BATTLESHIP

By Group 37

Frederik Tots s245710, Simon Christophersen s245805 & Mads Madsen s233971

Content

1 Introduction

Battleship is a strategic game for two players that originated in the early 20th century.

The game progresses on a square grid that is typically 10×10, within which each player hides a fleet of ships from the other. Players take turns calling coordinates on their opponent's grid in an effort to target and sink all enemy vessels. The long-range subs, located on either player's grid, cannot be located with a simple call to "1" through "10" or the equivalent in letters and numbers. The player's calls and the opponent's responses to whether those calls resulted in hits or misses form a kind of dialogue between the two players.

This report presents a MATLAB implementation of Battleship featuring three distinct AI difficulty levels.

Every stage utilizes increasingly intricate algorithms to compute the best shooting solutions. The implementation puts into practice fundamental ideas from probability theory, search methods, and gameplay state handling, while yielding a working and engaging game.

The main technical obstacle in the game is devising effective search algorithms that can find well-concealed ships with the fewest possible shots. This project takes on the challenge to not just solve the problem, but solve it in many different (and sometimes contrary) ways, to allow observed and noted comparisons of the computational efficiencies of those different ways over different problem difficulties.

2 User Guide

We designed the game for a single player who plays against the computer. The player can choose between three different degrees of difficulty to play against.

After choosing the degree of difficulty, the player must set up their fleet on a 10x10 board. The fleet consists of three warships: the first one is four squares long, the second ship is three squares long, and the last one is two squares long. All the warships must be placed either vertically or horizontally.

After setting up your fleet, you must lay out a strategy on how to hit the computer's ships faster than the computer can sink all your ships.

Now you and the computer take turns shooting at each other's ships. With a good strategy and a bit of luck, you will come out as the winner.

2.1 Program Execution

The Battleship implementation centers on battleshipGUI.m as the main entry point.

The program's architecture adheres to a modular design with three main parts: Core Game Logic (which manages the game's state, ship placement, and move validation), AI Components (which implement strategies for difficulty-based computer opponents), and User Interface (which manages the game's visualization and player interaction).

2.2 Interface overview

The three principal sections of the graphical interface are a control panel with selections for game difficulty and controls, the player's grid for the placement of ships and for incoming shots, and the grid for targeting the opponent. The game's visual state is communicated to the player through a system of icons and colors. Ships are represented as blue rectangles. Hits are indicated by red squares that also contain an X. Missed shots are indicated by the presence of a circle. The system of color and iconography allows the player to see, at a glance, the state of play.

2.3 Game Rules

The standard Battleship rules are used: players position their ships on a 10×10 grid, then alternate turns declaring where in the grid they think the enemy has placed ships, with the goal of locating and sinking those vessels. The player whose ships are all sunk first loses. An immediate feedback system is used: after each shot the declaring player is told whether it hit or missed ("splash!"). No player can use any kind of timer while taking his or her turn.

2.4 Strategy Guide

The best possible play embodies not only effective ship placement but also efficient targeting strategies. For maximum defensive strength, as well as offensive opportunity, arrange ships asymmetrically and with maximum separation. When targeting, employ a checkerboard pattern for your first shots. After achieving your first hit, make systematic adjacent-cell shots. If you achieve multiple hits, then extend the identified ship orientation to maximize information gain.

3. Mathematical, Algorithmic Foundational Implementation

3.1 Overview of the solution approach

The solution for the Battleship game that has been implemented uses a modular architecture and hierarchical function organization. The core algorithm implementation is centered around the function "getComputerShot.m," which contains three different targeting algorithms that are based on the difficulty level of the game. This function encapsulates the primary decision-making component that centers around the function "getComputerShot.m," and it determines the optimal shot coordinates. This occurs despite some appearance of random behavior that is based on the computer's use of previously acquired knowledge of the game state.

3.1.1 Mathematical Foundation and Probability Model

The game can be modeled as a probabilistic search problem where the probability P of a ship being at position (i, j) evolves with each shot.

3.1.1.1 Mathematical Foundations

The game state in Battleship can be formally represented using matrix notation. We define two primary grid matrices for each player:

$$G_{player} = [g_{ij}]_{10*10}$$

$$G_{computer} = [g_{ij}]_{10*10}$$

Where element g_{ij} represents the state of the cell at position (i, j) with the following values:

- $g_{ij} = 0$, represents empty water
- $g_{ij} = 1$, represents a Battleship (4 cells)
- $g_{ij} = 2$, represents a Cruiser (3 cells)
- $g_{ij} = 3$, represents a Destroyer (2 cells)

Similarly, we track shots using separate matrices:

$$S_{player} = [s_{ij}]_{10\times10}$$

$$S_{computer} = [s_{ij}]_{10\times10}$$

Where:

- $s_{ij} = 0$, indicates no shot has been fired at position (i, j)
- $s_{ij} = 1$, represents a miss
- $s_{ij} = 2$, represents a hits

These matrices are implemented in our code as follows in *"battleshipGUI.m"*:

```
% Initialize game data
gameData.playerGrid = zeros(10, 10); % Corresponds to G_player
gameData.computerGrid = zeros(10, 10); % Corresponds to G_computer
gameData.playerShots = zeros(10, 10); % Corresponds to S_player
gameData.computerShots = zeros(10, 10); % Corresponds to S_computer
```

3.1.1.2 Probability Model

Initially, with $N = 9$ ships occupying M total cells on a 10×10 grid, the prior probability is uniform:

For our implementation with ships of lengths $L_1 = 4$ (Battleship), $L_2 = 3$ (Cruiser), and $L_1 = 2$ (Destroyer):

Total ship cells: $M_{total} = \sum_{i=1}^{3} M_i = 4 + 3 + 2 = 9$

Base hit probability for a random shot: $P\big(Ship\ at\ (i,j)\big) = \frac{M_{total}}{100} = \frac{9}{100} = 0{,}09$

This means that in a completely random shooting strategy, approximately 9% of shots would hit a ship.

After each shot, we update this probability based on hit/miss information using Bayes' rule:

$$P\big(Ship\ at\ (i,j)\big|miss\ at\ (k,l)\big) = \frac{P\big(Ship\ at\ (i,j)\big)\big) \times P(miss\ at\ (k,l)|Ship\ at\ (i,j))}{P\big(miss\ at\ (k,l)\big)}$$

Similarly, when we register a hit, we update our beliefs about surrounding cells since ships occupy contiguous spaces.

Our three algorithm implementations vary in how they exploit this probability model:

- Easy: Uses uniform random sampling (ignores probability updates)
- Medium: Updates conditional probabilities for adjacent cells after hits
- Hard: Further refines probability estimates using pattern recognition

Thus, therefore our ship configuration looks the following:

- $P_{hit,1}(Battkeship) = 0{,}04 \approx 4\%$
- $P_{hit,2}(Cruiser) = 0{,}03 \approx 3\%$
- $P_{hit,3}(Destroyer) = 0{,}02 \approx 2\%$

These baseline probabilities help us evaluate the efficiency of our targeting algorithms against pure chance.

3.2 Algorithmic Difficulty Levels

3.2.1 Easy Mode

The computer shoots randomly. It chooses a random position using "randi" and checks if the chosen square has been shot at before.

```
% EASY MODE - random shots
    if difficulty == 1
        validShot = false;

        while ~validShot
            % Random position
            row = randi(rows);
            col = randi(cols);

            % Check if already shot at this position
            if shotGrid(row, col) == 0
                validShot = true;
            end
        end

        return;
    end
```

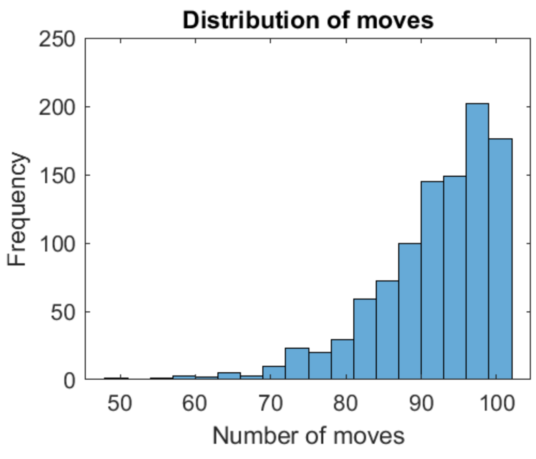


Figure 1 - Number of moves (easy)

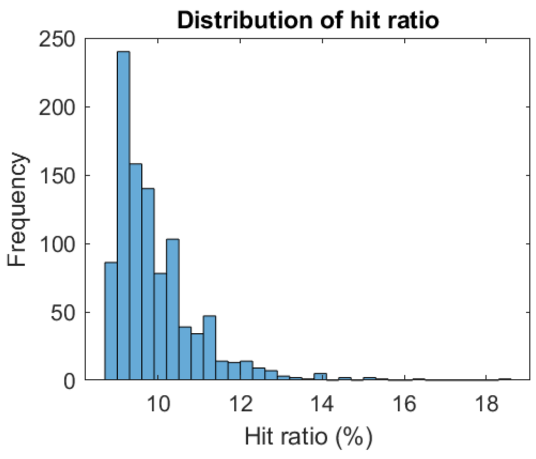


Figure 2 - Hit ratio (easy)

The random targeting algorithm can be formalized as follows:

Input: Shot grid S of size $n \times n$

Output: Next shot coordinates (i, j)

1. Define $U = \{\, (i,j) \mid 1 \leq i,j \leq n \ and \ S[i,j] = 0 \,\}$, as the set of untried positions.
2. If U is empty, return null (no valid shots remain)
3. Return random element (i, j) from U

The expected number of shots required to hit all 9 ship cells under this random strategy follows the coupon collector's problem with the expected value:

$$E[T_{eandom}] \approx \frac{100}{9} \times (1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{9}) \approx 91.4$$

This closely aligns with our empirical result of 91.06 average shots in the easy mode simulation.

3.2.2 Medium Mode

This algorithm employs a hunt-and-target strategy that significantly betters random targeting. As mentioned in the getComputerShot function (see Appendix E), the medium-difficulty algorithm systematically looks for the previous hits it has made and targets the adjacent cells in the cardinal directions. Using this approach, it employs positional information to constrain its search, so it doesn't have to revert to random shooting until it's exhausted any previously made hits.

