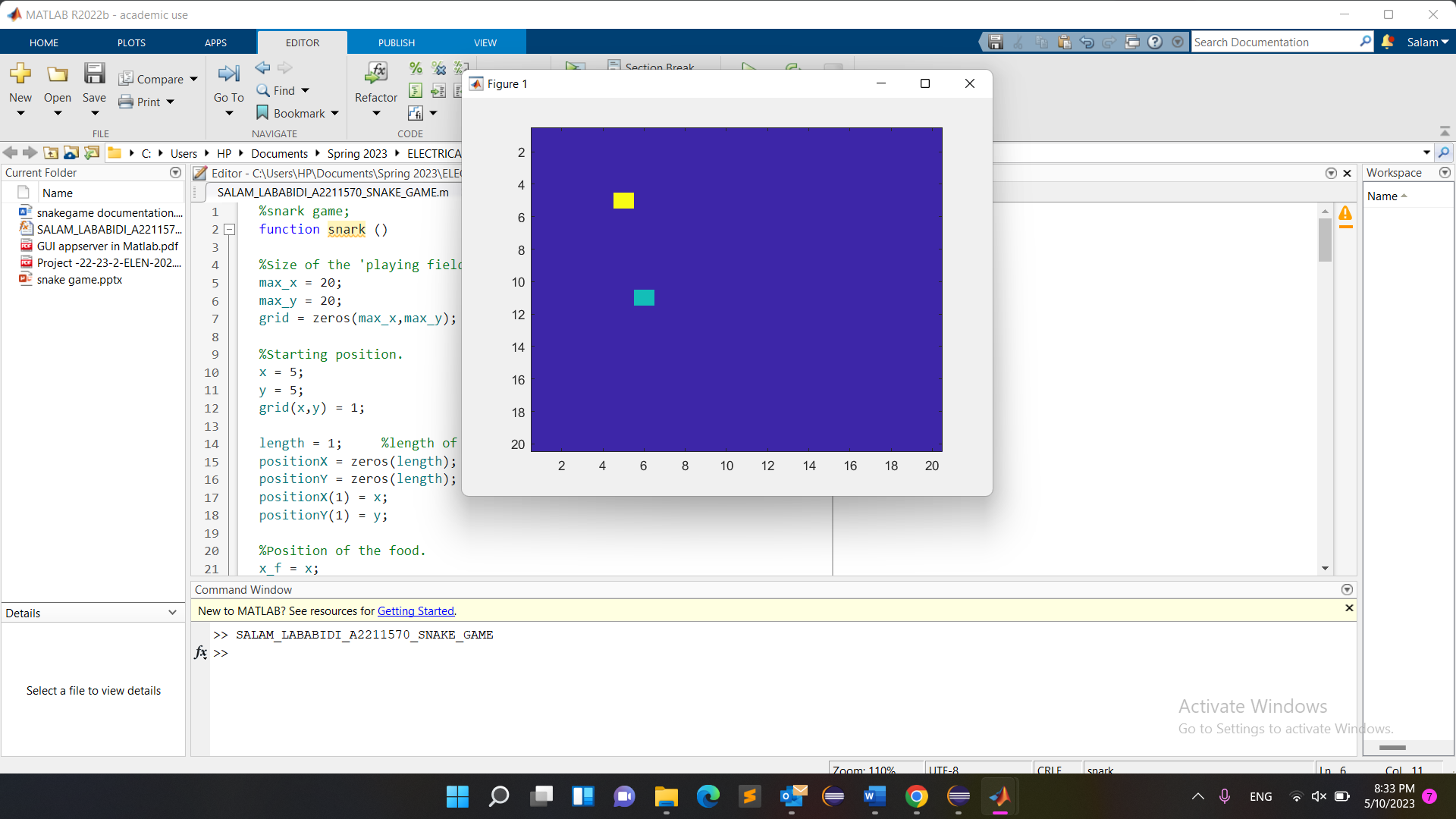
**MATLAB Snark Snake Game Code Report**

**1. Introduction:**

The MATLAB Snake Game is a classic game where the player controls a snake, navigating it through a grid to eat food and grow longer. This code report provides an overview of the implementation, functionality, and features of the Snake Game.



2. **Problem Statement:**

The objective of the Snake Game is to control the snake's movement using arrow keys while avoiding collisions with walls or the snake's own body. The goal is to eat the food, which causes the snake to grow longer. The game ends when the snake collides with a wall or itself.

3. **Methodology and Implementation:**

The Snake Game is implemented using MATLAB. The main components of the code include:

- Initializing the playing field, snake, and food positions.

- Handling user input to control the snake's movement.

- Updating the snake's position, checking for collisions, and growing the snake when it eats food.

- Generating new food elements at random positions.

- Displaying the game window and graphics using MATLAB's figure and imagesc functions.

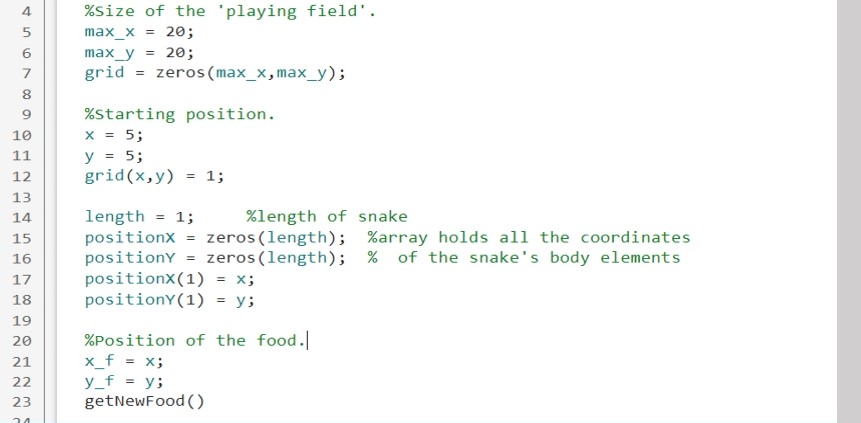
**4. Code Explanation:**

- The code begins by setting up the game's playing field and initializing the snake's position. Using the max\_x and max\_y variables. The grid is represented by a 2D matrix of zeros as we can see in line 7 of the code ( grid = zeros(max\_x, max\_y);

