MODERNIZING THE U.S. AIR TRAFFIC CONTROL SYSTEM

THE NEW MEXICO

SUPERCOMPUTING CHALLENGE

FINAL REPORT

APRIL 2, 2008

TEAM 7

ALBUQUERQUE ACADEMY

Project Members:

Punit Shah

Jack Ingalls

Teacher:

Jim Mims

Project Mentor:

Jim Mims

TABLE OF CONTENTS

EXECUTIVE SUMMARY

America's air traffic control system is becoming overloaded. With our national economy and national security relying on this critical infrastructure, a reliable and efficient air traffic control system is necessary. Even now, airports such as O'Hare in Chicago have only 53.49 percent of its flights arrive on time; air traffic issues like concession caused the majority of these delays.

In finding a solution to this dire situation, UPS has proposed a system known as "Continuous Descent Arrivals" to expedite the arrivals procedure. By offloading air traffic control for this process to pilots equipped with specialized instruments, UPS believes that the system could have large efficiency, environmental, and cost benefits.

Our project creates a computational simulation of the arrival process, a major bottleneck for the air traffic control system. Using UPS's "Continuous Descent Arrivals" system as an example, we created a simulation that policy and decision makers could use to test and determine the benefits and drawbacks of new, proposed systems to improve the efficiency and security of our nation's air traffic control structure.

We created an object-oriented model in Java to simulate the two systems under identical conditions. Run from the class *Driver*, the program also used other classes like *Airplane* to create a framework that allowed for full functionality with intuitive inheritance. The simulation modeled specifically what we needed: from the moment when the airplanes enter its holding patterns (where they hold due to heavy airport traffic) to when the airplanes land at the airport. Using two systems to space each aircraft as it approaches the airstrip, we were able to test each of the different air traffic control systems. The current system spaced based on distances, while the UPS system spaced based on time. The program also created string snapshots that contained all of the major information about each plane in the system. These snapshots were used in a separate MATLAB visualization software that we developed to debug and analyze the simulation as well as gain a greater perspective on what is happening in the model at any given time.

The results showed that the present-day air traffic control procedures for airplane arrivals are in fact more efficient. Using visualization software that we developed in MATLAB, we were able to see how models of the present-day and proposed systems diverged after a simulated hour.

However, testing this one solution was only part of our problem. We conclude that we successfully created a framework that can be used to test future solutions. By appropriately applying the principles of object-oriented programming along with effective algorithms to make the program simulation run reliably and efficiently, this program can provide the basis for many future studies. This is certainly an important conclusion of the capabilities of our work and also a very important achievement. With numerous areas to expand upon this project that we list in our report, this idea of a strong framework is ever more valuable.

INTRODUCTION

Our nation's air traffic control system is an important piece of infrastructure supporting our nation's economic and national security goals. As air travel is a key tool to quickly transport cargo, business travelers, and more, an efficient air traffic control system is essential. Additionally, the attacks on 9/11 demonstrated the need for a secure and reliable system of air traffic control for broader national security. Specifically, the 9/11 Commission Report cited our nation's failures in imagination, policy capabilities, and management in many areas of government, including air traffic control. This project fits into the larger puzzle of fixing this system by addressing issues of reliability and efficiency. While the system is facing a number of new challenges, like a large number of retiring controllers over the next decade or implementing the 9/11 Commission recommendations, the ideas presented in this report and project would be valuable in securing aspects of our nation's economy and security.

THE CURRENT AIR TRAFFIC CONTROL SYSTEM

The Federal Aviation Administration (FAA) manages the air traffic control system. Through the course of a flight, a plane passes through a number of phases as outlined in Appendix A, Figure 1. Through the phases, air traffic responsibly of the plane is transferred between numerous controllers.

In order to manage the traffic for the whole nation, the United States is divided into 21 control airspaces, each managed by its own air route traffic control center (ARTCC) (see Appendix A, Figure 2). The ARTCCs control airplanes in the "en route" phase. As planes cross between the various control centers' airspaces, centers pass the flight information and responsibilities onto the next center.

Within each center's airspace, there are TRACON (*T*erminal *R*adar *A*pproach *CON*trol) centers and air traffic towers that control the air space surrounding airports. During planes' ascent and descent phases, the local air traffic towers control the planes' movement. By directing departing and arriving airplanes through specific paths known as air corridors until the planes leave the TRACON's airspace, air traffic controllers are able to better manage the airplanes and prevent crashes in the high-traffic airspaces around airports.

While there may be approximately 5,000 planes above America during peak traffic periods, the current system for managing air traffic during the "en route" phase generally only needs to concentrate on keeping a safe separation between planes and on ensuring planes avoid areas with bad weather. With increasing air traffic, the FAA has reduced the horizontal safe separation distance to 5 mi. at cruising altitude.

While there may be significant traffic in various parts of the country, the bottleneck for air traffic has become the airports themselves. While there is significant space far above in the air to accommodate a number of planes, airports can only have a limited number of arriving and departing flights per minute. The number of available runways dictates this maximum rate.

Under today's system, air traffic controllers are directing planes as they pass into and through air corridors during the departure or arrival sequences. Planes are first directed to the initial approach fixes (IAF), specific points where airplanes congregate before entering the air corridors (see Appendix A, Figure 3). From the IAF, they go to the intermediate fix (IF), where airplanes go into "holding pattern." Here, the planes travel in a circle or oval that takes two to four minutes to travel per revolution. If multiple planes are in holding pattern, than air traffic controllers vertically space each plane by 1000 ft (see Appendix A, Figure 4). Each plane moves down 1000 ft as planes are cleared for landing. When a plane is at the bottom on the holding pattern queue, then it is next in line to be cleared for landing.

After air traffic controllers clear a plane for landing, the airplane exits its holding pattern and proceeds to the runway. Air traffic controllers are responsible for setting the distances between the planes that are making their way to the runway. If one plane encounters an air current that makes it faster or slower, the air traffic controller must compensate and adjust the spacing for all the planes behind it. For obvious safety reasons, this system requires significant spacing between individual planes to account for unforeseen velocity variations and for the time it takes air traffic controllers to correct these variations.

THE PUSH FOR A NEW SYSTEM

While this system has served the nation well for the past decades, with increasing air traffic, airports need to be able to land more planes per minute. John F. Kennedy Airport in New York City is currently handling 100 takeoffs/landings during peak hours even though it was only designed to handle 80. While JFK Airport has tried to apply flight caps during peak hours and use higher fares to encourage people to travel during off-peak hours, this only hurts consumers and detracts from the convenience that air travel is supposed to provide.

New, proposed systems of dealing with landings and arrivals have emerged because of these issues. A system that the FAA has been seriously considering and that we explore in our program was first proposed by the United Parcel Service (UPS). The name of the proposed system is “Continuous Descent Arrivals.” Rather than having TRACON and air tower controllers set the distances for landing and departing planes, UPS proposed that individual planes upgrade their 50-year old, onboard equipment to automatically space the plane based on the velocity and distance of the plane in front so that the planes would really be spaced by time. Initial trials by UPS showed that this system would allow nearly 50-100 gallons of fuel to be saved per flight as pilots can both turn off thrusters and other gas guzzling equipment as well as land a few minutes earlier. With the cost of gas hitting over $100 per barrel, they calculated that this improvement could account for millions of dollars in savings per year for their organization and improvements in the environmental impact of their business.

Other potential benefits of the proposed system include more on-time flights. O'Hare International Airport in Chicago had only 53.49 percent of flights arriving on time in January 2008 according to Department of Transportation statistics. The vast majority of these delays were related to air traffic delays, and only 0.7 percent of the O'Hare flights experienced weather-related delays. Clearly, these types of improvements are vital and will have a very significant impact on the reliability of our air travel system, both today and in the future. Therefore, it is essential that the nation and its government through the FAA fully analyze potential solutions as this project does.

