**Battleship Lab Report**

***Submitted to***

Dr. Corso Padova

***Prepared by***

Gold Team (Group 6)

Zach Lohrman

Zach Sentivany

Tyler Sullivan

Skylar Morley

**Engineering 1181**

The Ohio State University

College of Engineering

Columbus, Ohio

**December 11, 2019**

**Executive Summary**

The purpose of this lab was to design a functioning program that emulates popular board games. The game was to be user-interactive and written using the MATLAB program. The game chosen to replicate was Battleship, which is a two-player, turn-based strategy game. The design process began by breaking down the game into its fundamental components to establish what the program needed to do.

The coding process presented several obstacles including graphic representation, AI basic strategy, and nesting many functions together. Ultimately, the option that provided the most simplicity was chosen. The game is divided into ‘modules’ which each perform a specific task and allow for a shorter code and debugging process. The coders worked to prevent the game from malfunctioning due to invalid user input and provided thorough explanation to the user.

**Table of Contents**

List of Figures and Tables 3

1. Introduction 4
2. Software Documentation 5
   1. Introduction to Details of the Game 5
   2. User Manual 5
   3. Program Design for Developers 6
3. Outside Sources Used 25
4. Flowcharts 26
5. Discussion 28
6. Conclusion and Recommendations 30

**List of Figures and Tables**

Figure 1. Greeting Module 6

Figure 2. Difficulty Selection Module 6

Figure 3a. Tutorial for Ship Placement 7

Figure 3b. Grid Placement Tutorial 8

Figure 4a. Ship Placement Module 9

Figure 4b. Ship Reference Functions 10

Figure 4c. Ship Placement Function 11

Figure 4d. Ship Placement Coordinate System Function 12

Figure 4e. Column Assignment for Vertical Orientation 13

Figure 4f. Row Assignment for Vertical Orientation 14

Figure 4g. Ship Placement Coordinate Scan Function 15

Figure 5a. AI Grid Module 16

Figure 5b. AI Ship Placement Function 17

Figure 5c. AI Ship Coordinate Function 18

Figure 5d. AI Ship Placement Scan Function 19

Figure 6. Initialization of Global Variables 20

Figure 7a. Player Turn Module 21

Figure 7b. Shot Placement Scan Function 22

Figure 7c. Shot Check Function 22

Figure 8a. AI Turn Module 23

Figure 8b. AI Shot Placement Scan Function 23

Figure 8c. AI Shot Check Module 24

Figure 9. Ship Placement Flowchart 26

Figure 10. Gameplay Flowchart 27

**Introduction**

**The Project**

The team recreated the classic board game, Battleship, with MatLab programming. The main focus of this project was expanding knowledge on problem solving strategies, creating a fluid design process, and developing the team’s programming skills to create a playable game for a user’s enjoyment. The use of nested functions to simplify code was extensive and provided a challenge for the team.

**The Team**

Zach Lohrman: Recording, primary report writer

Zach Sentivany: Coder responsible for ship placement module

Tyler Sullivan: Coder responsible for gameplay module

Skylar Morley: Meeting notetaker

**The Design Process**

First, the team organized ideas of what was needed from the program to play the game using the ‘DR. PIE’ method from earlier in the semester. A list of knowns and unknowns was created to better illustrate where the focus of attention should be. From that list, pseudocode was written to more accurately describe what was needed as well as what functions and commands could accomplish the task. It was determined early on that an interactive game like Battleship that has several variables for each player and a large playing ‘board’ would require a large amount of code.

Because functions were the most recent MatLab tool taught in class, loops, conditionals, and simple matrices were used for the first iteration of the code. Very quickly, it was discovered that the code would be too long to realistically finish and fully debug in the allotted time. After discussing with a friend outside of class who is a computer programmer as well as amongst the team, functions were determined to be the best solution to the issue of length and simplicity.

The code was broken into two primary sections: the initial setup of the board and ships, and the interaction during the game involving the actual firing and sinking of ships. Using functions led to a much easier debugging process, but also presented its own new challenges while integrating functions together. Following the completion of the functions themselves, the primary challenges were limited to integration. Ultimately a successful game code was written, with the only issue being an ability to ‘break’ the game if the user were to intentionally take advantage of coding faults like placing a ship on top of an already placed ship.

**Software Documentation**

**Introduction to Details of The Game**

Battleship is a 2-player board game that involves placing 5 ship pieces of varying lengths on a 10x10 grid board, unknown to the other player. Each player takes turns ‘firing’ on the other player’s squares, until all of one player’s ships have been sunk. There is a large element of randomness to the game, as the players cannot see each other’s boards, only where they have fired shots previously. During the game only the player’s grid and fired shot locations are visible to the player.

**User Manual**

Restrictions within the game about things like ship placement were put in place to prevent the user from crashing the program or causing it to malfunction. The game begins by asking the user to select between easy or hard difficulty. The prompt informs the user that input is case sensitive and displays the two options available. Next, it asks the user if they would like to view a tutorial that explains how to play game by typing ‘help.’ The tutorial first describes how to use command window input to place ships correctly within the grid and avoid errors. Because the code relies on string compare (strcmp) to determine which action the user wants to make, the input must exactly match what was programmed. If a player does enter invalid input, rather than ending the game, it will simply ask them again. Within the explanation of how to setup the game in the tutorial, it informs the user of any restrictions on moves or input during play. All input is spelling and case sensitive.

Once the user has finished reading the tutorial, they begin placing their ships. There are 5 ships in total that occupy one unit on the grid by 5,4,3,3, and 2 units, respectively. Ships may be placed either horizontally or vertically; diagonals are not allowed. Ships must be entirely on the grid and trying to place one off of the grid will result in an error. Once all of the user’s ships have been placed, the computer’s ships will be randomly placed on a separate grid, invisible to the player. The computer and user take turns firing on each other’s grids, trying to hit the other player’s ships. When it is the user’s turn, the command window prompts them to input a grid coordinate that they wish to fire upon. The command window will then display the results of the shot, be it a hit or a miss. When all of either the computer or user’s ships have been hit, the game announces the winner to the user and ends the program.

**Program Description for Developers**

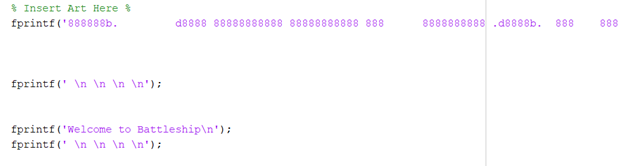


Figure 1. Greeting Module

The first module of the program is the greeting module. The first fprintf statement prints a greeting statement and the word “Battleship” across the command window using ascii art. Fprintf statements are used for blank, white space within the program.