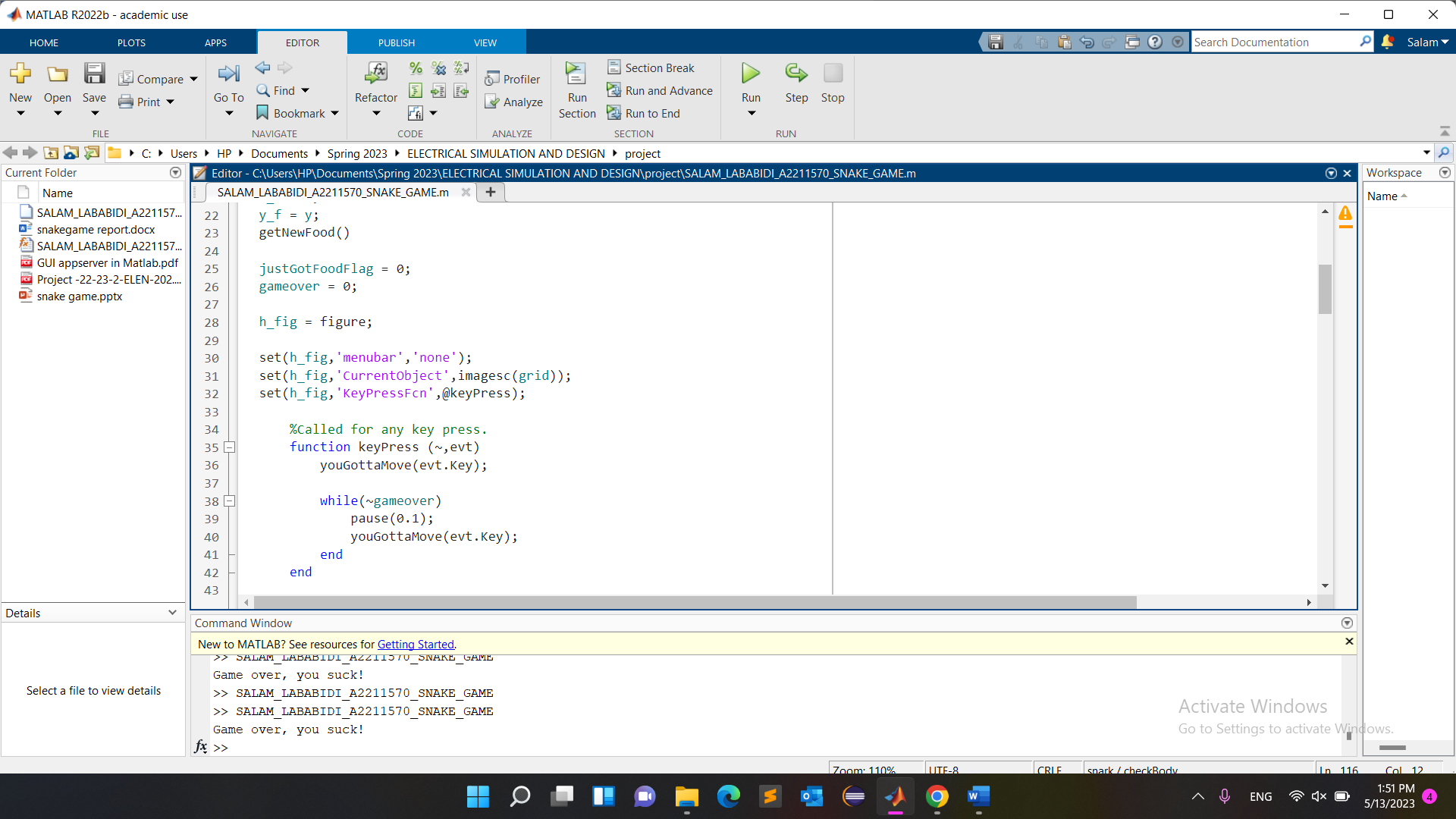
The starting position of the snake head is set to (5,5), and the grid is updated to mark the snake’s position.

The snake’s length isinitialized to 1, and the 2 arrays positionX and positionY are created to hold the coordinates of the snake’s body segments.

The position of the food is set to the same position as the snake’s head initially.

