Team Control Number

US-12392

2022

IMMC

Summary Sheet

Abstract

Regarding airlines, time is money. The more efficient the system, the more flights that can be scheduled throughout the day. One major source of delays and faults in the system is boarding and disembarking, because this is a part of the process that involves a large amount of coordination between not only airline workers but passengers as well. This paper aims to create a model that simulates how long different boarding and disembarking methods take, and how a range of different situations can impact each boarding method, in order to find a most efficient method. We will organize the problem into two sections, the generation of different queues of boarding or disembarking passengers, and the simulation of the plane and the actions of the passengers. In order to factor in the variety of potential situations a boarding method may have to work around, the model will be versatile, taking into account both a variety of different plane layouts as well as different passenger attributes. These passenger attributes will include things such as their amount of carry-on bags and the time it takes for them to be stowed or taken down, their adherence to the boarding method, and the amount of passengers who need to board the plane. This will allow us to better account for potential flaws in each boarding method, such as how disobedient passengers may disrupt and slow the pace of otherwise steady methods, and how increasing the quantity of carry-on bags each passenger has can change the speed of each individual method. Finally, we will apply the model and the data it outputs to real world situations, discussing the different boarding methods we tested and their results.

Table of Contents

(1) Introduction

(1.1) Context

Running and maintaining any vehicle requires economic investment, and when talking about something as large and costly as a commercial aircraft, every minute where it isn't in use is money lost by the business that owns it. One major cause of delays and wasted time for airlines is boarding and disembarking from the plane, both of which rely heavily on the behavior of the passengers. However, issues and delays caused from human error on the part of the passengers can be reduced through the use of more efficient methods of boarding and disembarking from the aircraft. Some examples of these boarding methods include completely unstructured, where any passenger can enter and leave the plane at any time, to boarding by section, where passengers are split into groups based on where their seat is located. While boarding and disembarking methods aren't perfect because not all passengers follow the rules, they can reduce boarding times if they are built to be versatile around small-scale disruptions.

When looking at different approaches to boarding and disembarking from a plane, a few points in the process are seen to take a greater quantity of time than others. Should a passenger have a carry-on, as 87% of passengers do[1], the passenger will take longer to get to their seat because they have to stow their luggage in an overhead bin. There is also a factor that when passengers need to get over others to reach a seat, since if someone has already sat down and is blocking any seats, there is not enough room in the aisles of most aircraft for passing. Finally, factors such as pandemic restrictions can lead to increased variation in the percent of seats being filled, which can lead to a potential change in time spent boarding or disembarking from the plane.

(1.2) Problem Restatement

The first task was to construct a mathematical model that returned the amount of time different boarding and disembarking methods would take. This meant defining two boarding methods we would implement into our model, along with three of the most widely used methods, which include unstructured boarding, boarding by section, and boarding by seat. The two methods we chose are explained in section 2.2.

The model also had to be versatile so that we could use it to estimate changes to boarding times on multiple different aircraft layouts, as well as in various situations. Specific situations included increased amounts of carry-ons and other luggage, percentages of people following the chosen boarding method, and finally different quantities of passengers being permitted to board. Besides comparing these factors and how they impacted each boarding method’s time, we also had to compare both the average practical maximum and minimum of each boarding time, using all of this information to choose an optimal boarding and disembarking method.

(1.3) Assumptions

All passengers will walk at the same speed when in the airplane. The aisles are narrow enough and there are enough obstructions that passengers are generally limited to walking at a specific speed when in an airplane. This assumption allowed our simulation to be tick-based, as opposed to continuous, which would have been nearly impossible to implement efficiently.

There is no "Priority Boarding" beyond what the boarding method dictates. If an airline would like priority boarding, they can build it into boarding methods in the model. The goal of this paper is to determine the most efficient boarding method, not necessarily the one that reflects First Class privileges. However, it is very doable for a boarding method to be designed that prioritizes certain passengers over others; the user of the model would simply input their desired boarding order to get an estimate of how long it would take.

Passengers standing in the aisle are spaced out at a density of one person per row length. This allows our simulation to function on a grid rather than in a continuous space. It assumes that people in the aisle will be spaced out at a density of about one person per approximately 2.5 feet. While we could not find any data to support this, our group members tested this spacing, and it felt relatively natural considering our experience being in airplanes during boarding.

Entrances will only be on one side of the plane. It is much more difficult to build a terminal capable of boarding a plane on both sides, so it is more likely that all the entrances to the plane will be on the same side. Our model assumed that it would be on the left side, but if there are planes with entrances only on the right, the model would be functionally the same except reflected horizontally.

People will be directed towards the entrance closest to their seat and will follow those directions. People might arrive too early or late in the boarding queue (thus disobeying the prescribed order), but on a multiple-entrance plane, they would have no reason to go one way or the other, so we assume that they would follow the flight attendants' directions to whichever entrance is closest to their seat.

Disembarking is equivalent to boarding with zero disobedience and no passengers jumping over others. The process itself is the same as boarding but in reverse, and it is very rare that passengers will pass over others during disembarking. People did this during boarding because they might arrive at the gate at different times, but during disembarking, they are all leaving the plane at the same time. Therefore, the only valid disembarking strategies are ones in which people leave the plane without having to jump over anyone, even assuming zero disobedience.

(2) Methods

Our model consists of two major algorithms. One is the ordering algorithm, which determines the order of boarding of the passengers for each of our methods based on their seat's coordinates, assuming none of them disobey the rules. The second is the timing algorithm, which uses the list generated from the ordering algorithm to simulate each passenger entering the plane, finding their seat, and sitting down, as well as any extraneous actions each passenger undergoes. Our simulation in pseudocode is in Appendix A, and our full program in Python 3 is in Appendices B and C.

(2.1) Variable Declaration

The following variables are parameters to input into the simulation:

- P_c = The probability (from 0 to 1) that a passenger has a carry-on
- P_d = The probability (from 0 to 1) that a passenger is "disobedient" (by disobeying the prescribed boarding order)
- c_{max} = The maximum number of carry-ons passengers might stow

(2.2) Ordering Algorithm

Given the layout of the plane, the ordering algorithm outputs a list of seat coordinates and assigns each coordinate to each passenger within a queue. A seat coordinate is composed of a row value followed by a column value. The layout of the plane is a 2D array of integers, which contains a set number of rows. Each row stores a list of chunks, each chunk being a numeric value that displays the quantity of seats from one wall or aisle to the next wall or aisle. An example of this is [2, 3, 2], which would signify a row of three chunks of seats where the two side rows contain two seats, and the middle row has three seats. Should a row have no chunks, then it is an entrance row, which is counted as an entire row on its own. Using this type of layout, we generated the queue of passengers in the optimal order for each boarding method, and by reversing the order of passengers we can also craft a similar queue for disembarking.

The first three boarding and disembarking methods were those explained in the problem. The first boarding method is random or unstructured boarding. This is when any passenger can board in any order, and each passenger's seat coordinate does not impact the ordering of the queue. The ordering algorithm implements unstructured boarding through iterating through each row of the plane, counting the length of each row, or the number of seats it contains. Using the number of seats in each row, as well as the row itself, it can generate a list of all possible seat coordinates, which the algorithm then shuffles, and converts from a list of coordinates to a queue of passengers.

The second boarding method is boarding by section, or splitting the plane into thirds. The first section is the bow, or the frontmost third of the rows, the second section is the middle third

of rows, and the final section is the aft, or the third of rows farthest back. Boarding by section begins in a similar fashion to unstructured boarding, generating a list of all possible seat coordinates. Then, iterating through the list, each seat is placed in whichever section they are considered to be within. This is done through taking the number of rows and using integer division to divide it by three, then adding the remainder of the number of rows divided by three divided by two, to find what is considered the final row in the bow section, inclusive. Then, adding the number of rows divided by three and rounded down to the previous number will give us the final row in the middle section, and the remaining rows all fall into the aft section. The reason for these equations is to give a bias towards the bow and aft sections, as the description within the problem shows less seats within the middle section of seats, and should the aft and bow sections not be equal, it is best to have the aft section contain more rows. This is because when the largest boarding group is seated in the back third, there is less likely to be a large group of passengers blocked in by other passengers placing their carry-ons in the overhead bins should there be little to no passengers disobeying the boarding method's protocols. After all the seat coordinates are placed in their corresponding sections of the plane, each section is shuffled, and turned individually into separate queues of passenger objects. Finally, the sections are added together to make a final queue, where three parameters are used to define what order each section boards the plane in.

