate the curve of the histogram itself. This distribution is discussed in further detail in the Statistics section.

Based on the patient’s previously assigned attributes, the relevant distributions for both triage time and the number of lab tests performed were used to assign those values to the patient. This distribution is a cumulative distribution function that is related to the probability distribution function for a given attribute. Values with higher densities in the distribution will have a higher chance of being assigned, while values with lower densities will have a lower chance. Once a patient has been assigned all five attributes outlined in this section, he is triaged by the appropriate nurse. Once the simulated patient has seen the triage nurse, he is placed in line for a bed.

Patient Bed Selection

Each time a new bed opens up in the ED, a competitive process begins between the patients in the waiting room; the triage nurse must decide which patient gets the open bed. The technique for establishing this probability of selection is called the analytic hierarchy process (AHP). The AHP is a frequently used technique in decision sciences that models how decisions are made in the face of complicated factors that resist quantification. This process involves pairwise comparisons between all possible solutions to a solution. The preferred option for this scale is given a score between 1 and 9, and the less preferred option is given the reciprocal of the other score. In this way, the product of the two scores is always 1. When indifferent between two options, each would be assigned score 1.

To use the AHP, we break our patients into multiple classes and consider these classes to be the alternatives in the decision making process. The classes are based on our deciding factors for patient bed selection: severity and amount of time waited. In particular, we break the patients into groups of severity (severity is unassigned, severity is 1 or 2, severity is 3, and severity is 4 or 5) and into groups of the number of times the patient has been passed over in selection for an empty bed (never having been passed over, having been passed over once, having been passed over 2 or 3 times, and having been passed over 4 or more times). Each of these categories was highly correlated with being selected, making the divisions a natural way to organize the data. A third attribute to further subdivide the data was not used due to data sparsity issues. Combining these options in every possible way, as show in Table 10, we are left with 16 classes of patients.

	Severity 1-2	Severity 3	Severity 4-5	Severity "NA"
Passed over zero times				
Passed over one time				
Passed over two or three times				
Passed over four or more times				

Table 10: AHP Category Selection

Whenever a patient is selected for a bed, that patient has outcompeted all other patients currently in the waiting room for the bed. More generally, that patient's class has outperformed the classes of all the other patients in the waiting room. By considering all bed selections in our four months of historical database data, we are able to determine the probability that one class will be selected over another.

Now, we need to map this probability of outperformance onto a ratio priority scale. For given classes A and B, A outperforms B with proportion p and B outperforms A with proportion 1-p. For the probability of these which is greater than 0.5, we mapped that probability to 16p-7, which was arbitrarily set based off a linear scale to map probability 0.5 to the priority 1 and the probability 1.0 to the priority 9. Since, multiplied, the two sides must equal 1 for us to have a ratio priority scale, the left side of the scale is determined by 1/(9-16p).

	0 Losses	1 Loss	2-3 Losses	4+ Losses
Severity NA	5.21	2.07	0.64	0.15
Severity 1-2	2.23	1.04	0.75	0.26
Severity 3	1.78	1.18	0.68	0.25
Severity 4-5	3.60	2.73	1.75	0.37

Table 11: Analytic Hierarchy Process Results with AZ Patients Included

	0 Losses	1 Loss	2-3 Losses	4+ Losses
Severity NA	5.89	1.58	0.58	0.18
Severity 1-2	5.19	3.25	2.07	0.52
Severity 3	3.11	1.91	1.26	0.63
Severity 4-5	0.64	0.31	0.32	0.19

Table 12: Analytic Hierarcy Process Results with AZ Patients Not Included

Using these pairwise scores, we are able to determine a priority scale in which each patient category is assigned a value between 1/9 and 9. As described in the statistics section, we calculate this value for class Q by taking the geometric mean of the pairwise score of Q competing against all other classes. A mean of 1 is the average weight. The higher the number, the quicker a patient will get called back compared to the average person. A score under one means the patient gets called back with lower probability than the average person.

At this point, we are able to determine when patients arrive, what their characterizations are, how long their triage takes, and their prioritization for getting a bed.

There is one final consideration for these patients – how to determine if the patient has left the waiting room before being selected for an ED bed. This event is commonly called a No Answer When Called (NAWC). Using probabilities from the historical database data, the simulation determines if a patient is a NAWC based on that patient's severity score and how long that patient has been in the waiting room (in hours).

Patient Length of Stay

To model the ED, we decided to use individual patient attributes and attributes of the ED to determine a single prediction of the time that a patient spends in an ED bed, or the patient length of stay (LOS). It is possible to create a model where the predictors of LOS were combinations of these attributes. However, we decided to capture the complexity of the ED by modeling individual decisions made by doctors about each of these attributes. We believe that this led to a more accurate and revealing model.

To obtain a patient's LOS, we divided the data in a tree-like fashion based on four categories: whether or not the patient entered into an inpatient ward after being discharged from the ED, whether or not the patient had labs processed during their stay, the severity score of the patient, and lastly the presence of residents (a binary Yes/No variable). See Figure 7 for a tree diagram of these subdivisions. This ultimately led to numerous subdivisions representing a different combination of values for each of these categories. Each subdivision corresponded to a subset of the data, which was used to produce a probability mass function of the patient LOS. In other words, a histogram was made of the distribution of LOS for a given combination of category values. Next, the empirical cumulative mass function was drawn from the probability mass function.

At this point in model development, we had two options for distributions. We could have used the empirical histogram, i.e. the bar graph drawn directly from the patient historical data. Alternatively, we could have fit the empirical histogram with a known distribution function, and use this function (with its calculated parameters) in the model. Ultimately, because some histograms were difficult to fit with a known function, we opted to simply use the empirical histograms. However, the graphs display the overlaid known distribution functions for reference.

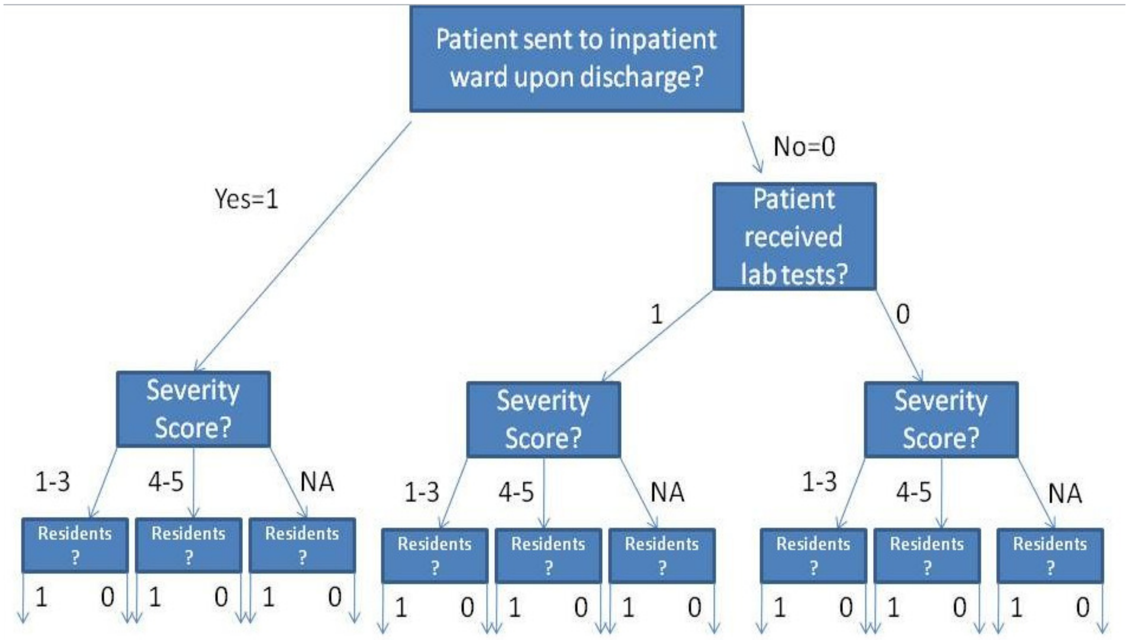

Patient sent to inpatient ward upon discharge?
Yes=1
No=0
Patient received lab tests?
1
0
Severity Score?
1-3
4-5
NA
Residents ?
1
0

Figure 7: Subdivisions of Patient Data

These divisions of the data were chosen not only for their relevance to the patient LOS, but also because they eliminated any data sparsity issues that we had. Any further divisions would have resulted in data subsets too small to be statistically significant.

Figure 8 shows an example of a significant division, where the resulting subsets were of reasonable size and the resulting distributions were very different.

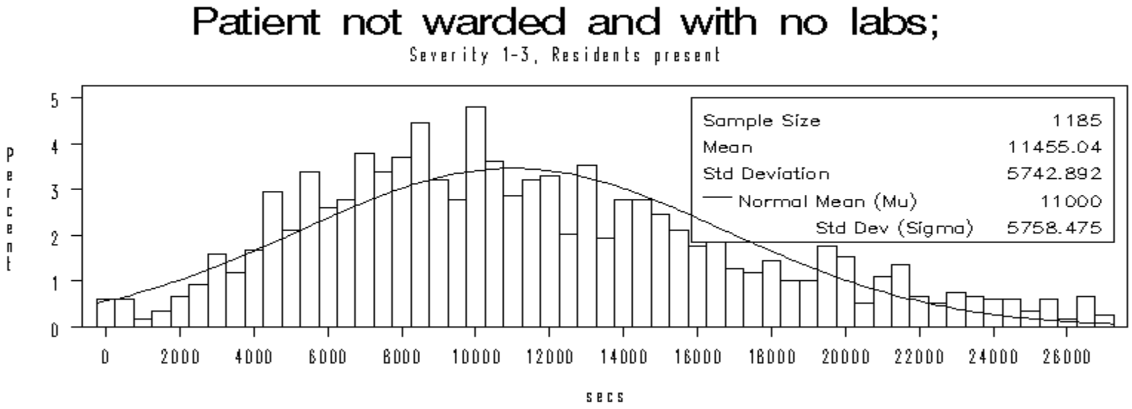


Figure 8: Sample Length of Stay Time Statistics

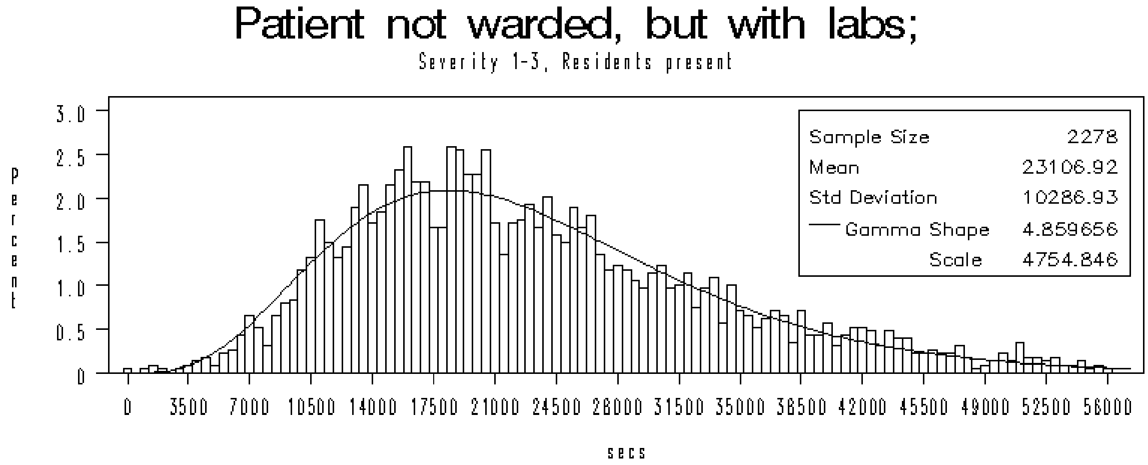


Figure 9: Sample Length of Stay Time Statistics

The histograms shown here were obtained using database queries that specified the particular attributes and retrieved all patients with those attributes. Figure 8 is the LOS time distribution for patients who do not get admitted to a ward, have no lab tests, have a severity score between 1 and 3, and have residents present during the time of their treatment. Likewise, Figure 9 is the LOS time distribution for patients who do not get admitted to a ward, have lab tests, have a severity score between 1 and 3, and have residents present during the time of their treatment. The only difference between the two figures is whether or not the patient has lab tests. Yet, the average length of stay for patients in Figure 8 is 11455.04 seconds while the average length of stay for patients in Figure 9 is 23106.92 seconds, indicating that this division was indeed important.

Statistics

The first division we made amongst the patients when creating these categories was whether the patient was admitted to an inpatient ward or not admitted to an inpatient ward.

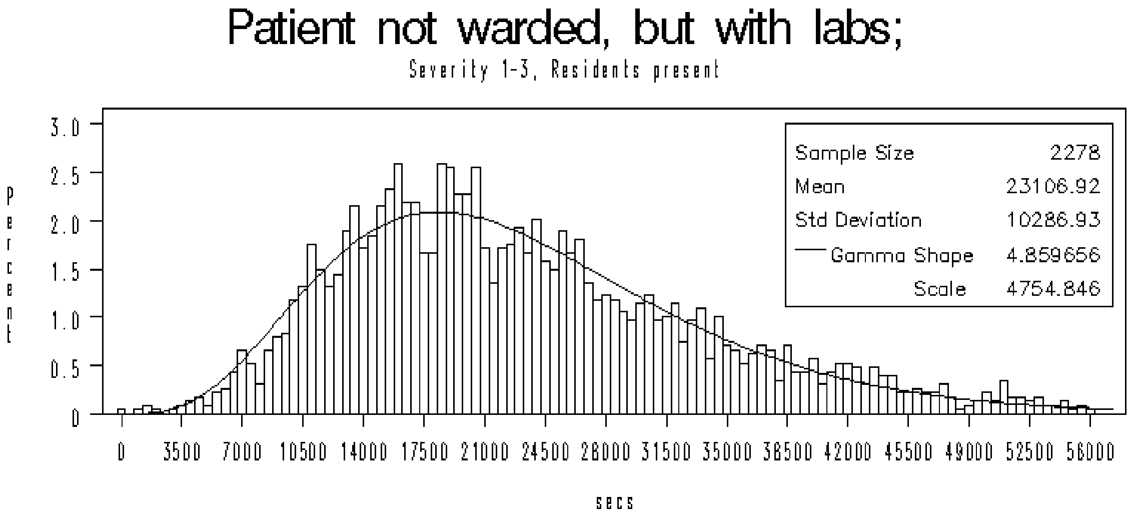


Figure 10: Sample Length of Stay Time Statistics

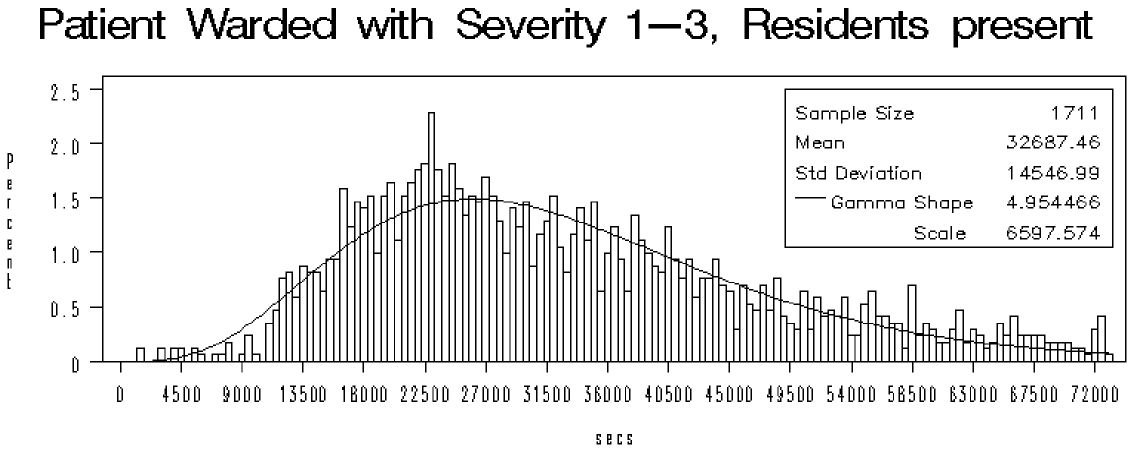


Figure 11: Sample Length of Stay Time Statistics

Figure 10 is LOS data for a patient who has lab tests, has a severity score of 1-3, has residents present during time of treatment, and is not admitted to an inpatient ward. The mean LOS for such a patient is 23,107 seconds. Figure 11 is LOS data for a patient who has lab tests, has a severity score of 1-3, has residents present during time of treatment, and is admitted to an inpatient ward. The mean LOS for such a patient is 32,687 seconds. The huge difference between these mean LOS values undeniably suggests that whether or not a patient was warded after discharge is related to the patient's length of stay, thereby justifying this subdivision..

The second division was also binary, based on whether or not the patient underwent any labs tests during his/her stay in the ED. Thus, the dividing variable had a value of 0 if there were no labs performed, and a value of 1 if there were any labs performed. If the patient was not warded upon discharge, then the lab test division was performed. However, if the patient *was* warded upon discharge, this division was not performed. This is because most warded patients possess serious or complicated medical complaints or history, thereby ensuring the ordering of labs. The division would have been inconsequential as only five patients in the data set were warded and had no lab tests. Such a division would have rendered any further divisions statistically insignificant because of data sparsity issues.

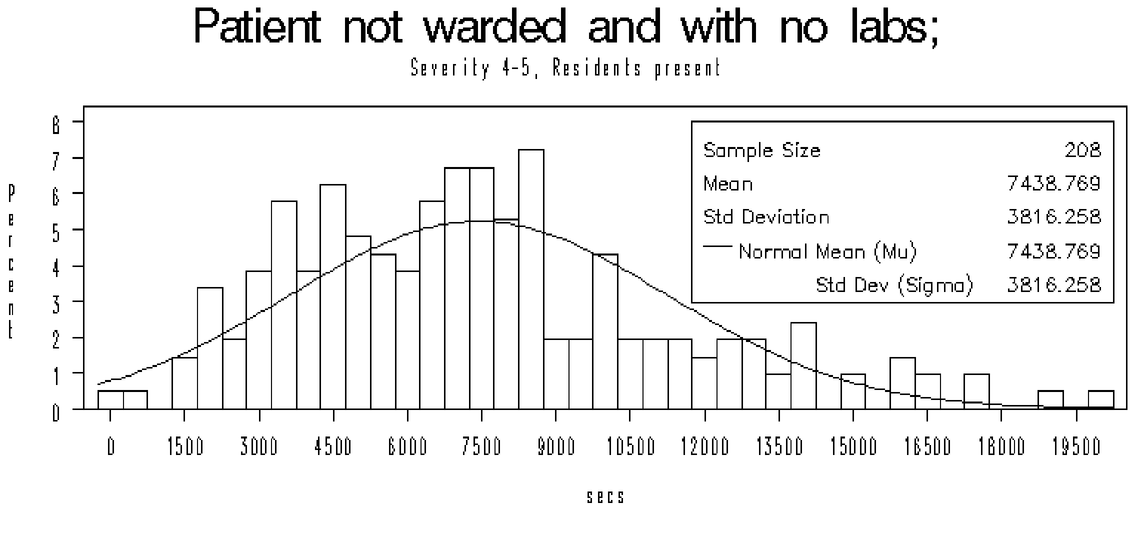


Figure 12: Sample Length of Stay Time Statistics

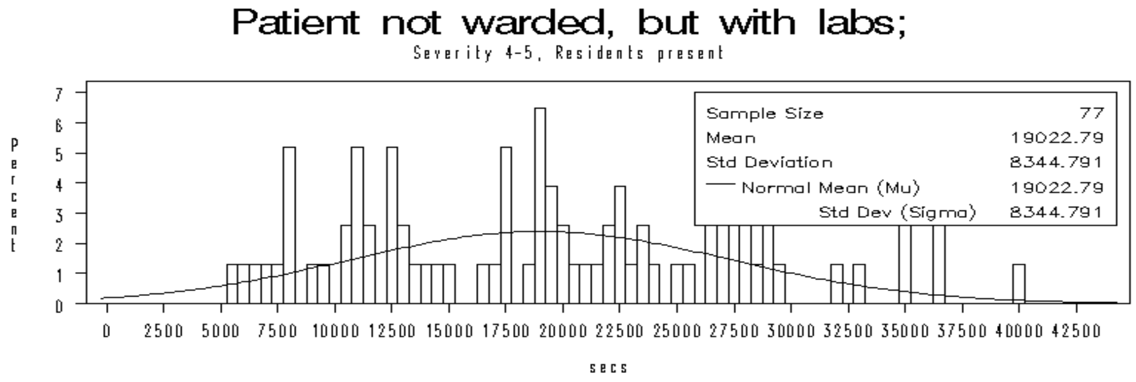


Figure 13: Sample Length of Stay Time Statistics

Figure 12 is LOS data for a patient who has no lab tests, has a severity score of 4-5, has residents present during time of treatment, and is not admitted to an inpatient ward. The mean LOS for such a patient is 7,439 seconds. Figure 13 is LOS data for a patient who has lab tests, has a severity score of 4-5, has residents present during time of treatment, and is not admitted to an inpatient ward. The mean LOS for such a patient is 19,022 second. Based on these figures, it is evident that if a patient has lab tests performed, then that patient's total LOS would be higher than if the patient does not have lab tests performed. In fact, the calculated Pearson's correlation coefficient between number of labs and total LOS was 0.64378 ($p < 0.0001$) suggesting a reasonable correlation and justifying this subdivision.

The next division we made concerned the patient's severity score, which was implemented using three possible categories. Patients with a severity score of 1, 2, or 3 were grouped together, and patients with a severity score of 4 or 5 were grouped together, while patients who had an NA severity score constituted a third set of patients. Patients were split this way for numerous reasons, the first being that it was natural to divide based on generalized higher and lower severities. Secondly, various metrics, such as number of lab tests as described earlier, showed similar patterns for severities 1 through 3, and another distinct pattern that applied to severities 4 and 5.

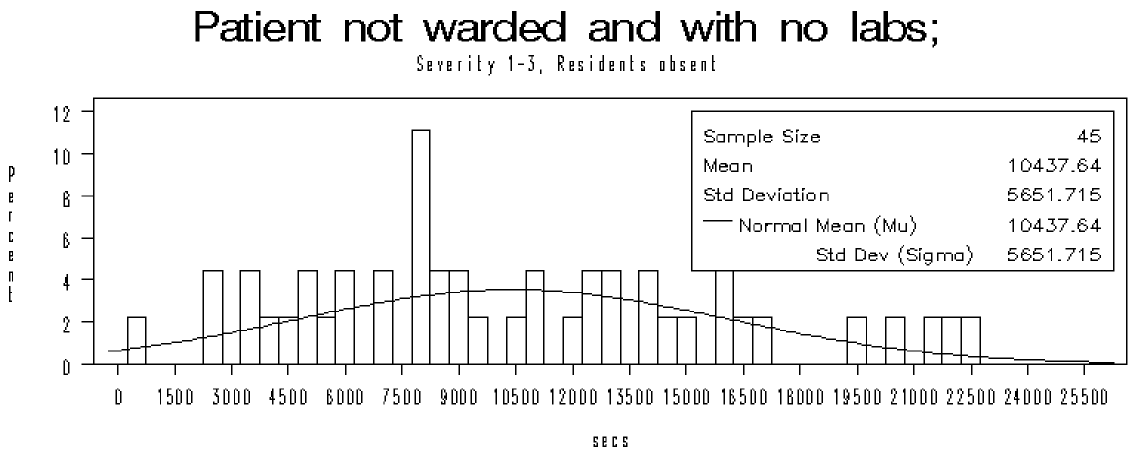


Figure 14: Sample Length of Stay Time Statistics

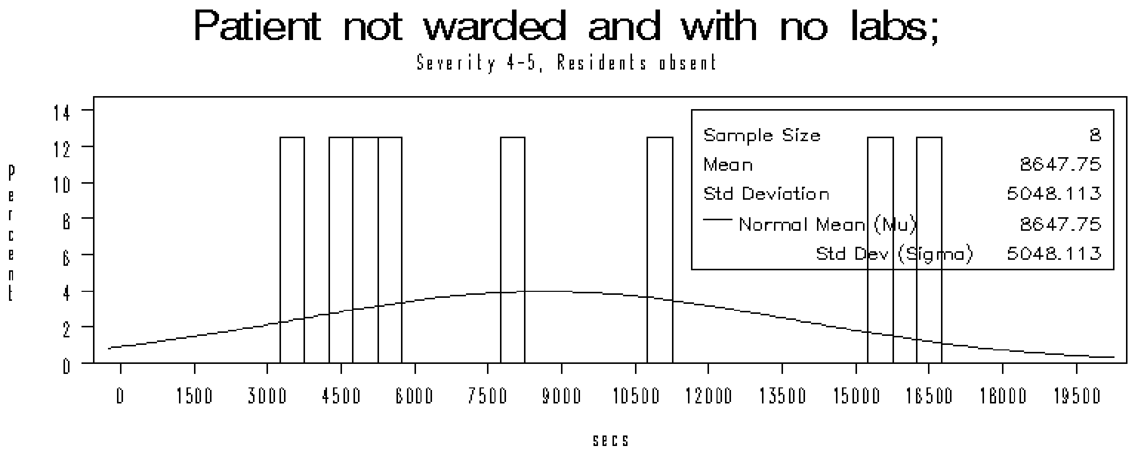


Figure 15: Sample Length of Stay Time Statistics

Figure 14 is LOS data for a patient who has no lab tests, has a severity score of 1-3, has no residents present during time of treatment, and is not admitted to an inpatient ward. The mean LOS for such a patient is 10,438 seconds. Figure 15 is LOS data for a patient who has no lab tests, has a severity score of 4-5, has no residents present during time of treatment, and is not admitted to an inpatient ward. The mean LOS for such a patient is 8,648 seconds. Furthermore, the correlation coefficient between patient LOS and their severity was -0.23253 ($p < 0.0001$). While the correlation coefficient is not optimally large, the p-value is very small for a sample size of more than six thousand; such a coefficient would justify this subdivision.

The last distinction made was whether residents were present during the time of the patient's treatment. This was also implemented using a binary variable, with 0 for "no resident" and 1 for "resident." A resident was determined to be "present" for a given patient based on whether or not there were residents working in the ED when the patient entered. In particular, if the patient entered the ED at any time other than 7am to 1pm on Wednesday mornings, the resident was deemed to be present.

The primary goal of our research was to determine the effect of the UMMC ED Residency Model on patient throughput in the ED. Without a division in the bins concerning whether residents were present or not, we would not obtain any valuable data to test our hypothesis.

However, as is further described in the experimental design section, a major technique we use in our simulation modeling is to predict the length of stay for a patient who is visited by a resident *outside* of Wednesday morning. To do this, we need to capture other effects that may affect a resident's work efficiency. As was suggested through our literature review, residents may be adversely affected by congestion in the ED. This occurs when the ED is very crowded, often near mid-day on a weekday. To be able to capture the effects of congestion in the ED on residents, we divided our data based on whether or not the ED was congested when the patient arrived. To measure congestion, we chose to use the metric of number of people in the waiting room. This made the metric simple to collect from our database data and clearly has an effect on the congestion in the ED. We chose to classify the ED as “congested” if at least 4 people were waiting in the waiting room for an ED bed. The ED is classified as “not congested” if 3 or fewer people are waiting. This division was selected to create divisions of roughly even size – in the database data there were provided, 52.3% of patients entered a room during a time classified as “congested,” while the remaining 47.7% entered during non-congested times.

Overall, these five divisions led to the creation of 30 categories, and for each a distribution was drawn. The data used for these distributions came from the historical database. We sampled the distributions empirically to determine a predicted value for average LOS in the ED for each patient.

Model Validation

After the creation of our model, we need to validate it. Only after our model is shown to be accurate can our results be meaningful. We used two different techniques to validate our model. The first technique we used was a graphical user interface, or GUI, which allows us to visualize our model and verify its validity. The second technique we used was comparing the similarity of our model’s output data to actual historical hospital data.

Graphical User Interface

The graphical user interface (GUI) allowed for a quick comparison between our simulation and the patient flow in the ED by visualizing the model. Visualization techniques allow for large amounts of data to be summarized and displayed to an observer in a quicker amount of time than if only the data was displayed. In addition, visualizations allow the researcher to steer the simulation and make changes to the simulation protocols without having to run another iteration. Finally, our visualization will enable us to quickly verify that our simulation mimics the emergency department, thus giving our results validity, and allow us to demonstrate the effect of any changes we make to hospital protocol.

Our GUI was programmed in the Java programming language. Java was selected for a multitude of reasons. The language is extremely versatile and works across many platforms; thus, our simulation can be run on a normal PC, Unix-based operating system, handheld device, or almost any other computer available. This versatility makes the model portable and allows for easy transfer between devices with almost no setup required for the receiving device.

Java also has a widely used application programming interface (API). The API is the mechanism by which programs access and utilize the library of a programming language. Java's API is well known among developers, and thus can be easily edited by future users of the simulation who wish to make improvements or modifications. Java can also be easily integrated into a web application. Our simulation could be built into a website for demonstration to hospitals across the country or to all the doctors in the emergency department in one hospital. The versatility of Java made it the best choice for the visualization of our simulation.

The Java GUI only outputs the data produced by the Python simulation. In order to connect the two programs, we had to write additional code. The Python simulation produces a “token” or string of code, each time a resource or patient is moved or used. That token includes an identifier of the resource or patient, the action being performed, the simulation timestamp, and the end location of the movement, if one occurred. The token is sent to the Java code, which deconstructs the string into its separate parts and displays the movement graphically based on the data in the token. This transmission protocol was uniquely developed for our project.

The Java code runs significantly slower than the Python code, however, because the GUI has to translate all of the simulation data into a graphical format and output it to the screen at a speed much slower than the Python code could run. In our first trial runs of the GUI, data would often get lost between the Python and Java code because the former code would be so far ahead of the latter; the Java could not backlog all the data in its buffer before outputting the graphics. To combat this problem, we added a subroutine into the Java code that halts the Python code when it starts getting too far ahead of the GUI. To prevent screen flickering when new data is displayed in the GUI, we added an off-screen buffer of the Java-produced graphics.

We added features to the GUI to make it more user-friendly. The background of our visualization is a reasonably accurate representation of the University of Maryland Medical Center, so it was easy for hospital administrators to see the simulation at work. We represented humans in the GUI using either avatars or dots. We also allowed the user to vary the speed of the GUI while the simulation is running, so the user can either look at the movement of one resource in slow motion or allow the simulation to run at almost full speed.

Using the GUI, we were able to visualize how our model's ED resources were being used, as well as how patients were moving through the ED. The GUI will become increasingly useful as we move toward capturing doctor decision-making in future work, making it a very useful tool both in this work and in the long term.

Output Data Analysis

The data output of our model, if valid, should closely resemble the actual historical data from hospital databases based on overall metrics. To determine if our model's output data was statistically similar to real hospital data, we compared the following parameters: patients per bed per day, NAWC rate, time until first bed for a patient, and total time from arrival to discharge for a patient. Producing output that was statistically similar to real hospital data was the primary indicator of a valid model. We chose the comparison metrics because they describe overall statistics for the ED. Furthermore, from a practical standpoint, they were the metrics that were readily available, having been obtained from historical database data. We simulated hospital operations for 30 million seconds, or slightly under a year of simulation time, to obtain average values for each of the metrics. Since we simulated operations over such a long period of time, and the average values of these metrics were robust, and we could reliably compare the values to the real life parameters.

Our model's metrics in the simulation matched closely with their counterparts in the historical data. Table 13 shows these similarities.

Metric	Historical Value	Our Value
Patients (per Bed per Day)	2.35	2.39
NAWC Rate	8.02%	7.76%
Time to First Bed	4819 seconds	4870 seconds

Table 13: Output Comparison between Model and Historical Values

Last, we used the Kolmogorov-Smirnov (K-S) statistic to quantify the similarity of the Time to First Bed and Total Length of Stay metrics from our simulation results and from historical database data. The K-S statistic for two samples measures the difference between the empirical cumulative distribution functions (ECDFs) of the two samples. The ECDFs are step functions that approximate the underlying distributions from which the samples are drawn. Assuming we believe the samples to have been drawn from a continuous distribution (as is the case with the Time to First Bed and Total LOS metrics), we are able to use the K-S statistic to measure the difference between these two ECDFs (the maximum vertical distance between the two ECDFs).

The K-S statistic for the Time to First Bed metric is 0.090, and the statistic for the Total LOS metric is 0.024. As each of these is reasonably low, this means our simulation has done a reasonably good job of creating capturing the real-world distributions of patient wait times. However, due to the large sizes of our samples (just under 8000 samples from our database data and 23205 samples from our simulation), we found that we cannot prove the similarity of our results using the K-S test. Even using a lenient alpha value such as 0.01 (which yields a critical value of roughly 0.017 for the test) we are forced to reject the null hypothesis that the two samples were drawn from the same distribution.

Despite not having a provably similar distribution, we were encouraged by the low K-S statistics and by how close our average scores for each of the metrics considered matched the historical data. Noting that simulations by their very nature simplify a complex system and therefore cannot perfectly match that system's performance, we felt comfortable validating the model generated and moving to the experimental phase of the project.

Experimental Design

Now that we are convinced that our model is a valid representation of the UMMC ED, there are a number of experiments we could run to test various potential changes to the resident care system. Such experiments would employ one key structural assumption.

We assume our ability to determine on a per-patient basis whether a resident is treating a patient or if that patient is being served entirely by an attending physician. In terms of execution, this would involve some patients having the physician provide all their care (which is what happens currently on Wednesday mornings) and other patients would be shared between the resident and the physician. As described in the Statistics Section, we use predictors based on patient attributes, whether a resident is treating the patient, and how congested the ED is to predict how long a patient will stay in an ED bed. Therefore, during the busiest parts of the week, when a patient is served only by an attending physician we will predict how long this treatment will take based on data from the busiest parts of Wednesday mornings, when the residents are in their seminar. During the least busy parts of the week, when a patient is served only by an attending we will predict how long this treatment will take based on data from the least busy parts of Wednesday mornings.

We believe this provides a good representation of the amount of time an attending physician will take to treat a patient. Though our simulation model allows attending physicians to have a fraction of their patients seen by a resident without all of them being seen by a resident – something that does not happen currently in the UMMC ED – we have controlled for the major factors affecting how long it will take an attending physician to treat a patient by controlling for congestion, the patient's attributes, and whether or not the physician is splitting work with a resident for that particular patient.

In our experiments, we varied the percentage of patients seen by residents (randomly selecting whether to treat each patient with a predetermined probability of treatment) and used the model to see the effects of this on patient wait times in the ED. We used our simulation to generate control data to compare against the data generated by these experiments. In our control model, residents would visit every single patient that enters the ED, as they do currently.