PROBLEM STATEMENT

Our project investigates new solutions to our nation's air traffic dilemma. Specifically, we investigate how the FAA can improve the air traffic procedures associated with arriving and departing planes to make the overall air travel system more reliable. By simulating solutions to this key bottleneck in the air traffic control system, we can determine the benefits and drawbacks of each a potential solution compared to both the current system and to other proposed solutions.

Described in the introduction, we concentrate on the "Continuous Descent Arrivals" solution championed by UPS. Our project thoroughly investigates this solution as an example of a use and benchmark of our simulation. With the model and software we develop, we hope to answer how much the UPS system can improve air traffic efficiency during the arrival phase of a flight (specifically, the time between entering the holding pattern to touching down on the runway). Furthermore, we hope to make this framework expandable so that our solution can be applied to future proposals to solving our country's critical situation regarding air traffic control.

METHOD & COMPUTATIONAL MODEL

BASIC OVERVIEW

In order to achieve our goal of simulating both a model of the current aircraft system and our proposed model, we decided to use a two program system. The first program would be our computational program in Java. We choose Java because it is an object-oriented language, it is easy to find compilers for it, it can be used on multiple platforms, it is fairly easy to program in, and we are more familiar with it than other languages. This allows us to use dynamic data structures as well as other intricate and beneficial features of the language easily without using up time looking through a textbook when we could be improving our algorithm.

This program runs both the current and proposed model. It simulates an hour at an airport where airplanes are randomly received at the holding fix, allowed to begin approach, and finally land at the airport. Even though the airplanes come to the airport randomly, the times that the airplanes arrive are saved so that each model runs the same simulation. This was implemented so that there aren't any anomalies causing one of the models to perform better when in reality it would have performed worse in either situation.

After the data is displayed on the screen, it is copied, saved, and entered into our visualization program. This program, written in MATLAB, provides us with an important visual representation of what is happening in the simulation. It is difficult to understand what is truly going on if all one sees is numbers displayed from a program, while a screen that shows were all the airplanes are and where they are going really lets people understand what the program is doing and what it is telling us. We used MATLAB because of its ease with three dimensional graphics.

Each of these components to the program are discussed in the following subsections.

COMPUTATIONAL JAVA SIMULATION

At the heart of our project is the Java simulation. It consists of 6 classes: *Airplane*, *Airport*, *Debugger*, *Driver*, *Program*, and *Runway*. *Debugger* was used to test some other classes to make sure they worked properly. *Runway* and *Airport*, are classes implemented in the program, but they are mostly for generalization purposes and will not be thoroughly discussed, through are important for applying our simulation outside of this single test of the UPS system. *Program* is a class that all the other inherits from, and it has some nice functions that all the classes might use like a random integer generator and calculations for how fast an airplane may travel in certain areas (as well as other related methods). The main classes are *Airplane*, which gives all the information about the airplane as well as what it can do, and *Driver*, which contains the main method and runs the simulation.

Because we have a simulation, the program has to define a time interval in which to run. In doing so, we decided that the program should run during precisely 1 hour and run on time-steps equivalent to 1 second each. Our original program had time steps of 15 seconds, but noticing that the program ran almost instantaneously, we decided to make it more precise with 1 second time-steps, giving greater precision to our outputs.

When the program runs, its first action is to create 30 random “times” in the range of 0 to 3600 (including 0, excluding 3600). This is equivalent of the program choosing 30 different times within the course of an hour. These times will then be sorted using an insertion sort method so that they can be used in the program’s model methods as the times when airplanes first arrive at the holding fix. Each model then runs almost completely inside one for loop which goes from 0 to 3600 (excludes 3600) to essentially run through an hour of aircraft traffic. We define each cycle through this for loop to be one time step.

Next, the program runs a simulation of what will happen if the current model for aircraft landing is used. Here, aircraft will wait in the holding pattern above the fix until they are cleared to begin landing (meaning that the aircraft in front of them is at least 5 nautical miles away). The method begins by seeing if an aircraft arrives or not. If an aircraft is supposed to arrive, a new object is placed at the beginning of the linked list of aircraft and plane is placed either at 3000 feet in the holding pattern above the fix (which is 10 nautical miles away from the airport) if there are no other aircraft there, or 1000 feet above the highest plane at the fix if aircraft are present. The planes will then begin to approach once as they are cleared (5 nautical miles away from plane in front of them) and the planes above that one will descend to fill in the gap below.

At this point in running the program, the planes are descending in their holding pattern. The program calls up each plane one-by-one starting with the one closest to land. It sets this plane to begin its approach to the runway while moving each of the other planes down in altitude. A method in the *Airplane* class called *hasClearance()* returns true if the plane can approach and false if it cannot. If it returns false, the plane stays in its holding pattern, but if it returns true, then the plane will move forward. The program then calculates precisely how far the plane may move in one second and how far it descends. The altitude (in feet) is always 300 times the number of nautical miles the plane is away from the airport. This gives a steady descent that is reasonable with given real life parameters.

The plane maintains a safe distance away from the one in front of it, and it will slow down if necessary. It will also try to go the speed that is designated inside of its zone. These speeds are: 200 knots for 5-10 nautical miles away from the airport, 170 knots for 3-5 nautical miles, 160 knots for 2-3 nm, 150 knots for 1-2, and 140 knots within 1 nautical mile. This insures that the planes travel at safe speeds and approach evenly. If a plane lands at any time, it is immediately removed from the linked list because it is on the ground and not affecting our simulation.

While all of this is going on, a method called *makeMATLABMatrix(Airplane plane, int timeStepNumber)* is called every 30 time steps. This method takes all the information of the planes and outputs it in a string that can then be copied into our visualization program so that we can see what is going on for that time step.

These above state procedures describe the current air traffic control system. The UPS system follows the majority of these steps as well with one key difference. Instead of making spacing calculations so that the aircraft are always 5 nautical miles apart, the UPS system's simulation makes sure that the planes are always 2 minutes apart, our assumption for the maximum capacity of the runway. Therefore, the only methods that change are the *hasClearance()* method and the *approach()* method since they both are affected by how far the next plane is from the current plane. This key difference reflects the basic principle differentiating the current system to the UPS system, according to our background research.

The program then gives us the data we need to input into our MATLAB program to see if the backups occur more in the current system or the UPS system.

ASSUMPTIONS AND THEIR EFFECTS ON THE SIMULATION

In order to construct the program, we made many assumptions and simplifications that we were able to reason would not significantly affect our results. First, planes that are outside of the system – i.e. aircraft that are taking off, en-route, approaching the fix, etc. – are not critical to our program as these aircrafts do not interfere with the arrival procedures and the proposed Continuous Descent Arrival system. Similarly, once the aircraft are on the runway, they no longer interfere with the arrival system and thus are removed from the simulation. Even the movement of planes within their holding pattern doesn't matter, so our program doesn't model those movements and just makes the planes in the holding patter stationary for our purposes. Ultimately, the only planes that are given a vector for movement are the planes that have been given the authorization to land and are directly approaching the airstrip.

Other minor assumptions were made about the approach (like the precise speed of the aircraft, how long it takes pilots to react to commands from both systems, variations from the exact course, weather, etc.), but those assumptions were made in simulating both the new and old air traffic control systems and thus should not positively or negatively affect their relative performance. Many of these assumptions were based on research including FAA manuals anyway, and thus are close approximations. We also assumed the simplest model for our airport: one with no weather, situated in a flat area at sea level, containing only one runway which is due north, and possessing a standard T-shape design fix pattern. For our purposes, these are all safe assumptions, and most of the assumptions can be generalized by just rotating and translated our grid system. Overall, the simulation we made is a fairly good indicator of what would happen in reality as long as we remember that we are only simulating the planes between their entrances into their holding pattern to the moment their tires touch the runway.