The third method is boarding by seat, also known as boarding by column. This method is similar to boarding by section, however, instead of going by row number, passengers board by column based on how far the passenger is from a window. In the case of aisle seats, space from a window is based instead on space from the center of the chunk of seats, with a bias towards the right in order to allow more passengers to walk a shorter distance from their entrance to their seat. In order to find which seats are in each column, the algorithm first finds how many boarding sections will be necessary to board the entire plane. This is equal to the largest chunk of seats in the layout data. After this, each section is created and stored by iterating inwards from the window seats inwards, and when an aisle is encountered, all chunks of seats before the aisle are added together, then the aisle is split in two by dividing its index by two and rounding upwards with the left half and downwards for the right half. By iterating downwards through the left half and iterating upwards through the right half, adding all the chunks before the aisle to each iteration, the algorithm creates each seat coordinate and adds it to the proper section. After all sections are created, they are shuffled and added together, and converted to a queue of passengers.

The last two boarding and disembarking methods were designed by us. The fourth method is stack boarding. In this method, all passengers are sorted by seat, and the seats farthest back in the plane are filled first. The remaining seats fill, or "stack" up until the seats farthest in the front of the plane are filled. This is done in a similar fashion to boarding by seat, as all aisles and sections must be split in similar ways, and seats have to be filled from window to aisle. The main differences to this algorithm are that instead of storing all seats in a corresponding section,

they are stored directly into a list of seat coordinates and are generated starting from the farthest rows from the entrance upwards.

The fifth method is window boarding. This method is also quite similar to boarding by seat, however, instead of shuffling each section the passengers board in order, so all of them can store their carry-ons and enter their seats at the same time. The only major difference from both boarding by seat and stack boarding is the order of iteration, as window boarding iterates from the back to the front of each column, instead of iterating from left to right of each row. This allows the function to better organize the seat coordinates in the form they are supposed to, and allows it to be versatile when approaching the order of aisles to be used. With window boarding, a large portion of its efficiency is making sure all aisles of a multi-aisle plane can be used at a time, so that while one aisle is placing their carry-ons in the overhead bins, another aisle can be utilized. In order to do this, the aisle farthest from the entrance is used first, specifically for the seats to the right of their windows, or the aisle equivalent. Then, it cycles back for the seats to the left of the aisle, then the next row on the right, so on and so forth until all seats have been added to the list. The list is then converted to a queue of passengers and returned.

(2.3) Timing Algorithm

Given the queue of passengers produced by the ordering algorithm, the timing algorithm uses a tick-based simulation to determine how long each boarding algorithm will take. The simulation represents the plane as a grid of seats, aisles, and walls, and passengers move through the aisles until they reach their seat, stopping in a few particular situations. Our implementation of the algorithm had five parameters: the queue (as generated by the timing algorithm), an encoded map of the layout of the desired plane, the chance that individual passengers will disobey the prescribed order (labeled "disobedience", $0 \leq P_d \leq 1$), the chance that individual passengers have carry-ons ($0 \leq P_c \leq 1$), and the maximum number of carry-ons each passenger can have. The default values for the last three parameters are 0.1, 0.87, and 1, respectively.

Because of the differing natures of our two algorithms, we created two separate systems for passengers' seat coordinates and two systems for representing the plane's layout. Therefore, before the simulation runs, our algorithm must translate between these systems. The ordering algorithm's system for seat coordinates had the same row values as that of the timing algorithm, whereas the column values are different. The ordering algorithm ignores aisles when counting columns, so a passenger in seat (12, 4) would be in the fourth seat from the left of the plane in row 12. However, the timing algorithm needs aisles to be counted as grid cells. This translation required a very simple algorithm that takes a seat's coordinate and adds the number of non-seat objects left of it to its coordinate's column value. On the other hand, the plane layouts turned out to be too complex to translate properly, since the simulation requires the layout to be a grid, whereas the ordering algorithm requires us to think of each row in terms of the sizes and order of the seat clusters. The "translation" for the plane layout is simply a hand-inputted system of symbols that maps the plane's layout: "e" for entrance, "A" for horizontal (major) aisle, "a" for

vertical (minor) aisle, "s" for seat, and "w" for wall. Walls were added to correct for the fact that in planes where some rows have more seats (e.g. in the First Class vs. Economy sections), the rows with fewer seats will not align grid-wise with the others. Theoretical "walls" are added to correct for this, positioned such that no passenger would have to cross them anyway. If a user were to want to input a different plane layout into our model, they would do so manually it via our system.

Before the simulation can begin, we need each passenger to know which entrance they will enter through and which vertical aisle they will travel down to get to their row. Both of these were implemented via algorithms that found each entrance/aisle, checked their distances to each passenger's seat, found the closest entrance/aisle, and stored that information with the passenger. Additionally, for each passenger, we need to determine whether or not they will be "disobedient" and how many (if any) carry-ons they have. We could not find any data for a normal amount of disobedient passengers, so we arbitrarily chose 10% as a default rate ($P_d = 0.1$). However, we did find that on average, 87% of passengers have a carry-on ($P_c = 0.87$), and we assumed that normally, passengers will have a maximum of one carry-on to stow ($c_{max} = 1$). For each passenger, these two attributes were stochastically determined according to the chances inputted into the algorithm. When we consider the scenario when people can have c_{max} carry-ons to stow, we use the carry-on chance P_c to randomly give them a bag independently c_{max} times (i.e. if $P_c = 0.5$ and $c_{max} = 2$, passengers have a 25% chance of having two bags, a 50% chance of having one, and a 25% chance of having none).

The simulation measures time in discrete steps, called "ticks". One tick is measured as the amount of time it takes for the average person to walk the vertical length of one airplane seat, considering that the aisle is relatively narrow, so people will, on average, walk a bit slower than normal. While we could not find data on how quickly people walk within an airplane, we found data on how quickly people walk when boarding a bus, which is a similar scenario. The average speed between males and females was approximately 0.86 meters per second[2]. The distance from the back of one seat to the back next seat in an average airplane is 0.7874 meters[3,4]. Therefore, one tick is approximately 0.9156 seconds.

The goal of the simulation is to find a way to take the current state of the plane and determine its next state after one tick passes. We accomplished this by looping through the passengers in the order of the queue and running a series of tests on them regarding their current position and surroundings to determine their next move. In general, the algorithm tells a passenger to board the plane if their entrance is empty, move forward if the next space in the path to their seat is empty, wait for a certain number of ticks if they are at their row and need to stow their bags, and wait indefinitely if there are people they need to skip over to get to their seat from the aisle, a situation we labeled "shuffling".

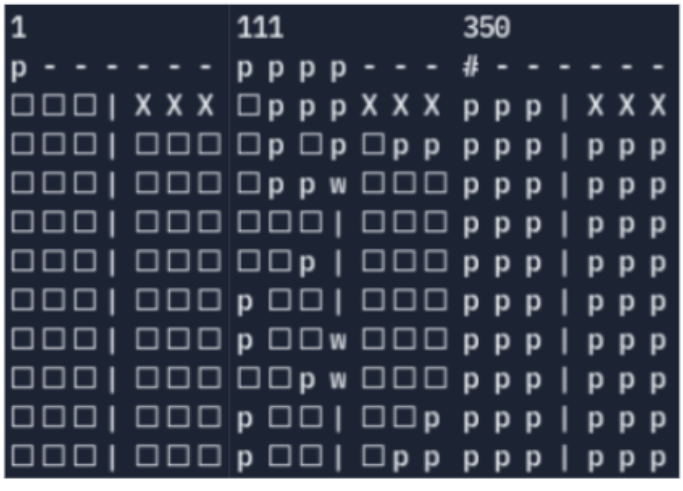
There are several special cases that our algorithm considers. We considered several algorithms for shuffling, but most were infeasible to implement, so we decided to go with a

"naive" algorithm. Let's say that there are n people who would have to get up from their seats so that passenger x could get to theirs. Those n people would need to go into the aisle ahead of passenger x such that they leave open the space to get into the row. Therefore, passenger x may only proceed when the next n + 1 spaces ahead of them are open, at which point passenger x moves forward, stows their bag(s) if necessary, "teleports" to their seat, and waits for the amount of time that it would have taken them to walk the distance they instantaneously traveled (so that the simulation doesn't end prematurely if the last person did this).