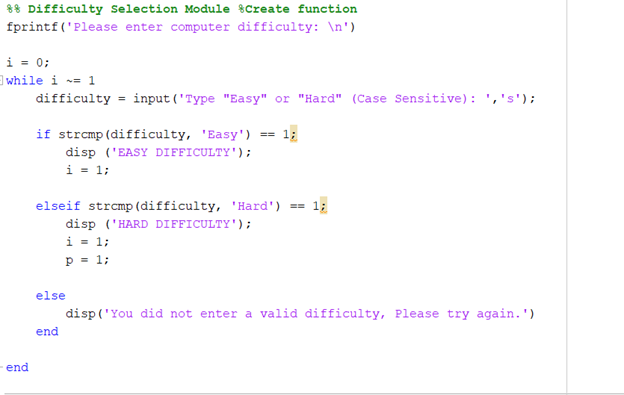


Figure 2. Difficulty Selection Module

First, the player is prompted to enter a difficulty. The player is given the option of easy or hard difficulty. The input must be case sensitive. Having case sensitive input prevents the player from entering invalid input. If the player enters an invalid input, an else statement captures it and will not allow the player to proceed until they have entered a valid difficulty. This decision is used in a later module. The function string compare (strcmp) is a logical used to compare two separate strings to each other, returning a 1 if the comparison is true and a 0 if it is false.

Variables

* **i**: sentinal variable for while loop; will not release while loop unless it is equal to 1.
* **p**: used later in program to reference the ‘hard’ difficulty
* **difficulty**: string variable used to determine user input

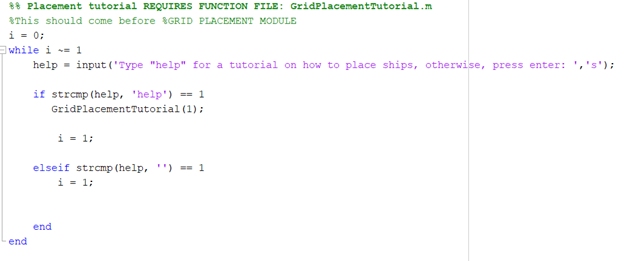


Figure 3a. Tutorial for Ship Placement

This module offers the player an opportunity to view a tutorial on how to place a ship on the gameboard. This module introduces the first programmed function ‘GridPlacementTutorial’ which will be covered in the next section. This is another string player prompt where if the player enters ‘help’, the program will display the tutorial. Otherwise, if the player enters no input, the game will continue to the next module. As in previous modules, the player must enter a specific input before proceeding. This step prevents the player from breaking the game.

Variables

* **i**: sentinel variable for while loop; will not release while loop unless it is equal to 1.
* **help**:a user input string, used to determine actions from within the while loop.

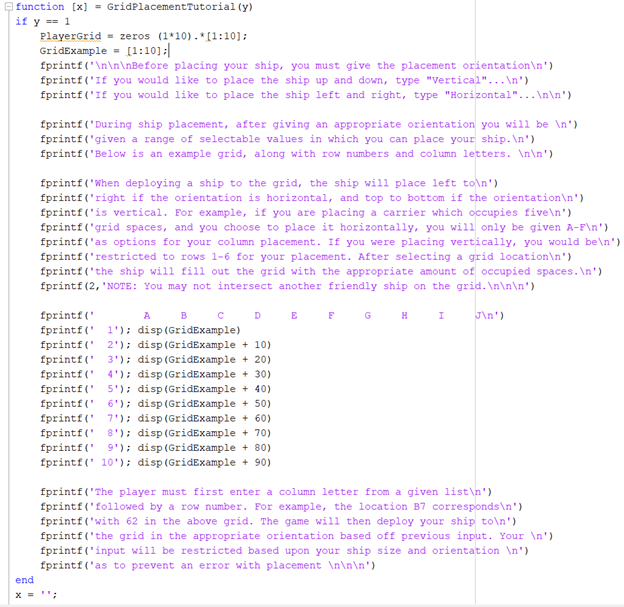


Figure 3b. Grid Placement Tutorial

This module contains the function for the grid placement tutorial, which depends upon input from the previous module. It uses fprintf statements to display text to the player. The module explains, step by step, how to setup the game board without breaking the program.

Variables

* **y**: used to initiate function.
* **x**: used as output to add white space below.

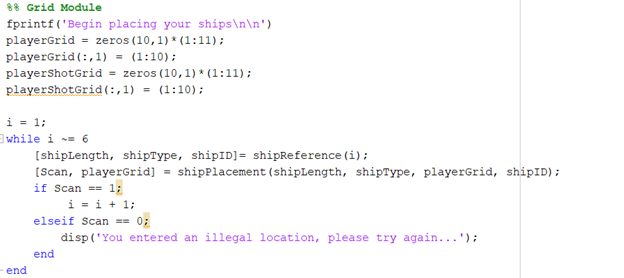


Figure 4a. Ship Placement Module

This is the first module that initiates a more complicated action from within the game. The first action of the module is to create a 10x10 empty grid for the player to place ships. **The grid is shown as a 10x11 grid because the first column is a reference row, not used for playing the game.** The loop shown in Figure 4a above is set to repeat until i = 6. This is because there are 5 ships in the game to be placed. This loop performs the same action for each ship type. This module calls the function ‘shipReference’and ‘shipPlacement’. If the player has entered valid input, it allows the program to proceed to the next ship type. This command is represented by the Scan variable.

Variables

* **i**: sentinel variable used to represent which ship the program is currently on.
* **shipLength**: output of ship reference; different for each ship type.
* **shipType**: output of ship reference; string variable used for fprintf statements
* **shipID**: output of ship reference; number used for grid placement.
* **playerGrid**: the grid, upon which the player will place ships, and the AI will target.
* **playerShotGrid**: 10x11 grid that keeps track of shots taken by the player.
* **Scan**: variable used to determine if the player entered valid input.

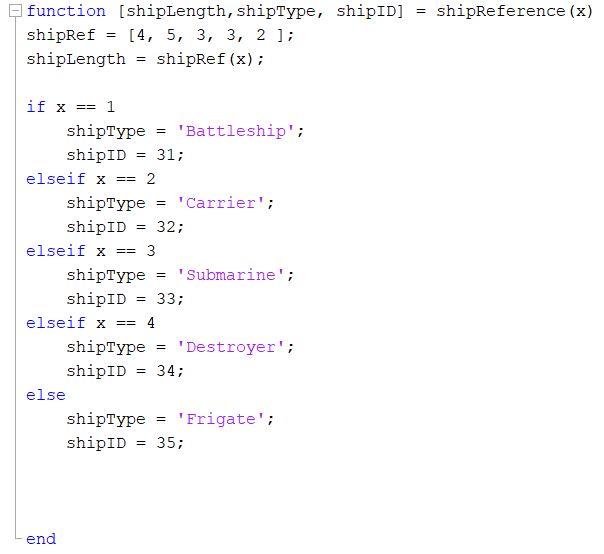


Figure 4b. Ship Reference Functions

The ship reference function gives the next function within the loop the appropriate information to place ships correctly. Within this function, the program gets information about the type of ship in a string format, the ID for grid placement, and the length of the ship. The outputs from this function are used in the next function.