VISUALIZATION OF SIMULATION

In order to benchmark, test, and fully understand the nature of our program's outputs, we created a program in MATLAB to display planes in relation to the airport. By seeing a visualization, we can quickly and easily see if planes are crashing with each other without being detected in the simulation and other such anomalies as well as garner a greater understanding of how each solution we test works.

We used MATLAB for the output due to its numerous built-in packages that allowed for us to create an elegant graphing solution. For example, when creating a 3D figure in MATLAB, there are numerous tools built in to change camera position relative to the graph, zoom, and lighting. Each of these manipulations done in real-time gives the user a very accurate perception about what is going on at a specific time step.

The *grapher.m* MATLAB function graphs each plane as a vector arrow and graphs a dot for the location of the airport (see an example output in Appendix B, Figure 3 or 4). The graph is time-dependant; in other words, the graph changes for each time step and only offers a snapshot of a specific time step. While we explored outputting the simulation to a video format, we found the technical constrains to be significant for first-time MATLAB programmers. Additionally, with a constantly changing graph, the viewer would not be able to see what is going on a specific time because it takes a few rotations and views to fully understand where each plane is in 3-space.

Grapher takes four inputs: the airport coordinates, the planes' coordinates, the handle for the figure, and the time step number as a string. The last two inputs are used to output the graph to the correct figure with an appropriate title. The airport coordinates are constant for every time step and are inputted as a 1x3 matrix in the form [x y z] were (x,y,z) is the location of the airport.

The Java-based simulation outputs a string in a specific form that complies with MATLAB's syntax for the matrix input representing the planes' coordinates. Two points define each airplane's arrow: the point at the base of the vector arrow, (x, y, z), and the point at the head of the vector arrow, (u, v, w). If $plane_1$'s vector is from (x_1,y_1,z_1) to (u_1,v_1,w_1), $plane_2$'s vector is from (x_2,y_2,z_2) to (u_2,v_2,w_2), ..., and $plane_n$'s vector (where n is the index for the last plane) is from (x_n,y_n,z_n) to (u_n,v_n,w_n), then the format of the string outputted by the Java simulation is:

$$[x_1\ y_1\ z_1\ u_1\ v_1\ w_1;\ x_2\ y_2\ z_2\ u_2\ v_2\ w_2;\ ......;\ x_n\ y_n\ z_n\ u_n\ v_n\ w_n]$$

This matrix ultimately looks like the one shown in Appendix A, Figure 5. The numbers between each semicolon form individual, successive rows of the matrix.

By copying this MATLAB syntax compliant string into the MATLAB command window and setting a variable equal to the matrix that the string represents, we avoid any data entry mistakes in trying to manually input each plane's coordinates and save time to concentrate on analyzing the data and visualization.

After clearing and setting up the visual styles of the figure that it will output to, *grapher* uses the built-in *scatter3* function to plot the yellow dot representing the airport.

Afterwards, the planes are graphed onto the same figure. Using a for loop, *grapher* extracts the positions of each plane from the inputted matrix representing the airplane vectors. While we intend to graph each plane as a vector, planes in holding pattern are better represented by dots because they remain in very close proximity to a specific position and we would like to visually differentiate between planes in their difference phases of motion. Therefore, *grapher* first determines if the plane is in holding pattern. In our matrix, we set the arrow's base and head points to be the same if the plane is in holding pattern. If the planes are in holding pattern, we graph the dot using the *scatter3* function.

If the plane is not in holding pattern, then we graph it as an arrow using the *quiver3* function. We directly input the base arrow point into *quiver3* and calculate the difference between the inputted arrowhead point and base point, passing the difference to *quiver3*; the function requires the relative (rather than absolute) position of the arrowhead point compared to the base point. Having set the “hold” option for the visualization we are creating to “on,” each of the visual elements, namely each plane we extract from the inputted matrix, are added onto the figure rather than replace all of the previous contents as is the default. This allows us to construct a single visualization containing all of the airplanes together.

RESULTS & ANALYSIS

The results are nicely summarized in Figure 1 and 2 in Appendix B. Figure 1 represents the current air control system shown at the time step 3570 (I.e., 59 minutes and 30 seconds). As described in the caption, the arrows represent the planes, the big yellow dot at (0, 0, 0) represents the airport, and the other dots represent the various planes in holding pattern. Figure 2 represents the Continuous Descent Arrivals system proposed by UPS also shown at time step 3570.

In comparing these two figures, we ensured that the same colored arrow represents the same airplane. By analyzing the figures, it appears that the current system is ahead of the UPS system. Comparing Figure 1 to Figure 2, we can see that in the current air traffic system simulation, the white plane has already landed, the blue plane is further along, the green plane has already been given clearance to land, and its planes in holding pattern are at a lower altitude, indicating that they are going to land sooner. After running the computational model for a simulated 59 minutes, 30 seconds, it is clear that there is a significant divergence between the present-day and proposed air traffic control systems.

CONCLUSIONS

In general, our project was successful in achieving its objectives and allows us to draw valuable conclusions. These conclusions include the following:

- In the simulation, the present-day air traffic control system actually seemed to perform better than the Continuous Descent Arrivals system. As described in the Results & Analysis section, the simulation for the current air traffic system seems to be significantly ahead of the UPS system after a simulated hour. Thus the current system is more efficient in terms of airplanes landed per hour. However, this does *not* commend about the reliability of the systems. Human error was never a factor in either system, and thus we have no way of determining whether which system is more reliable when the "human factor" is added.
- The simulation seemed to accurately model the key components of air traffic arrival procedures. We intended to model planes from when they reached the intermediate fix (where planes go into their holding pattern) to the time when their tires touched the ground. The airplanes in our model seemed to follow the expected patterns as outlined in the Federal Aviation Administration's *Aeronautical Information Manual*, where the procedures on air traffic control are published. This benchmarking of our outputs is a key step that helps make our project and its conclusion relevant and useful.
- The simulation worked to compare two different air traffic control system. The quickness and efficiency of a computer are key difference between our simulations and the real world. While the two air traffic control systems we tested can play out differently in the real-world, often the speed of a computer could have made the control systems appear infinitely efficient. Nevertheless, our simulation produced results that, based on the rules established by each air traffic control system, were significantly different for separate control systems.

- The framework we have established in our software is expandable and can be enhanced for professional applications beyond the Supercomputing Challenge. Apart from simple practices like documentation and well-named variables and methods, the basic framework utilizing objects, such as for airplanes, has allows us to expand to program to account for the factors between the air traffic control systems. These are important capabilities to give this project relevance outside of the Supercomputing Challenge, and the results of working in this framework show that our organization works.

These results and conclusions are validated by the visualization environment that displays our findings in an easy to understand and approach format. This format clearly worked to test and debug the program to ensure that our findings were both reasonable and are possible and helpful in the real world.

Achievements of the Project

Our project was able to effectively test the arrival procedures of a proposed solution to our impending air traffic predicament. As discussed in the introduction, the necessity of efficient and reliable air traffic control are important for a safe and secure nation. This project took the first step towards those goals by providing a computational simulation that not only tested the Continuous Descent Arrivals technique against the current system, but also provided a robust, flexible framework for future tests.

While programming, we proved, mostly to ourselves, that even if one doesn't ultimately use everything that was in an original program and framework, the ability to choose where to go later on makes new programs and accomplishments come with ease. This ability to modify and generalize our program and not worry about the smaller details like linking the planes together in a list allowed and will allow us to program quickly and effectively. So even though there are many variable, methods, comments, and debugging algorithms that we never used in our end program, we still kept those so that future expansion could be done easier and more efficiently.

The Continuous Descent Arrivals system certainly won't be the last proposed solution to our country's air traffic dilemma. By providing this framework, our program can be used for new tests and in new contexts. This is a major achievement.