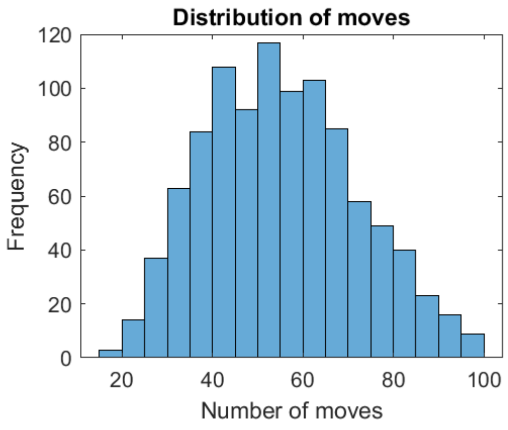


Figure 3 - Number of moves (medium)

The two-pronged operational mode achieves a neat balance between board exploration and targeted exploitation. This effectively improves upon the easy mode, which takes a purely stochastic approach and therefore wallows in the local optima that it finds:

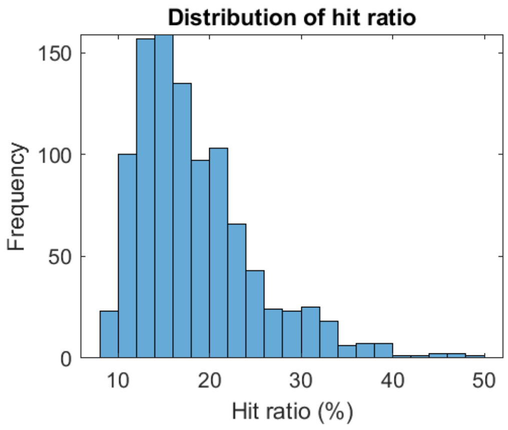


Figure 4 - Hit ratio (medium)

```
Input: Shot grid $S$ of size $n \times n$

Output: Next shot coordinates $(i, j)$

  1. Define $H = \{ (i,j) \mid S[i,j] = 2 \}$, as the set of successful hits.
  2. If $H$ is not empty:
     • For each $(i,j) \in H$:
         - Define $A = \{ (i-1,j),\ (i+1,j),\ (i,j-1),\ (1,j+1) \}$ as adjacent cells.
         - Define $V = \{ (x,y) \in A \mid 1 \leq x,y \leq n \ and \ S[x,y] = 0 \}$ as valid shots
         - If $V$ is not empty, return random element from $V$
  3. If no target is found, apply ***RandomTargeting*** algorithm
```

The efficiency of this algorithm can be modeled as a two-phase process:

1. Random phase: Expected $\frac{1}{0.09} \approx 11.1$ shots to find the first hit.
2. Targeting phase: Expected 2-3 shots to find subsequent cells of the same ship

This results in a theoretical efficiency factor $\alpha \approx 0.67$ compared to random targeting:

$$E[T_{medium}] \approx E[T_{random}] \times \alpha \approx 91.4 \times 0.67 \approx 61.2$$

Our empirical result of 61.66 average shots validates this theoretical model.

3.2.3 Hard Mode

This algorithm executes an improved targeting hierarchy to maximize efficiency in hitting the opponent's ships. As described in the getComputerShot function (see Appendix E), the hard difficulty level first examines the shot grid for any linear patterns comprising consecutive hits (i.e., the computer has previously hit at least two cells in a row).

If the hard level finds such a pattern, it attempts to extend that pattern in the identified direction to infer the ship's orientation. If the hard level does not identify any such linear hit patterns, it reverts to the adjacent cell targeting strategy of the medium difficulty level (see next section).

For scenarios where the computer has not previously hit any cells, the algorithm employs a checkerboard pattern in targeting. The use of this pattern significantly reduces the size of the search space that the computer has to work with.

The targeting possibilities are progressively narrowed by this systematic approach, based on the information that has been accumulated about the game state. This narrows the possible targets to which a given shot can effectively reach, making the shot much more efficient than if it had been fired to some other, less favorable target:

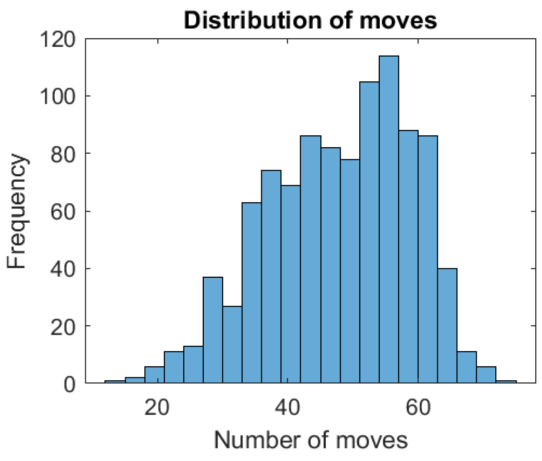


Figure 5 - Number of moves (hard)

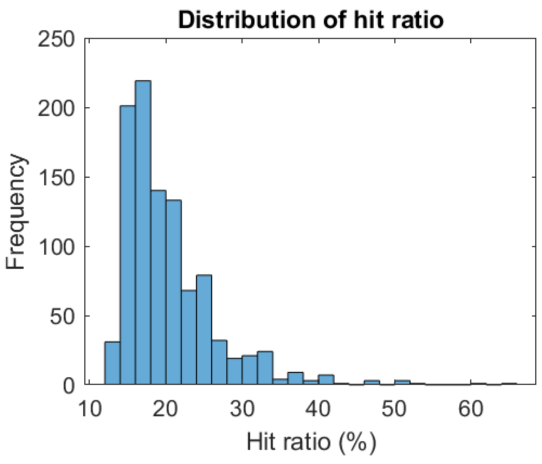


Figure 6 - Hit ratio (hard)

Input: Shot grid S of size $n \times n$

Output: Next shot coordinates (i, j)

```
1. Search for linear patterns in hits:
    • Define $H = \{ (i,j) \mid S[i,j] = 2 \}$, as the set of successful hits
    • For each pair $(h_1, h_2) \in H^2$, where $h_1$ and $h_2$ are adjacent:
        - Calculate direction vector $d = (i_2 - i_1, j_2 - i_1)$
        - Define extended positions $e_1 = (i_1 - d_1, j_1 - d_2)$, $e_1 = (i_2 - d_1, j_2 - d_2)$
        - If either $e_1$ or $e_2$ is valid and untried, return it.
2. If there is no linear pattern, apply HuntAndTargetStrategy
3. if no target from steps 1-2, optimize using probability density:
    • Apply checkerboard pattern where $P[i,j]$ is higher when $(i + j) \bmod 2 = 0$
    • Return position $(i, j)$, with highest probability $P[i,j]$
```

The theoretical efficiency factor for this algorithm compared to the medium strategy is $\beta \approx 0.85$:

$$E[T_{hard}] \approx E[T_{medium}] \times \beta \approx 61.2 \times 0.85 \approx 52.0$$

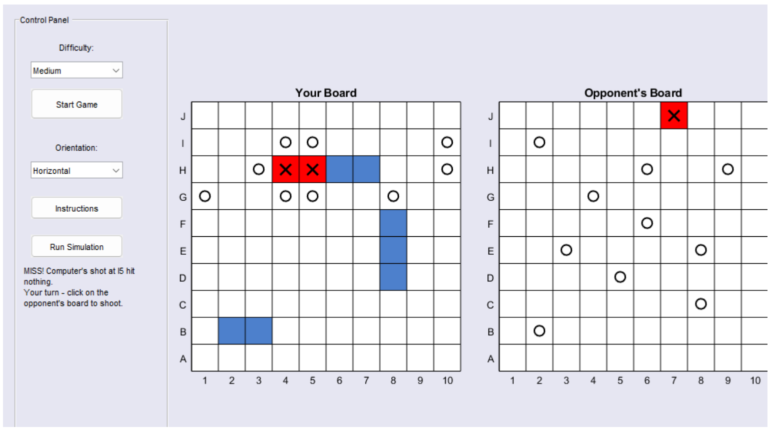
Our empirical result of 52.19 average shots confirms this theoretical prediction.

4. Visualization and User Interface

The Battleship game implementation has a complete graphical user interface (GUI) that is built with the MATLAB figure and uicontrol components.

This section will talk about the architecture and key implementation parts of the GUI.

4.1 GUI Architecture



MISS! Computer's shot at I5 hit nothing.
Your turn - click on the opponent's board to shoot.

Figure 7 - User Interface from MATLAB

The interface consists of three primary elements: a control panel on the left side with game options and status info, the player's grid in the center showing ship placements and the opponent's shots, and the opponent's grid on the right displaying the player's shots. This layout decomposes the game into an intuitive separation of elements, while still maintaining visual coherence.

The current game phase, ship locations, and shot history are kept in a single, centralized data structure. This makes it easy to track changes in the game state. It also makes it possible to write a very efficient update function that changes the visual elements of the GUI when the state of the game changes.

4.2 Interactive Elements

Interaction with the user occurs through MATLAB's event-driven callback system. Two principal interaction modes are set up: ship placement mode, in which players position their fleet on the grid, and shooting mode, in which players fire at the opponent's grid. The game automatically progresses through these modes.

The game gives visual feedback by color coding. Ships are represented by blue rectangles. Hits are shown by red squares with X markers. Missed shots are indicated by circles. This way, the players don't have to read anything to know the game state. They can see it instantly if they can look at the grid. Our function updateGridDisplay (see Appendix E) does the hard work for us of transforming the current numerical game state matrix into these visual elements.

4.3 Interactive Elements

The interface updates in real time following each player or computer action, with short animation delays to help maintain gameplay flow. The control panel issues all sorts of contextual information about what's happening in the game when it's happening—messages about hits, near misses, or whole ships being sunk, for instance. This happens at the same time as the panel is visually portraying all sorts of subtle state-changing actions that the game itself could be using as excuses for doing what it does. Figure 7 shows the game interface during an active gameplay session, with the difficulty selector, game controls, and both player and opponent boards visible.

5 Results

We evaluated our Battleship implementation across three difficulty levels through simulation experiments. This section presents key findings from these tests.

5.1 Performance Analysis Across Difficulty Levels

5.1.1 Computational Complexity Analysis

The time complexity of our targeting algorithms varies by difficulty level:

- **Easy (Random)**: $O(1) -$ Constant time to generate random coordinates.
- **Medium (Hunt and Target)**: $O(|H|)$ where $|H|$ is the number of hits (maximum 9).
- **Hard (Advanced):** $O(|H|^2 + n^2)$ in worst case, where $n = 10$ is the grid dimension.

The space complexity for all algorithms is $O(n^2)$ for storing the game state matrices.

5.1.2 Monte Carlo Simulation Results

To validate our theoretical models, we conducted extensive Monte Carlo simulations with 1000 games per difficulty level. The .csv files (easy.csv, medium.csv & hard.csv) shows the distribution of shots required to win at each difficulty level.

The simulation results confirm our theoretical predictions:

- Easy mode: $\mu = 91.06\, shots\ (\sigma = 10.24)$
- Medium mode: $\mu = 61.66\, shots\ (\sigma = 11.80)$
- Hard mode: $\mu = 52.19\, shots\ (\sigma = 10.97)$

The efficiency factors measured empirically are:

- Medium vs. Easy: $\alpha_{empirical} = \frac{61.66}{91.06} \approx 0.677$
- Hard vs. Medium: $\beta_{empirical} = \frac{52.19}{61.66} \approx 0.846$

These values closely match our theoretical efficiency factors of $\alpha = 0.67$ and $\beta = 0.85$, confirming the validity of our mathematical models.

5.2 Statistical Distribution of Game Outcomes

The distribution of shots required reveals distinct patterns for each algorithm, as shown in Figure 1 - 6. For random ship placements across simulations, the Easy mode exhibits a right-skewed distribution with many games approaching the maximum shot limit, while Medium and Hard modes show more normally distributed outcomes.