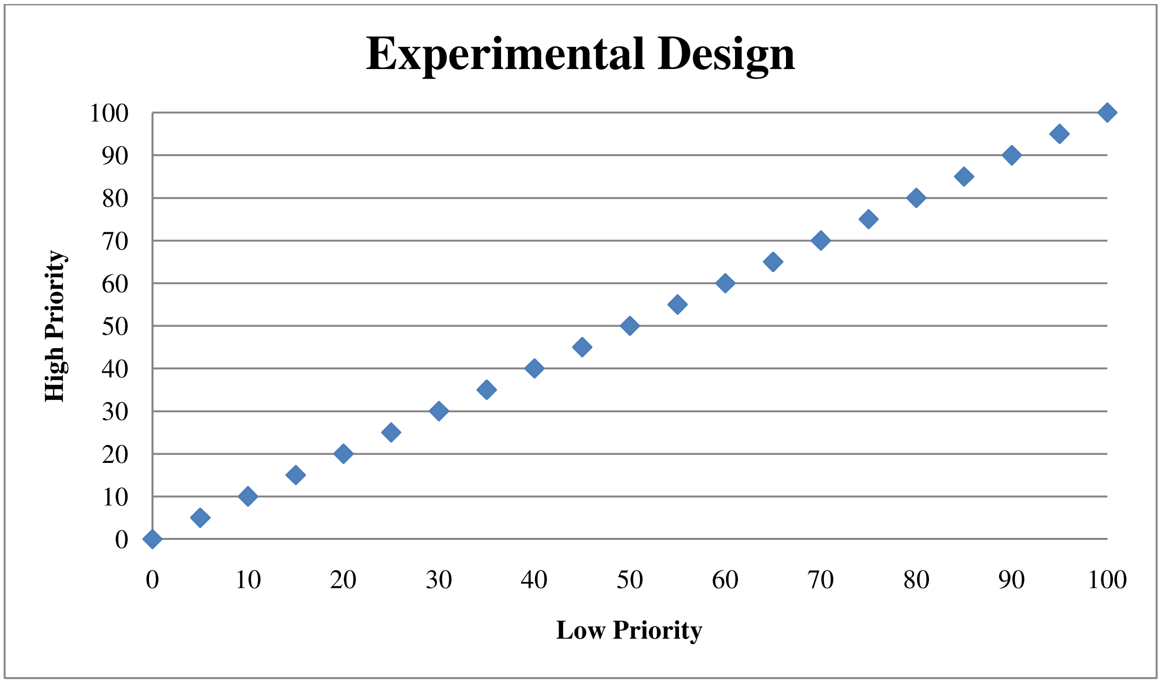


Figure 16: Experimental Design - Experiments Run

The next part of our experimental design is altering the percentage of both high- and low-priority patients residents treat. Figure 16 shows the different priority mixes that we tested in our experiment.

We ran experiments in which residents visited anywhere from none to all of the patients, stepping in increments of 5% through these percentages. Because residents need to gain experience working with both urgent and non-urgent patients, it would not be reasonable to have a resident visit vastly more high-priority patients than low-priority ones. We also did not test any scenarios in which residents visit more low- than high-priority patients because interns specializing in emergency medicine need more experience with those patients who are more severely injured.

Each experiment was run for 30 million seconds, or just under one year, the same amount of simulated time that was used in the control simulation. For each experiment, and the control group, we recorded two metrics to be compared: the average arrival-to-bed time of patients in the emergency department and the average arrival-to-discharge time. By comparing these metrics between the control group and each of the experimental groups, we determined whether there was a statistically significant difference in the length of a patient's stay in the ED based on residents treating patients.

4. Results

Experimental Results

Using solely historical data in our simulation model, we were able to generate initial results regarding the effects of the residency model on a number of metrics used to measure ED efficiency. These initial results showed that residents do have a significant impact on ED operations, especially when their activities are differentiated between low priority and high priority patients. Contrary to our original hypothesis that residents have a generally negative impact on ED efficiency, our results found that efficiency of care for all patients actually increases when the percentage of care performed by residents increases.

In our simulation, one metric we used to measure ED efficiency was the total stay time of a patient. After measuring total stay time against the percentage of care performed by a resident, a strongly linear relationship between the two emerged. When the percentage of care performed by a resident was set at 100 percent, the total stay time of low priority patients was 19,636 seconds, or 5.45 hours. When the percentage of care performed by a resident was decreased to 0 percent, the total stay time of low priority patients increased to 30,488 seconds, or 8.47 hours. Formalizing this relationship, we found a linear regression for this data to be y = -56.08x + 27614.23, where y is the total length of stay in seconds and x is the percentage of patients visited by residents, with an associated r value of -0.74. Given the large magnitude of the r-value, the relationship between resident participation and total length of stay for low-priority patients is a clear one – residents improve the total length of stay. Figure 17shows this relationship.

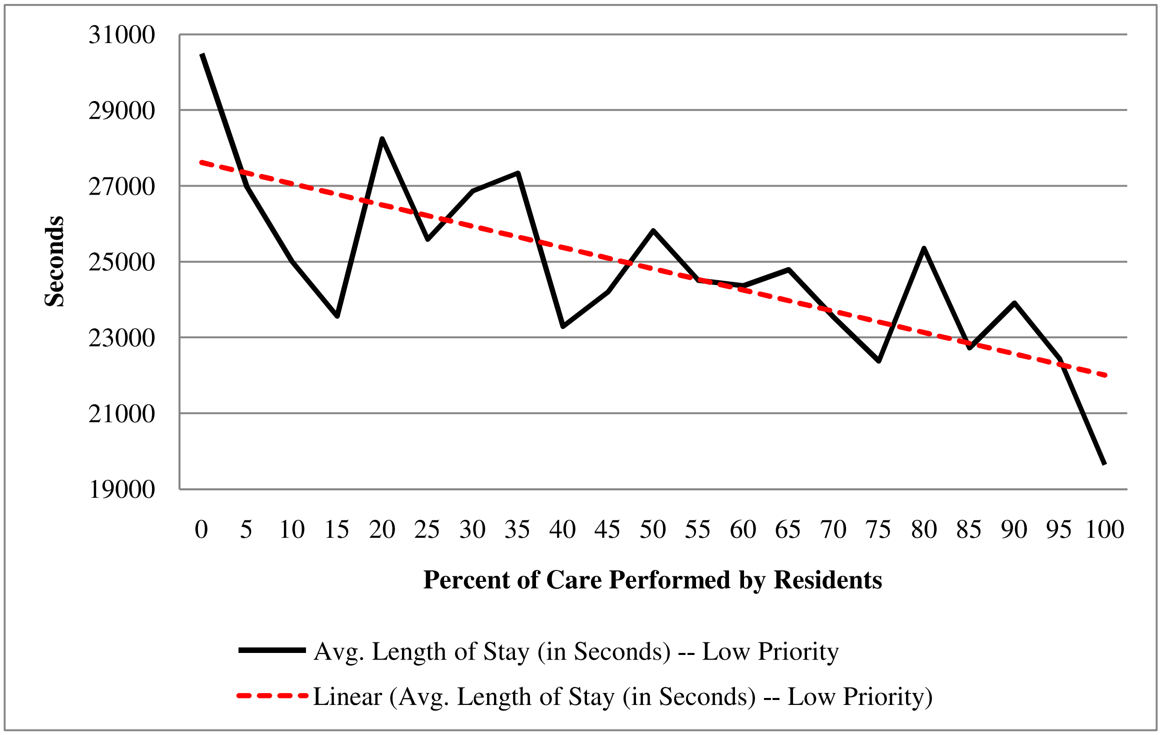


Figure 17: Total Stay Time by Percentage of Resident Care – Low Priority

Another metric we identified to be a reliable metric for measuring ED efficiency was a patient’s time to bed, or how long it takes a patient to receive a bed from when they register in the waiting room. When measured against percentage of care performed by a resident specifically for low priority patients, our findings reaffirmed the results generated from measuring the total stay time of a patient. When percentage of care performed by a resident was set to 100 percent in our simulation model, the average time to bed for low priority patients was found to be approximately 6,076 seconds, or 1.69 hours. When the percentage of care performed by a resident was reduced to 0 percent, the average low priority patient’s time to bed increased to 14,930 seconds, or 4.15 hours. Similar to the relationship between the total stay time of low priority patients to the percentage of care performed by a resident, the relationship between time to bed of low priority patients and the percentage of care performed by a resident was quite linear and showed that residents significantly increase efficiency in the ED when specifically treating low priority patients. Using linear regression, we matched the model y=-51.95x + 12580.56, where y is the time to bed in seconds and x is the percentage of patients cared for by the residents. This regression had an r-value of -0.82, again showing it to be a good model for the simulation data. Figure 18 shows this relationship below.

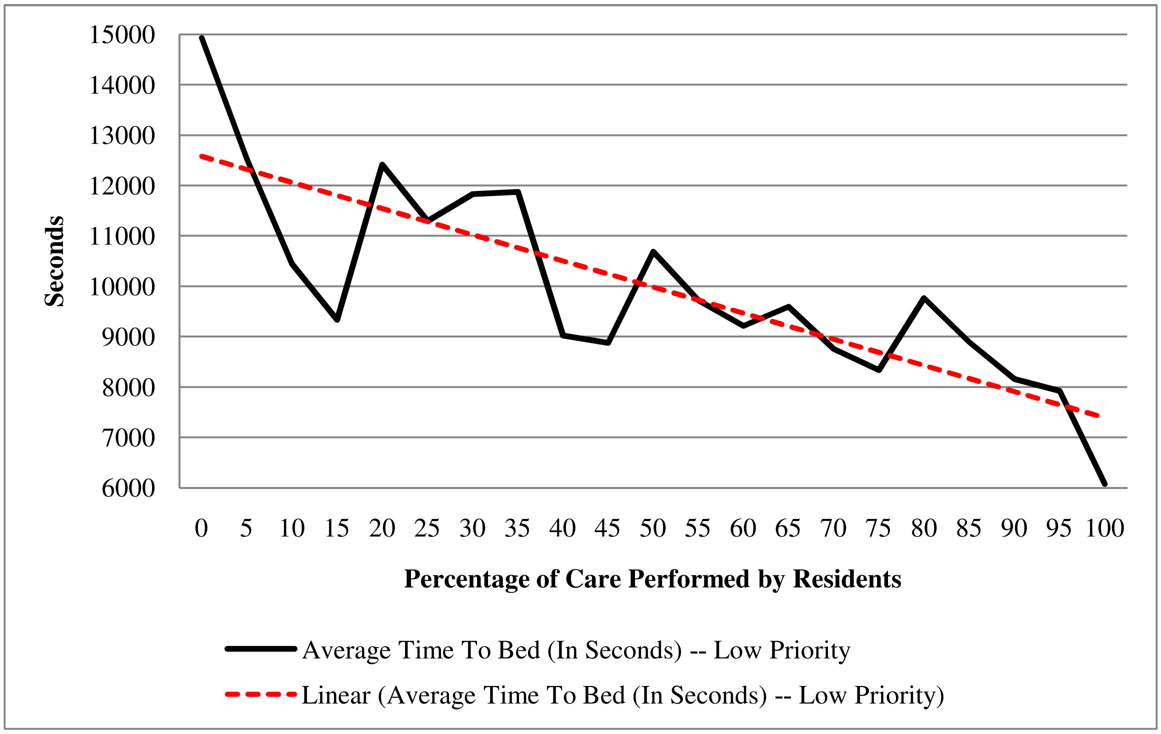


Figure 18: Time to Bed by Percentage of Resident Care – Low Priority

Though the changes in wait times differed in magnitude for higher priority patients, they followed the same trend. When the percentage of care performed by a resident was set to 0 percent in our simulation, the average time to bed for high priority patients equaled 6,465 seconds, or 1.80 hours. When the percentage of care performed by a resident was increased to 100 percent, the average time to bed for high priority patients decreased to 4,299 seconds, or 1.19 hours. Although this change is much smaller than that seen in low priority patients, it is still significant. Via regression testing, we matched the linear model y=-17.59x + 6311.34 to the data, where y is the time to bed in seconds and x is the percentage of care provided by residents. With an r-value of -0.94, this regression fits the data nearly perfectly. Therefore, increasing the percentage of care performed by residents also increases efficiency of care for high priority patients, though to a lesser degree than it does lower priority patients. Figure 19 shows this relationship below.

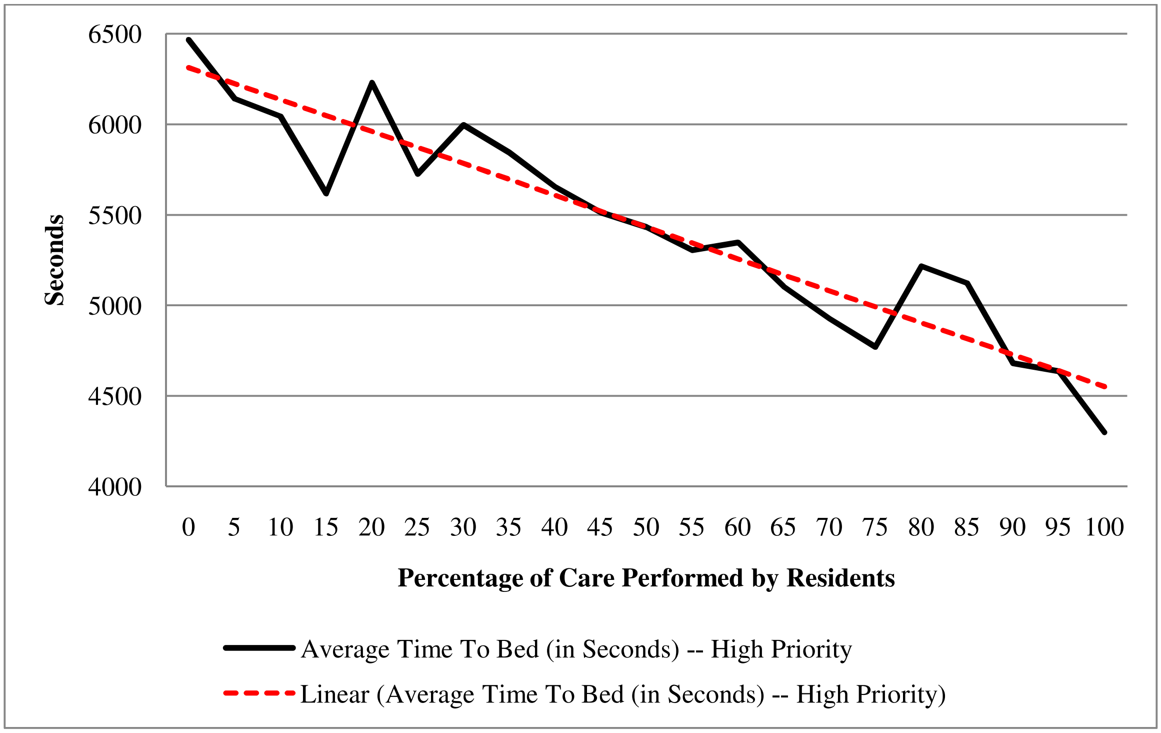


Figure 19: Time to Bed by Percentage of Resident Care – High Priority

Last, the total length of stay in the ED for high priority patients also was improved by the presence of residents. When no residents were providing patient care, the average length of stay for high priority patients was 37,262 seconds, or 10.35 hours. However, when residents cared for every patient, the average length of stay was 32,184 seconds, or 8.94 hours. By fitting our experimental data with a linear regression, we found the linear model y=-45.03x + 37339.08, where y is the total length of stay in the ED in seconds and x is the percentage of patients cared for by residents. This regression has an r-value of -0.97, meaning it nearly perfectly fits the experimental data. Figure Figure 20 shows this relationship below.

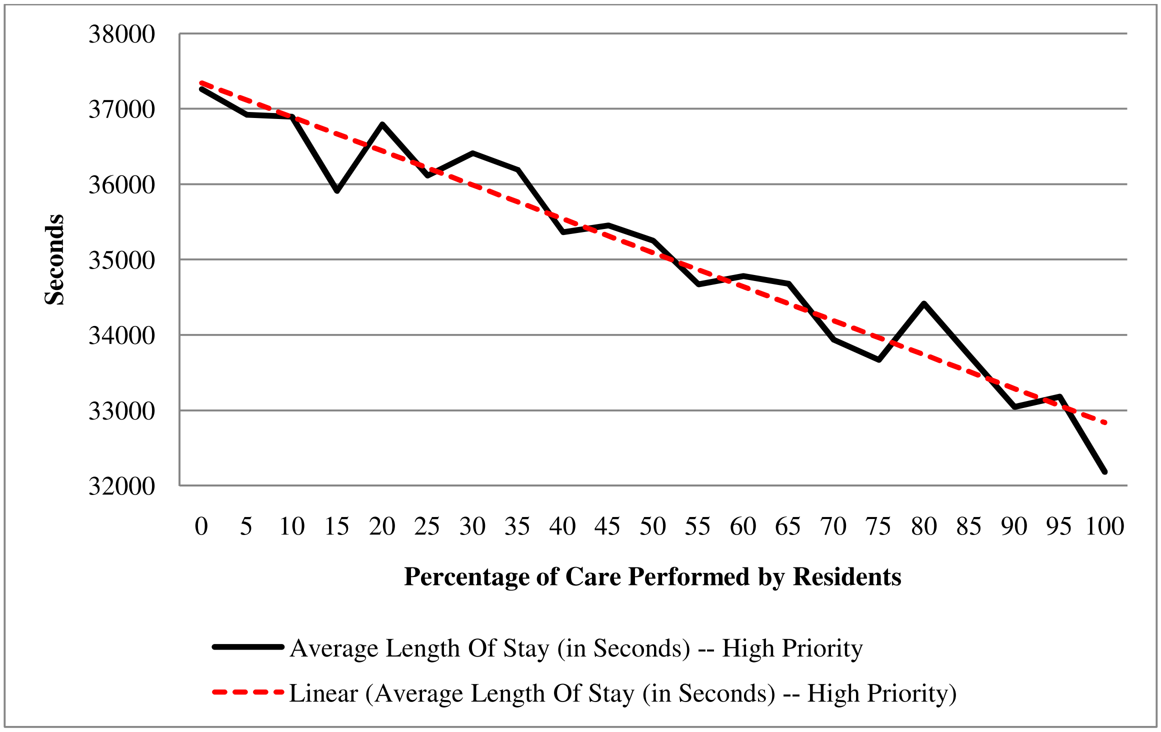


Figure 20: Total Stay Time by Percentage of Resident Care - High Priority

5. Discussion

Our study's results have demonstrated a statistically significant relationship between the resident education model and emergency department efficiency. Essentially, as resident activity in the ED increases, the ED's subsequent patient throughput and overall efficiency improves for patients of all acuity levels, though most sharply for patients of low acuity level.

This is a substantial contribution to the existing literature and studies conducted on the effects of the resident model in hospitals. Our study is among the first to introduce a significant volume of quantitative, non-observational data, which was used to accurately incorporate the teaching system into the simulation model. The model allowed us to evaluate the implications of our results, testing possible changes to determine whether they produced positive or detrimental changes in patient throughput.

Implications of Residents Improving ED Efficiency

The results demonstrates that residents have a positive effect on ED efficiency – the emergency department processes more patients than if they residents were removed in partial or complete capacity and not replaced.

Though residents improved ED efficiency for both low- and high-priority patients, they had the largest effect on low-priority patients. According to the regression results, for every extra percentage of patients visited by residents, the average low-priority patient gets out of the ED 56 seconds faster and the average high-priority patient gets out of the ED 45 seconds faster. Further, for every extra percentage of patients visited by residents, the average low-priority patients gets into an ED bed 52 seconds faster, while the average high-priority patients gets into an ED bed only 18 seconds faster. These results verified the observational findings of the study conducted at Waikato Hospital while the residents were on strike, which found that when residents were present in the ED, 25 percent of the lowest-level triage patients were seen within the suggested time frame, compared to zero percent when only the attending physicians were present (Harvey et al, 2008).

This seems to suggest a technique in which residents' work is selected to favor treating low-priority patients, since they have the largest positive effect on these patients. Despite this finding, residents cannot be limited to or be encouraged to focus on low-level patients, since this refutes the ideal of diverse patient exposure – residents need to focus on a diversity of interesting cases to be the best prepared for a medical career

Similarly, residents are not at liberty to focus exclusively on higher priority patients due to the potential politics involved – attending physicians are not present in the ED to be delegated to low-level patients. Therefore, a delicate balance must be struck, where both efficiency and politics are taken into consideration when modifying residents' patient loads. At the UMMC, a more tailored, reasonable appropriation of patients amongst residents could strive to address the "general erosion of the [residents'] learning environment" the GME task force observed during their evaluation of the general resident teaching model. From our simulation, it can be seen that residents still see a large percent of the patients in the ED. Maintaining the diversity each resident requires for proper training, while simultaneously limiting the overall number of patients each resident treats, may be sufficient to provide a good education for future physicians while improving overall ED throughput.

Future Research

Our results indicated that the residents have a positive effect on patients of all acuity levels. Future research could continue to look at attending physicians and residents, focusing on how to optimize the time spent on certain tasks that the doctors complete. For example, the GME task force found that a significant portion of residents' time is spent on non-educational work such as entering orders and filling out discharge papers. While an argument could be made that such tasks are integral to the learning process, the task force argues that too many of such tasks leads to a "decline in the overall educational content of their work" (AAMC, 2003, p.2). An earlier study established that the fragmentation caused by such one- to five-minute tasks increases inefficiency (Dassinger, Eubanks, & Langham, 2008). Using the simulation to better allocate tasks to residents could lead to increased efficiency in the ED. Due to the doctor data we collected, we are in prime position to investigate this sort of a question in future research.

Expanding the research on resident and attending physician actions, our simulation could be used to look at the choices that each doctor makes regarding admitting patients into the ED. From our initial round of data analysis, it appears that patients with a low severity level are often admitted into the ED before patients with higher severity levels due to low-priority patients being quickly shuttled into the ambulatory zone, which is the opposite of what common sense dictates should occur. We could further analyze this anomaly by checking if the actions of residents relating to such low-severity, admitted patients differ in any way from actions of attending physicians with said patients. If any differences are in fact identified, then it is possible that more experienced doctors, such as attending physicians, have a tendency to handle certain types of patients differently than less experienced residents. The reasons behind such differences in action can then be included in the resident teaching instruction.

Also, the interactions between the various doctors could be studied by using the simulation model. Most patients in the ED are examined by an intern, senior resident and attending physician, in no set order. Because there is no established order, if, for example, a senior resident examines a given patient significantly later than the intern and attending physician assigned to that patient, the attending physician may have to meet at different times with both the intern and senior resident to discuss the facts of the case. If the examinations had been timed differently, the two meetings could have been consolidated. As an alternative scenario, if the intern and senior resident examine a patient significantly earlier than the attending physician, the residents must wait on the attending before they can take significant action toward the patient's case. We could use the simulation to account for similar situations, seeking again to minimize process fragmentation.

Next, with the simulation model we can test the effectiveness of shortening the length of residents' shifts, keeping the overall work week at approximately 80 hours. In an earlier study, Jeanmonod, Jeanmonod, and Ngiam, tested the impact that shorter shifts had on overall efficiency. They compared second-year residents on 12-hour shifts against those on 9-hour shifts, and found that residents working shorter shifts were more productive, seeing 10 more patients during the shorter shifts on average treated (Jeanmonod, Jeanmonod, & Ngiam, 2007). This study did not take in to account for patient turnaround time, which should also be included.

Our study found that the more times a patient is passed over in the selection of the next patient to get a bed, the less likely that patient is to be selected for future bed openings. This finding is also extremely interesting in that one would expect that patients who have been waiting for a significant period of time would be admitted before other patients who had just arrived at the hospital. Although this can partially be explained by the fact that the patients with the longest wait times are often those with the least severe issues, this in combination with our above finding that patients with lower severity levels are often admitted before those with higher severity levels creates a quandary. In our second round of data analysis, we will attempt to identify the cause of this puzzling occurrence, as well as if it has any significant effects on the resident education model and/or ED efficiency.

Another interesting area of research is the residents' and attending physicians' action preferences, the order in which they choose to execute a number of activities. These activities include but are not limited to initial patient examinations; subsequent patient visits; lab, radiology, and medicine order placements; and consultations with other doctors, nurses, clerks, etc. On a more detailed level, these distributions could compare not only general preferences for residents and attending physicians, but more specific preferences broken down by individual residents and attending physicians in the UMMC

ED. The individuals’ distributions would allow us to consider how certain doctors optimized their time in the ED and determine whether their priorities can be generalized into hospital policy. Changes in action priorities can be used to determine the observed yields on metrics such as bed availability, patient lengths of stay, etc.

6. Conclusions

Healthcare costs in the United States have been increasing at an alarming rate for the past 20 years. By 2018, healthcare expenditures are expected to compose one-fifth of the country's GDP. Hospitals, and especially emergency departments, are widely recognized as some of the most inefficient cost centers in the healthcare system. Although all EDs are limited by bottlenecks in lab testing and in the number of beds available, research hospitals, such as UMMC, also sacrifice efficiency to accommodate the resident teaching model. At least two doctors, one resident and one attending, must visit each patient before initiating treatment, a process that adds extra time to a patient's length of stay.

Many studies have been conducted on the efficiency of the resident care model. One class of studies looked at the resident care model and found that it had a negative impact on ED efficiency; however, none of these studies used a simulation model to experiment with potential solutions. Another class of studies used simulations to model emergency departments, but these studies were not conducted at research hospitals, did not look at the residency care model, and relied completely on database data. Our study is unique in that it uses simulation techniques to model the residency care model and uses both database and manually collected data.

We studied the resident care model at the University of Maryland Medical Center emergency department. We developed a simulation of the department to model the flow of patients through their treatment process. Simulation modeling allowed us to experiment with changes to the resident care model without actual changes in the operating procedures of the hospital, an advantage over observational studies in an ED. Our goal was to develop alternative resident care models that improved patient flow, expediting the treatment process.

To accurately model the UMMC ED, we observed the actual ED to determine how its resources are allocated and how patients move through the ED. Observing the physical layout of the ED and having discussions with ED personnel allowed us to uncover the tangible and intangible factors that play a role in ED personnel decision making and patient throughput. We were then able to incorporate these attributes into our simulation model.

Using the insights we gathered from our observations of the department, we created a simulation model. We split up the patient's stay into three categories – the patient's arrival, selection to move into a ward bed, and time spent in the ED bed. To more accurately predict how long it will take a patient to move through the ED, we also generated a number of attributes for the patients, including their severity, whether they will require lab tests, and whether they will be released to an inpatient ward. Using historical database data from the UMMC, we were able to accurately seed the simulation model with parameters.

First, we used historical arrival rates from the database data to predict when patients will arrive at the ED. Next, we used the historical database data to generate attributes for the patient that did arrive. Based on the patient's selected attributes, we modeled that patient's chances of getting selected to move into a bed when that bed becomes available and the distribution for the patient's length of stay once in a bed.

In calculations to determine the preferences of triage nurses between different types of patients waiting for a bed, we found a number of interesting insights about the selection process for these triage nurses. As expected, patients with higher severity score were favored over patients with low severity score in receiving ED beds. However, when considering patients receiving either ED beds or beds in the ambulatory zone (AZ), an area for patients with less severe conditions, we found that patients with lower severity were often selected ahead of patients with more severe conditions because of the quick turnaround for the AZ beds. Further, we determined that patients who have been skipped over for a bed in the past are more likely to get skipped over in the present, regardless of that patient's severity score. This runs opposite to the logic that waiting longer will improve your preference for being selected for a bed.

To validate the simulation model, we used a graphical user interface (GUI) to visualize patient movements through the ED to identify and errors in the model formulation. Further, we used the Kolmogorov-Smirnov statistic to validate the similarity of our simulation model's output to historical database data for a number of important ED metrics.

Once the model was validated as operating in a similar manner to the real-world operations of the emergency department, we experimented with the simulation. Taking advantage of the fact that residents have a seminar Wednesday mornings, we were able to quantify the effects of residents treating patients, accounting for the effects of ED congestion on the work residents perform by using the number of people in the waiting room to help predict the overall wait time for a patient once they enter an ED bed. We experimented with having different percentages of patients receive care from residents.

In experimentation, we noted significant increases in patient waiting times for all patients when fewer patients were seen by the residents. These effects were seen both in the average time from entry into the waiting room to the time the patient reached a bed in the ED (*time to bed*) and the average time from entry into the waiting room to the time the patient was discharged from the ED (*length of stay*). We noted that residents provide the strongest improvements in wait times for low-priority patients.

A number of avenues for future work remain. Balancing the education of the residents and increasing efficiency makes it impossible to have residents only treat low severity patients or to have residents treat too small a proportion of the total patient population. Future research into what changes are actually feasible in an ED would assist in suggesting actual modifications that would improve efficiency in the UMMC and other research hospitals.

Members of the team and student volunteers collected data at the UMMC, gathering timestamps of when different doctors activities occurred in the ED. This data will enable significant expansion of the model in future work. In that work, characterizations of doctor decision making will provide fine-grained detail about how doctors interact with other doctors and with patients. Further future research will focus on optimization of tasks for both attending physicians and residents and on the comparison of preferences between residents and attending physicians. This analysis will help identify teaching opportunities for the attending physicians.

References

Abujudeh, H., Vuong, B., & Baker, S. R. (2005). Quality and operations of portable X-ray examination procedures in the emergency room: queuing theory at work. *Emergency Radiology, 11,* 262-266.

Advisory Board Company (2008). The high-performance ED: Optimizing capacity and throughput to meet ever-growing demand. *Clinical Advisory Board: 2008 Annual Meeting.*

Ahmed, M.A., & Alkhamis, T.M. (2009). Simulation optimization for an emergency department healthcare unit in Kuwait. *European Journal of Operational Research*, 198, 136-942.

Albeseder, D., Fuegger, M. (2005). Small PC-network simulation - a comprehensive performance case study. *Research Report 77/2005, TU Wien, Institut f̀ur Technische Informatik.*

American Hospital Association and Maryland Hospital Association. (2007). New ways of working: Improving workflow on patient care units pilot program. Maryland.

Association of American Medical Colleges. (2003). Integrating Education and Patient Care: Observations from the GME Task Force. Retrieved from https://services.aamc.org/publications/showfile.cfm?file=version10.pdf&prd_id=87&prv_id=85&pdf_id=10

Atzema, C., Bandiera, G., Schull, M.J., & Coon, T.P. (2005). Emergency Department Crowding: The Effect on Resident Education. *Annals of Emergency Medicine*, 45(3), 276-281.

Banerjea-Brodeur, M., Cordeau, J. F., Laporte, G., & Lasry, A. (1998). Scheduling linen deliveries in a large hospital. *Journal of the Operational Research Society, 49* (8), 777-780.

Blair, R. (2005). Capacity management: The bedrock of efficiency: Houston hospital takes the creative high road to managing patient flow and beds with wireless technology and hand-helds. *Health Management Technology.*

Boger, E. (2003). Electronic tracking board reduces ED patient length of stay at Indiana Hospital. *Journal of Emergency Nursing, 29*(1), 39-43.

Bowers, J., & Mould, G. (2005). Ambulatory care and orthopaedic capacity planning. *Healthcare Management Science,* 8(1), 41-47.

Brown, B., & Whiting, S.O. (2008). Speech Recognition 2008. *Healthcare Quarterly, 11*(4), 99-101.

Brunelle, D. & Rawlinson, C. (2006). PACS help control patient flow. *Nursing Management, 37*(11), 35-39.

Bush, S. H., Lao, M. R., Simmons, K. L., Goode, J. H., & Cunningham, S.A. (2007). Patient access and clinical efficiency improvement in a resident hospital-based Women's Medicine Center Clinic. *The American Journal of Managed Care*, 13(12), 686-690.

Calkins, K.G. (2005, May 10). *Queuing Theory and the Poisson Distribution.* Retrived February 2008, from Andrew University: http://andrews.edu/~calkins/math/webtexts/prod10.htm

Cassandras, C.G., & Lafortune, S. (2008). *Introduction to Discrete Event Systems*. New York: Springer.

Cavouras, C. A. (2002). Nurse staffing levels in American hospitals: A 2001 report. *Journal of Emergency Nursing, 28*(1), 40-43.

Centers for Medicare and Medicaid Services. (2009). National health expenditure projections 2009-2019. Retrieved from: http://www.cms.hhs.gov/NationalHealthExpendData/downloads/proj2009.pdf

Charap, M. (2004). Reducing resident work hours: unproven assumptions and unforeseen outcomes. *Annals of Internal Medicine, 140*(10), 814-815.

Clark, J.J. (2005). Unlocking hospital gridlock. *Healthcare Financial Management, 59*(11), 94-96.

Cooke, M., Irby, D.M., Sullivan, W., & Ludmerer, K.M. (2006). American Medical Education 100 Years After the Flexner Report. *The New England Journal of Medicine, 355*, 1339-1344.

Dassinger, M. S., Eubanks, E. W., & Langham, M. R. (2008). Full work analysis of resident work hours. *Journal of Surgical Research*, 147(2), 178-181.

Debhenke, D., O'Brien, S., & Leschke, R. (2007). Emergency Medicine resident work productivity in an academic Emergency Department. *Academic Emergency Medicine*, 7(1), 90-96.

Derlet, R. W. & Richards, J.R. (2002). Emergency Department overcrowding in Florida, New York, and Texas. *Southern Medical Journal,* 95(8), 846-849.

Featherstone, P., Chalmers, T., & Smith, G. B. (2008). RSVP: A system for communication of deterioration in hospital patients. *British Journal of Nursing, 17*(13), 860-864.

Fletcher, J.P. (1999). Making the surgical beds go around. *Journal of Quality in Clinical Practice, 4*, 208-210.

Gibson, D. (2000). *Finite State Machines – FSM Tutorial.* Retrieved April 2008, from SPLat Controls Pty Ltd: http://www.splatco.com/fsm_tute/fsm_tute01.htm

Green, L. V., Soares, J., Giglio, J. F., & Green, R. A. (2006). Using queuing theory to increase the effectiveness of emergency department staffing. *Academic Emergency Medicine, 1-8.*

Gorsha, N. & Stogoski, J. (2006). Transforming emergency care through an innovative tracking technology: an emergency department's extreme makeover. *Journal of Emergency Nursing, 32*(3), 254-257.

Hanada, E., Fujiki, T., Nakakuni, H., & Sullivan, C. V. (2006). The effectiveness of the installation of a mobile voice communication system in a university hospital. *Journal of Medical Systems, 30*(2), 101-106.

Harvey, M., Shaar, M.A., Cave, G., Wallace, M., & Brydon, P. (2008). Correlation of physician seniority with increased emergency department efficiency during a resident doctors’ strike. *Journal of the New Zealand Medical Association*, 121(1272), 59-68.

Hendrich, A. L. & Lee, N. (2005). Intra-unit patient transports: Time, motion, and cost impact on hospital efficiency. *Nursing Economics,* 23(4), 157-164.

Howard, R. (1960). *Dynamic Programming and Markov Processes.* Boston: MIT Press and Wiley.

Howell, E., E. Bessman, S. Kravet, K. Kolodner, R. Marshall, & S. Wright. (2008). Active bed management by hospitalists and emergency department throughput. *Annals of Internal Medicine, 149*(11), 804-812.

Humphries K.: *VHA: On-Line Survey Results.* Paper presented at the VHA conference, Optimizing Functional Capacity Conference 2006, Tucson, Arizona, Nov. 6, 2006.

Jeanmonod, R., Brook, C., Winther, M., Pathak, S., & Boyd, M. (2009). Resident productivity as a function of emergency department volume, shift time of day, and cumulative time in the emergency department. *The American Journal of Emergency Medicine,* 27(3), 313-319.

Jeanmonod, R., Jeanmonod, D, & Ngiam, R. (2007). Resident productivity: does shift length matter?. *The American Journal of Emergency Medicine*, 26(7), 789-791.

Jebali, A., Hadj Alouane, A. B, & Ladet, P. (2005). Operating rooms scheduling. *International Journal of Production Economics, 99*(1-2), 52-62.

Kalunga, J. (2006, March 04). *What is a probability distribution function*? Retrieved from: http://cnx.org/content/m13466/latest/

Khare, R. K., Powell, E. S., Reinhardt, G., & Lucenti, M. (2009). Adding more beds to the emergency department or reducing admitted patient boarding time: Which has a more significant influence of emergency department congestion? *Annals of Emergency Medicine, 53*(5), 575-585.

Komashie, A., & Mousavi, A. (2005). Modeling emergency departments using discrete event simulation techniques. *Winter Simulation Conference*, 2681-2685.