The other special case that the algorithm considered is if a passenger arrives at their row and has carry-ons that they need to stow. We could not find data regarding how long it takes, on average, for someone to stow a carry-on in overhead compartments, so we ran an experiment simulating it on our own. We estimated based on past experience that the compartments are about 6 feet above the ground, so we timed each of our group members in several attempts to hoist a heavy suitcase onto a shelf of that height. We found that typical, normal-seeming attempts ranged from 6 to 13 ticks, so in our algorithm, when a passenger needs to stow their bag(s), we generate a random integer between 6 and 13, inclusive, multiplied it by the number of carry-ons the passenger intends to stow, and had them "wait" in that position for that number of ticks while they are stowing their bags.

Our algorithm also indirectly accounts for disembarking strategies, since disembarking is essentially boarding in reverse with a few differences. First, we assume that there will be zero disobedience and that no passengers will ever have to pass over each other, since this rarely happens in real-world disembarking. Therefore, there are only three strategies that work for disembarking: Window, Stacking, and Seat-wise. To test for disembarking strategies, we simply tested only these strategies and with zero disobedience.

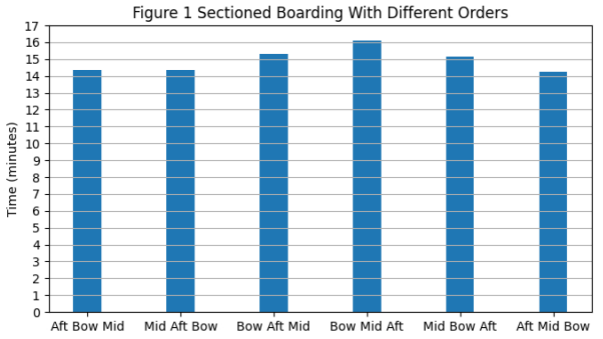
To test our algorithms, we created a system for animating the simulation. This animation used a modified representation of planes where "□" represents seats, "|" and "-" represent aisles, "#" represents entrances, "X" represents walls, "p" represents passengers, and "w" represents waiting passengers. This animation occurs in our Python console. It was implemented by clearing out the console and printing out the plane's layout every tick. The image to the right shows the beginning, middle, and end of the animation as applied to a short plane, following the random ordering algorithm.



(3) Results

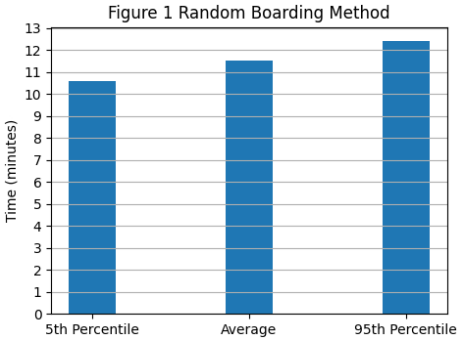
(3.1) Three Common Boarding Methods

Once we created our model, we needed to apply our model of the three widely used boarding methods to the standard "narrow-body" aircraft that was given to us. But before we could compare random boarding, boarding by section, and boarding by seat, we needed to compare the various combinations of boarding by section.

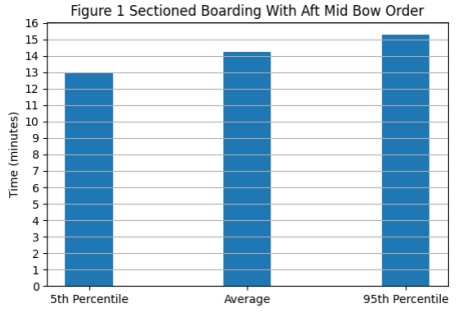


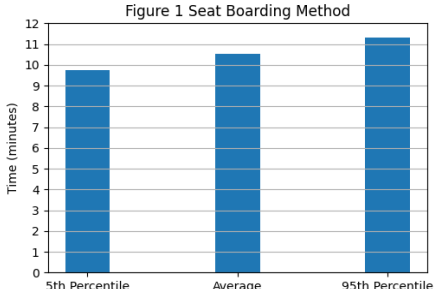
After examining the different combinations, we determined that boarding the aft section, then the middle section, and then the bow section was fastest and that the reverse – boarding the bow section, then the middle section, and then the aft section – was slowest. This makes sense, as when the bow or middle section is boarded before the aft section, people in the aft section will have to wait for people in the bow or middle section to finish boarding, slowing down the boarding process. Therefore, getting this result supported the accuracy of our model.

	Average	Practical Minimum	Practical Maximum
Aft-Mid-Bow	14.263	13.017	15.275
Aft-Bow-Mid	14.330	13.108	15.305
Mid-Aft-Bow	14.347	13.322	15.443
Bow-Aft-Mid	15.321	14.161	16.740
Bow-Mid-Aft	16.099	14.985	17.457
Mid-Bow-Aft	15.160	14.176	16.160



Once it was determined that boarding the aft section, then the middle section, and then the bow section was fastest, we decided it would be the combination we would use for boarding by section in the comparison of the three boarding methods, and consider in our recommendation letter to the airline executive. Using our model, we computed the average, practical maximum, and practical minimum of each of the three boarding methods by running our simulation one hundred times and then calculating the average, fifth percentile, and ninety-fifth percentile. For the random boarding method, the average is 11.528 minutes, the practical maximum is 12.421 minutes, and the practical minimum is 10.590 minutes. When it comes to the boarding by section method, the average is 14.263 minutes, the practical maximum is 15.275 minutes, and the practical minimum is 13.017 minutes. Finally, for the boarding by seat method, the average is 10.513 minutes, the practical maximum is 11.323 minutes, and the practical minimum is 9.736 minutes. Comparing this data, we can

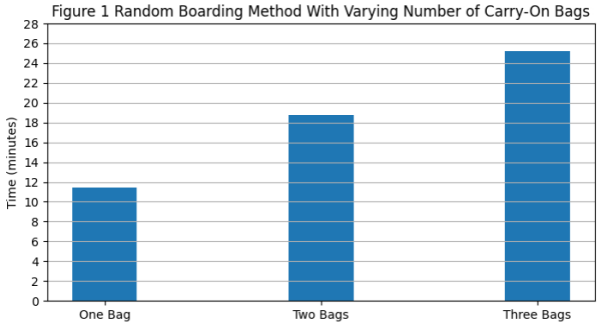




determine that boarding by seat is fastest, boarding by section is the slowest and random boarding falls in between the two.

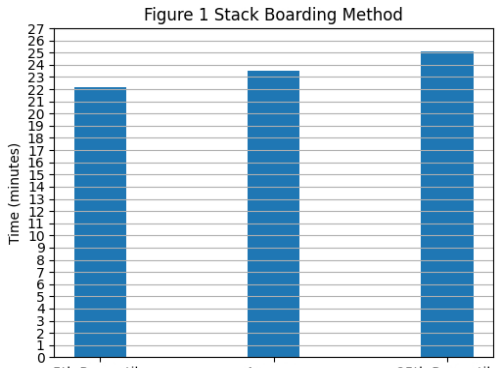
(3.2) Effect of More Carry-Ons

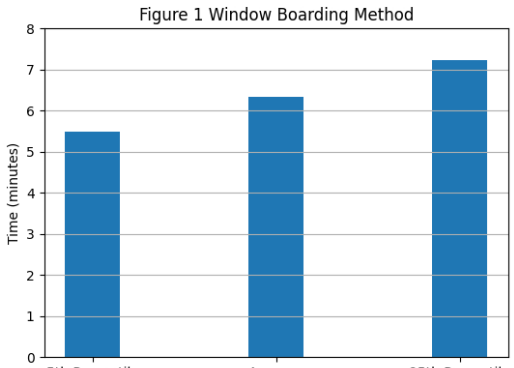
In addition to looking at those three boarding methods, we were also asked to analyze the effect that people having more carry-ons has on the boarding process. We predicted that people having more carry-ons to stow would increase the boarding time as it would take them longer than if they just stowed one carry-on, and as predicted, boarding time increased. We used the random boarding method on the "narrow-body" plane with changes in the maximum number of carry-ons one could bring to test this. The reason we picked the "narrow-body" plane is because it is the standard plane, and we picked the random boarding method as it was the boarding method that typically fell within the middle range of efficiency. When people could only have a maximum of one carry-on (the default) in this scenario, it took them an average of 11.446 minutes to board the plane, but when people could have a maximum of two carry-ons, it took them an average of 18.727 minutes to board. Furthermore, when people could have a maximum of three carry-ons, the boarding time continued to increase to an average of 25.171 minutes, thus proving our hypothesis that increasing the number of carry-ons people could bring would increase the boarding time.