When testing against predefined ship configurations (with identical ship placement across all trials), the performance patterns retain similar characteristics but with reduced variance, as detailed in Appendix A. This controlled environment allows for more direct algorithm comparison by eliminating the variability introduced by ship placement.

The Easy algorithm's random approach consistently demonstrates higher variance in game length across both testing methodologies. This variability decreases significantly in the Medium mode, where the hunt-and-target strategy produces more consistent results once initial hits are recorded, particularly evident in the controlled ship placement trials documented in Appendix A.

5.3 Analysis of Win Rates and Shot Efficiency

Our empirical results validate the theoretical performance models described in Section 3. The Easy mode's average of 91.06 shots aligns with our prediction of approximately 91.4 shots. Similarly, the medium mode's performance confirms our theoretical efficiency factor.

The Hard mode's advanced pattern recognition consistently outperforms simpler algorithms, completing 75% of games within 60 shots, compared to 70 shots for Medium and 96 shots for Easy. The performance gap between Medium and Hard is most pronounced in the 40-50 shot range, where the Hard algorithm's strategy demonstrates maximum efficiency.

6 Discussion and Conclusion

We learned that a simple game can be very difficult to create in MATLAB. Especially the GUI, which was very difficult to implement, and we had to seek help in the form of AI (1). However, optimizing the gameplay significantly improved the user experience, so we chose to spend some extra time on it. At the beginning of our brainstorming, we discussed that the user could decide how big the board should be, how many ships there should be, and their sizes. We quickly found out that it was much more difficult to implement. However, now that we have the basics of the game, it is easier to extend it.

In the future, we plan to add more features to the game, such as different types of ships and more. We also aim to improve the user interface to make it more intuitive and visually appealing. Additionally, we will consider implementing a multiplayer mode so that players can compete against each other.

This project taught us valuable lessons in problem solving, especially when dealing with complex programming tasks. We also learned the importance of seeking help and using available resources, such as the TA and AI to overcome challenges and keeping the game “simple”. Finally, we realized the significance of user feedback in improving the game's design and functionality, this we did by playing the game and taking notes, what was good and what we wanted to improve whit the time given.

7 Literature / references

1. **Chat GPT & Claude AI .**

2. Wikipedia. *Battleship (game)*. [Online] https://en.wikipedia.org/wiki/Battleship_%28game%29.

3. https://www.wikihow.com/Win-at-Battleship . [Online]

8 Work done by

There are many common points where everyone has contributed to the task solution, so the table should be seen more as the person who was responsible for getting that part of the task done.

Mads Madsen S233971	**Introduction, Different levels, Rules, Coding, Discussion and Conclusion, references, setup**
Frederik Tots S245710	**Code, GUI implementation. Mathematical and Algorithmic integration, setup**
Simon Christophersen s245805	**Coding, How fast will you lose, Strategy, Making plots (Statistics), setup**

9 Appendix

Appendix A: Statistics for Predefined Locations of Ships

When playing, there are three different levels that you can play against. To find out how quickly you lose to the different opponents, we created a randomized board, so the ships moved around the game board each time a new game was started. Then we had each level play alone to see how many turns/shots it would take at each level to win. This way, we get the most realistic results, of how it would look like when someone is playing against the computer. We have plotted these and added them in the appendix.

But to make it more visible, there is a significant difference when playing against the different difficulties, we also made some simulations with a predefined board, where ships would be placed the same positions for the whole simulation. By doing it like this it gets clearly, that it takes a lot less shots for the computer to win, while playing the "Difficult" mode compared to the "Easy" mode. It also shows how different games can be, even though they are played in the same difficulty mode. It gives a great insight into that strategy does matter, a lot! The average amount of shots made is almost halved, when comparing Easy and Difficult mode, which also can be seen in the last plot with colors, where the average amount of shots in all three game modes is compared to each other. However, a few times the Easy mode tempts us to win the game with far less attempts just as in the Difficult mode, which by that we also can conclude, that some time, all it takes is a bit of luck.

Underneath shows the plots for both situations, random and predefined game boards. We made a simulation of 100 games with the predefined board and 1000 with the random located board.

A.1 Easy

The plot below describes how many times games with the same amount of turn occur in the easy mode. It is obvious to see that due to most of the games, the computer had to use all its 100 turns. This occurred in 14 out of the 100 games. This is a good example of showing that the "Easy mode" throughout the whole game shoots randomly and does not take the last lost in consideration of where to make the next. However, in a few games, the computer wins by only using half of its shots. This is due to a lot of luck. We made the simulations a few times, and never got lower than 55 shots, even though it is possible.

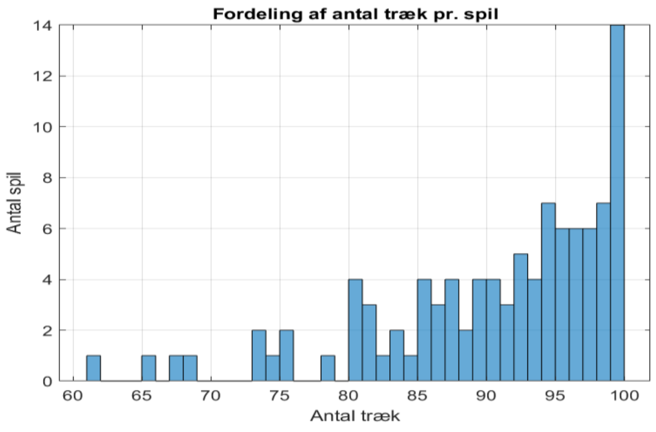


Figure 8 - Number of moves (Easy)

To these simulations we also made calculations and found that in this mode, the average amount of shots used was 91.06 per game out of a possible 100, which makes the name "Easy" quite fitting.

A.2 Medium

Now we changed to difficulty "Medium", and here we see some big changes of the amount of turns/shot per game, but also the distribution of the same is more width than the previous one. In the "Easy" mode most of the games were with a total of 95-100 turns, whereas the most played was 100 turns which happened 14 times. But now in the "Medium" mode, we see that we almost no longer have games that takes more than 90 turns, only a total of 3 games, which is a lot less than previously. The most occurred number of turns also changed to 61 which happened 7 times.

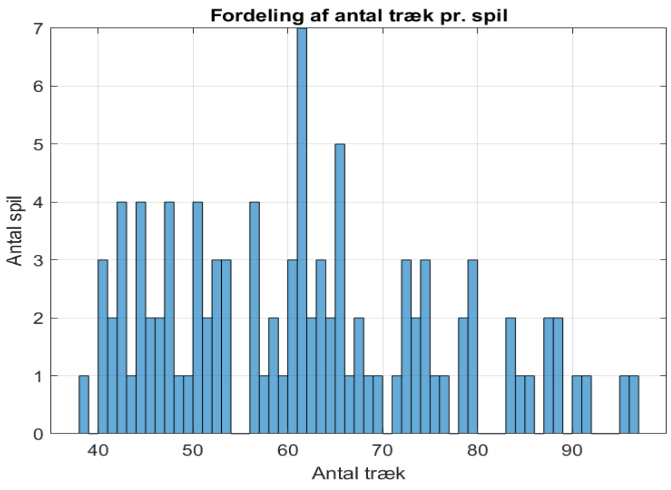


Figure 9 - Number of moves (Medium)

As earlier mentioned, this difficulty uses a strategy, where if it hits, it makes its next shots to the surrounding squares, searching for the remaining of the ship that got hit. So now it only depends on the time/number of turns for computers to find the location of the ships, but once they have been found, they go down fast!
In this game mode the average amount of shots made was 61.66 per game though a simulation of 100 games. The amount of turns per game which occurred the most was 61 and that happened 7 times in total.

A.3 Hard

For the last simulation we will notice a smaller gap between the average amount of turns per game. Between game mode "Easy" and "Medium" we had the biggest gap where we did go from an average of 91.06 to 61.66 per game, which means a total gap of 29.40 turn per game.

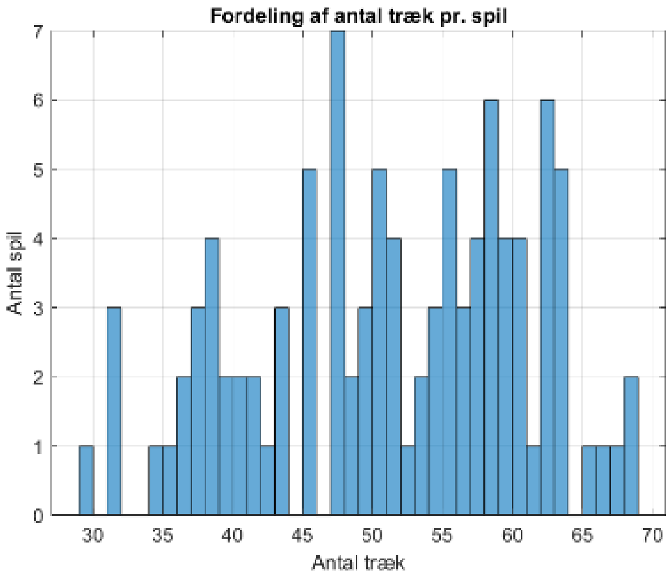


Figure 10 - Number of moves (Hard)

By analyzing the last plot for det "Difficult" level, we found the new average amount of turns to be 52.19, which is 9,47 less turns than the "Medium" game mode. Not just as significant as in the previous difficulty jump, but it still has a great impact, that now the computer only needs around half of possible attempts to sink all its opponent's ships. We also see that the least amount of turns now have gone down to only 29 and the most used amount was 68.

For the “Difficult” mode, we added a new strategy for computer, which we called the “chess strategy” which was explained earlier in this report. We see clearly in the simulation that by using this strategy, in which optimize the computers search for the ships, there occurs mores games played with less turns used. It went down from 39 turns in the “Medium” mode to 29 turns in the “Difficult” mode. A whole 10 turns less, is quite a lot when going from 39 to 29, and can do the difference from losing to winning.

A.4 Predefined Location Conclusion

Average number of turns. In this chart we have drawn the different average number of turns for each difficulty in order to compare them more visually. During the last points, we have compared and analyzed the different values which we have gotten from our simulations and explained what and why they look like they do, and with this last one to summarize it.

Strategy is an important part of playing Battleship and can determine whether you will lose or win the game. However, there will always be some luck included when searching for the opponent's ships, but it is important to think wisely, when you finally make a hit.

