

Study of Predator-Prey Dynamics

Vincent Edwards, Julia Corrales, and Rachel Gossard

April 13, 2025

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 $\frac{dy}{dx}$.

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

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 $y = \frac{d}{c}$.

Types of Simulation Outcomes

Conditions for Good Modeling

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

```

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(5600, 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)

```

breed_time

```

import measure_outcome_chances as tst

tst.run_standard_test("breed_time", True)

```

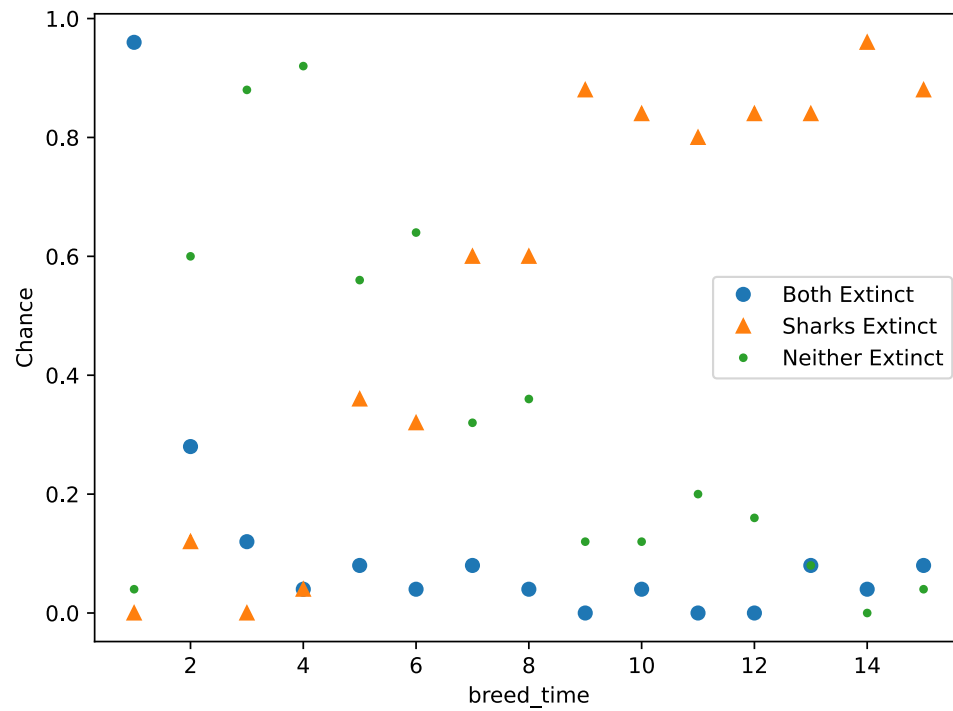


Figure 1: Outcome Chances vs breed_time

energy_gain

```
import measure_outcome_chances as tst  
tst.run_standard_test("energy_gain", True)
```

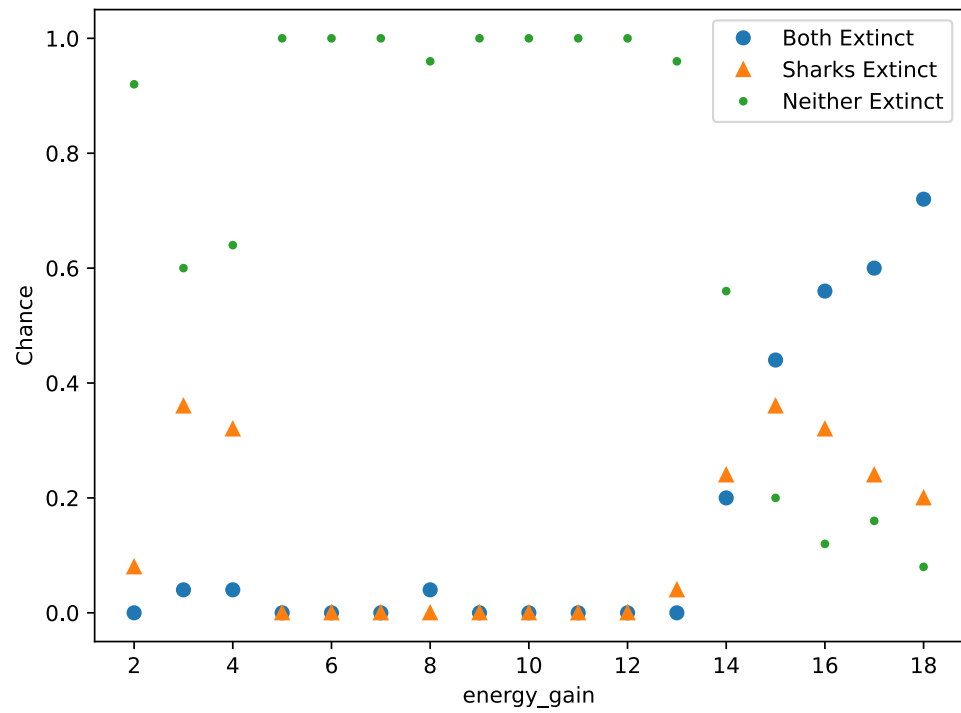


Figure 2: Outcome Chances vs energy_gain

breed_energy

```
import measure_outcome_chances as tst  
tst.run_standard_test("breed_energy", True)
```