Variables

* **x**: used as a reference for which ship the player is currently on. X corresponds with ‘i’ in the main module.
* **shipID**: an arbitrary number used to identify which ship occupies a place on the grid.
* **shipType**: string variable used for fprintf statements that identify which ship is being placed.
* **shipLength**: a number used to represent the length of each ship. Also used as a restricting variable in ship placement to prevent invalid input.
* **shipRef**: a vector used to identify the length of ships with element referencing.

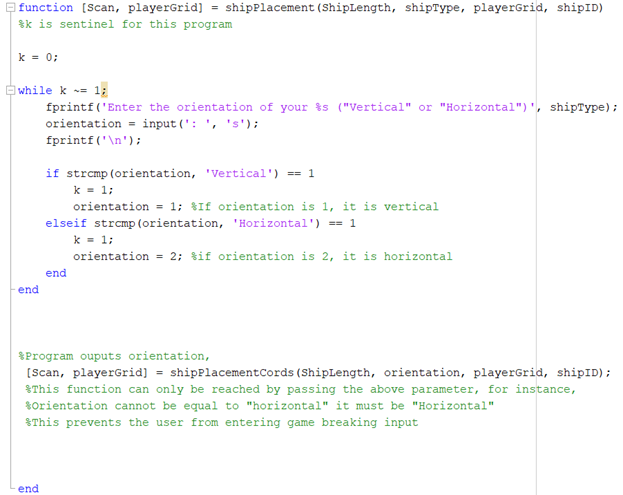


Figure 4c. Ship Placement Function

The first part of this function prompts the user to select the orientation of the ship; vertical or horizontal. The player must enter valid input, in order to proceed to the next step. Based upon player input, the orientation variable is set to either 1 or 2 and used in the next function. It is important to note that the player cannot advance to the next layer of this function, unless valid input is entered. This prevents the player from breaking the game.

Variables

* **orientation**: string variable used to determine which direction the player places their ships. Once valid input is entered, the string is converted to a standard numeric variable, which is used in the next function.

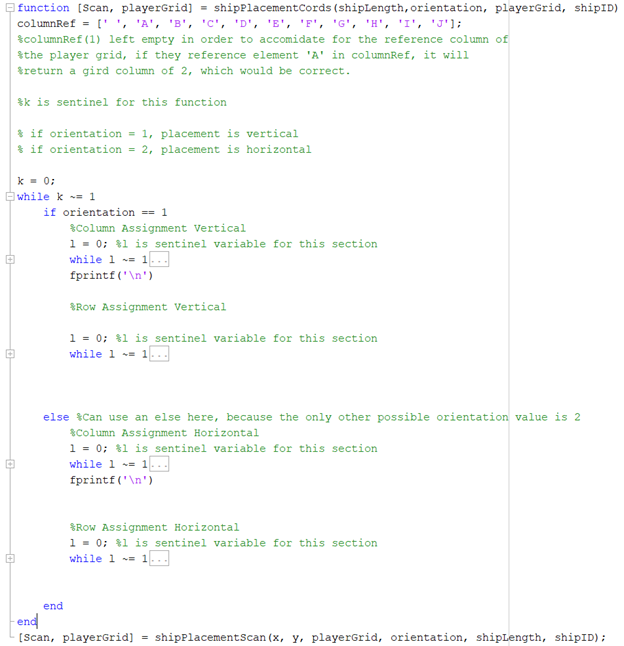


Figure 4d. Ship Placement Coordinate System Function

The purpose of this function is to convert player input into coordinates on the player grid. The function runs two different sets of commands based on input from the previous function, which is represented by the orientation variable. One set is for vertical placement, and the other is for horizontal placement. Only the vertical commands will be examined for the sake of redundancy.

Variables

* **columnRef**: vector used to convert player input to numeric values.
* **x**: corresponds with the row number on the grid.
* **y**: corresponds with the column letter on the grid.

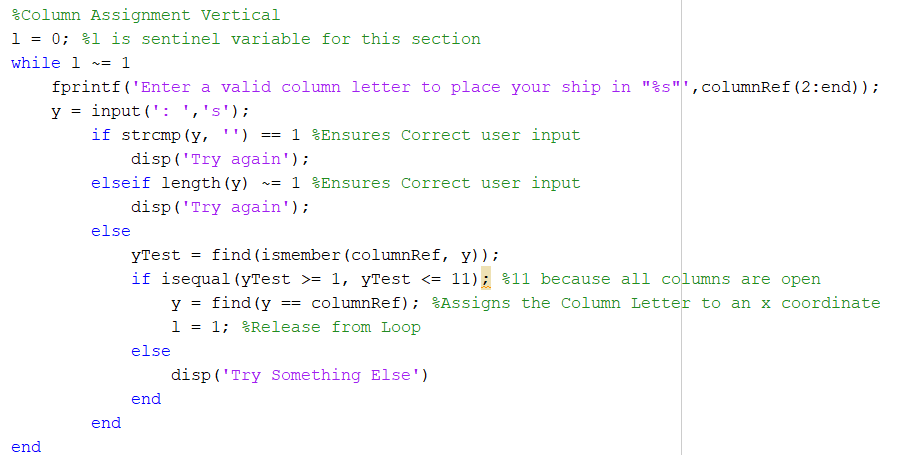


Figure 4e. Column Assignment for Vertical Orientation, shipPlacementCords function

This portion of code is in place to ensure that the user places ships in the correct orientation. The variable columnRef from the previous figure is used to assign a number to a string from player input. Any if, elseif, and else statements are used to ensure valid input. The player can only proceed to the next step if the first ‘else’ statement has been satisfied.

Variables

* **y**: begins as a string and is converted to a numeric variable; used for column placement on player grid

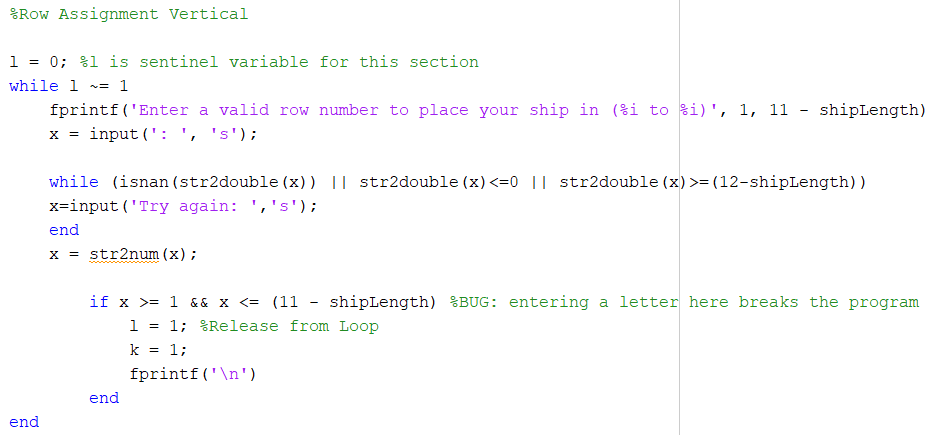


Figure 4f. Row Assignment for Vertical Orientation, shipPlacementCords function

This portion of code within the shipPlacementCords function assigns a row number to the variable x. Within this code, x begins as a string, despite the player being prompted for numeric input. Input is brought in as a string to prevent the program from recognizing the input as a previously defined variable and crashing. While this variable is a string, it is verified to be a number that fits within the grid in the second while statement. By subtracting the ship length from the number 12, it prevents the user from placing a ship in a position that would allow it to extend off the grid. Once these checks have been verified, the string is converted back to a number for use in later functions. The same process, but with restrictions on columns, is performed by similar code for horizontal orientation.

