1. Implementation

In our model, we store two NxN arrays: one called $food_grid$, tracking the current amount of food in each cell (capped at 100), and another called $bacteria_grid$, which holds a nonnegative real value denoting the bacteria population in that cell. We begin each time step by applying a logistic growth formula to the food, $f_{t+1} = f_t(1 + g_f(1 - \frac{f_t}{k_f}))$, ensuring it remains below the capacity of 100. With a small probability (1 %), a random "reseeding" event increments the food in a cell by +1, allowing even depleted cells to recover. Next, we model diffusion by letting a fraction d_f of each cell's food flow into its four neighbors (and an equivalent fraction flow in from those neighbors).

Following these food updates, the bacteria consume any available food at a rate $c_b\, x\ (bacteria)$. If the amount of food in a cell meets or exceeds this requirement, the bacteria all survive and the cell's food is reduced accordingly. If not, the bacteria that cannot eat effectively starve, so only $\frac{f}{c_b}$ individuals remain, and the cell's food is set to zero. Next, the bacteria that survive reproduce according to $b_{t+1} = b_t(1 + g_b)$. Finally, we apply a diffusion step for the bacteria as well, allowing a fraction dbd_bdb of each cell's bacterial population to move to its four neighbors. This entire sequence - logistic food growth, reseeding, food diffusion, consumption/starvation, reproduction, and bacteria diffusion - is implemented in the $step$ method to advance the system by one-time step.

2. Tests

The outputs are shown in the code. I analyze the outputs here.

Test 0: Default Small Grid

This test uses moderate parameters and initializes the grid with a mix of low‑to‑medium food (0 to 10) and a small fraction of cells containing bacteria. Over the five steps, the results show that cells initially high in bacteria cause local food depletion and, if consumption exceeds available food, some bacteria starve. Meanwhile, food diffuses from neighboring cells and undergoes logistic growth, so certain cells replenish as the simulation progresses. By the fifth step, we see some cells stabilizing with moderate food and bacteria levels, matching expectations that neither species completely overwhelms nor crashes under these balanced parameters.

Test 1: Minimal Food

Here, we begin with almost no food across the grid (range 0 to 1) but a relatively high fraction (40%) of cells containing bacteria. Unsurprisingly, many bacteria starve rapidly because most cells do not have enough food to support them. We observe the bacteria populations dropping in early steps, while the surviving bacteria are only those lucky enough to be in cells where logistic growth or random reseeding provides enough sustenance. This test confirms that our consumption logic (and the starvation mechanism) behaves as expected in near‑zero‑food conditions.

Test 2: Plentiful Food

In this scenario, each cell starts with a high level of food (30 to 50) and a moderate bacteria fraction. Over five steps, the printouts show that while bacteria consume significant amounts, food never exceeds its 100‑unit cap. The bacteria that do find abundant food reproduce quickly, but also diffuse out. Ultimately, we see most cells retaining well‑above‑zero food, demonstrating that strong initial resources, plus logistic growth and occasional reseeds, keep the system far from starvation. This confirms that even under resource‑rich conditions, food consumption is bounded and does not exceed capacity.

Test 3: No Bacteria

Here, we initialize cells with moderate food but set the bacteria fraction to zero. Since there is no consumption at all, food merely grows logistically and occasionally gains an extra +1 from reseeding. The output shows that every cell's food level steadily rises but remains under the capacity limit. Because no bacteria are present to consume it, these grids converge toward (but never exceed) the maximum. This passes the check that, in the absence of bacterial activity, the food distribution evolves solely by its growth and diffusion rules and respects the 100‑unit cap.

3. Simulations

$g_f =$ $food\ growth\ rate$, $d_f =$ $food\ diffusion\ rate$, $c_b =$ $bacteria\ consumption\ rate$, $g_b =$ $bacteria\ growth\ rate$, $d_b =$ $bacteria\ diffusion\ rate$

