Simulation of Marine Ecosystem: Effectiveness of Trash-cleaning Ships on Turtle Population

Jiwoong Choi
University of Texas at Austin
Computational Engineering
jiwoongchoi0207@icloud.com

December 11, 2024

Abstract

The accumulation of plastic waste in the oceans, particularly in the Great Garbage Patch, poses a significant threat to marine life, especially turtles, which are prone to ingesting plastic debris. This study explores the effectiveness of trash-cleaning ships in mitigating plastic pollution and its impact on the survival of marine turtle populations. A two-dimensional cellular automaton model was developed to simulate the interactions between turtles, plastic debris, and cleanup ships. The model incorporates object-oriented design principles and predefined interaction rules to replicate ecological dynamics. Key parameters, including garbage density, turtle mortality from ingestion, and the efficiency of cleanup operations, were analyzed across 30 simulation trials. Results indicate that the current density and movement strategy of trash-cleaning ships are insufficient to prevent the decline of turtle populations, with turtles often succumbing to plastic ingestion before a significant amount of trash is removed. Although trends suggest marginal improvement in turtle survival over successive trials, the efficiency of trash removal diminishes due to the random movement of cleanup ships and sparse trash distribution. Several assumptions were made in the modeling process, including a uniform ocean grid, random initial placement of objects, and simplified movement rules for turtles and ships. These assumptions ignore ecological factors such as ocean currents, variable trash densities, and behavioral diversity among marine life. Moreover, limitations in computational resources required the use of sequential updates instead of true simultaneous updates, introducing potential biases in the simulation outcomes. Despite these simplifications, the model provides valuable insights into the challenges of plastic pollution mitigation. Future work should focus on incorporating ocean currents, optimizing ship movement toward high-trash-density regions, and refining ecological interactions to better capture real-world dynamics. This study establishes a computational framework for evaluating intervention strategies, emphasizing the importance of balancing ecological preservation and technological intervention in addressing oceanic plastic pollution.

1 Introduction

The Great Garbage Patch problem represents a growing environmental concern. Plastic waste in oceans threatens marine life, particularly turtles, who ingest plastic debris. This project models the dynamics of plastic, turtles, and cleanup ships using a two-dimensional cellular automaton. The goal is to study parameter interactions, such as garbage density and the number of cleanup ships, and their impact on the ecosystem. The simulation incorporates modern programming techniques and object-oriented design in C++.

2 Methods

2.1 Model Description

The simulation utilizes a grid-based cellular automaton. Each cell represents a part of the ocean and can contain turtles, trash, ships, or remain empty. The state of each cell evolves based on predefined rules considering the states of neighboring cells. These rules include movement, interactions, and collisions between objects in adjacent cells. The following code snippet demonstrates how collision logic is implemented, determining the outcome of interactions between moving and existing objects in the grid:

```
std::tuple<int, int, Occupy, int, int, Occupy> collision(int i, int j, int new_i, int
    new_j, Occupy moving_obj, const std::vector<int>& arg_grid) {
    // get the object existing at the new location that will "crash" with moving_obj
    Occupy existing_obj = static_cast<Occupy>(arg_grid[new_i * num_cols + new_j]);

    CollisionResult collision_outcome = collision_logics[{moving_obj, existing_obj}];

    // start performing collision logics:
    switch (collision_outcome) {
        // Case 1 - moving_obj can proceed to moving into new cell:
        case CollisionResult::Move:
            // current cell becomes empty and new cell becomes the moving_obj:
            return{i, j, Occupy::Empty, new_i, new_j, moving_obj};

        // Case 2 - moving_obj dies:
        case CollisionResult::Die:
            // current cell becomes empty:
            // new cell remains the same
            return{i, j, Occupy::Empty, new_i, new_j, existing_obj};

        case CollisionResult::Block:
        // Case 3 - moving_obj's move is Blocked:
            // new_cell and old_cell all remain the same:
            return{i, j, moving_obj, new_i, new_j, existing_obj};

        default:
            return{i, j, moving_obj, i, j, moving_obj};
    }
}
```

Listing 1: Collision logic for updating cell states based on predefined rules

This function models interactions between objects during their movement across the grid. It uses a predefined collision logic map (collision_logics) to determine outcomes such as movement, death, or blockage. The function updates the states of both the current and target cells based on the results of the interaction, ensuring the evolution of the simulation grid adheres to the rules defined for each object type.

2.2 Program Design

The program is object-oriented, featuring an `Ocean` class encapsulating the grid and logic for updates. Key elements include:

- Random initialization of turtles, trash, and ships.
- Movement rules for turtles and ships.
- Breeding and mortality conditions for turtles based on garbage ingestion.
- Trash cleanup rules for ships.