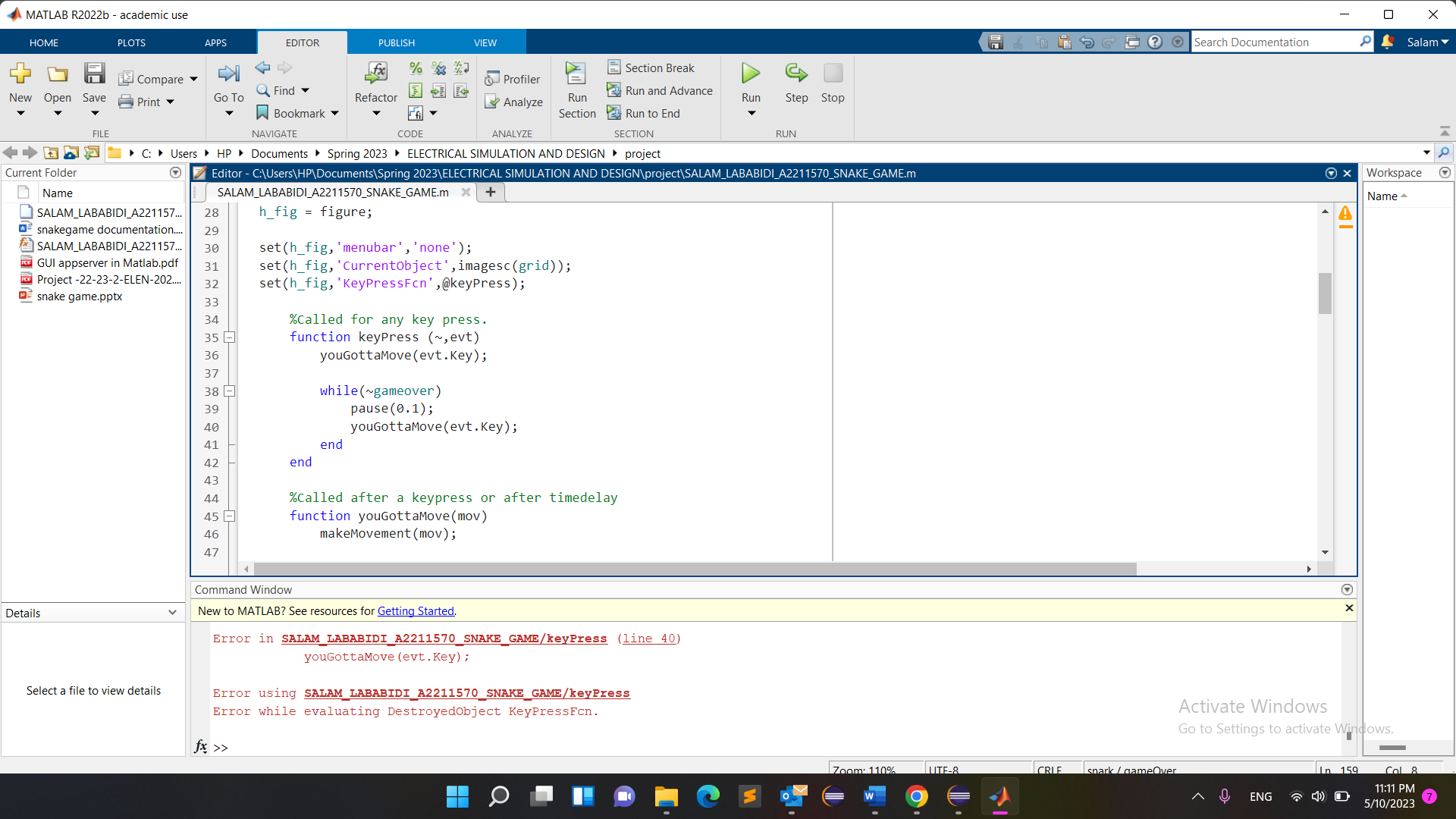


* Game Loop and User Input:



The code creates a figure window using h\_fig and sets up the necessary properties for the game display, such as removing the menu bar and setting the current object.

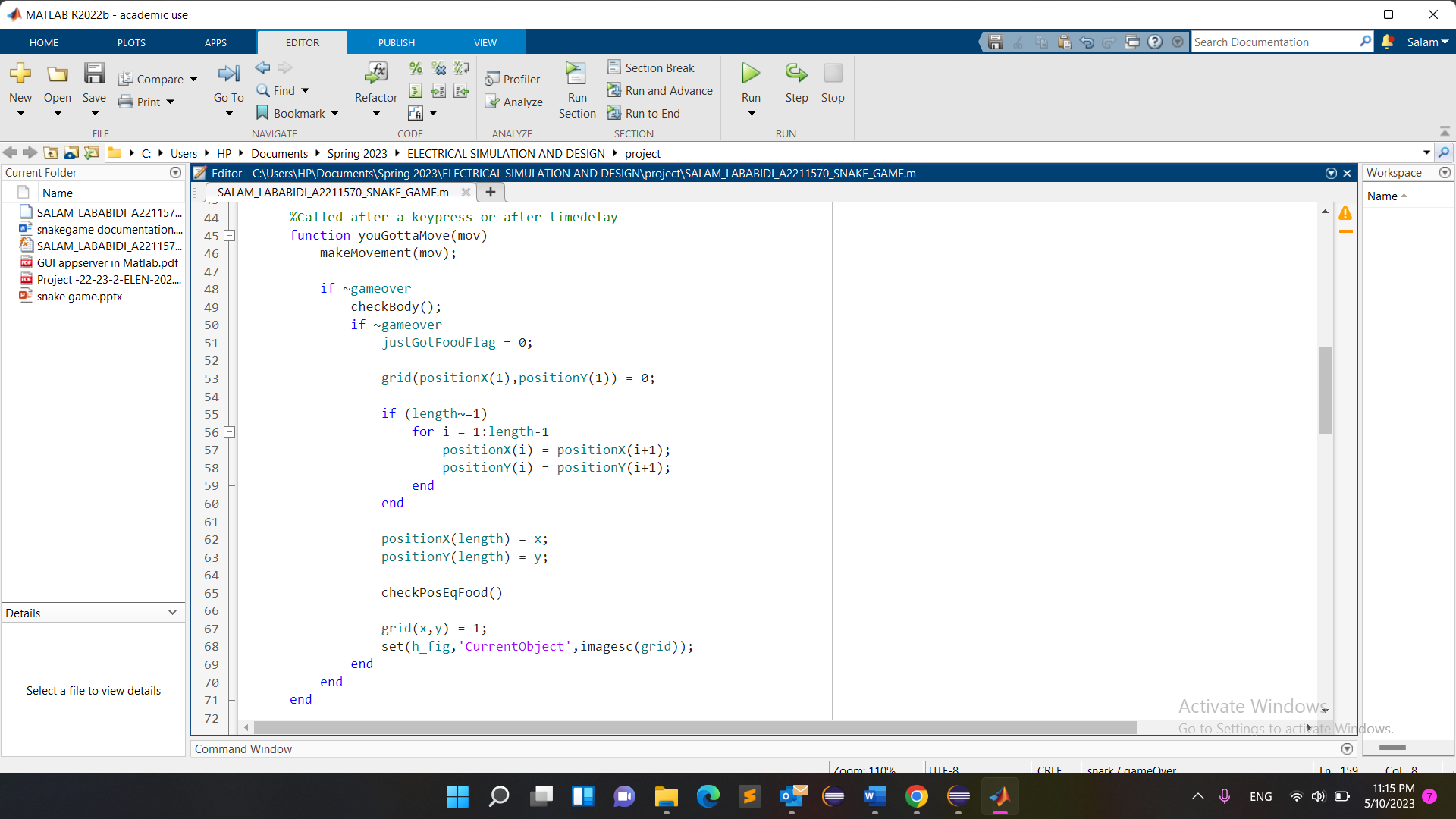
- User input from the arrow keys is captured to control the snake's movement.



*In this code, the keyPress function takes two input arguments: ~ (a placeholder for the handle to the object that triggered the event, which is not used in this case) and evt (an object that represents the key press event). The evt.Key property holds the key that was pressed.*

*The youGottaMove function is then called within keyPress and passed the evt.Key value, which corresponds to the arrow key that was pressed. The youGottaMove function is responsible for updating the snake's position based on the key input.*

*It's important to note that the keyPress function is invoked each time a key is pressed while the figure window has focus, allowing for continuous control of the snake's movement as long as the game is running.*

 - The code continuously updates the snake's position, checks for collisions with walls or the snake's body, and handles food consumption.