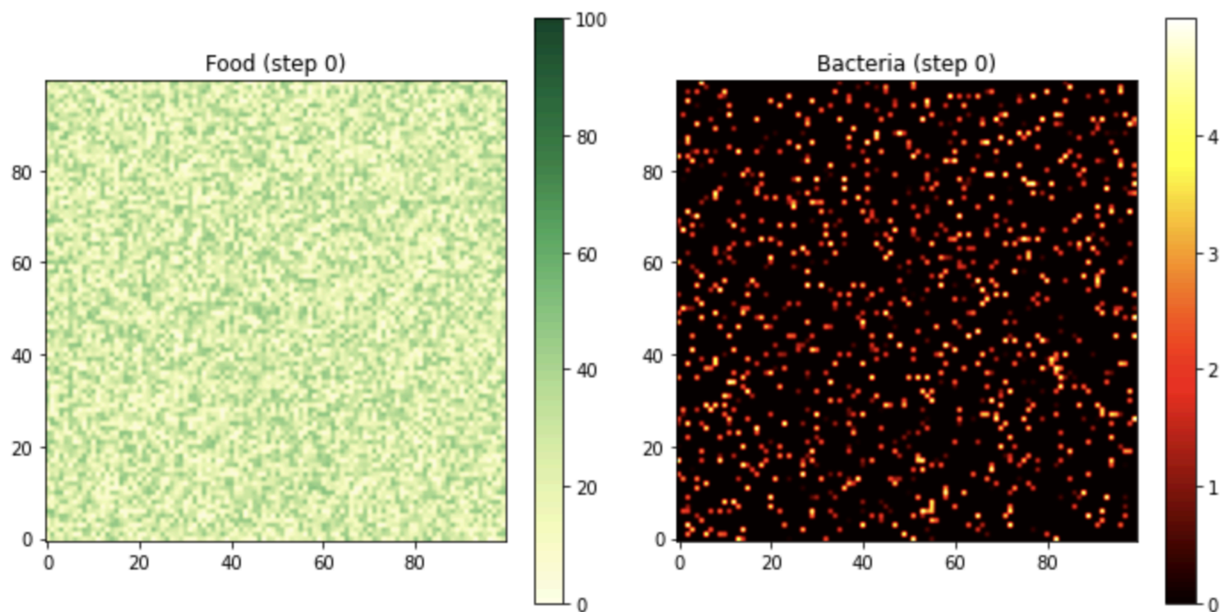


Figure 1: At the very beginning of the simulation, food is randomly scattered between 0 and 50 units per cell (depending on our chosen initialization), and bacteria appear in about 10% of cells with small populations. The left panel shows the initial food distribution, while the right panel shows the initial bacteria distribution.

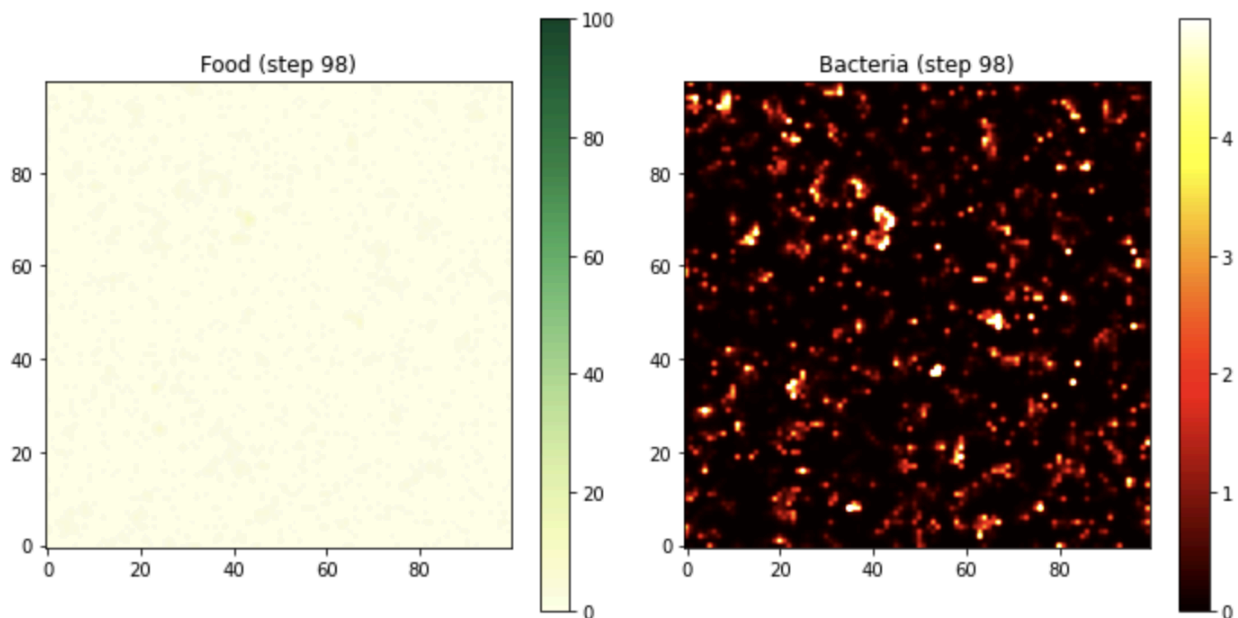


Figure 2: After almost 100 steps with parameters $(N = 100,\ g_f = 0.1,\ d_f = 0.05,\ c_b = 0.2,\ g_b = 0.5,\ d_b = 0.05)$, the food grid has evolved to a mostly near‑uniform background with some patchy regions, while the bacteria grid shows a scattered but persistent population. The left panel depicts the food distribution at step 98, and the right panel depicts the bacteria distribution at the same time.

We create these visualizations by instantiating our $BacteriaGrowthSimulation$ class with the default constructor, which has moderate rates for food/bacteria growth, diffusion, and consumption. After calling $sim.initialize()$, we generate an animation with $animate_simulation_jshtml(sim,\ total_steps = 100,\ steps_per_frame = 1,\ interval = 200)$. Running this code in a Jupyter notebook displays a frame‑by‑frame evolution of how the food and bacteria grids change over time.

From step 0 to step 98, we observe that initially abundant (but unevenly distributed) food gets partly consumed by the bacteria and diffuses to neighboring cells. Some bacteria thrive where food is plentiful and quickly reproduce, creating clusters. In other regions, lack of food forces bacteria to starve. Over time, we see a patchy but relatively stable distribution emerge, with the food recovering through logistic growth and occasional reseeding, while bacteria adapt their population sizes to local resources. By step 98, this interplay results in a broad background of near‑saturated food (just below the capacity of 100) with scattered bacterial "hot spots," each feeding on the replenishing patches of food.

