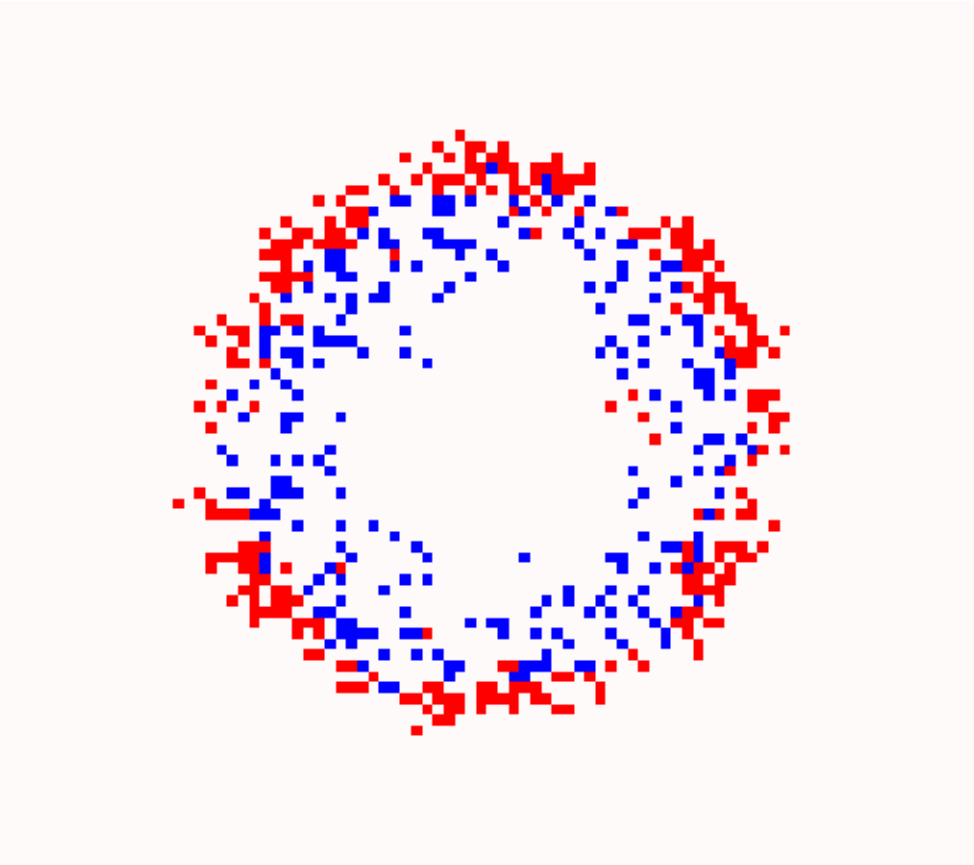
Study of Predator-Prey Dynamics

Vincent Edwards, Julia Corrales, and Rachel Gossard

April 13, 2025



Contents

1. Lotka-Volterra Model (LVM)

The Lotka-Volterra Model (LVM) models the dynamics between a predator species (y) and a prey species (x) over time (t).

$$\frac{dy}{dt} = -ay + bxy \quad \frac{dx}{dt} = +dx - cyx$$

The model depends on four parameters.

- a: decay rate of the predators
- b: proportionality for how predators grow due to eating prey
- c: proportionality for how prey decay due to being eaten by predators
- d: growth rate of the prey

Dividing the differential equations by each other yields $\frac{dy}{dx}$.

$$\frac{dy}{dx} = \frac{b}{c}\frac{y\left(x - \frac{a}{b}\right)}{x\left(\frac{d}{c} - y\right)}$$

Since this equation does not depend explicitly on time, it can be used to create phase portraits. The solutions swirl counter-clockwise around $x = \frac{a}{b}$ and $y = \frac{d}{c}$.

2. Types of Simulation Outcomes

The simulation has three main types of outcomes:

1. Both species go extinct
2. Sharks go extinct
3. Neither species goes extinct within steps simulated for

The first outcome occurs when the sharks eat all the fish. The surviving sharks, now left with no food, subsequently die out. The second outcome occurs when the sharks eat all the fish nearby, but some fish still remain further away. The sharks cannot reach those further fish before dying out. This leaves the fish population unchecked, allowing them to grow until they fill the board. The third outcome occurs when the simulation does not terminate early, as happens with the other two outcomes, and runs for the full amount of steps specified. The predator and prey populations oscillate in accord with the Lotka-Volterra model.

Note that the LVM only predicts that the first outcome will only occur if the prey population starts at 0, and the second outcome will only occur if the shark population starts at 0. But, due to the randomness of the simulation and its discrete nature, these outcomes still occur. In order to reduce the chances of those extinction outcomes, a relatively large board (`dims = (80, 90)`) was used. This allows for more fish and shark clusters to form, reducing the chances that the sharks eat *all* the food in the world. It also increases the chances that another fish cluster is nearby after the sharks finish devouring one. In addition, a larger board allows for larger populations and more creature interactions, helping the simulation be a closer approximation of continuous populations and time (the requirements of any differential model).

The graphs in Section 3 depict how common each of these outcomes are, at least for the sets of parameter tested. As discussed in the last paragraph, utilizing a large board greatly boosted the chance

of neither species going extinct. Certain parameter modifications increase the chances of extinction (at the expense of the chance of neither going extinct).

3. Conditions for Good Modeling

In order to test the conditions under which it appears the LVM models the simulation results well, tests were run plotting the measured chances of the three main types outcomes (Both Extinct, Sharks Extinct, & Neither Extinct) as each simulation parameter was varied one at a time. It was reasoned that by examining how the simulation responds to changes around a single point in the parameter space along different axes, any trends noticed could be applied more broadly to the simulation overall. Parameters that maximize the chance of neither species going extinct are good, as it suggests the simulation is following the trajectories of the LVM.

To start, the group played around with the simulation manually in order to find a set of control/default parameters that allowed the simulation to frequently run for 500 steps without terminating early. 500 steps seemed like a reasonable threshold to check if a simulation run was stable. The default parameters used are shown in the script below, along with some helper functions to grab the parameters needed to run different simulation functions.

default_parameters.py

```
parameters = {
    "breed_time": 3,
    "energy_gain": 4,
    "breed_energy": 15,
    "board_area": 7200,
    "aspect_ratio": 9/8,
    "initial_fish": 500,
    "initial_sharks": 400,
    "steps": 500,
    "start_energy": 9,
    "use_basic_setup": True,
}

def get_initialization_parameters(params):
    """
    Return a dictionary with just the parameters needed to initialize the game array.
    """
    result = {}
    desired_keys = ["initial_fish", "initial_sharks", "breed_time", "breed_energy"]
    for key in desired_keys:
        result[key] = params[key]
    return result

def get_simulation_parameters(params):
    """
    Return a dictionary with just the parameters needed to run the simulation.
    """
    result = {}
    desired_keys = ["steps", "breed_time", "energy_gain", "breed_energy",
"start_energy"]
```

```
    for key in desired_keys:
        result[key] = params[key]
    return result
```

Each graph depicts a single parameter varied along a range of values. For each value of the target parameter, 25 trials were run. After which, the probabilities for each outcome were calculated. The functions used to test the outcome chances and create a plot are shown below. Note that each of these plots took about 30–60 minutes to *bake*.

measure_outcome_chances.py

```
import wa_tor
import default_parameters
import matplotlib.pyplot as plt

# Specify the test values to use when testing each parameter
test_ranges = {
    "breed_time": range(1, 15 + 1),
    "energy_gain": range(2, 18 + 1),
    "breed_energy": range(default_parameters.parameters["start_energy"] + 1, 25 + 1),
    "board_area": range(3200, 9600 + 400, 400),
    "aspect_ratio": [i/8 for i in range(8, 16 + 1)],
    "initial_fish": range(200, 1000 + 50, 50),
    "initial_sharks": range(200, 1000 + 50, 50),
    "start_energy": range(1, default_parameters.parameters["breed_energy"] - 1),
}

def test_outcome_chances(target_param, test_values, trials, params=None):
    """
    Vary the target parameter to have the given test values.
    For each value, run the simulation for the specified number of trials and
calculate the chance of each of the possible outcomes.
    - Everything went extinct
    - Fish fill the board
    - Simulation could keep going

    Return a dictionary containing three lists of chances, one list for each outcome.
    Each list contains the chances found using each test value of the target
parameter.
    """
    # Set the parameters to the default if not specified
    if params is None:
        params = default_parameters.parameters.copy()

    overall_chances = {
        "everything_extinct": [],
        "fish_fill_board": [],
        "still_going": [],
    }

    for value in test_values:
```

```
        # Set the target parameter
        params[target_param] = value

        # Calculate the board dimensions based on board area and aspect ratio
        # h*w = Area; h*Ratio = w
        # h**2 * Ratio = Area
        h = int((params["board_area"] / params["aspect_ratio"])**0.5)
        w = int(h * params["aspect_ratio"])
        dims = (h, w)

        # Extract the needed parameters for later steps
        init_params = default_parameters.get_initialization_parameters(params)
        sim_params = default_parameters.get_simulation_parameters(params)
        # Keep track of the counts for the possible outcomes
        everything_extinct_count = 0
        fish_fill_count = 0

        for _ in range(trials):
            # Initialize the game array
            initial_game_array = wa_tor.create_empty_game_array(dims)
            if params["use_basic_setup"]:
                wa_tor.initialize_game_array_randomly(initial_game_array,
**init_params)
            else:
                wa_tor.initialize_game_array_circular(initial_game_array,
**init_params)

            # Run the simulation
            fish_counts, shark_counts =
wa_tor.run_simulation_minimal(initial_game_array, **sim_params)

            # Check whether fish filled the board or if sharks and fish both went
extinct
            # Update the counts for these events
            size = dims[0] * dims[1]
            if fish_counts[-1] + shark_counts[-1] <= 0:
                everything_extinct_count += 1
            elif fish_counts[-1] == size:
                fish_fill_count += 1

        # Store the chances of each possible outcome
        still_going_count = trials - everything_extinct_count - fish_fill_count
        overall_chances["everything_extinct"].append(everything_extinct_count /
trials)
        overall_chances["fish_fill_board"].append(fish_fill_count / trials)
        overall_chances["still_going"].append(still_going_count / trials)

    return overall_chances

def plot_and_test_outcome_chances(fname, target_param, test_values, trials,
params=None):
```

```
    """
    Run the function test_outcome_chances() with the given arguments, then plot the
results.
    Save the figure at the given file name.
    """
    outcome_chances = test_outcome_chances(target_param, test_values, trials, params)

    fig, ax = plt.subplots()
    ax.plot(test_values, outcome_chances["everything_extinct"], "o", label="Both
Extinct")
    ax.plot(test_values, outcome_chances["fish_fill_board"], "^", label="Sharks
Extinct")
    ax.plot(test_values, outcome_chances["still_going"], ".", label="Neither Extinct")
    ax.set(xlabel=target_param, ylabel="Chance")
    ax.legend()
    fig.tight_layout()
    fig.savefig(fname)

def run_standard_test(target_parameter, use_basic_setup):
    """
    Run a standard test on the target parameter.
    Perform 25 trials with use_basic_setup optionally toggled.
    """
    trials = 25
    test_values = test_ranges[target_parameter]
    params = default_parameters.parameters.copy()
    params["use_basic_setup"] = use_basic_setup

    if use_basic_setup:
        fname = f"media/outcome_chances_{target_parameter}.svg"
    else:
        fname = f"media/outcome_chances_{target_parameter}_circular.svg"

    plot_and_test_outcome_chances(fname, target_parameter, test_values, trials,
params)
```

3.a. `breed_time`

```
import measure_outcome_chances as tst

tst.run_standard_test("breed_time", True)
```

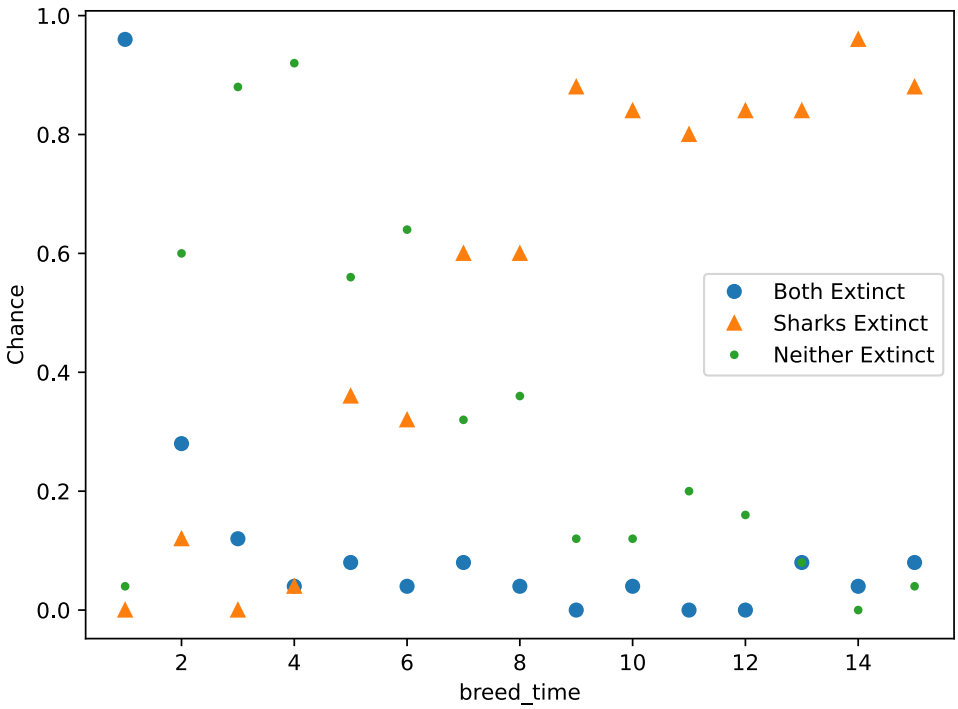


Figure 1: Outcome Chances vs `breed_time`

Figure 1 displays the outcome chances relative to the value of `breed_time` for the fish. When the `breed_time` of the fish increases, the chances of the sharks going extinct also increases. Since the fish take more steps before they can reproduce, their populations tend to grow more slowly, leaving the sharks with fewer sources of energy. On the other hand, when the `breed_time` is too low, the fish clusters grow too quickly and start to merge together, making it more common for the sharks to consume all the fish on the board. Thus, the LVM models the results better when the `breed_time` is not too high and not too low.