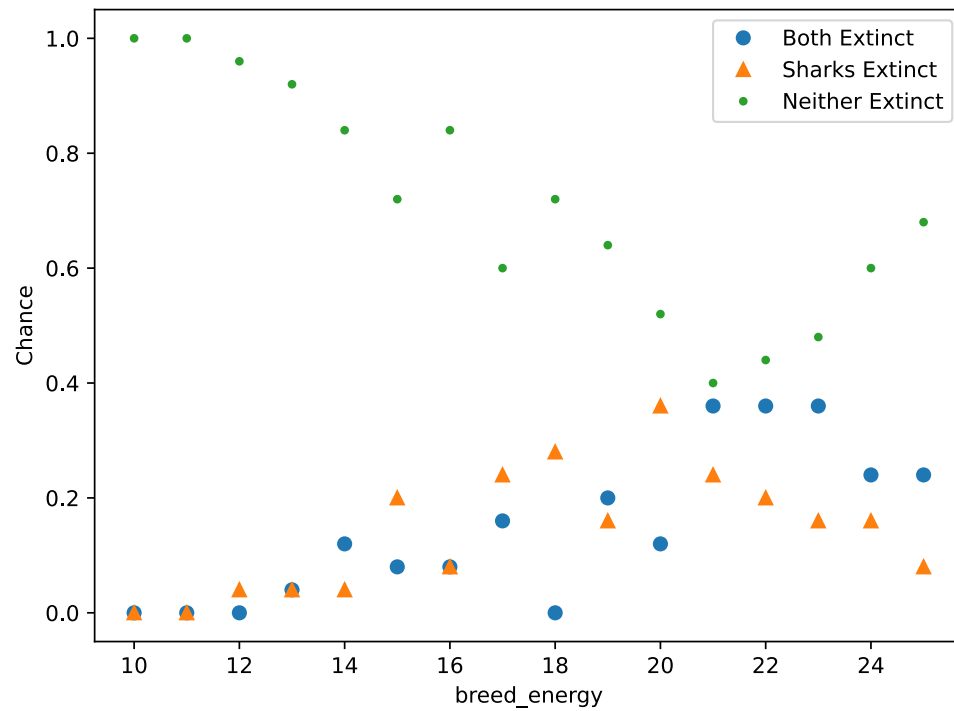


Figure 3: Outcome Chances vs breed_energy

board_area

```
import measure_outcome_chances as tst  
  
tst.run_standard_test("board_area", True)
```

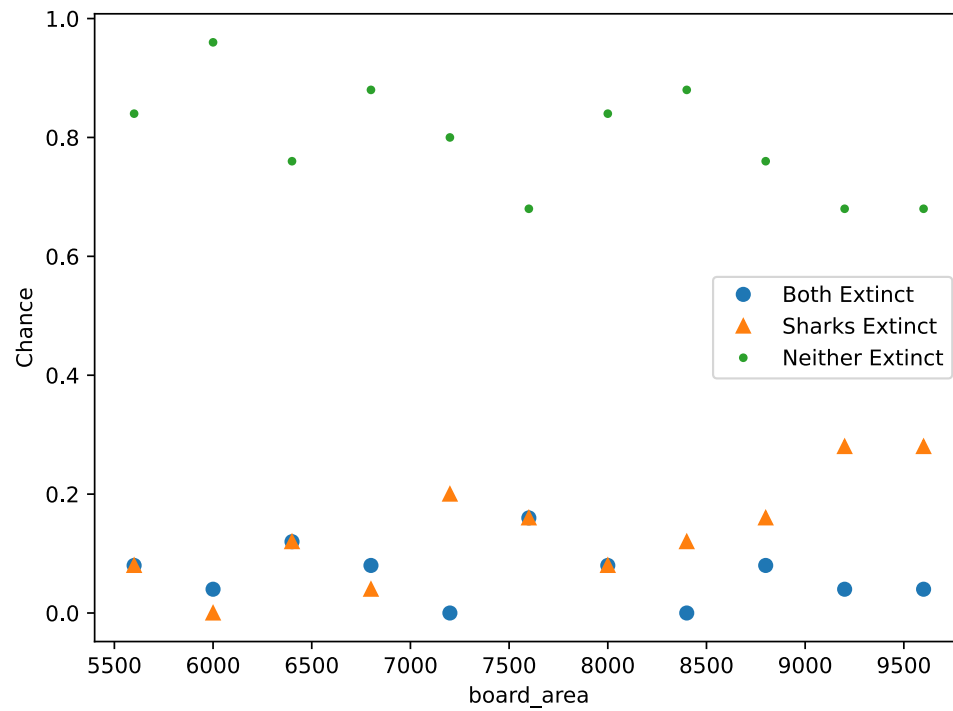


Figure 4: Outcome Chances vs board_area

aspect_ratio

```
import measure_outcome_chances as tst  
tst.run_standard_test("aspect_ratio", True)
```