RECOMMENDATIONS

As we feel this framework is such a critical achievement, we feel it is also important to list a few recommendations and suggestions on how to build upon our system. Our simulation, while useful in testing the Continuous Descent Arrival system and in establishing a basis for future work, has certain important limits. With these recommendations, we can test many more and broader cases and analyze more specific problems and issue when comparing possible air traffic control systems.

One direction we can take the project from here is to determine the performance differences of systems between peak and non-peak hours. By varying traffic, we may find some solutions are best during rush hours, while other solutions work better during off hours. This could help dictate decision as solutions that appear to improve the system overall may fail at the critical peak hours when the real traffic-related issues arise. Other variations that we can test include how bad weather, turbulence, and other factors that influence a pilot's ability to effectively control their plane could affect the particular air traffic control system's viability for general use.

Another possible direction that would require much more background research would be to take into account the "human factor." By creating a sort of artificial intelligence engine that errors at rates similar to humans, we can see how robust each air traffic control system would be in correction errors. If a system completely failed even after the most trivial of errors, we would see that the reliability of the system is low and would involve to great of a risk for use in the real world.

One last recommendation is to modify the spacing between aircraft and see if safety and reliability are still within standards. With even a slight reduction we may be able to reduce delays tremendously due to a snowball effect (every second more a plane takes to land means a second for the one behind it as well, and then another second or two may be added there until the system fails). A simulation like this would most certainly prove to be of great use to Air Traffic.

Acknowledgements & Citations

We would like to thank our faculty sponsor and mentor Jim Mims. He provided guidance when necessary, but also provided the independence and leeway necessary to explore, learn, and succeed.

We also appreciate all the help, guidance, and support from the everyone on the Supercomputing Challenge's staff and all of the volunteers who have helped with the Challenge. From their constructive comments during the February project evaluation to their dedication in organizing the Challenge, we truly appreciate all their time and effort.

Citations

References on Ramsey Numbers

The 9/11 Commission Report by The National Commission on Terrorist Attacks Upon the United States, (© 2004, Norton, New York)

Aeronautical Information Manual: Official Guide to Basic Flight Information and ATC Procedures by The Federal Aviation Administration, February 14, 2008 (http://www.faa.gov/airports_airtraffic/air_traffic/publications/media/aim.pdf)

Howstuffworks, "How Air Traffic Control Works" (http://www.howstuffworks.com/air-traffic-control.htm)

New York Times, September 5, 2007, "For Airlines, Hands-On Air Traffic Control"

New York Times Video, "Managing Air Traffic Control," (http://video.on.nytimes.com/?fr_story=ddf6ffdff959f2da930edb4d711ad7c3fe497e40)

SPACE.com, "Space-Based Air Traffic Control System" (http://www.space.com/businesstechnology/airlines_crowding_000804.html)

U.S. Department of Transportation, Research and Innovation Technology Administration, Bureau of Transportation Statistics (http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp?pn=1)

REFERENCES ON ALGORITHMS AND PROGRAMMING

Fundamentals of Java, Second Edition by Kenneth Lambert and Martin Osborne (© 2003 Thompson Learning)

Online MATLAB Documentation by The MathWorks (http://www.mathworks.com/access/helpdesk/help/helpdesk.html)

APPENDIX A: FIGURES FROM THE REPORT

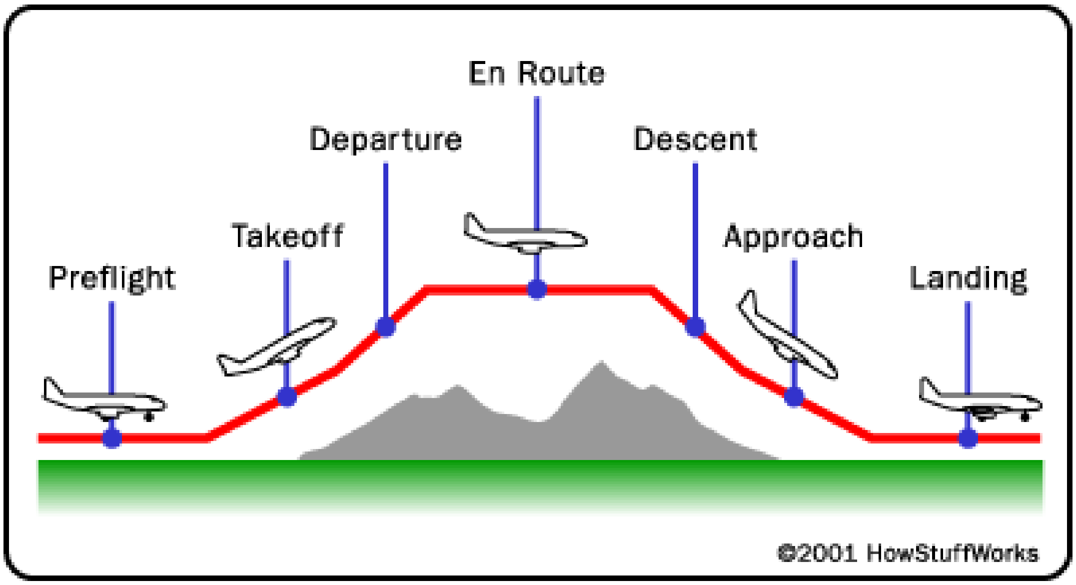

Figure 1: The various phases of a flight. Source: HowStuffWorks.com

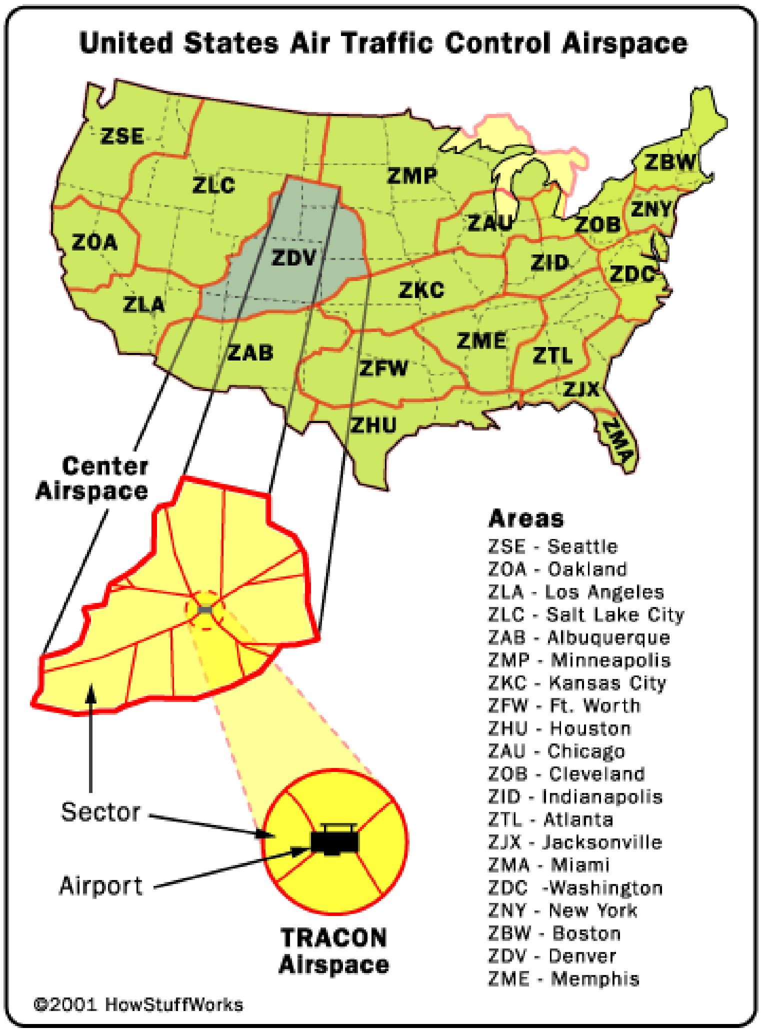

Figure 2: The Air Traffic Control Centers' zones and codes for the United States. Source: HowStuffWorks.com