3.b. `energy_gain`

```
import measure_outcome_chances as tst

tst.run_standard_test("energy_gain", True)
```

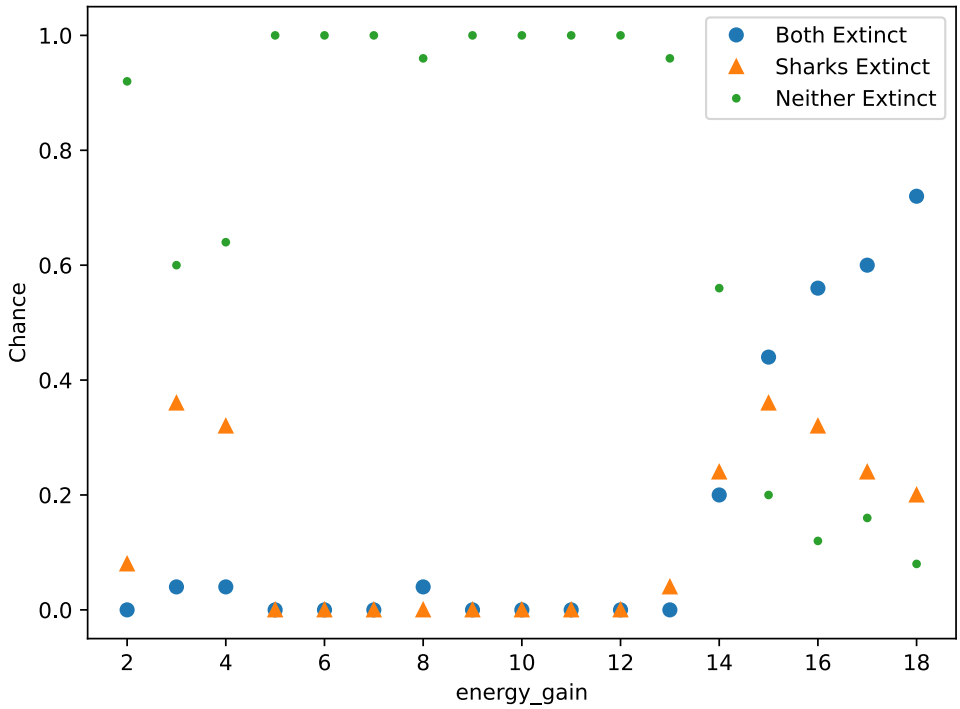


Figure 2: Outcome Chances vs `energy_gain`

Figure 2 displays the outcome chances relative to the value of `energy_gain` for the sharks. When the sharks have a larger `energy_gain`, it allows their population to grow at a greater rate since they gain more energy from consuming fish. Initial increases in `energy_gain` prevent the sharks from going extinct, as it helps them have more offspring exploring the board, discovering fish clusters, and keeping their species alive. But if the `energy_gain` gets too high, the sharks start consuming too many fish clusters, making it more likely that sharks consume most or all of the fish clusters and subsequently die out. Thus, the LVM models the results better when the `energy_gain` is not too low and not too high. For the default parameters used, the best `energy_gain` range is about 5–13.

Note that the `energy_gain` had to be kept above a value of 1. Since the sharks deplete 1 energy each time step, when `energy_gain = 1` it becomes impossible for the sharks to gain any energy overall and no new sharks can be born. Since the shark population cannot grow under these conditions, these trajectories definitely do not follow the LVM.

3.c. `breed_energy`

```
import measure_outcome_chances as tst

tst.run_standard_test("breed_energy", True)
```

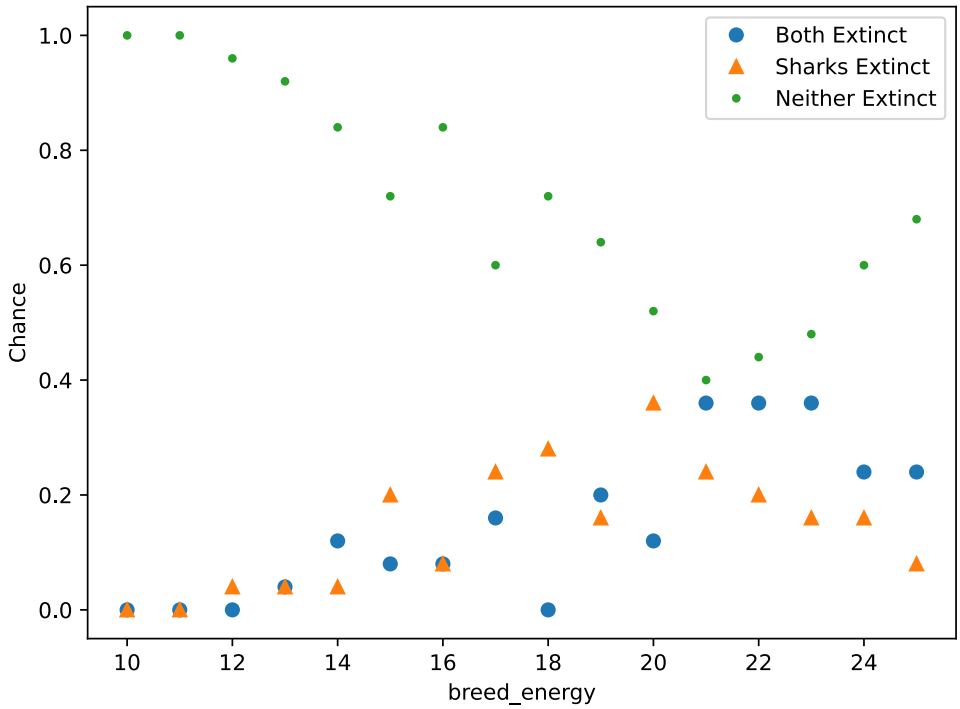


Figure 3: Outcome Chances vs `breed_energy`

Figure 3 displays the outcome chances relative to the value of `breed_energy`. Note that the `breed_energy` had to be kept higher than the the `start_energy`. When `breed_energy` is less than or equal to `start_energy`, it can lead to cases where a shark dies or turns into a fish after giving birth. This is due to the way creatures are stored in the `game_array`, using positive values for fish, negative values for sharks, and zero for unoccupied spaces. Such behavior is definitely not desired.

As `breed_energy` increases, sharks need to eat more fish in order to have children. As a result, sharks tend to hold onto more energy, swim around for longer, and have fewer offspring. This makes sharks more dangerous and better able to hunt down fish clusters. Thus, as `breed_energy` increases, the chances increase for sharks going extinct and for both species going extinct. Whether these hardier sharks successfully take out all the fish clusters or leave a distant one remaining becomes almost a coin flip. Therefore, the LVM models the results better when the `breed_energy` is not too high. For the default parameters used, the best `breed_energy` range is about 10–14.

3.d. `board_area`

```
import measure_outcome_chances as tst

tst.run_standard_test("board_area", True)
```

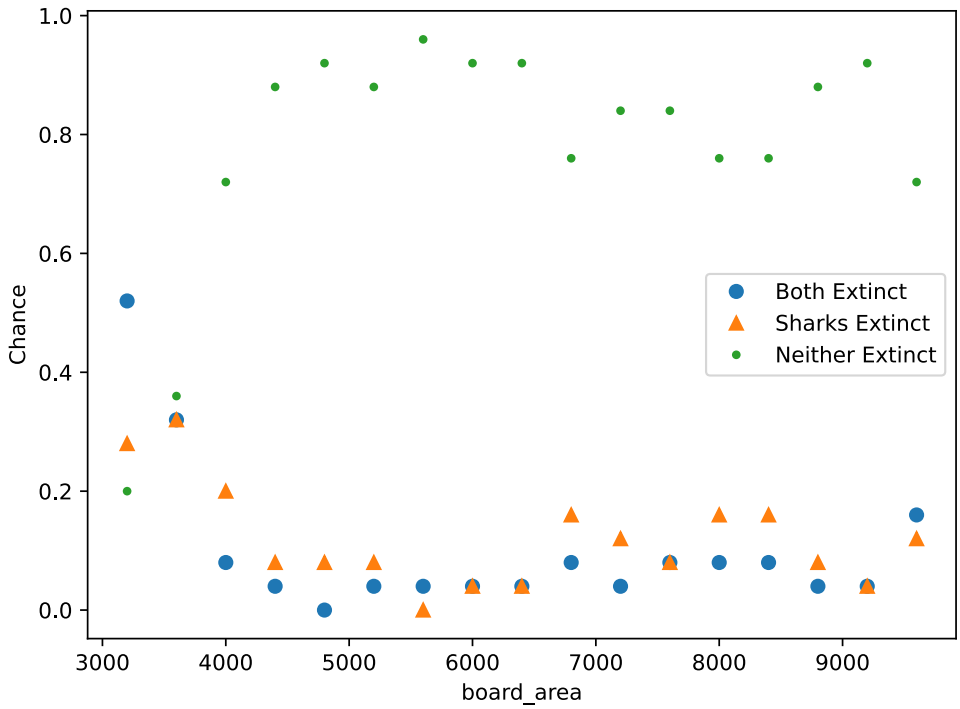


Figure 4: Outcome Chances vs `board_area`

Figure 4 displays the outcome chances as the `board_area` of the `game_array` is increased while the `aspect_ratio` is kept constant. When the `board_area` is too small, the creatures are packed more tightly and fewer separate fish clusters form. This makes it more common for the sharks to successfully eat all the fish cluster and leave none remaining, which makes both species go extinct. When the `board_area` gets too large, the creatures on the board get more spread out. This forces the sharks to travel further distances on average to find fish to eat, making it more likely that the sharks take out all fish or pass away before finding a new fish cluster. Thus, too small or too large a `board_area` leads to higher chances of the extinction. In other words, the LVM models the results better when the `board_area` is not too low and not too high. For the default parameters used, this balanced `board_area` range is about 4400–6400.

3.e. `aspect_ratio`

```
import measure_outcome_chances as tst

tst.run_standard_test("aspect_ratio", True)
```

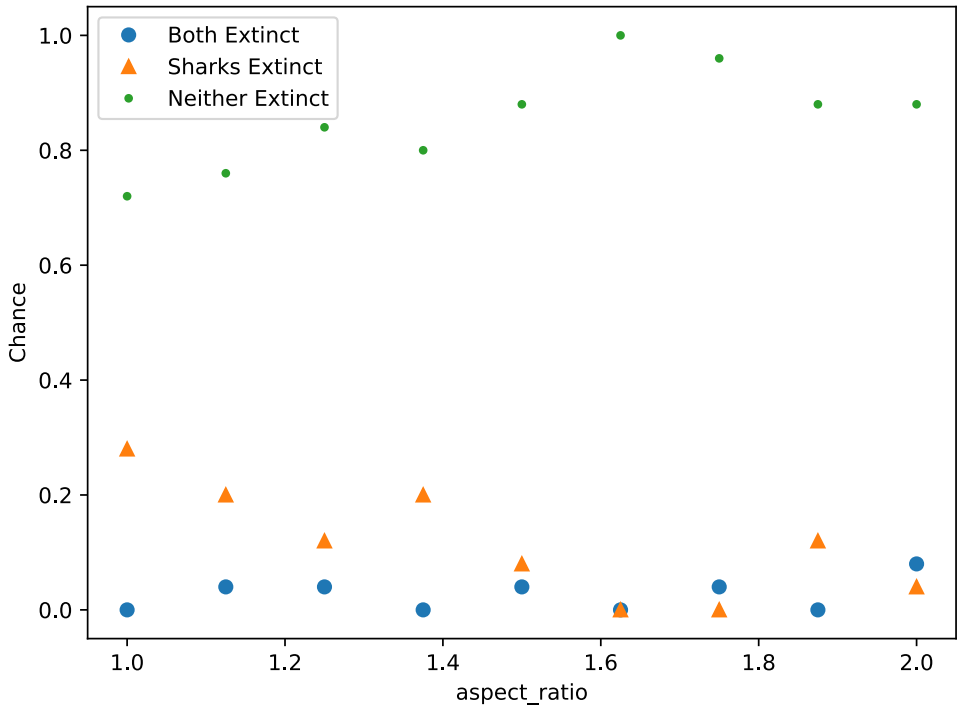


Figure 5: Outcome Chances vs `aspect_ratio`

Figure 5 displays the outcome chances as the `aspect_ratio` of the `game_array` is changed while the `board_area` is kept constant. A square board has an `aspect_ratio = 1` and is symmetric. Increasing the `aspect_ratio` above 1 adds asymmetry to the board, making it take more steps to travel around one side compared to the other. Higher `aspect_ratio` values lead to lower extinction chances. The difference in distance to travel horizontally vs vertically around the board seems to desynchronize when shark fronts hunting along different axes collide, which helps prevent the fish population from getting too low. Thus, the LVM models the result better when the `aspect_ratio` is higher.