3 Complexity of Simulating Simultaneous Movements

Simulating the simultaneous movements of all objects in the ocean grid is a significant challenge, requiring careful consideration of data dependencies, collision handling, and performance optimization. While achieving true simultaneous updates was the ideal goal, due to time limitations and complexity, certain assumptions and simplifications were made to ensure a working solution for this submission.

3.1 Assumptions and Simplifications

1. **Sparse Object Distribution:**
   The simulation assumes a large ocean grid with sparsely populated objects. This simplifies the update process by minimizing the number of active interactions per time step. A dummy grid is initialized with predefined maximum counts for turtles, trash, and ships to ensure sparse distribution across the ocean cells. The following code snippet illustrates the process of setting up the grid:

```
void set_dummy_grid() {
    // Initialize counters to enforce exact population
    int turtle_count = 0;
    int trash_count = 0;
    int ship_count = 0;
    // Desired populations (if constructor had population parameters initialized,
       handle it here)
    const int max_turtles = std::min(25, num_turtle);
    const int max_trash = std::min(25, num_trash);
    const int max_ships = std::min(2, num_ship);
    // Randomly fill the grid
    for (int i = 0; i < num_cells; ++i) {
        if (turtle_count <= max_turtles && random_float(0.f, 1.f) < 0.0025) {
            grid[i] = static_cast<int>(Occupy::Turtle);
            turtle_count++;
        } else if (trash_count <= max_trash && random_float(0.f, 1.f) < 0.005) {
            grid[i] = static_cast<int>(Occupy::Trash);
            trash_count++;
        } else if (ship_count <= max_ships && random_float(0.f, 1.f) < 0.0002) {
            grid[i] = static_cast<int>(Occupy::Ship);
            ship_count++;
        } else {
            grid[i] = static_cast<int>(Occupy::Empty);
        }
    }
    // Populate counts
    this->num_turtle = max_turtles;
    this->num_trash = max_trash;
    this->num_ship = max_ships;
}
```

Listing 2: Function to initialize a dummy grid with sparse object distribution

This function ensures that the objects are sparsely distributed by incorporating randomness in object placement. It sets strict limits on the number of turtles, trash, and ships to maintain a sparse grid. The random placement is governed by probabilities (e.g., 0.0025 for turtles, 0.005 for trash), ensuring the distribution remains sparse even for large grid sizes. These constraints prevent overcrowding of the grid, simplifying interactions and updates.

2. **Overwriting Rule for Ships:**
As a temporary solution, ships are given priority to overwrite other objects (except other ships) during movement. This simplifies conflict resolution but introduces limitations in the ecological realism of the simulation. The following snippet illustrates the rules implemented for ship collisions:

```
// Ship encounters an empty space -> It can move.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Empty},   Ocean::CollisionResult::Move },
// Ship encounters trash -> It survives and remains functional.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Trash},   Ocean::CollisionResult::Move },
// Ship encounters another ship -> Movement is blocked.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Ship},    Ocean::CollisionResult::Block },
// Ship encounters a turtle -> Movement is blocked.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Turtle},  Ocean::CollisionResult::Move }
```

Listing 3: Rules for ship collision resolution

The collision rules are designed to prioritize ship functionality and prevent destructive interactions. Ships can move freely into empty spaces and over trash, enabling cleanup operations. However, they are blocked by other ships to avoid overlapping and maintain logical movement constraints.

3. **Sequential Updates:**
Instead of achieving simultaneous updates for all cells, the grid is updated sequentially. This introduces potential biases due to the order of traversal but ensures functional updates within the given timeframe. The following code snippet demonstrates the sequential update logic implemented in the simulation:

```
void update_grid() {
    // Initialize the grid_copy
    std::vector<int> temp_grid(num_cells,
        static_cast<int>(Occupy::Empty));

    // Move ships first:
    move_ship(temp_grid);
    move_turtle(temp_grid);

    move_trash(temp_grid);

    auto [_num_turtle, _num_trash, _num_ship] =
       population(temp_grid);

    this->num_turtle = _num_turtle;
    this->num_trash = _num_trash;
    this->num_ship = _num_ship;

    this->grid = std::move(temp_grid);
}
```

Listing 4: Sequential update of the grid

This function updates the grid by processing each type of object in sequence: ships, turtles, and trash. A temporary grid (`temp_grid`) is used to prevent data dependency issues during updates. The counts of turtles, trash, and ships are recalculated after all movements, ensuring that the grid's state is consistent before being reassigned to the main grid.