Some more complicated commands are used within this code. The ‘is not a number’ command (isnan) is used to verify that there is a number in the string x. The ‘string to double’ command (str2double) is used to convert numbers from within a string to a numeric variable. The ‘string to number’ command (str2num) is used to initiate the conversion to a number.

Variables

* **x**: begins as a string and is converted to a numeric variable; used for row placement on player grid.

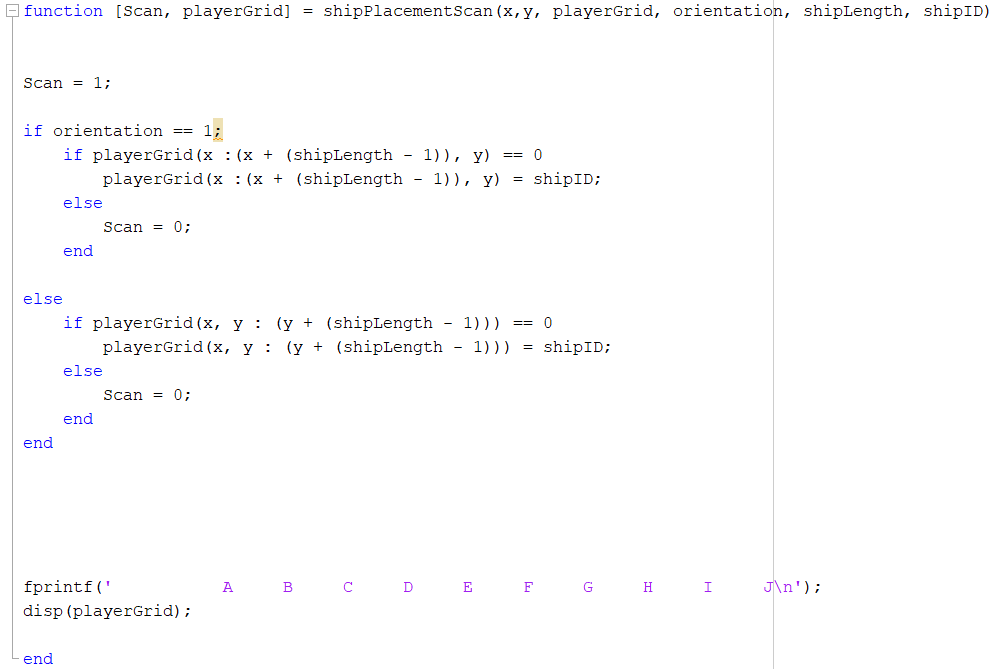


Figure 4g. Ship Placement Coordinate Scan Function

This is the final nested function within the ship placement module of the game. This function uses an if statement to ensure the player does not put ships on top of each other on the grid, thereby breaking the game. At the beginning of this function, Scan is set to 1; the number 1 meaning that there is nothing assigned to that position. The function then proceeds to scan each position that the ship will occupy until either complete or reaching an obstruction. In which case, scan will be set to 0 and the user will be sent back to ship placement. In Figure 4a, if this function outputs the number 1 for the variable Scan, the while loop proceeds to the next ship to be placed. The process is repeated until all ships are placed. At this point, all variables that are needed are output.

Variables

* **Scan**: gatekeeper variable that allows the program to proceed to the next ship to be placed.

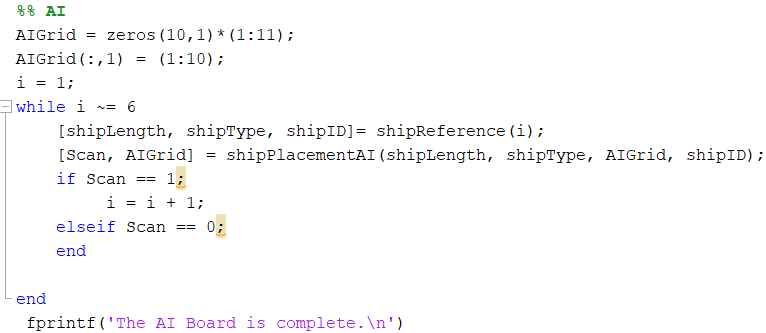


Figure 5a. AI Grid Module

The first lines of code establish a 10x11 grid of zeroes that function as the AI player grid. The grid is 10x11 rather than 10x10 for the same reasons stated before in Figure 4a. The while loop initiates the ship placement and is nearly identical to the while loop from 4a. After this has been completed, the user will be notified that the AI grid is complete.

Variables

* **AIGrid**: serves the same function as **playerGrid** in Figure 4a.
* All other variables are identical to those used in Figure 4a.

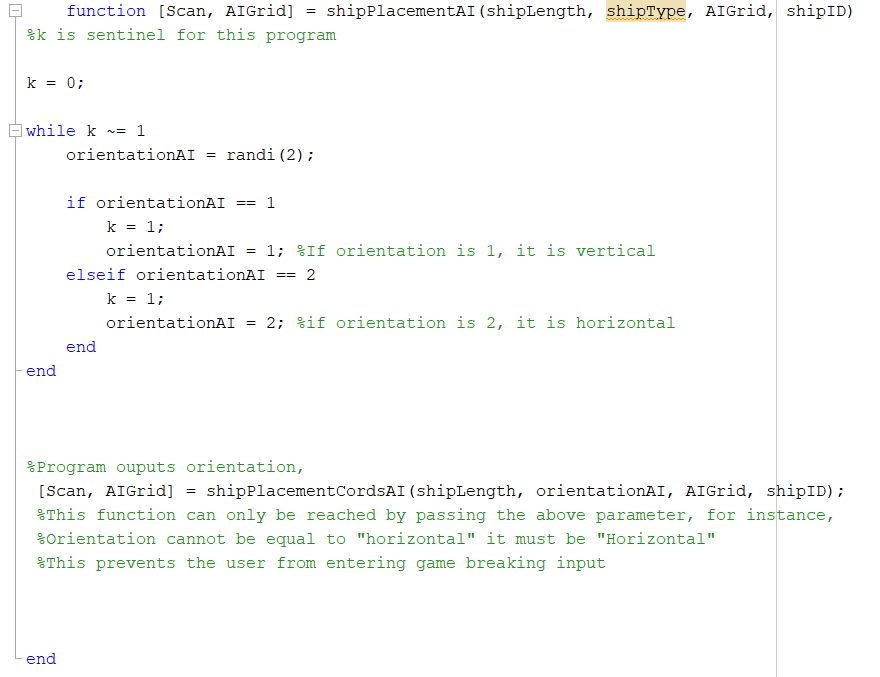


Figure 5b. AI Ship Placement Function

This function determines the orientation of the AI ships by generating a random number, either a 1 or 2. The function of this portion of code is the same as seen in Figure 4c but replacing player input with randomness from the AI.