3.f. `initial_fish`

```
import measure_outcome_chances as tst

tst.run_standard_test("initial_fish", True)
```

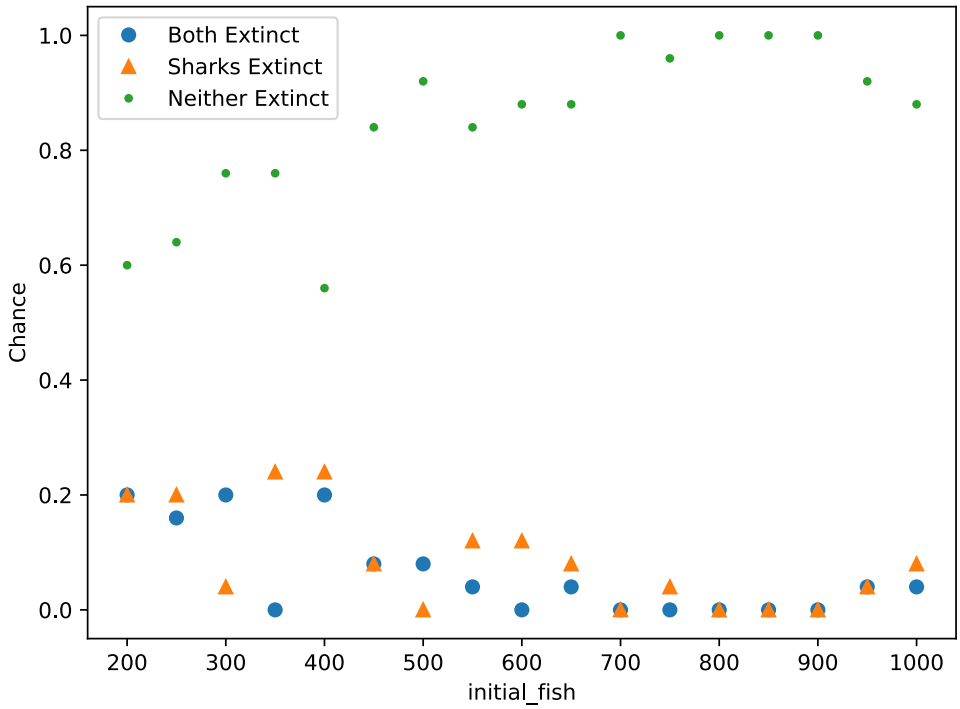


Figure 6: Outcome Chances vs `initial_fish`

Figure 6 displays the outcome chances relative to the `initial_fish` value. As the `initial_fish` value increases, more fish get scattered across the board with the potential to start clusters. At first, increasing `initial_fish` reduces the chances of extinct. With more food available, the sharks can more easily jump between fish clusters without the worry of being too far away from the next one or eating all of them. But once `initial_fish` gets too high, the separate clusters start to merge together and the sharks consume larger clusters at a time, leaving the sharks more susceptible to leaving too few fish alive. This reflects that starting with very low or high amounts of fish sets the populations on the outer streams predicted by the LVM. Since these trajectories go fairly close to $x = 0$ & $y = 0$, it makes it more likely that the populations randomly reach zero and cause extinction. Thus, the LVM models the results better when the `initial_fish` is not too low and not too high. For the default parameters used, the best `initial_fish` range is about 700–900.

3.g. `initial_sharks`

```
import measure_outcome_chances as tst

tst.run_standard_test("initial_sharks", True)
```

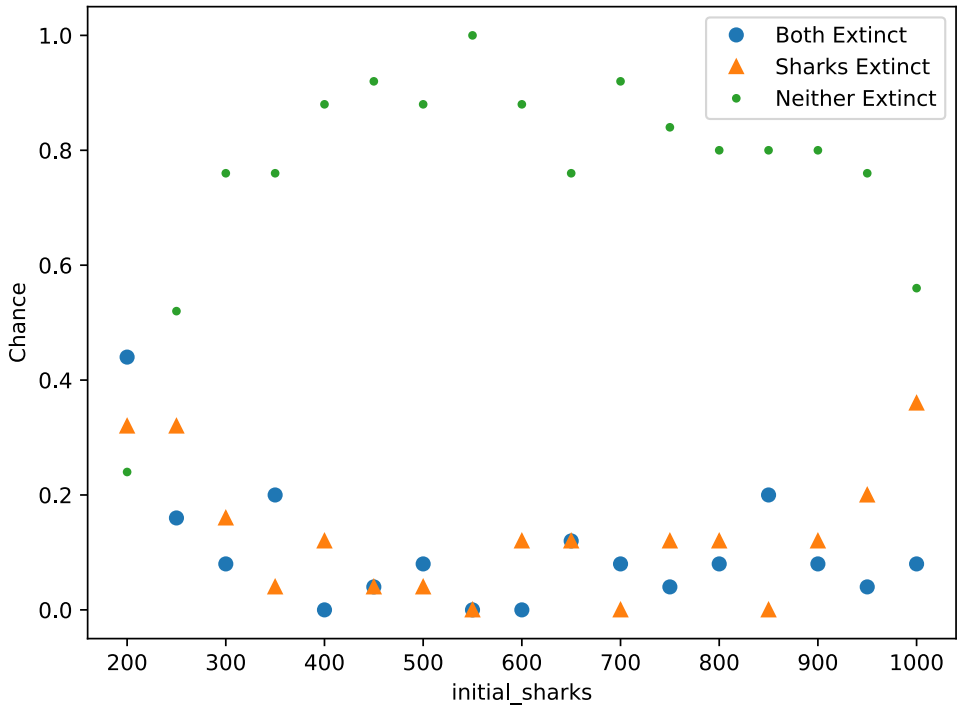


Figure 7: Outcome Chances vs `initial_sharks`

Figure 7 shows the outcome chances relative to the `initial_shark` value. As the `initial_sharks` value increases, more sharks get scattered across the board. At first increasing `initial_sharks` reduces the chance of sharks going extinct, but once `initial_sharks` gets too high the extinction chance starts to increase again. This reflects that starting with very low or high amounts of sharsk sets the populations on the outer streams predicted by the LVM. Since these trajectories go fairly close to $x = 0$ & $y = 0$, it makes it more likely that the populations randomly reach zero and cause extinction. Thus, the LVM models the results better when the `initial_sharks` is not too low and not too high. For the default parameters used, the best `initial_sharks` range is about 400–900.

3.h. `start_energy`

```
import measure_outcome_chances as tst

tst.run_standard_test("start_energy", True)
```

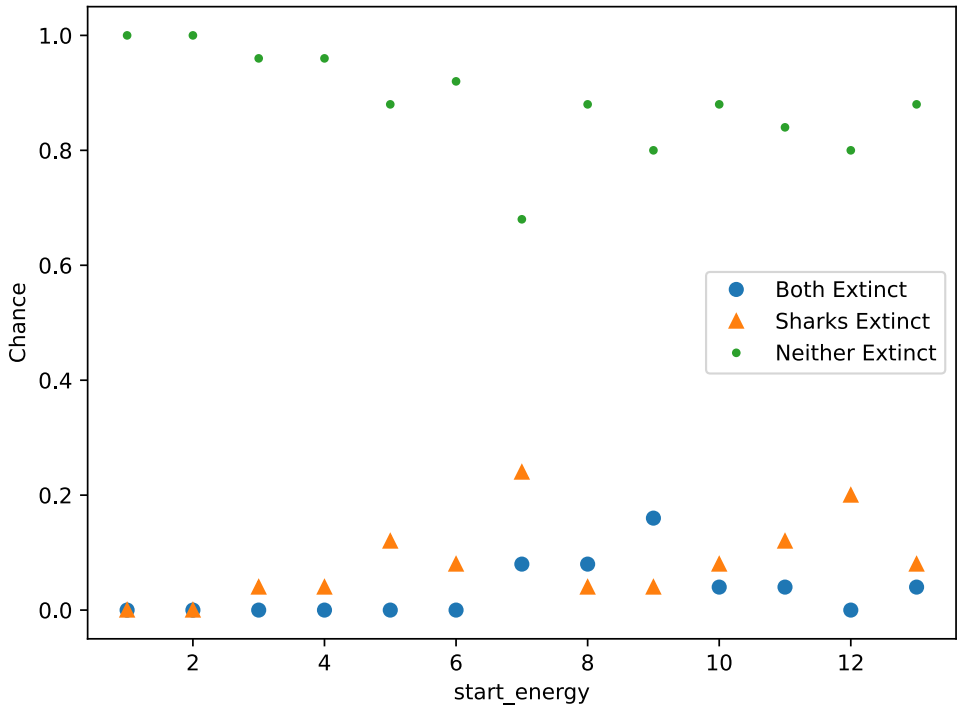


Figure 8: Outcome Chances vs `start_energy`

Figure 8 displays the outcome chances relative to the `start_energy` of child sharks. Note that the `start_energy` had to be kept lower than the the `breed_energy`. When `start_energy` is greater than or equal to `breed_energy`, it can lead to cases where a shark dies or turns into a fish after giving birth. This is due to the way creatures are stored in the `game_array`, using positive values for fish, negative values for sharks, and zero for unoccupied spaces. Such behavior is definitely not desired.

Lower `start_energy` values correspond to lower extinction chances. Since the `start_energy` for a newborn shark is taken from its parent, this suggests that it is better to leave most of the energy with the parent shark. If a parent shark just had a child, it likely just ate a fish. Due to the way sharks tend to form fronts that hunt down fish clusters, this puts the parent in a good position to eat another fish while the child gets put in a bad position with very few fish. This helps the parent shark build up more energy, making it *stronger* and better able to survive long enough to reach new fish clusters. In contrast, the child gets put in a position where it is likely to die, even if it had more `start_energy`. In other words, increasing `start_energy` hurts the parent more than it helps the child. As a result, higher `start_energy` values lead to higher extinction chances. Thus, the LVM models the results better when the `start_energy` is lower.

4. Main Simulation Parameters

The three main simulation parameters are `breed_time`, `energy_gain`, and `breed_energy`. These should relate in some way to the four LVM parameters a, b, c, and d. While it is difficult to measure the LVM parameters directly, it is easier to measure the ratios $\frac{a}{b}$ and $\frac{d}{c}$. When x is at a local maximum, $y = \frac{d}{c}$. When y is at a local maximum, $x = \frac{a}{b}$. In order to determine where those maxima are located, the population data was chunked into sections. A section starts once population values go above some fraction of the population range, and it ends when the values go below a lower fraction of the range. This method was chosen to deal with the random noise in the population data, avoiding falsely marking any small spikes as maxima.

Each graph depicts a single parameter varied along a range of values. For each value of the target parameter, 25 trials were run. The LVM ratios from each trial were averaged to get the values graphed. In order to determine the ratios from a single trial's data, the critical points ($x_{\text{crit}} = \frac{a}{b}, y_{\text{crit}} = \frac{d}{c}$) were found using the method for locating local extrema described above, then the critical values were averaged to get the ratios. Note that each of these plots took about 30–60 minutes to *bake*.

measure_ratios.py

```
import wa_tor
import default_parameters
import numpy as np
import matplotlib.pyplot as plt

# Specify the test values to use when testing each parameter
test_ranges = {
    "breed_time": range(1, 15 + 1),
    "energy_gain": range(2, 18 + 1),
    "breed_energy": range(default_parameters.parameters["start_energy"] + 1, 25 + 1),
}

def find_local_maxima(x_values, y_values):
    """
    Return the x values where y is at a local maximum.
    To deal with noise, chunk the data into sections.
    A section starts once values go above some threshold of the range, and it ends
when values go below another threshold of the range.
    Return the list of critical x values.
    """
    y_range = np.ptp(y_values)
    x_crit_list = []

    y_chunk_start = y_range / 4
    y_chunk_end = y_range / 5
    inside_y_chunk = False
    y_max = 0
    x_crit = 0

    # Walk through each (x, y) pair
    for x, y in zip(x_values, y_values):
        # Check if we are inside a y chunk
```

```
        if inside_y_chunk:
            # If we are still above the end threshold, keep going
            if y > y_chunk_end:
                # Check if this y value is the biggest we have seen so far
                if y > y_max:
                    y_max = y
                    x_crit = x
            # Otherwise, we need to save the critical value and leave the chunk
            else:
                x_crit_list.append(x_crit)
                inside_y_chunk = False
                # Reset the max y value
                y_max = 0
                x_crit = 0
        # Otherwise, check if we have entered a y chunk
        elif y > y_chunk_start:
            inside_y_chunk = True

    return x_crit_list

def calculate_critical_points(fish_counts, shark_counts):
    """
    Calculate the critical points (x = a/b, y = d/c) given lists fish and shark
counts.
    When x is at a local maximum, y = d/c.
    When y is at a local maximum, x = a/b.
    To estimate the ratios, average the x or y values found at each local maxima.
    Return the estimates for (a/b, d/c).
    """
    x_crit_list = find_local_maxima(fish_counts, shark_counts)
    y_crit_list = find_local_maxima(shark_counts, fish_counts)

    return np.mean(x_crit_list), np.mean(y_crit_list)

def test_lvm_ratios(target_param, test_values, trials, params=None):
    """
    Vary the target parameter to have the given test values.
    For each value, run the simulation for the specified number of trials and
calculate critical points (x = a/b & y = d/c) of the Lotka-Volterra model.

    Return a dictionary containing two lists of ratios, one list for a/b and another
for d/c.
    Each list contains the ratios found using each test value of the target parameter.
    """
    # Set the parameters to the default if not specified
    if params is None:
        params = default_parameters.parameters.copy()

    overall_ratios = {
        "a/b": [],
        "d/c": [],
```

```
    }

    for value in test_values:
        # Set the target parameter
        params[target_param] = value

        # Calculate the board dimensions based on board area and aspect ratio
        # h*w = Area; h*Ratio = w
        # h**2 * Ratio = Area
        h = int((params["board_area"] / params["aspect_ratio"])**0.5)
        w = int(h * params["aspect_ratio"])
        dims = (h, w)

        # Extract the needed parameters for later steps
        init_params = default_parameters.get_initialization_parameters(params)
        sim_params = default_parameters.get_simulation_parameters(params)
        # Keep track of the ratios found in each trial
        a_b_ratios = []
        d_c_ratios = []

        for _ in range(trials):
            # Initialize the game array
            initial_game_array = wa_tor.create_empty_game_array(dims)
            if params["use_basic_setup"]:
                wa_tor.initialize_game_array_randomly(initial_game_array,
**init_params)
            else:
                wa_tor.initialize_game_array_circular(initial_game_array,
**init_params)

            # Run the simulation
            fish_counts, shark_counts =
wa_tor.run_simulation_minimal(initial_game_array, **sim_params)

            # Calculate the critical points
            a_b, d_c = calculate_critical_points(fish_counts, shark_counts)
            a_b_ratios.append(a_b)
            d_c_ratios.append(d_c)

        # Store the average ratios found in the trials
        overall_ratios["a/b"].append(np.nanmean(a_b_ratios))
        overall_ratios["d/c"].append(np.nanmean(d_c_ratios))

    return overall_ratios

def plot_and_test_lvm_ratios(fname, target_param, test_values, trials, params=None):
    """
    Run the function test_outcome_chances() with the given arguments, then plot the
results.
    Save the figure at the given file name.
    """
```

```
    lvm_ratios = test_lvm_ratios(target_param, test_values, trials, params)

    fig, axes = plt.subplots(1, 2, figsize=(12.8, 4.8))

    axes[0].plot(test_values, lvm_ratios["a/b"], "o")
    axes[0].set(xlabel=target_param, ylabel="a/b")

    axes[1].plot(test_values, lvm_ratios["d/c"], "o")
    axes[1].set(xlabel=target_param, ylabel="d/c")

    fig.tight_layout()
    fig.savefig(fname)

def run_standard_test(target_parameter, use_basic_setup):
    """
    Run a standard test on the target parameter.
    Perform 25 trials with use_basic_setup optionally toggled.
    """
    trials = 25
    test_values = test_ranges[target_parameter]
    params = default_parameters.parameters.copy()
    params["use_basic_setup"] = use_basic_setup

    if use_basic_setup:
        fname = f"media/lvm_ratios_{target_parameter}.svg"
    else:
        fname = f"media/lvm_ratios_{target_parameter}_circular.svg"

    plot_and_test_lvm_ratios(fname, target_parameter, test_values, trials, params)
```