Kolb, E.M., Peck, J., Schoening, S., & Lee, T. (2008). Reducing emergency department overcrowding- 5 patient buffer concepts in comparison. *Winter Simulation Conference*, 1516-1525.

Korn, R. & Mansfield, M. (2008). ED overcrowding: An assessment tool to monitor ED registered nurse workload that accounts for admitted patients residing in the Emergency Department. *Journal of Emergency Nursing, 34,* 441-446.

Le, M.M., Zwemer, F. L. & Dickerson, V. J., & Paris, S. (2004). Providing mobile phones to emergency medicine residents: perceived effects on physician communication and work. *Annals of Emergency Medicine, 44*(4), S28-S28.

Littig, S.J. & Isken, M.W. (2007). Short term hospital occupancy prediction. *Health Care Management Science,* 10, 47-66.

Mackay, M., & Lee, M. (2005). Choice models for the analysis and forecasting of hospital beds. *Healthcare Management Science*, 8, 221-230.

Matsumoto, M.; Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation,* 8 (1): 3–30.

McGowan, J.E., Truwit, J. D., Cipriano P., Howell, R E., VanBree, M., Garson, A. & Hanks, J. B. (2007). Operating room efficiency and hospital capacity: factors affecting operating room use during maximum hospital census. *Journal of the American College of Surgeons, 204*(5), 865-871.

McLaughlin, S.A., Doezema, D., & Sklar, D.P. (2002). Human simulation in emergency medicine training: A model curriculum. *Academic Emergency Medicine,* 9(11), 1310-1318.

Miller, M.J., Ferrin, D.M., & Messer, M.G. (2004). Fixing the emergency department: A transformational journey with EDSIM. *Winter Simulation Conference*, 1-6.

Miller, M.J., Ferrin, D.M., & Szymanski, J.M. (2003). Emergency departments II: Simulating Six Sigma improvement ideas for a hospital emergency department. *Winter Simulation Conference*, 1926-1929.

Naylor, T.H., Finger, J.M., McKenney, J.L., Schrank, W.E., & Holt, C.C. (1967). Verification of computer model simulations. *Management Science*, *14*(2), B92-B106.

Parzen, E. (1999) *Stochastic Processes.* Oakland: Holden-Day.

Press Ganey Consultants. (2009). 2009 Emergency Department pulse report. Retrieved from: http://www.pressganey.com/galleries/ED_Pulse_2009_files/2009_ED_Pulse_Report.pdf

Rohrer, J.E. (1988). Efficiency and the supply of hospital beds in metropolitan areas. *Journal of Public Health Policy, 9*(3), 393-402.

Rossetti, M.D., Trzcinski, G.F., & Syverud, S.A. (1999). Emergency Department simulation and determination of optimal attending physician staffing schedules. *Winter Simulation Conference*, 1532-1540.

Rubinstein, R. Y. (1981). Simulation and the Monte Carlo Method. United States: John Wiley & Sons, Inc.

Ruohonen, T. (2007). Improving the operation an emergency department. *University of Jvaskyla*, 1-167.

Shayne, P., Lin, M., Ufberg, J.W., Ankel, F., & Barringer, K. (2008). The effect of emergency department crowding on education: Blessing or curse?. *Academic Emergency Medicine*, 16, 76-82.

Stainsby, H., Taboada, M., & Luque, E. (2009). Towards an agent-based simulation of hospital Emergency Departments. *Proceedings of the 2009 IEEE International Conference on Services Computing. Washington, D. C.*

Takakuwa, K.M., F.S. Shofer, & S.B. Abbuhl. (2007). Strategies for dealing with emergency department overcrowding: a one-year study on how bedside registration affects patient throughput times. *Journal of Emergency Medicine, 32*(4), 337-342.

Tat, F.H., K.C. Wah, Y.H. Hung. (2002). A follow-up study of electromagnetic interference of cellular phones on electronic medical equipment in the emergency department. *Emergency Medicine, 14*(3), 315-320.

Trama, M., Northam-Schuhmacher, C., & Intelisano, A. (2006, June 20). S.T.E.P. project. Lecture at Stony Brook University Medical Center, Stony Brook, NY.

Trzeciak, S., & Rivers, E.P. (2003). Emergency department overcrowding in the United States: an emerging threat to patient safety and public health. Emergency Medicine Journal, 20, 402-405.

University HealthSystem Consortium. (2006). UHC managing patient flow 2006 scorecard: University of Maryland Medical Center. Washington, DC: University HealthSystem Consortium.

University of Maryland Medical Center. (2007). University of Maryland Emergency Medicine: Univ. of MD Medical Center. Retrieved from http://umem.org/hos_umms.php

van der Togt, R., van Lieshout, E.G., Hensbroek, R., Beinat, E., Binnekade, J. M & Bakker, p. J. M. (2008). Electromagnetic interference from radio frequency identification inducing potentially hazardous incidents in critical care medical equipment. *JAMA, 299*(24), 2884-2890.

van Lieshout, E.J., van der Veer, S. N., Hensbroek R., Korevaar, J. C., Vroom, M. B., & Schultz, M. J. (2007). Interference by new-generation mobile phones on critical care medical equipment. *Critical Care, 21*, 267-270.

Vetto, JT, & Robbins, D. (2005). Impact of the recent reduction in working hours (the 80 hour work week) on surgical resident cancer education. *Journal of Cancer Education,* 20(1), 23-27.

Wagner, F., Schmuki, R., Wagner, T., & Wolstenholme, P. (2006) *Modeling Software with Finite State Machines: A Practical Approach*. Boca Raton: CRC Press.

Wang, L. (2009). An Agent-based Simulation for Workflow in Emergency Department. *Proceedings of the 2009 IEEE Systems and Information Engineering Design Symposium, University of Virginia,* pg. 19-23.

Weintraub, B., Hashemi, T., & Kucewicz, R. (2006). Creating an enhanced triage area improves emergency department throughput. *Journal of Emergency Nursing, 32*(6), 502-505.

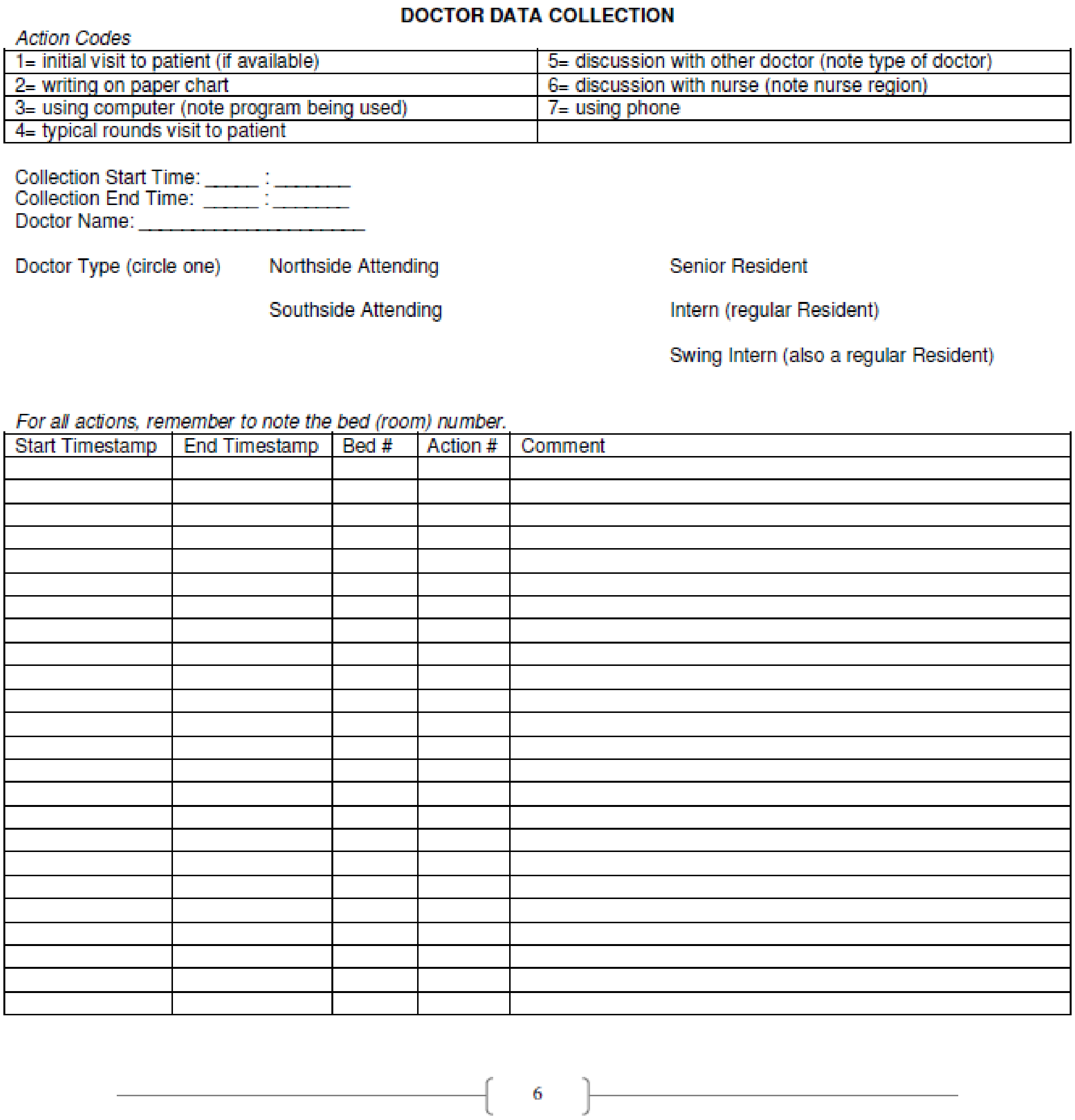
Welch, S., S. Jones, & T. Allen. (2007). Mapping the 24-hour emergency department cycle to improve patient flow. *The Joint Commission Journal on Quality and Patient Safety, 33*(5), 247-255.

Wiler, J.,Gentle, C., Halfpenny, J. M., Heins, A., Mehrotra, A., Mikhail, M. G. & Fite, D. (2009). Optimizing emergency department front-end operations. *Annals of Emergency Medicine, 20*(10), 1-20.

Yeh, J., & Lin, W. (2007). Using simulation technique and genetic algorithm to improve the quality care of a hospital emergency department. *Expert Systems with Applications*, 32, 1073-1083.

Appendix A

Doctor Data Collection Sheet



DOCTOR DATA COLLECTION

Action Codes

1= initial visit to patient (if available)	5= discussion with other doctor (note type of doctor)
2= writing on paper chart	6= discussion with nurse (note nurse region)
3= using computer (note program being used)	7= using phone
4= typical rounds visit to patient	

Collection Start Time: _____ : _______
Collection End Time: _____ : _______
Doctor Name: ____________________

Doctor Type (circle one) Northside Attending Senior Resident

Southside Attending Intern (regular Resident)

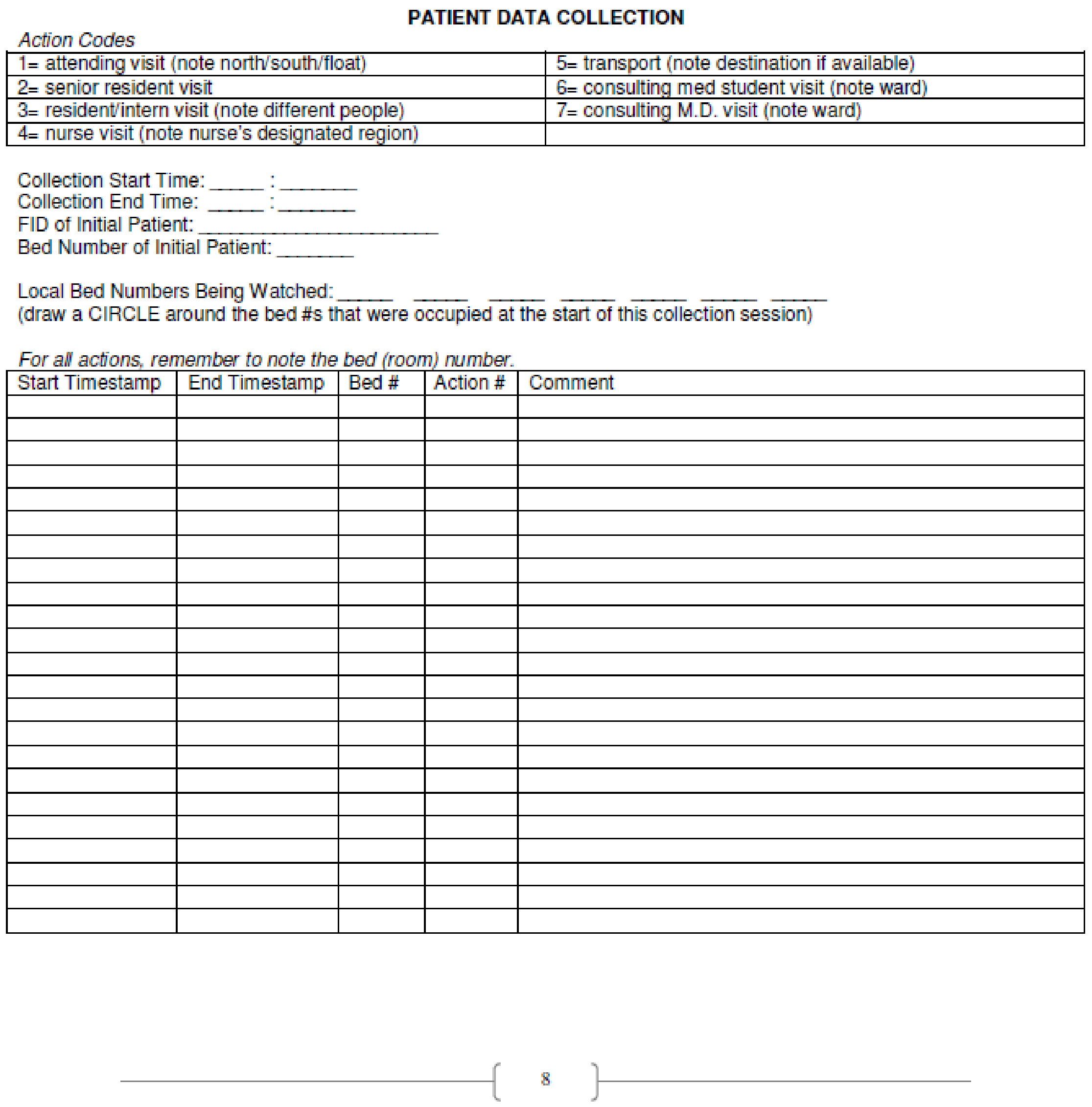
Swing Intern (also a regular Resident)

For all actions, remember to note the bed (room) number.

Start Timestamp	End Timestamp	Bed #	Action #	Comment

Appendix B

Patient Data Collection Sheet



PATIENT DATA COLLECTION

Action Codes

1= attending visit (note north/south/float)	5= transport (note destination if available)
2= senior resident visit	6= consulting med student visit (note ward)
3= resident/intern visit (note different people)	7= consulting M.D. visit (note ward)
4= nurse visit (note nurse's designated region)	

Collection Start Time: _____ : _______
Collection End Time: _____ : _______
FID of Initial Patient: ______________________
Bed Number of Initial Patient: _______

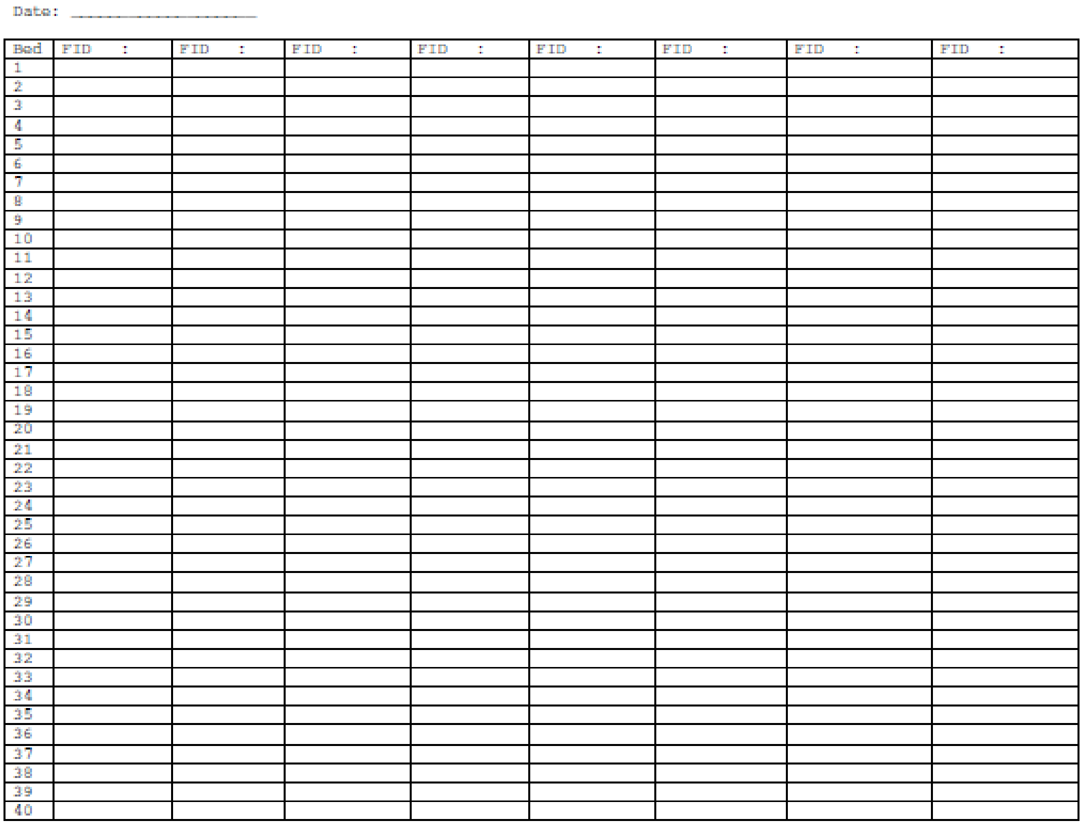
Local Bed Numbers Being Watched: _____ _____ _____ _____ _____ _____ _____
(draw a CIRCLE around the bed #s that were occupied at the start of this collection session)

For all actions, remember to note the bed (room) number.

Start Timestamp	End Timestamp	Bed #	Action #	Comment

Appendix C

FID Validation Sheet



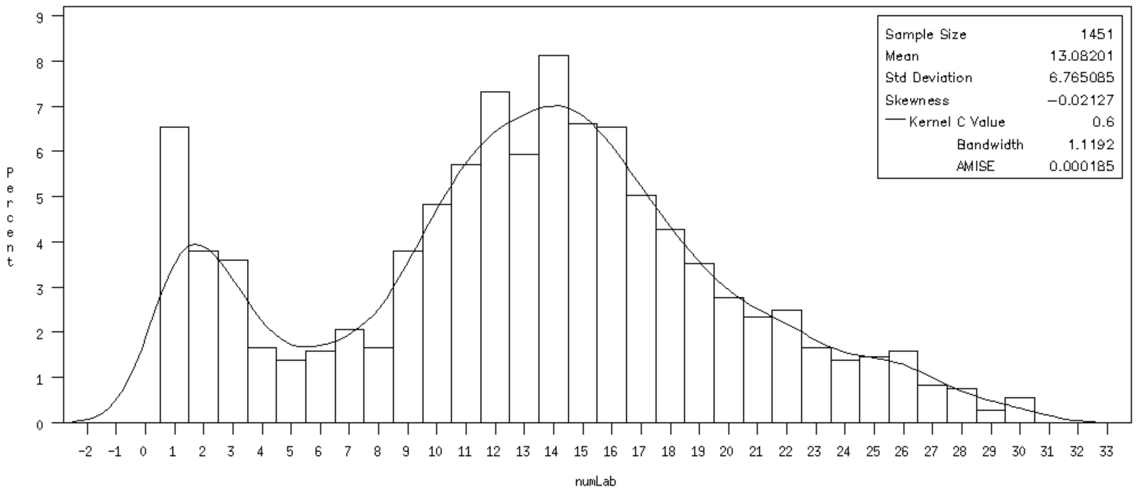
Date: ____________________

Bed	FID :	FID :	FID :	FID :	FID :	FID :	FID :	FID :
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								
32								
33								
34								
35								
36								
37								
38								
39								
40								