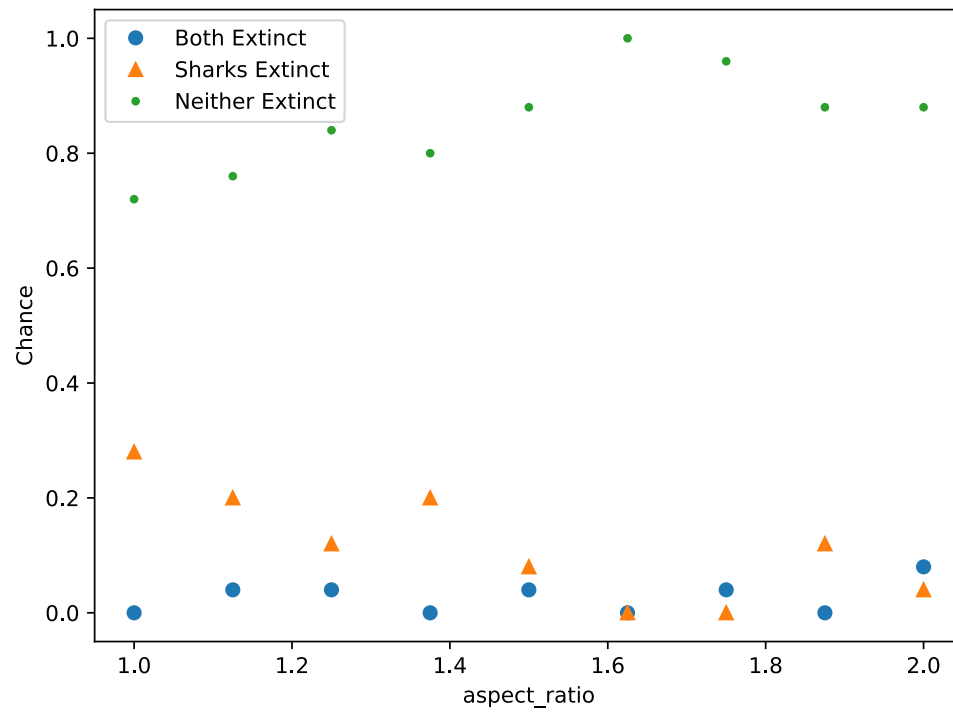



Figure 5: Outcome Chances vs aspect_ratio

initial_fish

```
import measure_outcome_chances as tst  
tst.run_standard_test("initial_fish", True)
```

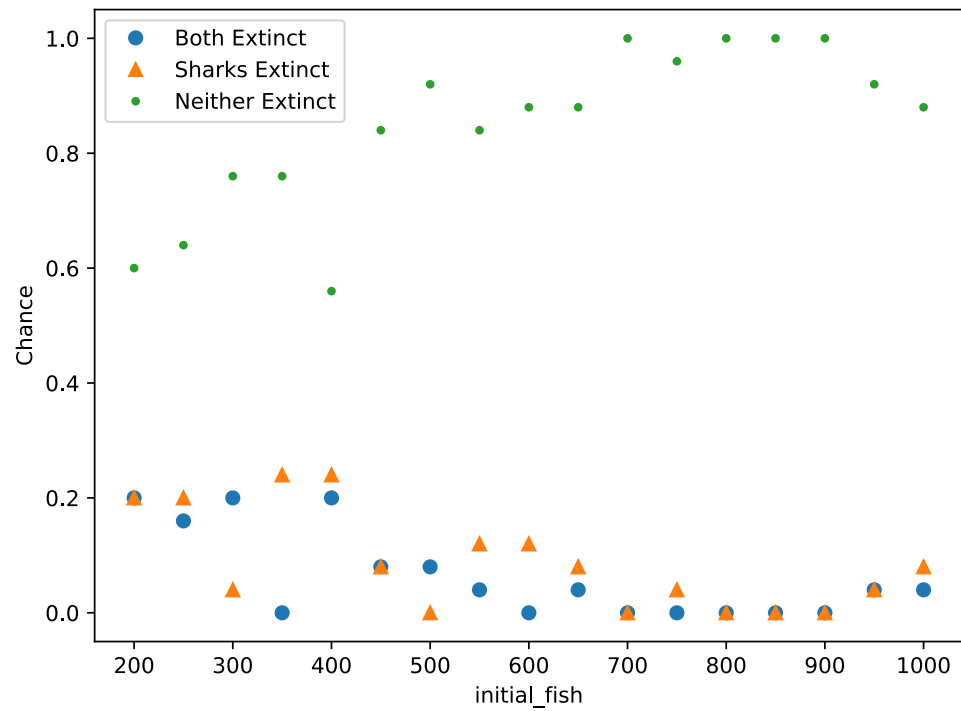


Figure 6: Outcome Chances vs initial_fish

initial_sharks

```
import measure_outcome_chances as tst  
tst.run_standard_test("initial_sharks", True)
```