4.a. `breed_time`

```
import measure_ratios as tst

tst.run_standard_test("breed_time", True)
```

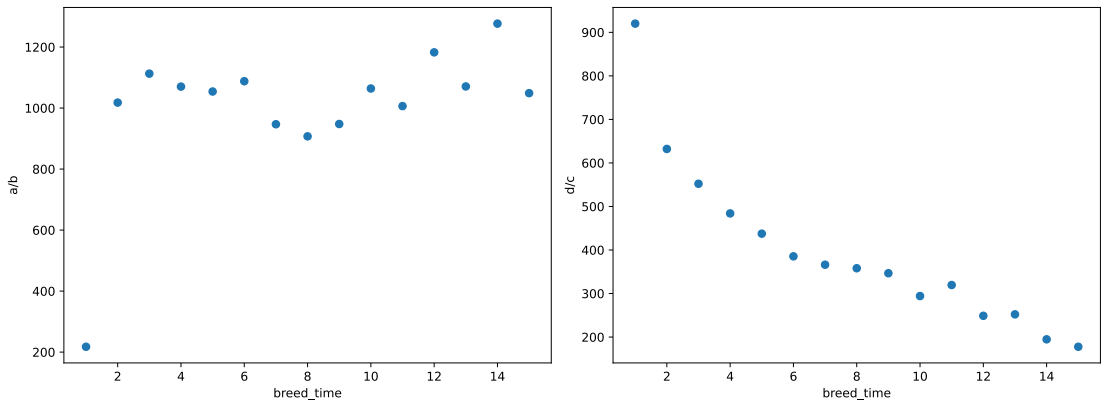


Figure 9: LVM Ratios vs `breed_time`

Figure 9 plots the LVM ratios as `breed_time` is varied. As `breed_time` increases, fish need to take more steps before they have children. Since this a property of how rapidly the fish grow, it is not expected to only impact d and not a, b, and c This is reflected in the plot of $\frac{a}{b}$, as the ratio remains roughly constant as `breed_time` changes. As `breed_time` gets larger, the fish reproduce more slowly d is expected to decrease. This is depicted in the graph of $\frac{d}{c}$, which shows the ratio decreasing as `breed_time` increases.

As shown in Figure 1, both species very often go extinct when `energy_gain = 1`. Since the simulation does not reflect the LVM under such conditions, it seems reasonable to ignore the sudden decrease in $\frac{a}{b}$ when `energy_gain = 1`.

4.b. `energy_gain`

```
import measure_ratios as tst

tst.run_standard_test("energy_gain", True)
```

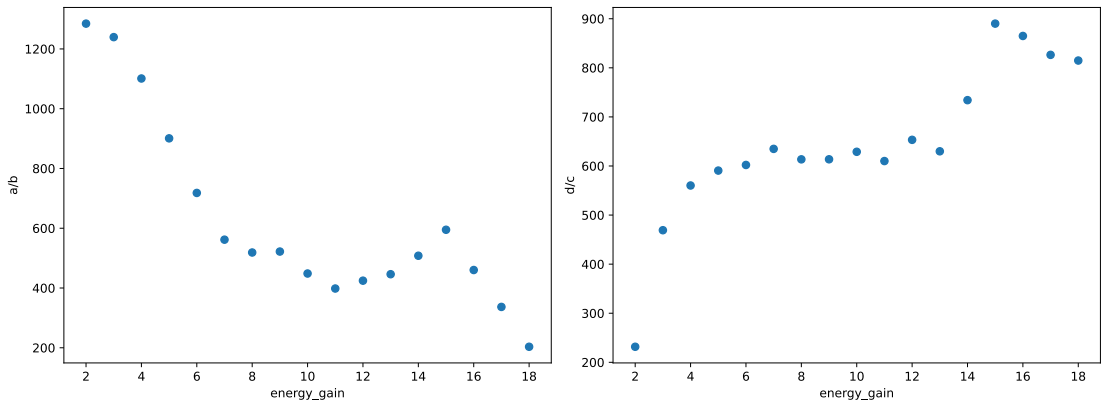


Figure 10: LVM Ratios vs `energy_gain`

Figure 10 plots the LVM ratios as `energy_gain` is varied. As noted when discussing Figure 2, the LVM models the situation best when `energy_gain` is kept within 5–13. Thus, the trend in that portion of the graphs should be focused on. Since `energy_gain` controls how much of an energy boost sharks get for eating fish, as it increases b should increase too, helping the shark population grow more quickly. a, c, and d should not be impacted, since they have to do with different factors. Thus, $\frac{d}{c}$ should remain constant and $\frac{a}{b}$ should decrease, both of which are depicted in the LVM ratio graphs.

4.c. `breed_energy`

```
import measure_ratios as tst

tst.run_standard_test("breed_energy", True)
```

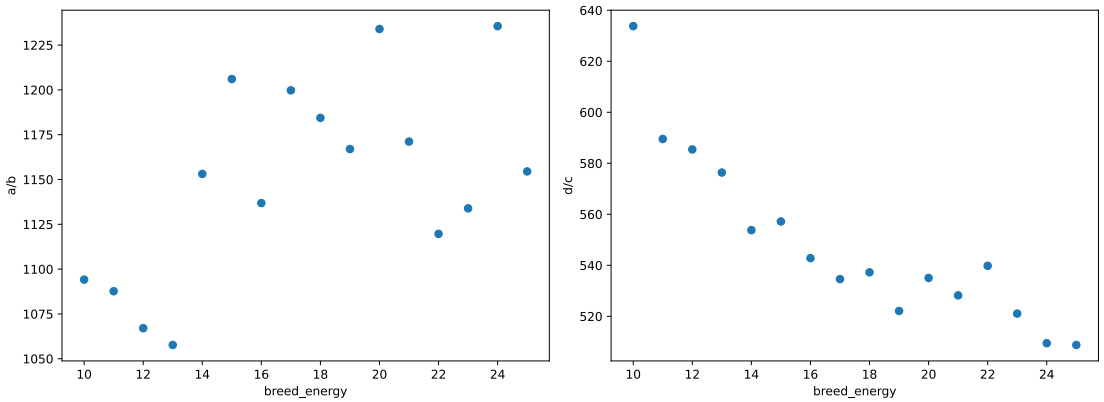


Figure 11: LVM Ratios vs `breed_energy`

Figure 11 plots the LVM ratios as `breed_energy` is varied. As `breed_energy` increases, sharks need to eat more fish in order to have children. As a result, sharks can build up larger amounts of energy, allowing them to travel further distances. As `breed_energy` increases, b is expected to decrease since each fish eaten causes a smaller boost in the shark population. In addition, higher `breed_energy` boosts shark longevity, thereby decreasing a. The loss of 1 energy per move is less significant compared to the increased average shark energy. If a and b decrease by similar amounts, that could cause their ratio to remain roughly the same, as depicted in the plot of $\frac{a}{b}$ in Figure 11.

While `breed_energy` should have no impact on d, as that is a property of the fish exclusively, it could impact c. As mentioned earlier, when `breed_energy` increases, sharks tend to hold onto more energy and swim around for longer. This makes sharks more dangerous and better able to hunt down fish clusters, thereby increasing c If c increases while d stays the same, that causes their ratio to decrease, as depicted in the plot of $\frac{d}{c}$ in Figure 11.

5. Circular Initialization

In the previous runs, the fish and sharks were initialized in the `game_array` at random locations. In simulations that use circular initialization, the predators and prey are initially clustered in a circular group in the center of the `game_array`. The sharks form a central disk, with the fish creating a ring around them.

This setup leads to different dynamics. Due to the fact that the sharks and fish start close to each other, the sharks encounter the fish immediately and a chase outward to the edges of the board ensues. As the diameter widens and the sharks get spread more thinly, some fish sneak into the center. What follows is a more consistent interaction between the species. With random initialization, most of the fish clusters grow and shrink in phase with each other. In contrast, when using circular initialization the fish clusters tend to grow and shrink out of phase with each other, making it easier for both species to maintain steady levels and avoid going extinct. This is reflected in the pattern that the chance of neither species going extinct went up for most points in all graphs when compared to their random initialization values.

5.a. `breed_time`

```
import measure_outcome_chances as tst

tst.run_standard_test("breed_time", True)
```

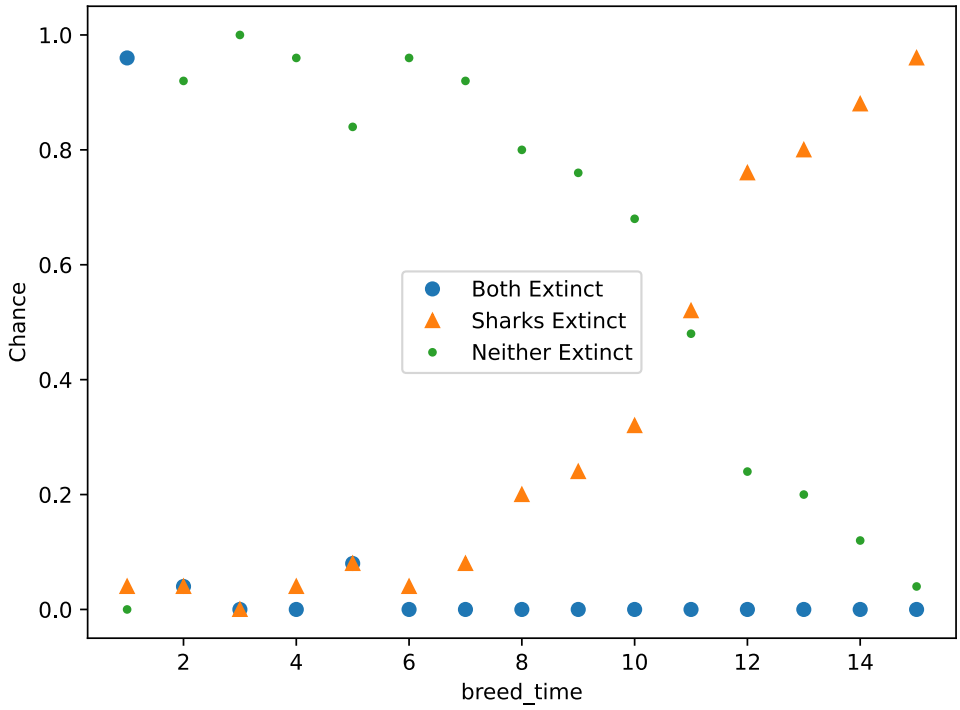


Figure 12: Outcome Chances vs `breed_time` (Circular Initialization)

Figure 12 displays the outcome chances relative to the value of `breed_time` for the fish when circular initialization is used. When compared to Figure 1 with random initialization, the overall shapes are similar. When the `breed_time` increases, the chances of the sharks going extinct also increases. But

with circular initialization, the chance of neither species going extinct is noticeably higher and decreases more slowly.

```
import measure_ratios as tst

tst.run_standard_test("breed_time", False)
```

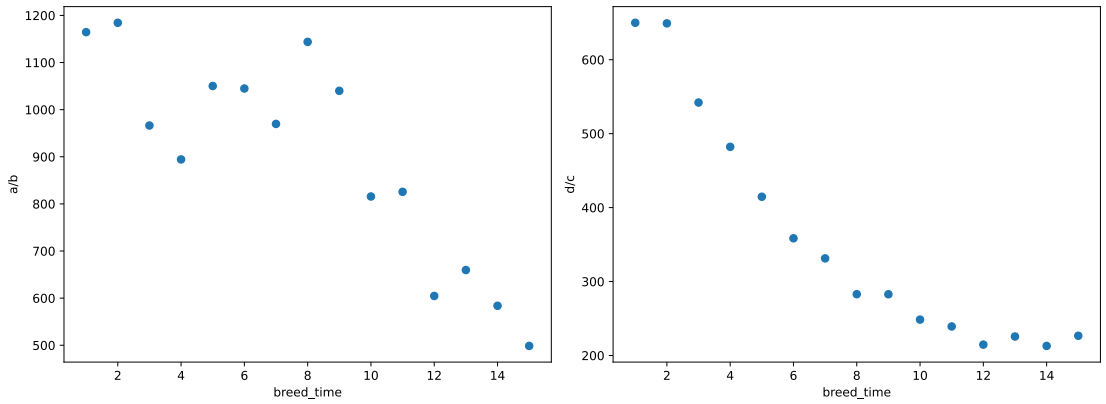


Figure 13: LVM Ratios vs `breed_time` (Circular Initialization)

Figure 13 plots the LVM ratios as `breed_time` is varied when circular initialization is used. When compared to Figure 9 with random initialization, the graph of $\frac{d}{c}$ decreases in both. If the graph of $\frac{a}{b}$ for circular initialization is focused on in the region where the neither extinct chance is high, 2–7, then it remains roughly constant as in the graph for random initialization.