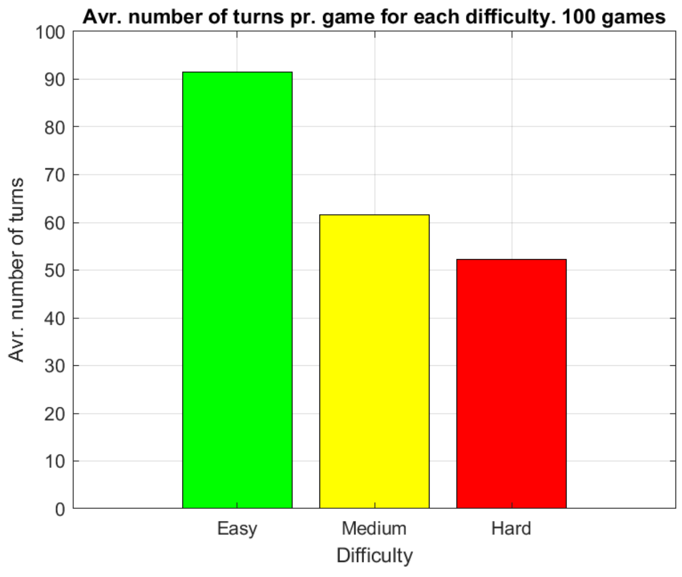


Figure 11 - Avr. amout of turn pr. game

Appendix E: Full Source Code

```matlab
% BATTLESHIPGUI - Graphical user interface for the Battleship game
% This is a GUI version of battleship.m that uses MATLAB's figure and
% uicontrol components to create an interactive game experience.

    % Initialize game data
    gameData = struct();
    gameData.playerGrid = zeros(10, 10);
    gameData.computerGrid = zeros(10, 10);
    gameData.playerShots = zeros(10, 10);
    gameData.computerShots = zeros(10, 10);
    gameData.gameState = 'setup'; % setup, placing, playing, gameover
    gameData.difficulty = 1;
    gameData.currentShip = 1;
    gameData.ships = struct('name', {'Battleship', 'Cruiser', 'Destroyer'}, ...
                           'length', {4, 3, 2}, 'placed', {false, false, false});
    gameData.playerHits = [0, 0, 0];  % Hits on each player ship
    gameData.computerHits = [0, 0, 0]; % Hits on each computer ship
    gameData.playerTurn = true;

    % Create main figure
    fig = figure('Name', 'Battleship', 'Position', [100, 100, 1000, 600], ...
                 'MenuBar', 'none', 'NumberTitle', 'off', 'Color', [0.9 0.9 0.95], ...
                 'CloseRequestFcn', @closeGame);

    % Save gameData in figure
    setappdata(fig, 'gameData', gameData);

    % Create control panel
    controlPanel = uipanel('Position', [0.02, 0.02, 0.2, 0.96], 'Title', 'Control Panel', ...
                          'BackgroundColor', [0.9 0.9 0.95]);

    % Create difficulty selector
    uicontrol('Parent', controlPanel, 'Style', 'text', 'Position', [20, 520, 120, 20], ...
              'String', 'Difficulty:', 'BackgroundColor', [0.9 0.9 0.95]);
    difficultySelector = uicontrol('Parent', controlPanel, 'Style', 'popupmenu', ...
                                   'Position', [20, 490, 120, 25], ...
                                   'String', {'Easy', 'Medium', 'Hard'}, ...
                                   'Callback', @setDifficulty);

    % Start game button
    startButton = uicontrol('Parent', controlPanel, 'Style', 'pushbutton', ...
                           'Position', [20, 440, 120, 40], ...
                           'String', 'Start Game', 'Callback', @startGame);

    % Orientation selector (for ship placement)
    uicontrol('Parent', controlPanel, 'Style', 'text', 'Position', [20, 390, 120, 20], ...
              'String', 'Orientation:', 'BackgroundColor', [0.9 0.9 0.95]);
    orientationSelector = uicontrol('Parent', controlPanel, 'Style', 'popupmenu', ...
                                    'Position', [20, 360, 120, 25], ...
                                    'String', {'Horizontal', 'Vertical'});

    % Instructions button
    instructionsButton = uicontrol('Parent', controlPanel, 'Style', 'pushbutton', ...
                                  'Position', [20, 310, 120, 30], ...
                                  'String', 'Instructions', 'Callback', @showInstructions);

    % Simulation button (NEW ELEMENT)
    simulationButton = uicontrol('Parent', controlPanel, 'Style', 'pushbutton', ...
                                'Position', [20, 260, 120, 30], ...
                                'String', 'Run Simulation', 'Callback', @runSimulation);

    % Status field
    statusText = uicontrol('Parent', controlPanel, 'Style', 'text', ...
                          'Position', [10, 50, 160, 200], ...
                          'String', 'Select difficulty and click Start Game', ...
                          'HorizontalAlignment', 'left', ...
                          'BackgroundColor', [0.9 0.9 0.95]);

    % Create game boards as axes
    playerBoard = axes('Position', [0.25, 0.1, 0.35, 0.8]);
    title('Your Board');

    enemyBoard = axes('Position', [0.65, 0.1, 0.35, 0.8]);
    title('Opponent''s Board');

    % Draw grid for both boards
    drawGrid(playerBoard);
    drawGrid(enemyBoard);

    % Save UI references for later use
    handles = struct();
    handles.fig = fig;
    handles.controlPanel = controlPanel;
    handles.difficultySelector = difficultySelector;
    handles.orientationSelector = orientationSelector;
    handles.startButton = startButton;
    handles.statusText = statusText;
    handles.playerBoard = playerBoard;
    handles.enemyBoard = enemyBoard;
    handles.simulationButton = simulationButton; % NEW ELEMENT
    setappdata(fig, 'handles', handles);

    % Disable enemy board until game is started
    set(enemyBoard, 'ButtonDownFcn', []);

    % Set playerBoard to handle ship placement
    % This is activated after the game starts
    set(playerBoard, 'ButtonDownFcn', []);

    % Show welcome screen
    showWelcomeScreen(fig);
end

```

10.1 Subfunction “closeGame”

```matlab
function closeGame(src, ~)
% CLOSEGAME - Handles closing of the game with confirmation dialog
% Inputs:
%   src - Source handle for callback

    % Confirm game exit
    choice = questdlg('Are you sure you want to exit the game?', ...
        'Exit Battleship', 'Yes', 'No', 'No');

    if strcmp(choice, 'Yes')
        delete(src);
    end
end

```

10.2 Subfunction “computerTurn”

```matlab
function computerTurn(fig)
% COMPUTERTURN - Handles the computer's turn
% Inputs:
%   fig - Handle to the main figure

    gameData = getappdata(fig, 'gameData');
    handles = getappdata(fig, 'handles');

    % Check if game is still active
    if ~strcmp(gameData.gameState, 'playing')
        return;
    end

    % Get computer shot based on difficulty
    [row, col] = getComputerShot(gameData.computerShots, gameData.playerGrid, gameData.difficulty);
    coordStr = sprintf('%c%d', 'A' + row - 1, col);

    % Update status to show computer's move
    set(handles.statusText, 'String', sprintf('Computer shoots at %s', coordStr));
    pause(0.5);

    % Process shot
    if gameData.playerGrid(row, col) > 0
        % Hit
        shipType = gameData.playerGrid(row, col);
        gameData.computerShots(row, col) = 2; % Mark as hit

        set(handles.statusText, 'String', sprintf('HIT! The computer hit your %s at %s!', ...
                                                gameData.ships(shipType).name, coordStr));

        % Track ship damage
        gameData.playerHits(shipType) = gameData.playerHits(shipType) + 1;

        % Check if ship sunk
        if gameData.playerHits(shipType) == gameData.ships(shipType).length
            set(handles.statusText, 'String', sprintf('Computer sank your %s!', gameData.ships(shipType).name));
        end
    else
        % Miss
        gameData.computerShots(row, col) = 1; % Mark as miss
        set(handles.statusText, 'String', sprintf('MISS! Computer''s shot at %s hit nothing.', coordStr));
    end

    % Update display
    updateGridDisplay(handles.playerBoard, gameData.playerGrid, gameData.computerShots, true);

    % Check for loss
    if sum(gameData.playerHits) == sum([gameData.ships.length])
        gameData.gameState = 'gameover';
        set(handles.statusText, 'String', 'DEFEAT! Computer sank all your ships!');
        set(handles.enemyBoard, 'ButtonDownFcn', []);
        setappdata(fig, 'gameData', gameData);

        % Display game over message
        showGameResult(fig, false);
        return;
    end

    % Switch turns back to player
    gameData.playerTurn = true;
    pause(0.5);

    % Prompt player for next move
    set(handles.statusText, 'String', sprintf('%s\nYour turn - click on the opponent''s board to shoot.', ...
                                              get(handles.statusText, 'String')));

    setappdata(fig, 'gameData', gameData);
end

```

10.3 Subfunktion ”drawGrid”

```matlab
function drawGrid(ax)
% DRAWGRID - Draw an empty 10x10 grid
% Inputs:
%   ax - Axes object to draw the grid on

    cla(ax);
    hold(ax, 'on');

    % Set axis properties
    axis(ax, [0 10 0 10]);
    axis(ax, 'square');

    % Draw grid lines
    for i = 0:10
        line(ax, [i i], [0 10], 'Color', 'k');
        line(ax, [0 10], [i i], 'Color', 'k');
    end

    % Add labels
    for i = 1:10
        text(ax, i-0.5, -0.3, num2str(i), 'HorizontalAlignment', 'center');
        text(ax, -0.3, i-0.5, char('A'+i-1), 'HorizontalAlignment', 'center');
    end

    % Remove standard axis ticks
    set(ax, 'XTick', [], 'YTick', []);

    hold(ax, 'off');
end

```

10.4 Subfunction “fireShot”

```
function fireShot(src, ~)
% FIRESHOT - Handles clicks on the enemy's board to shoot
% Inputs:
%   src - Source handle for callback

    fig = ancestor(src, 'figure');
    gameData = getappdata(fig, 'gameData');
    handles = getappdata(fig, 'handles');

    % Check if the game is active
    if ~strcmp(gameData.gameState, 'playing') || ~gameData.playerTurn
        return;
    end

    % Get coordinates from click
    coords = get(src, 'CurrentPoint');
    col = floor(coords(1,1)) + 1;
    row = floor(coords(1,2)) + 1;

    % Check if the click is within the board
    if col < 1 || col > 10 || row < 1 || row > 10
        return;
    end

    % Check if the field has already been shot
    if gameData.playerShots(row, col) > 0
        set(handles.statusText, 'String', 'You already shot here! Choose another field.');
        return;
    end

    % Display coordinates
    coordStr = sprintf('%c%d', 'A' + row - 1, col);

    % Register player's shot
    if gameData.computerGrid(row, col) > 0
        % Hit
        shipType = gameData.computerGrid(row, col);
        gameData.playerShots(row, col) = 2; % Mark as hit
        set(handles.statusText, 'String', sprintf('HIT! You hit a %s at %s!', gameData.ships(shipType).name, coordStr));

        % Track ship damage
        gameData.computerHits(shipType) = gameData.computerHits(shipType) + 1;

        % Check if ship sunk
        if gameData.computerHits(shipType) == gameData.ships(shipType).length
                     set(handles.statusText,  'String',  sprintf('You  sank  the  opponent''s  %s!',
gameData.ships(shipType).name));
        end
    else
        % Miss
        gameData.playerShots(row, col) = 1; % Mark as miss
        set(handles.statusText, 'String', sprintf('MISS! Your shot at %s hit nothing.', coordStr));
    end

    % Update display
    updateGridDisplay(handles.enemyBoard, zeros(10,10), gameData.playerShots, false);

    % Check for win
    if sum(gameData.computerHits) == sum([gameData.ships.length])
        gameData.gameState = 'gameover';
        set(handles.statusText, 'String', 'VICTORY! You sank all the opponent''s ships!');
        set(handles.enemyBoard, 'ButtonDownFcn', []);
        setappdata(fig, 'gameData', gameData);

        % Show victory message
        showGameResult(fig, true);
        return;
    end

    % Switch turn
    gameData.playerTurn = false;
    setappdata(fig, 'gameData', gameData);

    % Computer's turn - with a small delay so the player can see what happens
    pause(0.8);
    computerTurn(fig);
end