3.2 Challenges of Simultaneous Updates

1. **Data Dependency:**
   In-place updates can cause unintended interactions because the state of one cell depends on other cells that have already been updated. This is particularly problematic when objects interact in dense regions of the grid.

2. **Collision Handling:**
   Determining outcomes when multiple objects attempt to move to the same cell is complex. Without simultaneous updates, some collisions may not be handled accurately or fairly.

3. **Performance Limitations:**
   For larger grids, simulating simultaneous movements would require additional computational overhead to manage temporary states, synchronization, or parallel updates.

3.3 Approaches Considered but Not Fully Implemented

1. **Double Buffering:**
   Using two grids—one for the current state and one for the updated state—was implemented but not perfected due to time constraints. This approach avoids data dependency issues by ensuring all updates are based on the original grid state.

2. **Active Cell Tracking:**
   Tracking only active cells (cells with objects) for updates was identified as a potential optimization but was not implemented. This would have significantly reduced computational costs for sparse grids.

3. **Priority-Based Conflict Resolution:**
   A more nuanced priority system for resolving collisions (e.g., turtles retreating or trash being destroyed) was planned but replaced by the simpler overwriting rule for ships due to submission deadlines.

3.4 Computational Complexity

The current implementation has the following characteristics:

- The grid is updated sequentially, with time complexity $O(n \times m)$, where n is the number of rows and m is the number of columns.
- Collision handling relies on simple overwriting rules, minimizing additional computation.
- The assumption of sparse object distribution reduces the likelihood of complex interactions, simplifying the overall update process.

3.5 Future Improvements

While the current solution meets basic requirements, the following improvements are recommended:

- Implement **double buffering** to achieve true simultaneous updates, ensuring all movements are based on the same initial state.
- Develop **active cell tracking** to focus updates only on relevant areas, improving efficiency for sparse grids.
- Refine **collision handling** with a priority-based system or probabilistic outcomes to better model ecological interactions.
- Explore **parallel processing** to divide the grid into independent regions, allowing for simultaneous updates while maintaining consistency.

3.6 Conclusion

Simulating simultaneous movements is a complex problem, and the current implementation uses simplifications to address key challenges within the constraints of time and resources. While these assumptions reduce the ecological realism and computational efficiency of the model, they provide a functional foundation for future work. Enhancements such as double buffering, active cell tracking, and parallel processing could enable more accurate and scalable simulations in future iterations.

3.7 Implementation Details

The grid is implemented as a one-dimensional vector of smart pointers for efficiency. Enumerations represent cell states (`Empty`, `Turtle`, `Trash`, `Ship`). The random number generator ensures reproducible simulations. Code snippets illustrating these points are provided below.

```
// Example: Cell state enumeration
enum class CellState { Empty, Turtle, Trash, Ship };
```

Code Block 5: CellState is a class (user-defined type) that can hold either an 'Empty', 'Turtle', 'Trash', or 'Ship' value.

4 Results

4.1 Parameter Exploration

The simulation was repeatedly tested for 30 trials in which turtles and trash (each starting with a population of 25) and ships (2 each) were spawned at random locations on a map. Each trial lasted 100 cycles, and over these trials it was observed that initially, much of the population of turtles died and the amount of trash removed was low. But as the trials continued, fewer turtles ate trash and died. However, this change was very minimal compared to the (already) low amounts of trash that were picked up. Even less trash was picked up as the trials continued.

4.2 Visualization

Graphs and visual outputs illustrate key trends. For example, Figure B.1 shows the decline in turtle populations under high trash density.