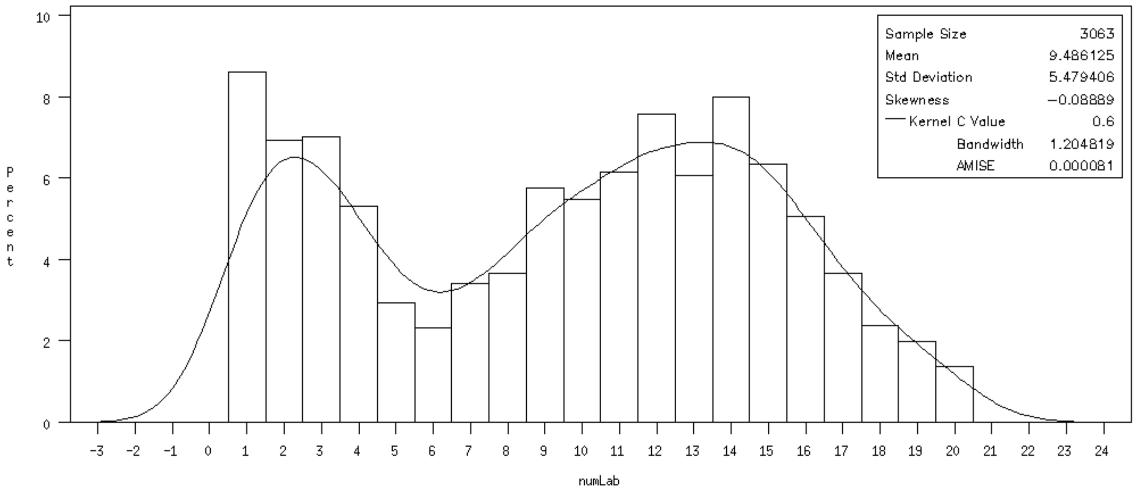
Appendix D

Statistical Graphs – # Lab Tests by Severity

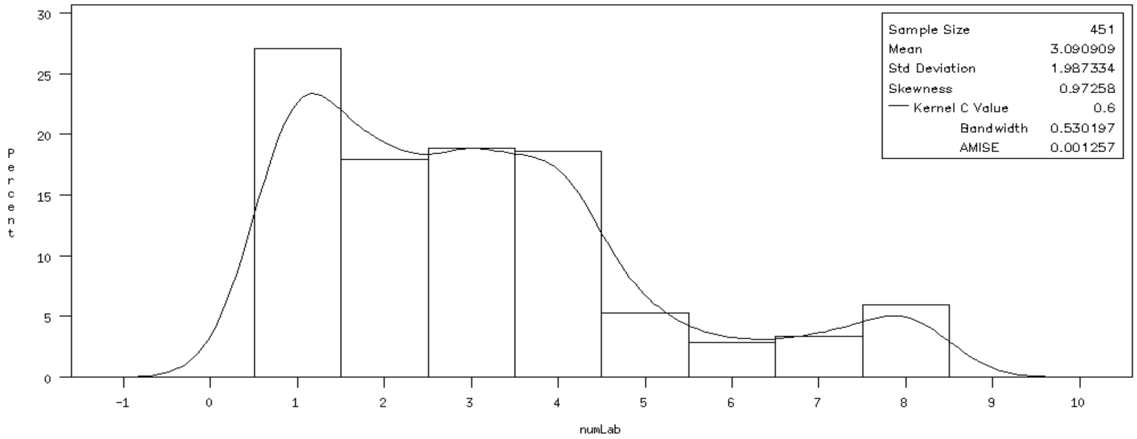
Labs Tests for Severity 1 or 2



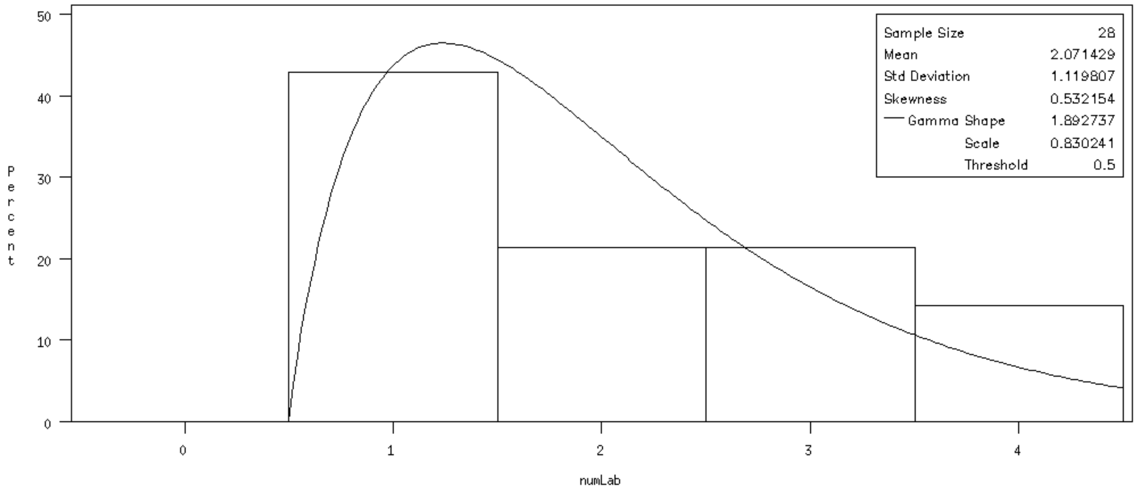
Labs Tests for Severity 3



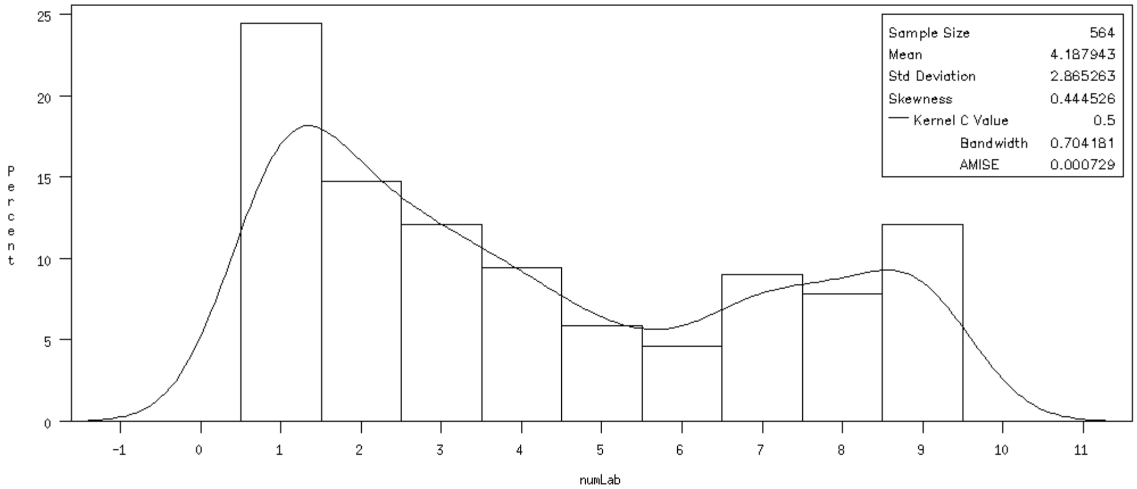
Labs Tests for Severity 4



Labs Tests for Severity 5



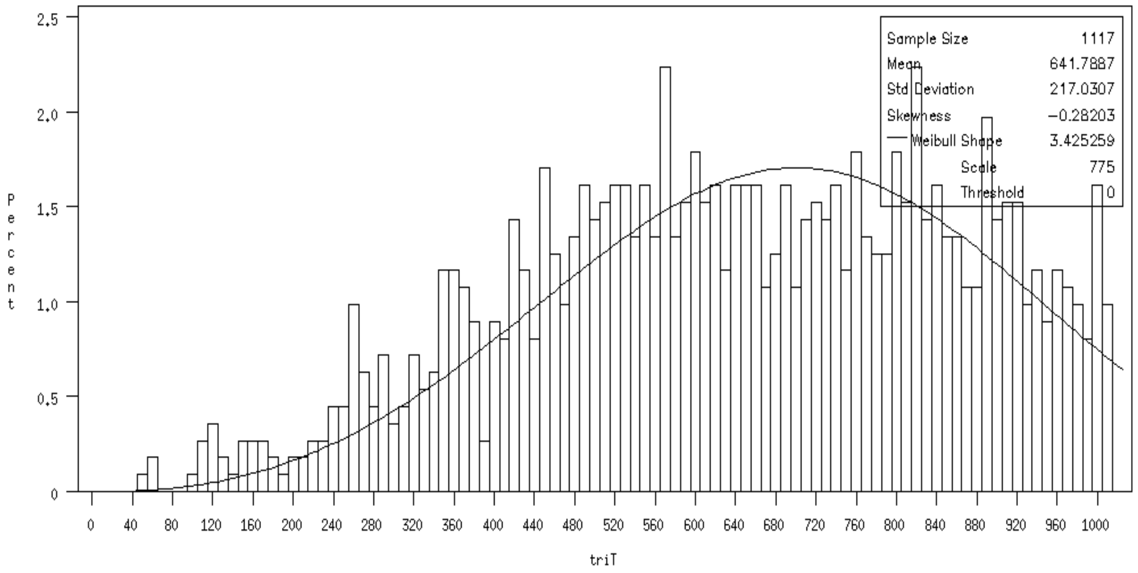
Labs Tests for Severity NA



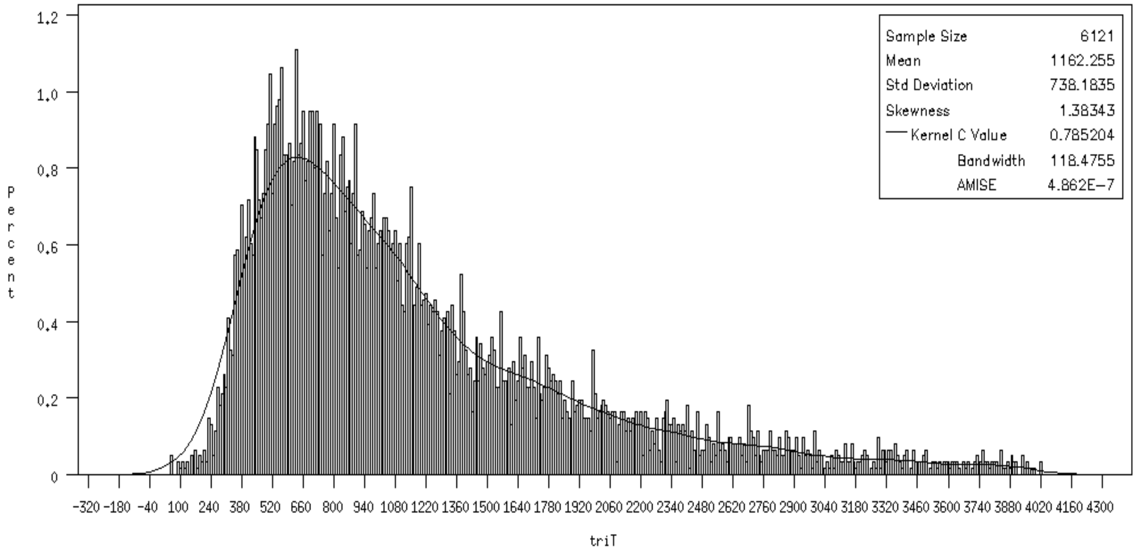
Appendix E

Statistical Graphs – Triage Times by Severity

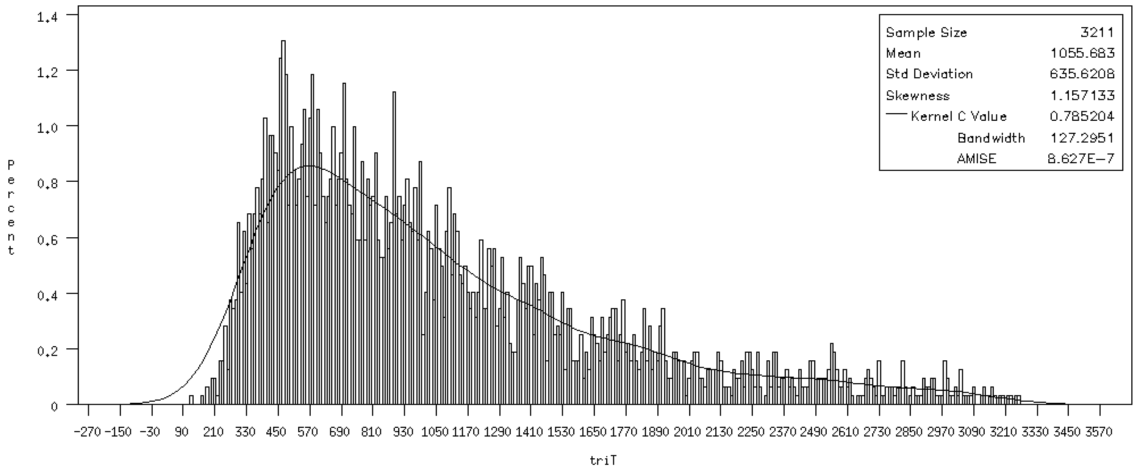
Triage Time for Severity 1 or 2



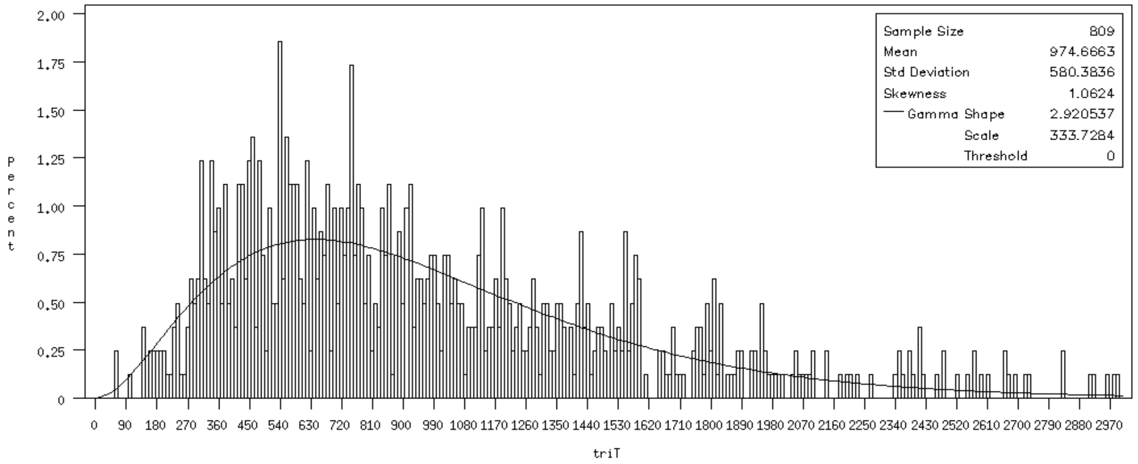
Triage Time for Severity 3



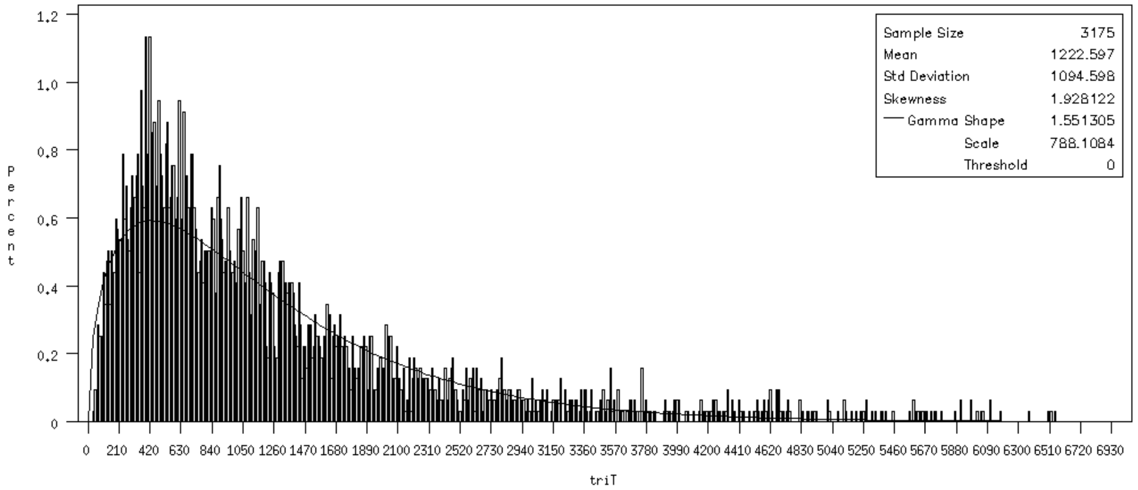
Triage Time for Severity 4



Triage Time for Severity 5

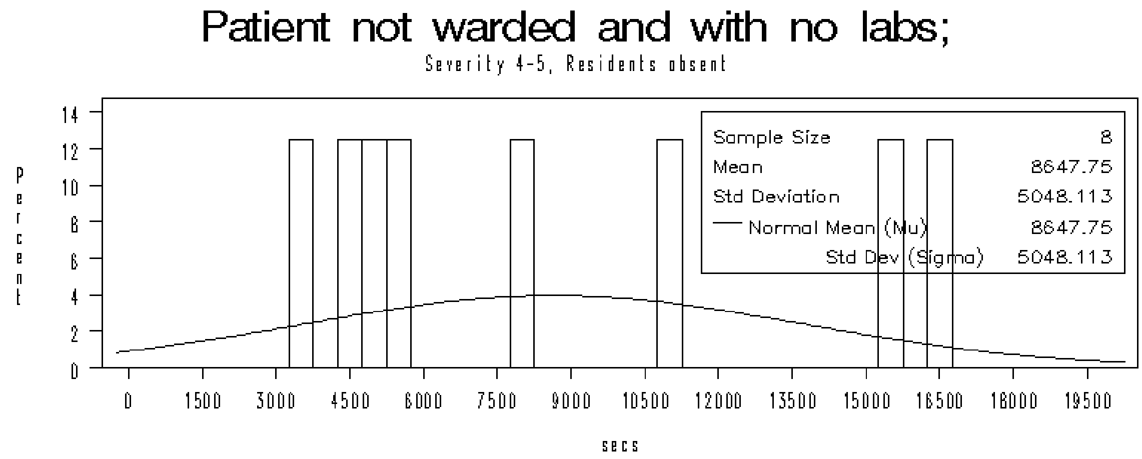


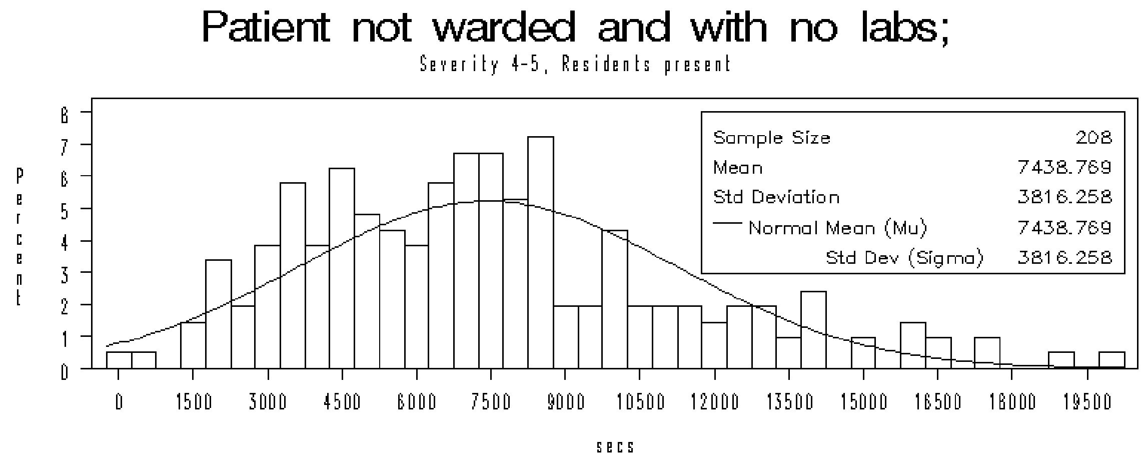
Triage Time for Severity NA

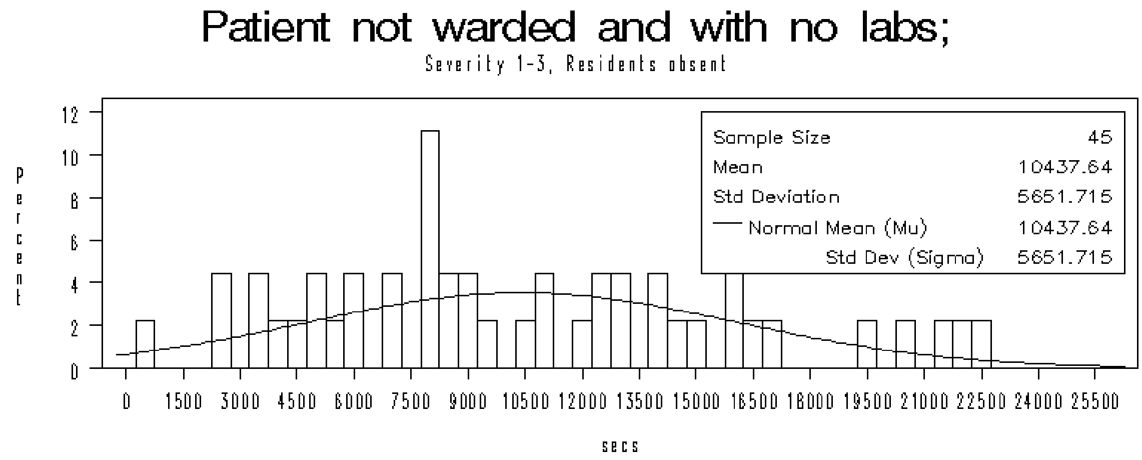


Appendix F

Statistical Graphs – Patients Not Warded with No Labs

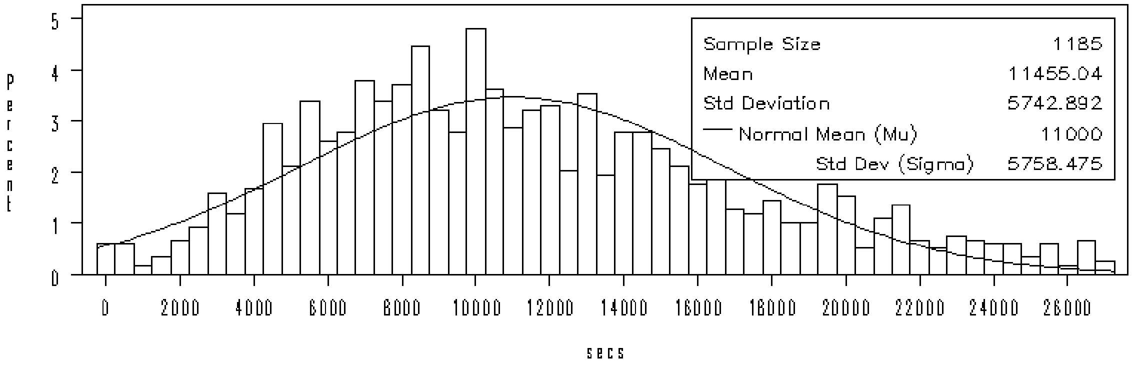






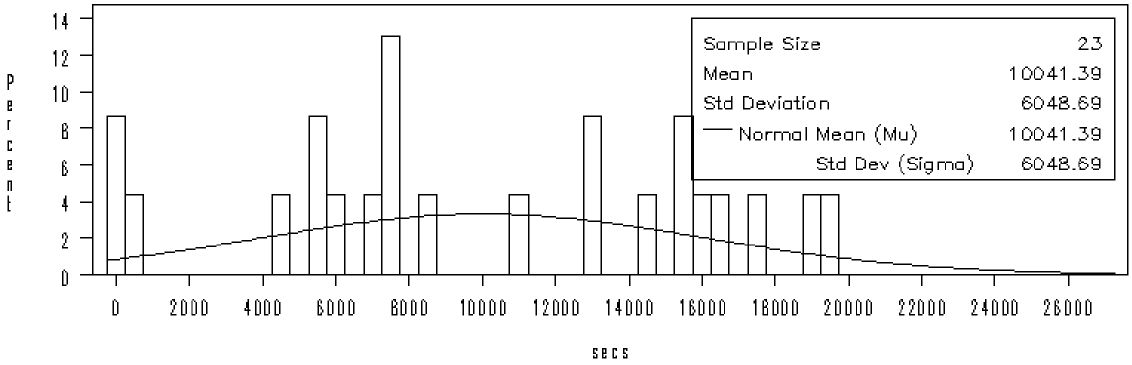
Patient not warded and with no labs;

Severity 1-3, Residents present



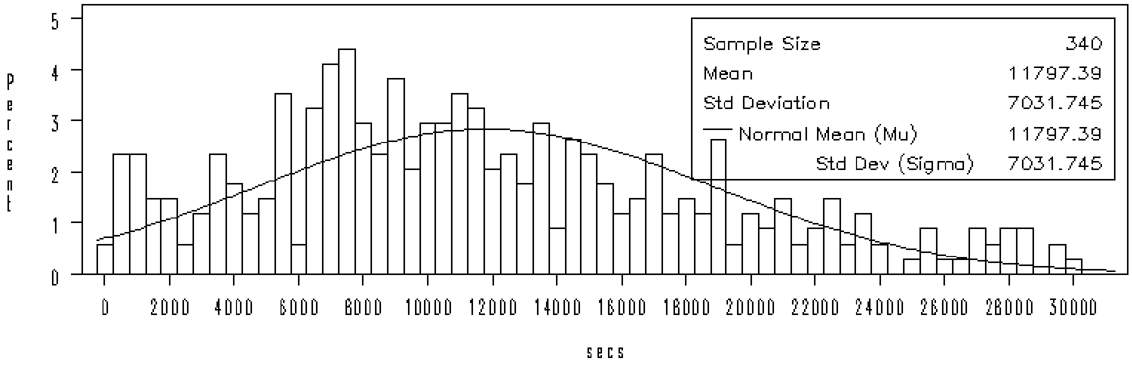
Patient not warded and with no labs;

Severity NA, Residents absent



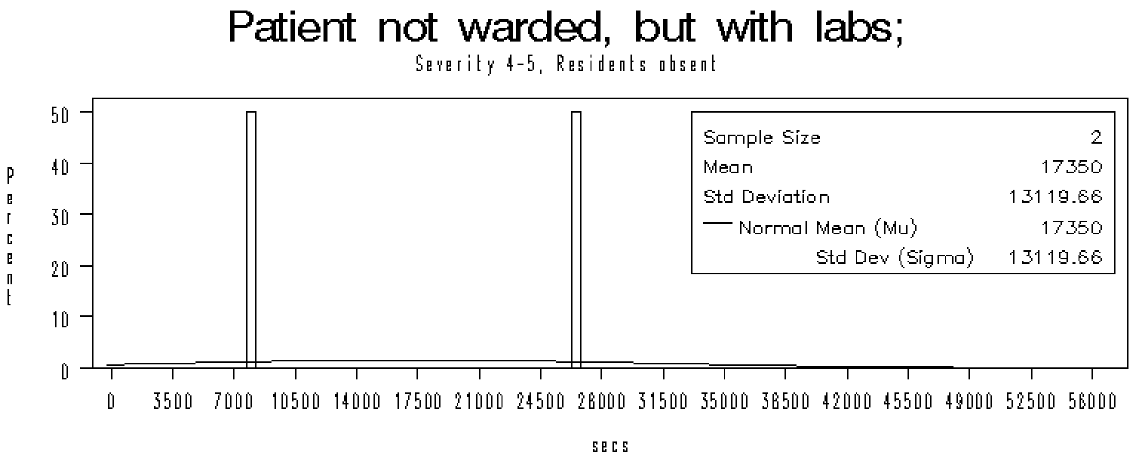
Patient not warded and with no labs;

Severity NA, Residents present

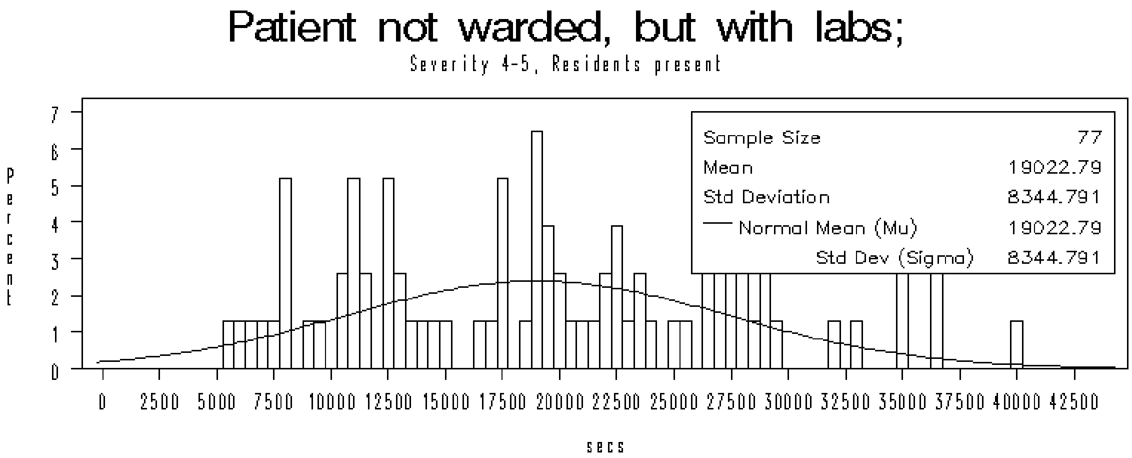


Appendix G

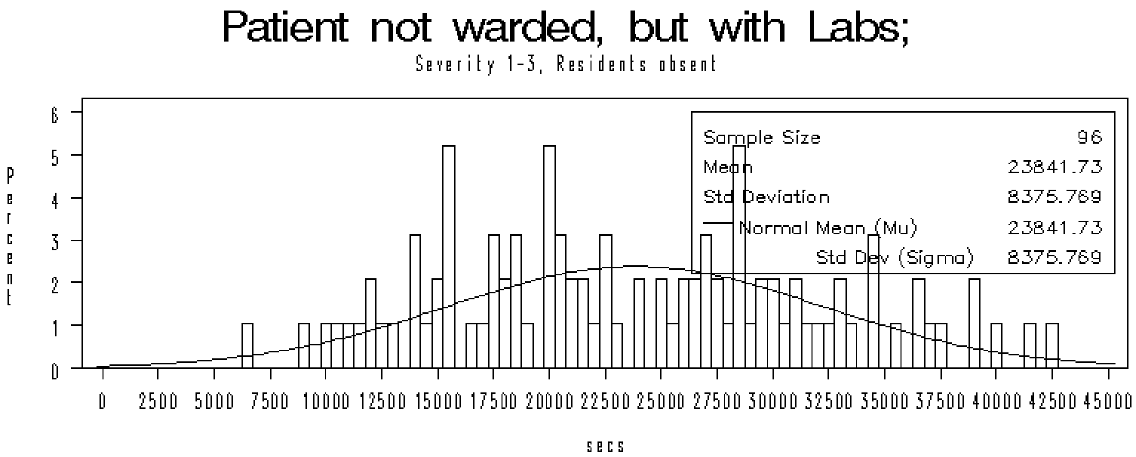
Statistical Graphs – Patients Not Warded with Labs


Patient not warded, but with labs;

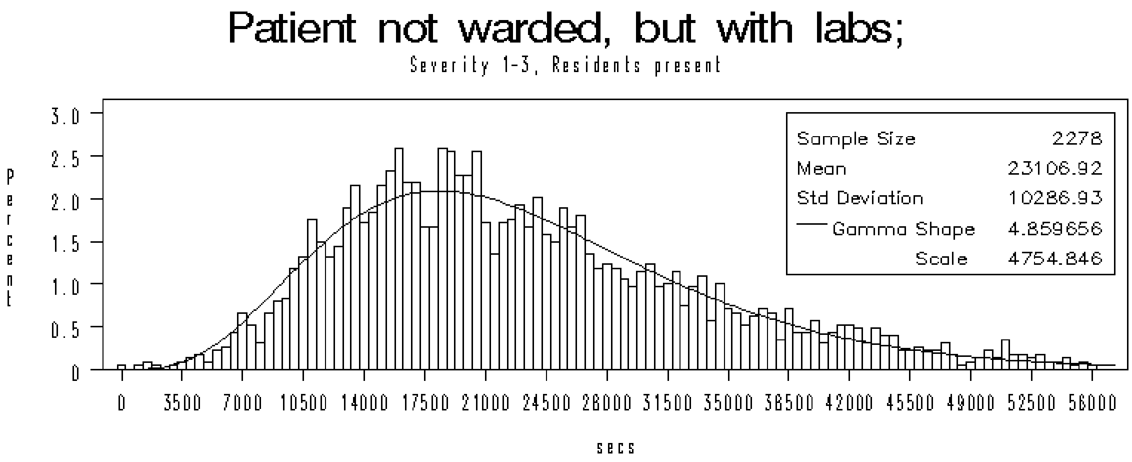
Severity 4-5, Residents absent

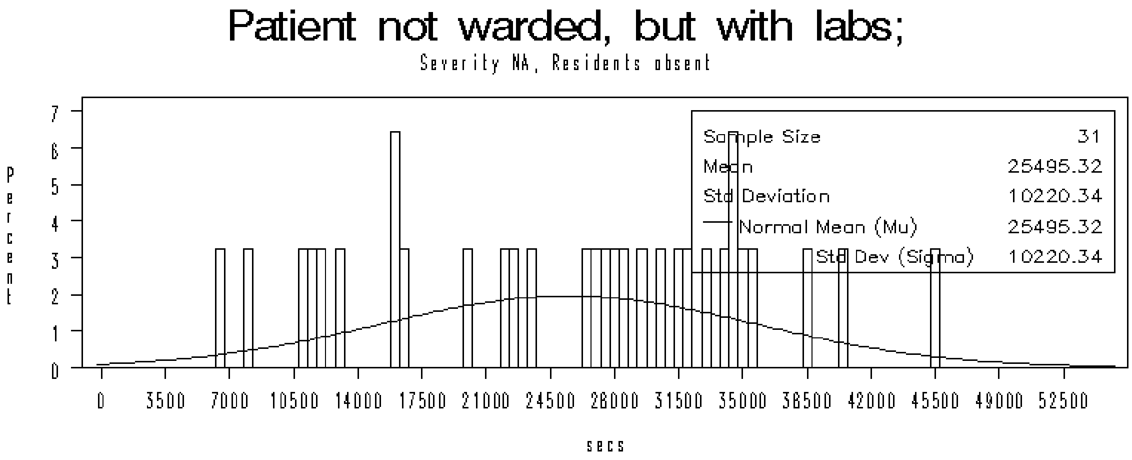

Patient not warded, but with labs;

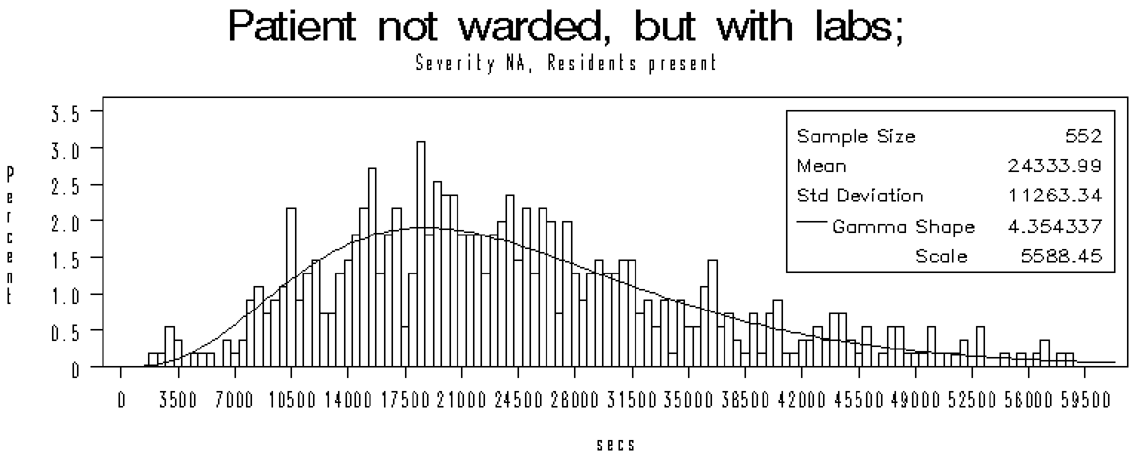
Severity 4-5, Residents present


Patient not warded, but with Labs;

Severity 1-3, Residents absent

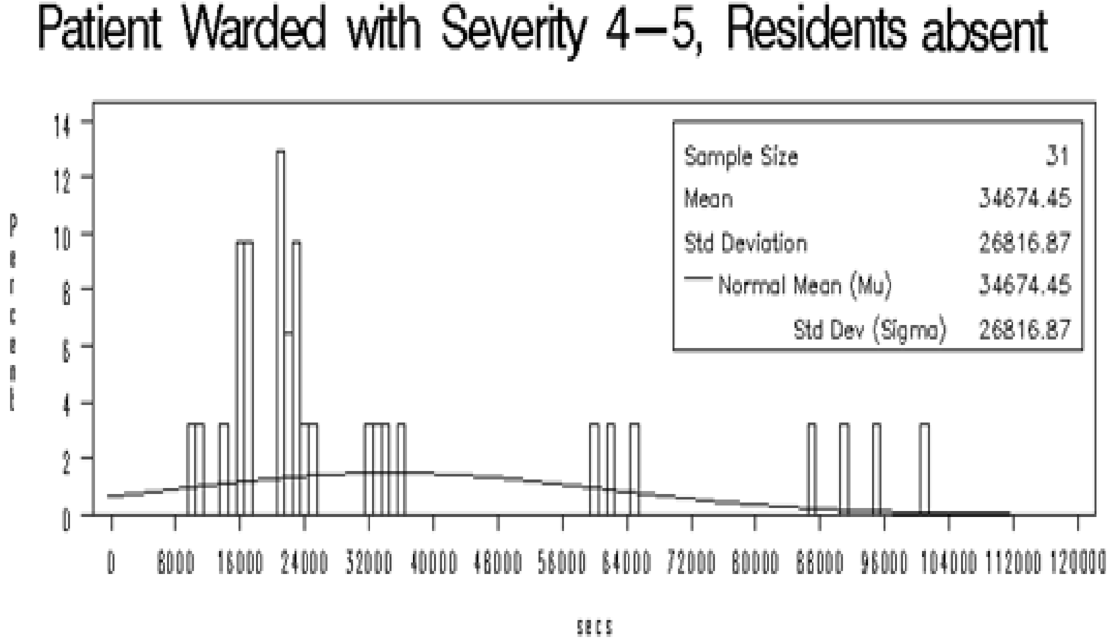

Patient not warded, but with labs;
Severity 1-3, Residents present
Percent
3.0
2.5
2.0
1.5
1.0
0.5
0
0
3500
7000
10500
14000
17500
21000
24500
28000
31500
35000
38500
42000
45500
49000
52500
56000
secs
Sample Size 2278
Mean 23106.92
Std Deviation 10286.93
Gamma Shape 4.859656
Scale 4754.846

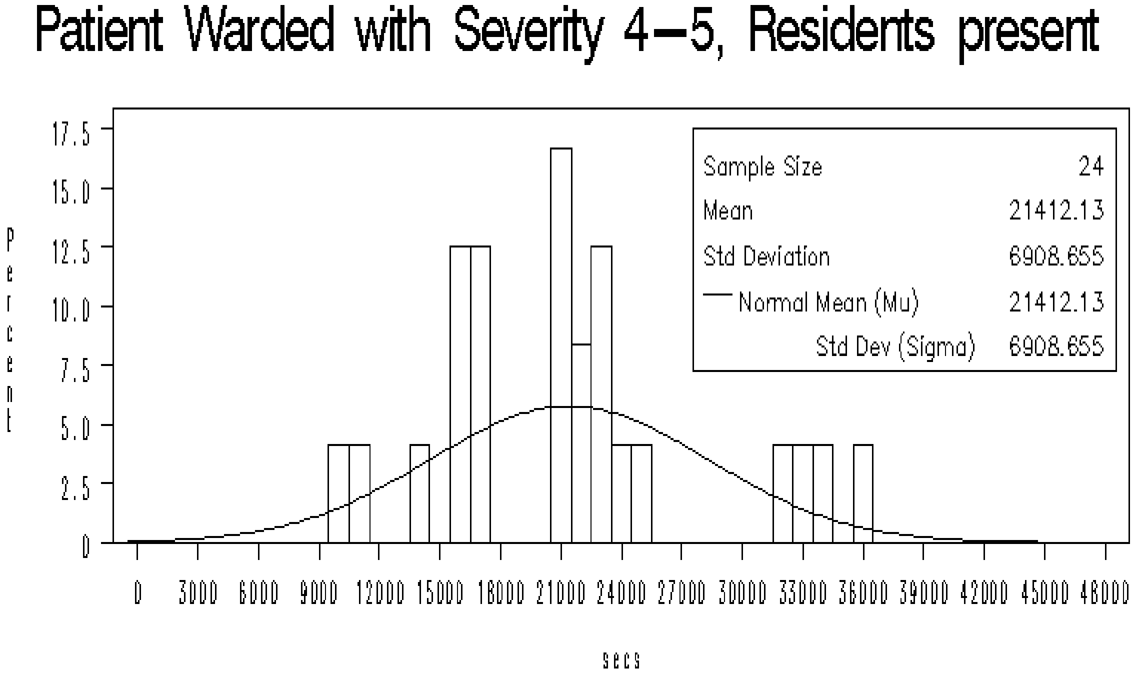

Patient not warded, but with labs;
Severity NA, Residents absent
Percent
7
6
5
4
3
2
1
0
0
3500
7000
10500
14000
17500
21000
24500
28000
31500
35000
38500
42000
45500
49000
52500
secs
Sample Size 31
Mean 25495.32
Std Deviation 10220.34
Normal Mean (Mu) 25495.32
Std Dev (Sigma) 10220.34


Patient not warded, but with labs;
Severity NA, Residents present
Percent
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
0
3500
7000
10500
14000
17500
21000
24500
28000
31500
35000
38500
42000
45500
49000
52500
56000
59500
secs
Sample Size 552
Mean 24333.99
Std Deviation 11263.34
Gamma Shape 4.354337
Scale 5588.45

Appendix H

Statistical Graphs – Patients Warded with Severity 4-5

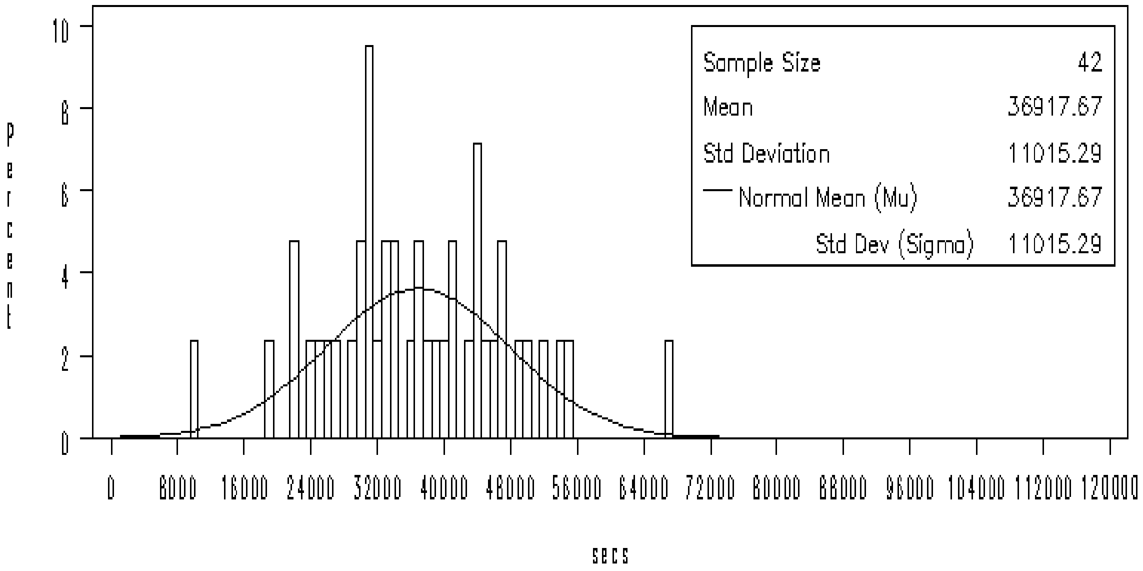




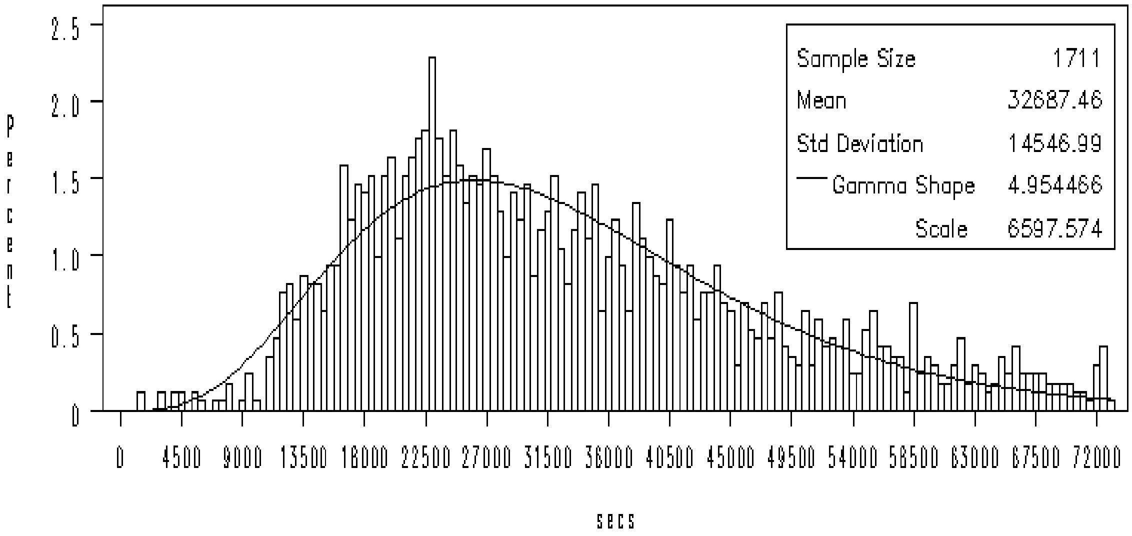
Appendix I

Statistical Graphs – Patients Warded with Severity 1-3

Patient Warded with Severity 1–3, Residents absent

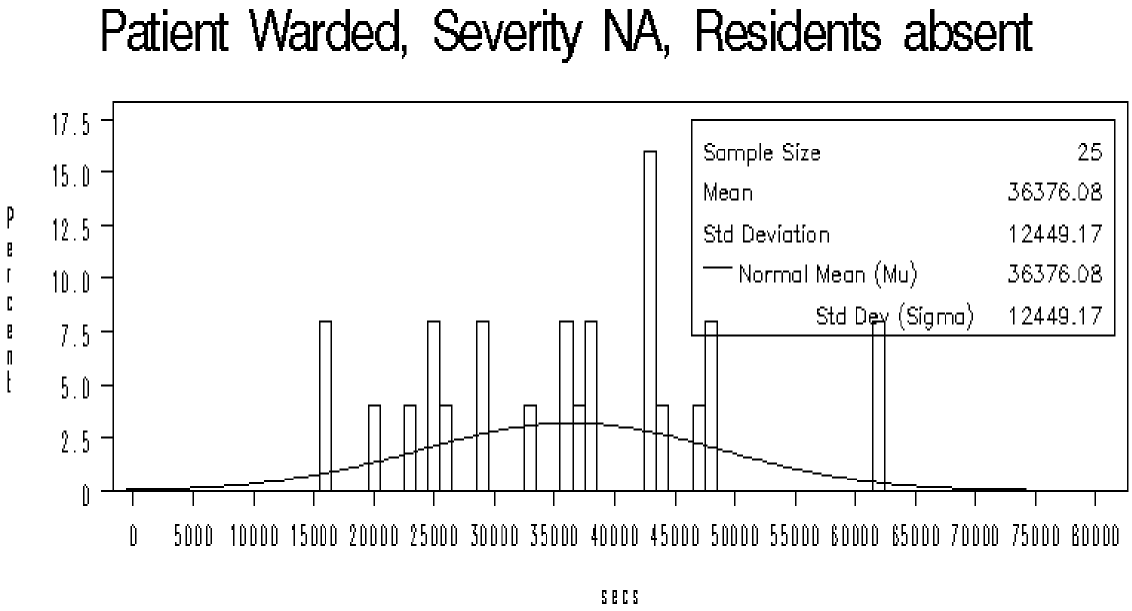


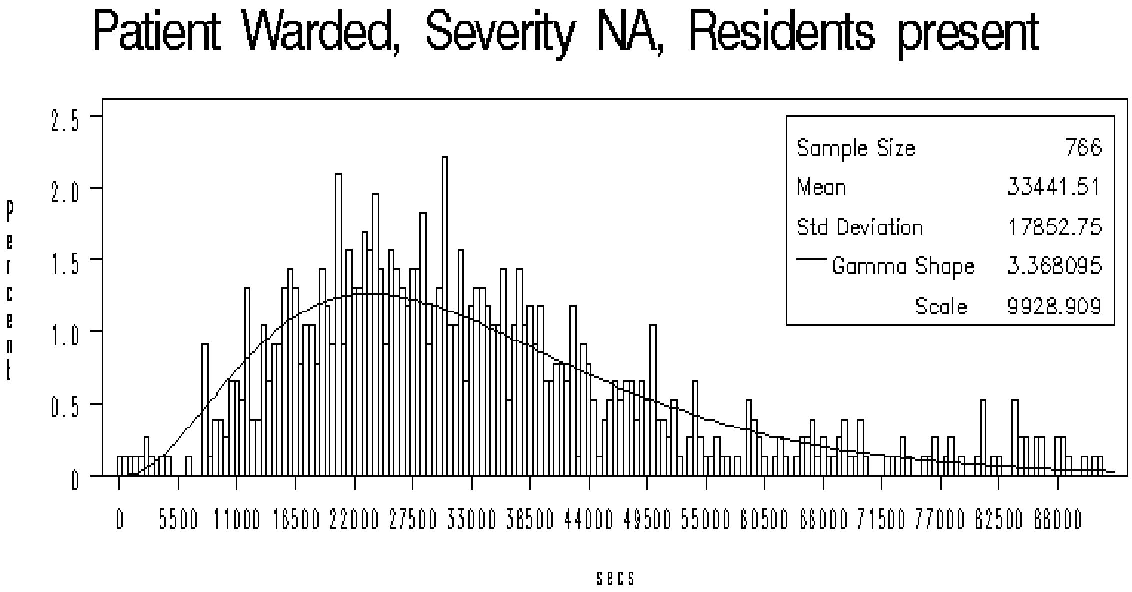
Patient Warded with Severity 1–3, Residents present