*In this code, the youGottaMove function is responsible for updating the snake's position and managing the game logic. Here's an explanation of the different actions performed within this function:*

*makeMovement(mov):*

*This function is called to update the snake's position based on the user's key input (mov).*

*The makeMovement function determines the new position of the snake's head based on the arrow key pressed and updates the x and y variables accordingly.*

*checkBody():*

*This function checks if the snake's head collides with its own body.*

*It iterates through the positions of the snake's body elements and checks if the current head position (x and y) matches any of the body positions.*

*If a collision is detected, the gameOver() function is called.*

*justGotFoodFlag:*

*This flag is used to determine if the snake has just eaten food on the current move.*

*It is initially set to 0 at the start of each move.*

*Updating the snake's body:*

*The code updates the positions of the snake's body elements based on the current head position (x and y).*

*If the snake's length is greater than 1, a loop shifts the body positions by one, effectively moving each body element to the position of the previous element.*

*The head position is then added to the end of the body position arrays.*

*checkPosEqFood():*

*This function checks if the snake's head position (x and y) coincides with the food position (x\_f and y\_f).*

*If a match is found, it means the snake has eaten the food, and the following actions are performed:*

*The snake's length is increased by 1.*

*The new head position is added to the end of the body position arrays.*

*A new food element is generated using the getNewFood() function.*

*The justGotFoodFlag is set to 1.*

*Updating the grid and graphics:*

*The code updates the grid variable to reflect the current positions of the snake's head and body.*

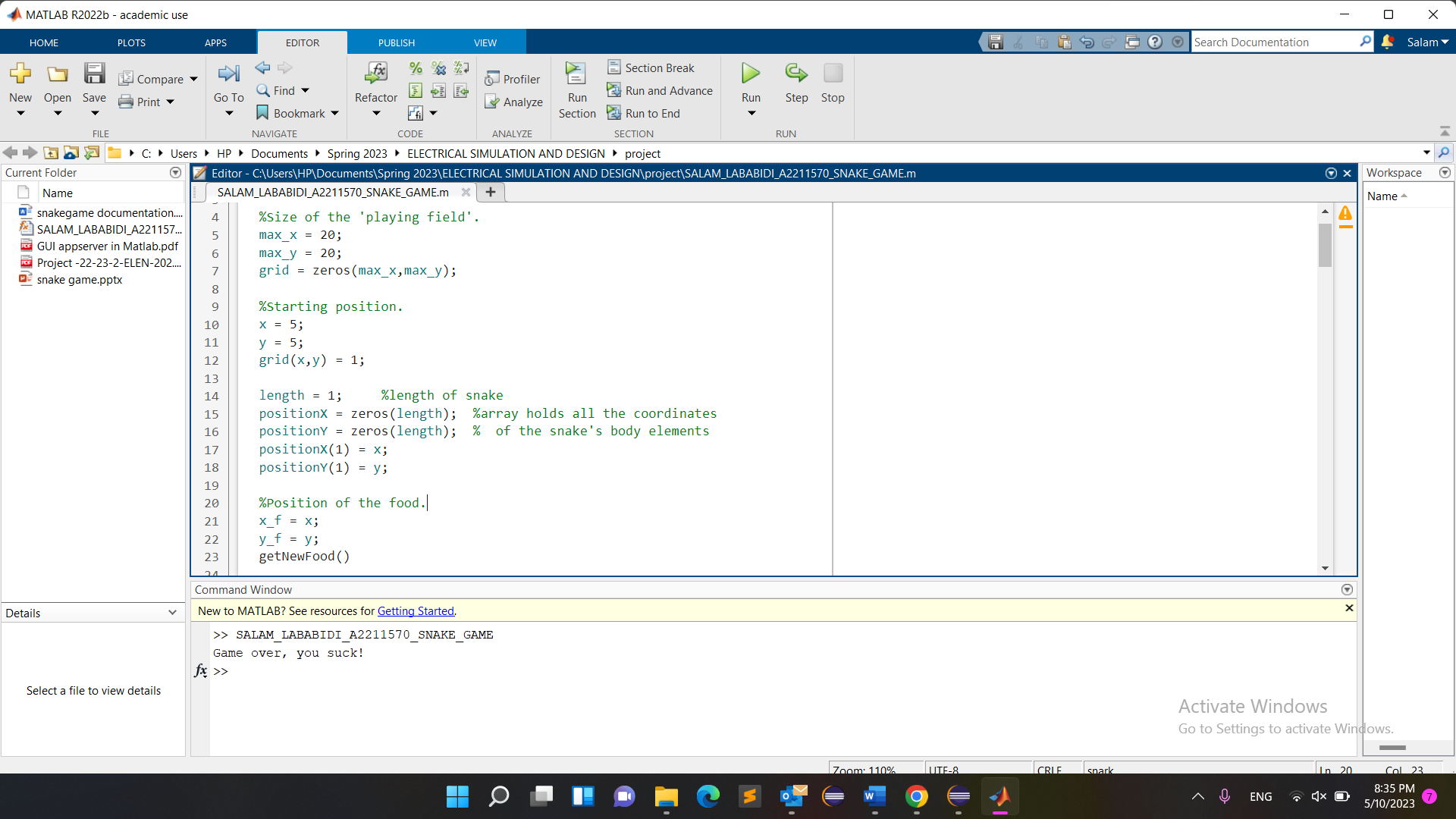
*The imagesc() function is used to display the updated grid in the figure window.*

*Handling game over:*

*If the gameover flag is not set (indicating the game is still running), the relevant actions for the current move are executed.*