Attempting Wave‑Like Patterns with a Single Bacterium

After experimenting with random initial conditions (Figures 1 and 2), we next created a more controlled setup to provoke wave‑like or spiral patterns. Specifically, we filled every cell of the grid with a uniform amount of food (100 units per cell) and placed only one bacterium (5 units) in the center. The rationale was that, with an initially homogenous resource layer, a single bacterial "seed" might produce outward‑moving consumption waves or swirling patterns when combined with diffusion and reproduction.

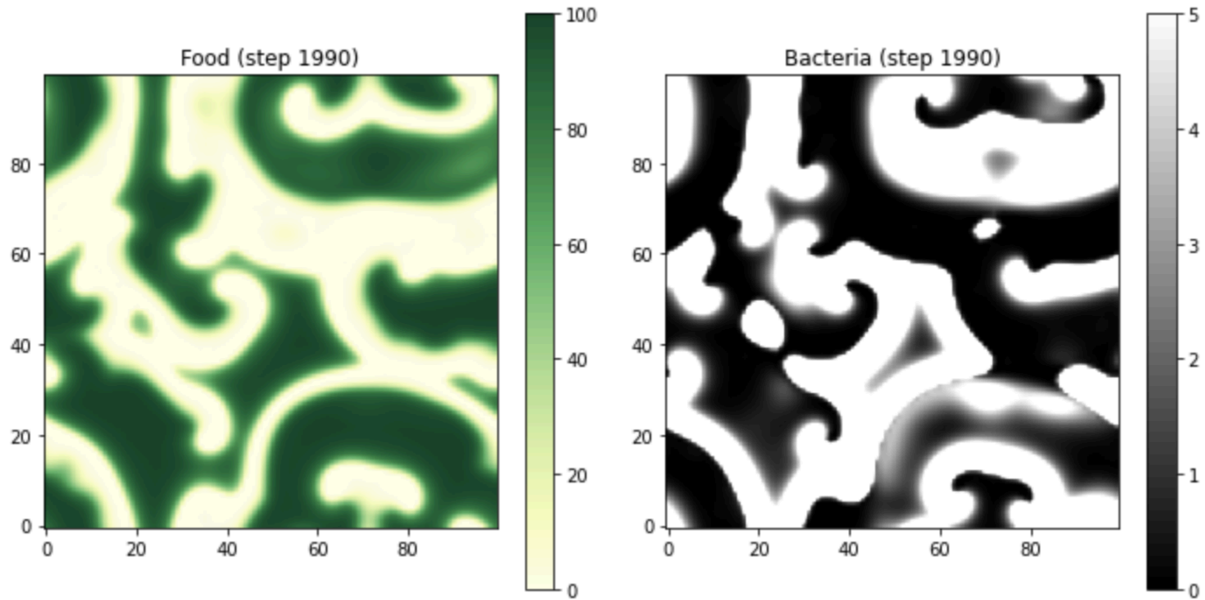


Figure 3: We start with an entire solid green grid corresponding to the uniform 100 units of food, while the bacterium grid is entirely black except for a single bright dot in the center. After around 2,000 steps, we see characteristic wave‑like or "spiral" shapes on the food grid - light areas indicate cells that still hold near‑maximal food, while dark regions indicate recently consumed patches. Meanwhile, the bacteria distribution forms complementary swirls of higher density where food has just been exploited, leaving behind partially starved areas. I changed the color of bacteria to gray because it was difficult to see spirals in red.

This outcome arises because diffusion and bacterial reproduction create a traveling front: the bacteria locally consume food until it depletes, then the hungry wavefront moves (diffuses) into neighboring cells that still have resources. Simultaneously, fresh food regrows logistically where bacteria have moved on. The result is a dynamic interplay of local depletion and regrowth that paints swirling, high‑contrast patterns in the grids. From a modeling standpoint, this highlights how starting from a single "seed" in a uniform resource environment can reveal emergent pattern formation - an effect that is less obvious when initial food and bacteria are randomly scattered.

4. Time‑Series and Spatial Distribution Analysis

Having looked at snapshots of the system under various initial conditions and parameter settings, I next wanted to track how average population levels evolve over time and how food/bacteria end up distributed across all 40,000 cells in our 200×200 grid. Below, I show both time‑series curves of the average food and bacteria, as well as final histograms indicating each cell's state after 10,000 steps.