(3.3) Two Additional Boarding Methods

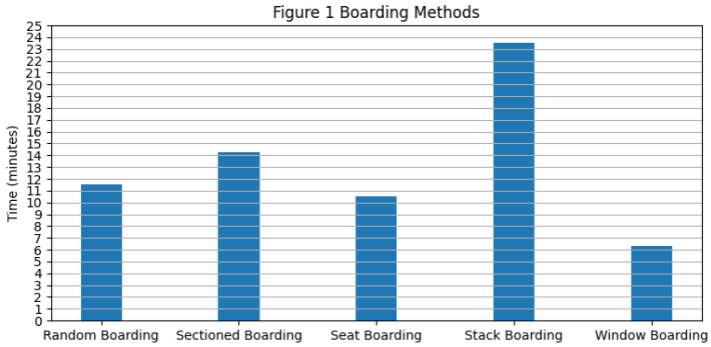
Not only did we analyze the three given boarding methods, we also came up with two additional boarding methods and analyzed them. The two additional boarding methods that we came up with were boarding by stacking and window boarding, which were explained earlier in the paper. We performed the same tests that we did on the three given boarding methods on these additional methods, running our simulation for these boarding methods one hundred times and then calculating the average, practical maximum, and practical minimum. For stack boarding, the average is 23.530 minutes, the practical maximum is 25.117 minutes, and the practical minimum is 22.172 minutes while the average, practical maximum, and practical minimum for window boarding are 6.334, 7.233, and 5.493 minutes, respectively. Comparing these to the three given boarding methods, we found that window boarding is the fastest while stack boarding is the slowest.

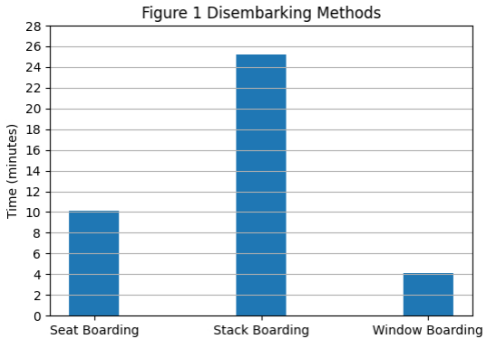




(3.4) Optimal Boarding and Disembarking Methods

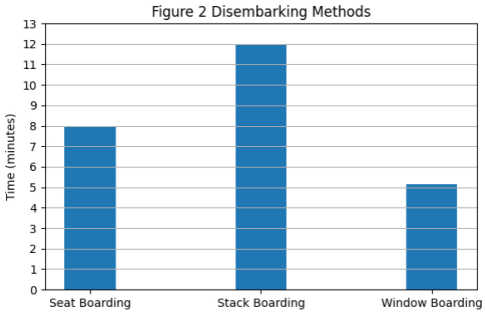
Using the information above, we determined that the optimal boarding method is window boarding as it is the fastest, but we still needed to determine what the optimal disembarking method would be. To determine the optimal disembarking method, we lowered disobedience as it is harder to disembark at the wrong time since it is easier for flight attendants to control who gets off of the plane at any given point. We also only used methods that didn't involve our "shuffling" algorithm as it is unlikely that someone in the window, for example, would be able to leave before the person sitting in the middle, given that people normally want to get off the plane as soon as possible, so they don't let people pass them. Using these restraints, we tested disembarking using the by seat, stacking, and window methods. Our results were that the disembarking-by-seat method took an average of 10.127 minutes to complete, the window disembarking method took an average of 4.094 minutes to complete, and the stack disembarking method took an average of 25.197 minutes to complete. Based on this data, we determined that window disembarking is the optimal method.





	Average	Practical Minimum	Practical Maximum
Figure 1 - Random	11.528	10.590	12.421
Figure 2 - Random	8.728	7.874	9.354
Figure 3 - Random	5.665	5.234	6.089
Figure 1 - by Section	14.263	13.017	15.275
Figure 2 - by Section	9.680	8.973	10.300
Figure 3 - by Section	6.224	5.631	6.974
Figure 1 - by Seat	10.513	9.736	11.323
Figure 2 - by Seat	8.082	7.431	8.561
Figure 3 - by Seat	5.458	5.005	5.921
Figure 1 - Stacking	23.530	22.172	25.117
Figure 2 - Stacking	11.790	11.094	12.421
Figure 3 - Stacking	13.255	12.330	14.146
Figure 1 - Window	6.334	5.493	7.233
Figure 2 - Window	5.986	5.509	6.47
Figure 3 - Window	10.447	9.659	11.475

After determining the optimal boarding and disembarking methods for the “narrow-body” plane, we determined the optimal boarding and disembarking methods for the “flying wing” and “two-entrance, two aisle” planes. The “flying wing” plane followed the same pattern the “narrow-body” plane did and had window boarding and disembarking as its optimal methods. Window boarding was not optimal for the “two-entrance, two aisle” plane however as its optimal boarding and disembarking methods are boarding and disembarking by seat.



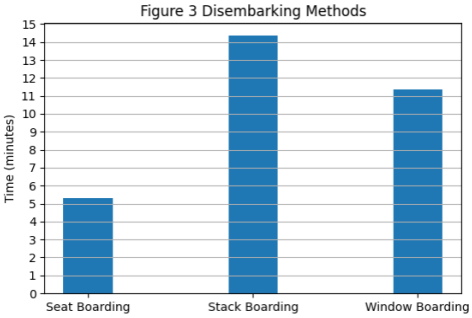
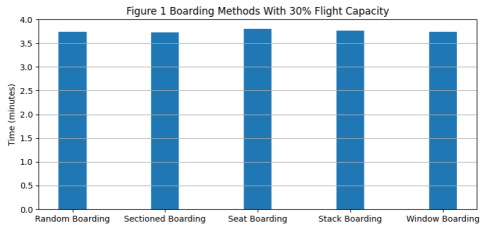
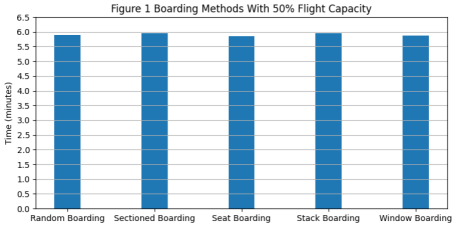


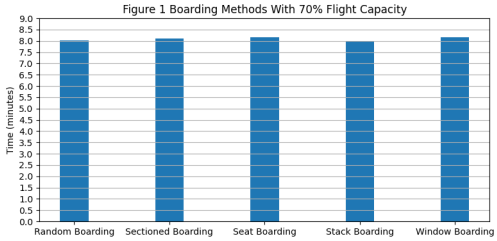
	Figure 1	Figure 2	Figure 3
Stacking Disembarking	25.197	11.983	14.345
Window Disembarking	4.094	5.156	11.381
by Seat Disembarking	10.127	7.946	5.315

(3.5) Effect of Lowering Capacity

When capacity was lowered to thirty, fifty, and seventy percent, the boarding and disembarking times for the three planes became much more similar and less varied. Because of this, there was no longer a set pattern of which methods were optimal. From this information, we determined that which boarding and disembarking methods are used becomes less important with less people on the plane as there are fewer people who can collide and prevent each other from reaching their seat.







(4) Analysis

(4.1) Advantages and Limitations

When it comes to modeling different combinations of passenger attributes, our model is very versatile. It isn't difficult to modify how disobedient a passenger is, their carry-on luggage, or even the amount of passengers that are on the plane. It is also quite easy to change the model between different plane layouts and boarding methods. However, should a specific plane layout or boarding method the user wishes to test not already be implemented, it can be difficult for the user to implement it. With the layout of each plane, it is a simple matter of time and learning how adding new layouts works, but with boarding methods it is much more strenuous and involves extensively testing any added code.