```

10.5 Subfunction “getComputerShot”

```
function [row, col] = getComputerShot(shotGrid, playerGrid, difficulty)
    % Get computer's shot based on difficulty level

    % Grid size
    [rows, cols] = size(shotGrid);

    % EASY MODE - random shots
    if difficulty == 1
        validShot = false;

        while ~validShot
            % Random position
            row = randi(rows);
            col = randi(cols);

            % Check if already shot at this position
            if shotGrid(row, col) == 0
                validShot = true;
            end
        end

        return;
    end

    % MEDIUM MODE - hunt and target
    if difficulty == 2
        % Look for hits to target adjacent cells
        for i = 1:rows
            for j = 1:cols
                if shotGrid(i, j) == 2  % Found a hit
                    % Try adjacent cells (up, down, left, right)
                    directions = [[-1, 0]; [1, 0]; [0, -1]; [0, 1]];

                    for d = 1:length(directions)
                        newRow = i + directions(d, 1);
                        newCol = j + directions(d, 2);

                        % Check if valid and not already tried
                            if newRow >= 1 && newRow <= rows && newCol >= 1 && newCol <= cols && shotGrid(newRow, newCol) == 0
                            row = newRow;
                            col = newCol;
                            return;
                        end
                    end
                end
            end
        end

        % If no hits found, take random shot
        validShot = false;
        while ~validShot
            row = randi(rows);
            col = randi(cols);

            if shotGrid(row, col) == 0
                validShot = true;
            end
        end

        return;
    end

    % HARD MODE - advanced targeting
    if difficulty == 3
        % First, look for two adjacent hits to extend the line
        for i = 1:rows
            for j = 1:cols-1
                if shotGrid(i, j) == 2 && shotGrid(i, j+1) == 2  % Horizontal hits
                    % Try left
                    if j > 1 && shotGrid(i, j-1) == 0
```

```
                        row = i;
                        col = j-1;
                        return;
                    end

                    % Try right
                    if j+2 <= cols && shotGrid(i, j+2) == 0
                        row = i;
                        col = j+2;
                        return;
                    end
                end
            end
        end

        for j = 1:cols
            for i = 1:rows-1
                if shotGrid(i, j) == 2 && shotGrid(i+1, j) == 2  % Vertical hits
                    % Try up
                    if i > 1 && shotGrid(i-1, j) == 0
                        row = i-1;
                        col = j;
                        return;
                    end

                    % Try down
                    if i+2 <= rows && shotGrid(i+2, j) == 0
                        row = i+2;
                        col = j;
                        return;
                    end
                end
            end
        end

        % If no adjacent hits, use medium difficulty strategy
        % Look for single hits
        for i = 1:rows
            for j = 1:cols
                if shotGrid(i, j) == 2  % Found a hit
                    % Try adjacent cells
                    directions = [[-1, 0]; [1, 0]; [0, -1]; [0, 1]];

                    for d = 1:length(directions)
                        newRow = i + directions(d, 1);
                        newCol = j + directions(d, 2);

                            if newRow >= 1 && newRow <= rows && newCol >= 1 && newCol <= cols && shotGrid(newRow, newCol) == 0
                            row = newRow;
                            col = newCol;
                            return;
                        end
                    end
                end
            end
        end

        % If no hits found, take random shot using checkerboard pattern
        validShot = false;
        attempts = 0;

        % Try checkerboard pattern first
        while ~validShot && attempts < 50
            attempts = attempts + 1;

            % Get random position adhering to checkerboard pattern
            r = randi(rows);
            c = randi(cols);

            % Only consider positions where r+c is even (checkerboard)
            if mod(r+c, 2) == 0 && shotGrid(r, c) == 0
                row = r;
                col = c;
                validShot = true;
            end
        end

        % If checkerboard failed, take any valid shot
        if ~validShot
            while ~validShot
                row = randi(rows);
                col = randi(cols);

                if shotGrid(row, col) == 0
                    validShot = true;
                end
            end
        end

        return;
    end
end

```

10.6 Subfunction ”placeComputerShips”

```
function grid = placeComputerShips(grid, ships)
% PLACECOMPUTERSHIPS - Automatically place computer ships on the board
% Inputs:
%   grid - 10x10 matrix to place ships on
%   ships - Struct array with ship information
% Outputs:
%   grid - Updated grid with computer ships placed

    % Ship lengths
    shipLengths = [ships.length];

    % Place each ship
    for i = 1:length(shipLengths)
        placedSuccessfully = false;

        while ~placedSuccessfully
            % Random position and orientation
            row = randi(10);
            col = randi(10);
            isHorizontal = randi(2) == 1;

            % Check if ship fits on grid
            if isHorizontal && col + shipLengths(i) - 1 > 10
                continue;
            elseif ~isHorizontal && row + shipLengths(i) - 1 > 10
                continue;
            end

            % Check if space is already occupied
            occupied = false;

            if isHorizontal
                for j = 0:shipLengths(i)-1
                    if grid(row, col+j) ~= 0
                        occupied = true;
                        break;
                    end
                end
            else
                for j = 0:shipLengths(i)-1
                    if grid(row+j, col) ~= 0
                        occupied = true;
                        break;
                    end
                end
            end

            if occupied
                continue;
            end

            % Place the ship
            if isHorizontal
                for j = 0:shipLengths(i)-1
                    grid(row, col+j) = i;
                end
            else
                for j = 0:shipLengths(i)-1
                    grid(row+j, col) = i;
                end
            end

            placedSuccessfully = true;
        end
    end
end

```

10.7 Subfunction "placeShip"

```
function placeShip(src, ~)
% PLACESHIP - Handles clicks on the player's board to place ships
% Inputs:
```

```matlab
%   src - Source handle for callback

    fig = ancestor(src, 'figure');
    gameData = getappdata(fig, 'gameData');
    handles = getappdata(fig, 'handles');

    % Check if we are in the placement phase
    if ~strcmp(gameData.gameState, 'placing')
        return;
    end

    % Get coordinates from click
    coords = get(src, 'CurrentPoint');
    col = floor(coords(1,1)) + 1;
    row = floor(coords(1,2)) + 1;

    % Check if the click is within the board
    if col < 1 || col > 10 || row < 1 || row > 10
        return;
    end

    % Get orientation (1=horizontal, 2=vertical)
    orientation = get(handles.orientationSelector, 'Value');

    % Check if the ship can be placed
    currentShip = gameData.currentShip;
    shipLength = gameData.ships(currentShip).length;

    % Validation of the placement
    valid = validateShipPlacement(gameData.playerGrid, row, col, orientation, shipLength);

    if valid
        % Place the ship
        if orientation == 1  % Horizontal
            for i = 0:(shipLength-1)
                gameData.playerGrid(row, col+i) = currentShip;
            end
        else  % Vertical
            for i = 0:(shipLength-1)
                gameData.playerGrid(row+i, col) = currentShip;
            end
        end

        % Update display
        updateGridDisplay(handles.playerBoard, gameData.playerGrid, gameData.computerShots, true);

        % Mark the ship as placed
        gameData.ships(currentShip).placed = true;

        % Go to next ship or start the game
        if currentShip < length(gameData.ships)
            gameData.currentShip = currentShip + 1;
            set(handles.statusText, 'String', sprintf('Place your %s (%d cells)\nSelect orientation and click on your board.', ...
                                                    gameData.ships(gameData.currentShip).name, ...
                                                    gameData.ships(gameData.currentShip).length));
        else
            % All ships placed - start the game
            gameData.gameState = 'playing';

            % Show "game starts" message
            gameStartPanel = uipanel('Parent', fig, 'Position', [0.35, 0.45, 0.3, 0.1], ...
                'BackgroundColor', [0.9 1 0.9]);

            uicontrol('Parent', gameStartPanel, 'Style', 'text', ...
                'Position', [10, 35, 280, 25], 'String', 'All ships placed! Game starts!', ...
                'FontSize', 12, 'FontWeight', 'bold', 'BackgroundColor', [0.9 1 0.9]);

            % Add a timer to remove the panel after 2 seconds
            t = timer('ExecutionMode', 'singleShot', 'StartDelay', 2, ...
                'TimerFcn', @(~,~) delete(gameStartPanel));
            start(t);

             set(handles.statusText, 'String', 'Game in progress! Click on the opponent''s board to
shoot.');
            set(handles.playerBoard, 'ButtonDownFcn', []);
            set(handles.enemyBoard, 'ButtonDownFcn', @fireShot);
        end
    else
        set(handles.statusText, 'String', 'Invalid placement! Try again.');
    end

    setappdata(fig, 'gameData', gameData);
end

```

10.8 Subfunktion " restartGame"

```
function restartGame(startButton, resultPanel)
% RESTARTGAME - Helper function to remove the result panel and start a new game
% Inputs:
% startButton - Handle to the start button
% resultPanel - Handle to the result panel

    delete(resultPanel);
    startGame(startButton, []);
end