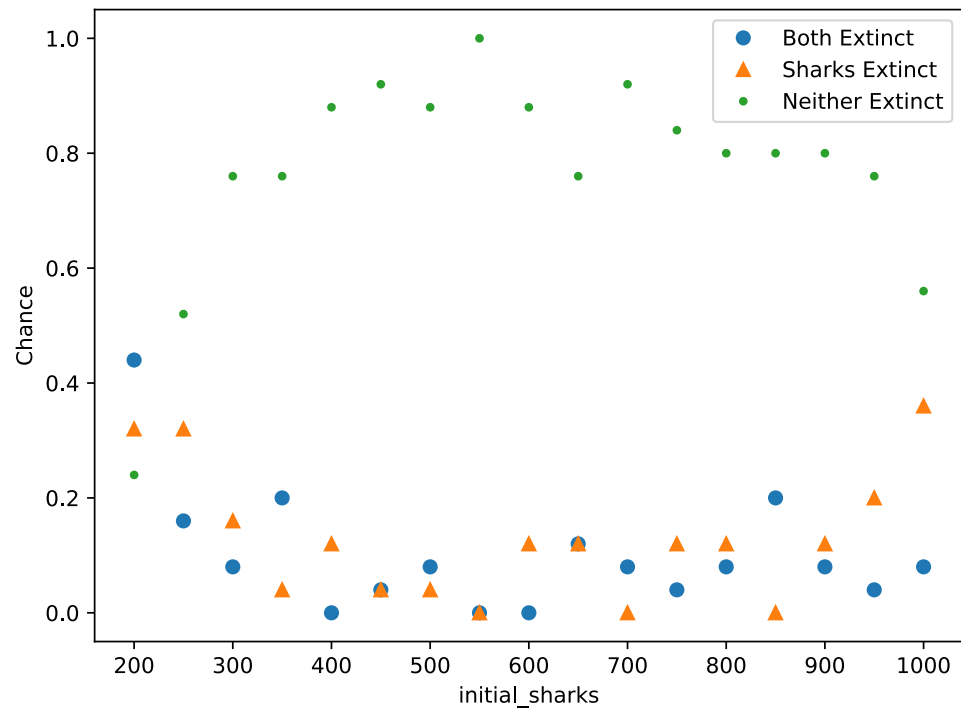


Figure 7: Outcome Chances vs initial_sharks

start_energy

```
import measure_outcome_chances as tst  
tst.run_standard_test("start_energy", True)
```

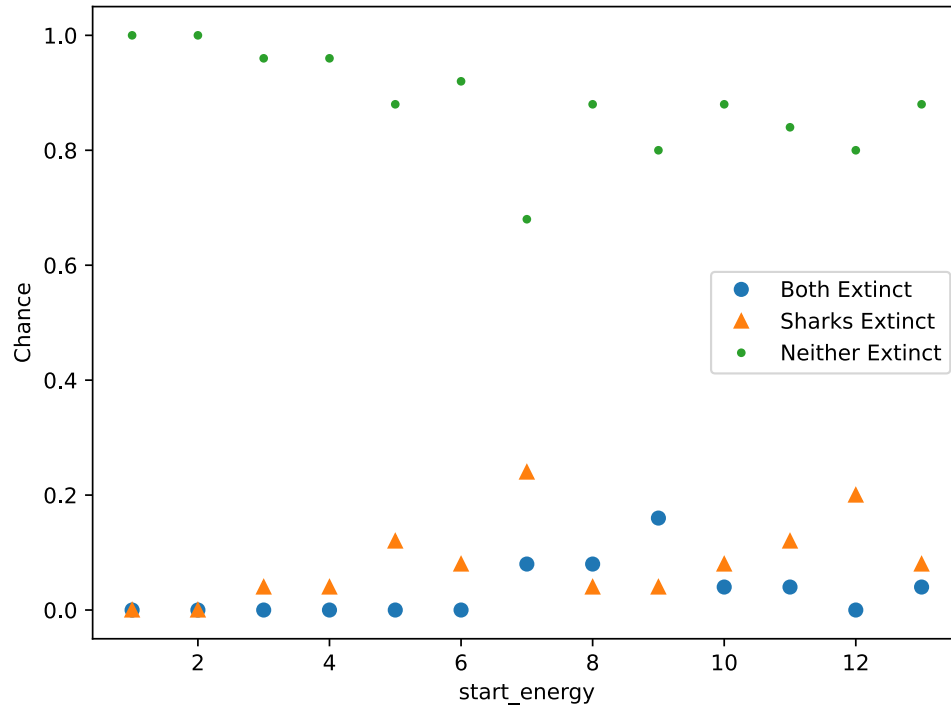


Figure 8: Outcome Chances vs start_energy

Main Simulation Parameters

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)

```

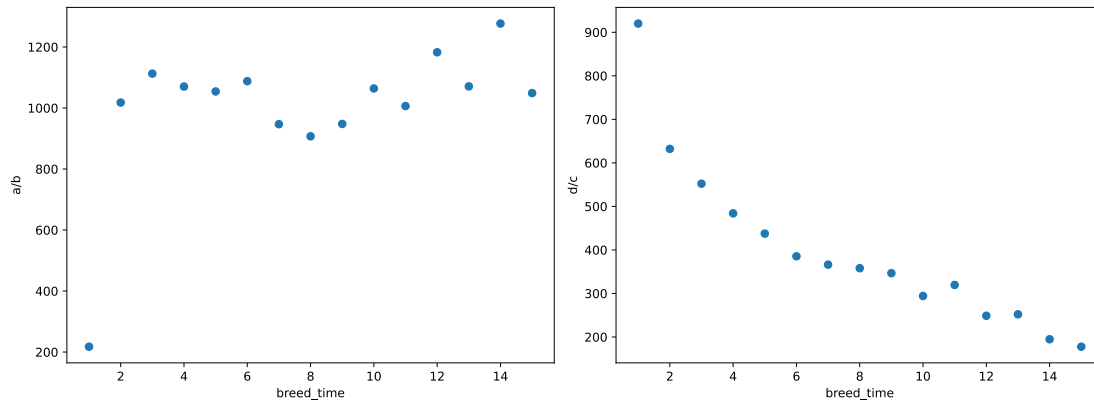
breed_time