5.b. `energy_gain`

```
import measure_outcome_chances as tst

tst.run_standard_test("energy_gain", True)
```

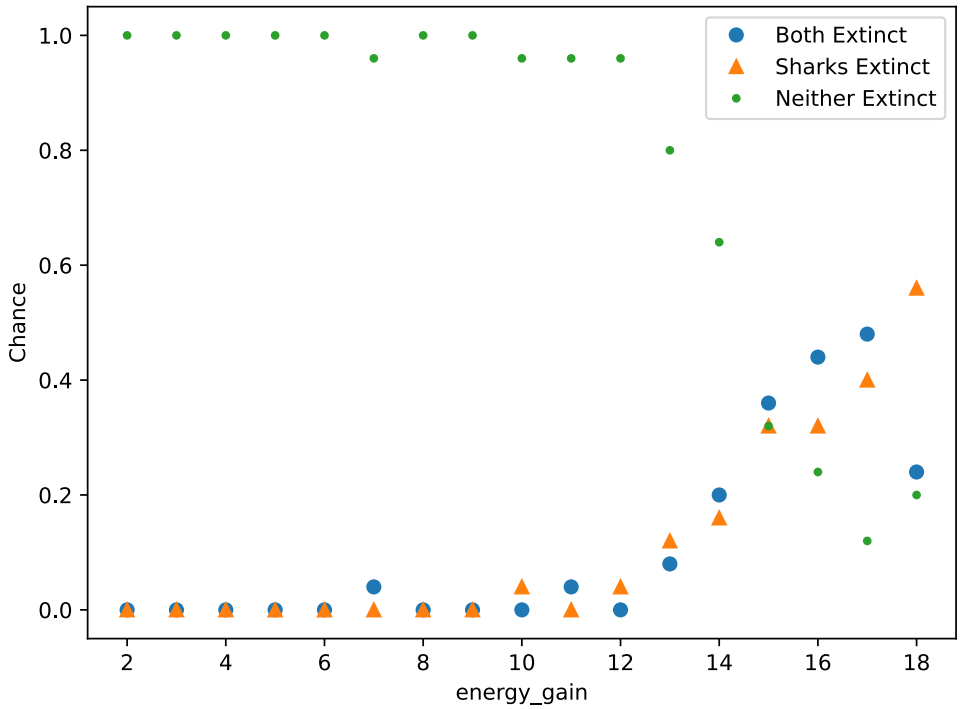


Figure 14: Outcome Chances vs `energy_gain` (Circular Initialization)

Figure 14 displays the outcome chances relative to the value of `energy_gain` for the sharks when circular initialization is used. When compared to Figure 2 with random initialization, the overall shapes are similar. When `energy_gain` gets too large, the extinction chances start to increase. But with circular initialization, smaller `energy_gain` values can be supported without causing extinction. The circular initialization helps keep the sharks closer to the fish, which allows the simulation to keep going even when sharks do not get as much of an energy boost from eating fish.

```
import measure_ratios as tst

tst.run_standard_test("energy_gain", False)
```

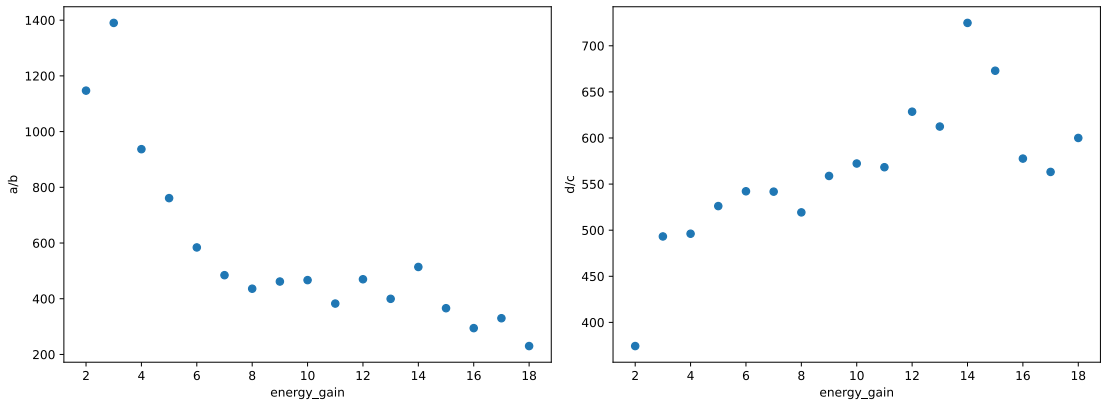


Figure 15: LVM Ratios vs `energy_gain` (Circular Initialization)

Figure 15 plots the LVM ratios as `energy_gain` is varied when circular initialization is used. When compared to Figure 10 with random initialization, the overall shapes are very similar. Thus, circular initialization does not noticeably change how `energy_gain` impacts the LVM ratios.

5.c. `breed_energy`

```
import measure_outcome_chances as tst

tst.run_standard_test("breed_energy", True)
```

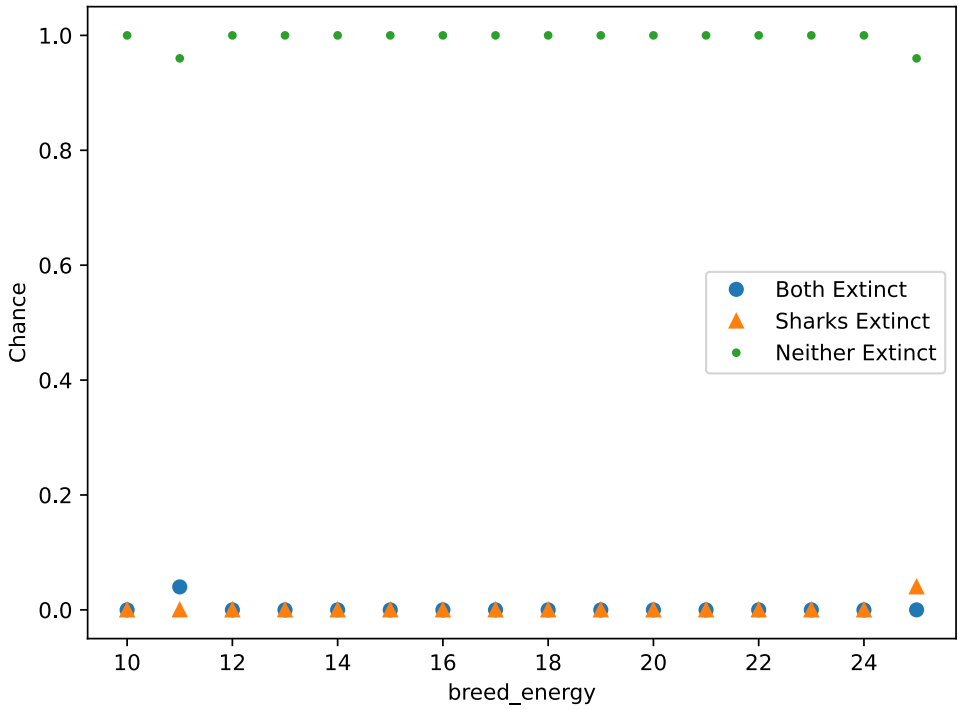


Figure 16: Outcome Chances vs `breed_energy` (Circular Initialization)

Figure 16 displays the outcome chances relative to the value of `breed_energy` with circular initialization. The chance of both species surviving remains close to 1 regardless of `breed_energy`. In contrast, in Figure 3 with random initialization the extinction chances start to rise when `breed_energy` gets too high. Even though the sharks get hardier and hold on to more energy when `breed_time` is large, they still do not consume too many fish. The circular initialization helps remove powerful sharks with lots of energy more regularly, preventing them eating more fish clusters than they should.

```
import measure_ratios as tst

tst.run_standard_test("breed_energy", False)
```

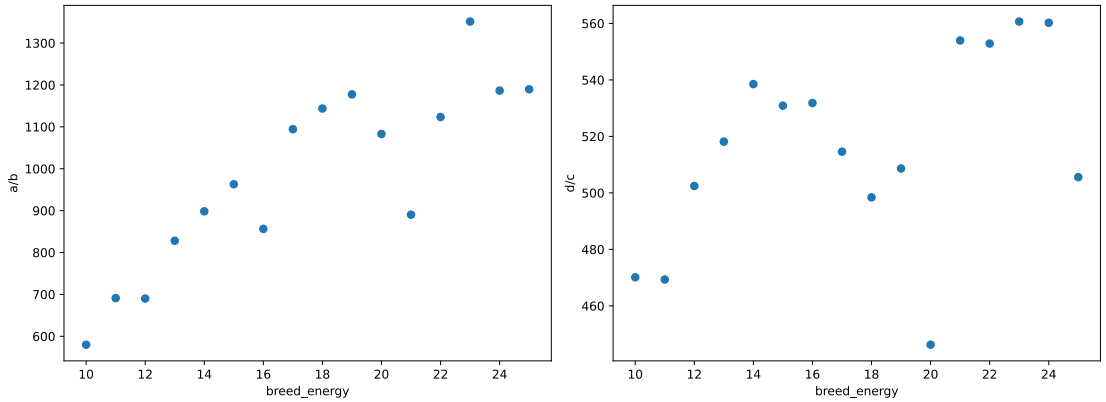


Figure 17: LVM Ratios vs `breed_energy` (Circular Initialization)

Figure 17 plots the LVM ratios as `breed_energy` is varied with circular initialization. As `breed_time` increases, $\frac{a}{b}$ increases while $\frac{d}{c}$ remains roughly constant. In contrast, Figure 11 with random initialization showed $\frac{a}{b}$ remaining constant and $\frac{d}{c}$ decreasing. Since the circular initialization makes sharks with lots of energy less of a threat, that could have caused changes in `breed_energy` to no longer impact a or c, factors that previously relied on changes in shark longevity. But b should still decrease as `breed_energy` increases, since sharks that need more energy to breed produce fewer offspring after eating fish. That would result in $\frac{a}{b}$ increasing and $\frac{d}{c}$ remaining constant as `breed_energy` gets larger, which is the pattern observed in Figure 17.

5.d. `board_area`

```
import measure_outcome_chances as tst

tst.run_standard_test("board_area", True)
```

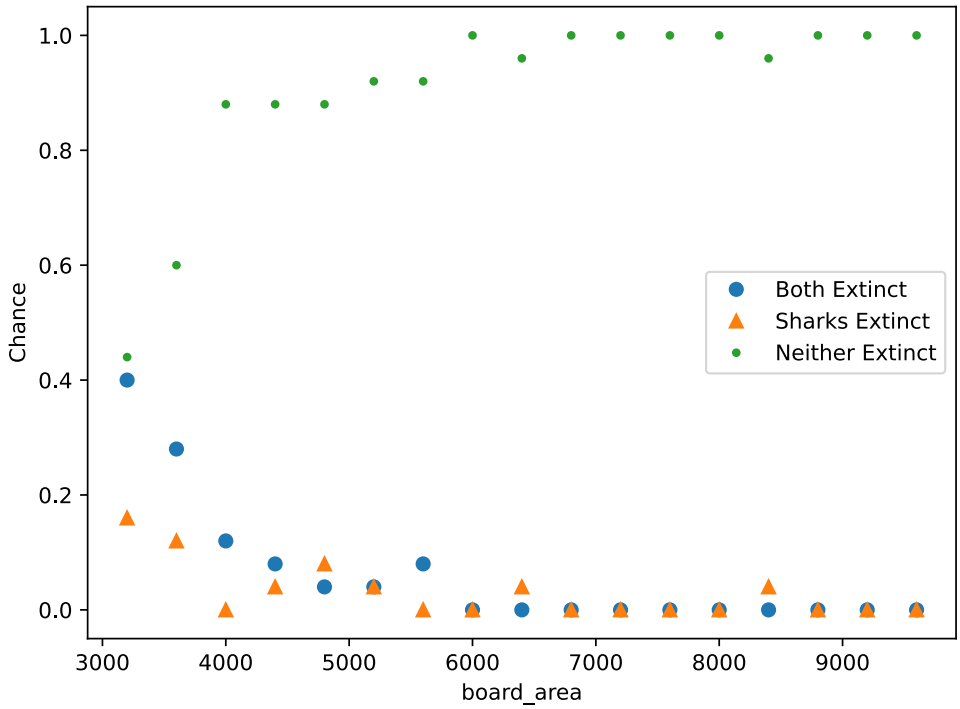


Figure 18: Outcome Chances vs `board_area` (Circular Initialization)

Figure 18 displays the outcome chances relative to the `board_area` with circular initialization. When compared with Figure 4, the graphs have similar shapes when the `board_area` is low. But as the `board_area` gets larger, simulations using circular initialization are better able to support the populations with lower chances of them going extinct. Perhaps the initial chase at the start of circular initialization helps spread the creatures more evenly and out of sync. That prevents either population from getting too low in the oscillations that follow. This effect could happen regardless of `board_area`, assuming it is sufficiently large.