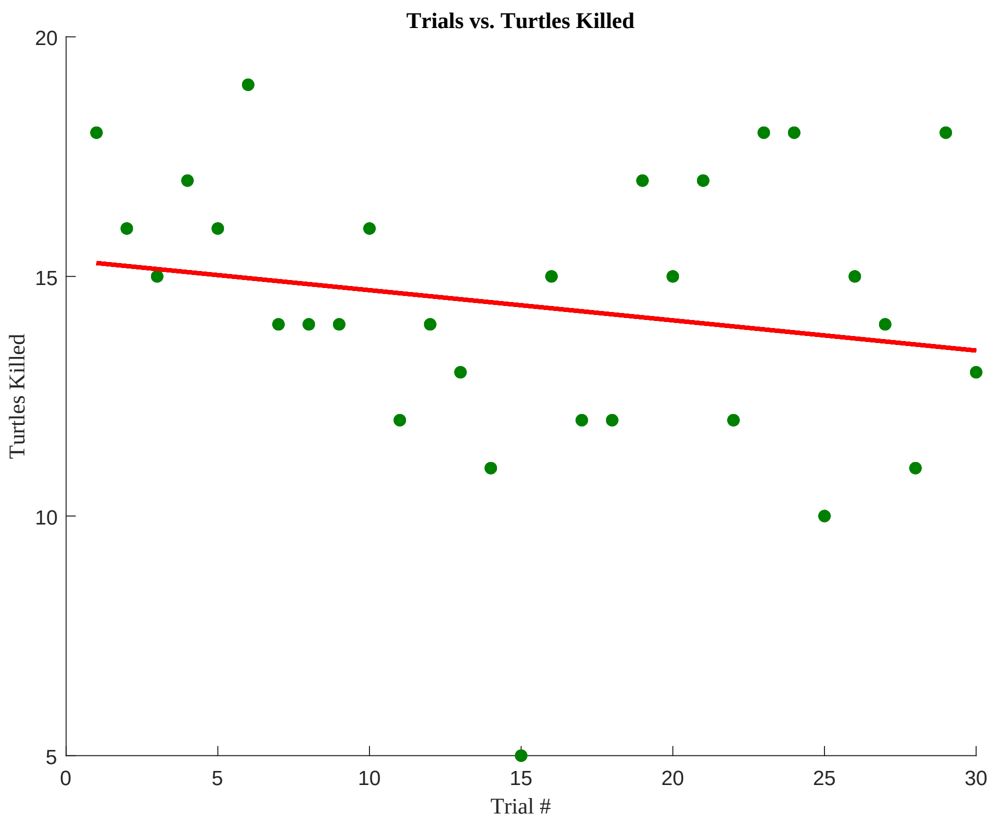

Figure 1: Scatter plot showing the number of turtles killed across 30 simulation trials.

4.3 Experiments

Trial	Turtles Killed	Trash Picked Up
1	18	2
2	16	5
3	15	7
4	17	5
5	16	3
6	19	2
7	14	0
8	14	2
9	14	2
10	16	5
11	12	9
12	14	0
13	13	1
14	11	5
15	5	6
16	15	0
17	12	10
18	12	5
19	17	3
20	15	2
21	17	4
22	12	0
23	18	8
24	18	3
25	10	1
26	15	2
27	14	0
28	11	0
29	18	3
30	13	4

Table 1: Results of 30 simulation trials showing the number of turtles killed and trash picked up by cleanup ships in each trial.