TAA Area

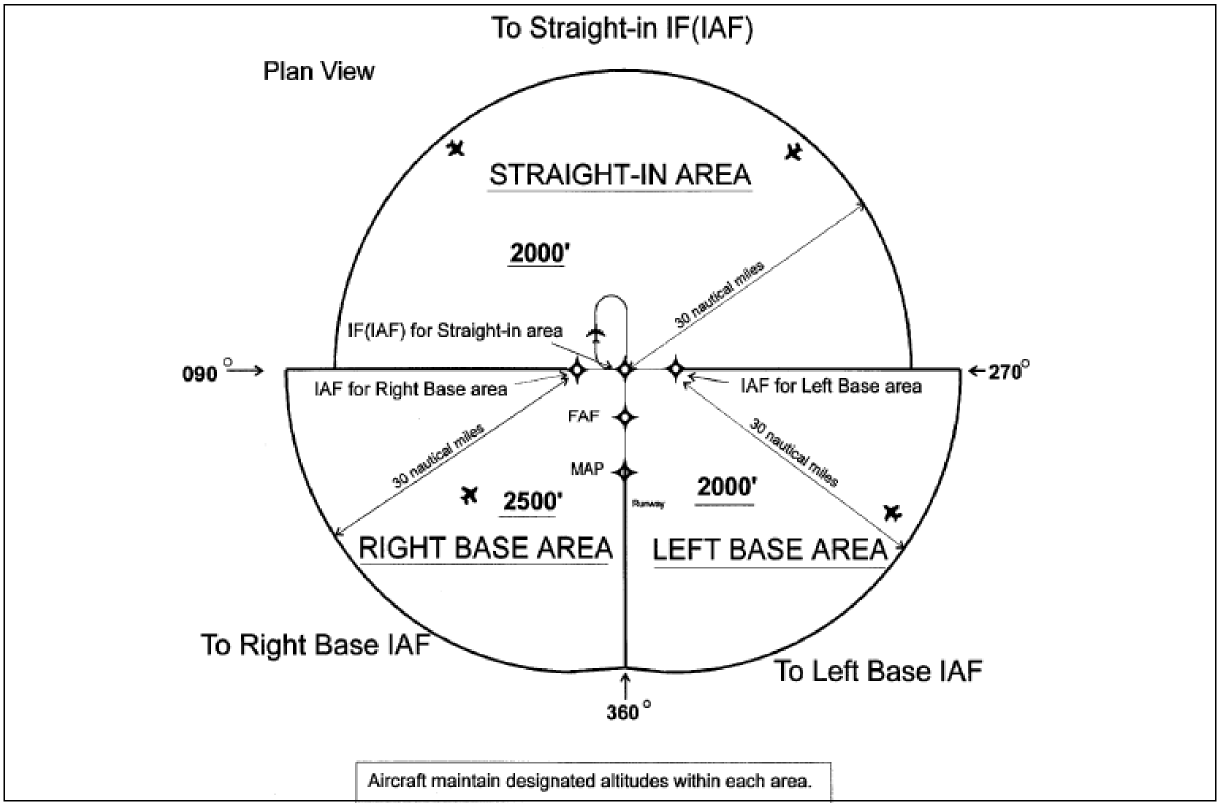

Figure 3: The radar fixes are represented by the stars. Planes converge at the various initial approach fixes (IAF) depending on where they are coming from, but eventually go to the intermediate fix (IF), where holding patterns take place. Source: FAA Aeronautical Information Manual

Timed Approaches from a Holding Fix

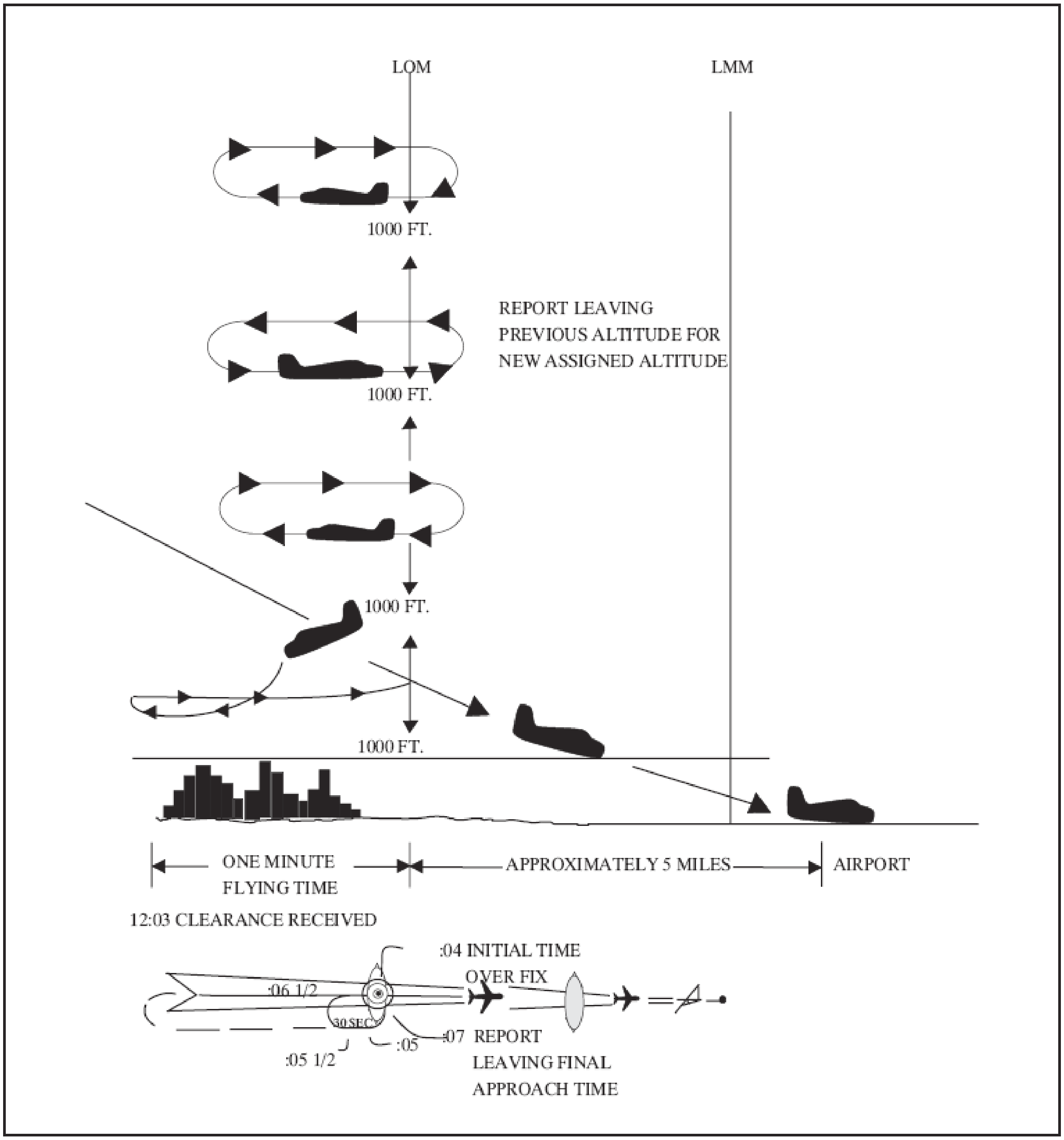

Figure 4: This diagram shows various planes in their holding pattern. Each are separated by 1000 ft in altitude. Source: FAA Aeronautical Information Manual

$$\begin{bmatrix} x_1 & y_1 & z_1 & u_1 & v_1 & w_1 \\ x_2 & y_2 & z_2 & u_2 & v_2 & w_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_2 & y_2 & z_2 & u_2 & v_2 & w_2 \end{bmatrix}$$

Figure 5: The matrix outputted by the Java-based simulation to input airplane positions, directions, and velocities into the MATLAB visualization software.