```

10.9 Subfunktion " runSimulation"

```
function runSimulation(src, ~)
% RUNSIMULATION - Runs automatic simulations to test the computer's difficulty level
% Inputs:
%   src - Source handle for callback

    fig = ancestor(src, 'figure');
    gameData = getappdata(fig, 'gameData');
    handles = getappdata(fig, 'handles');

    % Get difficulty level from GUI
    difficulty = gameData.difficulty;
    difficultyNames = {'Easy', 'Medium', 'Hard'};

    % Ask for number of simulations and speed
    answer = inputdlg({'Number of simulations:', 'Show graphics? (1=yes, 0=no)', 'Simulation speed (1-10, where 10 is fastest)'}, ...
                     'Simulation Settings', 1, {'100', '0', '10'});
    if isempty(answer)
        return;
    end

    numSimulations = str2double(answer{1});
    showGraphics = str2double(answer{2}) == 1;
    simulationSpeed = str2double(answer{3});

    % Limit speed between 1 and 10
    simulationSpeed = max(1, min(10, simulationSpeed));

    % Create simulation panel to show progress
    simPanel = uipanel('Parent', fig, 'Position', [0.3, 0.4, 0.4, 0.2], ...
        'Title', sprintf('Running %d simulations (%s difficulty, speed: %d/10)', ...
                       numSimulations, difficultyNames{difficulty}, simulationSpeed), ...
        'FontSize', 12, 'BackgroundColor', [0.95 0.95 1]);

    statusText = uicontrol('Parent', simPanel, 'Style', 'text', ...
        'Position', [20, 30, 320, 30], 'String', 'Simulation in progress...', ...
        'FontSize', 12, 'BackgroundColor', [0.95 0.95 1]);

    progressBar = uicontrol('Parent', simPanel, 'Style', 'slider', ...
        'Position', [20, 10, 320, 20], 'Min', 0, 'Max', 1, 'Value', 0, ...
        'Enable', 'off');

    drawnow;

    % Prepare data structures to store results
    results = struct();
    results.difficulty = difficulty;
    results.difficultyName = difficultyNames{difficulty};
    results.totalMoves = zeros(numSimulations, 1);
    results.totalHits = zeros(numSimulations, 1);
    results.totalMisses = zeros(numSimulations, 1);
    results.hitRatio = zeros(numSimulations, 1);
    results.gameWon = zeros(numSimulations, 1);

    % Target for total number of hits needed (based on ship sizes)
    totalRequiredHits = sum([gameData.ships.length]);

    % Add option for fast batch processing at high speed
    batchSize = 10; % Number of simulations to run in one batch without graphical updates

    if simulationSpeed >= 8 && numSimulations > 10
        batchProcessing = true;
        % Only update status for each batch
        updateFrequency = batchSize;
    else
        batchProcessing = false;
        updateFrequency = 1;
    end

    % Run simulations
    sim = 1;
    while sim <= numSimulations
        % Determine number of simulations in this batch
        currentBatchSize = min(batchSize, numSimulations-sim+1);
        batchEnd = sim + currentBatchSize - 1;

        % Update status
        if mod(sim-1, updateFrequency) == 0 || sim == 1
            set(statusText, 'String', sprintf('Simulation %d-%d of %d...', sim, batchEnd, numSimulations));
            set(progressBar, 'Value', (sim-1)/numSimulations);
            drawnow;
        end

        % Run simulations in the batch
        for batchSim = sim:batchEnd
            % Reset game data for this simulation
            simData = gameData;
            simData.playerGrid = zeros(10, 10);
            simData.computerGrid = zeros(10, 10);
            simData.playerShots = zeros(10, 10);
            simData.computerShots = zeros(10, 10);
            simData.playerHits = [0, 0, 0];
            simData.computerHits = [0, 0, 0];

            % Place ships randomly
            simData.playerGrid = placeComputerShips(simData.playerGrid, simData.ships);
            simData.computerGrid = placeComputerShips(simData.computerGrid, simData.ships);

            % Store hits and misses for this simulation
            hits = 0;
            misses = 0;
            moves = 0;
            gameWon = false;

            % Simulate the game (computer shoots at the random player board)
            while true
                moves = moves + 1;

                % Let computer shoot
                [row, col] = getComputerShot(simData.computerShots, simData.playerGrid, difficulty);

                % Register result
                if simData.playerGrid(row, col) > 0
                    % Hit
                    shipType = simData.playerGrid(row, col);
                    simData.computerShots(row, col) = 2;
                    hits = hits + 1;

                    % Track ship damage
                    simData.playerHits(shipType) = simData.playerHits(shipType) + 1;
                else
                    % Miss
                    simData.computerShots(row, col) = 1;
                    misses = misses + 1;
                end

                % Update graphics if specified and not in batch mode
                if showGraphics && ~batchProcessing
                    % Only update with a frequency based on simulation speed
                    updateGraphicsInterval = 11 - simulationSpeed; % 1 to 10 speed gives 10 to 1 interval
                    if mod(moves, updateGraphicsInterval) == 0
                        updateGridDisplay(handles.playerBoard, simData.playerGrid, simData.computerShots, true);
                        drawnow;
                    end
                end

                % Check for victory
                if sum(simData.playerHits) >= totalRequiredHits
                    gameWon = true;
                    break;
                end

                % Safety check: Abort if too many moves (avoid infinite loops)
                if moves > 200
                    break;
                end
            end

            % Save results
            results.totalMoves(batchSim) = moves;
            results.totalHits(batchSim) = hits;
            results.totalMisses(batchSim) = misses;
            results.hitRatio(batchSim) = hits / moves;
            results.gameWon(batchSim) = gameWon;
        end

        % Update progress
        if batchProcessing
            set(progressBar, 'Value', batchEnd/numSimulations);
            drawnow;
        end

        % Go to next batch
        sim = batchEnd + 1;
    end

    % Restore normal view
    set(handles.playerBoard, 'ButtonDownFcn', @placeShip);
    drawGrid(handles.playerBoard);
    drawGrid(handles.enemyBoard);

    % Remove simulation panel
    delete(simPanel);

    % Save results and display summary
    displaySimulationResults(fig, results);
end

function displaySimulationResults(fig, results)
    % Create figures with results
    resultsFig = figure('Name', sprintf('Battleship Simulation Results - %s', results.difficultyName), ...
                      'Position', [200, 200, 800, 600]);

    % Calculate average values
    avgMoves = mean(results.totalMoves);
    avgHits = mean(results.totalHits);
    avgMisses = mean(results.totalMisses);
    avgHitRatio = mean(results.hitRatio);
    winRate = mean(results.gameWon) * 100;

    % Panel with summary
    summaryPanel = uipanel('Parent', resultsFig, 'Position', [0.05, 0.7, 0.9, 0.25], ...
           'Title', 'Summary', 'FontSize', 14, 'BackgroundColor', [0.95 0.95 1]);

    summaryText = sprintf(['Difficulty: %s\n' ...
                         'Average number of moves: %.2f\n' ...
                         'Average number of hits: %.2f\n' ...
                         'Average number of misses: %.2f\n' ...
                         'Average hit ratio: %.2f%%\n' ...
                         'Win rate: %.2f%%'], ...
                         results.difficultyName, avgMoves, avgHits, avgMisses, ...
                         avgHitRatio*100, winRate);

    uicontrol('Parent', summaryPanel, 'Style', 'text', ...
             'Position', [20, 20, 680, 120], 'String', summaryText, ...
             'FontSize', 14, 'HorizontalAlignment', 'left', 'FontWeight', 'bold', ...
             'BackgroundColor', [0.95 0.95 1]);

    % Show message to the user about saved results
    uicontrol('Parent', summaryPanel, 'Style', 'text', ...
             'Position', [20, 5, 680, 20], ...
             'String', ['Results are saved in the workspace as ' ...
                      '"battleshipSimResults" (struct) and "battleshipSimTable" (table).'], ...
             'FontSize', 12, 'FontWeight', 'normal', 'HorizontalAlignment', 'left', ...
             'BackgroundColor', [0.95 0.95 1]);

    % Generate plots
    % Plot 1: Histogram of number of moves
    subplot(2, 2, 3);
    histogram(results.totalMoves);
    title('Distribution of moves');
    xlabel('Number of moves');
    ylabel('Frequency');

    % Plot 2: Histogram of hit ratio
    subplot(2, 2, 4);
    histogram(results.hitRatio * 100);
    title('Distribution of hit ratio');
    xlabel('Hit ratio (%)');
    ylabel('Frequency');

    % Save data in workspace
    assignin('base', 'battleshipSimResults', results);

    % Create table with data
    resultsTable = table(results.totalMoves, results.totalHits, results.totalMisses, ...
                        results.hitRatio*100, results.gameWon, ...
                        'VariableNames', {'NumberOfMoves', 'NumberOfHits', 'NumberOfMisses', ...
                                         'HitRatio_Percent', 'GameWon'});

    % Save table in workspace
    assignin('base', 'battleshipSimTable', resultsTable);

    % Export data button
    uicontrol('Parent', resultsFig, 'Style', 'pushbutton', ...
              'Position', [350, 360, 120, 30], 'String', 'Export data', ...
              'Callback', @(src,~) exportSimulationData(resultsTable));
end

function exportSimulationData(resultsTable)
    % Export data to a CSV file
    [fileName, filePath] = uiputfile('*.csv', 'Save simulation data');

    if fileName ~= 0
        fullPath = fullfile(filePath, fileName);
        writetable(resultsTable, fullPath);
        msgbox(sprintf('Data exported to %s', fullPath), 'Export completed');
    end
end

```

10.10 Subfunktion ”setDifficulty”

```
function setDifficulty(src, ~)
% SETDIFFICULTY - Updates difficulty based on user selection
% Inputs:
% src - Source handle for callback

    % Update difficulty
    fig = ancestor(src, 'figure');
    gameData = getappdata(fig, 'gameData');
    gameData.difficulty = get(src, 'Value');
    setappdata(fig, 'gameData', gameData);
end

```

10.11 Subfunktion ”showGameResult”

```
function showGameResult(fig, isVictory)
% SHOWGAMERESULT - Shows final result when the game is finished
% Inputs:
%   fig - Handle to the main figure
%   isVictory - Boolean, true if the player won, false if the computer won

    handles = getappdata(fig, 'handles');

    % Create overlay panel with result
    resultPanel = uipanel('Parent', fig, 'Position', [0.3, 0.4, 0.4, 0.2], ...
        'BackgroundColor', [0.9 0.9 1], 'BorderType', 'line', ...
        'HighlightColor', 'blue', 'BorderWidth', 2);

    if isVictory
```

```
        resultText = 'VICTORY! You sank all of the opponent''s ships!';
        textColor = [0 0.5 0];
    else
        resultText = 'DEFEAT! The computer sank all your ships!';
        textColor = [0.8 0 0];
    end

    % Add text
    uicontrol('Parent', resultPanel, 'Style', 'text', ...
        'Position', [20, 30, 320, 50], 'String', resultText, ...
        'FontSize', 14, 'FontWeight', 'bold', 'ForegroundColor', textColor, ...
        'BackgroundColor', [0.9 0.9 1]);

    % Play again button
    uicontrol('Parent', resultPanel, 'Style', 'pushbutton', ...
        'Position', [120, 10, 120, 30], 'String', 'Play again', ...
        'Callback', @(~,~) restartGame(handles.startButton, resultPanel));
end

```

10.12 Subfunktion " showInstructions"

```
function showInstructions(src, ~)
% SHOWINSTRUCTIONS - Shows game instructions in a new window
% Inputs:
%   src - Source handle for callback

    fig = ancestor(src, 'figure');

    % Show instructions in a new figure
    instructFig = figure('Name', 'Battleship Instructions', 'Position', [200, 200, 500, 400], ...
                        'MenuBar', 'none', 'NumberTitle', 'off');

    uicontrol('Parent', instructFig, 'Style', 'text', 'Position', [20, 20, 460, 360], ...
             'String', {
                 'HOW TO PLAY BATTLESHIP:', '', ...
                 'OBJECTIVE:', ...
                 '  Sink all of the opponent''s ships before yours are sunk!', '', ...
                 'GRID COORDINATES:', ...
                 '  - Rows are labeled with letters (A, B, C, ...)', ...
                 '  - Columns are labeled with numbers (1, 2, 3, ...)', ...
                 '  - Select position by clicking on the board', '', ...
                 'GAME SYMBOLS:', ...
                 '  - Blue square: Your ship', ...
                 '  - Red square with X: Hit (ship hit)', ...
                 '  - O: Miss (you hit water)', '', ...
                 'GAMEPLAY:', ...
                 '  1. Place your ships on your grid', ...
                 '  2. Take turns with the computer to fire shots', ...
                 '  3. The first to sink all enemy ships wins!'
             }, ...
             'HorizontalAlignment', 'left');
end