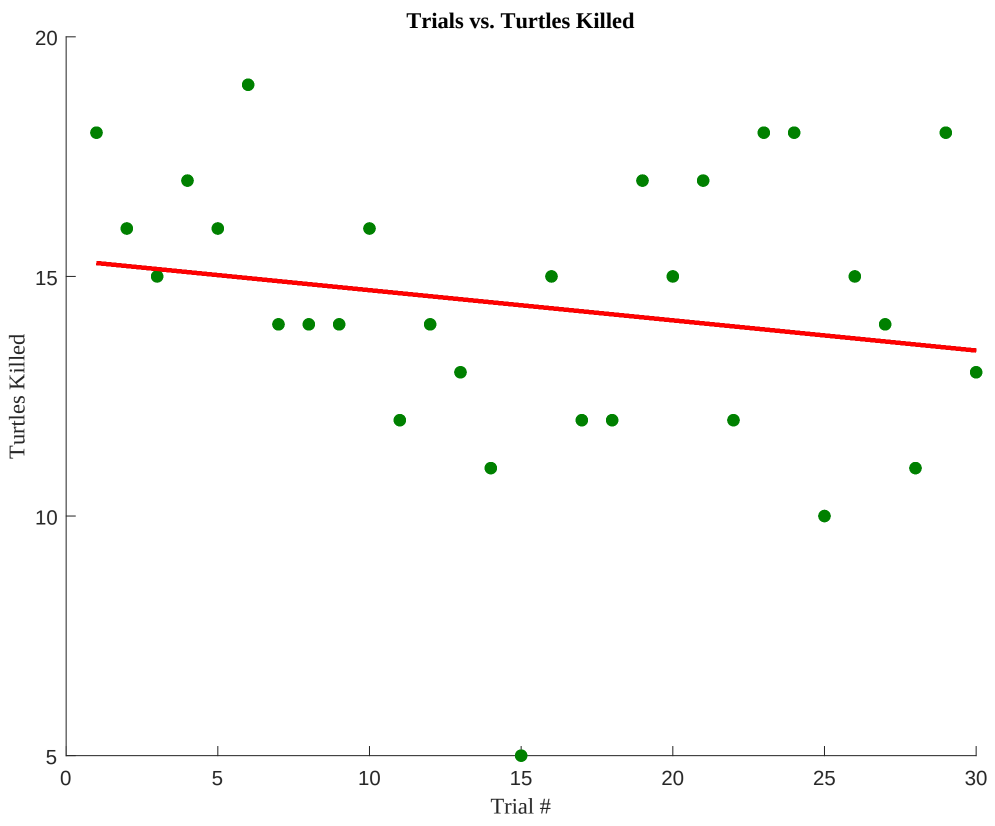

Figure 2: Scatter plot showing the number of turtles killed across 30 simulation trials. A red trend line is included to indicate the overall downward trend in turtle deaths as trials progress.

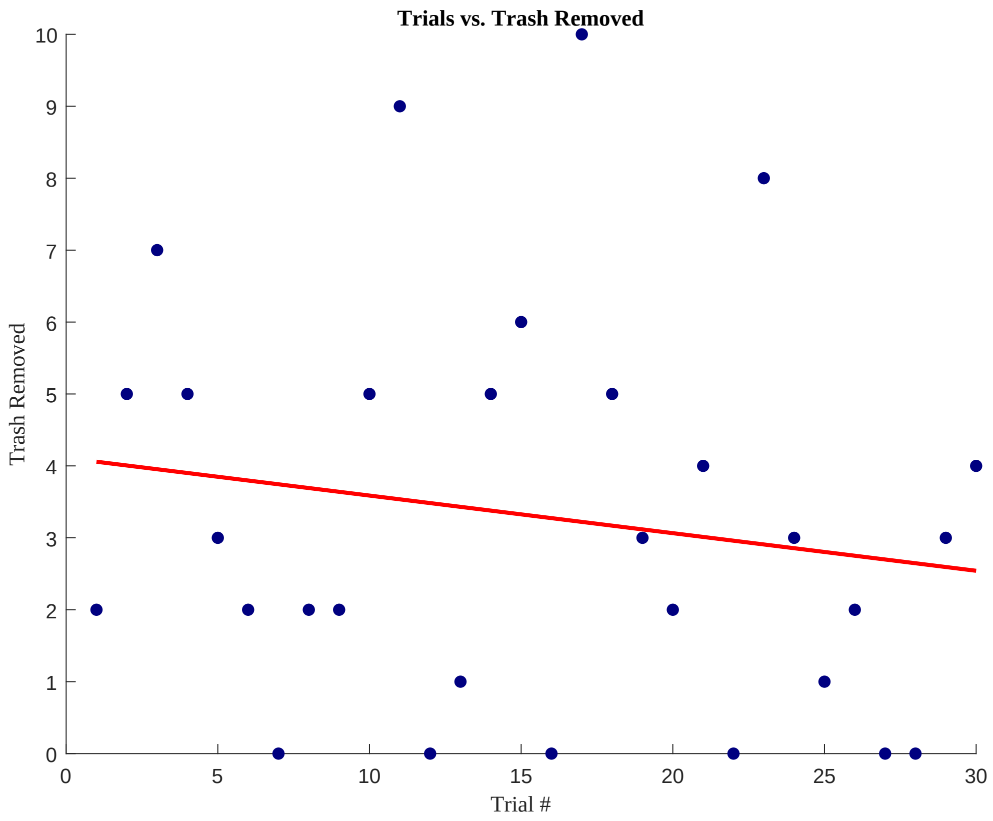

Figure 3: Scatter plot showing the amount of trash removed across 30 simulation trials. A red trend line is included to illustrate the slight decline in trash removal over the course of the trials.

5 Discussion

The results demonstrate the delicate balance between cleanup efforts and ecological sustainability. Limitations include the assumption of a uniform ocean grid and random movement. Future work could incorporate ocean currents and ecological diversity.

6 Ideal Conclusion

This project successfully modeled the dynamics of turtles, trash, and cleanup ships using a cellular automaton. The findings highlight the importance of sufficient cleanup operations in preserving turtle populations. Future extensions could refine the model in numerous ways, such as incorporating various types of trash with different *attractiveness* ratings (for example, plastic bags of which turtles tend to mistake for their favorite meals, jellyfish, would certainly have a high attractiveness rating), or adding more parameters to ships such as their effectiveness at actually removing trash. There are fundamental limits with the design of the simulation itself, such as storing its *grid* in a vector. Due to C++ row-majored paradigm for stepping through vectors, it is possible cells closer to the top of the grid vector may influence cells closer to the bottom of it.