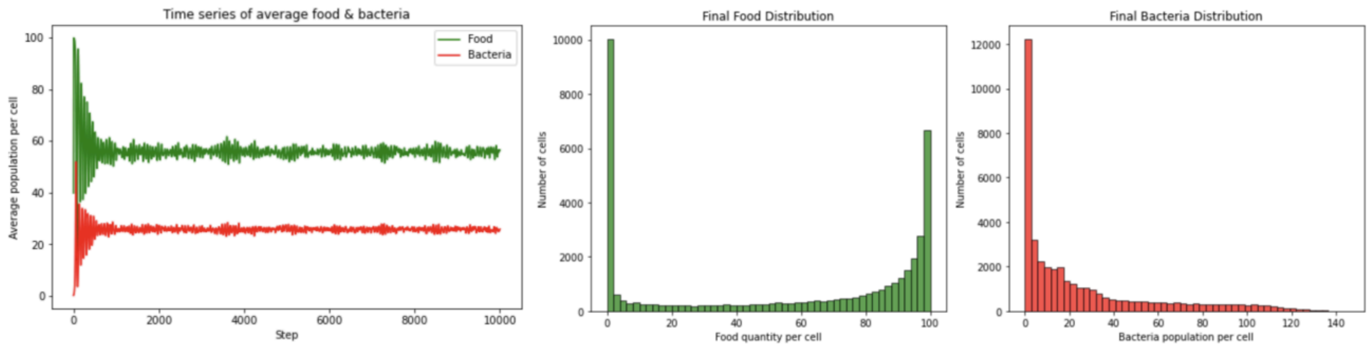


Figure 4: This shows a time series of average food and bacteria, alongside the final distributions across a $200x200$ grid, using parameters ($g_f = 0.9,\ d_f = 0.2,\ c_b = 0.3,\ g_b = 0.1,\ d_b = 0.05$). Because g_f is relatively high compared to the consumption and bacteria growth rates, food replenishes so quickly that cells rarely remain depleted for long. As a result, we observe that the average food settles around a high equilibrium value, and the bacteria population, though sustained, remains moderate. The histogram at the end shows a pronounced cluster near the maximum food capacity (close to 100) for many cells, while most bacteria concentrations remain either low or moderate, reflecting occasional local depletion where strong bacterial clusters form.

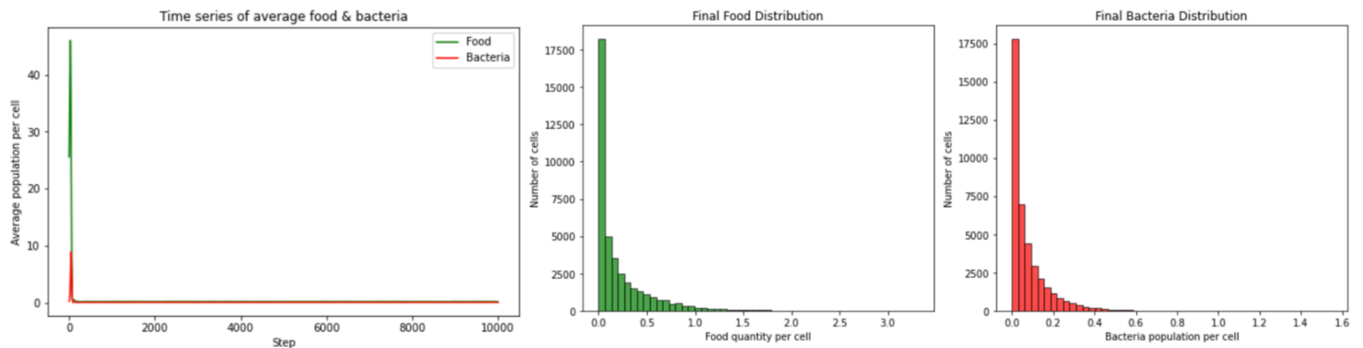


Figure 5: This shows the case ($g_f = 0.05,\ d_f = 0.2,\ c_b = 0.3,\ g_b = 0.1,\ d_b = 0.05$) in which the food growth rate is significantly reduced. With insufficient replenishment, the system experiences a near crash: both food and bacteria rapidly drop almost to zero and remain low for the remainder of the simulation. The final food histogram confirms that most cells end up with very little food, while the bacteria distribution similarly skews near zero. This outcome highlights how, when g_f is too small relative to consumption, bacteria cannot be maintained in any substantial density.