```

import measure_ratios as tst

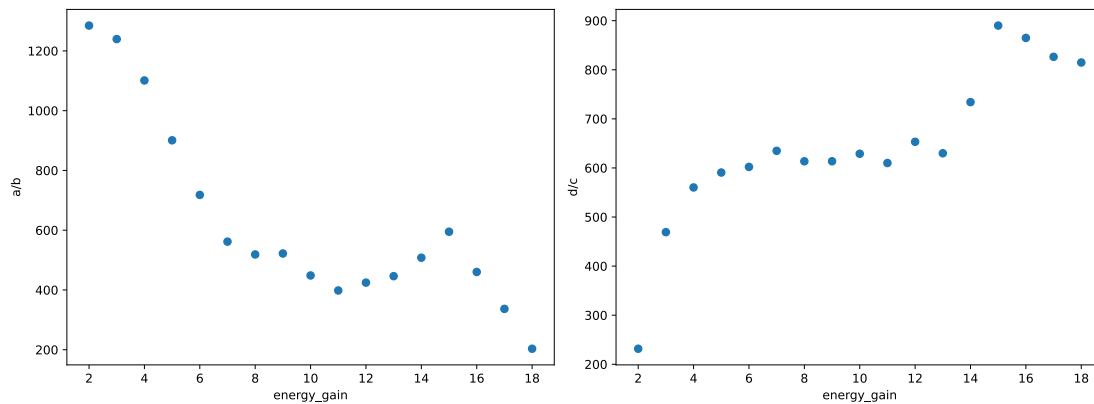
tst.run_standard_test("breed_time", True)

```

Figure 9: LVM Ratios vs `breed_time`**energy_gain**

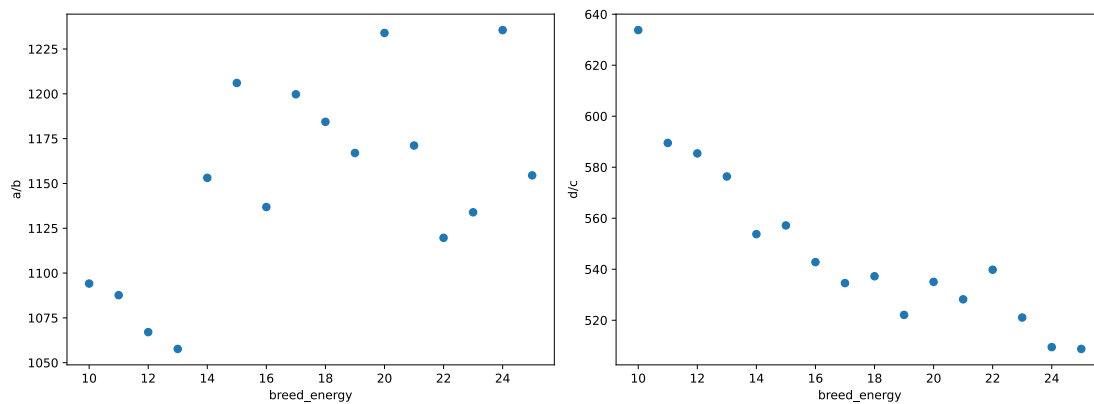
```
import measure_ratios as tst

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

Figure 10: LVM Ratios vs `energy_gain`**breed_energy**

```
import measure_ratios as tst

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

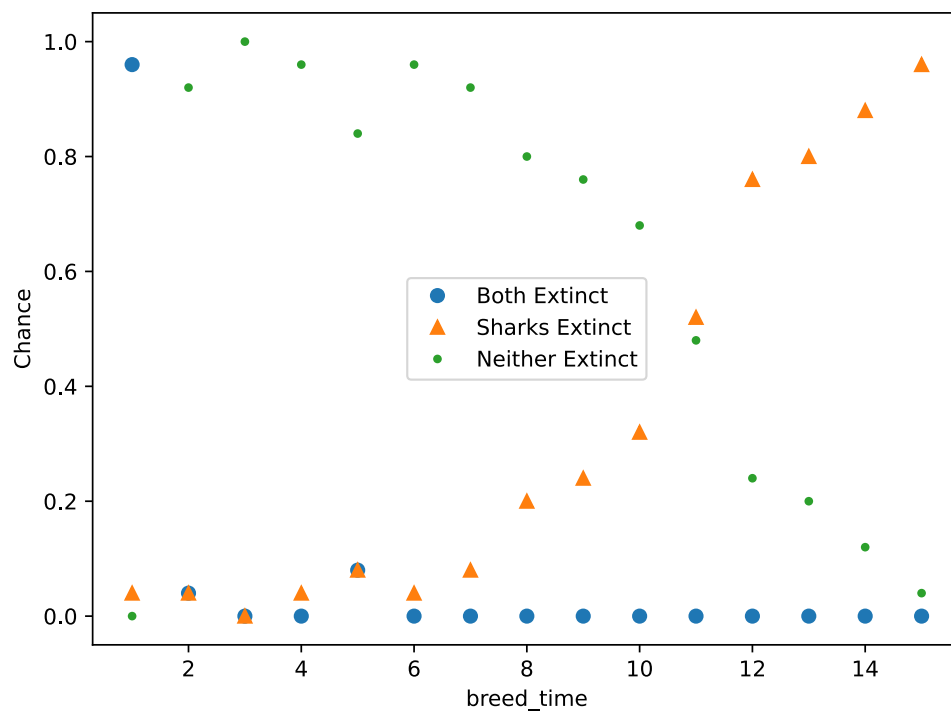

Figure 11: LVM Ratios vs $breed_energy$

Circular Initialization

$breed_time$