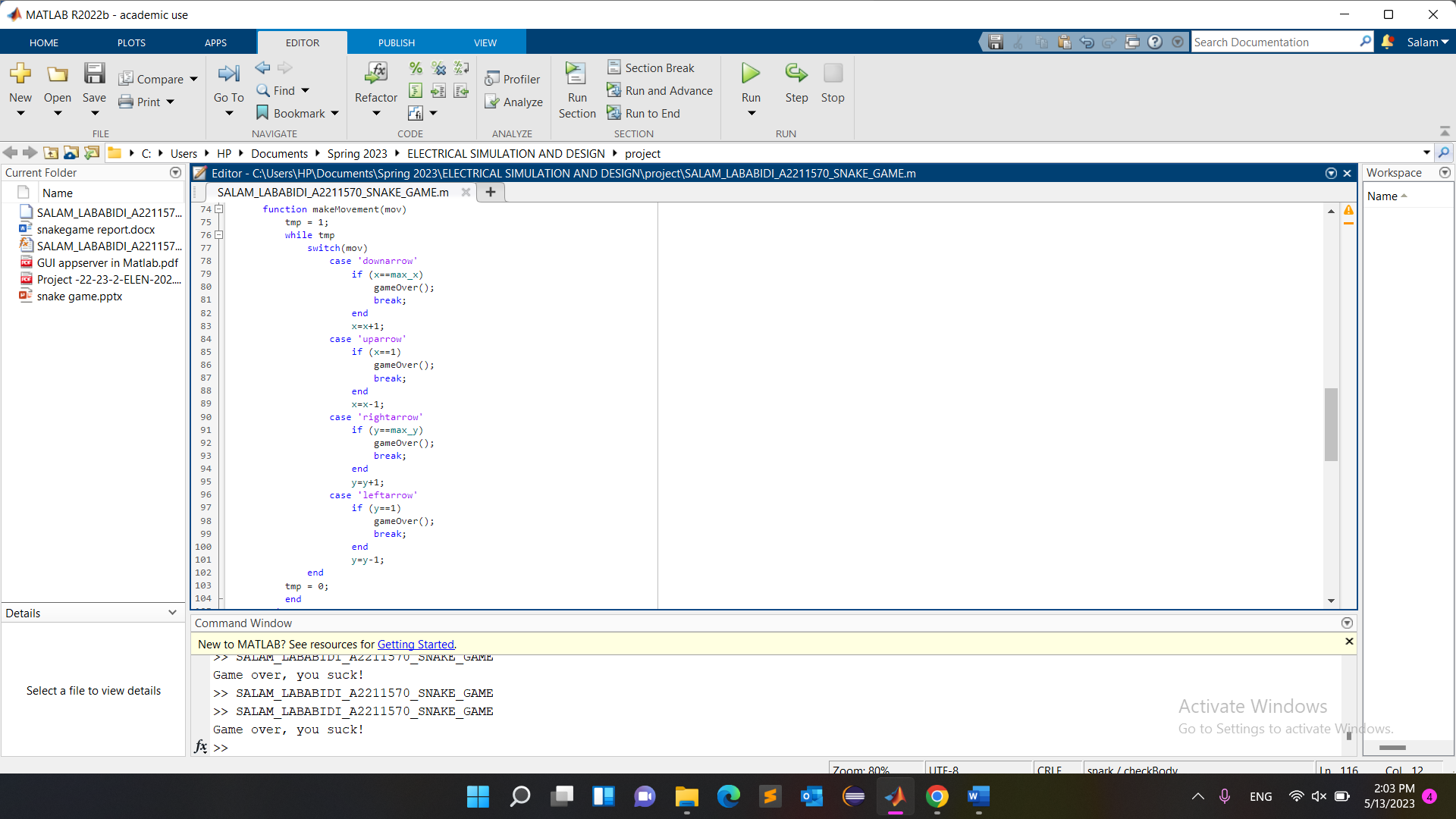
*If the gameover flag is set (indicating a collision occurred), the gameOver() function is called to handle the game over scenario.*

- The game ends when a collision occurs, and displays a Game Over message



- Various helper functions, such as checkBody, checkPosEqFood, getNewFood, and gameOver, are utilized to manage game logic.

makeMovement(mov):

The makeMovement function takes a single argument mov, which represents the player's input or the key that was pressed to change the snake's direction.

The variable tmp is initialized to 1, and a while loop is entered with tmp as the condition.

Inside the while loop, a switch statement is used to check the value of mov, representing the direction the player wants the snake to move.

Depending on the direction, the code performs the following actions:

If the direction is 'downarrow', the code checks if the snake's head is at the maximum x-coordinate (x == max\_x). If true, the gameOver() function is called to end the game. Otherwise, the x coordinate is incremented to move the snake down one position.

If the direction is 'uparrow', the code checks if the snake's head is at the minimum x-coordinate (x == 1). If true, the gameOver() function is called to end the game. Otherwise, the x coordinate is decremented to move the snake up one position.

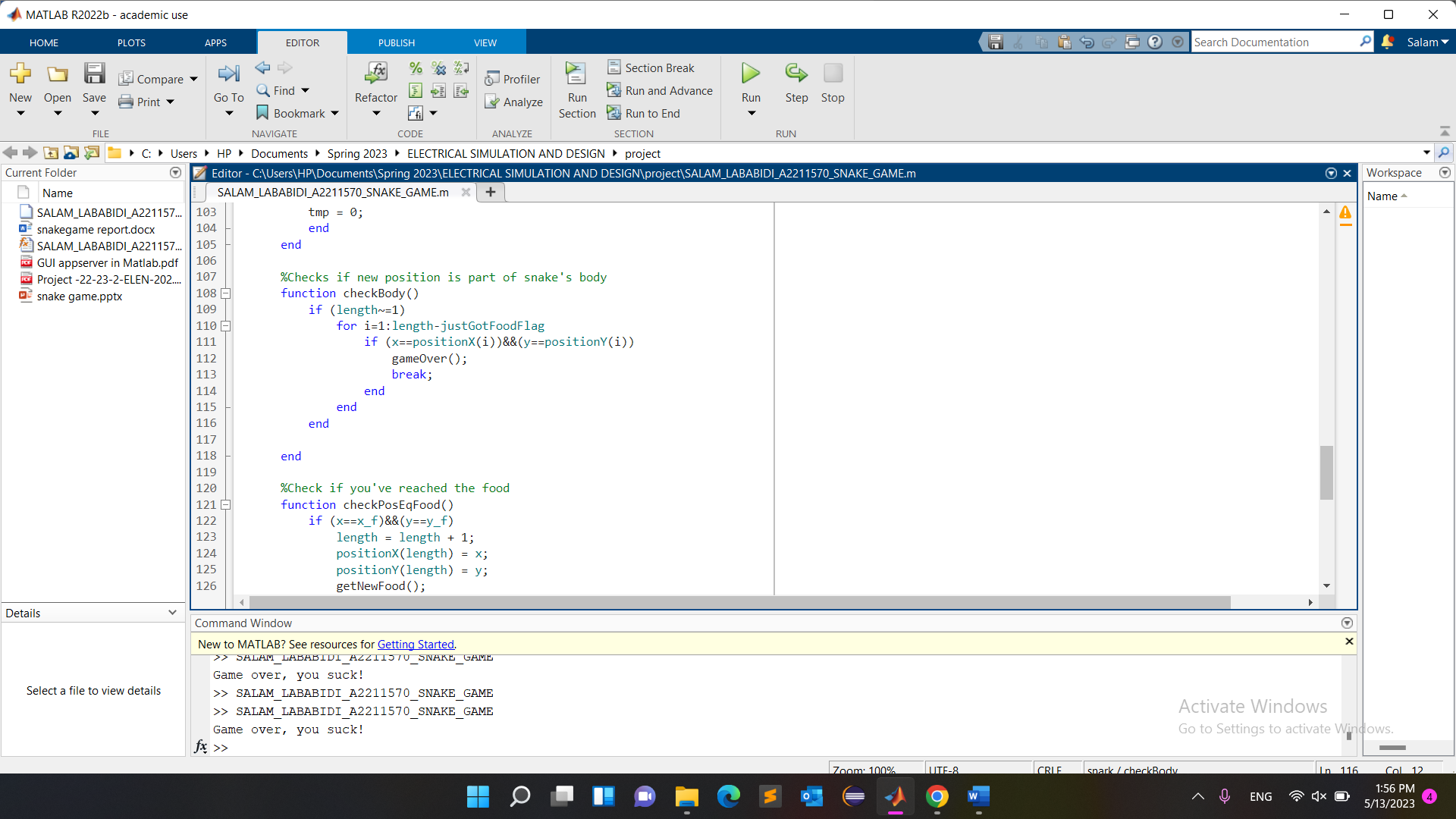
If the direction is 'rightarrow', the code checks if the snake's head is at the maximum y-coordinate (y == max\_y). If true, the gameOver() function is called to end the game. Otherwise, the y coordinate is incremented to move the snake right one position.