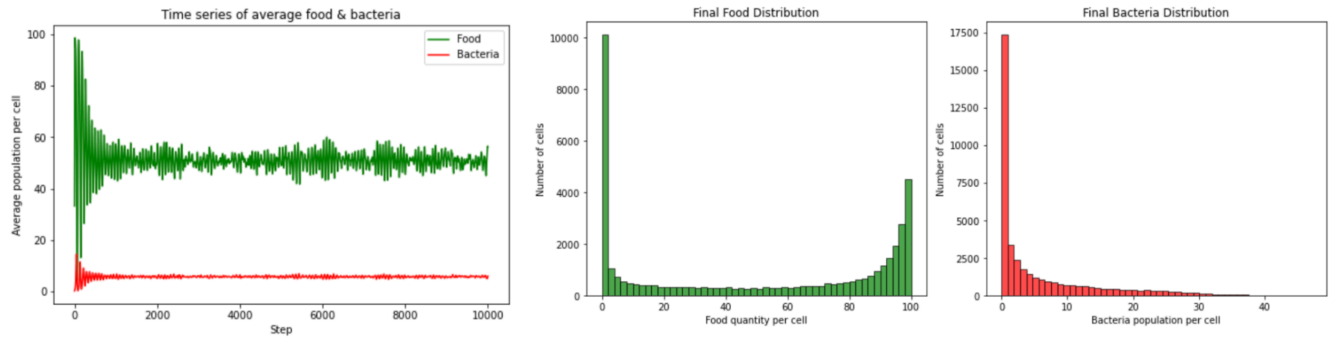


Figure 6: Here ($g_f = 0.5$, $d_f = 0.8$, $c_b = 0.8$, $g_b = 0.1$, $d_b = 0.05$), both the food growth and diffusion increase. Food spreads widely across the grid, yet it is consumed at a substantial rate ($c_b = 0.8$). Early in the run, we see noticeable oscillations as regions become momentarily depleted, then refill via diffusion. Over time, an intermediate balance emerges, with moderately high average food levels and a lower but persistent population of bacteria. The final histograms reflect a widespread. Many cells accumulate food near capacity (thanks to rapid diffusion), but a significant fraction show partial or minimal food due to high local consumption by bacteria. Bacteria remain scattered, with small pockets reaching moderate densities.

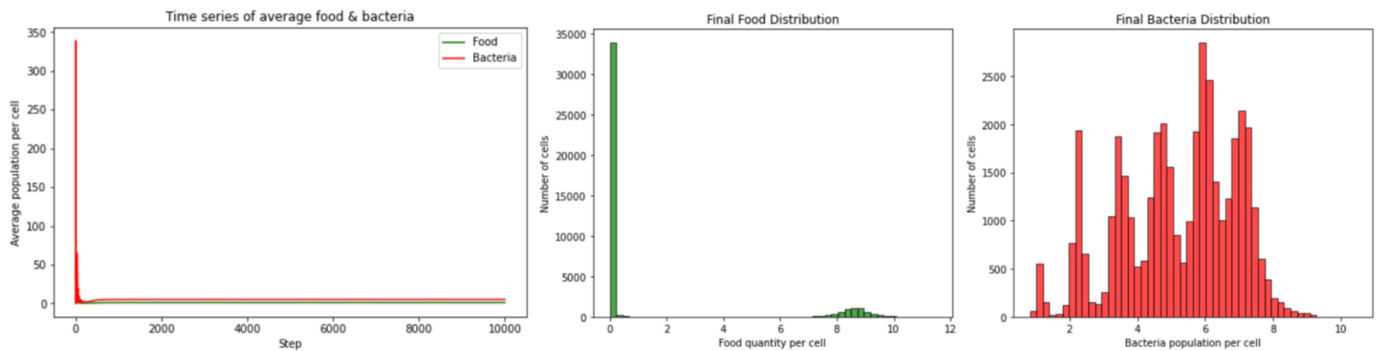


Figure 7: This ($g_f = 0.5$, $d_f = 0.2$, $c_b = 0.2$, $g_b = 0.8$, $d_b = 0.8$) focuses on high bacterial reproduction and diffusion. Early in the simulation, bacteria surge wherever food is sufficiently available. As these local patches of bacteria grow and consume resources, they may collapse if diffusion and logistic growth do not replace the food quickly enough. Meanwhile, bacteria themselves diffuse widely, sometimes recolonizing food-rich areas. The time series often shows notable spikes or wave-like fluctuations as bacterial clusters expand and contract. In the final snapshot, the food histogram might have peaks at both low and high values (reflecting local depletion vs. unexploited cells), while the bacteria histogram tends to have a more spread-out shape, indicating that some cells house dense populations and others have very few bacteria.

In all of these parameter sweeps, each plot shows how the average amount of food and bacteria per cell evolves over 10,000 steps on a $200x200$ grid, along with final histograms of the food and bacteria distributions. Several common patterns emerge. A large food growth rate (g_f) compared to consumption (c_b) and bacterial growth (g_b) generally maintains abundant food, thus stabilizing a moderate-sized bacteria population but rarely allowing it to overrun the system. Conversely, a small g_f can push both species toward near-zero if consumption outpaces replenishment. Diffusion (d_f or d_b) controls the spatial character of these dynamics: higher diffusion yields more uniform distributions and can quickly refill depleted areas, while lower diffusion fosters localized pockets of intense consumption and bacterial growth. Finally, the relative rates of bacterial growth (g_b) and consumption (c_b) determine whether bacteria can surge to high densities or remain at minimal levels. By tuning just these few parameters, one can produce a wide range of outcomes: from high-food, low-bacteria "stable" states, to near-starvation scenarios, or balanced oscillations that may form striking spiral or wave-like patterns in the grid.

5. Parameter Sweeps and Final Population Analysis

In the previous section, we visualized how food and bacteria populations evolve over time under a fixed parameter set by plotting time-series curves and spatial distributions. That approach highlighted the dynamics of the system - how cells transition from their initial states to a (possibly) stable or oscillatory configuration. In this section, we shift our focus to how the final outcome of the simulation depends on a single parameter: rather than watching a time series for one chosen parameter set, we run multiple simulations (each for a large number of steps) under different values of a parameter, then record only the final (converged) average populations of food and bacteria.