Another benefit to our model is that due to its simplicity in returning how long each boarding method takes, it is easy to compare how changes to certain parameters and boarding methods can affect the overall speed of boarding or disembarking from each plane layout. Such comparison can be particularly useful should the user have data from previous flights, as the user could better choose a boarding method that matches their group of passengers.

Regarding the runtime in our model, running the simulation many times—especially for less efficient boarding methods on larger planes—can take a long time to run. However, this significant runtime doesn't harm the feasibility of our model's use until planes get to be of an unreasonable size and layout; otherwise, runtime is generally just a few seconds. This is longer than most computer programs would take to run.

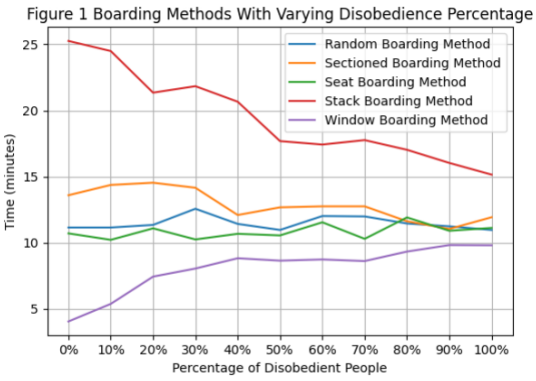
Finally, due to our model's grid-based system, it does not take into account variations in the amount of space each passenger may take up. Someone without a carry-on takes up as much space as someone with multiple, which is unreasonable in a real scenario. The same grid-based system also means all passengers walk at the same speed, which is also unlikely in a real scenario for much the same reason. Even with such limitations, our model should be a good basis for comparison of different boarding methods and situations, as variation in walking speeds and passenger space will not cause any major differences in time. All passenger space and speeds will more or less even out to a somewhat constant value.

(4.2) Sensitivity Analysis

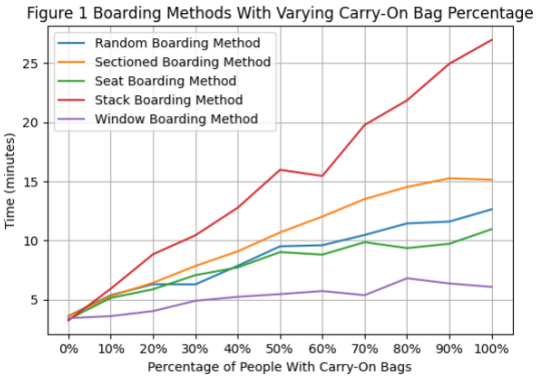
We tested the sensitivity of the model with respect to two variables: the percentage of passengers who don't follow the prescribed boarding method, which we called disobedience, and the average number of passengers with carry-on bags in the plane. The graphs below show the time, in minutes, required for passengers to board the narrow-body passenger aircraft with varying percentages of disobedience from 0% to 100% and varying percentages of people who

have carry-on bags from 0% to 100%. Each graph has 5 lines to represent the 5 boarding methods: random, sectioned, seat, stack, and window.

Increasing the percentage of passengers who are disobedient has a considerable effect on the boarding time for stack window boarding whereas there is significantly less impact on random, seat, and sectioned boarding. This relationship is reasonable because disobedience adds randomness to the system. With stack boarding, people always have to wait for other passengers with carry-on bags to stow the bags in the overhead bins. However, with increased disobedience, people won't always have to wait for each other to stow carry-on bags, decreasing the boarding time. With window seating on the other hand, people rarely have to wait for others to stow their bags. However, with the randomness that disobedience adds, people are more likely to have to wait for each other, increasing the boarding time. Sectioned, seat, and random boarding aren't as impacted by disobedience because these methods already are randomized to some extent. Adding randomness doesn't impact overall boarding time significantly. Overall, though, as disobedience increases, the boarding time for the different boarding methods approaches the boarding time for the random boarding method. As mentioned above, disobedience adds randomness, so it makes sense that the other boarding methods become increasingly similar to the random boarding method as disobedience rises.



The percentage of people who have carry-ons also has a significant impact on the time it takes for the passengers to finish boarding. Specifically, when the carry-on bag percentage increases, the boarding time increases in an approximately linear relationship. This relationship exists because if more passengers have carry-on bags, then more people have to wait for each other to finish stowing those bags, increasing the wait time as well as the overall time for everyone to be boarded. It is important to notice that stack boarding is impacted the most by an increase in the percentage of passengers whereas window boarding is least impacted. This makes sense because in stack boarding, people in the same row board together, causing more people to wait when someone in front of them has to stow a bag. On the other hand, with window boarding, people who are in the same column board at the same time. Therefore, not as many people have to wait for each other to stow their bags.



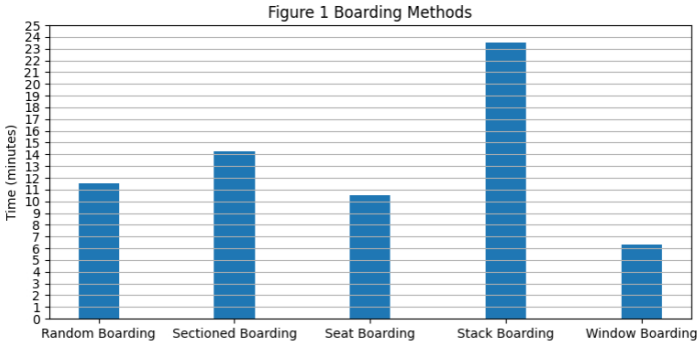
Although everyone being disobedient or everyone bringing carry-on bags isn't realistic, increasing the disobedience or the percentage of people with carry-ons will change the time it takes for the passengers to board to some degree. As a result, the percentage of disobedient passengers and the percentage of passengers with carry-on bags are critical and impactful factors to consider when determining the boarding time.

(5) Letter

To whom it may concern,

Boarding and disembarking is a time consuming process, one that can lead to minor delays that can build up into much more major issues. And any delay, whether it be five minutes or an hour, can lead to a decreased economic benefit, either due to unhappy passengers choosing other airlines or simply that delays mean less time for flights.

Surprisingly, many of the most common forms of boarding are quite slow, particularly boarding from the back of the plane to the front. Boarding by splitting the plane into distinct sections of rows, and boarding by section, is another common form of boarding, and while it is significantly faster, as seen by the graph to the right, it is still technically slower than unstructured, or random, boarding.



So what is the optimal method for boarding and disembarking? Well, according to our model and simulation, the best method of both is window boarding. Window boarding is when a plane boards by column of seats, instead of by row, and fills in from the windows to the center aisles, or center to the aisle for chunks of seats surrounded by two aisles. When disembarking, this would be done in reverse. The window method also requires passengers to line up and enter from back to front of each column of seats, so everyone can stow their carry-ons at the same time. This is due to the fact that carry-ons are the greatest source of time in any boarding method, forcing other passengers to wait as someone in front of them lifts their bag. The window method is meant to reduce such time by coordinating everyone's stowing as best as possible.

It is of note that the window boarding method would make it difficult for larger groups of people, such as families, to board together, and that it would not emphasize any form of priority boarding, another source of income for airline companies. However, the second most efficient, as given by our model, is boarding by column, a similar method. It is the same as earlier, where passengers board by column of seats instead of rows, and fill in towards the aisles. The main difference is that instead of a rigid order, each column of seats is considered a section, and all passengers in a section board at the same time. This would allow for priority boarding, as well as flexibility for groups who want to stay together. It is slightly less efficient, as passengers can find themselves blocked by people stowing or taking down their bags, however, it still reduces such issues as multiple people wouldn't try to reach the same row and chunk of seats at the same time.