5.e. `aspect_ratio`

```
import measure_outcome_chances as tst

tst.run_standard_test("aspect_ratio", True)
```

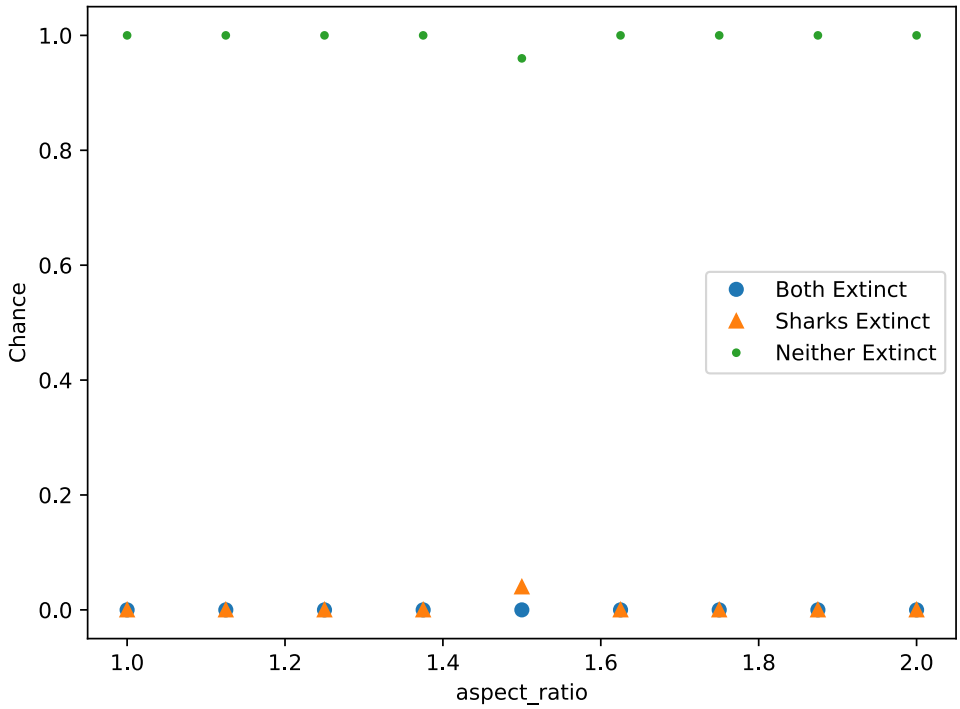


Figure 19: Outcome Chances vs `aspect_ratio` (Circular Initialization)

Figure 19 displays the outcome chances relative to the `aspect_ratio` with circular initialization. The extinction chances remain very close to 0 for all `aspect_ratio` values. In contrast, Figure 5 shows that simulations with higher `aspect_ratio` values do better when random initialization is used. This suggests that the out of phase fish clusters that characterize circular initialization get created regardless of `aspect_ratio`. Whether the expanding fish+shark ring reaches the top+bottom edges of the board at the same time as the left+right edges does not effect the outcome chances. Perhaps fish sneaking inside the expanding ring is what leads to the desynced clusters that help simulations with circular initialization avoid extinction, as that can occur in similar ways no matter what the `aspect_ratio` is.

5.f. `initial_fish`

```
import measure_outcome_chances as tst

tst.run_standard_test("initial_fish", True)
```

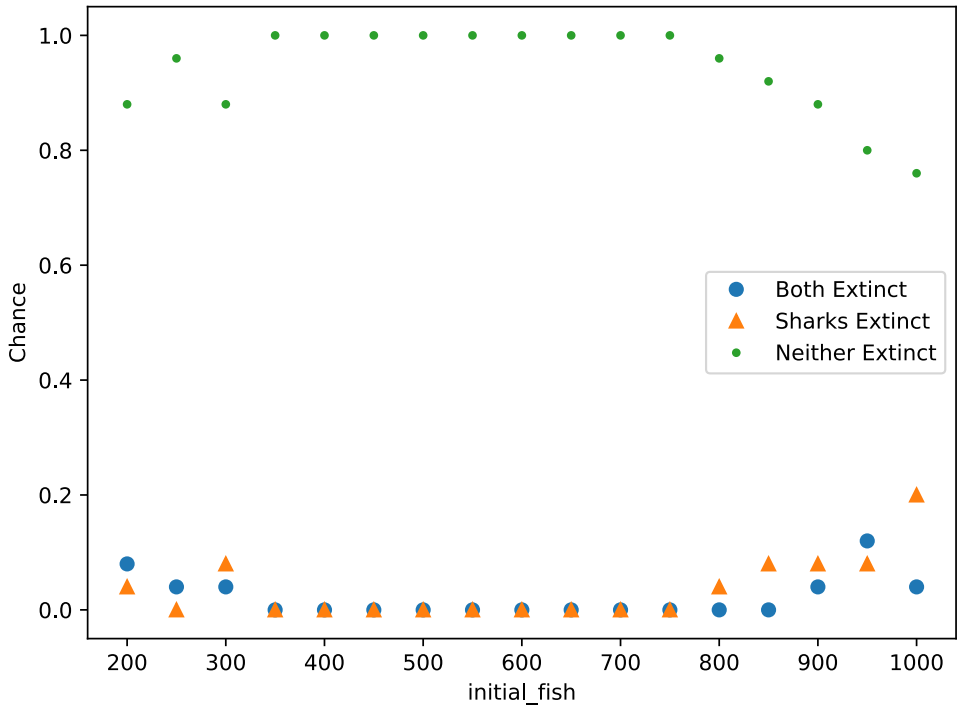


Figure 20: Outcome Chances vs `initial_fish` (Circular Initialization)

Figure 20 displays the outcome chances relative to the `initial_fish` with circular initialization. When compared with Figure 6 with random initialization, the overall graph shapes are similar. Initially, increases in `initial_fish` cause extinction chances to drop. Then, outcome chances remain constant for a period. Finally, the extinction chances start to rise again as `initial_fish` gets larger. In contrast, the simulations with circular initialization seem to have higher chances of neither species going extinct for a wider range of `initial_fish` values. This suggests that circular initialization is more flexible in regards to the starting number of fish, since many of them get eaten as soon as the expanding ring of fish+sharks reaches the edges of the `game_array`.

5.g. `initial_sharks`

```
import measure_outcome_chances as tst

tst.run_standard_test("initial_sharks", True)
```

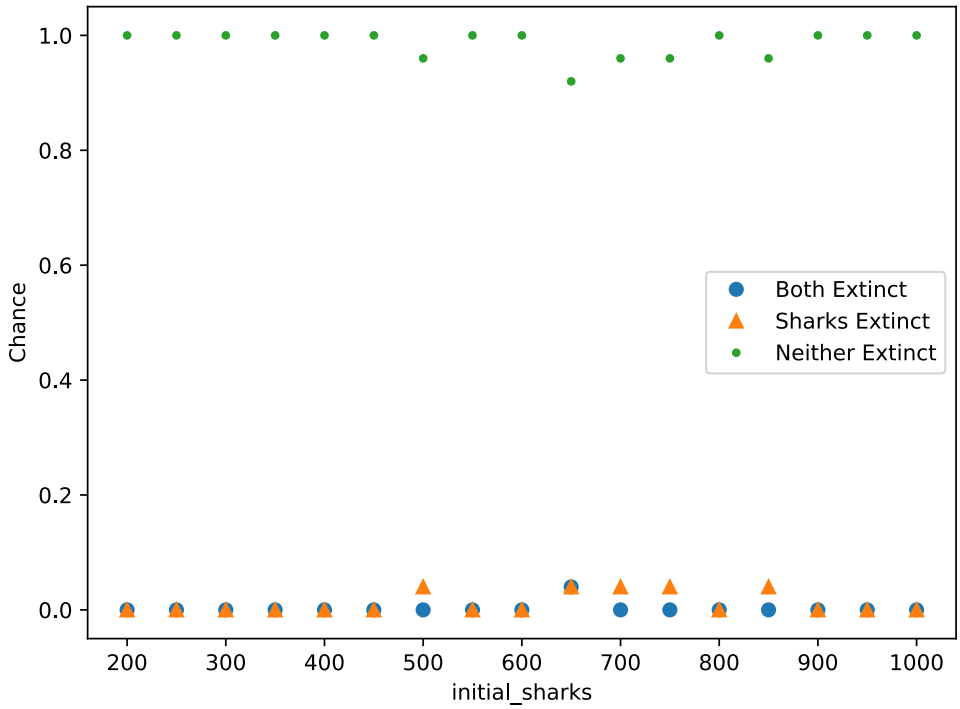


Figure 21: Outcome Chances vs `initial_sharks` (Circular Initialization)

Figure 21 shows the outcome chances relative to the `initial_shark` with circular initialization. Unlike with random initialization where setting `initial_sharks` too high or too low causes extinction chances to rise, a trend depicted in Figure 7, circular initialization seems to have low extinction chances regardless of the `initial_sharks` value. Many of the sharks that start in the center of the disk die early. They do not get an opportunity to eat any fish because the sharks on the outer edge of the disk block them. As a result, simulations that use circular initialization are not noticeably impacted by the `initial_sharks` value.

5.h. `start_energy`

```
import measure_outcome_chances as tst

tst.run_standard_test("start_energy", True)
```

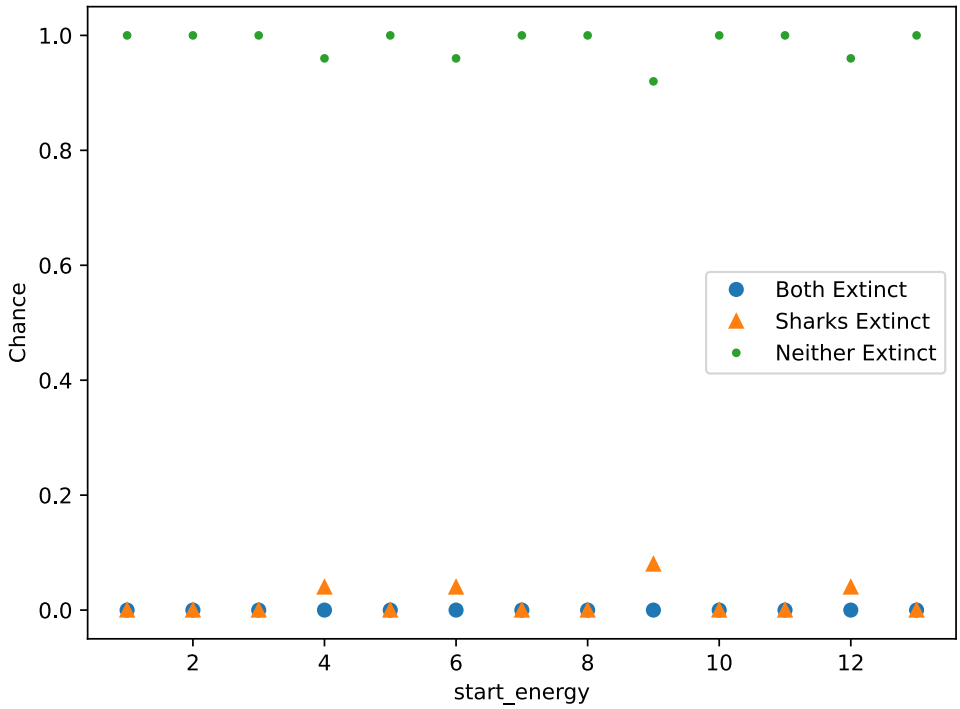


Figure 22: Outcome Chances vs `start_energy` (Circular Initialization)

Figure 22 displays the outcome chances relative to the `start_energy` with circular initialization. The chances of neither species going extinct are higher compared to Figure 8 with random initialization. In addition, with circular initialization there is no noticeable increase in extinction chances as `start_energy` rises. Having parent sharks build up more energy no longer makes them more dangerous hunters, as the circular initialization helps keep the population churning and makes sure enough child sharks are not too far away from their next fish cluster.

6. Modified Lotka-Volterra Model

The differential equations of the LVM do not set any hard limit on the sizes of the predator or prey populations. In the absence of predators ($y = 0$), $\frac{dx}{dt} = +dx$, suggesting that the prey will experience exponential growth indefinitely. For the predators, so long as $ay < bxy$, the predator population will continue to grow. In contrast, population sizes in the Wa-Tor simulation are limited by the dimensions of the `game_array`. Each nonempty cell can only hold one shark or fish, so both population sizes are capped at the total number of cells in the `game_array`. Thus, it seems reasonable to modify the LVM to have terms that involve a carrying capacity (N_∞) for both predators and prey.

To start, the $+dx$ term could be changed into $+dx\left(1 - \frac{x}{N_\infty}\right)$ to model the effects of scarcity and competition between prey. The $+d\left(1 - \frac{x}{N_\infty}\right)$ portion acts like a variable growth rate of the prey. This variable grow rate starts close to d when x is small, then decreases as x gets larger. When $x = N_\infty$, that variable growth rate equals 0, reflecting the limitation that no growth can occur beyond the carrying capacity. In the simulation, fish need to move into empty cells in order to accumulate time and eventually produce an offspring once they reach the `breed_time` requirement. Since the supply of empty cells is limited, fish effectively compete for those free spaces. As more cells get occupied, less fish are able to move and reproduce, thereby limiting the growth rate. And if fish filled the board, no further growth could occur since there are no more spaces to move into.