If the direction is 'leftarrow', the code checks if the snake's head is at the minimum y-coordinate (y == 1). If true, the gameOver() function is called to end the game. Otherwise, the y coordinate is decremented to move the snake left one position.

After executing the switch statement, tmp is set to 0, which allows the while loop to exit.

The purpose of the makeMovement function is to update the position of the snake's head according to the player's input and handle collisions with the game boundaries. If the snake hits a wall, the gameOver function is called to end the game.

checkBody():

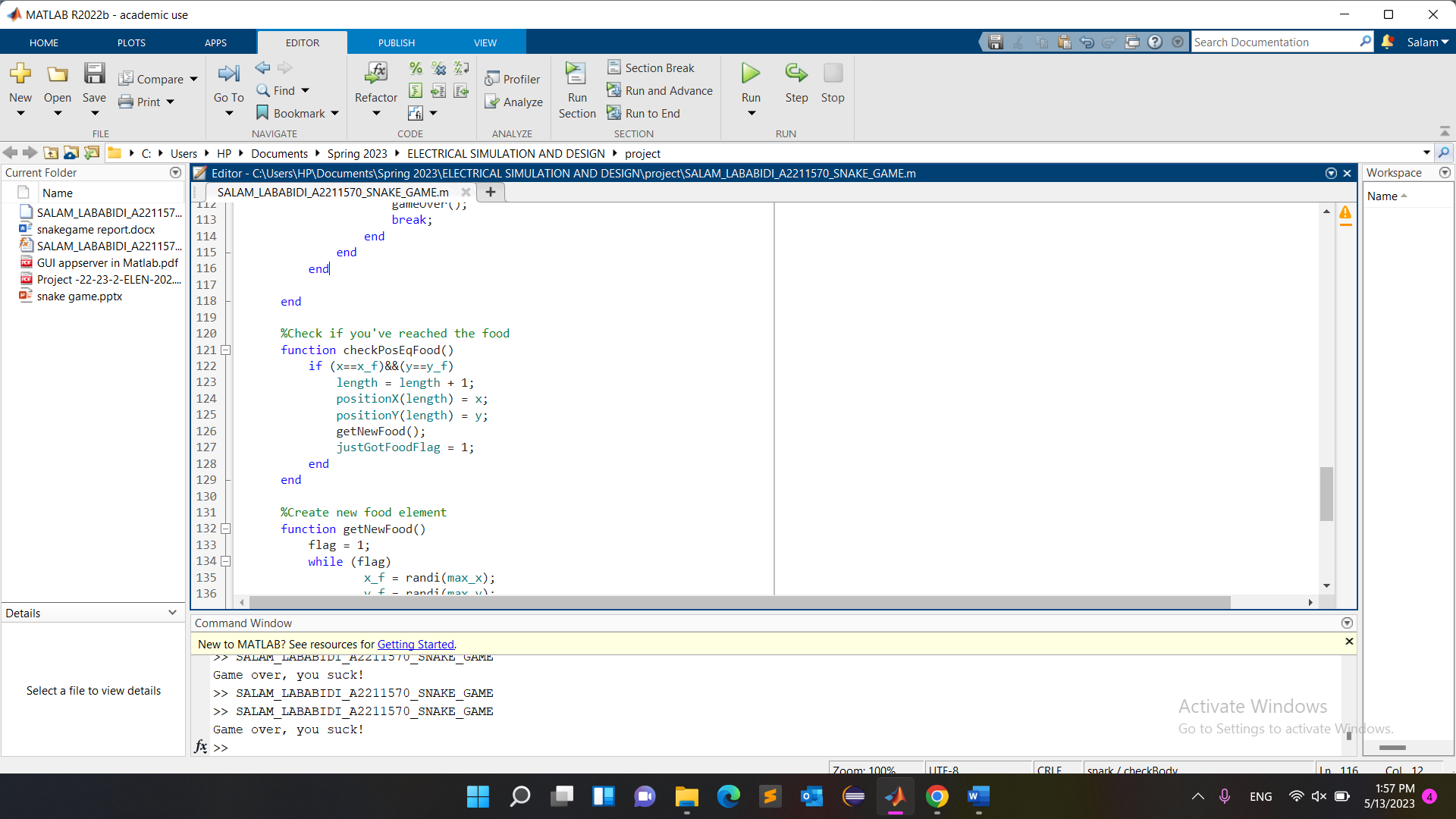


*This function is used within the youGottaMove function.*