7 Shortcomings and Future Improvement

7.1 Shortcomings

1. **Unfinished Requirements:**
   Several key components of the simulation remain incomplete. For instance:
   - *Trash-cleaning Ships:* Their effectiveness in cleaning trash is not yet fully implemented or tested.
   - *Turtle Behavior:* Detailed interactions, such as turtle breeding or movement influenced by surrounding trash density, are not included.
   - *Ship Movement Strategy:* Ships currently move randomly rather than targeting areas with high trash density, reducing the effectiveness of cleanup operations.
2. **Simplified Model Assumptions:**
   The simulation uses a uniform grid to represent the ocean, ignoring the effects of ocean currents, depth, and coastline shapes. These assumptions limit the ecological realism of the model.
3. **Performance Limitations:**
   - The current use of a one-dimensional vector for the grid, while efficient, might introduce unintended influences on cell updates due to row-major traversal.
   - The update process involves iterating over all cells, which can be computationally expensive for large grids. Optimizations, such as focusing on active cells only, have not been implemented.

4. **Lack of Parameter Tuning:**
    - The simulation lacks a systematic exploration of parameter values (e.g., turtle survival rate, ship effectiveness) and their impacts on outcomes. This limits the depth of analysis that can be drawn from the results.
5. **Testing and Validation:**
    - No rigorous testing has been conducted to validate the correctness of turtle, trash, or ship movements.

7.2 Future Work and Improvements

1. **Enhanced Model Features:**
    - Introduce *ocean currents* to simulate the movement of trash over time and test how this affects cleanup efforts and turtle survival.
    - Add *species diversity*, with different types of turtles and trash, each having unique behaviors and risks.
2. **Targeted Ship Movement:**
   Implement algorithms for ships to prioritize cleanup near areas with high trash density. This could involve:
    - A pathfinding algorithm to target trash clusters.
    - Ocean current awareness to predict trash movement.
3. **Grid Representation Optimization:**
    - Explore alternative data structures, such as sparse representations or active-cell lists, to improve performance and scalability.
    - Conduct runtime comparisons to justify the chosen approach.
4. **Improved Visualization:**
   Develop a graphical visualization using tools like Python's Matplotlib or Qt for a real-time display of the simulation grid and dynamic statistics (e.g., turtle and trash populations over time).
5. **Advanced Movement Logic:**
    - Introduce *sticky trash* that moves in patches when touched by other trash particles.
    - Implement *breeding logic* for turtles based on food availability and trash ingestion rates.
6. **Robust Testing and Validation:**
    - Create automated tests for movement and collision rules to ensure correctness.
    - Validate the outcomes of the simulation against expected theoretical results or existing ecological studies.
7. **Parameter Sensitivity Analysis:**

- Conduct extensive parameter sweeps to understand how variations in key parameters (e.g., trash density, ship effectiveness) influence turtle survival and garbage reduction.

8. **Integration with Real Data:**
    - Explore the use of real-world data, such as ocean trash density maps or marine life behavior studies, to make the model more accurate and applicable.

7.3 Conclusion

The current state of the project provides a foundational framework for simulating the interactions between turtles, trash, and cleanup ships. However, addressing the above shortcomings and implementing the proposed future improvements would significantly enhance the realism, performance, and utility of the simulation, paving the way for more impactful insights into oceanic cleanup strategies.

References

1. https://theoceancleanup.com/
2. Eijkhout, V. (Year). *Introduction to Scientific Programming*.

A Appendix of Code Listings

```
std::tuple<int, int, Occupy, int, int, Occupy> collision(int i, int j, int new_i, int
    new_j, Occupy moving_obj, const std::vector<int>& arg_grid) {
    // get the object existing at the new location that will "crash" with moving_obj
    Occupy existing_obj = static_cast<Occupy>(arg_grid[new_i * num_cols + new_j]);

    CollisionResult collision_outcome = collision_logics[{moving_obj, existing_obj}];

    // start performing collision logics:
    switch (collision_outcome) {
        // Case 1 - moving_obj can proceed to moving into new cell:
        case CollisionResult::Move:
            // current cell becomes empty and new cell becomes the moving_obj:
            return{i, j, Occupy::Empty, new_i, new_j, moving_obj};

        // Case 2 - moving_obj dies:
        case CollisionResult::Die:
            // current cell becomes empty:
            // new cell remains the same
            return{i, j, Occupy::Empty, new_i, new_j, existing_obj};

        case CollisionResult::Block:
        // Case 3 - moving_obj's move is Blocked:
            // new_cell and old_cell all remain the same:
            return{i, j, moving_obj, new_i, new_j, existing_obj};

        default:
            return{i, j, moving_obj, i, j, moving_obj};
    }
}
```