Variables

* **orientationAI**: serves the same function as **orientation** in Figure 4c, but for the AI.

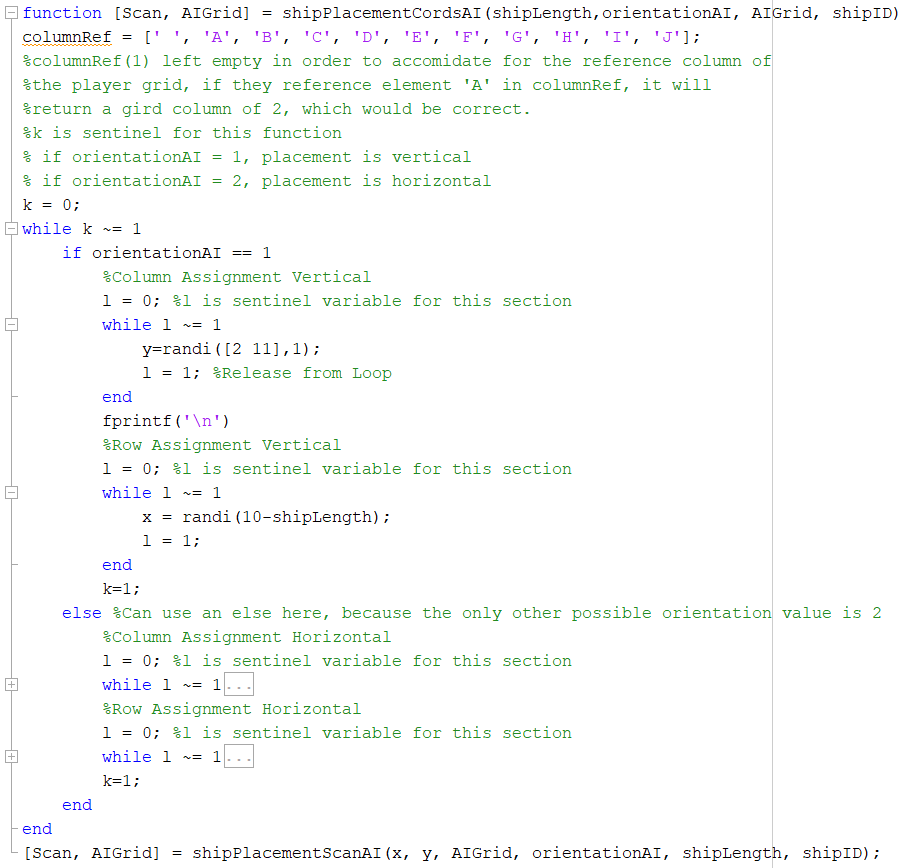


Figure 5c. AI Ship Coordinate Function

This function randomly generates coordinates within the AI grid. Restrictions are based on the value of **orientationAI** from the function in Figure 5b. The placement works the same as the function in Figure 4d, but for the AI.

Variables

* **x**: a randomly assigned row value for the ship to be placed on in the AI grid.
* **y**: a randomly assigned column value for the ship to be placed on in the AI grid.

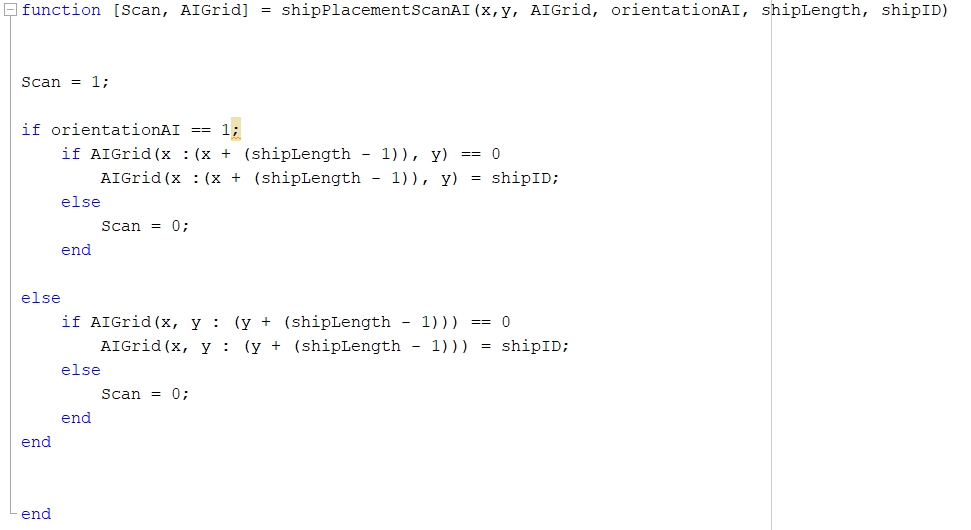


Figure 5d. AI Ship Placement Scan Function

This function verifies that there are no ships where the current ship is to be placed. It functions identically to the function in Figure 4g, in the way that it will not proceed until valid coordinates have been generated.

Variables

* **Scan**: functions the same as **Scan** in Figure 4g, but for the AI.