APPENDIX B: PROGRAM OUTPUTS

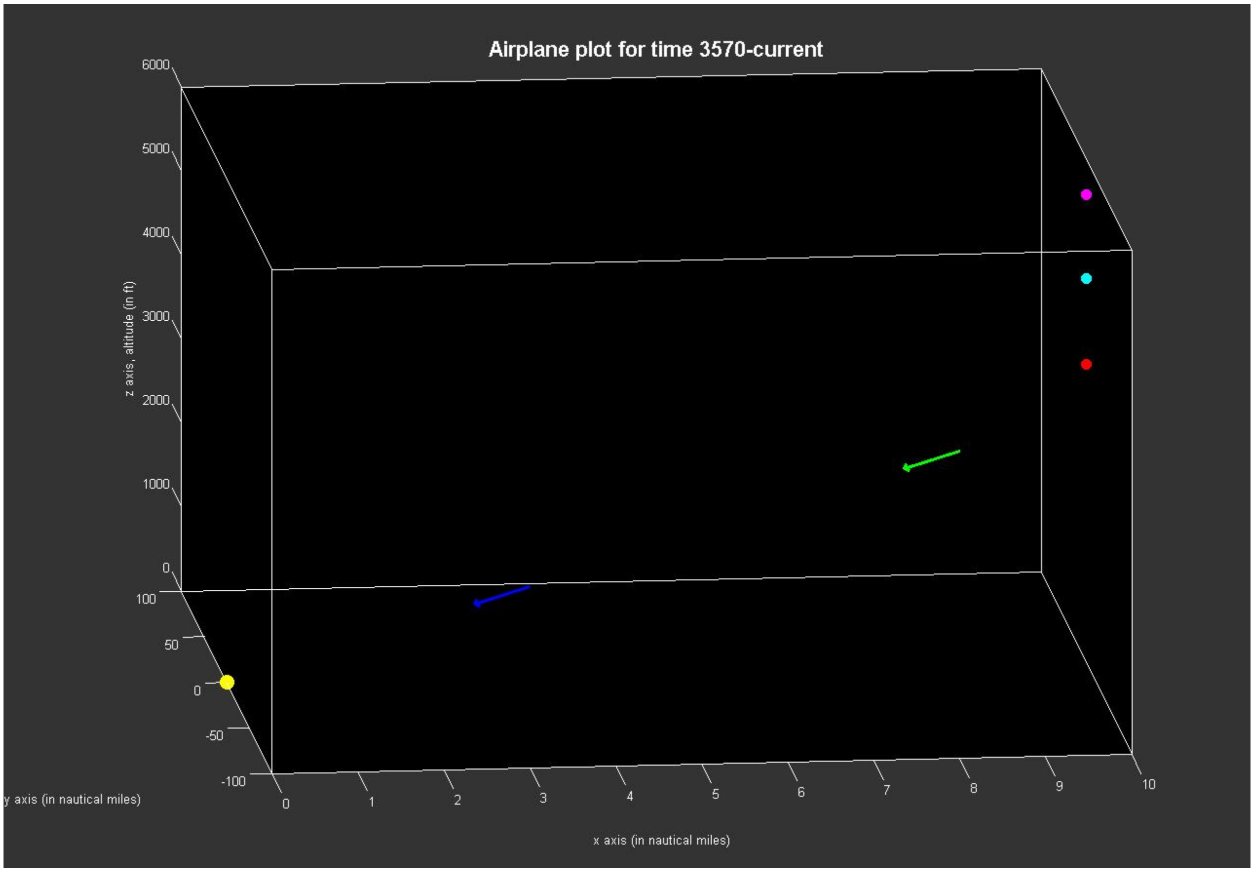

Figure 1: Output of airplane vectors in the MATLAB visualization at time step 3570 using the present-day air traffic control system. The arrows each represent and airplane and its respective location, direction, and velocity. The yellow dot at (0, 0, 0) represents the location of the airport. The dots along the right represent various planes in their holding pattern.

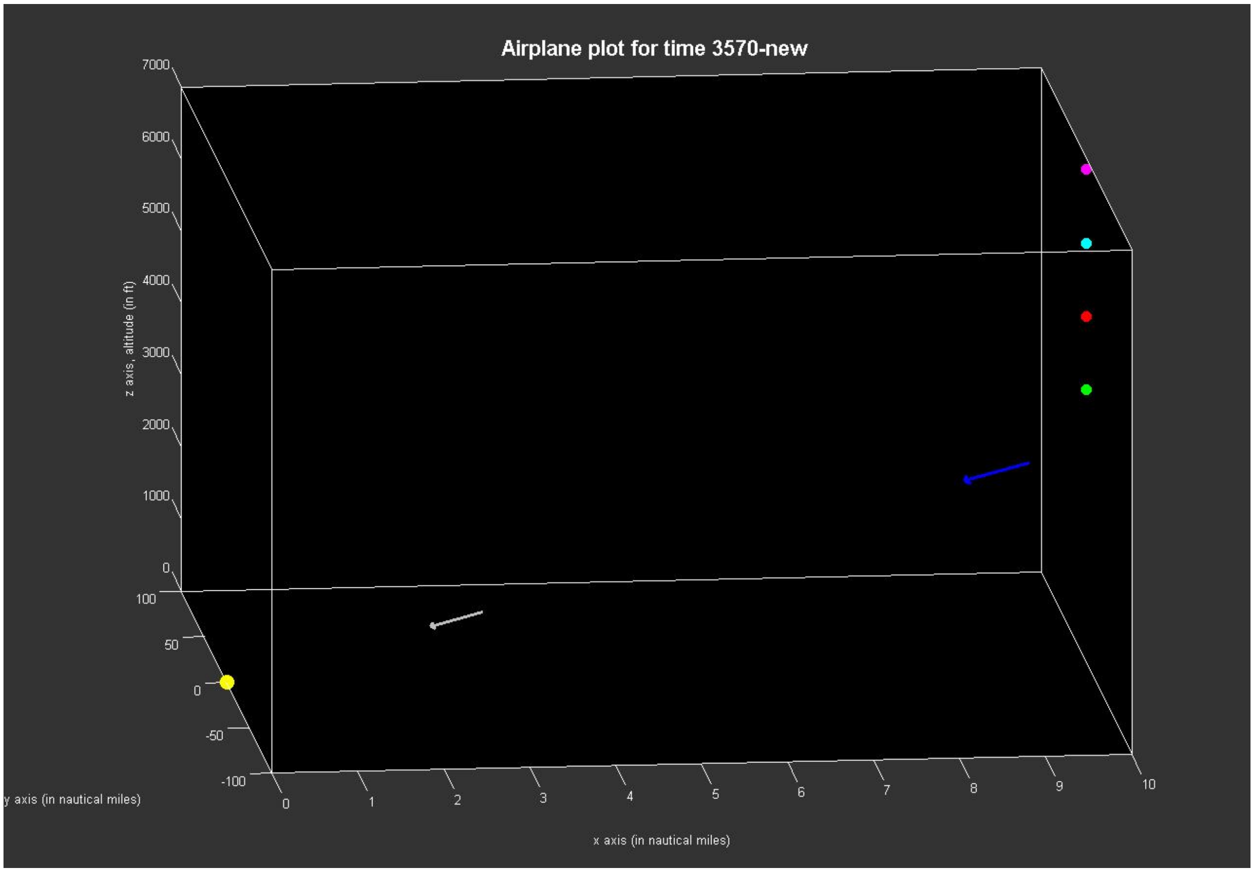

Figure 2: Another output of airplane vectors in the MATLAB visualization. While this visualization is at time step 3570 like in Figure 1, it uses UPS's "Continuous Descent Arrivals" system. In comparing the two graphs, note that the axis measure are slightly different and each arrow colors represent the same plane between the two figures. Thus, the dark blue plane is much further along in landing in the present-day system and the green plane is still in its holding pattern in the UPS system.