*It is called to check if the snake's head collides with its own body.*

*It is used to determine if a game over condition should be triggered.*

*checkPosEqFood():*

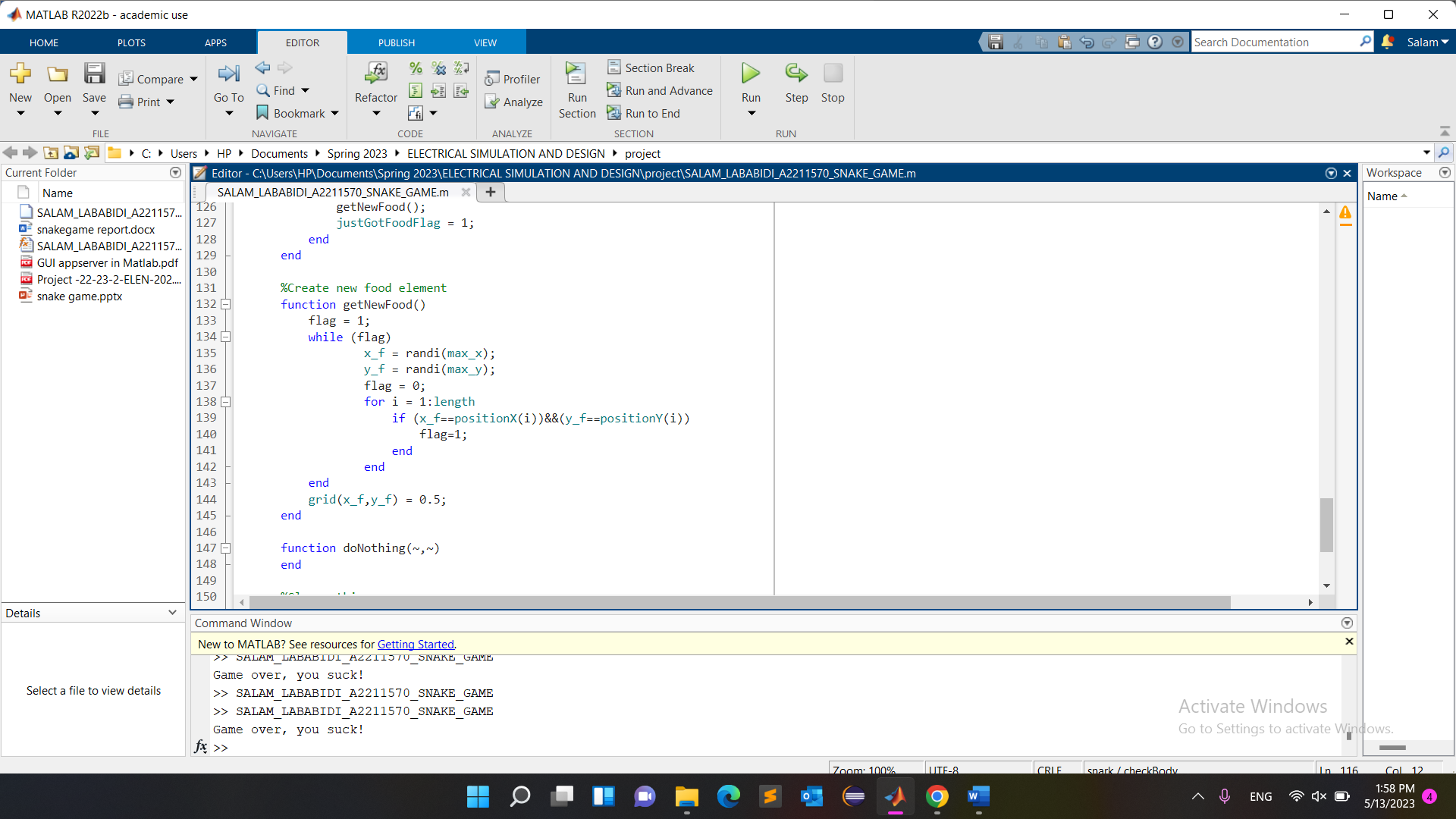
**

*This function is used within the youGottaMove function.*

*It is called to check if the snake's head position coincides with the food position.*

*It is used to determine if the snake has eaten the food and to handle the relevant actions when a food is consumed.*

*getNewFood():*

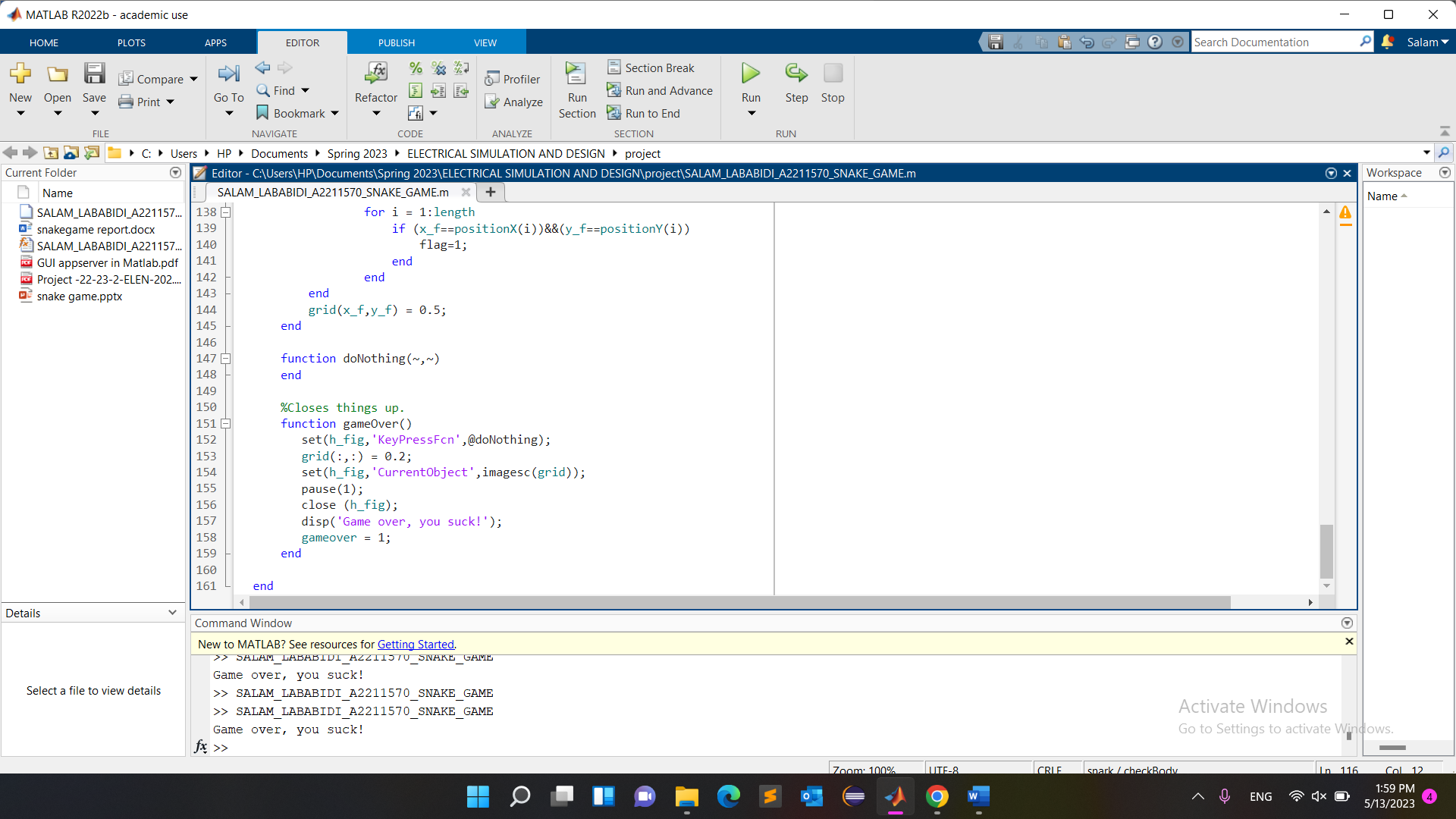
**

*This function is used within the checkPosEqFood function.*

*It is called to generate a new food element with random coordinates on the game grid.*