Sincerely,
Team US-12392

(6) Works Cited

1. Freed, J. (2012, March 5). Bulky suitcase? Overhead bins are getting bigger. NBC News. Retrieved April 15, 2022, from https://www.nbcnews.com/id/wbna46632596
2. Ali, M. F. M., Abustan, M. S., Talib, S. H. A., Abustan, I., Rahman, N. A., & Gotoh, H. (2018, March 19). A case study on the walking speed of pedestrian at the bus terminal area. E3S Web of Conferences. Retrieved April 15, 2022, from https://www.e3s-conferences.org/articles/e3sconf/pdf/2018/09/e3sconf_cenviron2018_01023.pdf
3. Long-haul economy class comparison chart. (n.d.). SeatGuru. Retrieved April 15, 2022, from https://www.seatguru.com/charts/longhaul_economy.php
4. Short-haul economy class comparison chart. (n.d.). SeatGuru. Retrieved April 15, 2022, from https://www.seatguru.com/charts/shorthaul_economy.php

(7) Appendices

(7.1) Appendix A: Simulation in Pseudocode

```
Set tick counter to 0
While there are passengers who are not seated or are waiting {
  For each passenger in the order of the original queue {
    If passenger is not seated {
      If passenger is waiting {
        Reduce tick counter by one and continue waiting
      }
      Else if passenger is in entrance or horizontal aisle {
        If in column of the vertical aisle the passenger should take {
          If space down the vertical aisle is open {
            Move into vertical aisle
          }
        }
        Else if next space in horizontal aisle is open {
          Move into next space in horizontal aisle
        }
      }
      Else if in vertical aisle {
        If one row behind/ahead of desired row and will have to shuffle {
          Count the number of people (n) passenger will have to skip over
          If the next n + 1 spaces are open {
            Move forward
          }
        }
        Else if in the aisle in the desired row {
          If passenger has carry-ons {
            Generate a random number (n) from 6-13
            Set wait tick counter to n*[number of carry-ons]
          }
          Else if passenger would have to skip over people {
            Move to seat and wait for the number of ticks moving would take
          }
          Else if next space in direction of seat is empty {
            Move into next space
          }
        Else if out of aisle and in row {
          If not at seat yet {
            Move one space toward seat
          }
          Else if at seat {
            Set passenger to seated
          }
        }
      }
    }
  }
  Add one to tick counter
}
Return tick counter's value
```

(7.2) Appendix B: Ordering Algorithm in Python 3

Note: The code below is less organized than its descriptions in the Ordering Algorithm section because we solved problems with the code as we created it, whereas our explanations in this paper are retrospective.

```
class Passenger:
  def __init__(self, seat):
    self.seat = seat # A tuple with the location of each passenger
    self.hasCarryOn = None
    self.disobedience = None
    self.waitTicks = 0 # The number of ticks this person has left to wait
    self.status = "queued" # Becomes "boarded" when boarding the plane
    self.position = [None, None] # The current position of the passenger
    self.aisle = None # The coordinate of the minor aisle this passenger will use
    self.numCarryOns = 0 # The number of carry-ons this passenger has to stow

  def __repr__(self):
    return str(self.seat) # Passengers are identified by their seat's location

  def __str__(self):
    return str(self.seat)
```

```
import random
import math
from passenger import Passenger

# Returns ordered queues of passengers for different boarding and disembarking
methods on different plane layouts.
class Plane:

  # Initializer and Test Functions

  def __init__(self, chunks):
    # Will store the layout of the plane.
    self.data = chunks

  # Returns a representation of self.data to check if different layouts correctly
match plane figures.
  def __str__(self):
    temp = ""
    for lst in self.data:
      if (len(lst) == 0):
        temp += "e"
      else:
        for c in range(len(lst)):
          for i in range(lst[c]):
            temp += "s "
          if (c != len(lst)-1):
            temp += "r "
      temp += "\n"
    return temp
```

```
  # Minor Functions and Calculations

  # Returns number of rows not including entrances.
  def numRows(self):
    temp = [x.copy() for x in self.data]
    while [] in temp:
      temp.remove([])
    return len(temp)

  # Returns the number of seats in each row, by adding all chunks of seats
together.
  def sumChunks(self, row):
    temp = 0
    for chunk in self.data[row]:
      temp += chunk
    return temp

  # Returns the largest chunk of seats in the plane.
  def greatestChunk(self):
    max = 0
    for lst in self.data:
      for i in range(len(lst)):
        if (i == 0 or i == len(lst)-1):
          if (lst[i] > max):
            max = lst[i]
        else:
          if (math.ceil(lst[i]/2) > max):
            max = math.ceil(lst[i]/2)
    return max

  # Converts a list of seat coordinates to a list of passengers in the same order,
each assigned each coordinate.
  def convertToPassengers(self, seats):
    passengerList = []
    for i in range(len(seats)):
      passengerList.append(Passenger(seats[i]))
    return passengerList

  # Returns a list of all seat coordinates on the plane, in no particular order.
  def getSeats(self):
    seats = []
    for l in range(len(self.data)):
      rowLength = 0
      for chunk in self.data[l]:
        rowLength += chunk
      for i in range(rowLength):
        seats.append((l, i))
    return seats

  # Boarding Order Generation Functions

  # Returns the order of Random or Unstructured boarding.
  def getRandomSeats(self):
    seats = self.getSeats()
```

```
    random.shuffle(seats)
    return self.convertToPassengers(seats)

  # Returns the order of boarding by Section. Allows for mixing of the sections.
  def getSectionedSeats(self, aI=0, mI=1, bI=2):
    seats = self.getSeats()
    numRows = self.numRows()
    aft = []
    mid = []
    bow = []
    for i in range(len(seats)):
      seat = seats[i]
      if (seat[0] <= numRows//3 + numRows%3/2):
        bow.append(seat)
      elif (seat[0] <= numRows//3 + (numRows//3 + numRows%3/2)):
        mid.append(seat)
      else:
        aft.append(seat)
    ret = [None, None, None]
    random.shuffle(aft)
    ret[aI] = aft
    random.shuffle(mid)
    ret[mI] = mid
    random.shuffle(bow)
    ret[bI] = bow
    return self.convertToPassengers(ret[0]) + self.convertToPassengers(ret[1]) +
self.convertToPassengers(ret[2])

  # Returns the order of Stack boarding.
  def getStackingSeats(self):
    seats = []
    for l in range(len(self.data)-1, -1, -1):
      rowSeats = self.sumChunks(l)

      # Checks to make sure the row is not an entrance.
      if (len(self.data[l]) != 0):

        # Iterates through the first row of window seats.
        for c in range(rowSeats, rowSeats-self.data[l][-1], -1):
            seats.append((l, c-1))

        # Checks to make sure there are multiple rows of seats.
        if (len(self.data[l]) > 1):

          # Iterates through the aisles, calculating each seat there.
          for a in range(len(self.data[l])-2, 0, -1):
            s = 0
            for x in range(0, a):
              s += self.data[l][x]

            # Rightmost aisle seats.
            for c in range(s+self.data[l][a]//2, s+self.data[l][a]):
              seats.append((l, c))
```

```
            # Leftmost aisle seats.
            for c in range(s+math.ceil(self.data[l][a]/2), s, -1):
              seats.append((l, c-1))

          # Iterates through the final row of window seats.
          for c in range(0, self.data[l][0]):
            seats.append((l, c))

    return self.convertToPassengers(seats)

  # Returns the order of boarding by Seat.
  def getColumnSeats(self):

    # Generates storage list for each boarding group.
    seats = []
    for i in range(self.greatestChunk()):
      seats.append([])

    for l in range(len(self.data)):

      # Checks to make sure the row is not an entrance.
      if (len(self.data[l]) != 0):

        # Stores how far from a window seat iterator has moved.
        temp = 0

        # Iterates through the first row of window seats.
        for c in range(0, self.data[l][0]):
          seats[temp].append((l, c))
          temp += 1

        # Checks to make sure there are at least two rows of window seats.
        if (len(self.data[l]) >= 2):
          temp = 0

          # Iterates through the second row of window seats.
          for c in range(self.sumChunks(l)-1, self.sumChunks(l)-self.data[l][-1]-1,
-1):
            seats[temp].append((l, c))
            temp += 1

          # Check if there are aisle seats.
          if (len(self.data[l]) >= 3):

            # Iterates through the aisles, calculating each seat there.
            for i in range(1, len(self.data[l])-1):
              s = 0
              for x in range(0, i):
                s += self.data[l][x]
              temp = 0

              # Leftmost aisle seats.
              for c in range(math.ceil(self.data[l][i]/2), 0, -1):
                seats[temp].append((l, s + c - 1))
```

```
              temp += 1
            temp = 0

            # Rightmost aisle seats.
            for c in range(math.ceil(self.data[l][i]/2)+1, self.data[l][i]+1):
              seats[temp].append((l, s + c - 1))
              temp += 1

    newSeats = []
    for i in seats:
      random.shuffle(i)
      newSeats = newSeats + i
    return self.convertToPassengers(newSeats)

  # Returns the order of Window boarding.
  def getWindowSeats(self):
    seats = []
    for i in range(self.greatestChunk()):
      seats.append([])

    # Iterates through each column instead of through each row.
    for i in range(1, len(self.data[1])-1):
      for l in range(len(self.data)):

        # Check if there are aisle seats.
        if (len(self.data[l]) >= 3):
          s = 0
          for x in range(0, i):
            s += self.data[l][x]
          temp = 0

          # Leftmost aisle seats.
          for c in range(math.ceil(self.data[l][i]/2), 0, -1):
            seats[temp].append((l, s + c - 1))
            temp += 1

    # Iterates through the rightmost columns of window seats.
    for l in range(len(self.data)):
      if (len(self.data[l]) >= 2):
        temp = 0
        for c in range(self.sumChunks(l)-1, self.sumChunks(l)-self.data[l][-1]-1,
-1):
          seats[temp].append((l, c))
          temp += 1

    # Iterates through leftmost columns of window seats.
    for l in range(len(self.data)):
      if (len(self.data[l]) != 0):
        temp = 0
        for c in range(0, self.data[l][0]):
          seats[temp].append((l, c))
          temp += 1

    # Iterates through the other half of all aisle seats.
```

```
    for i in range(1, len(self.data[1])-1):
      for l in range(len(self.data)):
        if (len(self.data[l]) >= 3):
          s = 0
          for x in range(0, i):
            s += self.data[l][x]
          temp = 0
          for c in range(math.ceil(self.data[l][i]/2)+1, self.data[l][i]+1):
            seats[temp].append((l, s + c - 1))
            temp += 1

    newSeats = []
    for i in seats:
      i.reverse()
      newSeats = newSeats + i
    return self.convertToPassengers(newSeats)
```