Appendix J

Statistical Graphs – Patients Warded with Severity NA

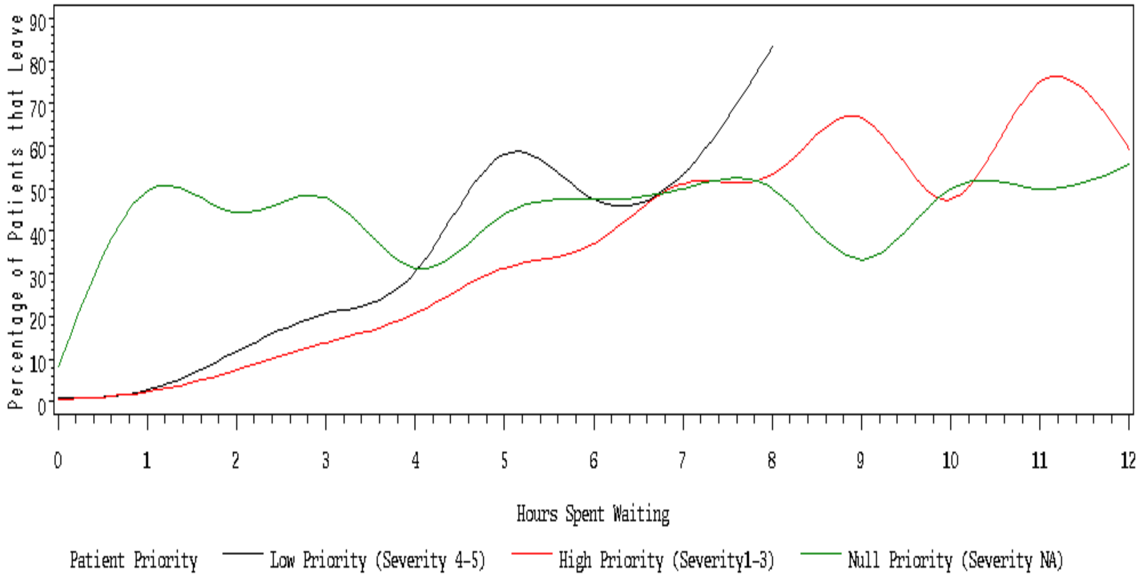

Patient Warded, Severity NA, Residents absent


Patient Warded, Severity NA, Residents present

Appendix K

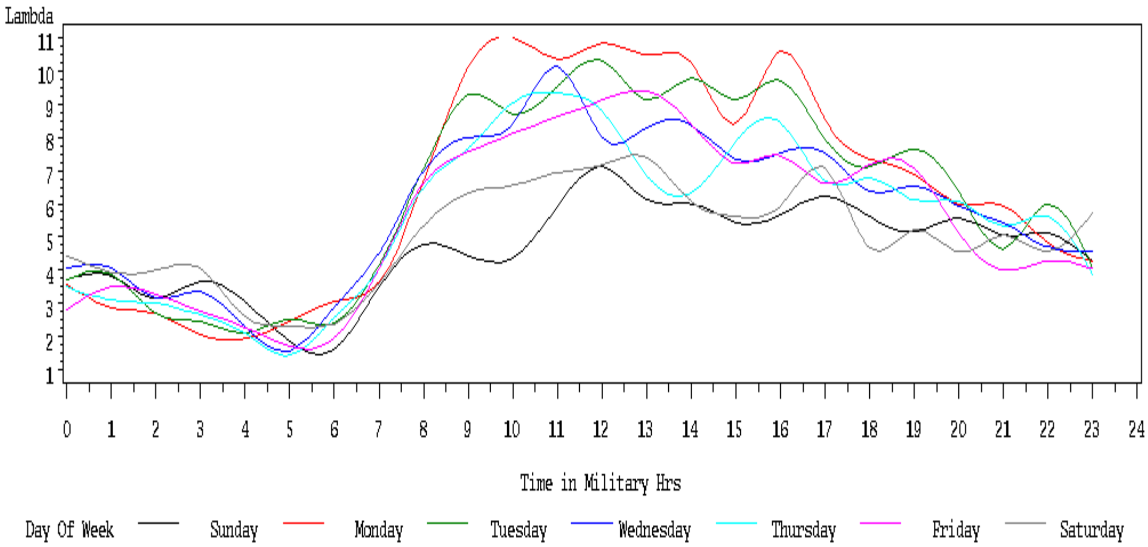
Additional Statistical Graphs

Probability of NAWCing over Time



Poisson Coefficients (Rates) of Patient Arrivals

Organized by Day and Time



Appendix L

Simulation Model – JAVA GUI Programming Code

```
package hope;



/**

  The AddAction class is a UIAction to add a new entity to the UI.

 */



public class AddDoctorAction extends UIAction {

   private int type;

   private String name;

   private String description;

   private int location;



   /**

     Initialize the AddAction. This stores information about the new

     entity.
   @param type          The type of the patient (e.g. OutputterUI.ID_PATIENT)

   @param id            A unique ID (of any entity in the whole OutputterUI)

   @param name          The name of the entity

   @param description   The entity's initial description

   @param location      The location ID of the entity's initial location

   @param time          The simulation model time when the entity will be added

   @param oui           The OutputterUI for the entity to be added to

  */

public AddDoctorAction(int id, String name, String description,

                       int location, double time, OutputterUI oui) {

        super(time, id, oui);

        this.name = name;

        this.description = description;

        this.location = location;

}



/**

   Please see {@link UIAction}.

  */
  public void execute() {


        Entity newOne = new Doctor(getId(), name, description,

                                     location, super.getOutputterUI(),

                                     super.getTime());

        super.getOutputterUI().insertEntity(newOne);



        }

}
```

```
package hope;


/**
   The AddAction class is a UIAction to add a new entity to the UI.
 */


public class AddNurseAction extends UIAction {
    private int type;
    private String name;
    private String description;
    private int location;


    /**
       Initialize the AddAction. This stores information about the new
       entity.


       @param type          The type of the patient (e.g. OutputterUI.ID_PATIENT)
       @param id           A unique ID (of any entity in the whole OutputterUI)
       @param name          The name of the entity
   @param description    The entity's initial description
   @param location       The location ID of the entity's initial location
   @param time           The simulation model time when the entity will be added
   @param oui            The OutputterUI for the entity to be added to
 */
public AddNurseAction(int id, String name, String description,
                     int location, double time, OutputterUI oui) {
       super(time, id, oui);
       this.name = name;
       this.description = description;
       this.location = location;
}


/**
   Please see {@link UIAction}.
 */



public void execute() {
		Entity newOne = new Nurse(getId(), name, description,

						location, super.getOutputterUI(),

						super.getTime());

		super.getOutputterUI().insertEntity(newOne);



		}

	}
```

```
package hope;

/**
   The AddAction class is a UIAction to add a new entity to the UI.
 */

public class AddPatientAction extends UIAction {
    private int type;
    private String name;
    private String description;
    private int location;
              private String diagnosisCode;
              private int age;
              private String race;
    private int triageNum;
    /**
       Initialize the AddAction. This stores information about the new
       entity.
```

```
     @param type          The type of the patient (e.g. OutputterUI.ID_PATIENT)
     @param id            A unique ID (of any entity in the whole OutputterUI)
     @param name          The name of the entity
     @param description   The entity's initial description
     @param location      The location ID of the entity's initial location
     @param time          The simulation model time when the entity will be added
     @param oui           The OutputterUI for the entity to be added to
    */
   public AddPatientAction(int id, String name, String diagnosisCode, int age, String race, int triageNum,
String description,
                          int location, double time, OutputterUI oui) {
        super(time, id, oui);
        this.name = name;
        this.description = description;
        this.location = location;
        this.diagnosisCode = diagnosisCode;
        this.age = age;
        this.race = race;
        this.triageNum = triageNum;
    }



    /**

       Please see {@link UIAction}.

     */




    public void execute() {

              Entity newOne = new Patient(getId(), name, description,

                                              location, super.getOutputterUI(),

                                              super.getTime(), diagnosisCode, age, race, triageNum);

              super.getOutputterUI().insertEntity(newOne);



              }

}
```

```
package hope;


import javax.swing.filechooser.FileFilter;
import java.io.File;


/**
   The AllImageFileFilter class creates a file filter to only select image files
   (and folders).
 */
public class AllImageFileFilter extends FileFilter
{
    private String description;


    /**
       Initialize the filter.


       @param description The filter's description (shown in the bottom of the select file dialog)
     */
    public AllImageFileFilter(String description)
```

```
{
        this.description = description;
}


/**
  Only accepts image files and folders.
 */
public boolean accept(File f)
{
        if (f.isDirectory())
           return true;
        String name = f.getName();
        if (name.indexOf('.') < 0)
           return false;
        String extension = name.substring(name.lastIndexOf('.')).
           toLowerCase();
        return (extension.equals(".jpg") || extension.equals(".jpeg") ||
                   extension.equals(".png") || extension.equals(".bmp") ||
                   extension.equals(".gif") || extension.equals(".tif") ||
                        extension.equals(".tiff"));

    }



    /**

       Returns the description.

     */

    public String getDescription()

    {

            return description;

    }

}
```

```
package hope;
import java.math.*;


/**
  The AvgBedLabel class is a metric that measure the average number of seconds
  patients have been in beds.
 */
public class AvgBedLabel extends MetricLabel
{
   private static String text = "Avg. Time In Beds";
   /**
     This initializes the AvgBedLabel.

     @param enabled Whether the AvgBedLabel is initially enabled (displayed on the MetricPanel)
     @param mp      The MetricPanel associated with this AvgBedLabel
   */
   public AvgBedLabel(boolean enabled, MetricPanel mp)
   {
          super(text, enabled, text+": --", mp);
```

```
}

/**
   Please see {@link MetricLabel}
   Again, assumes only bed location IDs are >= 500
*/
public void update()
{
          int n = 0;
          double sum = 0;
          for (Entity q : getMetricPanel().getEntities()) {
              if (q instanceof Patient && q.getLocation() >= 500)
                        {
                            double startTime = q.getFinalTime();
                            double nowTime = getMetricPanel().getTime();
                            double interTime = nowTime - startTime;
                            sum += interTime;
                            n++;
                        }
```

```
       }
       if (n == 0)
         {
              setText(text+": --");
              return;
         }
       double avgBedTime = sum / n;

       int decimalPlace = 0;
       BigDecimal bd = new BigDecimal(avgBedTime);
       bd = bd.setScale(decimalPlace,BigDecimal.ROUND_UP);
       double avgBedFinal = bd.doubleValue();
       setText(text+": "+avgBedFinal);
  }
}
```

```
package hope;

import java.math.*;



/**

  The AvgWaitingLabel class is a MetricLabel that shows the average number of seconds ED patients have been in the ED.

 */

public class AvgWaitingLabel extends MetricLabel

{

    private static String text = "Avg. Time In Waiting Room";

    /**

       This initializes the AvgWaitingLabel.



       @param enabled Whether the AvgWaitingLabel is initially enabled (displayed on the MetricPanel)

       @param mp      The MetricPanel associated with this AvgWaitingLabel

    */

    public AvgWaitingLabel(boolean enabled, MetricPanel mp)

    {

            super(text, enabled, text+": --", mp);
```

```
}



/**

   Please see {@link MetricLabel}

*/

public void update()

{

          int n = 0;

          double sum = 0;

          for (Entity q : getMetricPanel().getEntities()) {

              if (q instanceof Patient && q.getLocation() == 2)

                        {

                            double startTime = q.getEntryTime();

                            double nowTime = getMetricPanel().getTime();

                            double interTime = nowTime - startTime;

                            sum += interTime;

                            n++;

                        }

          }
```

```
            if (n == 0)
                {
                        setText(text+": --");
                        return;
                }
            double avgWait = sum / n;

            int decimalPlace = 0;
            BigDecimal bd = new BigDecimal(avgWait);
            bd = bd.setScale(decimalPlace,BigDecimal.ROUND_UP);
            double avgWaitFinal = bd.doubleValue();
            setText(text+": "+avgWaitFinal);
    }
}
```

```
package hope;


/**
   The BedLabel class is a MetricLabel that says the number of patients
   in beds.
 */
public class BedLabel extends MetricLabel
{
    private static String text = "# In Beds";
    /**
       This initializes the BedLabel.

       @param enabled Whether the BedLabel is initially enabled (displayed on the MetricPanel)
       @param mp      The MetricPanel associated with this BedLabel
     */
    public BedLabel(boolean enabled, MetricPanel mp)
    {
              super(text, enabled, text+": 0", mp);
    }
    /**
       Please see {@link MetricLabel}
       This method assumes all bed location IDs are >= 500, and that no other location IDs are
     */
    public void update()
    {
              int inBeds = 0;
              for (Entity q : getMetricPanel().getEntities())
                  if (q instanceof Patient && q.getLocation() >= 500)
                            inBeds++;
              setText(text+": "+inBeds);
    }
}
```

```
package hope;

import java.awt.Color;

/**
  The PatientRepresentation class represents any patient in the simulation
  display.
 */
public class BlackPatientRepresentation extends EntityRepresentation
{
    /**
      This sets up the default patient display (6x6 red dots).
     */
    public BlackPatientRepresentation()
    {
            super("Patient", Color.red, 12, "evil.png", 23, -1,
                "evil2.png", 20, -1, true, new PatientRepresentation());
    }
    /**

       This represents any Entity that is a patient.

     */

    public boolean represents(Entity e)

    {

            return (e instanceof Patient && ((Patient)e).getRace() == "Black");

    }

}
```

```
package hope;



/**

   The CanvasRegion class represents a rectangular region on the ED display.

 */



public class CanvasRegion

{

   private Pixel middle;

   private int width;

   private int height;

   /**

      Initializes the CanvasRegion.



      @param middle  The pixel in the middle of the rectangular region

      @param width   The width of the region in pixels

      @param height  The height of the region in pixels

    */

   public CanvasRegion(Pixel middle, int width, int height)
```

```
          {
              this.middle = middle;
              this.width = width;
              this.height = height;
          }


/**
   Copy constructor


   @param other  The CanvasRegion to copy
 */
public CanvasRegion(CanvasRegion other)
          {
          this(other.middle, other.width, other.height);
          }


/**
   Assumes CanvasRegions are equal if they have the same middle and the
   same width and height.
 */
public boolean equals(Object o)
{
          if (o instanceof CanvasRegion)
              {
                        CanvasRegion cr = (CanvasRegion)o;
                        return (cr.getMiddle().equals(middle) &&
                                    cr.getWidth() == width && cr.getHeight() == height);
              }
          else
              return false;
}


/**
   Getter for the height of the region


   @return The height in pixels of the CanvasRegion
 */
public int getHeight()
```

```
{
      return height;
}



/**
  Getter for the middle pixel in the region


  @return The middle pixel of the CanvasRegion
 */
public Pixel getMiddle()
{
      return middle;
}



/**
  Getter for the width of the region


  @return The width in pixels of the CanvasRegion
 */
```

```
public int getWidth()

{

        return width;

}




/**

   A check to see if two CanvasRegions overlap each other



   @param cr The CanvasRegion to be checked against this one for overlap

   @return   A boolean saying if they overlap

 */

public boolean overlaps(CanvasRegion cr)

{

        int l = middle.getX() - width/2;

        int r = l + width;

        int t = middle.getY() - height/2;

        int b = t + height;

        int ol = cr.getMiddle().getX() - cr.getWidth()/2;

        int or = ol + cr.getWidth();
        int ot = cr.getMiddle().getY() - cr.getHeight()/2;
        int ob = ot + cr.getHeight();
        if (l >= or || r <= ol || t >= ob || b <= ot)
           return false;
        else
           return true;
}
/**
  A check to see if a CanvasRegion contains a pixel coordinate

  @param p The Pixel to be checked for containment in this CanvasRegion
  @return  A boolean saying if this contains p
 */
public boolean contains(Pixel p){
        int l = middle.getX() - width/2;
        int r = l + width;
        int t = middle.getY() - height/2;
        int b = t + height;
        int x = p.getX();
```

```
            int y = p.getY();

            if (l >= x || r <= x || t >= y || b <= y)

                return false;

            else

                return true;

    }

}
```

```
package hope;


/**
   The CensusLabel class is a MetricLabel showing the current ED census.
 */
public class CensusLabel extends MetricLabel
{
    private static String text = "Census";
    /**
       This initializes the CensusLabel.


       @param enabled Whether the CensusLabel is initially enabled (displayed on the MetricPanel)
       @param mp      The MetricPanel associated with this CensusLabel
     */
    public CensusLabel(boolean enabled, MetricPanel mp)
    {
              super(text, enabled, text+": 0", mp);
    }
    /**

       Please see {@link MetricLabel}

     */

    public void update()

    {

              int census = 0;

              for (Entity q : getMetricPanel().getEntities())

                  if (q instanceof Patient)

                            census++;

              setText(text+": "+census);

    }

}
```

```
package hope;

/**
   The ChangeDescriptionAction class is a UIAction to change the description
   of an entity in the UI.
 */
public class ChangeDescriptionAction extends UIAction {
    private String newDescription;

    /**
       Initializes the ChangeDescriptionAction. Stores information about the
       new description.

       @param id             The unique ID of the entity being modified
       @param newDescription The new description of the entity
       @param time           The simulation time when the description will be changed
       @param oui            The OutputterUI the entity is in
     */
    public ChangeDescriptionAction(int id, String newDescription,
                                        double time, OutputterUI oui) {

              super(time, id, oui);

              this.newDescription = newDescription;

    }



    /**

       Please see {@link UIAction}.

     */

    public void execute() {

              super.getOutputterUI().getEntity(getId()).

                  setDescription(newDescription);

    }

}
```

```
package hope;



import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;



/**

   The ClickListener class is a mouse listener for the OutputterUI that

   triggers an attempt to display Entity info on a click.

 */

public class ClickListener implements MouseListener {

              OutputterUI oui;



    /**

       Initialize the ClickListener.



       @param oui The OutputterUI the ClickListener will be notifying of clicks

     */

    public ClickListener(OutputterUI oui){

              this.oui = oui;
    }

    /**
       Forwards mouse clicks to the associated OutputterUI.
     */
    public void mouseClicked(MouseEvent e){
              oui.displayEntityInfo(new Pixel(e.getX(), e.getY()));
    }

    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}

}
```

```
package hope;



import javax.swing.JFrame;

import javax.swing.JButton;

import javax.swing.JPanel;

import javax.swing.border.EmptyBorder;

import java.awt.event.ActionListener;

import java.awt.event.ActionEvent;

import java.awt.GridLayout;



/**

   The CustomizationFrame class provides the user display customization options. These are things like
customizing the avatars and customizing the metrics displayed.

 */

public class CustomizationFrame extends JFrame implements ActionListener

{

    private JButton customizeAvatarsButton;

    private JButton customizeMetricsButton;

    private OutputterUI oui;
private MetricPanel mp;



/**

   This initializes the CustomizationFrame. It lays out all the components.



   @param oui The OutputterUI for which we are customizing the display

   @param mp  The MetricPanel that we could be customizing

 */

public CustomizationFrame(OutputterUI oui, MetricPanel mp)

{

          super("Customize your display");

          this.oui = oui;

          this.mp = mp;

          customizeAvatarsButton = new JButton("Customize Avatars");

          customizeAvatarsButton.addActionListener(this);

          customizeMetricsButton = new JButton("Customize Metrics");

          customizeMetricsButton.addActionListener(this);

          JPanel cabPanel = new JPanel();

          cabPanel.add(customizeAvatarsButton);
          cabPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

          JPanel cmbPanel = new JPanel();

          cmbPanel.add(customizeMetricsButton);

          cmbPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

          JPanel wholePanel = new JPanel(new GridLayout(2, 1));

          wholePanel.add(cabPanel);

          wholePanel.add(cmbPanel);

          getContentPane().add(wholePanel);

          setSize(300, 150);

          setVisible(true);

}



/**

   Handle button clicks

 */

public void actionPerformed(ActionEvent e)

{

          if (e.getSource().equals(customizeAvatarsButton))

              {
```

```
              setVisible(false);

              new CustomizeAvatarFrame(oui);

              dispose();

         }

       if (e.getSource().equals(customizeMetricsButton))

         {

              setVisible(false);

              new CustomizeMetricsFrame(mp);

              dispose();

         }

  }

}
```

```
package hope;


import java.awt.Canvas;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;


/**
   The CustomizeAvatarCanvas class displays the current version of an avatar.
   It shows either the default dot or the selected image.
 */
public class CustomizeAvatarCanvas extends Canvas
{
    private Image img;
    private String imageName;
    private Color bgColor;
    private Color dotColor;
    private int dotEdge;


    /**
       This initializes the canvas. The initial canvas will display nothing; it
       has neither a dot nor an image selected.


       @param width   The width of the canvas in pixels
       @param height  The height of the canvas in pixels
       @param bgColor The background color of the canvas
     */
    public CustomizeAvatarCanvas(int width, int height, Color bgColor)
    {
        setSize(width, height);

        this.bgColor = bgColor;

        img = null;

        imageName = null;

        dotColor = null;

        dotEdge = -1;

}


/**

  Draws the selected display (a dot, an image, or nothing)

 */

public void paint(Graphics g)

{

        g.setColor(bgColor);

        g.fillRect(0, 0, getWidth(), getHeight());

        if (img != null)

           g.drawImage(img, (getWidth() - img.getWidth(null))/2,

                              (getHeight() - img.getHeight(null))/2, null, null);

        else if (dotColor != null)

           {

                   g.setColor(dotColor);

                   g.fillOval((getWidth() - dotEdge) / 2,

                                (getHeight() - dotEdge) / 2, dotEdge, dotEdge);

           }

}


/**

  This gets the color of the dot associated with this Canvas. If the dot

  is not the currently displayed item, null is returned.
```

```
   @return The dot color (or null if the dot is not active)
 */
public Color getDotColor()
{
        return dotColor;
}

/**
   This gets the width and height of the dot associated with this Canvas.
   If the dot is not the currently displayed item, -1 is returned.

   @return The width and height of the dot in pixels (or -1 if the dot is not active)
 */
public int getDotEdge()
{
        return dotEdge;
}

/**
   This gets the image associated with this Canvas. If there is no active
   image (the dot is active), null is returned.

   @return The image associated with the Canvas (or null if no image is active)
 */
public Image getImage()
{
        return img;
}
```

```
/**
   This gets the image name of the image associated with this Canvas. If
   there is no active image (the dot is active), null is returned. The name
   is based in the avatars folder.

   @return The name of the image associated with the Canvas (or null if no image is active)
 */
public String getImageName()
{
        return imageName;
}

/**
   This sets the thing to be displayed to be the default dot. It specifies
   the size and color of the dot.

   @param dotColor The color of the dot
   @param dotEdge  The width and height in pixels of the dot
 */
public void useDot(Color dotColor, int dotEdge)
{
        this.dotColor = dotColor;
        this.dotEdge = dotEdge;
        img = null;
        imageName = null;
        repaint();
}
```

```
    /**
      This setes the thing to be displayed to be an image.


      @param img       The image to be used
      @param imageName The name of the image being used (based in the avatars directory)
     */
    public void useImage(Image img, String imageName)
    {
            this.img = img;
            this.imageName = imageName;
            dotColor = null;
            dotEdge = -1;
            repaint();
    }
}
```

```
package hope;



import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JButton;

import javax.swing.JScrollPane;

import javax.swing.JOptionPane;

import javax.swing.border.EmptyBorder;

import java.awt.Container;

import java.awt.GridLayout;

import java.awt.BorderLayout;

import java.awt.event.ActionListener;

import java.awt.event.ActionEvent;

import java.awt.event.WindowListener;

import java.awt.event.WindowEvent;

import java.util.List;



/**
```

```
  The CustomizeAvatarFrame class provides the frame used to customize the
  avatars. It provides scroll bar services and sets up the CustomizeAvatarRows
  that do the actual customization work.
 */
public class CustomizeAvatarFrame extends JFrame implements ActionListener,
                                                          WindowListener
{
    private OutputterUI oui;
    private JButton okButton;
    private JButton cancelButton;
    private CustomizeAvatarRow[] rows;
    private static final int CANCEL_ACTION_CANCEL = 0;
    private static final int CANCEL_ACTION_YES = 1;
    private static final int CANCEL_ACTION_NO = 2;


    /**
      Create the avatar customization frame. This adds a vertical scroll bar.


      @param oui The OutputterUI on which these avatars will be displayed
 */

public CustomizeAvatarFrame(OutputterUI oui)

{

        super("Select custom avatars");

        this.oui = oui;

        Container c = getContentPane();

        List<EntityRepresentation> representations =

           oui.getEntityRepresentations();

        JPanel total = new JPanel(new BorderLayout());

        JPanel major = new JPanel(new GridLayout(representations.size(), 1));

        rows = new CustomizeAvatarRow[representations.size()];

        int pos = 0;

        for (EntityRepresentation er : representations)

           {

                   rows[pos] = new CustomizeAvatarRow(er, oui);

                   major.add(rows[pos]);

                   pos++;

           }

        JPanel bottomPanel = new JPanel(new GridLayout(1, 4));
```

```
bottomPanel.setBorder(new EmptyBorder(15, 15, 15, 15));

okButton = new JButton("OK");

okButton.addActionListener(this);

cancelButton = new JButton("Cancel");

cancelButton.addActionListener(this);

JPanel okPanel = new JPanel();

okPanel.add(okButton);

JPanel cancelPanel = new JPanel();

cancelPanel.add(cancelButton);

bottomPanel.add(new JLabel(""));

bottomPanel.add(okPanel);

bottomPanel.add(cancelPanel);

bottomPanel.add(new JLabel(""));

total.add(major, BorderLayout.NORTH);

total.add(bottomPanel, BorderLayout.SOUTH);

JScrollPane jsp = new JScrollPane(total);

jsp.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

c.add(jsp);

setSize(700, 620);
```

```
        setVisible(true);
        addWindowListener(this);
}

/**
  Handle mouse clicks.
 */
public void actionPerformed(ActionEvent e)
{
        if (e.getSource().equals(okButton))
           saveAction();
        else if (e.getSource().equals(cancelButton))
           {
                   switch(cancelAction())
                      {
                      case CANCEL_ACTION_CANCEL:
                              return;
                      case CANCEL_ACTION_YES:
                              saveAction();
```

```
                       break;
                 case CANCEL_ACTION_NO:
                       break;
                 default:
                       SimulationOutputter.die("Internal error: CustomizeAvatarFrame");
                 }
        }
      setVisible(false);
      dispose();
}

/**
  Pop a dialog if changes were made confirming the exit.
 */
public void windowClosing(WindowEvent e)
{
      switch (cancelAction())
        {
        case CANCEL_ACTION_CANCEL:
            return;
        case CANCEL_ACTION_YES:
            saveAction();
            break;
        case CANCEL_ACTION_NO:
            break;
        default:
            SimulationOutputter.die("Internal error: CustomizeAvatarFrame");
        }
      setVisible(false);
      dispose();
}

public void windowActivated(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}
```

```
private int cancelAction()

{

        boolean changed = false;

        for (CustomizeAvatarRow car : rows)

           if (car.isChanged())

                   changed = true;

        if (!changed)

           return CANCEL_ACTION_NO;

        int resp =

           JOptionPane.showConfirmDialog(null, "Would you like to save your changes?",

                                                   "Save changes?",

                                                   JOptionPane.YES_NO_CANCEL_OPTION);

        if (resp == JOptionPane.YES_OPTION)

           return CANCEL_ACTION_YES;

        else if (resp == JOptionPane.NO_OPTION)

           return CANCEL_ACTION_NO;

        else

           return CANCEL_ACTION_CANCEL;
```

```
    }


    private void saveAction()
    {
            for (CustomizeAvatarRow car : rows)
                car.writeChanges();
    }
}
```

```
package hope;

import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import javax.swing.JSlider;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;
import javax.swing.JColorChooser;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Image;
import java.awt.Dimension;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import javax.swing.filechooser.FileFilter;
import javax.swing.JFileChooser;
import java.io.File;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;

import javax.swing.JOptionPane;

/**
```

```
   The CustomizeAvatarRow class is an actual row that provides customization for
   an EntityRepresentation.
 */
public class CustomizeAvatarRow extends JPanel implements ChangeListener,
                                                            ActionListener
{
    private EntityRepresentation er;
    private CustomizeAvatarCanvas dotCanvas;
    private CustomizeAvatarCanvas usProvided1Canvas;
    private CustomizeAvatarCanvas usProvided2Canvas;
    private CustomizeAvatarCanvas themProvidedCanvas;
    private JButton uploadYourOwnButton;
    private JButton showAllOptionsButton;
    private JButton changeColorButton;
    private JPanel[] panels;
    private JPanel[] buttonPanels;
    private JRadioButton[] buttons;
    private JSlider[] sliders;
    private static final int MAX_DIMENSION = 80;
    private boolean userPictureUploaded;
    private ButtonGroup bg;
    private boolean changed;
    private int initButton;
    private OutputterUI oui;

    /**
       This initializes the row in the customization UI. It adds all the
       displays for the different options, the sliders, the selection buttons,
       and the further customization buttons.
   @param er The EntityRepresentation customized by this row
 */
public CustomizeAvatarRow(EntityRepresentation er, OutputterUI oui)
{
          super(new GridLayout(1, 6));
          this.er = er;
          this.oui = oui;
          JPanel[] canvasses = new JPanel[4];
          for (int i=0;i<4;i++)
             canvasses[i] = new JPanel();
          panels = new JPanel[6];
          for (int i=0;i<6;i++)
             panels[i] = new JPanel(new BorderLayout());
          bg = new ButtonGroup();
          buttons = new JRadioButton[4];
          for (int i=0;i<4;i++)
             {
                    buttons[i] = new JRadioButton();
                    bg.add(buttons[i]);
             }
          buttonPanels = new JPanel[4];
          for (int i=0;i<4;i++)
             {
                    buttonPanels[i] = new JPanel(new GridLayout(1, 3));
                    buttonPanels[i].add(new JLabel(""));
                    buttonPanels[i].add(buttons[i]);
                    buttonPanels[i].add(new JLabel(""));
             }
```

```
sliders = new CustomizeAvatarSlider[4];
panels[0].add(new JLabel(er.getName()+":"), BorderLayout.NORTH);


changeColorButton = new JButton("New Color");
changeColorButton.addActionListener(this);
dotCanvas = new CustomizeAvatarCanvas(90, 90, Color.blue);
int dotEdge = downScaleWidth(er.getDotEdge());
dotCanvas.useDot(er.getDotColor(), dotEdge);
canvasses[0].add(dotCanvas);
sliders[0] = new CustomizeAvatarSlider(1, MAX_DIMENSION,
                                        sliderBind(dotEdge), 90);
panels[1].setMaximumSize(new Dimension(100, 150));
panels[1].add(canvasses[0], BorderLayout.NORTH);
panels[1].add(sliders[0], BorderLayout.CENTER);
JPanel newOne = new JPanel(new BorderLayout());
newOne.add(changeColorButton, BorderLayout.NORTH);
newOne.add(buttonPanels[0], BorderLayout.SOUTH);
panels[1].add(newOne, BorderLayout.SOUTH);
validate();


usProvided1Canvas = new CustomizeAvatarCanvas(90, 90, Color.blue);
int img1Width = downScaleWidth(er.getImage1Width(true));
int img1Height = downScaleHeight(er.getImage1Height(true));
Image img1 = ImageCache.getInstance(usProvided1Canvas).
  getImage(OutputterUI.AVATAR_PREFIX + er.getImage1Name(),
          img1Width, img1Height);
usProvided1Canvas.useImage(img1, er.getImage1Name());
canvasses[1].add(usProvided1Canvas);
sliders[1] = new CustomizeAvatarSlider(1, MAX_DIMENSION,
                                       sliderBind(Math.max(img1Width,
                                                           img1Height)),
                                       90);
panels[2].add(canvasses[1], BorderLayout.NORTH);
panels[2].add(sliders[1], BorderLayout.CENTER);
JPanel newOne2 = new JPanel(new BorderLayout());
newOne2.add(new FillerLabel(changeColorButton.getPreferredSize().width,
                            changeColorButton.getPreferredSize().
                            height),
            BorderLayout.NORTH);
newOne2.add(buttonPanels[1], BorderLayout.SOUTH);
panels[2].add(newOne2, BorderLayout.SOUTH);


usProvided2Canvas = new CustomizeAvatarCanvas(90, 90, Color.blue);
int img2Width = downScaleWidth(er.getImage2Width(true));
int img2Height = downScaleHeight(er.getImage2Height(true));
Image img2 = ImageCache.getInstance(usProvided2Canvas).
   getImage(OutputterUI.AVATAR_PREFIX + er.getImage2Name(),
            img2Width, img2Height);
usProvided2Canvas.useImage(img2, er.getImage2Name());
canvasses[2].add(usProvided2Canvas);
sliders[2] = new CustomizeAvatarSlider(1, MAX_DIMENSION,
                                       sliderBind(Math.max(img2Width,
                                                           img2Height)),
                                       90);
panels[3].add(canvasses[2], BorderLayout.NORTH);
panels[3].add(sliders[2], BorderLayout.CENTER);
JPanel newOne3 = new JPanel(new BorderLayout());
newOne3.add(new FillerLabel(changeColorButton.getPreferredSize().width,
                                        changeColorButton.getPreferredSize().
                                        height),
                BorderLayout.NORTH);
newOne3.add(buttonPanels[2], BorderLayout.SOUTH);
panels[3].add(newOne3, BorderLayout.SOUTH);


themProvidedCanvas = new CustomizeAvatarCanvas(90, 90, Color.blue);
uploadYourOwnButton = new JButton("Load a pic");
uploadYourOwnButton.addActionListener(this);
sliders[3] = new CustomizeAvatarSlider(1, MAX_DIMENSION, 1, 90);
if (er.getUserImageName() != null)
  {
        Image img = ImageCache.getInstance(null).
          getImage(OutputterUI.AVATAR_PREFIX + er.getUserImageName(),
                  er.getUserImageWidth(), er.getUserImageHeight());
        themProvidedCanvas.useImage(img, er.getUserImageName());
        canvasses[3].add(themProvidedCanvas);
        panels[4].add(canvasses[3], BorderLayout.NORTH);
        panels[4].add(sliders[3], BorderLayout.CENTER);
        userPictureUploaded = true;
  }
else
  {
        canvasses[3].add(uploadYourOwnButton);
        panels[4].add(canvasses[3], BorderLayout.CENTER);
        buttons[3].setEnabled(false);
        userPictureUploaded = false;
  }
JPanel newOne4 = new JPanel(new BorderLayout());
```

```
newOne4.add(new FillerLabel(changeColorButton.getPreferredSize().width,
                            changeColorButton.getPreferredSize().
                            height),
          BorderLayout.NORTH);
newOne4.add(buttonPanels[3], BorderLayout.SOUTH);
panels[4].add(newOne4, BorderLayout.SOUTH);

showAllOptionsButton = new JButton("Show all");
showAllOptionsButton.addActionListener(this);
if (userPictureUploaded)
  {
       JPanel j1 = new JPanel();
       JPanel j2 = new JPanel();
       j1.add(uploadYourOwnButton);
       j2.add(showAllOptionsButton);
       panels[5].add(j1, BorderLayout.NORTH);
       panels[5].add(j2, BorderLayout.CENTER);
  }
else
  {
       JPanel j1 = new JPanel();
       j1.add(showAllOptionsButton);
       panels[5].add(j1, BorderLayout.CENTER);
  }

for (int i=0;i<4;i++)
  sliders[i].addChangeListener(this);
for (int i=0;i<6;i++)
  add(panels[i]);
```

```
          setBorder(new EmptyBorder(5, 0, 5, 0));
          if (er.usingDefault())
             initButton = 0;
          else if (er.getImageName().equals(er.getImage1Name()))
             initButton = 1;
          else if (er.getImageName().equals(er.getImage2Name()))
             initButton = 2;
          else
             initButton = 3;
          buttons[initButton].setSelected(true);
          changed = false;
}

/**
   Handles button clicks.
 */
public void actionPerformed(ActionEvent e)
{
          if (e.getSource().equals(uploadYourOwnButton))
             {
                    // get the selected image file from the user


                    JFileChooser jfc = new JFileChooser();
                    AllImageFileFilter ff = new AllImageFileFilter("All images");
                    jfc.setFileFilter(ff);
                    if (jfc.showOpenDialog(null) != JFileChooser.APPROVE_OPTION)
                       return;
                    File f = jfc.getSelectedFile();
                    if (f.isDirectory() || !ff.accept(f))
   return;

// copy the file to our "secret" avatars folder

String name = f.getName();
if (name.indexOf('.') < 0)
   return;
String extension = name.substring(name.lastIndexOf('.')).
   toLowerCase();
String avatarName = "";
boolean foundName = false;
for (int i=0;!foundName;i++)
   {
          avatarName = "user"+i+extension;
          File checkItOut = new File(OutputterUI.AVATAR_PREFIX +
                                          avatarName);
          if (!checkItOut.exists())
             foundName = true;
   }
File f2 = new File(OutputterUI.AVATAR_PREFIX + avatarName);
try
   {
          FileInputStream fis = new FileInputStream(f);
          FileOutputStream fos = new FileOutputStream(f2);
          byte buff[] = new byte[4096];
          int bytesRead = 0;
          while ( (bytesRead = fis.read(buff)) >= 0)
             fos.write(buff, 0, bytesRead);
          fis.close();
          fos.close();
   }
catch (IOException ee) { return; }

// update the UI


Image img = ImageCache.getInstance(themProvidedCanvas).
   getImage(OutputterUI.AVATAR_PREFIX + avatarName, 23, -1);
sliders[3].setValue(23);
panels[4].removeAll();
themProvidedCanvas.useImage(img, avatarName);
JPanel canvasPanel = new JPanel();
canvasPanel.add(themProvidedCanvas);
panels[4].add(canvasPanel, BorderLayout.NORTH);
panels[4].add(sliders[3], BorderLayout.CENTER);
JPanel jp = new JPanel(new BorderLayout());
jp.add(new FillerLabel(changeColorButton.getPreferredSize().
                         width,
                         changeColorButton.getPreferredSize().
                         height),
     BorderLayout.NORTH);
jp.add(buttonPanels[3], BorderLayout.SOUTH);
panels[4].add(jp, BorderLayout.SOUTH);
panels[5].removeAll();
JPanel j1 = new JPanel();
JPanel j2 = new JPanel();
j1.add(uploadYourOwnButton);
j2.add(showAllOptionsButton);
panels[5].add(j1, BorderLayout.NORTH);
```

```
                        panels[5].add(j2, BorderLayout.CENTER);
                        buttons[3].setEnabled(true);
                        validate();
                        changed = true;
               }
          else if (e.getSource().equals(showAllOptionsButton))
               JOptionPane.showMessageDialog(null, "NYI");
          else if (e.getSource().equals(changeColorButton))
               {
                        Color c = JColorChooser.showDialog(null, "Choose new color",
                                                                          er.getDotColor());
                        if (c != null)
                           {
                                   changed = true;
                                   dotCanvas.useDot(c, dotCanvas.getDotEdge());
                           }
               }
}

/**
   Returns whether this row has been changed so far.

   @return Whether the row has been changed
 */
public boolean isChanged()
{
          return (changed || !buttons[initButton].isSelected());
}
```

```
/**
  Handles slider movement
 */
public void stateChanged(ChangeEvent e)
{
        changed = true;
        int i=0;
        if (e.getSource().equals(sliders[0]))
           dotCanvas.useDot(dotCanvas.getDotColor(), sliders[0].getValue());
        else if (e.getSource().equals(sliders[1]))
           {
                   Image img = null;
                   if (er.getImage1Width(true) > er.getImage1Height(true))
                      img = ImageCache.getInstance(usProvided1Canvas).
                              getImage(OutputterUI.AVATAR_PREFIX+er.getImage1Name(),