Code Block A.1: Collision logic for updating cell states based on predefined rules

```cpp
void set_dummy_grid() {
    // Initialize counters to enforce exact population
    int turtle_count = 0;
    int trash_count = 0;
    int ship_count = 0;
    // Desired populations (if constructor had population parameters initialized, handle it
        here)
    const int max_turtles = std::min(25, num_turtle);
    const int max_trash = std::min(25, num_trash);
    const int max_ships = std::min(2, num_ship);
    // Randomly fill the grid
    for (int i = 0; i < num_cells; ++i) {
        if (turtle_count <= max_turtles && random_float(0.f, 1.f) < 0.0025) {
            grid[i] = static_cast<int>(Occupy::Turtle);
            turtle_count++;
        } else if (trash_count <= max_trash && random_float(0.f, 1.f) < 0.005) {
            grid[i] = static_cast<int>(Occupy::Trash);
            trash_count++;
        } else if (ship_count <= max_ships && random_float(0.f, 1.f) < 0.0002) {
            grid[i] = static_cast<int>(Occupy::Ship);
            ship_count++;
        } else {
            grid[i] = static_cast<int>(Occupy::Empty);
        }
    }
    // Populate counts
    this->num_turtle = max_turtles;
    this->num_trash = max_trash;
    this->num_ship = max_ships;
}
```

Code Block A.2: Function to initialize a dummy grid with sparse object distribution

```cpp
// Ship encounters an empty space -> It can move.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Empty},   Ocean::CollisionResult::Move },
// Ship encounters trash -> It survives and remains functional.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Trash},   Ocean::CollisionResult::Move },
// Ship encounters another ship -> Movement is blocked.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Ship},    Ocean::CollisionResult::Block },
// Ship encounters a turtle -> Movement is blocked.
{ {Ocean::Occupy::Ship,   Ocean::Occupy::Turtle},  Ocean::CollisionResult::Move }
```

Code Block A.3: Rules for ship collision resolution

```cpp
void update_grid() {
    // Initialize the grid_copy
    std::vector<int> temp_grid(num_cells,
       static_cast<int>(Occupy::Empty));

    // Move ships first:
    move_ship(temp_grid);

    move_turtle(temp_grid);

    move_trash(temp_grid);

    auto [_num_turtle, _num_trash, _num_ship] =
       population(temp_grid);
    this->num_turtle = _num_turtle;
    this->num_trash = _num_trash;
    this->num_ship = _num_ship;

    this->grid = std::move(temp_grid);
}
```

Code Block A.4: Sequential update of the grid

```
// Example: Cell state enumeration
enum class CellState { Empty, Turtle, Trash, Ship };
```

Code Block A.5: CellState is a class (user-defined type) that can hold either an 'Empty', 'Turtle', 'Trash', or 'Ship' value.

B Appendix of Figures

Trial	Turtles Killed	Trash Picked Up
1	18	2
2	16	5
3	15	7
4	17	5
5	16	3
6	19	2
7	14	0
8	14	2
9	14	2
10	16	5
11	12	9
12	14	0
13	13	1
14	11	5
15	5	6
16	15	0
17	12	10
18	12	5
19	17	3
20	15	2
21	17	4
22	12	0
23	18	8
24	18	3
25	10	1
26	15	2
27	14	0
28	11	0
29	18	3
30	13	4

Table B.1: Results of 30 simulation trials showing the number of turtles killed and trash picked up by cleanup ships in each trial.