(7.3) Appendix C: Timing Algorithm in Python 3

Note: The code below is less organized than its descriptions in the Timing Algorithm section because we solved problems with the code as we created it, whereas our explanations in this paper are retrospective.

Included at the end of the program are some examples of us using it to generate data.

```
import random # To generate random numbers
from passenger import Passenger # Passenger object
from plane import Plane # Plane object
import os # Allows us to clear the console for our diagnostic animation
from time import sleep # Allows us to slow down the algorithm for the animation
from matplotlib import pyplot as plt # Allows us to plot results

tickTime = 0.2 # The amount of time between ticks in the animation
runs = 0 # Global variable that stores the number of simulations that have been run

# Takes a queue and removes 100*(1-decimalPercent)% of the passengers
def changePercentPassengers(decimalPercent, passengers):
  copiedPassengers = passengers.copy()
  numPassengers = len(copiedPassengers)
  numFinalPassengers = int(decimalPercent * numPassengers)
  finalPassengers = []
  for i in range(numFinalPassengers):
    randIdx = random.randint(0, numPassengers-1)
    finalPassengers.append(copiedPassengers.pop(randIdx))
    numPassengers -= 1
  return finalPassengers

# Takes a list of numbers and finds the nth percentile among them
def percentileCalc(percentile, timesLst):
  sortedTimesLst = sorted(timesLst)
```

```
  decimalPercentile = percentile/100
  percentileIdx = int(decimalPercentile*len(sortedTimesLst)-1)
  return(sortedTimesLst[percentileIdx])

# Takes a plane layout in s/a/w form and makes it look like a plane (for animation)
def reprPlane(plane):
  arr = [[0 for item in row] for row in plane]
  dct = {'s': '□', 'a': '|', 'A': '-', 'w': 'X', 'e': '#'}
  for r in range(len(plane)):
    for c in range(len(plane[r])):
      if plane[r][c][0] is not None:
        pgr = plane[r][c][0]
        if pgr.waitTicks > 0:
          arr[r][c] = "w"
        else:
          arr[r][c] = "p"
      else:
        arr[r][c] = dct[plane[r][c][1]]
    arr[r] = ' '.join(arr[r])
  return "\n".join(arr)

# Clears the console and prints the plane again, along with the current tick count
def display(plane, ticks):
  os.system("clear")
  print(ticks)
  print(reprPlane(plane))
  sleep(tickTime)

# Plane definitions according to the Ordering Algorithm's layout system ("blocks")
figure1 = Plane([[]] + [[3, 0]] + [[3, 3]] * 31)
figure2 = Plane([[]] + [[0, 6, 6, 6, 0]] * 3 + [[3, 6, 6, 6, 3]] * 11)
figure3 = Plane([[2, 2, 2]] * 3 + [[]] + [[2, 3, 2]] * 14 + [[2,0,2]] + [[2,3,2]] *
21 + [[]])

# Plane definitions according to the Timing Algorithm's layout system ("s/a/w")
plane1layout = [["e"] + ["A"]*6] + [["s", "s", "s", "a", "w", "w", "w"]] +   [["s",
"s", "s", "a", "s", "s", "s"]]*31
plane2layout = [["e"] + ["A"]*27] + [["w"]*3 + (["a"] + ["s"]*6)*3 + ["a"] +
["w"]*3]*3 + [["s"]*3 + (["a"] + ["s"]*6)*3 + ["a"] + ["s"]*3]*11
plane3layout = [["s", "s", "a", "s", "w", "s", "a", "s", "s"]]*3 + [["e"] +
["A"]*8] + [["s", "s", "a", "s", "s", "s", "a", "s", "s",]]*14 + [["s", "s", "a",
"w", "w", "w", "a", "s", "s"]] + [["s", "s", "a", "s", "s", "s", "a", "s",
"s",]]*21 + [["e"] + ["A"]*8]

# Layout in s/a/w form -> A 3D array plane within which the simulation occurs
def generatePlane(layout):
  plane = [[[None, item] for item in row] for row in layout]
  return plane

# The simulation function
def simulation(queue, layout, disProb=0.1, carryProb=0.87, maxCarryOns=1):
  # Helper function that tests if everyone is seated or not
  def allSeated():
    for passenger in seated:
```

```
      if seated[passenger] == False:
        return False
    return True

  # Helper function that checks if there is someone in a row between a seat and the
aisle (both exclusive)
  def shufflingTest(row, seat, aisle):
    if seat < aisle:
      low = seat + 1
      high = aisle
    else:
      low = aisle + 1
      high = seat

    for i in range(low, high):
      if plane[row][i][0] != None:
        return True
    return False

  # Helper function that counts how many people are within a row between seat and
aisle, exclusive
  def countPeople(row, seat, aisle):
    if seat < aisle:
      low = seat + 1
      high = aisle
    else:
      low = aisle + 1
      high = seat

    count = 0
    for i in range(low, high):
      if plane[row][i][0] != None:
        count += 1

    return count

  # Choosing the closest entrance to each passenger
  whichEntrance = dict()
  entrances = []
  for r in range(len(layout)):
      if layout[r][0] == 'e':
        entrances.append(r)

  for passenger in queue:
    shortest = entrances[0]
    for entrance in entrances:
      if abs(passenger.seat[0] - entrance) < abs(passenger.seat[0] - shortest):
        shortest = entrance
    whichEntrance[passenger] = shortest

  # Creating a list that will be populated with people who are disobedient
  disobedients = []

  # Loop through each passenger in queue-order
```

```
  for passenger in queue:
    # Determining whether or not each passenger will be disobedient and have a
carry on
    passenger.disobedience = random.random() < disProb

    for n in range(maxCarryOns):
      passenger.numCarryOns += int(random.random() < carryProb)
    passenger.hasCarryOn = bool(passenger.numCarryOns)

    # Redoing the coordinate system for the simulation's purposes
    row = layout[passenger.seat[0]]
    temp = 0
    col = 0

    for item in row:
      if item == "a" or item == "w":
        temp += 1
      if item == "s":
        col += 1
        if col-1 == passenger.seat[1]:
          break
    passenger.seat = (passenger.seat[0], passenger.seat[1]+temp)

    # Determining which aisle the passenger would use
    aisles = []
    for i, item in enumerate(row):
      if item == "a":
        aisles.append(i)
    minDist = aisles[0]

    for ais in aisles:
      if abs(ais - passenger.seat[1]) < abs(minDist - passenger.seat[1]):
        minDist = ais

    passenger.aisle = minDist

  # New loop through each passenger
  for passenger in queue:
    # Adding disobedient passengers to their list and removing them from the queue
    if passenger.disobedience == True:
      disobedients.append(passenger)
      queue.remove(passenger)

  # Re-inserting disobedient passengers into the queue at a random spot
  for passenger in disobedients:
    queue.insert(random.randint(0, len(queue) + 1), passenger) # Remove the "+ 1"
from this statement if error

  # Simulation
  plane = generatePlane(layout) # Setting the stage
  ticks = 0
  seated = {passenger : False for passenger in queue} # Makes it easy to check if a
particular passenger is seated
```

```
  # While not everyone is seated
  while allSeated() == False:
    # For each passenger in the order of the queue
    for passenger in queue:
      ent = whichEntrance[passenger] # The entrance the passenger will use
      pos = passenger.position # The passenger's current position
      row = passenger.seat[0] # The passenger's seat's row

      # Passenger is seated if they are in their seat and not waiting
      if passenger.position == list(passenger.seat) and passenger.waitTicks == 0:
        seated[passenger] = True

      # If passenger is waiting, subtract one from tick counter
      if passenger.waitTicks > 0:
        passenger.waitTicks -= 1

      # If passenger is seated, skip all other tests
      elif seated[passenger] == True:
        pass

      # Else if passenger is not on the plane and their entrance is open, go to the
entrance
      elif passenger.status == "queued" and plane[ent][0][0] == None:
        passenger.status = "boarded"
        passenger.position = [whichEntrance[passenger], 0]
        plane[ent][0][0] = passenger

      # Else if passenger is not on the plane, wait
      elif passenger.status == "queued" or pos[0] == None or pos[1] == None:
        pass

      # Else if in horizontal aisle and wants to turn down vertical aisle
      elif (plane[pos[0]][pos[1]][1] == "A" or plane[pos[0]][pos[1]][1] == "e") and
pos[1] == passenger.aisle:
        # If they are behind their seat, set their direction to forward
        if pos[0] > passenger.seat[0]:
          dir = (-1, 0)
        # Else, set their direction to backward
        else:
          dir = (1, 0)
        # If the spot in the direction they want to go is open, go there
        if plane[pos[0] + dir[0]][pos[1] + dir[1]][0] == None:
          passenger.position = [pos[0] + dir[0], pos[1] + dir[1]]
          plane[pos[0] + dir[0]][pos[1] + dir[1]][0] = passenger
          plane[pos[0]][pos[1]][0] = None

      # Else if in horizontal aisle and next space is empty, move to next space
      elif (plane[pos[0]][pos[1]][1] == "A" or plane[pos[0]][pos[1]][1] == "e") and
plane[pos[0]][pos[1] + 1][0] is None:
        passenger.position[1] += 1
        plane[pos[0]][pos[1] - 1][0] = None
        plane[pos[0]][pos[1]][0] = passenger

      # Else if shuffling around needs to happen (and passenger is ahead of their
```

```
seat)
      elif pos == [row - 1, passenger.aisle] and shufflingTest(row,
passenger.seat[1], pos[1]):
        openSpaces = True
        requiredSpaces = countPeople(row, passenger.seat[1], pos[1]) + 1

        # Check if next n + 1 spaces are open
        for n in range(1, requiredSpaces + 1):
          if pos[0] + n < len(plane):
            if plane[pos[0] + n][pos[1]][0] != None:
              openSpaces = False

        # If they are, move into seat row
        if openSpaces:
          passenger.position[0] += 1
          plane[pos[0] - 1][pos[1]][0] = None
          plane[pos[0]][pos[1]][0] = passenger
        if pos == passenger.seat:
            seated[passenger] = True

      # Else if shuffling around needs to happen (and passenger is behind their
seat)
      elif pos == [row + 1, passenger.aisle] and shufflingTest(row,
passenger.seat[1], pos[1]):
        openSpaces = True
        requiredSpaces = countPeople(row, passenger.seat[1], pos[1]) + 1

        for n in range(1, requiredSpaces + 1):
          if pos[0] - n >= 0:
            if plane[pos[0] - n][pos[1]][0] != None:
              openSpaces = False

        if openSpaces:
          passenger.position[0] -= 1
          plane[pos[0] + 1][pos[1]][0] = None
          plane[pos[0]][pos[1]][0] = passenger
        if pos == passenger.seat:
            seated[passenger] = True

      # Else if in vertical aisle and reached target row and have carry-on, stow
carry-on
      elif plane[pos[0]][pos[1]][1] == "a" and pos[0] == passenger.seat[0] and
passenger.hasCarryOn:
        passenger.waitTicks = random.randint(6, 13)*passenger.numCarryOns
        if pos == passenger.seat:
            seated[passenger] = True
        passenger.hasCarryOn = False

      # Else if in vertical aisle and reached target row, move into row
      elif plane[pos[0]][pos[1]][1] == "a" and pos[0] == passenger.seat[0]:
        # If there are people between the aisle and the seat, skip over them and
wait
        if shufflingTest(row, passenger.seat[1], pos[1]):
          passenger.waitTicks = abs(passenger.seat[1] - pos[1]) - 1
```

```
          passenger.position = list(passenger.seat)
          plane[pos[0]][pos[1]][0] = None
          pos = passenger.position
          plane[pos[0]][pos[1]][0] = passenger

        # Move one space toward their seat
        elif pos[1] > passenger.seat[1]:
          passenger.position[1] -= 1
          plane[pos[0]][pos[1] + 1][0] = None
          plane[pos[0]][pos[1]][0] = passenger
          if pos == passenger.seat:
            seated[passenger] = True
        else:
          passenger.position[1] += 1
          plane[pos[0]][pos[1] - 1][0] = None
          plane[pos[0]][pos[1]][0] = passenger
          if pos == passenger.seat:
            seated[passenger] = True

      # Else if in vertical aisle, move one space forward
      elif plane[pos[0]][pos[1]][1] == "a":
        if passenger.seat[0] > pos[0]:
          if plane[pos[0] + 1][pos[1]][0] is None:
            passenger.position[0] += 1
            plane[pos[0] - 1][pos[1]][0] = None
            plane[pos[0]][pos[1]][0] = passenger
        else:
          if plane[pos[0] - 1][pos[1]][0] is None:
            passenger.position[0] -= 1
            plane[pos[0] + 1][pos[1]][0] = None
            plane[pos[0]][pos[1]][0] = passenger

      # Else if in row but not in seat, move towards seat
      elif pos[0] == passenger.seat[0] and pos[1] != passenger.seat[1]:
        if pos[1] > passenger.seat[1]:
          passenger.position[1] -= 1
          plane[pos[0]][pos[1] + 1][0] = None
          plane[pos[0]][pos[1]][0] = passenger
          if pos == passenger.seat:
            seated[passenger] = True
        else:
          passenger.position[1] += 1
          plane[pos[0]][pos[1] - 1][0] = None
          plane[pos[0]][pos[1]][0] = passenger
          if pos == passenger.seat:
            seated[passenger] = True

    ticks += 1 # Update tick counter
    display(plane, ticks) # Clear console and display the plane and the tick
counter

  global runs # Allows us to update the runs variable within the function
  runs += 1 # Each time the simulation runs, add one to "runs"
  return ticks
```

```
# Examples of how to use the algorithm

# Default function call (default vals: P_d = 0.1, P_c = 0.87, and c_max = 1)
print(simulation(figure2.getWindowSeats(), plane2layout))

# Function call with different parameters (P_d = 0.3, P_c = 0.5, c_max = 2)
print(simulation(figure1.getRandomSeats(), plane1layout), 0.3, 0.5, 2))

# Generating a list to run the algorithm 100 times for one boarding method
print([simulation(figure1.getRandomSeats(), plane1layout), 0.3, 0.5, 2) for _ in
range(100)])

# Generating a list to test sensitivity of model to disobedience (P_d)
print([simulation(figure1.getWindowSeats(), plane1layout, n*0.1, 0.87, 1) for n in
range(11)])

# Generating a list to test sensitivity of model to carry-on chance (P_c)
print([simulation(figure1.getSectionedSeats(), plane1layout, 0.1, n*0.1, 1) for n
in range(11)])

# Generating a list to run the algorithm 10 times for one disembarking method
print([simulation(figure3.getColumnSeats(), plane3layout, 0, 0.87, 1) for n in
range(10)])

# Generating a list to run the algorithm 25 times for a 30%-capacity plane
print([simulation(changePercentPassengers(0.3, figure1.getColumnSeats()),
plane1layout) for _ in range(25)])
```