```
import measure_outcome_chances as tst

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

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

$energy_gain$

```
import measure_outcome_chances as tst
```

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

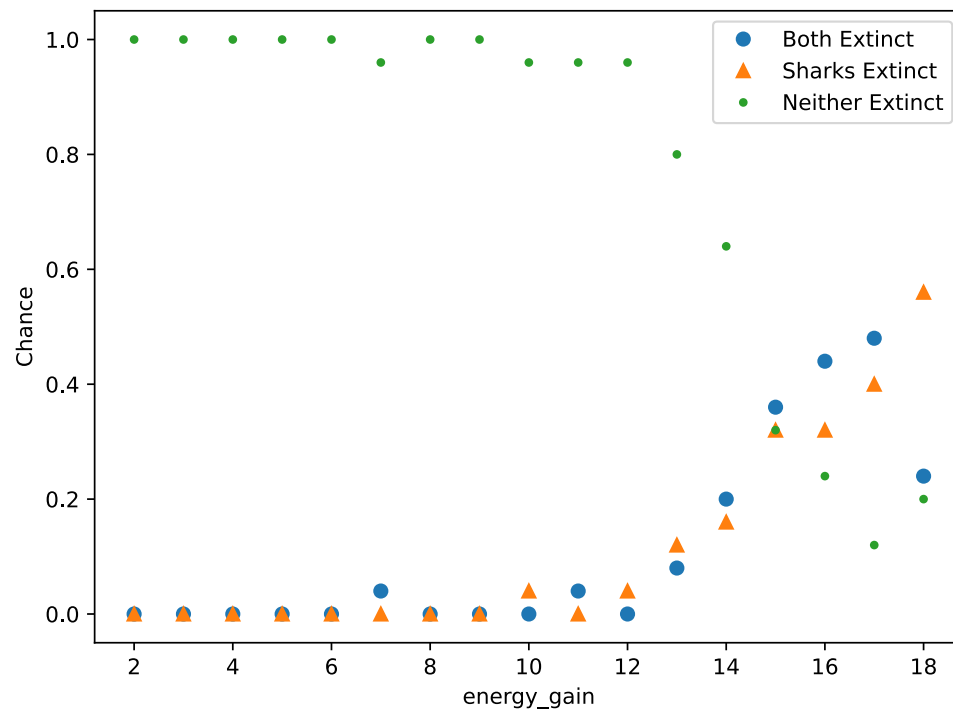


Figure 13: Outcome Chances vs energy_gain (Circular Initialization)

breed_energy

```
import measure_outcome_chances as tst  
  
tst.run_standard_test("breed_energy", True)
```

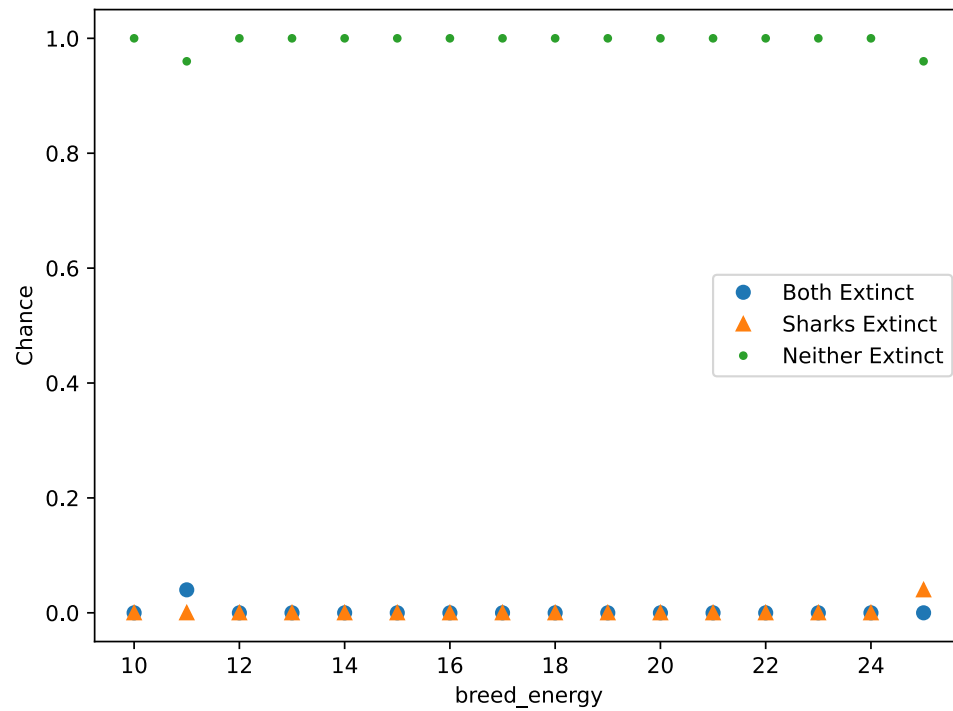


Figure 14: Outcome Chances vs breed_energy (Circular Initialization)

board_area

```
import measure_outcome_chances as tst  
  
tst.run_standard_test("board_area", True)
```