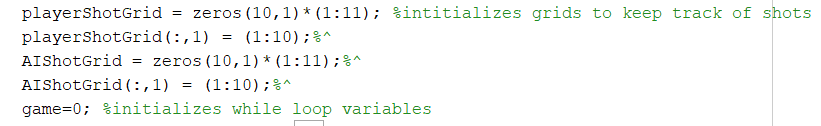
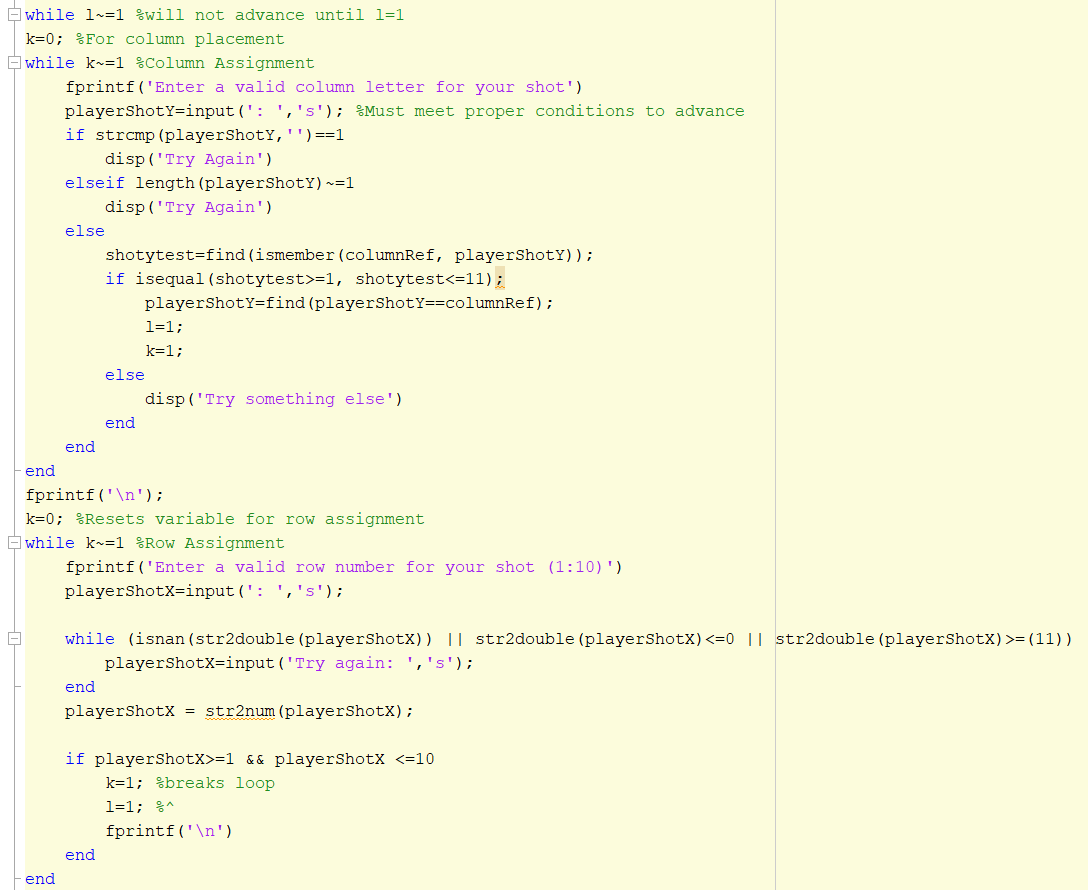


Figure 6. Initialization of Global Variables

The purpose of this code is to create variables that will be used later to keep track of where shots have been taken throughout the game. Two 10x11 grids of zeroes are created for the program to reference.

Variables

* **playerShotGrid**: a grid to keep track of shots that the player has taken.
* **AIShotGrid**: a grid to keep track of shots that the computer has taken.
* **game**: runs program while equal to zero and will only be changed when victory conditions have been met, ending the game.



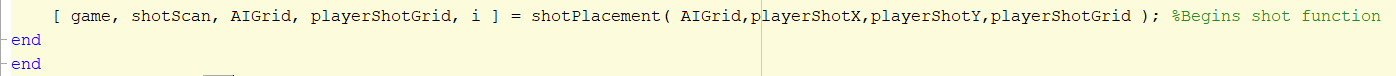


Figure 7a. Player Turn Module

This module is used to generate coordinates for player shots using input commands that have been verified by several while loops for valid input. Once shot coordinates have been entered by the player, the module will begin the shot placement function in Figure 7b. The column assignment works identically to the function in Figure 4e. Similarly, the row assignment works identically to the function in Figure 4f.

Variables

* **playerShotY**: string input used to determine column assignment for player shots.
* **shotytest**: used to ensure that **playerShotY** is valid input.
* **playerShotX**: string input used to determine row assignment for player shots.

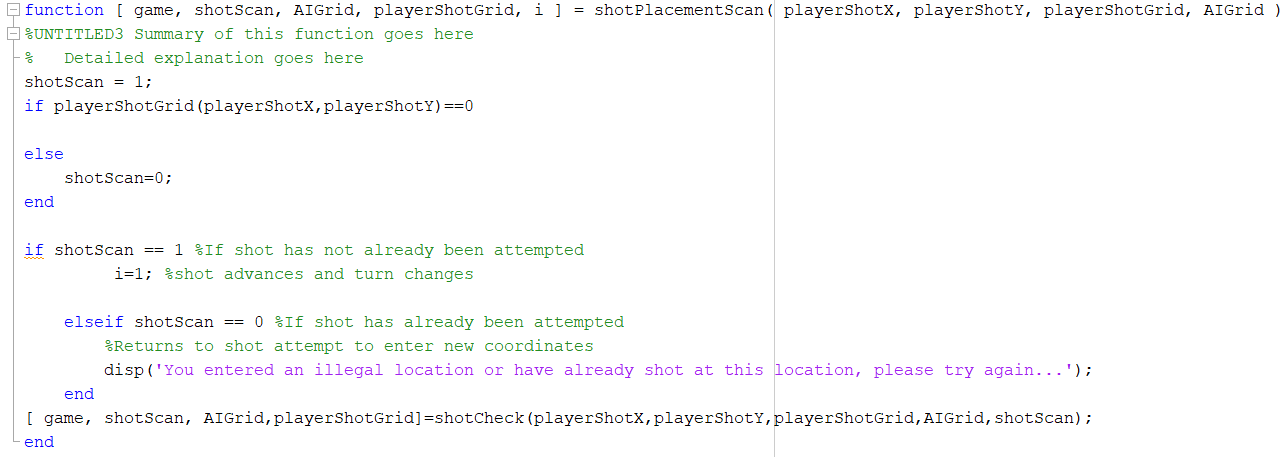


Figure 7b. Shot Placement Scan Function

The purpose of this function is to make sure the player does not repeat a shot by scanning the player shot grid and comparing it to the user inputs for **playerShotX** and **playerShotY**. If the shot has already been attempted, the user will be sent back to enter new shot coordinates.

Variables

* **shotScan**: works identically to the variable **Scan** from Figure 4g.