                                          sliders[1].getValue(), -1);
                   else
                      img = ImageCache.getInstance(usProvided1Canvas).
                              getImage(OutputterUI.AVATAR_PREFIX+er.getImage1Name(),
                                          -1, sliders[1].getValue());
                   usProvided1Canvas.useImage(img, er.getImage1Name());
           }
        else if (e.getSource().equals(sliders[2]))
           {
                   Image img = null;
                   if (er.getImage2Width(true) > er.getImage2Height(true))
                      img = ImageCache.getInstance(usProvided2Canvas).
                              getImage(OutputterUI.AVATAR_PREFIX+er.getImage2Name(),
                                          sliders[2].getValue(), -1);
```

```
          else
             img = ImageCache.getInstance(usProvided2Canvas).
                   getImage(OutputterUI.AVATAR_PREFIX+er.getImage2Name(),
                         -1, sliders[2].getValue());
          usProvided2Canvas.useImage(img, er.getImage2Name());
      }
    else if (e.getSource().equals(sliders[3]))
      {
          Image img = null;
          if (themProvidedCanvas.getImage() == null)
             return;
          if (themProvidedCanvas.getImage().getWidth(null) >
             themProvidedCanvas.getImage().getHeight(null))
             img = ImageCache.getInstance(themProvidedCanvas).
                   getImage(OutputterUI.AVATAR_PREFIX +
                         themProvidedCanvas.getImageName(),
                         sliders[3].getValue(), -1);
          else
             img = ImageCache.getInstance(themProvidedCanvas).
                   getImage(OutputterUI.AVATAR_PREFIX +
                         themProvidedCanvas.getImageName(),
                         -1, sliders[3].getValue());
          themProvidedCanvas.useImage(img,
                                themProvidedCanvas.getImageName());
      }
}


/**
  Write the state specified by the user to the EntityRepresentations.
```

```
 */
public void writeChanges()
{
        int scaleDotEdge = scaleWidth(dotCanvas.getDotEdge());
        er.setDotDisplayInformation(dotCanvas.getDotColor(), scaleDotEdge);
        int sImg1Wid = scaleWidth(usProvided1Canvas.getImage().getWidth(null));
        int sImg1Ht = scaleHeight(usProvided1Canvas.getImage().getHeight(null));
        er.setImage1Size(sImg1Wid, sImg1Ht);
        int sImg2Wid = scaleWidth(usProvided2Canvas.getImage().getWidth(null));
        int sImg2Ht = scaleHeight(usProvided2Canvas.getImage().getHeight(null));
        er.setImage2Size(sImg2Wid, sImg2Ht);
        if (themProvidedCanvas.getImage() != null)
           {
                   int nw =
                      scaleWidth(themProvidedCanvas.getImage().getWidth(null));
                   int nh =
                      scaleHeight(themProvidedCanvas.getImage().getHeight(null));
                   Image scl = ImageCache.getInstance(null).
                      getImage(OutputterUI.AVATAR_PREFIX +
                                  themProvidedCanvas.getImageName(), nw, nh);
                   er.setUserImage(scl, themProvidedCanvas.getImageName());
           }
        if (buttons[0].isSelected())
           er.useDefault();
        else if (buttons[1].isSelected())
           er.useImage1();
        else if (buttons[2].isSelected())
           er.useImage2();
        else if (buttons[3].isSelected())
```

```
            er.useUserImage();
    }

    private int downScaleHeight(int pixels)
    {
        return (int)(pixels / oui.getHeightRatio());
    }

    private int downScaleWidth(int pixels)
    {
        return (int)(pixels / oui.getWidthRatio());
    }

    private int scaleHeight(int pixels)
    {
        return (int)(pixels * oui.getHeightRatio());
    }

    private int scaleWidth(int pixels)
    {
        return (int)(pixels * oui.getWidthRatio());
    }

    private int sliderBind(int numPixels)
    {
        if (numPixels < 1)
            return 1;
        if (numPixels > MAX_DIMENSION)
            return MAX_DIMENSION;
            return numPixels;
    }
}
```

```
package hope;


import javax.swing.JSlider;

import java.awt.Dimension;


/**

   The CustomizeAvatarSlider class provides a JSlider that has an explicit

   width. It was created due to layout issues in the customization dialog.

 */

public class CustomizeAvatarSlider extends JSlider

{

    private int pixelWidth;


    /**

       This initializes a JSlider with a specified width.


       @param minValue   The minimum tick value for the JSlider

       @param maxValue   The maximum tick value for the JSlider

       @param initValue  The initial tick value for the JSlider

       @param pixelWidth The explicit width of the JSlider in pixels

     */

    public CustomizeAvatarSlider(int minValue, int maxValue, int initValue,

                                                    int pixelWidth)

    {

             super(minValue, maxValue, initValue);

             this.pixelWidth = pixelWidth;

    }


    /**
```

```
       Override method to explicitly set the width.
     */
    public Dimension getPreferredSize()
    {
            Dimension dim = super.getPreferredSize();
            return new Dimension(pixelWidth, dim.height);
    }
}
```

```
package hope;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import java.util.List;
import java.util.ArrayList;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Container;
import java.awt.GridLayout;

/**
   The CustomizeMetricsFrame class displays a prompt to choose which metrics
   are displayed. It notifies the MetricPanel of changes.
 */
public class CustomizeMetricsFrame extends JFrame implements ActionListener
{
    private List<MetricLabel> metrics;
    private List<JCheckBox> boxes;
    private JButton okButton;
    private JButton cancelButton;
    private MetricPanel mp;

    /**
       This initializes the dialog, laying it out.

       @param mp The MetricPanel associated with this dialog
 */
public CustomizeMetricsFrame(MetricPanel mp)
{
        super("Which metrics should be displayed?");
        this.mp = mp;
        metrics = mp.getMetrics();
        Container c = getContentPane();
    c.setLayout(new GridLayout(metrics.size()+1, 1));
        boxes = new ArrayList<JCheckBox>();
        for (MetricLabel ml : metrics)
          {
                JCheckBox newOne = new JCheckBox(ml.getName());
                newOne.setSelected(ml.isEnabled());
                c.add(newOne);
                boxes.add(newOne);
          }
        JPanel bottom = new JPanel(new GridLayout(1,2));
        JPanel useless1 = new JPanel();
        JPanel useless2 = new JPanel();
        okButton = new JButton("OK");
        okButton.addActionListener(this);
        useless1.add(okButton);
        cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(this);
        useless2.add(cancelButton);
        bottom.add(useless1);
        bottom.add(useless2);
        c.add(bottom);
        setSize(400, 50*metrics.size()+50);
        setVisible(true);
    }


    /**
       Handle clicks on the buttons appropriately.
     */
    public void actionPerformed(ActionEvent e)
    {
            if (e.getActionCommand().equals(okButton.getText()))
                {
                        for (int i=0;i<metrics.size();i++)
                            metrics.get(i).setEnabled(boxes.get(i).isSelected());
                        mp.notifyDisplayChange();
                }
            setVisible(false);
            dispose();
    }
}
```

```
package hope;


import java.io.File;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.FileReader;
import java.io.FileOutputStream;
import java.io.IOException;


/**
   The DisplayControl class contains all the display controls of the simulation
   display. This is things like the speed of the simulation.
 */
public class DisplayControl
{
    private double speedMultiplier;
    private int refreshRateMs;
    private static final double DEFAULT_SPEED_MULTIPLIER = 100;
    private static final int DEFAULT_REFRESH_RATE_MS = 10;
```

```
/**
  This initializes the DisplayControl. It reads the initial values from a
  configuration file.

  @param configFile The file from which all display control values are read
 */
public DisplayControl(String configFile)
{
        speedMultiplier = DEFAULT_SPEED_MULTIPLIER;
        refreshRateMs = DEFAULT_REFRESH_RATE_MS;
        File f = new File(configFile);
        if (!f.exists())
            return; // we just use the default values in this case
        try {
            BufferedReader brconfig =
                    new BufferedReader(new FileReader(configFile));
            String line;
            while ((line = brconfig.readLine()) != null) {
                    if (line.startsWith("TIME_MULT"))

                        try {

                                speedMultiplier =

                                    Double.parseDouble(line.substring(10));

                        }

                        catch (Exception e) {}

            }

        }

        catch (IOException e) {}

}



/**

   The getter for the number of milliseconds between screen refreshes.



   @return The output refresh rate in milliseconds

 */

public int getRefreshRateMs()

{

        return refreshRateMs;
```

```
}



/**

   The getter for the speed multiplier of the simulation display.



   @return The speed multiplier of the simulation display

 */

public double getSpeedMultiplier()

{

          return speedMultiplier;

}



/**

   This prints all display control values to a specified configuration file.



   @param configFile The configuration file to output to

 */

public void printToFile(String configFile)

{
```

```
          try {

              PrintWriter pwconfig = new PrintWriter(new FileOutputStream(configFile));

              pwconfig.println("TIME_MULT "+speedMultiplier);

              pwconfig.flush();

              pwconfig.close();

          } catch (IOException e) {

          }



}



/**

   The setter for the number of milliseconds between screen refreshes.



   @param refreshRateMs The new output refresh rate in milliseconds

 */

public void setRefreshRateMs(int refreshRateMs)

{

          this.refreshRateMs = refreshRateMs;

}
```

```
    /**
       The setter for the speed multiplier of the simulation display

       @param speedMultiplier The new speed multiplier of the simulation display
     */
    public void setSpeedMultiplier(double speedMultiplier)
    {
            this.speedMultiplier = speedMultiplier;
    }
}
```

```
package hope;


import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.BorderLayout;
import java.awt.GridLayout;


/**
   The DisplayControlPanel class is the JPanel showing all the display controls.
   These include things like the speed of the simulation model relative to
   real time.
 */
public class DisplayControlPanel extends JPanel implements ChangeListener,
                                                        ActionListener
{
    private JSlider speedSlider;

    private JTextField speedField;

    private DisplayControl dc;

    private OutputterUI oui;

    private String newFieldText;



    /**

       This initializes the JPanel.



       @param dc  The DisplayControl being represented by this panel

       @param oui The OutputterUI on which this panel will be placed

     */

    public DisplayControlPanel(DisplayControl dc, OutputterUI oui)

    {

            super(new BorderLayout());

            this.dc = dc;

            this.oui = oui;
```

```
        add(new JLabel("Simulation speed:"), BorderLayout.WEST);
        double initSpeed = dc.getSpeedMultiplier();
        int tick = convertSpeedToTick(initSpeed);
        speedSlider = new JSlider(0, 100, tick);
        speedSlider.addChangeListener(this);
        add(speedSlider, BorderLayout.CENTER);
        speedField = new JTextField(convertSpeedToString(initSpeed));
        speedField.setHorizontalAlignment(JTextField.RIGHT);
        speedField.addActionListener(this);
        JPanel multPanel = new JPanel(new BorderLayout());
        multPanel.add(speedField, BorderLayout.WEST);
        multPanel.add(new JLabel("x"), BorderLayout.EAST);
        add(multPanel, BorderLayout.EAST);
        newFieldText = null;
    }


    /**
      This handles the JTextField into which new speed multipliers can be
      typed.
```

```
 */

public void actionPerformed(ActionEvent e)

{

        if (e.getSource().equals(speedField))

           {

                    double newSpeed = convertStringToSpeed(speedField.getText());

                    speedSlider.setValue(convertSpeedToTick(newSpeed));

                    dc.setSpeedMultiplier(newSpeed);

           }

}



/**

  This handles the JSlider being slid.

 */

public void stateChanged(ChangeEvent e)

{

        if (e.getSource().equals(speedSlider))

           {

                    double newSpeed = convertTickToSpeed(speedSlider.getValue());
```

```
                        String speedString = convertSpeedToString(newSpeed);

                        newFieldText = speedString;

                        dc.setSpeedMultiplier(newSpeed);

                        oui.validate();

                }

}



/**

   This updates the UI at each time slice of the simulation display. In

   particular, it serves to throttle the updates of the JTextField with the

   speed multipliers to prevent freeze-up.

 */

public void updateAtTimeSlice()

{

          if (newFieldText != null)

                {

                        speedField.setText(newFieldText);

                        newFieldText = null;

                }
```

```
}

private String convertSpeedToString(double speed)
{
          if (Math.abs(speed) < 10)
              return ""+((double)((int)speed*100))/100.0;
          else if (Math.abs(speed) < 100)
              return ""+((double)((int)speed*10))/10.0;
          else
              return ""+(int)speed;
}

private int convertSpeedToTick(double speed)
{
          if (speed <= 1)
              {
                        return 0;
              }
          else if (speed >= 10000)
```

```
      return 100;
    else
      {
          double exactTick = 25 * Math.log(speed)/Math.log(10);
          if (exactTick - (int)exactTick < .5)
            return (int)exactTick;
          else
            return ((int)exactTick + 1);
      }
}

private double convertStringToSpeed(String speed)
{
    double parsed = 1;
    try
      {
          parsed = Double.parseDouble(speed);
      }
    catch (Exception e) {} // we are fault tolerant; 1 is the default
```

```
        return parsed;
    }

    private double convertTickToSpeed(int tick)
    {
        return Math.pow(10, ((double)tick)/25.0);
    }
}
```

```
package hope;



public class Doctor extends Entity

{



        public Doctor(int id, String name, String description,

                 int location, OutputterUI oui, double entryTime) {

                        super(id, name, description, location, oui, entryTime);

         }



}
```

```
package hope;


import java.awt.Color;


/**
   The DoctorRepresentation class represents any doctor in the simulation
   display.
 */
public class DoctorRepresentation extends EntityRepresentation
{
    /**
       This sets up the default doctor display (6x6 pink dots).
     */
    public DoctorRepresentation()
    {
            super("Doctor", Color.pink, 12, "evil.png", 23, -1,
                  "evil2.png", 22, -1, true, null);
    }
    /**

       This represents any Entity that is a doctor.

     */

    public boolean represents(Entity e)

    {

            return (e instanceof Doctor);

    }

}
```

```
package hope;

/**
   The Entity class stores entities in the hospital (patients, nurses, and
   doctors) and associated information.
 */
public class Entity {
    private OutputterUI oui;
    private EntityRepresentation er;
    private int id;
    private String name;
    private String description;
    private int location;

    private EntityDisplayInfo oldEDI;
    private CanvasRegion crClaimed;

    private boolean moving;
    private int fromLocation;
private double moveStartTime;
private double moveDuration;
private int[] cachedPath;
private double[] cachedTimes;
private double finalTime;
private double entryTime;


/**
  Create an Entity given initial information. This function claims the
  Entity's location on the map and loads its associated image, as well.
  It is called by the {@link AddAction} class.


  @param id          The unique ID of this entity in the OutputterUI
  @param name        The name of the Entity
  @param description The initial description of the Entity
  @param location    The location ID of the initial starting location of the Entity
  @param oui         The OutputterUI this Entity is in
  @param entryTime   The simulation time when the Entity entered the ED
 */
```

```
public Entity(int id, String name, String description,
                    int location, OutputterUI oui, double entryTime) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.location = location;
        this.oui = oui;
        this.entryTime = entryTime;
        er = oui.getAssociatedEntityRepresentation(this);
        oldEDI = new EntityDisplayInfo(er, oui);
        crClaimed = oui.claimRegion(oui.getLocationPixel(location),
                                    oldEDI.getEntityRepresentation().getWidth()
                                    + 2,
                                    oldEDI.getEntityRepresentation().getHeight()
                                    + 2);
}


/**
  This generates the display information for the Entity. It is called by
  the {@link OutputterCanvas} to figure out what to display. If the

  Entity is moving, it figures out where the entity currently is based on

  the current simulation time, which is passed as a parameter. This

  function has the side effect of triggering the Entity's observer if the

  patient has moved. More severely, it has the side effect of removing

  data about old positions and adding in new ones; hence it should only

  be called by the OutputterCanvas.



  @param time  The current simulation time

  @return      The display information for the Entity

 */

public EntityDisplayInfo generateEDI(double time) {

      if (moving && time >= moveStartTime + moveDuration)

        moving = false;

      if (moving) {

        int i;

        for (i = 0; i < cachedTimes.length; i++)

              if (cachedTimes[i] > time)

                break;
```

```
    if (i == 0)
            SimulationOutputter.die("Move added prematurely");
    else if (i == cachedTimes.length)
            SimulationOutputter.die("cachedTimes messed up");
    double prop = (time - cachedTimes[i - 1])
            / (cachedTimes[i] - cachedTimes[i - 1]);
    Pixel loc1 = oui.getLocationPixel(cachedPath[i - 1]);
    Pixel loc2;
    if (i != cachedTimes.length-1)
            loc2 = oui.getLocationPixel(cachedPath[i]);
    else
            loc2 = crClaimed.getMiddle();
    int x = (int) (prop * loc2.getX() + (1 - prop) * loc1.getX());
    int y = (int) (prop * loc2.getY() + (1 - prop) * loc1.getY());
    oldEDI = new EntityDisplayInfo(x, y, oldEDI);
    return oldEDI;
}
else {
    Pixel base = crClaimed.getMiddle();
```

```
            oldEDI = new EntityDisplayInfo(base.getX(),base.getY(),oldEDI);
            return oldEDI;
        }
}


/**
  This is the getter for the Entity's description


  @return The Entity's description
 */
public String getDescription() {
        return description;
}


/**
  This is the getter for the time when the Entity entered the ED.


  @return The time (in simulation seconds)
 */
```

```
public double getEntryTime()
{
        return entryTime;
}


/**
   Returns the time when last this patient arrived at a location.


   @return The time (in simulation seconds)
 */
public double getFinalTime()
{
        return finalTime;
}


/**
   This is a getter for the patient's unique ID.


   @return The unique ID for the Entity's OutputterUI.
 */
public int getId() {
        return id;
}


/**
   This is the getter for the location ID of the Entity's location. If
   the Entity is moving, it returns the destination location's ID.

   @return The Entity's location (as a location ID)
 */
public int getLocation() {
        return location;
}



/**
        This is the getter for the Entity's name.
```

```
          @return The name of the Entity

 */

public String getName() {

          return name;

}



/**

   This gets the pixel at which the Entity is currently located.



   @return The current location of the Entity

 */

public Pixel getPixelCoordinate()

{

          return oldEDI.getCurrPixel();

}



/**

   Returns the rectangle corresponding to the Entity's current display.
```

```
  @return The rectangle
 */
public CanvasRegion getCurrentEntityRegion(){
        return new CanvasRegion(getPixelCoordinate(),
                                    er.getWidth(),
                                    er.getHeight());
}


/**
  This is the getter for the Entity's type. The type is like
  OutputterUI.ID_PATIENT


  @return The Entity's type


public int getType() {
        return type;
}
        */
```

```
/**
  This handles the Entity leaving the ED. It unclaims all regions of the
  map currently claimed by the Entity so the UI doesn't leak claimed
  regions.
 */



public void leave()
{
        if (crClaimed != null)
           oui.unclaimRegion(crClaimed);
        crClaimed = null;
}


/**
  This moves this Entity to a new location. This is called by
  {@link MoveAction}. The Entity's location will be updated until the
  move is complete, routing the Entity at even visual speed through the
  shortest path to its destination. The position at the final destination
```

```
   is claimed and at the current location is unclaimed.



   @param newLoc    The location ID of the final destination for the Entity

   @param time      The simulation time when this move started

   @param moveTime  This is the number of simulation seconds the move will take

 */

public void move(int newLoc, double time, double moveTime) {

        if (crClaimed != null)

            oui.unclaimRegion(crClaimed);

        crClaimed = oui.claimRegion(oui.getLocationPixel(newLoc),

                                    oldEDI.getEntityRepresentation().getWidth() + 2,

                                    oldEDI.getEntityRepresentation().getHeight() + 2);

        moving = true;

        fromLocation = location;

        location = newLoc;

        moveStartTime = time;

        moveDuration = moveTime;

        finalTime = time + moveTime;

        Path p = oui.getGraph().getShortestPath(fromLocation, location);
```

```
          cachedPath = p.getNodes();

          cachedTimes = new double[cachedPath.length];

          double length = p.getLength();

          double soFar = 0;

          for (int i = 0; i < cachedPath.length; i++) {

              cachedTimes[i] = time + (soFar / length) * moveTime;

              if (i != cachedPath.length - 1)

                      soFar += oui.getGraph().getEdge(cachedPath[i],

                                                                cachedPath[i + 1]);

          }

}

/**

   This sets the description of this entity. It is called by

   {@link ChangeDescriptionAction}.
```

```
      @param newDescription The new description of this entity
     */
    public void setDescription(String newDescription) {
            description = newDescription;
    }
}
```

```
package hope;


import java.awt.Image;


/**
  The EntityDisplayInfo stores all the information needed for the
  OutputterCanvas to draw an Entity. In particular, it stores information
  about the old display of the Entity as well as the current display to
  allow fast display of Entities.
 */
public class EntityDisplayInfo
{
   private Pixel currPixel;
   private boolean moved;
   private Pixel oldPixel;
   private int oldWidth;
   private int oldHeight;
   private int width;
   private int height;
   private OutputterUI oui;
   private EntityRepresentation er;


   /**
      This initializes an Entity's EntityDisplayInfo for the first time. Any
      time after, the new EntityDisplayInfo is bootstrapped from the old one
      with the other constructor.


      @param er  The EntityRepresentation for the associated Entity
      @param oui The OutputterUI for the associated Entity
 */
public EntityDisplayInfo(EntityRepresentation er, OutputterUI oui)
{
        this.oui = oui;
        currPixel = new Pixel(-1, -1);
        oldPixel = null;
        moved = false;
        this.er = er;
        oldWidth = width = er.getWidth();
        oldHeight = height = er.getHeight();
}

/**
   This creates an EntityDisplayInfo based off the an old one. It uses the
   old one's "current" values as this one's "old" values.

   @param x      The x-coordinate (in pixels) of the associated entity
   @param y      The y-coordinate (in pixels) of the associated entity
   @param oldOne The old EntityDisplayInfo of the associated Entity
 */
public EntityDisplayInfo(int x, int y, EntityDisplayInfo oldOne)
{
        this.oui = oldOne.oui;
        this.er = oldOne.getEntityRepresentation();
        currPixel = new Pixel(x, y);
        oldPixel = oldOne.getCurrPixel();
        moved = (!currPixel.equals(oldOne.getCurrPixel()));
        oldWidth = oldOne.width;
        oldHeight = oldOne.height;
        width = er.getWidth();
        height = er.getHeight();
}

/**
   This is the getter for the current pixel of the location of the
   associated Entity.

   @return The associated Entity's location
 */
public Pixel getCurrPixel()
{
        return currPixel;
}

/**
   This is the getter for the EntityRepresentation of the associated Entity.

   @return The EntityRepresentation
 */
public EntityRepresentation getEntityRepresentation()
{
        return er;
}

/**
   This says if the Entity has moved between its last displayed location
   and its current location.
   @return Whether or not the associated Entity moves since it was last displayed
 */
public boolean getMoved()
{
        return moved;
}

/**
   This is the getter for the old height of the associated Entity's display

   @return The old height of the Entity's display
 */
public int getOldHeight()
{
        return oldHeight;
}

/**
   This is the getter for the old width of the associated Entity's display

   @return The old width of the Entity's display
 */
public int getOldWidth()
{
        return oldWidth;
}

/**
   This is the getter for the last displayed pixel of the location of the
```

```
     associated Entity.


     @return The associated Entity's last displayed location
    */
   public Pixel getOldPixel()
   {
           return oldPixel;
   }
}
```

```
package hope;



import java.awt.Color;

import java.awt.Image;



/**

   The EntityRepresentation has the info necessary to draw an Entity. However,

   it has no information about where to display it or how. It stores things like

   the image associated with the Entity or the color of the dot representing

   the Entity.

 */

public abstract class EntityRepresentation

{

    private String name;

    private Image img;

    private String imageName;

    private Color dotColor;

    private int dotEdge;

    private boolean enabled;
private EntityRepresentation override;
private boolean useDefault;
private String image1Name;
private int image1Width;
private int image1Height;
private String image2Name;
private int image2Width;
private int image2Height;
private String userImageName;
private int userImageWidth;
private int userImageHeight;

/**
  This initializes the entity representation. It makes a "factory default"
  that will be the same no matter what customization the user does to the
  Entity. Customization is performed in the other class routines. The
  override functionality means this ER overrides a more general ER. Think
  of a representation of a high-priority patient overriding the
  representation of a patient. The override is only in effect if this
```

```
   EntityRepresentation is enabled. The images are used in the customize
   avatars dialog as the developer-provided images for each entity. The
   simulation will exit if null values are passed for image names.

   @param name             The name of this representation ("Patient" for patient representation)
   @param defaultColor     The color of the default representation, a dot
   @param defaultDotEdge   The width and height (in pixels) of the default representation, a dot
   @param image1Name       The name of the first provided image option
   @param image1Width      The width in pixels of the first provided image option
   @param image1Height     The height in pixels of the first provided image option
   @param image2Name       The name of the second provided image option
   @param image2Width      The width in pixels of the second provided image option
   @param image2Height     The height in pixels of the second provided image option
   @param enabled          Whether this representation is enabled by default
   @param override         The (more general) EntityRepresentation this ER (optionally) overrides.
 */
public EntityRepresentation(String name, Color defaultColor,
                            int defaultDotEdge,
                            String image1Name, int image1Width,
                    int image1Height, String image2Name,

                    int image2Width, int image2Height,

                    boolean enabled, EntityRepresentation override)

{

    this.name = name;

    this.enabled = enabled;

    this.override = override;

    this.image1Name = image1Name;

    this.image1Width = image1Width;

    this.image1Height = image1Height;

    this.image2Name = image2Name;

    this.image2Width = image2Width;

    this.image2Height = image2Height;

    userImageName = null;

    userImageWidth = userImageHeight = -1;

    if (image1Name == null || image2Name == null)

      SimulationOutputter.

          die("Invalid initialization of EntityRepresentation");

    img = null;
```

```
        dotColor = defaultColor;

        dotEdge = defaultDotEdge;

        useDefault = true;

}



/**

   This returns the most recent user-defined color for the Entity's dot

   representation.



   @return The dot color

 */

public Color getDotColor()

{

        return dotColor;

}



/**

   This gets the width and height of the default display (a colored dot) in pixels.
   @return The dot width and height in pixels

 */

public int getDotEdge()

{

          return dotEdge;

}



/**

   This returns the height of the selected display in pixels.



   @return The height in pixels

 */

public int getHeight()

{

          if (useDefault)

              return dotEdge;

          else

              return img.getHeight(null);

}
```

```
/**
   This returns the Image associated with this EntityRepresentation. It
   returns null if no Image is associated.

   @return The associated image
 */
public Image getImage()
{
        return img;
}


/**
   This returns the name of the image associated with the
   EntityRepresentation. It returns null if no Image is associated. The
   name is based in the avatars folder.

   @return The image name
 */
```

```
    public String getImageName()

    {

            return imageName;

    }




    /**

       Returns the height of the first developer-provided image for Entities

       represented by this ER.



       @param expandNegativePixelValue Whether to expand a negative pixel value (which stands for scaling
on the width)

       @return                        The height in pixels

     */

    public int getImage1Height(boolean expandNegativePixelValue)

    {

            if (image1Height < 0 && expandNegativePixelValue)

                return ImageCache.getInstance(null).

                        getImage(OutputterUI.AVATAR_PREFIX + image1Name, image1Width,

                                image1Height).getHeight(null);
```

```
          else

              return image1Height;

    }




    /**

       Returns the height of the second developer-provided image for Entities

       represented by this ER.



       @param expandNegativePixelValue Whether to expand a negative pixel value (which stands for scaling
 on the width)

       @return The height in pixels

     */

    public int getImage2Height(boolean expandNegativePixelValue)

    {

          if (image2Height < 0 && expandNegativePixelValue)

              return ImageCache.getInstance(null).

                        getImage(OutputterUI.AVATAR_PREFIX + image2Name, image2Width,

                                image2Height).getHeight(null);

          return image2Height;
```

```
}

/**
   Returns the name of the first developer-provided image for Entities
   represented by this ER. The name is based in the avatars folder.

   @return The first image's file name
 */
public String getImage1Name()
{
          return image1Name;
}

/**
   Returns the name of the second developer-provided image for Entities
   represented by this ER. The name is based in the avatars folder.

   @return The second image's file name
 */
```

```
    public String getImage2Name()

    {

            return image2Name;

    }




    /**

       Returns the width of the first developer-provided image for Entities

       represented by this ER.




       @param expandNegativePixelValue Whether to expand a negative pixel value (which stands for scaling
on the height)

       @return The width in pixels

     */

    public int getImage1Width(boolean expandNegativePixelValue)

    {

            if (image1Width < 0 && expandNegativePixelValue)

                return ImageCache.getInstance(null).

                        getImage(OutputterUI.AVATAR_PREFIX + image1Name, image1Width,

                                image1Height).getWidth(null);
            else
               return image1Width;
    }


    /**
      Returns the width of the second developer-provided image for Entities
      represented by this ER.

      @param expandNegativePixelValue Whether to expand a negative pixel value (which stands for scaling on the height)
      @return The width in pixels
     */
    public int getImage2Width(boolean expandNegativePixelValue)
    {
            if (image2Width < 0 && expandNegativePixelValue)
               return ImageCache.getInstance(null).
                      getImage(OutputterUI.AVATAR_PREFIX + image2Name, image2Width,
                                 image2Height).getWidth(null);
            else
```

```
		return image1Width;
	}



	/**
	   This returns the name of this EntityRepresentation. Examples would be
	   "Patient" or "Priority 1 Patient".


	   @return The representation's name
	 */
	public String getName()
	{
		return name;
	}



	/**
	   This returns the EntityRepresentation that this ER overrides. It returns
	   null if no representation is overridden.


	   @return The representation this ER overrides
 */

public EntityRepresentation getOverride()

{

          return override;

}




/**

   Returns the height of the user uploaded image.



   @return The height in pixels (-1 if no image uploaded)

 */

public int getUserImageHeight()

{

          return userImageHeight;

}




/**

   Returns the name of the user uploaded image.
   @return The name of the image file (in the avatars folder; null if no image uploaded)

 */

public String getUserImageName()

{

          return userImageName;

}



/**

   Returns the width of the user uploaded image.



   @return The width in pixels (-1 if no image uploaded)

 */

public int getUserImageWidth()

{

          return userImageWidth;

}



/**

   This returns the width of the selected display in pixels.
   @return The width in pixels
 */
public int getWidth()
{
          if (useDefault)
              return dotEdge;
          else
              return img.getWidth(null);
}


/**
   This returns whether this entity representation is enabled. If it is not
   enabled, it will not be given to any entities.


   @return Whether this representation is enabled
 */
public boolean isEnabled()
{
        return enabled;

}



/**

   This method says whether this EntityRepresentation can represent the

   passed-in Entity. It must be overridden by any class that extends the

   EntityRepresentation.



   @param e The Entity to test for matching this ER

   @return  Whether the Entity matched

 */

public abstract boolean represents(Entity e);



/**

   This sets the color and size of the default dots displayed instead of

   avatars. It does not caused this EntityRepresentation to start using the

   dots. The function useDefault is used for that.



   @param dotColor The color of the default display dots
   @param dotEdge  The width and height in pixels of the default display dots

   @see            hope.EntityRepresentation#useDefault useDefault

 */

public void setDotDisplayInformation(Color dotColor, int dotEdge)

{

          this.dotColor = dotColor;

          this.dotEdge = dotEdge;

}



/**

   This sets whether this EntityRepresentation is enabled. If it is not

   enabled, no entities will be assigned this representation.



   @param enabled Whether this ER is enabled

 */

public void setEnabled(boolean enabled)

{

          this.enabled = enabled;

}
```

```
/**
   This sets the size of the factory default image 1. This will be used in
   later customization dialogs and will become the new default size.

   @param width  The new width (in pixels)
   @param height The new height (in pixels)
 */
public void setImage1Size(int width, int height)
{
        image1Width = width;
        image1Height = height;
}


/**
   This sets the size of the factory default image 2. This will be used in
   later customization dialogs and will become the new default size.

   @param width  The new width (in pixels)
   @param height The new height (in pixels)

 */

public void setImage2Size(int width, int height)

{

          image2Width = width;

          image2Height = height;

}



/**

   This sets the user image associated with this EntityRepresentation. This

   will be used if useUserImage() is called.



   @param userImage The image

   @param name      The name of the image's file (in the avatars folder)

 */

public void setUserImage(Image userImage, String name)

{

          userImageName = name;

          userImageWidth = userImage.getWidth(null);
```

```
		userImageHeight = userImage.getHeight(null);
	}

	/**
	   This sets the EntityRepresentation to use the default display, which is
	   colored dots.
	 */
	public void useDefault()
	{
		useDefault = true;
		img = null;
		imageName = null;
	}

	/**
	   This sets the EntityRepresentation to use the factory default image 1.
	 */
	public void useImage1()
	{
```

```
          img = ImageCache.getInstance(null).getImage(OutputterUI.AVATAR_PREFIX +
                                                   image1Name, image1Width,
                                                   image1Height);
          imageName = image1Name;
          useDefault = false;
}


/**
   This sets the EntityRepresentation to use the factory default image 2.
 */
public void useImage2()
{
          img = ImageCache.getInstance(null).getImage(OutputterUI.AVATAR_PREFIX +
                                                   image2Name, image2Width,
                                                   image2Height);
          imageName = image2Name;
          useDefault = false;
}
```

```
/**
   This sets the EntityRepresentation to use the user-provided image. It
   exits the simulation if no user image has previously been set.
 */
public void useUserImage()
{
        if (userImageName == null)
            SimulationOutputter.die("No user image in useUserImage");
        img = ImageCache.getInstance(null).getImage(OutputterUI.AVATAR_PREFIX +
                                                                  userImageName,
                                                                  userImageWidth,
                                                                  userImageHeight);
        imageName = userImageName;
        useDefault = false;
}


/**
   This returns whether the EntityRepresentation is using the default