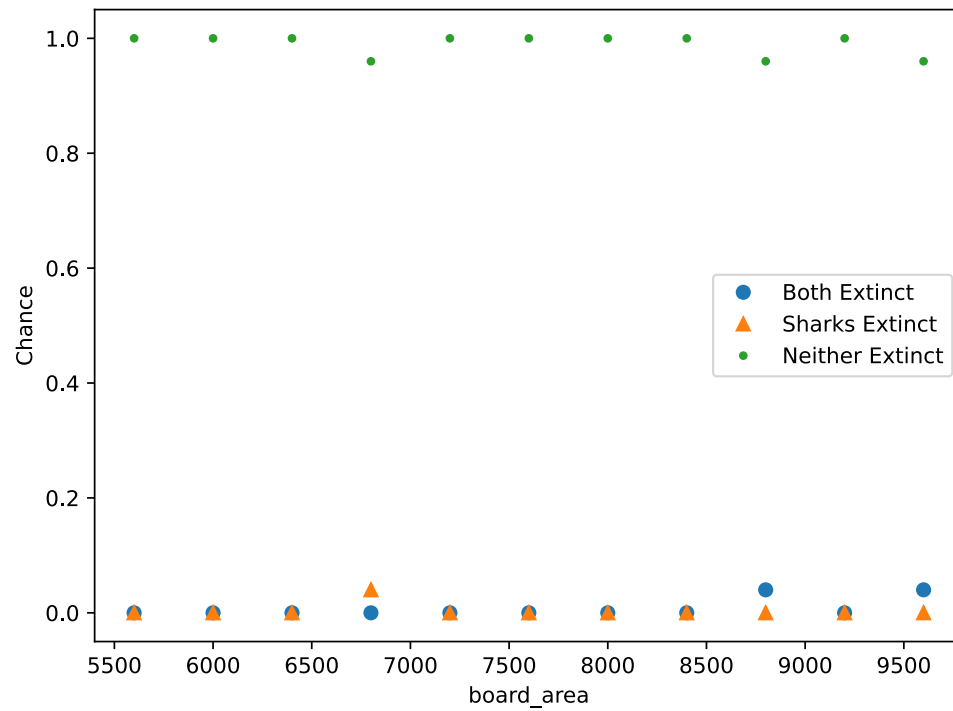


Figure 15: Outcome Chances vs board_area (Circular Initialization)

aspect_ratio

```
import measure_outcome_chances as tst  
tst.run_standard_test("aspect_ratio", True)
```

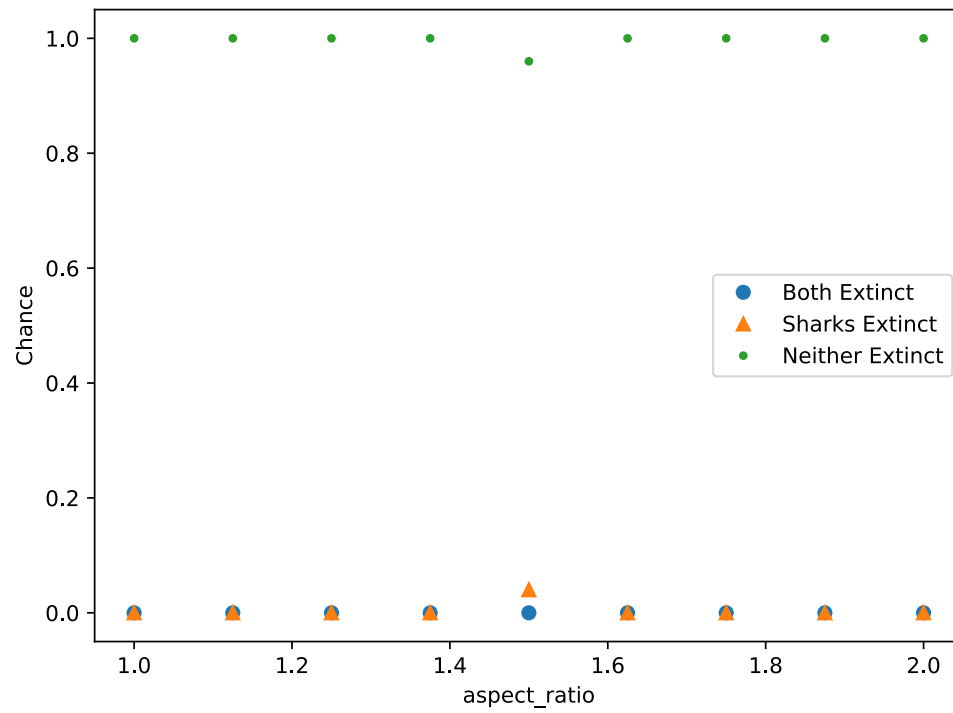


Figure 16: Outcome Chances vs aspect_ratio (Circular Initialization)

initial_fish

```
import measure_outcome_chances as tst  
tst.run_standard_test("initial_fish", True)
```

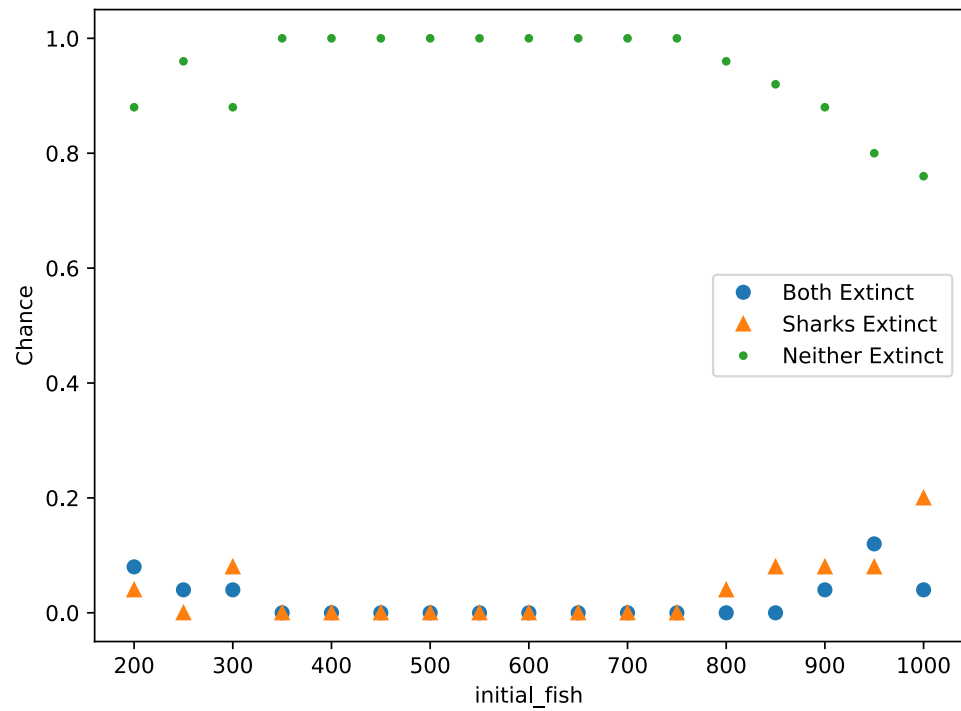


Figure 17: Outcome Chances vs initial_fish (Circular Initialization)

initial_sharks

```
import measure_outcome_chances as tst  
tst.run_standard_test("initial_sharks", True)
```