We created a function called $param_sweep$ (initial parameters: $(N = 100,\ g_f = 0.1,\ d_f = 0.05,\ c_b = 0.2,\ g_b = 0.5,\ d_b = 0.05)$ that receives the name of the parameter to vary (e.g., g_f or g_b) and a range of values to test. For each value, it re-initializes the $BacteriaGrowthSimulation$, runs it for a specified number of steps (e.g., 2000 or 3000, enough for the system to stabilize), and then measures the mean of each grid. By calling $param_sweep$ with arrays like $np.linspace(0, 1, 21)$, we explore 21 evenly spaced points, ranging from 0.0 to 1.0. This is broad enough to capture changes from minimal to maximal rates or probabilities but still computationally feasible. We then visualize the final means of food and bacteria using a function $plot_param_sweep$, producing straightforward 2D plots: the parameter values on the x-axis, and the resulting mean populations on the y-axis.

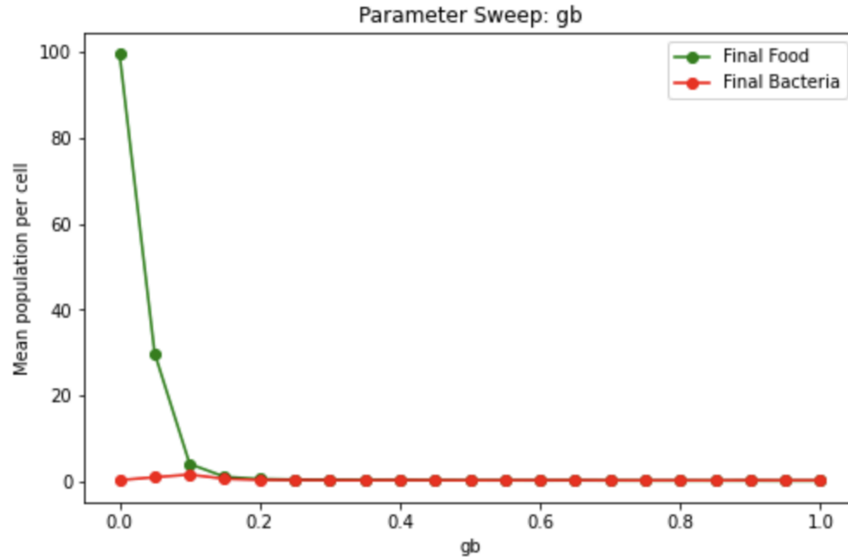


Figure 8: Results for Bacteria Growth (g_b). The figure displays how the final (mean) populations of food (green curve) and bacteria (red curve) change when the bacteria growth rate g_b is swept from 0 to 1. We observe that for extremely small g_b, the food remains quite high while bacteria stay near zero; as g_b increases slightly above zero, the system still quickly depletes most food, driving the final bacteria to negligible levels. Above a certain threshold, the bacteria do not manage to expand sufficiently before consumption limits their growth. Ultimately, the entire system collapses to a near-zero bacteria state across most g_b values, with food likewise dropping for moderate rates. It highlights that in these particular default settings, neither the food nor the bacteria can stably coexist if the bacteria's reproduction rate is non-trivial, unless some other parameter (e.g., food growth) offsets that consumption.

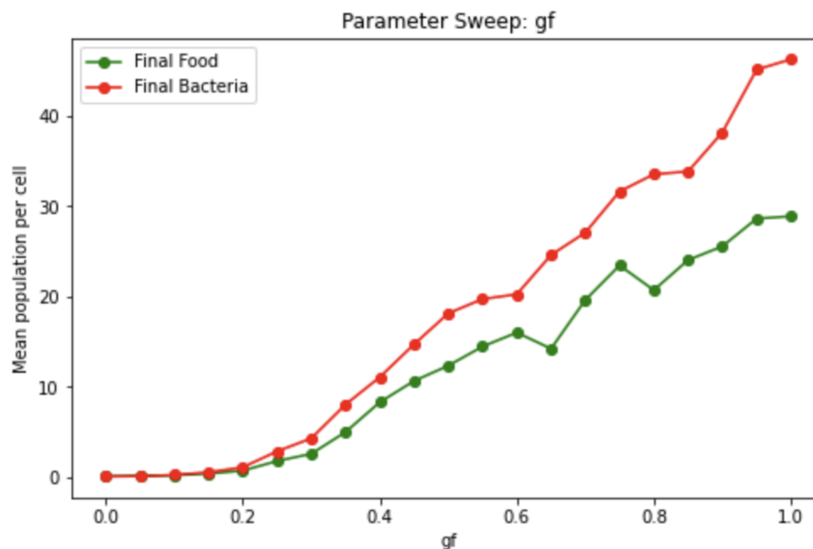


Figure 9: Results for Food Growth (g_f). Figure 2 shows what happens when we sweep the food growth rate g_f from 0 to 1. Unlike the near collapse in the g_b sweep, here we see a smooth upward trend for both final food and bacteria as g_f increases. Around lower values of g_f (below 0.2), the final populations remain small - food does not regenerate fast enough, and bacteria starve. However, as g_f climbs beyond 0.3 or 0.4, more food accumulates, enabling the bacteria population to flourish as well, leading to an overall higher equilibrium in both species by the time $g_f = 1.0$.