*It is used to ensure that the food is placed in a valid location that is not occupied by the snake's body.*

*doNothing(~,~):*

**

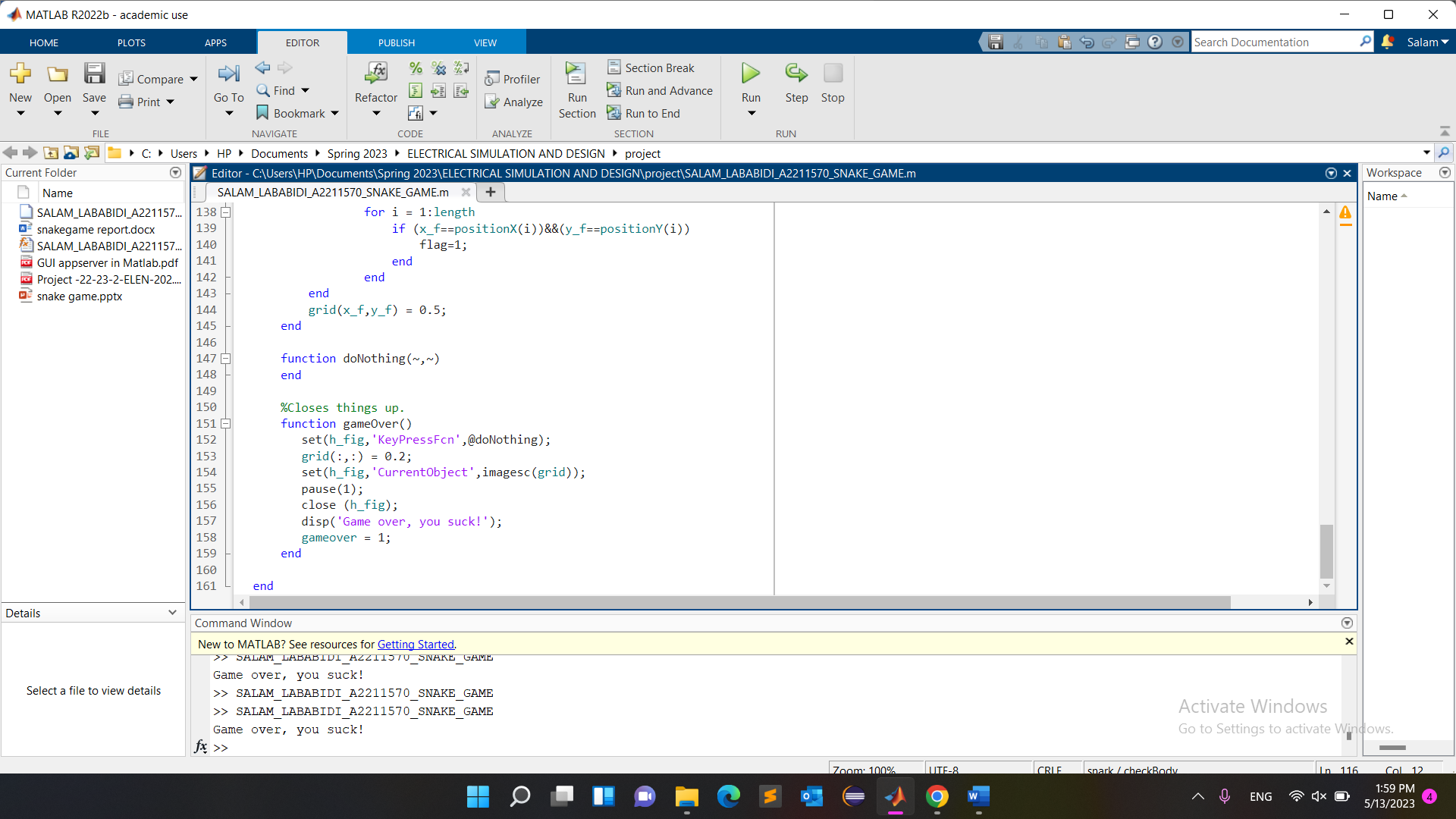
The doNothing function in the provided code serves as an empty callback function. Its purpose is to be assigned to the KeyPressFcn property of the figure window when the game is over.

By assigning @doNothing to the KeyPressFcn property, the function ensures that no action is taken when a key is pressed after the game has ended. Essentially, it disables any further input from the player once the game is over.

In the code, it is used within the gameOver function, where it sets the KeyPressFcn property of the figure to @doNothing. This prevents any key presses from affecting the game state or triggering unintended behavior once the game has concluded.

The doNothing function itself does not contain any code or perform any actions. Its purpose is solely to act as a placeholder callback function to disable key input after the game is over.

*gameOver():*

**

*This function is used within the youGottaMove and keyPress functions.*

*It is called to handle the game over scenario when a collision occurs.*

*It sets the gameover flag, updates the game grid, displays a game over message, and closes the figure window.*

*These helper functions contribute to the overall management of the game's logic, including collision detection, food consumption, and game over handling. They are called at specific points in the code to perform their designated tasks and ensure the smooth execution of the game.*

5. **Results and Analysis:**

- The Snake Game successfully creates an interactive game window where the snake moves and grows based on user input.

- The game's graphics and grid provide a visual representation of the snake, food, and playing field.

- The snake's movement is responsive, and collisions with walls or the snake's body are detected accurately.

6. **Discussion:**

- The Snake Game code demonstrates effective use of MATLAB's graphics capabilities to create an engaging gaming experience.

- The code's modular structure and use of functions allow for easy understanding and extensibility.

- Possible improvements could include adding sound effects, implementing levels with increasing difficulty, or introducing power-ups for added gameplay elements.

7. **Conclusion:**

The MATLAB Snake Game provides an entertaining and nostalgic gaming experience. By navigating the snake, avoiding collisions, and eating food, players can challenge their reflexes and strategic thinking. The code's implementation successfully captures the essence of the Snake Game, and the modular structure allows for future enhancements.

8. **Future** **Work**:

- Levels of Difficulty: implementing multiple levels with increasing difficulty. Introducing faster movement of the snake, narrowing or increasing playing field as levels of difficulty increase, or additional obstacles as the player progresses through levels.

- Multiplayer Mode: adding a multiplayer mode where players can compete against each other or against an AI-controlled snake as an opponent. This could be implemented locally on the same device or through online connectivity.

- High Scores and Leaderboard: implementing a high score system to track and display the top scores achieved by players. As well as adding a leaderboard that showcases the highest scores globally or among friends.

- Customization Options: Allow players to customize the appearance of the snake, the food, or the game grid. Providing options to choose different colors, patterns, or enable themes.

Developed by Mark Abdelmassih