APPENDIX C: CLASS *DRIVER*

```
/*****************************************************************************
 * Modernizing the U.S. Air Traffic Control System
 * Super Computing 2008
 * Team 7
 * Jack Ingalls & Punit Shah
 * v1.1a

*****************************************************************************/

import java.util.*;

public class Driver extends Project
{
	public static void main(String[] args)
	{
		int n = 30;	//	Number of total airplanes
		int times [] = new int [n];
		populateTimes(times);

		System.out.println("\n Beginning model for the current Air Traffic Control
System.");
		runModel(times, false);

		System.out.println("\n Beginning model for the proposed Air Traffic Control
System.");
		runModel(times, true);

		System.out.println("DONE!!!");
	}

	public static void runModel (int times [], boolean newMethod)
	{
		int planeNum = 0;
		Airplane firstPlane = null;
		Airplane currentPlane = firstPlane;
		Airplane prevPlane = null;

		for (int time = 0; time < 3600/* || firstPlane != null*/; time ++)
		{
			while (planeNum < times.length && time == times[planeNum])
			{
				if (firstPlane != null && firstPlane.isAtFix())
				{
```

```
					firstPlane = new Airplane(1000 + firstPlane.getAltitude(),
firstPlane, newMethod);		//	Creates a new plane at the holding point.
				}
				else
				{
					firstPlane = new Airplane(3000, firstPlane, newMethod);
	//	Creates a new plane at the holding point.
				}
				planeNum ++;
			}

			currentPlane = firstPlane;
			prevPlane = null;

			while (currentPlane != null)
			{
				if (currentPlane.getXCord() == 10)
				{
					if (currentPlane.getAltitude() > 3000)
					{
						if ((currentPlane.getAltitude() -
currentPlane.getNext().getAltitude()) > 1000)
						{
							currentPlane.descend();
						}
					}
					else
					{
						if(currentPlane.hasClearance())
						{
							currentPlane.approach();
						}
					}
				}
				else
				{
					currentPlane.approach();
				}

				if (currentPlane.getXCord() == 0)
				{
					if (prevPlane != null)
					{
						prevPlane.setNext(null);
					}
					else
```

```
                {
                    firstPlane = null;
                }
            }
            else
            {
                prevPlane = currentPlane;
            }
            currentPlane = currentPlane.getNext();
        }
        if (time % 30 == 0)
        {
            makeMATLABMatrix(firstPlane, time);
        }
    }
}

public static void populateTimes (int times [])
{
    Random rand = new Random();
    int temp = 0;
    int j = 0;

    for (int i = 0; i < times.length; i++)
    {
        temp = rand.nextInt(3600);
        j = i;
        while (j > 0 && times[j - 1] > temp)
        {
            times[j] = times[j - 1];
            j--;
        }
        times[j] = temp;
    }
}

public static void outputTimes (int times [])
{
    for (int i = 0; i < times.length; i++)
    {
        System.out.println(times[i]);
    }
}

public static void makeMATLABMatrix (Airplane inputPlane, int timeStepNumber)
{
```

```
        Airplane plane = inputPlane;
        double u;
        double v;
        int w;

        System.out.println ("Time step: " + timeStepNumber);
        System.out.print ("[");

        while (plane != null)
        {

            if (plane.hasClearance())
            {
                u = plane.getXCord() - plane.getSpeed() / 240.0;
                v = 0;
                w = (int)(u*300);
            }

            else
            {
                u = plane.getXCord();
                v = plane.getYCord();
                w = plane.getAltitude();

            }

            System.out.print (plane.getXCord() + " " + plane.getYCord() + " " +
plane.getAltitude() + " ");
            System.out.print (u + " " + v + " " + w + "; ");

            plane = plane.getNext();


        }

        System.out.println ("]");

    }
}
```

APPENDIX D: CLASS *AIRPLANE*

```
/*****************************************************************************
 * Modernizing the U.S. Air Traffic Control System
 * Super Computing 2008
 * Team 7
 * Jack Ingalls & Punit Shah
 * v1.1a

*****************************************************************************/

public class Airplane extends Project
{
	private int speed;	//	Expressed in miles/hour
	private int heading;	//	Expressed in degrees, values range from 0 to 359.
	private int altitude;	//	Expressed in feet
	private Airport closestAirport;
	// private String flightNumber;	//	Found unnecessary.
	// private int bearing;	//	Found unnecessary.
	private double x;	//	x component of position in nautical miles
	private double y;	//	y component of position in nautical miles
	private Airplane next;
	private boolean newMethod;

	public Airplane()
	{
		setSpeed(0);
		setHeading(0);
		setAltitude(2000);
		x = 10;
		y = 0;
		next = null;
	}

	public Airplane(int a, Airplane nextPlane, boolean NM)
	{
		setSpeed(200);
		setHeading(0);
		setAltitude(a);
		x = 10;
		y = 0;
		setNext(nextPlane);
		newMethod = NM;
	}

	public Airplane(String flight, int s, int h, int a, Airport port)
```

```
{
	setSpeed(s);
	setHeading(h);
	setAltitude(a);
	closestAirport = port;
	x = randomInt(0, 100);
	y = randomInt(0, 100);
	next = null;
}

public boolean isAtFix()
{
	if (x == 10 && y == 0)
	{
		return true;
	}
	return false;
}

public void takeOff()
{
	setSpeed(140);
	ascend();
	System.out.println();
	System.out.println("Aircraft took-off.");
	stateStatus();
}

public void land()
{
	setAltitude(closestAirport.getAltitude());
	setSpeed(0);
	System.out.println();
	System.out.println("Aircraft landed.");
	stateStatus();
}

public boolean hasClearance()
{
	if (next == null)
	{
		return true;
	}
	else
	{
		if (newMethod == false && x - next.getXCord() >= 5)
```

```
				{
					return true;
				}
				if (newMethod == true && next.getXCord() <= 43.0 / 12.0)
				{
					return true;
				}
			}
			return false;

		}

		public void approach()
		{
			if (newMethod == false)
			{
				if (next != null)
				{
					double newNextX = next.getXCord() - next.getSpeed() / 3600.0;
					int newNextSpeed = speedAtDist(newNextX);
					newNextX = next.getXCord() - newNextSpeed / 3600.0;
					if (newNextX <= 0)
					{
						double nextAriveTime = next.getXCord() / newNextSpeed;
		//	In hours
						setSpeed (checkedSpeed((int)((x - 5) / nextAriveTime), x));
					}
					else
					{
						setSpeed (checkedSpeed((int)((x - newNextX - 5) * 3600),
x));
					}
				}
				else
				{
					setSpeed(speedAtDist(x));
				}
			}
			else
			{
				setSpeed(speedAtDist(x));
			}

			if (differentSpeeds(x, x - speed / 3600.0))
			{
				double remainingTime = 1 / 3600.0 - (x - (int)x) / speed;
```

```
            x = (int)x;
            setSpeed(speedAtDist(x));
            x = x - remainingTime * speed;
        }
        else
        {
            x -= speed / 3600.0;
        }

        if (x < 0)
        {
            x = 0;
        }

        altitude = (int)(x * 300);
    }

    public void stateStatus()
    {
        System.out.println();
        System.out.println("We are travelling at " + speed + " miles per hour");
        System.out.println("in a heading of " + heading + " degrees");
        System.out.println("at an altitude of " + altitude + " feet.");
    }

    public void turn(int degrees)
    {
        setHeading(heading + degrees);
        System.out.println();
        System.out.println("Aircraft turned " + degrees + " degrees.");
        stateStatus();
    }

    public void ascend()
    {
        setAltitude(altitude + 20);
    }

    public void descend()
    {
        setAltitude(altitude - 20);
    }

    public void setHeading(int newHeading)
    {
        heading = newHeading % 360;
```

```
}

public void setAltitude(int newAltitude)
{
	if (newAltitude >= 0)
	{
		altitude = newAltitude;
	}
	else
	{
		throw new RuntimeException ("Altitude cannot be negative.");
	}
}

public void setSpeed(int newSpeed)
{
	if (newSpeed >= 0)
	{
		speed = newSpeed;
	}
	else
	{
		throw new RuntimeException ("Speed cannot be negative.");
	}
}

public int getSpeed()
{
	return speed;
}

public int getHeading()
{
	return heading;
}

public int getAltitude()
{
	return altitude;
}

public Airport getAirport()
{
	return closestAirport;
}
```

```
	public double getXCord()
	{
		return x;
	}

	public double getYCord()
	{
		return y;
	}

	public Airplane getNext()
	{
		return next;
	}

	public void setNext(Airplane nextPlane)
	{
		next = nextPlane;
	}

	public double Distance()
	{
		return Math.sqrt(x*x + y*y);
	}
}
```