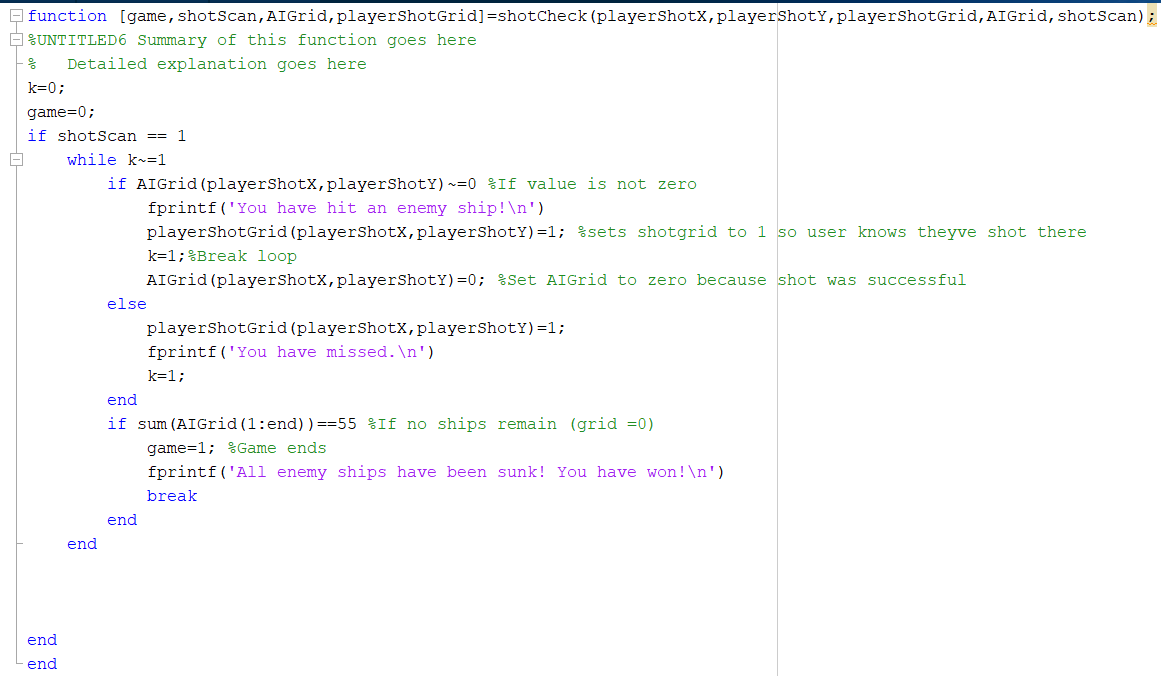


Figure 7c. Shot Check Function

The purpose of this function is to determine whether the player’s shot has missed or hit an enemy ship. Regardless of the outcome of the shot, its location is recorded in the player shot grid. This function also refers to the AI grid to scan for any remaining ships. If there are no remaining ships, the variable **game** is changed to 1, ending the game.

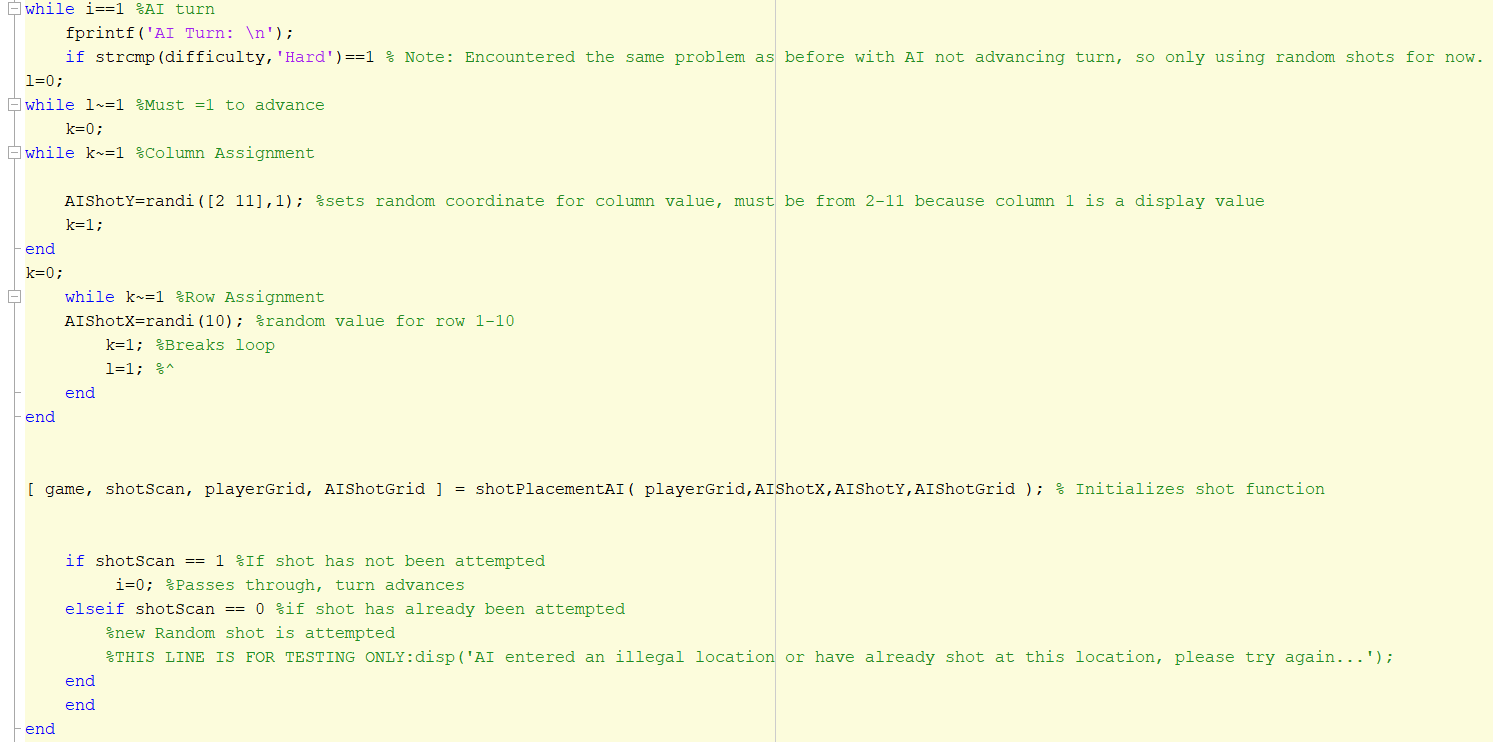


Figure 8a. AI Turn Module

The purpose of this module is identical to the module in Figure 7a, but for the AI.

Variables

* **AIShotY**: generates a random value for the column coordinate used in AI shot placement.
* **AIShotX**: generates a random value for the row coordinate used in AI shot placement.
* **shotScan**: functions the same as **shotScan** in Figure 7c, but for the AI.

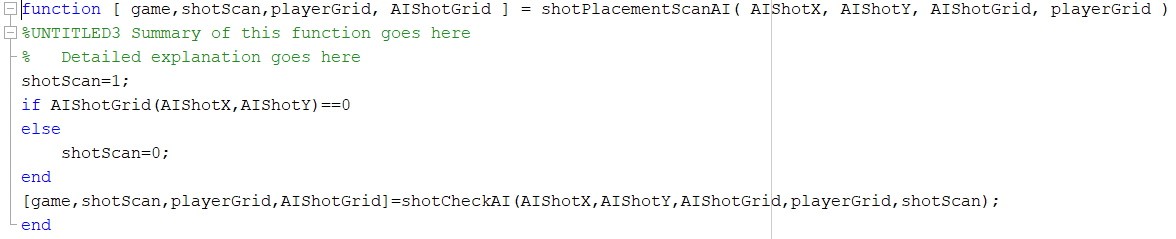


Figure 8b. AI Shot Placement Scan Function

This function serves the same purpose as the function in Figure 7b, but for the AI.