Next, the $+bxy$ term could be changed into $+bxy\left(1 - \frac{y}{N_\infty}\right)$ to model the effects of competition between predators. The $+b\left(1 - \frac{y}{N_\infty}\right)$ portion acts like a variable proportionality of the predator's growth rate. This proportionality starts close to b when y is small, then decreases as y gets larger. When $y = N_\infty$, that proportionality equals 0, reflecting the limitation that no growth can occur beyond the carrying capacity. In the simulation, sharks produce offspring once their energy passes the `breed_energy` requirement and they have an open space to move into. Sharks gain energy from eating fish and lose 1 energy each step of the simulation. In short, sharks need to eat fish and be next to an empty cell in order to reproduce. When the number of sharks is low, they can easily reach the fish and achieve lots of growth. But as more cells get occupied by sharks, they block each other from reaching the fish more frequently. Once overcrowding starts to take effect, additional sharks do not help boost the population growth any further since those extra hunters are stuck idling. And if sharks filled the board, no further growth could occur since there are no more spaces to move into.

Finally, the $-cyx$ term could be changed into $-cyx\left(1 - \frac{y}{N_\infty}\right)$ for similar reasons as the modification to $+bxy$. As more sharks fill the board, not as many additional fish get eaten due to the sharks starting to crowd each other out.

Making those changes yields the following modified Lotka-Volterra Model.

$$\frac{dy}{dt} = -ay + bxy\left(1 - \frac{y}{N_\infty}\right) \quad \frac{dx}{dt} = +dx\left(1 - \frac{x}{N_\infty}\right) - cyx\left(1 - \frac{y}{N_\infty}\right)$$

Running the simulation using the `simulation_playground.py` script with the default parameters set yields the plot shown in Figure 23. Notice that the paths on the stream plot tend to spiral counterclockwise inward. In contrast, the streams of the LVM form closed loops and stay at fixed levels, as shown in Figure 24. But with the modified LVM depicted in Figure 25, the streams spiral inward similar to what was seen in Figure 23. Thus, this adjusted model successfully captures additional aspects of the simulation behavior.

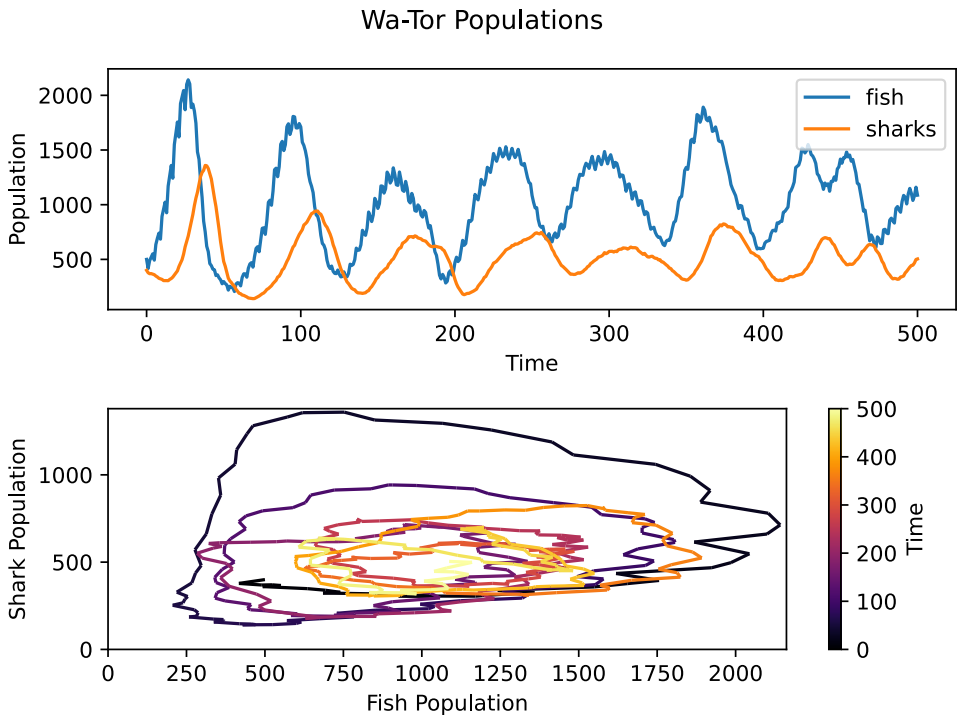


Figure 23: Sample Plot with Default Parameters

```
import stream_plotter as sp

def y_prime(x, y, a, b):
    return -a*y + b*x*y

def x_prime(x, y, c, d):
    return d*x - c*y*x

board_size = 80 * 90
points = 1000
padding = 10

x_min = -padding
x_max = board_size + padding
x_points = points

y_min = -padding
y_max = board_size + padding
y_points = points

a = 1100
b = 1
c = 1
d = 550

y_diff = lambda x, y: y_prime(x, y, a, b)
```

```
x_diff = lambda x, y: x_prime(x, y, c, d)

fig, ax = sp.create_stream_plot(x_min, x_max, x_points, y_min, y_max, y_points,
y_diff, x_diff)
ax.set(xlabel="Fish Population", ylabel="Shark Population")
fig.savefig("media/lvm_stream_plot.svg")
```

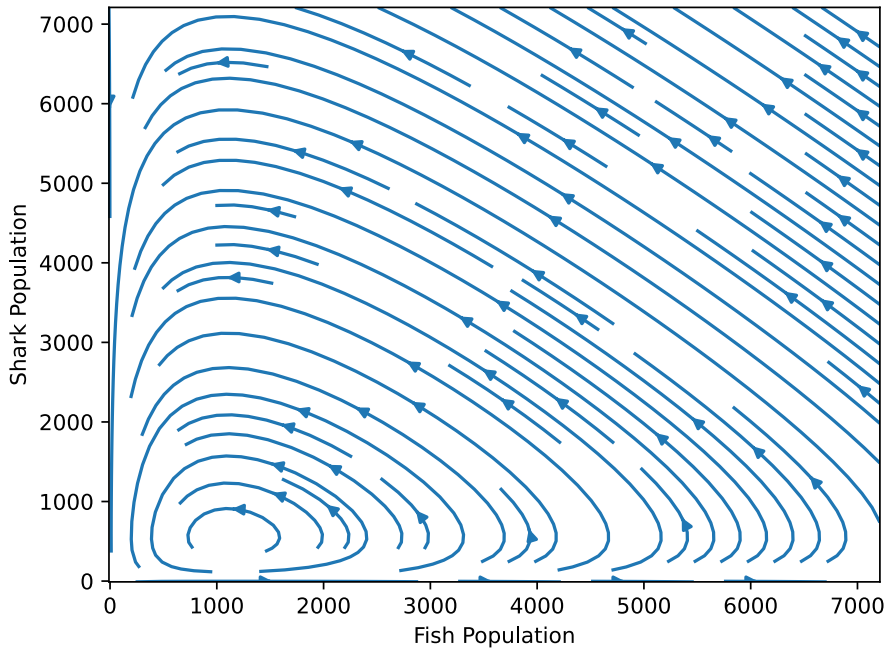


Figure 24: LVM Stream Plot

```
import stream_plotter as sp

def y_prime(x, y, a, b, N_oo):
    return -a*y + b*x*y*(1 - y/N_oo)

def x_prime(x, y, c, d, N_oo):
    return d*x*(1 - x/N_oo) - c*y*x*(1 - y/N_oo)

board_size = 80 * 90
points = 1000
padding = 10

x_min = -padding
x_max = board_size + padding
x_points = points

y_min = -padding
y_max = board_size + padding
y_points = points
```

```
a = 1100
b = 1
c = 1
d = 550

y_diff = lambda x, y: y_prime(x, y, a, b, board_size)
x_diff = lambda x, y: x_prime(x, y, c, d, board_size)

fig, ax = sp.create_stream_plot(x_min, x_max, x_points, y_min, y_max, y_points,
y_diff, x_diff)
ax.set(xlabel="Fish Population", ylabel="Shark Population")
fig.savefig("media/lvm_modified_stream_plot.svg")
```