APPENDIX E: CLASS *AIRPORT*

```
/****************************************************************************
 * Modernizing the U.S. Air Traffic Control System
 * Super Computing 2008
 * Team 7
 * Jack Ingalls & Punit Shah
 * v1.1a

****************************************************************************/

public class Airport extends Project
{
	private Runway myRunway;
	private int altitude;

	public Airport()
	{
		altitude = 0;
	}

	public Airport(int A)
	{
		altitude = A;
	}

	public Airport(Runway Run)
	{
		altitude = 0;
		myRunway = Run;
	}

	public Airport(int A, Runway Run)
	{
		altitude = A;
		myRunway = Run;
	}

	public void stateStatus()
	{
		System.out.println();
		System.out.println("This is the airport.");
		System.out.println("We are at an altitude of " + altitude + " feet.");
		System.out.println(myRunway.getDesignation() + " is cleared for landing.");
	}
```

```
	public Runway getRunway()
	{
		return myRunway;
	}

	public int getAltitude()
	{
		return altitude;
	}
}
```

APPENDIX F: CLASS *RUNWAY*

```
/****************************************************************************
 * Modernizing the U.S. Air Traffic Control System
 * Super Computing 2008
 * Team 7
 * Jack Ingalls & Punit Shah
 * v1.1a

****************************************************************************/

public class Runway extends Project
{
	private int direction;	//	Expressed in degrees from 0 to 179
	private String designation;

	public Runway()
	{
		direction = 0;
		setDesignation(direction);
	}

	public Runway(int dir)
	{
		direction = dir % 180;
		setDesignation(direction);
	}

	private void setDesignation(int degrees)
	{
		designation = "Runway " + degrees;
	}

	public int getDirection ()
	{
		return direction;
	}

	public String getDesignation ()
	{
		return designation;
	}
}
```

APPENDIX G: CLASS *DEBUGGER*

```
/*****************************************************************************
 * Modernizing the U.S. Air Traffic Control System
 * Super Computing 2008
 * Team 7
 * Jack Ingalls & Punit Shah
 * v1.1a

*****************************************************************************/

public class Debugger
{
	public static void main(String[] args)
	{
		boolean bugs = false;
		Runway testRunway = new Runway();
		Airport testPort = new Airport(testRunway);
		testPort.stateStatus();
		Airplane testPlane = new Airplane(testPort, 2000);
		testPlane.stateStatus();
		testPlane.approach(testRunway);
		testPlane.land();
		testPlane.turn(180);
		testPlane.takeOff();
		testPlane.ascend();
		testPlane.stateStatus();

		statusOfProgram(bugs);
	}

	public static boolean checkPlane(Airplane plane)
	{
		return false;
	}

	public static boolean checkPort(Airport port)
	{
		return false;
	}

	public static void statusOfProgram(boolean bugs)
	{
		if (!bugs)
		{
			System.out.println();
```

```
				System.out.println("Everything checks out.");
				System.out.println("No bugs were found.");
			}
		}
	}
```

APPENDIX H: CLASS *PROJECT*

```
/*****************************************************************************
 * Modernizing the U.S. Air Traffic Control System
 * Super Computing 2008
 * Team 7
 * Jack Ingalls & Punit Shah
 * v1.1a

*****************************************************************************/

import java.util.*;

public class Project
{
	Random rand;

	public Project()
	{
		rand = new Random();
	}

	public int randomInt(int min, int max)		//	inclusive min, exclusive max
	{
		return min + rand.nextInt(max - min);
	}

	public int speedAtDist(double dist)
	{
		if(dist > 5)
		{
			return 200;
		}
		else if (dist > 3)
		{
			return 170;
		}
		else if (dist > 2)
		{
			return 160;
		}
		else if (dist > 1)
		{
			return 150;
		}
		else
```

```
        {
            return 140;
        }
    }

    public boolean differentSpeeds (double x1, double x2)
    {
        if (speedAtDist(x1) == speedAtDist(x2))
        {
            return false;
        }
        return true;
    }

    public int checkedSpeed (int speed, double x)
    {
        return Math.min(speed, speedAtDist(x));
    }
}
```

APPENDIX I: MATALB VISUALIZATION FUNCTION: *GRAPHER.M*

```
% Punit Shah & Jack Ingalls
% Team 7
% Albuquerque Academy
% 2008 Supercomputing Challenge
% Modernizing the U.S. Air Traffic Control System

function [] = grapher (runway, planes, h, t)
    % Program to graph the planes
    % Inputs:
    %   runway: 1x3 matrix containing location of airport
    %   planes: nx6 matrix containing the location of the n planes
    %     where the first three cols are the coordinates for the base of the
    %     arrow, while the second three rows are the coordinates for the head
    %   h: figure handle
    %   t: time for which this plot is running (passed as string)

    %SETUP THE PLOT
    % set the current plot the user requested figure #:
    figure(h);
    % clear the figure:
    clf
    % set the figure to a nice color scheme:
    colordef(gcf,'black')
    % set the plot to hold (rather than clear) each element as we build the
    % plot:
    hold on

    %PLOT THE AIRPORT
    scatter3(runway(1), runway(2), runway(3),'o','filled','SizeData',125)

    %PLOT THE PLANES
    n = size(planes);
    % we only want the height of the matrix to count the total planes:
    n = n(1);

    for i=1:n,

        x=planes(i,1);
        y=planes(i,2);
        z=planes(i,3);
        u=planes(i,4);
        v=planes(i,5);
        w=planes(i,6);
        u=u-x;
        v=v-y;
        w=w-z;

        % draw dot if zero vector
        if (u==0 && v==0 && w==0)
            % plane is in holding pattern
            scatter3(x,y,z,'o','filled','SizeData',80);
        else
```

```
            % plane is not in holding pattern
            quiver3(x,y,z,u,v,w,'LineWidth',2.5);
        end

    end

    %FINALIZE PLOT FOR VIEW
    box on;
    grid off;
    name = ['Airplane plot for time ', t];
    set(gcf, 'Name', name);
    title(['\bf\fontsize{16}' name]);
    ylim([-100 100]);
    xlabel('x axis (in nautical miles)');
    ylabel('y axis (in nautical miles)');
    zlabel('z axis, altitude (in ft)')
end
```