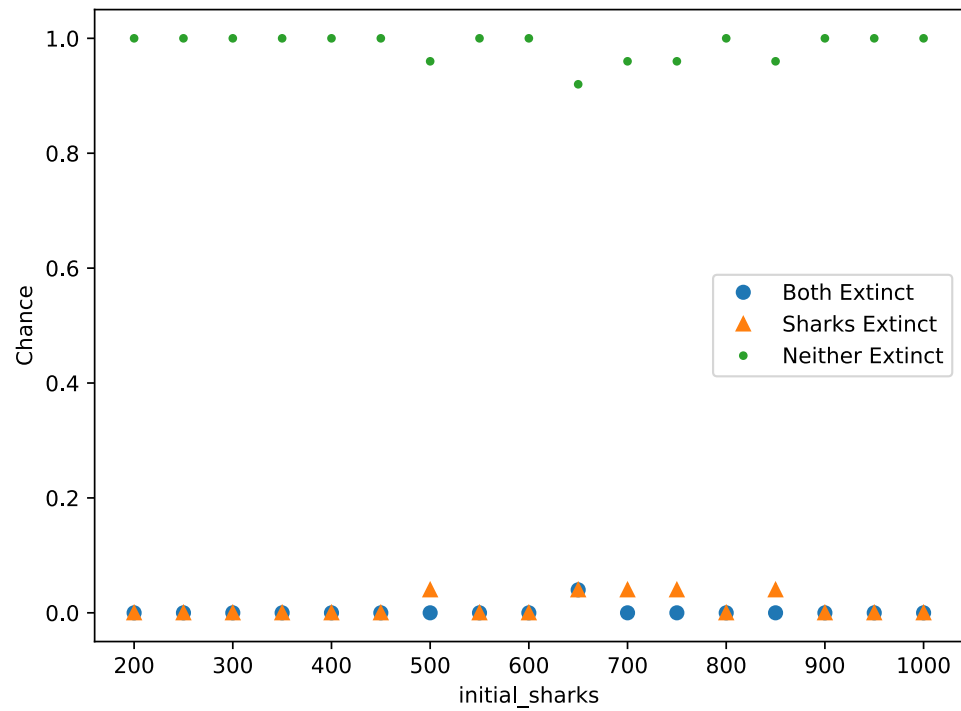


Figure 18: Outcome Chances vs initial_sharks (Circular Initialization)

start_energy

```
import measure_outcome_chances as tst  
tst.run_standard_test("start_energy", True)
```

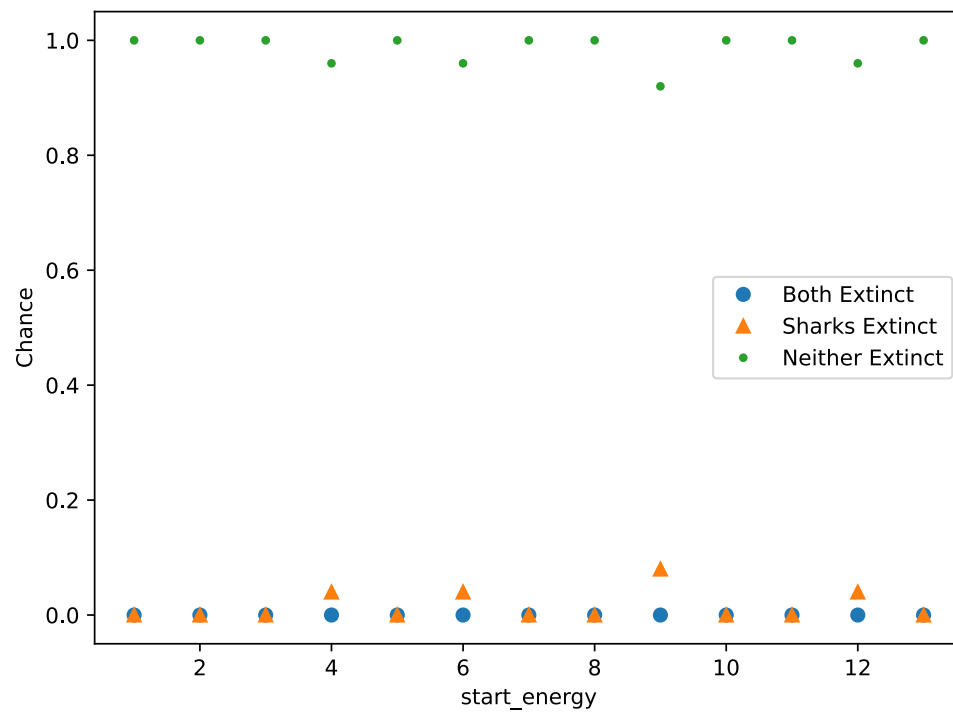


Figure 19: Outcome Chances vs start_energy (Circular Initialization)

Extension