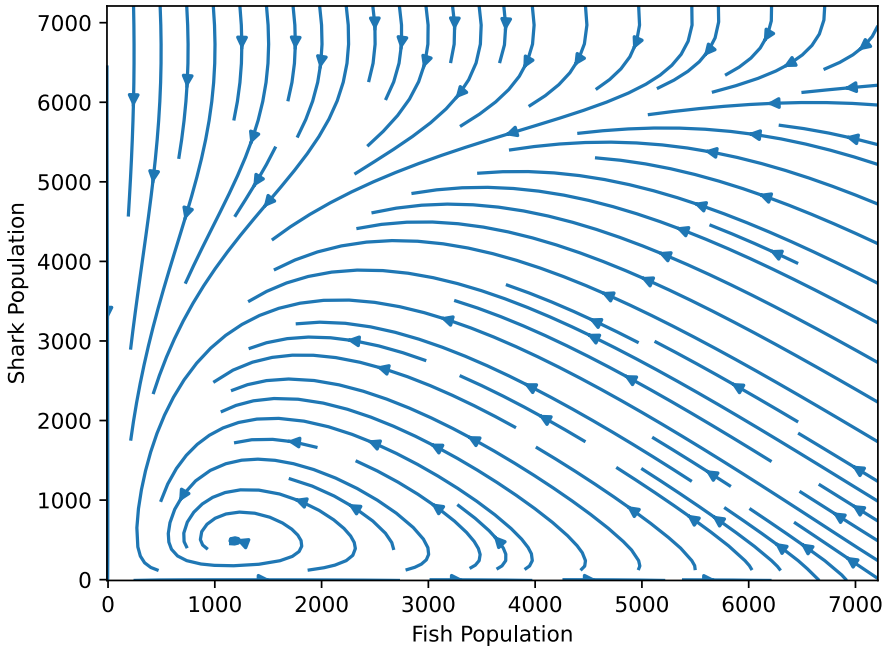


Figure 25: Modified LVM Stream Plot

7. Simulation Code

wa_tor.py

```
# A program implementing and measuring a Wa-Tor (water torus) Simulation

import numpy as np              # Library needed for numerical functions
import matplotlib.pyplot as plt # Library needed to plot results
import imageio.v2 as io         # Library for converting a collection of image files
to a gif
rng = np.random.default_rng()   # Random number generator

from matplotlib.collections import LineCollection

# Functions for running the simulation

def run_simulation(game_array, steps, breed_time, energy_gain, breed_energy,
start_energy, print_progress=False):
    """
    Run the simulation for the given number of steps, performing all the movements,
hunts, breedings, and deaths.
    If the fish population fills the board or all the sharks and fish die, terminate
early.
    Pass in the relevant simulation parameters.
    Return a list containing the game_array at each step.
    """
    game_array_list = [game_array]
    percent = 0

    for k in range(steps):
        game_array = step_game(game_array, breed_time, energy_gain, breed_energy,
start_energy)
        game_array_list.append(game_array)

        # Print the current progress if the percentage has changed
        if print_progress:
            new_percent = np.floor((k + 1) / steps * 100).astype(int)
            if new_percent > percent:
                percent = new_percent
                print(f"{percent:3}%", end="\r")

        # If the array is full of fish or both species have gone extinct, stop
simulating early
        if check_if_fish_fill_board(game_array) or
check_if_everything_extinct(game_array):
            break

    return game_array_list

def run_simulation_minimal(game_array, steps, breed_time, energy_gain, breed_energy,
start_energy):
    """
```

```
    Run the simulation for the given number of steps, performing all the movements,
hunts, breedings, and deaths.
    If the fish population fills the board or all the sharks and fish die, terminate
early.
    Pass in the relevant simulation parameters.
    Return two lists containing the fish and shark populations at each step.
    """
    fish_counts = [count_fish(game_array)]
    shark_counts = [count_sharks(game_array)]

    for _ in range(steps):
        game_array = step_game(game_array, breed_time, energy_gain, breed_energy,
start_energy)
        fish_counts.append(count_fish(game_array))
        shark_counts.append(count_sharks(game_array))

        # If the array is full of fish or both species have gone extinct, stop
simulating early
        if check_if_fish_fill_board(game_array) or
check_if_everything_extinct(game_array):
            break

    return fish_counts, shark_counts

def step_game(old_array, breed_time, energy_gain, breed_energy, start_energy):
    """
    Increment the simulation by 1 step, performing all the movements, hunts,
breedings, and deaths.
    Pass in the relevant simulation parameters.
    Return a new game array with the updates.
    """
    # Copy the old array to avoid overwriting the original
    old_array = old_array.copy()
    # Create a new array to return with the updates
    new_array = create_empty_game_array(old_array.shape)

    # Visit each cell in the array in a random order
    locs = create_random_location_sequence(old_array)
    for loc in locs:
        cell_value = old_array[loc]

        # Handle fish behavior
        if cell_value > 0:
            # Find the adjacent cells that are open in both arrays
            old_locs = get_empty_adjacent_locations(old_array, *loc)
            new_locs = get_empty_adjacent_locations(new_array, *loc)
            available_locs = list_intersection(old_locs, new_locs)
            # If there are open adjacent cells, randomly move the fish into one
            if len(available_locs) > 0:
                chosen_loc = choose_random_location(available_locs)
                # Check the fish is eligible to breed
```

```
                if cell_value > breed_time:
                    # Place the fish in the new location, reset, and place a new fish
in the old location
                    new_array[chosen_loc] = 1
                    new_array[loc] = 1
                else:
                    # Place the fish in the new location, incrementing its time by 1
                    new_array[chosen_loc] = cell_value + 1
            # If there are no open cells, the fish stays in place
            else:
                new_array[loc] = cell_value

        # Handle shark behavior
        elif cell_value < 0:
            # Find the adjacent cells that contain fish in either array
            old_locs = get_fish_occupied_adjacent_locations(old_array, *loc)
            new_locs = get_fish_occupied_adjacent_locations(new_array, *loc)
            available_locs = list_union(old_locs, new_locs)
            # If there are fish occupied adjacent cells, randomly move the shark into
one
            if len(available_locs) > 0:
                chosen_loc = choose_random_location(available_locs)
                # Check the shark is eligible to breed
                if cell_value < -breed_energy:
                    # Place the shark in the new location, and place a new shark at
the old location
                    # Share the energy from the eating the fish
                    new_array[chosen_loc] = cell_value + start_energy -
round(energy_gain / 2) + 1
                    new_array[loc] = -start_energy - round(energy_gain / 2)
                else:
                    # Place the shark in the new location
                    # Give it all the energy from eating the fish
                    new_array[chosen_loc] = cell_value - energy_gain + 1
                # Clear the eaten fish from the old array, if it came from there
                if old_array[chosen_loc] > 0:
                    old_array[chosen_loc] = 0
            # Try to move the shark randomly into an empty adjacent cell
            else:
                # Find the adjacent cells that are open in both arrays
                old_locs = get_empty_adjacent_locations(old_array, *loc)
                new_locs = get_empty_adjacent_locations(new_array, *loc)
                available_locs = list_intersection(old_locs, new_locs)
                # If there are open adjacent cells, randomly move the shark into one
                if len(available_locs) > 0:
                    chosen_loc = choose_random_location(available_locs)
                    # Check the shark is eligible to breed
                    if cell_value < -breed_energy:
                        # Place the shark in the new location, and place a new shark
at the old location
                        new_array[chosen_loc] = cell_value + start_energy + 1
```

```
                        new_array[loc] = -start_energy
                    else:
                        # Place the shark in the new location
                        new_array[chosen_loc] = cell_value + 1
                # The shark can't move and stays in place
                else:
                    new_array[loc] = cell_value + 1

        # Remove the creature from the old array
        old_array[loc] = 0

    return new_array

# Functions for game array initialization

def create_empty_game_array(dims):
    """
    Create an empty game array (filled with zeros) with the given dimensions.
    """
    return np.zeros(dims, dtype=int)

def initialize_game_array_randomly(game_array, initial_fish, initial_sharks,
breed_time, breed_energy):
    """
    Randomly fill the game array with the given number of fish and sharks.
    Each fish will be given a random time.
    Each shark will be given a random amount of energy.
    """
    # Check that there are enough spaces to fit all the fish and sharks
    initial_creatures = initial_fish + initial_sharks
    assert game_array.size >= initial_creatures
    # Randomly place fish then sharks into the game array
    locs = create_random_location_sequence(game_array)
    for i in range(initial_creatures):
        loc = locs[i]
        if i < initial_fish:
            # Generate a random initial time for fish
            initial_value = generate_random_fish_time(breed_time)
        else:
            # Generate a random initial energy for sharks
            initial_value = generate_random_shark_energy(breed_energy)
        # Place the creature in the game array
        game_array[loc] = initial_value

def initialize_game_array_circular(game_array, initial_fish, initial_sharks,
breed_time, breed_energy):
    """
    Fill the game array with the given number of fish and sharks in a circular
pattern.
    Populate a central disk with sharks, and surround them with a ring of fish.
    Each fish will be given a random time.
```

```
    Each shark will be given a random amount of energy.
    """
    rows = game_array.shape[0]
    cols = game_array.shape[1]
    row_center = rows / 2
    col_center = cols / 2
    for i in range(rows):
        for j in range(cols):
            y = i - row_center
            x = j - col_center
            # Check if the position is within the central shark disk
            if x**2 + y**2 < initial_sharks / np.pi:
                # Place a shark in the game array with a random energy
                game_array[i, j] = generate_random_shark_energy(breed_energy)
            # Check if the position is within the surrounding fish ring
            elif x**2 + y**2 < (initial_sharks + initial_fish) / np.pi:
                # Place a fish in the game array with a random time
                game_array[i, j] = generate_random_fish_time(breed_time)

# Functions for randomization

def create_random_location_sequence(array):
    """
    Create a list of (i, j) indices for each location in the array.
    Return them in a random order.
    """
    # Randomly generate a sequence of positions in the array
    # Cells are numbered starting at 0 in the upper left corner, increasing by 1 as
you move right then down
    N = array.size
    positions = rng.choice(N, N, replace=False)
    # Map each position to an (i, j) pair
    coordinates = []
    cols = array.shape[1]
    for pos in positions:
        i = pos // cols
        j = pos % cols
        coordinates.append((i, j))
    return coordinates

def choose_random_location(locs):
    """
    Return a random location (i, j) in the given list of locations.
    """
    return tuple(rng.choice(locs))

def generate_random_fish_time(breed_time):
    """
    Return a random initial time for a fish.
    """
    return rng.integers(1, breed_time, endpoint=True)
```

```
def generate_random_shark_energy(breed_energy):
    """
    Return a random initial energy for a shark.
    """
    return rng.integers(-breed_energy, -1, endpoint=True)

# Functions for getting useful information out of the game array

def count_fish(game_array):
    """
    Return the fish count for the given game array.
    Fish are represented by positive values, sharks by negative values, and empty
spaces by 0.
    """
    fish_count = (game_array > 0).sum()
    return fish_count

def count_sharks(game_array):
    """
    Return the shark count for the given game array.
    Fish are represented by positive values, sharks by negative values, and empty
spaces by 0.
    """
    shark_count = (game_array < 0).sum()
    return shark_count

def check_if_everything_extinct(game_array):
    """
    Return whether all creatures in the game array have gone died.
    In other words, whether the fish and sharks are both extinct.
    """
    empty_spaces = (game_array == 0).sum()
    return empty_spaces == game_array.size

def check_if_fish_fill_board(game_array):
    """
    Return whether all creatures in the game array are fish.
    In other words, whether the fish have filled the board.
    This happens when the sharks go extinct.
    """
    fish_count = count_fish(game_array)
    return fish_count == game_array.size

def get_adjacent_locations(game_array, i, j):
    """
    Return a list of locations in the game array adjacent to the given location.
    Cells are adjacent if they can be reached by moving up, down, left, or right.
    Moves at the edges of the array wrap around to the other side.
    """
    locs = []
```

```
    for nudge in [+1, -1]:
        for axis in [0, 1]:
            loc = [i, j]
            loc[axis] = (loc[axis] + nudge) % game_array.shape[axis]
            locs.append(tuple(loc))
    return locs

def get_empty_adjacent_locations(game_array, i, j):
    """
    Return a list of locations in the game array adjacent to the given location that
are empty.
    """
    locs = get_adjacent_locations(game_array, i, j)
    empty_locs = []
    for loc in locs:
        if game_array[loc] == 0:
            empty_locs.append(loc)
    return empty_locs

def get_fish_occupied_adjacent_locations(game_array, i, j):
    """
    Return a list of locations in the game array adjacent to the given location that
are occupied by fish.
    """
    locs = get_adjacent_locations(game_array, i, j)
    fish_occupied_locs = []
    for loc in locs:
        if game_array[loc] > 0:
            fish_occupied_locs.append(loc)
    return fish_occupied_locs

# Functions to help with comparing/combining lists

def list_intersection(list_1, list_2):
    """
    Return the common elements of two lists without repeats.
    """
    result = []
    for item in list_1:
        if item in list_2:
            result.append(item)
    return result

def list_union(list_1, list_2):
    """
    Return the combined elements of two lists without repeats.
    """
    result = []
    for item in list_1 + list_2:
        if item not in result:
            result.append(item)
```

```
    return result

# Functions for creating visualizations

def create_simulation_paramater_str(dims, breed_time, energy_gain, breed_energy,
start_energy, initial_fish, initial_sharks):
    """
    Return a standardized string summarizing the simulation parameters.
    This is useful for creating file names that describe the setup.
    """
    dimensions = f"{dims[0]}x{dims[1]}"
    paramaters = f"{breed_time},{energy_gain},{breed_energy},{start_energy}"
    initial_conditions = f"{initial_sharks},{initial_fish}"
    return f"{dimensions}_({paramaters})_({initial_conditions})"

def create_image_array(game_array):
    """
    Return the game array converted into a format for visual display.
    Each cell is given a different color:
    - empty: white
    - fish: red
    - sharks: blue
    The cell is translated into a 16x16 square of the RGB value for its color.
    """
    square_width = 16
    rows = game_array.shape[0] * square_width
    cols = game_array.shape[1] * square_width
    result = np.zeros((rows, cols, 3), dtype="uint8")
    empty_color = np.array([255, 255, 255], dtype="uint8")
    fish_color = np.array([255, 0, 0], dtype="uint8")
    shark_color = np.array([0, 0, 255], dtype="uint8")

    for i, row in enumerate(game_array):
        row_range = slice(i*square_width, (i + 1)*square_width)
        for j, cell_value in enumerate(row):
            col_range = slice(j*square_width, (j + 1)*square_width)
            if cell_value > 0:
                color = fish_color  # Fish are positive
            elif cell_value < 0:
                color = shark_color # Sharks are negative
            else:
                color = empty_color # Empty spaces are 0
            result[row_range, col_range, :] = color

    return result

def create_simulation_animation(game_array_list, fname, fps=20):
    """
    Create a gif file animating the simulation.
    Pass in a list of game arrays and the desired file name.
    """
```

```
    img_data = [create_image_array(game_array) for game_array in game_array_list]
    io.mimwrite(fname, img_data, format=".gif", fps=fps)

def create_simulation_plots(fish_counts, shark_counts, fname):
    """
    Create plots describing the simulation.
    Plot the fish counts and shark counts over time, and create a phase plot of the
two populations.
    Save the plot at the given file name.
    """
    assert len(fish_counts) == len(shark_counts)
    actual_steps = len(fish_counts)
    fig, axes = plt.subplots(2, 1)
    fig.suptitle("Wa-Tor Populations")

    # Plot each population over time
    axes[0].plot(range(actual_steps), fish_counts, label="fish")
    axes[0].plot(range(actual_steps), shark_counts, label="sharks")
    axes[0].legend()
    axes[0].set(xlabel="Time", ylabel="Population")

    # Plot each population against the other
    # Create a color gradient for time
    segs = [[(fish_counts[i], shark_counts[i]), (fish_counts[i + 1], shark_counts[i +
1])] for i in range(actual_steps - 1)]
    line_collection = LineCollection(segs, array=range(actual_steps), cmap="inferno")
    fig.colorbar(line_collection, ax=axes[1], label="Time")
    axes[1].add_collection(line_collection)
    axes[1].set(xlabel="Fish Population", ylabel="Shark Population")
    padding = 20
    axes[1].set(xlim=(0, max(fish_counts) + padding), ylim=(0, max(shark_counts) +
padding))

    fig.tight_layout()
    fig.savefig(fname)
```

simulation_playground.py

```
import wa_tor

# Main parameters of the simulation
breed_time = 3          # Number of steps before a fish is capable of duplicating
energy_gain = 4         # Additional steps granted to a shark after eating a fish
breed_energy = 15       # Number of stored steps before a shark is capable of
duplicating

# Other simulation parameters
dims = (80, 90)         # Size of the simulation window
initial_fish = 500      # Starting number of fish
initial_sharks = 400    # Starting number of sharks
```

```
steps = 500             # Time duration of the simulation
start_energy = 9        # Number of moves a child shark begins with
use_basic_setup = True  # Whether to use a random initial distribution (or not)

# Initialize the game array
initial_game_array = wa_tor.create_empty_game_array(dims)
if use_basic_setup:
    wa_tor.initialize_game_array_randomly(initial_game_array, initial_fish,
initial_sharks, breed_time, breed_energy)
else:
    wa_tor.initialize_game_array_circular(initial_game_array, initial_fish,
initial_sharks, breed_time, breed_energy)

# Run the Simulation
print("Playing game...")
game_array_list = wa_tor.run_simulation(initial_game_array, steps, breed_time,
energy_gain, breed_energy, start_energy, print_progress=True)
print("\nSimulation finished")

# Create an animation and plots
actual_steps = len(game_array_list)
fish_counts = [wa_tor.count_fish(game_array) for game_array in game_array_list]
shark_counts = [wa_tor.count_sharks(game_array) for game_array in game_array_list]

paramater_str = wa_tor.create_simulation_paramater_str(dims, breed_time, energy_gain,
breed_energy, start_energy, initial_fish, initial_sharks)
animation_fname = f"media/TestAnimation_{paramater_str}_{actual_steps}.gif"
plot_fname = f"media/TestPlot_{paramater_str}_{actual_steps}.png"

wa_tor.create_simulation_animation(game_array_list, animation_fname)
wa_tor.create_simulation_plots(fish_counts, shark_counts, plot_fname)
```

stream_plotter.py

```
import numpy as np
import matplotlib.pyplot as plt

def create_stream_plot(x_min, x_max, x_points, y_min, y_max, y_points, numerator,
denominator=(lambda x, y: 1)):
    """
    Create a stream plot for the differential equation dy/dx = numerator/denominator.
    Numerator and denominator are functions that take two inputs (x, y) and return a
value.
    Denominator defaults to returning 1 regardless of input.
    The grid should have the specified number of points in the x-direction and y-
direction within the specified ranges.
    Return the figure and axes created.
    """
    x_values = np.linspace(x_min, x_max, x_points)
    y_values = np.linspace(y_min, y_max, y_points)
    x, y = np.meshgrid(x_values, y_values, sparse=True)
```

```
    top = numerator(x, y)
    bottom = denominator(x, y)
    magnitude = np.sqrt(top**2 + bottom**2)
    x_arrows = bottom / magnitude
    y_arrows = top / magnitude

    fig, ax = plt.subplots()
    ax.streamplot(x_values, y_values, x_arrows, y_arrows)

    return fig, ax
```