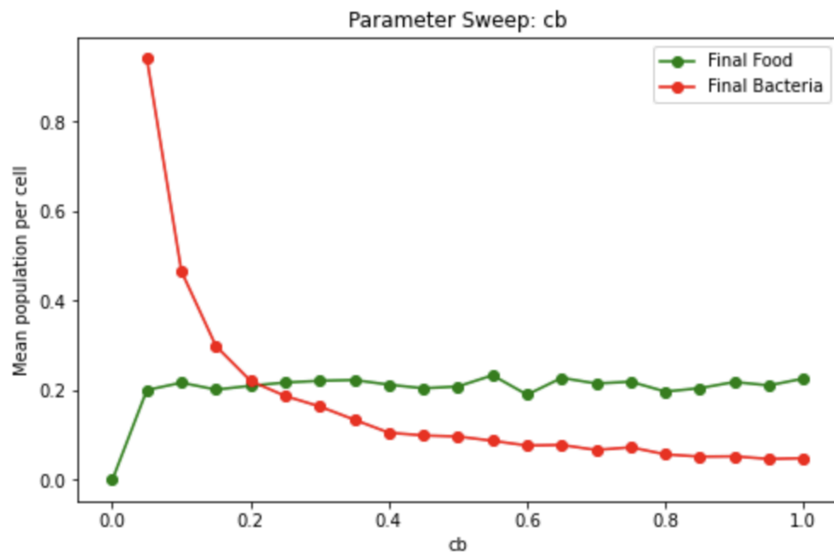


Figure 10: Results for Consumption (c_b). Figure 3 explores the final outcome when we vary the consumption rate c_b from 0 to 1. Initially, at very small c_b, bacteria need little food, so they can expand rapidly, yielding a high final bacteria population. Food is quickly consumed if it's available but never fully collapses because the consumption rate remains small. As c_b increases, bacteria start requiring more resources. They fail to secure enough food in many cells, leading to decreased final bacteria. Interestingly, the final food curve hovers in a moderate range and can even rise a bit for some mid-range c_b values, reflecting partial depletion followed by regrowth. Past a certain point, however, bacteria remain relatively low because they consume too much (or too quickly) and eventually starve in many regions.

I also tested sweeping the diffusion parameters of food (d_f) and bacteria (d_b) in $[0, 1]$, but the outcomes (shown in the final two plots in the code) were less nuanced under our default rates. Both curves typically collapse to near-zero or saturate. I found that near $d_f = 0$ or $d_b = 0$, the system leaves food or bacteria stuck in local patches without significant mixing, sometimes

yielding large disparities. Conversely, at $d_f = 1$ or $d_b = 1$, strong diffusion rapidly homogenizes each grid, leading either to near-uniform depletion (if consumption is high enough) or uniform persistence (if growth dominates). Even intermediate diffusion values (like $d_b = d_b = 0.7$ - I made separate graphs to test this out) produced similar end states in these default conditions, suggesting that other parameters (such as g_f or c_b) overshadow diffusion's role in shaping final outcomes (as we saw in Figures 8 and 9).

In the last section, we tracked how the system's state evolves step by step under a single parameter set, observing whether it stabilizes or oscillates over time. This time-series perspective focuses on when and how cells transition from their initial conditions to a final configuration. By contrast, the parameter sweeps reported here show what final states the system settles into when one specific parameter is varied across a range of possible values - while all other parameters are kept at their default. Where time-series analyses answer “How does the system behave from step 0 to step 10,000 for a fixed parameter set?”, the sweeps address “How do the end-point populations shift if we tweak one parameter from 0.0 to 1.0 (or beyond) and allow each scenario to run long enough to converge?”

From these sweeps, it is clear that food growth (g_f) and consumption (c_b) strongly determine whether the system settles into a near-zero population “crash” or a moderately thriving coexistence. In contrast, diffusion (d_f and d_b) showed relatively subtle effects under the same default assumptions: either homogenizing the grid (at high diffusion) or leaving resources and bacteria in localized patches (at low diffusion), without drastically altering the fundamental population levels. Overall, these results highlight how the interplay of growth, consumption, and diffusion drives the final distribution of food and bacteria, ranging from near-desert outcomes to stable, partially populated states.

6. Theoretical Analysis

In this section, we aim to derive a simple mathematical relationship that connects certain model parameters - particularly the food growth rate g_f, the bacteria consumption rate c_b, and the bacteria reproduction rate g_b - to observable outcomes such as the average amounts of food and bacteria once the system has stabilized. This theoretical viewpoint is valuable because it offers insights into the model's behavior without exhaustively running large-scale simulations for every parameter setting. In other words, if we can find a reasonable formula or approximate equilibrium condition, we might identify thresholds (e.g., a minimal g_f needed to avoid population collapse) and stable states (e.g., a high-food, moderate-bacteria equilibrium), all

from a few equations. This approach complements our simulation-based approach in the last two sections by providing a higher-level understanding of how parameters interact to shape the final outcome.

Simplified Mathematical Relationship

We begin by specifying which parameters and observable quantities we want to tie together. The main parameters in our model are g_f (food growth rate), which drives the logistic growth of food in each cell, c_b, (bacterial consumption rate), governing how quickly bacteria consume available food, and g_b (bacteria growth rate), indicating how quickly the bacteria population reproduces after eating.