```

10.13 Subfunktion ”showWelcomeScreen”

```
function showWelcomeScreen(fig)
% SHOWWELCOMESCREEN - Shows welcome screen when the game starts
% Inputs:
%   fig - Handle to the main figure

    % Overlay panel for welcome
    welcomePanel = uipanel('Parent', fig, 'Position', [0.25, 0.25, 0.5, 0.5], ...
        'Title', 'Welcome to Battleship!', 'FontSize', 14, ...
        'BackgroundColor', [0.95 0.95 1]);

    % Add game description
    uicontrol('Parent', welcomePanel, 'Style', 'text', ...
        'Position', [20, 100, 460, 180], 'String', {...
        'HOW TO PLAY BATTLESHIP:', '', ...
        '1. Choose difficulty and click on "Start Game"', ...
        '2. Place your 3 ships by selecting orientation and clicking on your board', ...
        '3. Shoot at the computer''s ships by clicking on the opponent''s board', ...
        '4. The first to sink all of the opponent''s ships wins!', ...
        '', ...
        'Blue squares show your ships', ...
        'X marks a hit ship', ...
        'O marks a miss (water)'}, ...
        'FontSize', 11, 'HorizontalAlignment', 'left', ...
        'BackgroundColor', [0.95 0.95 1]);

    % Start game button
    uicontrol('Parent', welcomePanel, 'Style', 'pushbutton', ...
        'Position', [180, 30, 120, 40], 'String', 'I''m ready!', ...
        'FontSize', 12, 'Callback', @(src,~) delete(welcomePanel));
end

```

10.14 Subfunktion ”simulateGames”

```
function results = simulateGames(numSimulations, difficulty, simulationSpeed)
% SIMULATEGAMES - Runs simulations of Battleship games without GUI
% Inputs:
%   numSimulations - Number of games to simulate
%   difficulty - Difficulty level (1=Easy, 2=Medium, 3=Hard)
%   simulationSpeed - Speed of the simulation (1-10, where 10 is fastest)
% Outputs:
%   results - Struct with simulation results

    % Handle optional parameter
    if nargin < 3
        simulationSpeed = 10; % Default: Maximum speed
    end

    % Define ships (same as in battleshipGUI.m)
    ships = struct('name', {'Battleship', 'Cruiser', 'Destroyer'}, ...
                  'length', {4, 3, 2}, 'placed', {true, true, true});

    % Target for the total number of hits needed
```

```matlab
    totalRequiredHits = sum([ships.length]);

    % Prepare data structures to store results
    results = struct();
    results.difficulty = difficulty;
    results.difficultyNames = {'Easy', 'Medium', 'Hard'};
    results.difficultyName = results.difficultyNames{difficulty};
    results.totalMoves = zeros(numSimulations, 1);
    results.totalHits = zeros(numSimulations, 1);
    results.totalMisses = zeros(numSimulations, 1);
    results.hitRatio = zeros(numSimulations, 1);
    results.gameWon = zeros(numSimulations, 1);
    results.numShots = cell(numSimulations, 1);  % To store number of shots for each ship

    % Show console progress bar
    fprintf('Simulating %d games with %s difficulty (speed: %d/10):\n', ...
       numSimulations, results.difficultyName, simulationSpeed);
    progress = 0;
    fprintf('[%s]', repmat(' ', 1, 50));
fprintf('\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b');

    % Set up batch size based on speed
    batchSize = simulationSpeed^2; % Higher speed gives larger batches

    % Run simulations in batches to increase speed
    sim = 1;
    while sim <= numSimulations
        % Determine batch size
        currentBatchSize = min(batchSize, numSimulations - sim + 1);
        batchEnd = sim + currentBatchSize - 1;

        % Update progress bar
        if floor(sim/numSimulations*50) > progress
            progress = floor(sim/numSimulations*50);
fprintf('\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b');
            fprintf('[%s%s]', repmat('=', 1, progress), repmat(' ', 1, 50-progress));
        end

        % Reset game data for this simulation
        playerGrid = zeros(10, 10);
        computerShots = zeros(10, 10);
        playerHits = [0, 0, 0];

        % Place ships randomly for the "player" (which the computer plays against)
        playerGrid = placeComputerShips(playerGrid, ships);

        % Store hits and misses for this simulation
        hits = 0;
        misses = 0;
        moves = 0;
        gameWon = false;

        % Keep track of number of shots for each ship
        shotsByShipType = zeros(1, length(ships));

        % Simulate the game (computer shoots at the random player board)
        while true
            moves = moves + 1;

            % Let the computer shoot
            [row, col] = getComputerShot(computerShots, playerGrid, difficulty);

            % Register result
            if playerGrid(row, col) > 0
                % Hit
                shipType = playerGrid(row, col);
                computerShots(row, col) = 2;
                hits = hits + 1;

                % Track ship damage
                playerHits(shipType) = playerHits(shipType) + 1;
                shotsByShipType(shipType) = shotsByShipType(shipType) + 1;
            else
                % Miss
                computerShots(row, col) = 1;
                misses = misses + 1;
            end

            % Check for victory
            if sum(playerHits) >= totalRequiredHits
                gameWon = true;
                break;
            end

            % Safety check: Abort if too many moves (avoid infinite loops)
            if moves > 200
                break;
            end
        end

        % Save results
        results.totalMoves(sim) = moves;
        results.totalHits(sim) = hits;
        results.totalMisses(sim) = misses;
        results.hitRatio(sim) = hits / moves;
        results.gameWon(sim) = gameWon;
        results.numShots{sim} = shotsByShipType;
    end

    fprintf('\nSimulation completed!\n');

    % Calculate aggregate statistics
    results.avgMoves = mean(results.totalMoves);
    results.avgHits = mean(results.totalHits);
    results.avgMisses = mean(results.totalMisses);
    results.avgHitRatio = mean(results.hitRatio);
    results.winRate = mean(results.gameWon) * 100;

    % Calculate average number of shots per ship type
    shipTypeShots = zeros(length(ships), 1);
    for i = 1:numSimulations
        for j = 1:length(ships)
```

```
            shipTypeShots(j) = shipTypeShots(j) + results.numShots{i}(j);
        end
    end
    results.avgShotsByShipType = shipTypeShots / numSimulations;

    % Show summary
    fprintf('\nSummary for %s difficulty:\n', results.difficultyName);
    fprintf('Average number of moves: %.2f\n', results.avgMoves);
    fprintf('Average number of hits: %.2f\n', results.avgHits);
    fprintf('Average number of misses: %.2f\n', results.avgMisses);
    fprintf('Average hit ratio: %.2f%%\n', results.avgHitRatio*100);
    fprintf('Win rate: %.2f%%\n', results.winRate);

    % Convert to table for easy access
    results.table = table(results.totalMoves, results.totalHits, results.totalMisses, ...
                         results.hitRatio*100, results.gameWon, ...
                         'VariableNames', {'NumberOfMoves', 'NumberOfHits', 'NumberOfMisses', ...
                                          'HitRatio_Percent', 'GameWon'});
end

```

10.15 Subfunction “simulationManager”

```
function simulationManager()
% SIMULATIONMANAGER - GUI for managing Battleship simulations

    % Create figure
    fig = figure('Name', 'Battleship Simulation Manager', 'Position', [300, 300, 500, 400], ...
                'MenuBar', 'none', 'NumberTitle', 'off');

    % Main panel
    mainPanel = uipanel('Parent', fig, 'Position', [0.05, 0.05, 0.9, 0.9], ...
                       'Title', 'Simulation Configuration', 'FontSize', 12);

    % Simulation options
    uicontrol('Parent', mainPanel, 'Style', 'text', 'Position', [20, 300, 200, 20], ...
             'String', 'Number of simulations per difficulty:', 'HorizontalAlignment', 'left');

    numSimsEdit = uicontrol('Parent', mainPanel, 'Style', 'edit', 'Position', [230, 300, 100, 25],
...
                           'String', '100');

    % Simulation speed
    uicontrol('Parent', mainPanel, 'Style', 'text', 'Position', [20, 270, 200, 20], ...
             'String', 'Simulation speed (1-10):', 'HorizontalAlignment', 'left');

     speedSlider = uicontrol('Parent', mainPanel, 'Style', 'slider', 'Position', [230, 270, 100,
25], ...
                            'Min', 1, 'Max', 10, 'Value', 10, 'SliderStep', [0.1 0.1]);

    speedText = uicontrol('Parent', mainPanel, 'Style', 'text', 'Position', [340, 270, 30, 20], ...
                         'String', '10');

    % Update speed text when slider changes
    addlistener(speedSlider, 'Value', 'PostSet', @(src,evt) set(speedText, 'String', ...
               num2str(round(get(speedSlider, 'Value')))));

    % Difficulty level checkboxes
    diffPanel = uipanel('Parent', mainPanel, 'Position', [0.05, 0.6, 0.9, 0.25], ...
                      'Title', 'Difficulty levels to simulate:', 'FontSize', 10);

    easyCheck = uicontrol('Parent', diffPanel, 'Style', 'checkbox', 'Position', [20, 60, 100, 20], ...
                        'String', 'Easy', 'Value', 1);

    mediumCheck = uicontrol('Parent', diffPanel, 'Style', 'checkbox', 'Position', [20, 35, 100, 20], ...
                          'String', 'Medium', 'Value', 1);

    hardCheck = uicontrol('Parent', diffPanel, 'Style', 'checkbox', 'Position', [20, 10, 100, 20], ...
                        'String', 'Hard', 'Value', 1);

    % Output options
    outputPanel = uipanel('Parent', mainPanel, 'Position', [0.05, 0.3, 0.9, 0.25], ...
                        'Title', 'Output options:', 'FontSize', 10);

    plotCheck = uicontrol('Parent', outputPanel, 'Style', 'checkbox', 'Position', [20, 60, 220, 20], ...
                        'String', 'Show plots during simulation', 'Value', 0);

    summaryCheck = uicontrol('Parent', outputPanel, 'Style', 'checkbox', 'Position', [20, 35, 220, 20], ...
                           'String', 'Show summary plots', 'Value', 1);

    saveCheck = uicontrol('Parent', outputPanel, 'Style', 'checkbox', 'Position', [20, 10, 220, 20], ...
                        'String', 'Save results to workspace', 'Value', 1);

    % Buttons
    runButton = uicontrol('Parent', mainPanel, 'Style', 'pushbutton', 'Position', [150, 40, 200, 40], ...
                        'String', 'Run simulations', 'FontSize', 12, ...
                        'Callback', @runSimulations);

    % Add help button
    uicontrol('Parent', mainPanel, 'Style', 'pushbutton', 'Position', [380, 10, 60, 30], ...
             'String', 'Help', 'Callback', @showHelp);

    function runSimulations(~, ~)
        % Get simulation configuration
        numSims = str2double(get(numSimsEdit, 'String'));
        runEasy = get(easyCheck, 'Value');
        runMedium = get(mediumCheck, 'Value');
        runHard = get(hardCheck, 'Value');
        showPlots = get(plotCheck, 'Value');
        showSummary = get(summaryCheck, 'Value');
        saveToWorkspace = get(saveCheck, 'Value');
        simulationSpeed = round(get(speedSlider, 'Value'));

        % Validate number of simulations
        if isnan(numSims) || numSims <= 0 || numSims > 10000
          errordlg('Number of simulations must be a positive number (max 10000)', 'Invalid input');
```

```
            return;
        end

        % Show wait screen
        waitPanel = uipanel('Parent', fig, 'Position', [0.2, 0.4, 0.6, 0.2], ...
                          'Title', 'Simulation in progress', 'FontSize', 12);

        statusText = uicontrol('Parent', waitPanel, 'Style', 'text', ...
                             'Position', [20, 30, 260, 20], 'String', 'Preparing simulation...');

        % Prepare results
        allResults = {};
        difficultyLabels = {};

        % Run simulations
        try
            if runEasy
                 set(statusText, 'String', sprintf('Simulating EASY difficulty (speed: %d/10)...', simulationSpeed));
                drawnow;
                easyResults = simulateGames(numSims, 1, simulationSpeed);
                allResults{end+1} = easyResults;
                difficultyLabels{end+1} = 'Easy';

                if saveToWorkspace
                    assignin('base', 'battleshipEasyResults', easyResults);
                end
            end

            if runMedium
               set(statusText, 'String', sprintf('Simulating MEDIUM difficulty (speed: %d/10)...', simulationSpeed));
                drawnow;
                mediumResults = simulateGames(numSims, 2, simulationSpeed);
                allResults{end+1} = mediumResults;
                difficultyLabels{end+1} = 'Medium';

                if saveToWorkspace
                    assignin('base', 'battleshipMediumResults', mediumResults);
                end
            end

            if runHard
                 set(statusText, 'String', sprintf('Simulating HARD difficulty (speed: %d/10)...', simulationSpeed));
                drawnow;
                hardResults = simulateGames(numSims, 3, simulationSpeed);
                allResults{end+1} = hardResults;
                difficultyLabels{end+1} = 'Hard';

                if saveToWorkspace
                    assignin('base', 'battleshipHardResults', hardResults);
                end
            end

            % Remove wait panel
            delete(waitPanel);

            % Show summary plots if selected
            if showSummary && length(allResults) > 0
                createSummaryPlots(allResults, difficultyLabels);
            end

            % Create comparison table
            if length(allResults) > 0
                comparisonTable = createComparisonTable(allResults, difficultyLabels);

                if saveToWorkspace
                    assignin('base', 'battleshipComparisonTable', comparisonTable);
                end

                % Show result message
                msgbox('Simulations completed!', 'Done');
            else
                msgbox('No simulations selected!', 'Warning');
            end

        catch e
            % Handle errors
            delete(waitPanel);
            errordlg(['An error occurred during simulation: ' e.message], 'Error');
        end
    end

    function createSummaryPlots(results, labels)
        % Compare results with plots
        figure('Name', 'Comparison of difficulty levels', 'Position', [100, 100, 1200, 800]);

        % Compile data
        allMoves = [];
        allHitRatios = [];
        groupLabels = {};

        for i = 1:length(results)
            allMoves = [allMoves; results{i}.totalMoves];
            allHitRatios = [allHitRatios; results{i}.hitRatio];
            groupLabels = [groupLabels; repmat(labels(i), length(results{i}.totalMoves), 1)];
        end

        % Plot 1: Number of moves
        subplot(2, 2, 1);
        boxplot(allMoves, groupLabels);
        title('Comparison of number of moves');
        ylabel('Number of moves');

        % Plot 2: Hit ratio
        subplot(2, 2, 2);
        boxplot(allHitRatios, groupLabels);
        title('Comparison of hit ratio');
        ylabel('Hit ratio');

        % Plot 3: Average values
        subplot(2, 2, 3);
        avgMoves = cellfun(@(x) x.avgMoves, results);
        avgHitRatios = cellfun(@(x) x.avgHitRatio*100, results);