```

```
	 display, colored dots. If not, avatar images are being used.



	 @return Whether the default display (colored dots) is being used

	 */

	public boolean usingDefault()

	{

		return useDefault;

	}

}
```

```
package hope;


import javax.swing.JLabel;

import java.awt.Dimension;


/**

   The FillerLabel class provides an explicitly sized empty JLabel for display

   layout purposes.

 */

public class FillerLabel extends JLabel

{

    private int width, height;


    /**

       This initializes the FillerLabel to a particular size.


       @param width  The width of the label (in pixels)

       @param height The height of the label (in pixels)

     */
    public FillerLabel(int width, int height)

    {

              super("");

              this.width = width;

              this.height = height;

    }



    /**

       This actually enforces the custom size of the label.

     */

    public Dimension getPreferredSize()

    {

              return new Dimension(width, height);

    }

}
```

```
package hope;


import java.util.ArrayList;
import java.util.HashMap;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;
import java.util.NoSuchElementException;
import java.util.Set;


/**
   The Graph object represents a graph with nodes as locations in the ED. It
   reads in the graph from a file and provides shortest path calculations.
 */
public class Graph
{
    HashMap<Integer, Node> nodeLookup;
```

```
/**

   This creates a graph from a file. It exits the simulation if the

   file is malformatted.



   @param fileName The name of the file

 */

public Graph(String fileName)

{

          nodeLookup = new HashMap<Integer, Node>();

          try

              {

                        BufferedReader br =

                            new BufferedReader(new FileReader(fileName));

                        String line = null;

                        while ((line = br.readLine()) != null)

                            {

                                      StringTokenizer st = new StringTokenizer(line);

                                      String type = st.nextToken();

                                      if (type.equals("N"))
          {
                int number = Integer.parseInt(st.nextToken());
                String desc = "";
                while (st.hasMoreElements())
                  desc = desc + st.nextToken() + " ";
                desc = desc.trim();
                addNode(number, desc);
          }
        else if (!type.equals("E"))
          SimulationOutputter.die("Bad line: "+line);
  }
br.close();
br = new BufferedReader(new FileReader(fileName));
line = null;
while ((line = br.readLine()) != null)
  {
        StringTokenizer st = new StringTokenizer(line);
        String type = st.nextToken();
        if (type.equals("E"))
```

```
                   {
                         int i1 = Integer.parseInt(st.nextToken());
                         int i2 = Integer.parseInt(st.nextToken());
                         double i3 = Double.parseDouble(st.nextToken());
                         addEdge(i1, i2, i3);
                   }
          }
  }
catch (IOException e)
  {
        SimulationOutputter.die("Could not open input file "+fileName);
  }
catch (NoSuchElementException e)
  {
        SimulationOutputter.die("Error in file being parsed: " +
                                 fileName);
  }
catch (NumberFormatException e)
  {
```

```
              SimulationOutputter.die("Malformed line in file: "+fileName);
        }
}


/**
  This returns the description of a node in the Graph. It exits the
  simulation if the node ID passed is invalid.

  @param number The node ID of the node
  @return       The description of the node
 */
public String getDescription(int number)
{
      if (!nodeLookup.containsKey(new Integer(number)))
        SimulationOutputter.die("Node "+number+" isn't there for "+
                                 "getDescription.");
      return nodeLookup.get(new Integer(number)).getDescription();
}
```

```
/**

   This returns the weight of an edge given the endpoints. It exits the

   simulation if either of the nodes passed are not valid node IDs.



   @param n1 The node ID of one endpoint of the edge

   @param n2 The node ID of the other endpoint of the edge

   @return   The length of the edge (or -1 if it does not exist)

 */

public double getEdge(int n1, int n2)

{

          if (!nodeLookup.containsKey(new Integer(n1)) ||

              !nodeLookup.containsKey(new Integer(n2)))

              SimulationOutputter.die("Either don't have "+n1+" or don't have "+

                                                  n2+" in getEdge.");

          return nodeLookup.get(new Integer(n1)).getEdge(n2);

}



/**

   This returns an array with the node IDs of all the nodes in the Graph.
   @return An array of the node IDs of all the nodes in the Graph
  */
 public int[] getNodes()
 {
         Set<Integer> s = nodeLookup.keySet();
         int[] ret = new int[s.size()];
         int pos = 0;
         for (Integer q : s)
             ret[pos++] = q.intValue();
         return ret;
 }

 /**
   This returns the shortest path between two nodes in the Graph. It exits
   the simulation if either of the node IDs in invalid.

   @param n1 The node ID of one endpoint of the path
   @param n2 The node ID of the other endpoint of the path
   @return   The shortest path between the nodes
  */
 public Path getShortestPath(int n1, int n2)
 {
         if (!nodeLookup.containsKey(new Integer(n1)) ||
             !nodeLookup.containsKey(new Integer(n2)))
             SimulationOutputter.die("Node "+n1+" or node "+n2+" doesn't exist "+
                                         "in getShortestPath");
         if (n1 == n2)
             {
                 Path p = new Path(this);
                 p.addAtBeginning(n1);
                 return p;
             }
         else
             return nodeLookup.get(new Integer(n1)).getShortestPath(n2);
 }


 private void addNode(int number)
```

```
{
        addNode(number, "");
}

private void addNode(int number, String description)
{
        Node n = new Node(number, description, this);
        nodeLookup.put(new Integer(number), n);
}


private void addEdge(int n1, int n2, double weight)
{
        if (!nodeLookup.containsKey(new Integer(n1)) ||
           !nodeLookup.containsKey(new Integer(n2)))
           SimulationOutputter.die("Either don't have "+n1+
                                            " or don't have "+n2+" in addEdge.");
        Node a = nodeLookup.get(new Integer(n1));
        Node b = nodeLookup.get(new Integer(n2));
        a.addEdge(n2, weight);
```

```
          b.addEdge(n1, weight);
}

private HashMap<Integer, Double> getEdges(int number)
{
          if (!nodeLookup.containsKey(new Integer(number)))
              SimulationOutputter.die("Node "+number+" not there in getEdges");
          return nodeLookup.get(new Integer(number)).getEdges();
}

private class Node
{
          private final int INT_MAX = 10000000;
          private int number;
          private String description;
          private HashMap<Integer, Double> edges;
          private HashMap<Integer, Path> shortestPaths;
          private Graph g;
          public Node(int number, String description, Graph g)
```

```
{
    this.description = description;
    this.number = number;
    edges = new HashMap<Integer, Double>();
    shortestPaths = null;
    this.g = g;
}


public void addEdge(int otherNumber, double weight)
{
    if (shortestPaths != null)
              SimulationOutputter.die("Don't add node edge after caching "+
                                                    "shortest path info");
    if (weight < 0)
              SimulationOutputter.die("No negative-weight edges allowed");
    if (edges.containsKey(new Integer(otherNumber)))
              SimulationOutputter.die("in addEdge, edge "+number+" -> "+
                                                    otherNumber+" already exists.");
    else
```

```
            edges.put(new Integer(otherNumber), new Double(weight));

}



public double getEdge(int otherNumber)

{

    if (edges.containsKey(new Integer(otherNumber)))

            return edges.get(new Integer(otherNumber)).doubleValue();

    else

            return -1;

}



public HashMap<Integer, Double> getEdges()

{

    return edges;

}



public int getNumber()

{

    return number;
```

```
}


public String getDescription()
{
    return description;
}


public Path getShortestPath(int toNode)
{
    if (shortestPaths == null)
              {
                  shortestPaths = new HashMap<Integer, Path>();
                  Set<Integer> all = nodeLookup.keySet(); // from the Graph

                  HashMap<Integer, int[]> tmp = new HashMap<Integer, int[]>();
                  // tmp maps each node in the graph to its Dijkstra info
                  // for this node. The D-info has fields:
                  // 0: currently best-known distance to this node
                  // 1: last hop for best-known path to this node
// 2: 0/1 value for whether we've added this node
for (Integer q : all)
      {
        int[] dInfo = new int[3];
        if (q.intValue() == number)
              dInfo[0] = 0;
        else
              dInfo[0] = INT_MAX;
        dInfo[1] = -1;
        dInfo[2] = 0;
        tmp.put(q, dInfo);
      }
while (true)
      {
        Integer bval = null;
        int blen = INT_MAX;
        for (Integer q : all)
              {
                int[] info = tmp.get(q);
```

```
           if (info[2] == 0 && info[0] < blen)
                   {
                      bval = q;
                      blen = info[0];
                   }
        }
if (bval == null)
        break;
int[] info = tmp.get(bval);
info[2] = 1;
tmp.put(bval, info);
Path p = new Path(g);
int curr = bval.intValue();
while (curr >= 0)
        {
           p.addAtBeginning(curr);
           int[] values = tmp.get(new Integer(curr));
           curr = values[1];
        }
```

```
    shortestPaths.put(bval, p);

    HashMap<Integer, Double> edges =

            g.getEdges(bval.intValue());

    Set<Integer> adj = edges.keySet();

    for (Integer q : adj)

            {

                int[] adjInfo = tmp.get(q);

                if (adjInfo[2] == 0 &&

                        info[0] + edges.get(q).intValue() <

                        adjInfo[0])

                        {

                            adjInfo[0] =

                                    info[0] +

                                    edges.get(q).intValue();

                            adjInfo[1] = bval.intValue();

                            tmp.put(q, adjInfo);

                        }

            }

}
```

```
                        } // done caching this node's shortest path info

                    if (!shortestPaths.containsKey(new Integer(toNode)))

                        SimulationOutputter.die("No shortest path to node "+toNode);

                    return shortestPaths.get(new Integer(toNode));

                }

        }

    }
```

```
package hope;



import java.awt.MediaTracker;

import java.awt.Image;

import java.awt.Toolkit;

import java.awt.Component;

import java.awt.Canvas;

import java.awt.image.FilteredImageSource;

import java.awt.image.ReplicateScaleFilter;

import java.util.HashMap;



/**

   The ImageCache caches images of a given size so they only need to be read

   off disk once. It caches images for each output component (only the

   OutputterCanvas should be used).

 */

public class ImageCache

{

    private static HashMap<Component, ImageCache> caches;
private HashMap<String, HashMap<int[], Image>> images;
private Component c;


private ImageCache(Component c)
{
        images = new HashMap<String, HashMap<int[], Image>>();
        this.c = c;
}


/**
   This gets an unscaled image (with size as stored in its image file).


   @param fileName The name of the file in which the image is stored
   @return         The image
 */
public Image getImage(String fileName)
{
        return getImage(fileName, -1, -1);
}
```

```
   /**

      This gets a scaled image. Scaling can be done explicitly for both

      dimensions or in one dimension with the other held proportional.



      @param fileName The name of the file in which the image is stored

      @param xscale   The width in pixels of the scaled image (-1 to keep width proportional to the new
height)

      @param yscale   The height in pixels of the scaled image (-1 to keep height proportional to the new
width)

      @return         The image

    */

   public Image getImage(String fileName, int xscale, int yscale)

   {

          HashMap<int[], Image> extractedMap;

          if (images.containsKey(fileName))

             extractedMap = images.get(fileName);

          else

             {

                 extractedMap = new HashMap<int[], Image>();
```

```
              images.put(fileName, extractedMap);

    }

if (xscale < 0)

    xscale = -1;

else if (xscale == 0)

    xscale = 1;

if (yscale < 0)

    yscale = -1;

else if (yscale == 0)

    yscale = 1;

int[] toFind = new int[2];

toFind[0] = xscale;

toFind[1] = yscale;

if (extractedMap.containsKey(toFind))

    return extractedMap.get(toFind);

else

    {

              Image scaleImg = null;

              try {
```

```
MediaTracker mt = new MediaTracker(c);

Image img = Toolkit.getDefaultToolkit().getImage(fileName);

mt.addImage(img, 0);

mt.waitForID(0);

if (xscale < 0 && yscale < 0)

          {

              extractedMap.put(toFind, img);

              return img;

          }

else if (xscale < 0)

          xscale = img.getWidth(null)*yscale/img.getHeight(null);

else if (yscale < 0)

          yscale = img.getHeight(null)*xscale/img.getWidth(null);

if (xscale == 0)

          xscale = 1;

if (yscale == 0)

          yscale = 1;

scaleImg = Toolkit.getDefaultToolkit().

          createImage(new FilteredImageSource(img.getSource(),
                                                        new ReplicateScaleFilter(xscale, yscale)));
                mt.addImage(scaleImg, 1);
                mt.waitForID(1);
              }
              catch (Exception e)
                {
                      SimulationOutputter.die("Error loading image " +
                                             fileName);
                }
              extractedMap.put(toFind, scaleImg);
              return scaleImg;
        }
}


/**
   This gets the ImageCache for a given output component.


   @param c The Component on which the image will be displayed
   @return  The ImageCache
```

```
     */

    public static ImageCache getInstance(Component c)

    {

            if (caches == null)

                caches = new HashMap<Component, ImageCache>();

            if (caches.containsKey(c))

                return caches.get(c);

            else

                {

                        ImageCache newCache;

                        if (c == null)

                            newCache = new ImageCache(new Canvas());

                        else

                            newCache = new ImageCache(c);

                        caches.put(c, newCache);

                        return newCache;

                }

    }

}
```

```
package hope;


import java.awt.Canvas;

import java.util.ArrayList;

import java.awt.MediaTracker;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.Toolkit;

import java.awt.Image;


/**

   The MapperCanvas displays the ED with user-selected node locations and

   collects mouse clicks.

 */

public class MapperCanvas extends Canvas

{

    private Image img;

    private ArrayList<int[]> marks;

    /**

       This initializes the canvas. In particular, it loads the background

       image.


       @param imageFile The file containing the background image

     */

    public MapperCanvas(String imageFile)

    {

              marks = new ArrayList<int[]>();

              try

                  {
```

```
                img = Toolkit.getDefaultToolkit().getImage(imageFile);

                MediaTracker mt = new MediaTracker(this);

                mt.addImage(img, 0);

                mt.waitForID(0);

            }

        catch (Exception e)

            {

                System.out.println("Error loading image "+imageFile);

                System.exit(-1);

            }

        setSize(img.getWidth(null), img.getHeight(null));

        setVisible(true);

}


/**

   Marks a new location on the map and repaints the canvas so it shows up.


   @param x The x-coordinate of the new mark

   @param y The y-coordinate of the new mark

 */

public void mark(int x, int y)

{

        int[] mk = new int[2];

        mk[0] = x;

        mk[1] = y;

        marks.add(mk);

        repaint();

}
```

```
    /**
       Displays the hospital image with all the nodes already marked.
     */
    public void paint(Graphics g)
    {
            g.drawImage(img, 0, 0, Color.white, null);
            for (int[] q : marks)
                g.fillOval(q[0]-2, q[1]-2, 4, 4);
    }
}
```

```
package hope;



import javax.swing.JFrame;

import java.awt.Container;

import java.awt.BorderLayout;

import java.awt.event.WindowListener;

import java.awt.event.WindowEvent;

import java.awt.event.MouseListener;

import java.awt.event.MouseEvent;



/**

   The MapperFrame is the JFrame for the NodeMapper application. It captures

   mouse clicks and forwards them appropriately to the NodeMapper.

 */

public class MapperFrame extends JFrame implements WindowListener, MouseListener

{

    private MapperCanvas mc;

    private NodeMapper nm;

    /**
   This initializes the JFrame for the NodeMapper application. It sets up
   the MapperCanvas that shows the ED and previously marked nodes.

   @param imageFile The image file for the ED
   @param nm        The NodeMapper running the entire application
 */
public MapperFrame(String imageFile, NodeMapper nm)
{
        super("Team HOPE Node Mapper");
        this.nm = nm;
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        mc = new MapperCanvas(imageFile);
        c.add(mc, BorderLayout.CENTER);
        setSize(mc.getWidth(), mc.getHeight());
        mc.addMouseListener(this);
        addWindowListener(this);
        setVisible(true);
}
```

```
/**
   Collects mouse clicks from the MapperCanvas and informs the NodeMapper
   about them.
 */
public void mouseReleased(MouseEvent e)
{
         mc.mark(e.getX(), e.getY());
         nm.click(e.getX(), e.getY());
}


/**
         Exits the application if the application window is closed.
 */
public void windowClosing(WindowEvent e)
{
         System.exit(0);
}
```

```
    public void mouseClicked(MouseEvent e) {}

    public void mouseEntered(MouseEvent e) {}

    public void mouseExited(MouseEvent e) {}

    public void mousePressed(MouseEvent e) {}

    public void windowActivated(WindowEvent e) {}

    public void windowClosed(WindowEvent e) {}

    public void windowDeactivated(WindowEvent e) {}

    public void windowDeiconified(WindowEvent e) {}

    public void windowIconified(WindowEvent e) {}

    public void windowOpened(WindowEvent e) {}

}
```

```
package hope;


import javax.swing.JLabel;
import javax.swing.border.EmptyBorder;


/**
   The MetricLabel class is a JLabel for a particular metric that can be
   displayed.
 */
public abstract class MetricLabel extends JLabel
{
    private String name;
    private boolean enabled;
    private MetricPanel mp;


    /**
       This initializes the MetricLabel. It sets its common name, initial
       display state, and initial text.


       @param name     The common name of the metric
       @param enabled  If the metric is initially enabled
       @param initText The initial text in the label
     */
    public MetricLabel(String name, boolean enabled, String initText,
                                   MetricPanel mp)
    {
              super(initText);
              this.name = name;
              this.enabled = enabled;
```

```
		this.mp = mp;
		setBorder(new EmptyBorder(0, 5, 0, 5));
	}

	/**
	 * Getter for the common name of the metric
	 * 
	 * @return The metric's common name
	 */
	public String getName()
	{
		return name;
	}

	/**
	 * Getter for whether the metric is enabled. This means whether or not it
	 * is actually showing up in the MetricPanel.
	 * 
	 * @return Whether the metric is enabled
	 */
	public boolean isEnabled()
	{
		return enabled;
	}

	/**
	 * Setter for whether the metric is enabled. This means whether or not it
	 * is actually showing up in the MetricPanel.
```

```
	  @param enabled Whether the metric is now enabled
	 */
	public void setEnabled(boolean enabled)
	{
		this.enabled = enabled;
	}

	/**
	  Getter for the MetricPanel associated with this MetricLabel.

	  @return The MetricPanel
	 */
	public MetricPanel getMetricPanel()
	{
		return mp;
	}

	/**
	  The update method for the MetricLabel. It uses its stored MetricPanel
	  to gather the appropriate information for its display. All MetricLabels
	  must override this method.
	 */
	public abstract void update();
}
```

```
package hope;



import javax.swing.JPanel;

import javax.swing.JLabel;

import javax.swing.JButton;

import java.awt.GridLayout;

import java.util.Set;

import java.util.ArrayList;

import java.util.List;



/**

   The MetricPanel class provides a custimizable JPanel with metrics about the

   simulation.

 */

public class MetricPanel extends JPanel

{

    private ArrayList<MetricLabel> allLabels;

    private OutputterUI oui;

    private Set<Entity> entities;
private double time;



/**

   This creates the panel.



   @param oui The OutputterUI on this this panel rests

 */

public MetricPanel(OutputterUI oui)

{

          super();

          allLabels = new ArrayList<MetricLabel>();

          AvgBedLabel abl = new AvgBedLabel(false, this);

          allLabels.add(abl);

          AvgWaitingLabel al = new AvgWaitingLabel(false, this);

          allLabels.add(al);

          BedLabel bl = new BedLabel(false, this);

          allLabels.add(bl);

          WaitingLabel wl = new WaitingLabel(false, this);

          allLabels.add(wl);
          CensusLabel cl = new CensusLabel(true, this);

          allLabels.add(cl);

          TimeLabel tl = new TimeLabel(true, this);

          allLabels.add(tl);

          this.oui = oui;

          notifyDisplayChange();

}



/**

   Getter for the set of all entities in the display at the most recent

   MetricPanel update.



   @return The Entity set

 */

public Set<Entity> getEntities()

{

          return entities;

}
```

```
/**

   Getter for the simulation time at the most recent MetricPanel update.



   @return The simulation time (in seconds)

 */

public double getTime()

{

          return time;

}



/**

   Getter for the list of all the MetricLabels possible.



   @return The list of all MetricLabels

 */

public List<MetricLabel> getMetrics()

{

          return allLabels;

}
```

```
/**
   Re-layout the metrics based on potentially new preferences. This is in
   response to the CustomizeMetricsFrame doing its job.
 */
public void notifyDisplayChange()
{
          removeAll();
          int numMetrics = 0;
          for (MetricLabel ml : allLabels)
              if (ml.isEnabled())
                        numMetrics++;
          if (numMetrics == 0)
              {
                        validate();
                        return;
              }
          for (MetricLabel ml : allLabels)
              if (ml.isEnabled())
```

```
                add(ml);
        validate();
}


/**
   This is called to update the metrics. It makes sure the JPanel is
   sized properly.

   @param entities All the Entity objects currently displayed
   @param time     The simulation time (in seconds)
 */
public void updateAtTimeSlice(Set<Entity> entities, double time)
{
        this.time = time;
        this.entities = entities;
        for (MetricLabel ml : allLabels)
            ml.update();
        oui.validate();
}
```

```
}
```

```
package hope;

import java.awt.Color;

/**
   The PatientRepresentation class represents any patient in the simulation
   display.
 */
public class MidPatientRepresentation extends EntityRepresentation
{
    /**
       This sets up the default patient display (6x6 red dots).
     */
    public MidPatientRepresentation()
    {
              super("Patient", Color.red, 12, "evil.png", 23, -1,
                    "evil2.png", 20, -1, true, new PatientRepresentation());
    }
    /**

       This represents any Entity that is a patient.

     */

    public boolean represents(Entity e)

    {

            return (e instanceof Patient && ((Patient)e).getAge() <= 60 && ((Patient)e).getAge() >= 40);

    }

}
```

```
package hope;



/**

  The MoveAction class is a UIAction to move an Entity in the UI.

 */

public class MoveAction extends UIAction {

    private String description;

    private int newLoc;

    private double moveTime;



    /**

       This stores the information about the Entity's move.



       @param id          The Entity's unique ID within this OutputterUI

       @param description The Entity's new description as a result of this move

       @param newLoc      The location ID of the Entity's destination in this move

       @param moveTime    The number of simulation seconds this move will take

       @param time        The simulation time when this move will start

       @param oui         The OutputterUI in which this move will take place
```

```
     */
    public MoveAction(int id, String description, int newLoc,
                      double moveTime, double time, OutputterUI oui) {
        super(time, id, oui);
        this.description = description;
        this.newLoc = newLoc;
        this.moveTime = moveTime;
    }


    /**
      Please see {@link UIAction}.
     */
    public void execute() {
        Entity e = super.getOutputterUI().getEntity(getId());
        e.setDescription(description);
        e.move(newLoc, getTime(), moveTime);
    }
}
```

```
package hope;

import javax.swing.JOptionPane;

/**
   The NodeMapper class handles all the logic for requesting and recording
   node locations.
 */
public class NodeMapper
{
    private Graph g;
    private int[] nodes;
    private boolean[] used;
    private int position;
    private int numRem;
    private MapperFrame mf;
    private int currNode;
    /**
       This creates the NodeMapper, loading the graph and kicking off the UI.

       @param graphFile The file with the ED's graph
       @param imageFile The file with the background image to be calibrated
     */
    public NodeMapper(String graphFile, String imageFile)
    {
            g = new Graph(graphFile);
            nodes = g.getNodes();
            used = new boolean[nodes.length];
            position = 0;
```

```
          numRem = nodes.length;

          mf = new MapperFrame(imageFile, this);

          request();

}


/**

   This is called when the user clicks on a particular pixel on the

   MapperCanvas. It outputs the mapping for the node and requests a new one.


   @param x The x-coordinate of the mouse click

   @param y The y-coordinate of the mouse click

 */

public void click(int x, int y)

{

          numRem--;

          System.out.println(currNode+" "+x+" "+y);

          for (int i=0;i<nodes.length;i++)

              if (nodes[i] == currNode)

                      {

                          used[i] = true;

                          break;

                      }

          request();

}


/**

   This returns the ID of the next node for the user to place.


   @return The node ID of the next node to place
 */
public int getNext()
{
    if (numRem == 0)
      return -1;
    else
      {
          while (used[position])
            {
                position++;
                if (position == used.length)
                  position = 0;
            }
          int ret = position;
          position++;
          if (position == used.length)
            position = 0;
          return nodes[ret];
      }
}

/**
  This asks the user if they want to place a node. It contains a skip
  option so they can skip and come back to nodes they don't want to
  place yet.
 */
public void request()
{
    currNode = getNext();
```

```
        if (currNode < 0)
          System.exit(0); // We are done!!
        String description = g.getDescription(currNode);
        Object[] options = {"Place it!", "Skip", "Quit"};
        int resp = JOptionPane.showOptionDialog(null,
                                              "Would you like to place node "+
                                              currNode+" : "+description+"?",
                                              "Node placement?",
                                              JOptionPane.DEFAULT_OPTION,
                                              JOptionPane.QUESTION_MESSAGE,
                                              null, options, options[0]);
        switch (resp)
          {
          case 0:
                return;
          case 1:
                request();
                return;
          case 2:
          default:
                System.exit(0);
          }
  }
}
```

```
package hope;


/**
  The NodeMapperBase class starts off the NodeMapper application, handling
  the command-line parameters.
 */
public class NodeMapperBase
{
   /**
     The main method collects the command-line parameters and passes them
     to the NodeMapper.
     <p>
     Parameter 1: The graph file for the ED
     <p>
     Parameter 2: The image hospital to be handled.
    */
   public static void main(String[] args)
   {
          if (args.length < 2)
             {
                    System.out.println("java NodeMapperBase graph_file image_file");
                    System.exit(-1);
             }
          new NodeMapper(args[0], args[1]);
   }
}
```

```
package hope;



public class Nurse extends Entity

{



        public Nurse(int id, String name, String description,

                int location, OutputterUI oui, double entryTime) {

                        super(id, name, description, location, oui, entryTime);

         }



}
```

```
package hope;

import java.awt.Color;

/**
  The NurseRepresentation class represents any nurse in the simulation
  display.
 */
public class NurseRepresentation extends EntityRepresentation
{
    /**
      This sets up the default nurse display (6x6 cyan dots).
     */
    public NurseRepresentation()
    {
            super("Nurse", Color.cyan, 12, "evil.png", 23, -1,
                "evil2.png", 18, -1, true, null);
    }
    /**
       This represents any Entity that is a nurse.
     */
    public boolean represents(Entity e)
    {
            return (e instanceof Nurse);
    }
}
```

```
package hope;

import java.awt.Color;

/**
   The PatientRepresentation class represents any patient in the simulation
   display.
 */
public class OldPatientRepresentation extends EntityRepresentation
{
    /**
       This sets up the default patient display (6x6 red dots).
     */
    public OldPatientRepresentation()
    {
              super("Patient", Color.red, 12, "evil.png", 23, -1,
                    "evil2.png", 20, -1, true, new PatientRepresentation());
    }
    /**

       This represents any Entity that is a patient.

     */

    public boolean represents(Entity e)

    {

              return (e instanceof Patient && ((Patient)e).getAge() > 60);

    }

}
```

```
 */

package hope;



import java.awt.Canvas;

import java.util.ArrayList;

import java.awt.Image;

import java.awt.image.BufferedImage;

import java.awt.image.FilteredImageSource;

import java.awt.image.CropImageFilter;

import java.awt.image.ReplicateScaleFilter;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.Toolkit;

import java.util.Set;

import java.util.Date;



/**

  The OutputterCanvas class is the canvas displaying the ED and all the

  Entities within.

 */
```

```
public class OutputterCanvas extends Canvas {
    private static final int defaultCanvasWidth = 720;
    private static final int defaultCanvasHeight = 450;
    private Image img;
    private OutputterUI oui;
    private ArrayList<EntityDisplayInfo> points;
    private Image offImage;
    private boolean movedLastTime;

    /**
      This initializes the Canvas, in part by reading in the ED image.

      @param imageFile The file for the ED background image
      @param oui       The OutputterUI this canvas represents
     */
    public OutputterCanvas(String imageFile, OutputterUI oui) {
          this.oui = oui;
          img = ImageCache.getInstance(this).getImage(imageFile);
          offImage = new BufferedImage(img.getWidth(null),
                                img.getHeight(null),

                                BufferedImage.TYPE_INT_RGB);

    Graphics gOff = offImage.getGraphics();

    gOff.drawImage(img, 0, 0, Color.white, null);

    setSize(defaultCanvasWidth, defaultCanvasHeight);

    setVisible(true);

    points = new ArrayList<EntityDisplayInfo>();

    movedLastTime = false;

}



/**

  Get the height ratio between the hospital image and the default size. For

  instance, if the hospital image has height 900 and the default is a

  height of 450 for this canvas, the return will be 2.0.



  @return The height ratio

 */

public double getHeightRatio()

{
        return (((double)img.getHeight(null))/defaultCanvasHeight);
}



/**
   Get the width ratio between the hospital image and the default size. For
   instance, if the hospital image has width 1440 and the default is a
   width of 720 for this canvas, the return will be 2.0.

   @return The width ratio
 */
public double getWidthRatio()
{
        return (((double)img.getWidth(null))/defaultCanvasWidth);
}



/**
   Overridden to use the custom update method.
 */
public void paint(Graphics g)
```

```
{
          update(g);
}



/**
   This updates the ED image, displaying all entities in their current
   positions. It uses off-screen buffering to minimize flickering.
 */
public void update(Graphics g) {
          Graphics gOff = offImage.getGraphics();
          gOff.drawImage(img, 0, 0, Color.white, null);
          Object[] oentities = points.toArray();
          EntityDisplayInfo[] entities = new EntityDisplayInfo[oentities.length];
          for (int i=0;i<oentities.length;i++)
              entities[i] = (EntityDisplayInfo)oentities[i];
          for (EntityDisplayInfo q : entities) {
              EntityRepresentation er = q.getEntityRepresentation();
              gOff.setColor(er.getDotColor());
              if (!er.usingDefault())
```

```
             gOff.drawImage(er.getImage(),

                      q.getCurrPixel().getX() - er.getWidth()/2,

                      q.getCurrPixel().getY() - er.getHeight()/2,

                      null, null);

          else

             gOff.fillOval(q.getCurrPixel().getX() - er.getWidth()/2,

                      q.getCurrPixel().getY() - er.getHeight()/2,

                      er.getWidth(), er.getHeight());

        }

        g.drawImage(getScaledImage(offImage, getSize().width, getSize().height),

               0, 0, Color.white, null);

    }



    /**

     This updates the display at a given simulation time. It polls all

     active entities for their display information and then repaints the

     canvas is necessary.



     @param e    The set of entities active in the OutputterUI
```

```
      @param time The simulation time (in seconds)

     */

    public void updateAtTimeSlice(Set<Entity> e, double time) {

            points = new ArrayList<EntityDisplayInfo>();

            for (Entity q : e)

                points.add(q.generateEDI(time));

            repaint();

    }



    private Image getScaledImage(Image img, int w, int h){

            return Toolkit.getDefaultToolkit().

                createImage(new FilteredImageSource(img.getSource(),

                                                                new ReplicateScaleFilter(w, h)));

    }

}
```

```
package hope;



import java.awt.event.ComponentEvent;

import java.awt.event.ComponentListener;

import java.awt.event.WindowListener;

import java.awt.event.WindowEvent;

import java.awt.event.ActionListener;

import java.awt.event.ActionEvent;

import java.util.Date;

import java.util.HashSet;

import java.util.HashMap;

import java.util.Set;

import java.util.NoSuchElementException;

import java.util.StringTokenizer;

import java.util.ArrayList;

import java.util.List;

import java.io.IOException;

import java.io.FileReader;

import java.io.BufferedReader;
import java.awt.BorderLayout;

import java.awt.Container;

import javax.swing.JPanel;

import javax.swing.JFrame;

import javax.swing.JButton;

import javax.swing.JOptionPane;

import javax.swing.border.EmptyBorder;



/**

   The OutputterUI class is the base frame for the simulation model display. It

   handles many of the logistics of the simulation display (like all the

   entities in the ED, where they are displayed, and how they are manipulated).

 */

public class OutputterUI extends JFrame implements WindowListener,

                                                   ActionListener,

                                                   ComponentListener

{

    private Graph g;

    private OutputterCanvas oc;
```

```
/** The identifier for a patient. This is an Entity "type." */

public static final int ID_PATIENT = 1;

/** The identifier for a nurse. This is an Entity "type." */

public static final int ID_NURSE = 2;

/** The identifier for a doctor. This is an Entity "type." */

public static final int ID_DOCTOR = 3;

private List<UIAction> futureActions;

private static final int HALT_SIZE = 1024;

private static final int UNHALT_SIZE = 512;

private boolean paused = false;

private long lastUpdateTime;

private HashMap<Integer, Entity> entityMap;

private HashMap<Integer, Pixel> locationPixels;

private double displayTime;

private JPanel bottom;

private JButton stopStart;

private HashSet<CanvasRegion> claimedRegions;

private MetricPanel mp;

private JButton customizeButton;
private DisplayControlPanel dcp;
private DisplayControl dc;
private ArrayList<EntityRepresentation> representations;
/** The folder where avatars are located */
public static final String AVATAR_PREFIX = "avatars/";


/**
  This initializes the UI for the application. It creates the
  canvas and all display controls.


  @param locationToPixelFile The name of the file mapping location ID to pixel location
  @param hospitalImageFile   The name of the file containing the ED image
  @param dc                  The initial DisplayControl for the display
 */
public OutputterUI(String locationToPixelFile, String hospitalImageFile,
                      DisplayControl dc) {
   super("Team HOPE Simulation Model");
        this.dc = dc;
   this.g = SimulationOutputter.getGraph();
```

```
readLocationPixels(locationToPixelFile);

Container c = getContentPane();

c.setLayout(new BorderLayout());

oc = new OutputterCanvas(hospitalImageFile, this);

c.add(oc, BorderLayout.CENTER);

bottom = new JPanel(new BorderLayout());

    JPanel ssPanel = new JPanel();

    JPanel customPanel = new JPanel();

stopStart = new StopStartButton(this);

    ssPanel.add(stopStart);

    customizeButton = new JButton("Custom Display");

    customizeButton.addActionListener(this);

    customPanel.add(customizeButton);

    JPanel buttonPanel = new JPanel(new BorderLayout());

    buttonPanel.add(ssPanel, BorderLayout.WEST);

    buttonPanel.add(customPanel, BorderLayout.EAST);

oc.addMouseListener(new ClickListener(this));

bottom.add(buttonPanel, BorderLayout.WEST);

    mp = new MetricPanel(this);
```

```
        bottom.add(mp, BorderLayout.EAST);
        dcp = new DisplayControlPanel(dc, this);
        dcp.setBorder(new EmptyBorder(0, 20, 0, 20));
        bottom.add(dcp, BorderLayout.CENTER);
   c.add(bottom, BorderLayout.SOUTH);
   setSize(oc.getWidth(), oc.getHeight());
   addWindowListener(this);
   addComponentListener(this);
   setVisible(true);
   futureActions = new ArrayList<UIAction>();
   entityMap = new HashMap<Integer, Entity>();
   displayTime = -1;
   lastUpdateTime = new Date().getTime();