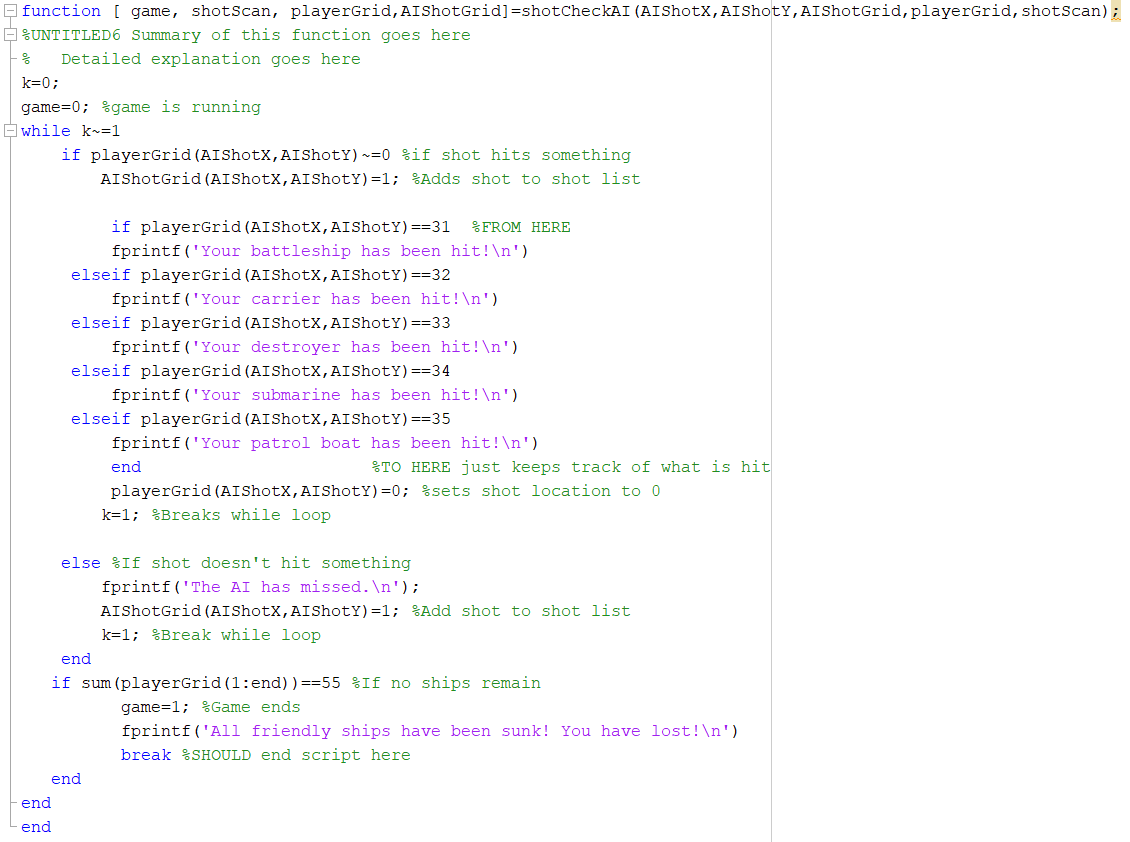


Figure 8c. AI Shot Check Module

The purpose of this function is to check if an AI shot has hit or missed a player’s ship. This function works similarly to the function in Figure 7c, with the difference being the program informing the player which ship has been hit.

**Outside Sources Used**

Congos, V. (2019, November 23). Personal Interview.

MATLAB Answers. (n.d.). Retrieved from <https://www.mathworks.com/matlabcentral/answers/index>.

MATLAB Help. (n.d.). Retrieved from <https://www.mathworks.com/help/matlab/>.

**Flowcharts**

**A close up of a map

Description automatically generated**

Figure 9. Ship Placement Flowchart

A screenshot of a cell phone

Description automatically generated

Figure 10. Gameplay Flowchart

**Discussion**

**Testing Process**

The team began with a pseudocode that relied heavily on loops, matrices, and conditional statements; building off techniques the team had learned over the semester. The second iteration of the design was a more complicated approach, using object-oriented programming to create multiple states of matrices and interact with them using predefined functions. The programmers originally set out to write two different programs, just in case the other failed or took too long to develop. In the end, the two were merged. The following are the difficulties encountered while writing the code based on conditional statements.

**Issues and Obstacles**

As with any complicated program, there will be roadblocks in the form of bugs and malfunctioning code. What follows is a list of issues encountered by the coders during the construction and testing process. Originally, the intention was to use many nested conditional statements, overlooking the potential benefits of functions. As a result, the code for the actual playing portion of the game was over 900 lines long. When bugs were reached, they were very difficult to find. Following the bug phase of the conditional-heavy code, it was redesigned using separate function files to separate out different portions of the code. The drawback that was immediately noticed with functions was its integration with the rest of the code. If there is an error in the function, the whole code will malfunction.

The first issue encountered was keeping the game functional when invalid input was entered by the user. For example, if the user is prompted to enter a number, but instead they enter a letter, MatLab will interpret it as a variable and cause the program to crash when that variable is called later. The coders managed to get around this problem in a somewhat inefficient way by converting the number to a string, using ‘strcmp’ to check if the input is valid, and converting it back to a number if so. This method also allows the program to loop back and return to prompting the user for the same information without crashing.

The next issue was determining how to end the game. The coders analyzed to code for a way to scan the grid and declare a winner. Because the number of positions on the grid is known, it was decided to perform a scan of the grid and sum up the numbers. If a pre-programmed number was found by the scan, the game would announce the winner and end.

The issue of changing players for each turn was originally designed to be controlled by a counter variable, which adds one to itself each turn. Using the rem function (rem()), the logical would determine if the number was even or odd and begin the turn of the appropriate player. The final iteration of the player change utilizes a switch case command. At the end of the turn, the loop will switch the player counter variable back and forth, allowing for the player and computer to exchange turns.

An area of much focus by the coders was in the prevention of program failures as a result of invalid user input. For example, if either player tried to place a ship on a preexisting ship or off the grid, it would cause the game to crash. Code was put in place specifically to prevent user error. When a player enters input such as the orientation of their ship or selecting difficulty, the program converts that input to a string, and using the string compare function (strcmp), evaluates the input for validity and returns a one or zero to correspond with true and false, respectively. If the logical returns a zero, it loops back and repeats the same action, asking the user for the same input. If the logical returns a one, it converts the string back to a numeric value and proceeds to the next step.

**Conclusion and Recommendations**

Following the completion of the game, object-oriented programming was determined to be the simplest and most efficient approach to this type of game. By creating simple and translatable functions, the amount of time needed to code a relatively complex program is reduced significantly. If the team were to redo this project, the same initial approach would be taken. A ‘DR. PIE’ list would be created, however the pseudocode would be based around nested functions rather than loops and conditionals. If the team had more time to fine-tune the program a number of upgrades could be created. The board would have a more eye-catching design, sound effects would be added, and the ships would be represented by a graphic, rather than numbers. Putting the coding and nesting function practice aside, the team’s primary takeaway from this project was the importance of time, both in organization and in simplicity of code.