```

```matlab
        avgData = [avgMoves; avgHitRatios];
        bar(avgData');
        title('Average values');
        set(gca, 'XTickLabel', labels);
        legend({'Number of moves', 'Hit ratio (%)'}, 'Location', 'northwest');

        % Plot 4: Win rates
        subplot(2, 2, 4);
        winRates = cellfun(@(x) x.winRate, results);
        bar(winRates);
        title('Win percentage');
        set(gca, 'XTickLabel', labels);
        ylabel('Percentage (%)');
        ylim([0, 100]);
    end

    function table = createComparisonTable(results, labels)
        % Create comparison table
        numDifficulties = length(results);

        avgMoves = zeros(numDifficulties, 1);
        avgHits = zeros(numDifficulties, 1);
        avgMisses = zeros(numDifficulties, 1);
        avgHitRatio = zeros(numDifficulties, 1);
        winRates = zeros(numDifficulties, 1);

        for i = 1:numDifficulties
            avgMoves(i) = results{i}.avgMoves;
            avgHits(i) = results{i}.avgHits;
            avgMisses(i) = results{i}.avgMisses;
            avgHitRatio(i) = results{i}.avgHitRatio*100;
            winRates(i) = results{i}.winRate;
        end

        % Convert to table
        table = array2table([avgMoves, avgHits, avgMisses, avgHitRatio, winRates], ...
                           'VariableNames', {'Avg_Moves', 'Avg_Hits', 'Avg_Misses', ...
                                          'Avg_HitRatio', 'Win_Percentage'});

        % Add difficulty level as first column
       table = addvars(table, labels', 'Before', 'Avg_Moves', 'NewVariableNames', {'Difficulty'});

        % Show table
        figure('Name', 'Comparison Table', 'Position', [400, 400, 700, 200]);
        uitable('Data', table{:,:}, 'ColumnName', table.Properties.VariableNames, ...
               'RowName', {}, 'Units', 'Normalized', 'Position', [0, 0, 1, 1]);
    end

    function showHelp(~, ~)
        helpFig = figure('Name', 'Simulation Help', 'Position', [400, 300, 500, 400], ...
                       'MenuBar', 'none', 'NumberTitle', 'off');

        helpText = {...
            'BATTLESHIP SIMULATION MANAGER', '', ...
            'This tool allows you to simulate Battleship games with different difficulty levels.', '', ...
            'HOW TO USE THE TOOL:', '', ...
            '1. Specify number of simulations for each difficulty level (e.g. 100)', ...
            '2. Choose which difficulty levels to test', ...
            '3. Select output options:', ...
            '   - "Show plots during simulation" shows game boards during simulation (slow)', ...
            '   - "Show summary plots" shows statistical plots after simulation', ...
            '   - "Save results to workspace" saves data for further analysis', ...
            '4. Click on "Run simulations" to start', '', ...
            'RESULTS:', '', ...
            'The results include:', ...
            '- Average number of moves to win', ...
            '- Hit/miss ratio', ...
            '- Win percentage', ...
            '- Comparison between different difficulty levels'};

        uicontrol('Parent', helpFig, 'Style', 'text', 'Position', [20, 20, 460, 350], ...
                 'String', helpText, 'HorizontalAlignment', 'left');
    end
end

```

10.16 Subfunction “startGame”

```
function startGame(src, ~)
% STARTGAME - Starts a new game and resets all game data
% Inputs:
%   src - Source handle for callback

    % Start a new game
    fig = ancestor(src, 'figure');
    gameData = getappdata(fig, 'gameData');
    handles = getappdata(fig, 'handles');

    % Reset game data structure
    gameData.playerGrid = zeros(10, 10);
    gameData.computerGrid = zeros(10, 10);
    gameData.playerShots = zeros(10, 10);
    gameData.computerShots = zeros(10, 10);
    gameData.gameState = 'placing';
    gameData.currentShip = 1;
    gameData.playerTurn = true;
    gameData.playerHits = [0, 0, 0];
    gameData.computerHits = [0, 0, 0];
    for i = 1:length(gameData.ships)
        gameData.ships(i).placed = false;
    end

    % Update status
   set(handles.statusText, 'String', sprintf('Place your %s (%d cells)\nSelect orientation and click on your board.', ...
                                           gameData.ships(1).name, gameData.ships(1).length));

    % Update boards
    drawGrid(handles.playerBoard);
    drawGrid(handles.enemyBoard);

    % Activate placement function
    set(handles.playerBoard, 'ButtonDownFcn', @placeShip);

    % Place computer's ships
    gameData.computerGrid = placeComputerShips(gameData.computerGrid, gameData.ships);

    % Save updated game data
    setappdata(fig, 'gameData', gameData);
end

```

10.17 Subfunction "updateGridDisplay"

```
function updateGridDisplay(ax, shipGrid, shotGrid, showShips)
% UPDATEGRID - Update grid display based on game state
% Inputs:
%   ax - Axes object to update
%   shipGrid - 10x10 matrix with ship placements
%   shotGrid - 10x10 matrix with shot information
%   showShips - Boolean, whether to display ships or not

    cla(ax);
    hold(ax, 'on');

    % Draw basic grid
    axis(ax, [0 10 0 10]);
    axis(ax, 'square');

    % Draw grid lines
    for i = 0:10
        line(ax, [i i], [0 10], 'Color', 'k');
        line(ax, [0 10], [i i], 'Color', 'k');
    end

    % Add labels
    for i = 1:10
        text(ax, i-0.5, -0.3, num2str(i), 'HorizontalAlignment', 'center');
        text(ax, -0.3, i-0.5, char('A'+i-1), 'HorizontalAlignment', 'center');
    end

    % Remove standard axis ticks
    set(ax, 'XTick', [], 'YTick', []);

    % Visualize ships, hits and misses
    for i = 1:10
        for j = 1:10
            if shotGrid(i, j) == 2
                % Hit
                rectangle(ax, 'Position', [j-1, i-1, 1, 1], 'FaceColor', 'red');
                plot(ax, j-0.5, i-0.5, 'kx', 'LineWidth', 2, 'MarkerSize', 15);
            elseif shotGrid(i, j) == 1
                % Miss
                plot(ax, j-0.5, i-0.5, 'ko', 'MarkerSize', 10, 'LineWidth', 1.5);
            elseif showShips && shipGrid(i, j) > 0
                % Ship
                rectangle(ax, 'Position', [j-1, i-1, 1, 1], 'FaceColor', [0.3 0.5 0.8]);
            end
        end
    end

    hold(ax, 'off');
end

```

10.18 Subfunction "validateShipPlacement"

```
function valid = validateShipPlacement(grid, row, col, orientation, shipLength)
% VALIDATESHIPPLACEMENT - Validate if a ship can be placed at the specified location
% Inputs:
%   grid - 10x10 matrix with current ship placements
%   row - Row index (1-10)
%   col - Column index (1-10)
%   orientation - 1 for horizontal, 2 for vertical
%   shipLength - Length of the ship to place
% Outputs:
%   valid - Boolean, true if placement is valid

    % Check if the ship fits on the board
    if orientation == 1 && col + shipLength - 1 > 10  % Horizontal
        valid = false;
        return;
    elseif orientation == 2 && row + shipLength - 1 > 10  % Vertical
        valid = false;
        return;
    end

    % Check if the fields are available
    valid = true;
    if orientation == 1  % Horizontal
        for i = 0:(shipLength-1)
            if grid(row, col+i) ~= 0
                valid = false;
                return;
            end
        end
    else  % Vertical
        for i = 0:(shipLength-1)
            if grid(row+i, col) ~= 0
                valid = false;
                return;
            end
        end
    end
end

```