        claimedRegions = new HashSet<CanvasRegion>();
        representations = new ArrayList<EntityRepresentation>();
        registerEntityRepresentation(new PatientRepresentation());
        registerEntityRepresentation(new NurseRepresentation());
        registerEntityRepresentation(new DoctorRepresentation());
}
```

```
/**
  Handler for control buttons, like the customize display button
 */
public void actionPerformed(ActionEvent e)
{
        if (e.getActionCommand().equals(customizeButton.getText()))
            new CustomizationFrame(this, mp);
}


/**
  Add an Entity at the specified simulation time. Note that this won't
  cause the Entity to be added until the time specified.


  @param id          The Entity's unique ID
  @param type        The Entity's type (as in OutputterUI.ID_PATIENT)
  @param name        The Entity's name
  @param description The Entity's initial description
  @param location    The Entity's initial location ID
```

```
     @param time        The simulation time at which this Entity will be added

    */



             public void addPatient(int id, String name, String diagnosisCode, int age, String race, int triageNum,
String description,

           int location, double time) {



             AddPatientAction aa = new AddPatientAction(id, name, diagnosisCode, age, race, triageNum,
description, location,

              time, this);

       handleNewAction(aa);

   }



             public void addNurse(int id, String name, String description,

           int location, double time) {




       AddNurseAction aa = new AddNurseAction(id, name, description, location,

              time, this);

       handleNewAction(aa);
```

```
        }



        public void addDoctor(int id, String name, String description,

      int location, double time) {





   AddDoctorAction aa = new AddDoctorAction(id, name, description, location,

         time, this);

   handleNewAction(aa);

        }



/**

   Change an entity's description at the specified simulation time. Note

   that this won't cause the Entity to change its description until the

   time specified.



   @param id             The Entity's unique ID

   @param newDescription The Entity's new description

   @param time           The simulation time at which this Entity's description will be changed
```

```
 */
public void changeEntityDescription(int id, String newDescription,
    double time) {
  ChangeDescriptionAction ca = new ChangeDescriptionAction(id,
      newDescription, time, this);
  handleNewAction(ca);
}


/**
     This sets the canvas to the proper size when the whole window is
     resized.
 */
public void componentResized(ComponentEvent e) {
     oc.setSize(getWidth(), getHeight());
}


/**
  This displays the info about an Entity if it was clicked on. It has no
  effect if the passed mouse click position was not over any Entity.
```

```
    @param click The position of the mouse click
   */
  public void displayEntityInfo(Pixel click) {
        Entity e;
        CanvasRegion entityCR;
        for(Integer i : entityMap.keySet()){
          e = entityMap.get(i);
          entityCR = e.getCurrentEntityRegion();
          if(entityCR.contains(click)){
                String message = e.getDescription();
                JOptionPane.showMessageDialog(this, message);
          }
        }
  }


  /**
    This claims a rectangular region of the ED display. The region is selected such that it does not overlap
previously chaimed regions and is as close as possible to the desired location.
```

```
  @param location The middle of the region being claimed

  @param width    The width in pixels of the region being claimed

  @param height   The height in pixels of the region being claimed

  @return         The region that was claimed

 */

public CanvasRegion claimRegion(Pixel location, int width, int height)

{

        CanvasRegion closest = getClosestUnclaimedRegion(location, location,

                                                                 width, height, false,

                                                                 false, false, false);

        claimedRegions.add(closest);

        return closest;

}



/**

  This deletes an Entity immediately. It should not be called directly, and

  is really only used by RemoveAction. This causes the simulation to exit

  if no Entity with this ID exists.
  @param id The unique Entity ID of the Entity to be removed
  @see hope.OutputterUI#removeEntity removeEntity
 */
public void deleteEntity(int id) {
   if (entityMap.containsKey(new Integer(id)))
      entityMap.remove(new Integer(id));
   else
      SimulationOutputter.die("can't delete entity " + id + " as that id "
                                          + "does not exist");
}


/**
  Getter for the OutputterCanvas displaying the ED.

  @return The ED's OutputterCanvas
 */
public OutputterCanvas getCanvas()
{
        return oc;
}

/**
  Getter for the EntityRepresentation associated with a given Entity.
  This causes the simulation to exit if there were multiple registered
  EntityRepresentations that matched or none.

  @param e The entity to have its EntityRepresentation searched for
  @return  The matching EntityRepresentation
 */
public EntityRepresentation getAssociatedEntityRepresentation(Entity e)
{
        EntityRepresentation erMatch = null;
        for (EntityRepresentation er : representations)
                   if (er.isEnabled() && er.represents(e))
                      {
                              if (erMatch == null || er.getOverride().equals(erMatch))
                                 erMatch = er;
```

```
                        else
                            SimulationOutputter.die("To many ERs match Entity");
                    }
          if (erMatch == null)
              SimulationOutputter.die("No EntityRepresentations match Entity");
          return erMatch;
}


/**
   This returns the DisplayControl associated with the display. It
   exhibits any changes made in the DisplayControl since the display was
   started. For instance, if the simulation speed was changed, this is
   exhibited.

   @return The DisplayControl
 */
public DisplayControl getDisplayControl()
{
          return dc;
```

```
}



/**

   Returns the number of real-life milliseconds the simulation has run

   since the last getElapsedTime call. By "run" we mean operated, so if it

   is paused this function will return 0.



   @return The number of milliseconds the simulation has been running

 */

public long getElapsedTime() {

          if (paused)

              {

                        lastUpdateTime = (new Date()).getTime();

                        return 0;

              }

          else

              {

                        long toReturn = (new Date()).getTime() - lastUpdateTime;

                        lastUpdateTime = (new Date()).getTime();
                        return toReturn;
              }
}


/**
   Get an Entity given its unique ID. This causes the simulation to
   exit if no Entity exists with this ID.

   @param id The unique ID of the Entity desired
   @return   The Entity retrieved
 */
public Entity getEntity(int id) {
    if (entityMap.containsKey(new Integer(id)))
        return entityMap.get(new Integer(id));
    else
        SimulationOutputter.die("No entity with ID " + id +
                                                         " in getEntity");
    return null; // I have no idea why Java wanted this here.....
}
```

```
/**

   Get the list of all available EntityRepresentations.



   @return The representations

 */

public List<EntityRepresentation> getEntityRepresentations()

{

        return representations;

}



/**

   Getter for the Graph representing the ED. This contains all location,

   their descriptions, and edges between locations.



   @return The ED's Graph

 */

public Graph getGraph() {

    return g;
}



/**

   Gets the factor by which the canvas will shrink heights.



   @return The ratio

 */

public double getHeightRatio()

{

          return oc.getHeightRatio();

}



/**

   Returns the pixel associated with an ED location, given its location ID.

   This causes the simulation to exit if the location ID is invalid.



   @param location The location ID of the desired location

   @return         The pixel corresponding to the desired location

 */
```

```
public Pixel getLocationPixel(int location) {

    if (locationPixels.containsKey(new Integer(location)))

        return locationPixels.get(new Integer(location));

    else

        SimulationOutputter.die("invalid location to getLocationPixel: "

                                                  + location);

    return null;

}



/**

   Gets the factor by which the canvas will shrink widths.



   @return The ratio

 */

public double getWidthRatio()

{

          return oc.getWidthRatio();

}
```

```
/**
   This adds a new Entity immediately. It should not be called directly, and
   is really only used by AddAction. This causes the simulation to exit if
   an Entity with this ID already exists.

   @param e The Entity to be added
   @see hope.OutputterUI#addEntity addEntity
 */
public void insertEntity(Entity e) {
    if (entityMap.containsKey(new Integer(e.getId())))
        SimulationOutputter.die("entity with ID " + e.getId() +
                                                        " already in the map");
    entityMap.put(new Integer(e.getId()), e);
}


/**
   Returns whether the simulation display is paused.

   @return Whether the simulation display is paused
 */
public boolean isPaused()
{
        return paused;
}


/**
   Move an Entity to a new location starting at the specified simulation
   time. Note that this won't cause the Entity to move until the time
   specified.

   @param id          The Entity's unique ID
   @param description The Entity's new description as a result of this move
   @param newLoc      The destination location ID for this Entity's travel
   @param time        The simulation time at which this Entity will move
   @param moveTime    The number of simulation seconds the move will take
 */
public void moveEntity(int id, String description, int newLoc, double time,
        double moveTime) {
```

```
    MoveAction ma = new MoveAction(id, description, newLoc, moveTime, time,
            this);
    handleNewAction(ma);
}


/**
   This pauses the simulation model display.
 */
public void pause() {
    paused = true;
}


/**
   This registers a potential entity representation, which future entities
   being generated will be checked against. If the EntityRepresentation
   overrides a representation that does not exist, the simulation will exit.

   @param er The EntityRepresentation to be registered
 */
```

```
public void registerEntityRepresentation(EntityRepresentation er)
{
        if (er.getOverride() != null &&
            !representations.contains(er.getOverride()))
            SimulationOutputter.die("Invalid EntityRepresentation override");
        representations.add(er);
}


/**
  Remove an Entity from the simulation at the specified simulation time.
  Note that this won't cause the Entity to be removed until the time
  specified.


  @param id          The Entity's unique ID
  @param time        The simulation time at which this Entity will be removed
 */
public void removeEntity(int id, double time) {
    RemoveAction ra = new RemoveAction(id, time, this);
    handleNewAction(ra);
```

```
}

/**
   Whether we should halt the Python simulation based on the number of
   queued UIActions.

   @return Whether we should halt
 */
public boolean shouldHalt() {
    return futureActions.size() > HALT_SIZE;
}

/**
   Whether we should unhalt the Python simulation based on the number of
   queued UIActions.

   @return Whether we should unhalt
 */
public boolean shouldUnHalt() {
    return futureActions.size() < UNHALT_SIZE;
}

/**
   This unclaims a previously claimed region of the display.

   @param cr The region to be unclaimed
 */
public void unclaimRegion(CanvasRegion cr)
{
        claimedRegions.remove(cr);
}

/**
   This unpauses the simulation model display.
 */
public void unpause() {
    paused = false;
}
```

```
/**
  Update all UI components at a given simulation time. This is called by
  SimulationOutputter periodically.

  @param time The simulation time (in seconds) to be displayed
 */
public void updateAtTimeSlice(double time) {
    while (!futureActions.isEmpty()
            && futureActions.get(0).getTime() < time) {
        futureActions.remove(0).execute();
    }
    Set<Entity> entities = new HashSet<Entity>();
    for (Integer q : entityMap.keySet())
        entities.add(entityMap.get(q));
          mp.updateAtTimeSlice(entities, time);
    oc.updateAtTimeSlice(entities, time);
          dcp.updateAtTimeSlice();
}
```

```
/**
   This exits the simulation display when the window is closed.
 */
public void windowClosing(WindowEvent e) {
    SimulationOutputter.die(0);
}


// This recursive function returns the closest region of appropriate size
// to the requested one (in terms of distances of the centers)
private CanvasRegion getClosestUnclaimedRegion(Pixel curr, Pixel want,
                                               int width, int height,
                                               boolean beenLeft,
                                               boolean beenRight,
                                               boolean beenUp,
                                               boolean beenDown)
{
          // Could add some logic here to deal with off-screen positioning
          // Doesn't seem like a big deal, so I'll leave it for now
          // Further, banning off-screen claims could lead to infinite loops on
          // full maps, which would be annoying.
          CanvasRegion crHere = new CanvasRegion(curr, width, height);
          HashSet<CanvasRegion> crOverlap = new HashSet<CanvasRegion>();
          for (CanvasRegion cr : claimedRegions)
             if (cr.overlaps(crHere))
                      crOverlap.add(cr);
          if (crOverlap.size() == 0)
             return crHere;
          int pushLeft = 0;
          int pushRight = 0;
          int pushUp = 0;
          int pushDown = 0;
          int l = curr.getX() - width/2;
          int r = l + width;
          int t = curr.getY() - height/2;
          int b = t + height;
          for (CanvasRegion cr : crOverlap)
             {
```

```
            int ol = cr.getMiddle().getX() - cr.getWidth()/2;

            int or = ol + cr.getWidth();

            int ot = cr.getMiddle().getY() - cr.getHeight()/2;

            int ob = ot + cr.getHeight();

            pushLeft = Math.max(pushLeft, r - ol);

            pushRight = Math.max(pushRight, or - l);

            pushUp = Math.max(pushUp, b - ot);

            pushDown = Math.max(pushDown, ob - t);

    }

Pixel pl = new Pixel(curr.getX() - pushLeft, curr.getY());

Pixel pr = new Pixel(curr.getX() + pushRight, curr.getY());

Pixel pu = new Pixel(curr.getX(), curr.getY() - pushUp);

Pixel pd = new Pixel(curr.getX(), curr.getY() + pushDown);

CanvasRegion[] c = new CanvasRegion[4];

for (int i=0;i<4;i++)

    c[i] = null;

if (!beenRight)

    c[0] = getClosestUnclaimedRegion(pl, want, width, height, true,

                                                 false, beenUp, beenDown);
```

```
if (!beenLeft)
    c[1] = getClosestUnclaimedRegion(pr, want, width, height, false,
                                    true, beenUp, beenDown);
if (!beenDown)
    c[2] = getClosestUnclaimedRegion(pu, want, width, height, beenLeft,
                                    beenRight, true, false);
if (!beenUp)
    c[3] = getClosestUnclaimedRegion(pd, want, width, height, beenLeft,
                                    beenRight, false, true);
double dbest = 100000000;
CanvasRegion crBest = null;
for (int i=0;i<4;i++)
    {
            if (c[i] == null)
                continue;
            int dx = c[i].getMiddle().getX() - want.getX();
            int dy = c[i].getMiddle().getY() - want.getY();
            double dst = Math.sqrt(dx*dx + dy*dy);
            if (dst < dbest)
```

```
                            {

                                      dbest = dst;

                                      crBest = c[i];

                            }

              }

          return crBest;

}



private void handleNewAction(UIAction uia) {

    if (uia.getTime() > displayTime) {

        if (futureActions.isEmpty()

                        || futureActions.get(futureActions.size() - 1).getTime()

                        <= uia.getTime())

            futureActions.add(uia);

        else {

            SimulationOutputter.die("A message came in out of order "

                                                                   + "with regards to timestamp");

        }

    } else
```

```
        uia.execute();
}

private void readLocationPixels(String locationToPixelFile) {
    locationPixels = new HashMap<Integer, Pixel>();
    try {
        BufferedReader br = new BufferedReader(new FileReader(
                locationToPixelFile));
        String line;
        while ((line = br.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(line);
            int node = Integer.parseInt(st.nextToken());
            int x = Integer.parseInt(st.nextToken());
            int y = Integer.parseInt(st.nextToken());
            locationPixels.put(new Integer(node), new Pixel(x, y));
        }
    } catch (IOException e) {
        SimulationOutputter.die("Error reading location to pixel file");
    } catch (NoSuchElementException e) {
        SimulationOutputter.die("Malformed location to pixel file");

    } catch (NumberFormatException e) {

        SimulationOutputter.die("Malformed location to pixel file");

    }

}



public void windowActivated(WindowEvent e) {

}



public void windowClosed(WindowEvent e) {

}



public void windowDeactivated(WindowEvent e) {

}



public void windowDeiconified(WindowEvent e) {

}



public void windowIconified(WindowEvent e) {
    }


    public void windowOpened(WindowEvent e) {

    }


    public void componentHidden(ComponentEvent e) {

    }


    public void componentMoved(ComponentEvent e) {

    }


    public void componentShown(ComponentEvent e) {

    }

}
```

```
package hope;


import java.util.ArrayList;


/**
   The Path class stores a path through the ED graph.
 */
public class Path
{
    private ArrayList<Integer> path; // stores list in *reverse* order
    private double length;
    private Graph g;
    private int[] cachedGraph;

    /**
       This initializes the path. No nodes are present at the beginning; they
       are added later with the public class methods. The Graph parameter is
       used to calculate the path length.
```

```
  @param g The ED's Graph
 */
public Path(Graph g)
{
        this.g = g;
        path = new ArrayList<Integer>();
        length = 0;
        cachedGraph = null;
}


/**
  This adds a node at the beginning of the path. The Path is optimized for
  this sort of addition, as it operates in O(1) time.

  @param n The ID of the node to be added
 */
public void addAtBeginning(int n)
{
        cachedGraph = null;
```

```
          path.add(new Integer(n)); // O(1) operation

          if (path.size() > 1)

              length += g.getEdge(n, path.get(path.size()-2).intValue());

}



/**

   This adds a node at the end of the path. The Path is not optimized for

   this sort of addition, as it operates in O(n) time.



   @param n The ID of the node to be added

 */

public void addAtEnd(int n)

{

          cachedGraph = null;

          path.add(new Integer(n), 0); // O(n) operation

          if (path.size() > 1)

              length += g.getEdge(n, path.get(1));

}
```

```
/**
   This returns the length of the Path.



   @return The length
 */
public double getLength()
{
          return length;
}


/**
   This returns an array with all the nodes in the Path.



   @return All the nodes
 */
public int[] getNodes()
{
          cachedGraph = new int[path.size()];
          int insPos = path.size()-1;
          for (Integer q : path)

              cachedGraph[insPos--] = q.intValue();

          return cachedGraph;

  }



  /**

     Display the Path's nodes and length.

   */

  public String toString()

  {

          String ret = "Path:";

          int[] nodes = getNodes();

          for (int i=0; i<nodes.length; i++)

              ret = ret + " " + nodes[i];

          return ret + " Length: " + getLength();

  }

}
```

```
package hope;



public class Patient extends Entity

{



  private String diagnosisCode;

  private int triageNum;

  private int age;

  private String race;



      public Patient(int id, String name, String description,

                              int location, OutputterUI oui, double entryTime, String diagnosisCode, int age, String
race, int triageNum) {

                                          super(id, name, description, location, oui, entryTime);

                                          this.diagnosisCode = diagnosisCode;

                                          this.triageNum = triageNum;

                                          this.age = age;

                                          this.race = race;

                }
```

```
public String getDiagnosis(){

            return diagnosisCode;

}



public int getTriageNum(){

              return triageNum;

}



public int getAge(){

              return age;

}



public String getRace(){

              return race;

}
```

```
}
```

```
package hope;


import java.awt.Color;


/**
   The PatientRepresentation class represents any patient in the simulation
   display.
 */
public class PatientRepresentation extends EntityRepresentation
{
    /**
       This sets up the default patient display (6x6 red dots).
     */
    public PatientRepresentation()
    {
              super("Patient", Color.red, 12, "evil.png", 23, -1,
                    "evil2.png", 20, -1, true, null);
              // null >> new PatientRepresentation()
    }
    /**

       This represents any Entity that is a patient.

     */

    public boolean represents(Entity e)

    {

             return (e instanceof Patient);

    }

}
```

```
package hope;



/**

   The Pixel class stores a pixel with x- and y-coordinates.

 */

public class Pixel

{

    private int x;

    private int y;



    /**

       Create the pixel, given its coordinates.



       @param x The x-coordinate of the pixel

       @param y The y-coordinate of the pixel

     */

    public Pixel(int x, int y)

    {

              this.x = x;
```

```
          this.y = y;
}

/**
   Pixels are equal if their x- and y-coordinates are the same.
 */
public boolean equals(Object o)
{
          if (o instanceof Pixel)
              return (x == ((Pixel)o).getX() && y == ((Pixel)o).getY());
          else
              return false;
}

/**
   Getter for the Pixel's x-coordinate.

   @return The x-coordinate
 */
```

```
    public int getX()

    {

              return x;

    }



    /**

       Getter for the Pixel's y-coordinate.



       @return The y-coordinate

     */

    public int getY()

    {

              return y;

    }

}
```

```
package hope;


/**
   The RemoveAction class is a UIAction to remove an Entity from the UI.
 */
public class RemoveAction extends UIAction {
    /**
       Initialize the RemoveAction.


       @param id             A unique ID (of any entity in the whole OutputterUI)
       @param time           The simulation model time when the entity will be added
     */
    public RemoveAction(int id, double time, OutputterUI oui) {
            super(time, id, oui);
    }


    /**
       Please see {@link UIAction}.
     */
    public void execute() {
            super.getOutputterUI().getEntity(getId()).leave();
            super.getOutputterUI().deleteEntity(getId());
    }
}
```

```
package hope;


import java.net.Socket;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Date;
import java.io.FileReader;
import java.io.FileOutputStream;
import java.io.File;
import java.io.IOException;
import java.util.StringTokenizer;


/**
  The SimulationOutputter class is the main loop for the simulation display.
  It also provides some package-wide functionalty (exiting).
 */
public class SimulationOutputter {
   private static final int NURSE_OFFSET = 0;
   private static final int DOCTOR_OFFSET = 1000;
   private static final int PATIENT_OFFSET = 2000;


   private static double time;
   private static Graph g;
   private static final String configFile = ".config";
   private static PrintWriter pw;


   /**
     The causes the simulation display to exit with error code and no message.
    */
   public static void die()
   {
           die(-1, null);
   }


   /**
     This causes the simulation display to exit with the specified exit code
     and no message.
```

```
   @param code The exit code
 */
public static void die(int code)
{
        die(code, null);
}

/**
   This causes the simulation display to exit with error code and the
   specified message.

   @param message The message to be displayed
 */
public static void die(String message)
{
        die(-1, message);
}
```

```
/**
   This causes the simulation display to exit with the specified exit code
   and the specified message.

   @param code    The exit code
   @param message The message to be displayed
 */
public static void die(int code, String message)
{
          if (message != null)
              System.out.println("JAVA MESSAGE: "+message);
          if (pw != null)
              {
                        try {
                            pw.println("015");
                        } catch (Exception e) {
                            System.exit(-1);
                        }
              }
          System.exit(code);
}



/**
   This is a getter for the Graph of ED locations and links.


   @return The Graph
 */
public static Graph getGraph() {
          return g;
}



/**
   The main method contains the main loop of the while simulation display.
   It parses messages from the Python code and passes them to the
   OutputterUI. It also tells the OutputterUI when to display itself.
   Last, it handles the reading and outputting of the display preferences.
   It takes several parameters from the Python code:
   <p>
   Parameter 1: The port number for the interprocess communication.

   <p>

   Parameter 2: The name of the file with the ED graph.

   <p>

   Parameter 3: The name of the file mapping location ID to pixel on the

   ED image.

   <p>

   Parameter 4: The name of the image file for the ED image

 */

public static void main(String args[]) {

        int port = 0;

        if (args.length < 4)

            die("java SimulationOutputter port graph_file "

                    + "location_to_pixel_file, hospital_image");

        g = new Graph(args[1]);

        try {

            port = Integer.parseInt(args[0]);

        } catch (Exception e) {

            die("Error parsing port");
        }

        Socket s = null;

        try {

            s = new Socket((String) null, port);

        } catch (Exception e) {

            die("Error opening socket");

        }

        BufferedReader br = null;

        pw = null;

        try {

            br = new BufferedReader(new InputStreamReader(s.getInputStream()));

            pw = new PrintWriter(s.getOutputStream());

        } catch (Exception e) {

            die("Error opening socket streams");

        }



        // here is where initial configuration info will be set up

        // numbers of doctors/nurses, amounts of materials, etc. will be set up
```

```
DisplayControl dc = new DisplayControl(configFile);

OutputterUI oui = new OutputterUI(args[2], args[3], dc);

pw.println("001"); // now that the UI is up, we are ready to go

pw.flush();



Date d = new Date();

double doneSec = 0;

double currSec = 0;

boolean halted = false;

while (doneSec == 0 || currSec < doneSec) {

   currSec += ((double)oui.getElapsedTime()) / 1000 *

            dc.getSpeedMultiplier();

   if (doneSec != 0 && currSec > doneSec)

            currSec = doneSec;

   if (halted && oui.shouldUnHalt()) {

            halted = false;

            pw.println("001");

            pw.flush();

   }
```

```
try {

        String diagnos = null;

        String line = null;

        while (doneSec == 0 && br.ready() && !halted) {

            line = br.readLine();

            String[] tokens = split(line);

            if (tokens.length < 2)

                    die("Bad line received: "+line);

            double sec = Double.parseDouble(tokens[1]);

            String description;

            switch (Integer.parseInt(tokens[0])) {

            case 2:

                    doneSec = sec;

                    break;

            case 3:

                    if (tokens.length < 5)

                        die("Bad line received: "+line);

                    description = readDesc(tokens, 5);

                    oui.moveEntity(Integer.parseInt(tokens[2])
            + PATIENT_OFFSET, description, Integer
            .parseInt(tokens[3]), sec, Double
            .parseDouble(tokens[4]));
    break;
case 4:
    if (tokens.length < 3)
      die("Bad line received: "+line);
    description = readDesc(tokens, 3);
    oui.changeEntityDescription(Integer.parseInt(tokens[2])
                      + PATIENT_OFFSET, description, sec);
    break;
case 5:
    if (tokens.length < 4)
      die("Bad line received: "+line);
    description = readDesc(tokens, 5);
    oui.addPatient(Integer.parseInt(tokens[2])
            + PATIENT_OFFSET,
            "Patient" + Integer.parseInt(tokens[2]),
                                              tokens[4], Integer.parseInt(tokens[5]), tokens[6],
Integer.parseInt(tokens[7]),

                                        description, Integer.parseInt(tokens[3]), sec);

                    break;

            case 6:

                    if (tokens.length < 4)

                        die("Bad line received: "+line);

                    description = readDesc(tokens, 4);

                    oui.addNurse(Integer.parseInt(tokens[2])

                                        + NURSE_OFFSET, "Nurse"

                                        + Integer.parseInt(tokens[2]), description,

                                        Integer.parseInt(tokens[3]), sec);

                    break;

            case 7:

                    if (tokens.length < 4)

                        die("Bad line received: "+line);

                    description = readDesc(tokens, 4);

                    oui.addDoctor(Integer.parseInt(tokens[2])

                                        + DOCTOR_OFFSET,
                              "Doctor" + Integer.parseInt(tokens[2]),
                              description, Integer.parseInt(tokens[3]), sec);
                    break;
          case 8:
                    if (tokens.length < 3)
                        die("Bad line received: "+line);
                    oui.removeEntity(Integer.parseInt(tokens[2])
                                   + PATIENT_OFFSET, sec);
                    break;
          case 9:
                    if (tokens.length < 3)
                        die("Bad line received: "+line);
                    oui.removeEntity(Integer.parseInt(tokens[2])
                                   + NURSE_OFFSET, sec);
                    break;
          case 10:
                    if (tokens.length < 3)
                        die("Bad line received: "+line);
                    oui.removeEntity(Integer.parseInt(tokens[2])
                + DOCTOR_OFFSET, sec);
    break;
case 11:
    if (tokens.length < 3)
      die("Bad line received: "+line);
    description = readDesc(tokens, 3);
    oui.changeEntityDescription(Integer.parseInt(tokens[2])
                       + NURSE_OFFSET, description, sec);
    break;
case 12:
    if (tokens.length < 3)
      die("Bad line received: "+line);
    description = readDesc(tokens, 3);
    oui.changeEntityDescription(Integer.parseInt(tokens[2])
                       + DOCTOR_OFFSET, description, sec);
    break;
case 13:
    if (tokens.length < 5)
      die("Bad line received: "+line);
```

```
        description = readDesc(tokens, 5);

        oui.moveEntity(Integer.parseInt(tokens[2])

                    + NURSE_OFFSET, description, Integer

                    .parseInt(tokens[3]), sec, Double

                    .parseDouble(tokens[4]));

        break;

case 14:

        if (tokens.length < 5)

            die("Bad line received: "+line);

        description = readDesc(tokens, 5);

        oui.moveEntity(Integer.parseInt(tokens[2])

                    + DOCTOR_OFFSET, description, Integer

                    .parseInt(tokens[3]), sec, Double

                    .parseDouble(tokens[4]));

        break;

default:

        die("Bad line received: "+line);

}
```

```
         if (doneSec == 0 && oui.shouldHalt() && !halted) {
              halted = true;
              pw.println("000");
              pw.flush();
         }
       }
       oui.updateAtTimeSlice(currSec);
       Thread.sleep(dc.getRefreshRateMs());
  } catch (InterruptedException e) {
       SimulationOutputter.die("InterruptedException: " + e);
  }
  catch (IOException e) {
       SimulationOutputter.die("IOException: " + e);
  }
}


// we still need to incorporate the summary UI


try {
    pw.close();

    br.close();

    pw = null;

    br = null;

}

catch (Exception e) {

    die("Error closing socket streams");

}

dc.printToFile(configFile);



while(true)

    {

              try

                  {

                            oui.updateAtTimeSlice(doneSec);

                            Thread.sleep(dc.getRefreshRateMs());

                  }

              catch (Exception e)

                  {
```

```
                                SimulationOutputter.die("Exception: "+e);
                        }
                }
        }

        private static String[] split(String s) {
                StringTokenizer st = new StringTokenizer(s);
                String[] ret = new String[st.countTokens()];
                for (int i = 0; i < ret.length; i++)
                    ret[i] = st.nextToken();
                return ret;
        }

        private static String readDesc(String[] tokens, int start) {
                String desc = "";
                for (int i = start; i < tokens.length; i++)
                    desc = desc + tokens[i] + " ";
                return desc.trim();
        }
```

```
}
```

```
package hope;


import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


/**
   The StopStartButton class is the Button for pausing and unpausing the
   simulation display.
 */
public class StopStartButton extends JButton {
    private OutputterUI parentOUI;


    private class StopStartButtonListener implements ActionListener {
              private StopStartButton b = StopStartButton.this;


              public void actionPerformed(ActionEvent e) {
                  if (b.getParentOUI().isPaused()) {
                            b.getParentOUI().unpause();
                            b.setText("Pause");
                  } else {
                            b.getParentOUI().pause();
                            b.setText("Resume");
                  }
              }
    }


    /**
       This creates the button.
       @param oui The OutputterUI in which the button is displayed
     */
    public StopStartButton(OutputterUI oui) {
            super("Pause");
            parentOUI = oui;
            this.addActionListener(new StopStartButtonListener());
    }

    /**
       This returns the OutputterUI in which the button is displayed.

       @return The OutputterUI
     */
    public OutputterUI getParentOUI()
    {
            return parentOUI;
    }
}
```

```
package hope;



/**

   The TimeLabel class is a MetricLabel showing the current simulation time

 */

public class TimeLabel extends MetricLabel

{

    private static String text = "Time";

    /**

       This initializes the TimeLabel.



       @param enabled Whether the TimeLabel is initially enabled (displayed on the MetricPanel)

       @param mp      The MetricPanel associated with this TimeLabel

     */

    public TimeLabel(boolean enabled, MetricPanel mp)

    {

            super(text, enabled, text+": 0", mp);

    }
    /**

       Please see {@link MetricLabel}

     */

    public void update()

    {

              setText(text+": "+(int)getMetricPanel().getTime());

    }

}
```

```
package hope;



/**

   The UIAction class is any simulation time-triggered UI event.

 */

public abstract class UIAction {

    private double time;

    private int id;

    private OutputterUI oui;



    /**

       Create the UIAction.



       @param time The simulation time at which the event will be triggered

       @param id   The unique ID of the Entity to be acted upon by the event

       @param oui  The OutputterUI with which the Event is associated

     */

    public UIAction(double time, int id, OutputterUI oui) {

          this.time = time;
```

```
        this.oui = oui;

        this.id = id;

}



/**

  What happens when the event is triggered. This must be overridden by

  and UIAction implementation.

 */

public abstract void execute();



/**

  Getter for the unique ID of the Entity to be acted upon by the event.



  @return The Entity's unique ID

 */

public int getId() {

        return id;

}
```

```
    /**

       Getter for the OutputterUI assocated with this event.



       @return The OutputterUI

     */

    public OutputterUI getOutputterUI() {

              return oui;

    }



    /**

       Getter for the simulation time at which the event will be triggered.



       @return The trigger simulation time

     */

    public double getTime() {

              return time;

    }

}
```

```
package hope;



/**

   The WaitingLabel class is a MetricLabel that says the number of patients

   waiting in the waiting room.

 */

public class WaitingLabel extends MetricLabel

{

    private static String text = "# in Waiting Room";

    /**

       This initializes the WaitingLabel.



       @param enabled Whether the WaitingLabel is initially enabled (displayed on the MetricPanel)

       @param mp      The MetricPanel associated with this WaitingLabel

     */

    public WaitingLabel(boolean enabled, MetricPanel mp)

    {

              super(text, enabled, text+": 0", mp);

    }
    /**
       Please see {@link MetricLabel}
     */
    public void update()
    {
              int waiting = 0;
              for (Entity q : getMetricPanel().getEntities())
                  if (q instanceof Patient && q.getLocation() == 2)
                            waiting++;
              setText(text+": "+waiting);
    }
}
```

```
package hope;



import java.awt.Color;



/**

   The PatientRepresentation class represents any patient in the simulation

   display.

 */

public class WhitePatientRepresentation extends EntityRepresentation

{

    /**

       This sets up the default patient display (6x6 red dots).

     */

    public WhitePatientRepresentation()

    {

              super("Patient", Color.red, 12, "evil.png", 23, -1,

                    "evil2.png", 20, -1, true, new PatientRepresentation());

    }
    /**
       This represents any Entity that is a patient.
     */
    public boolean represents(Entity e)
    {
            return (e instanceof Patient && ((Patient)e).getRace() == "White");
    }
}
```

```
package hope;

import java.awt.Color;

/**
   The PatientRepresentation class represents any patient in the simulation
   display.
 */
public class YoungPatientRepresentation extends EntityRepresentation
{
    /**
       This sets up the default patient display (6x6 red dots).
     */
    public YoungPatientRepresentation()
    {
              super("Patient", Color.red, 12, "evil.png", 23, -1,
                    "evil2.png", 20, -1, true, new PatientRepresentation());
    }
    /**
      This represents any Entity that is a patient.
     */
    public boolean represents(Entity e)
    {
            return (e instanceof Patient && ((Patient)e).getAge() < 40);
    }
}
```