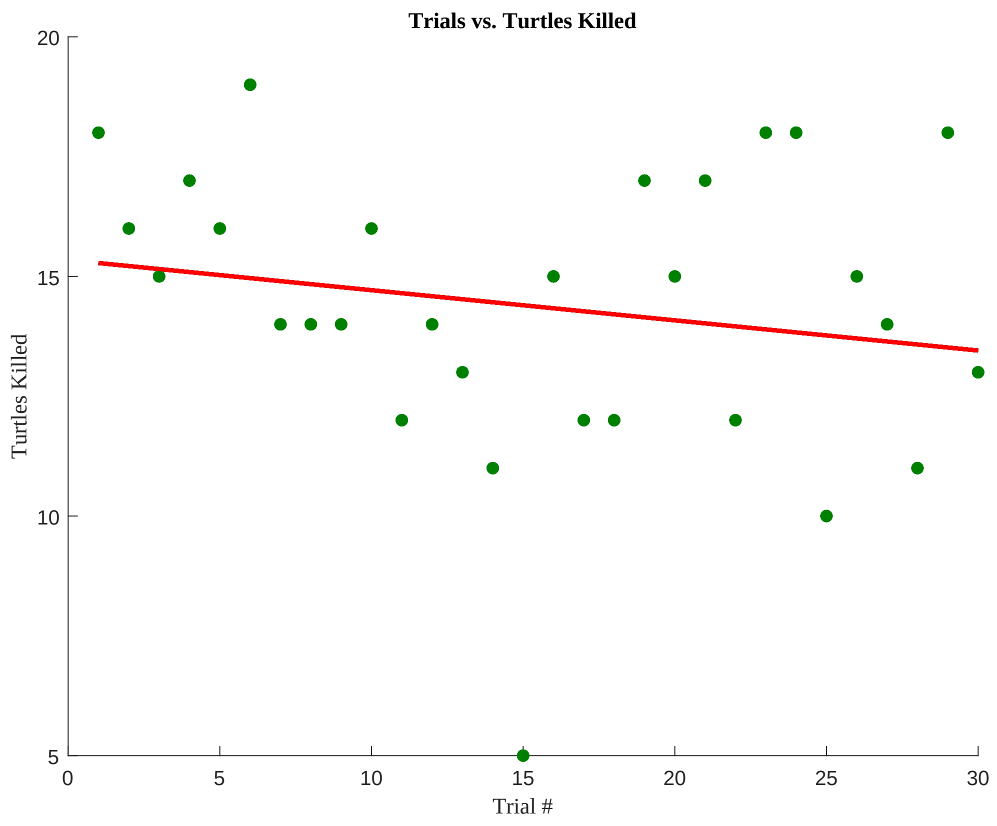

Figure B.1: Scatter plot showing the number of turtles killed across 30 simulation trials.

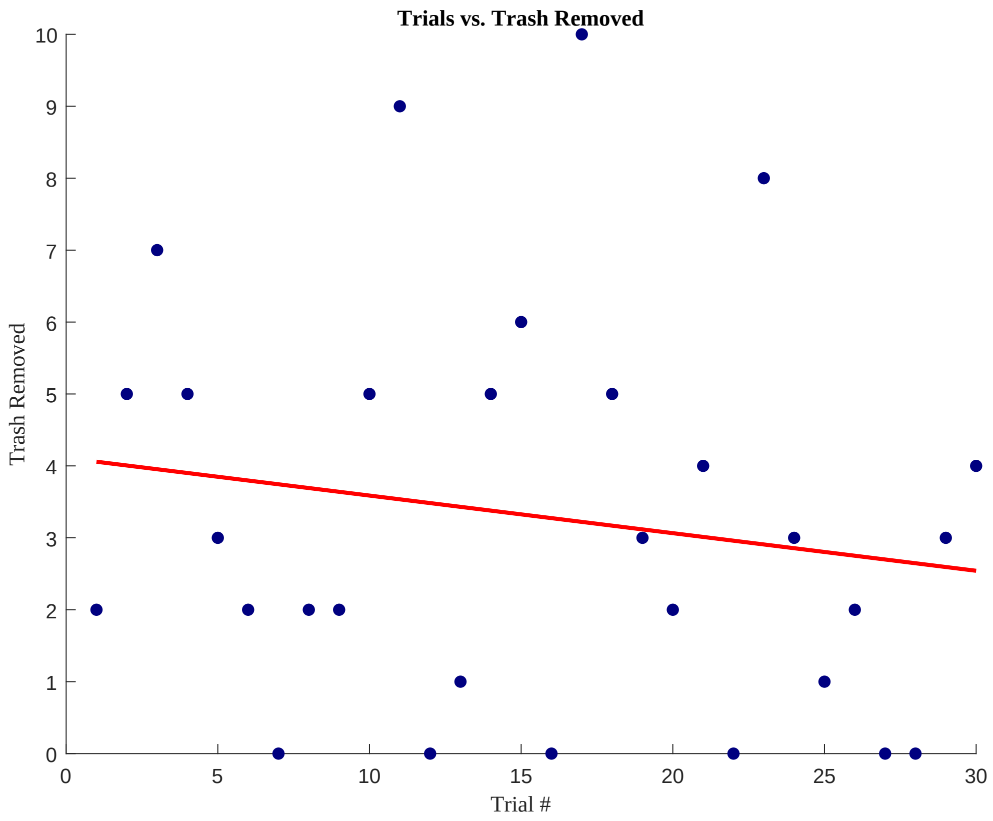

Figure B.2: Scatter plot showing the amount of trash removed across 30 simulation trials. A red trend line is included to illustrate the slight decline in trash removal over the course of the trials.