On the output side, we focus on F^* (the final or "equilibrium" average food across all cells) and B^* (the final average bacteria). Although local cell-to-cell variations exist, a mean-field approximation captures the broad trends by treating every cell as if it experiences the same average conditions.

To formalize this mean-field view, let $F(t)$ represent the average food across all cells at time t and $B(t)$ represent the average bacteria. We ignore spatial heterogeneities and random reseeding for simplicity, writing approximate difference equations that mirror the logistic growth, consumption, and reproduction steps:

$$F_{t+1} \approx F_t + g_f F_t (1 - \frac{F_t}{100}) - c_b B_t \qquad \text{(food update)}$$

$$B_{t+1} \approx B_t + g_b B_t - \ . \ . \ . \qquad \text{(bacteria update)}$$

In the first equation, the term $g_f F_t (1 - F_t/100)$ models logistic growth of food (capped at 100), while $c_b B_t$ approximates total consumption by the bacteria. In our partial-starvation approach, if the consumption needed exceeds newly grown food, we reduce F to zero and allow only the fraction of bacteria that managed to eat to survive. This lumps local, cell-by-cell starvation into a single global fraction, which can be harsher or more forgiving than the actual simulation's spatially distributed survival. The second equation (for $B_{t+1}-$ would also include a starvation effect if not enough food is present. However, for a first-pass mean-field model, we might treat consumption simply as a net loss or factor it into the term . . .. By combining or refining these approximate equations, we can look for equilibrium conditions:

$$F_{t+1} = F_t = F^*, \qquad B_{t+1} = B_t = B^*$$

Solving for (F^*, B^*) in terms of g_f, c_b, g_b reveals which combinations of growth and consumptions allow for stable coexistence. If it turns out that a large consumption rate c_b and a small food growth rate g_f force the system to a trivial equilibrium $F^* \approx 0$, $B^* \approx 0$, then we can interpret that as a "collapse" threshold. On the other hand, if g_f is sufficiently large relative to c_b or if g_b is moderate, the system may reach a non-trivial equilibrium (F^*, B^*) where both populations stay above zero. In fact, from a net perspective, if g_f/c_b is too small, total consumption outstrips logistic replenishment, guaranteeing a crash. Conversely, once g_f/c_b exceeds some critical ratio, the bacteria population can persist or grow. This ratio is approximate - g_f also matters - but highlights that balancing food inflow and consumption is central to stable coexistence.

Even if the exact solution for (F^*, B^*) proves complex, reasoning about these approximate equations clarifies why, for instance, a minimal g_f is needed to sustain bacteria, or why increasing c_b can destroy that equilibrium. Such a simplified analysis, while not capturing every spatial detail (such as local starvation patches or diffusion differences between neighboring cells), is often enough to predict major phase transitions, like moving from a near-desert outcome to a stable coexistence, without performing dozens of simulations.

Comparison to Empirical Simulation Results

Having established a mean-field (MF) model in which we treat the entire grid as a single "average cell" (with average food F and average bacteria B), we next compared the MF predictions against the actual simulation's final states under four different parameter configurations. These configurations varied the food growth rate (g_f), the bacterial growth rate (g_b), or the bacteria consumption rate (c_b), all while the simulation itself included local diffusion, partial starvation, and random reseeding. Our goal here is to see whether the simplified equations truly mirror the more detailed spatial model and to glean further insight into thresholds and stable states that the mean-field approach might reveal.

In practice, we observed that the mean-field code often produced either a collapse to $(F^*, B^*) = (0, 0)$ for all parameter values, or a saturated equilibrium (e.g., very large F^* or B^*) for every parameter value, whereas the simulation displayed more nuanced outcomes - varying from near-zero populations, to modest coexistence, to surprisingly large surges in food or bacteria.

This discrepancy arises because our mean-field approach lumps the entire system into a single equation. If, at any step, the bacteria's total consumption demands exceed the newly grown food, the MF model effectively starves almost all bacteria at once. Subsequent steps cannot recover, and the system remains at ($F = 0,\ B = 0$). By contrast, the actual simulation has local patches where food remains abundant, or pockets of bacteria that find just enough resources to survive. Over multiple spatial cells and repeated reseeding, these pockets can regrow, diffusing outward into areas of the grid that have had time to recover. Hence, the real system might persist with moderate or even high average bacteria - despite the mean-field logic dictating a complete crash.

Conversely, we also see parameter sets where the mean-field saturates around high ($F^*,\ B^*$), whereas the real simulation hovers at moderate or even lower values. There, the single-cell perspective ignores local wavefronts of consumption and diffusion that dynamically keep overall food from saturating, or it neglects the partial starvation that might occur at the cell level. Additionally, random "+1" reseeding in the simulation can create local bursts of resource that keep the bacteria from uniform exhaustion. In short, the spatial simulation tends to produce more complicated "patchy" or wave-driven patterns that defy a one-number-per-step approach.

Still, the mean-field approximation usefully shows a fundamental threshold competition. If g_f is sufficiently large relative to c_b and g_b, one may expect a nonzero equilibrium of both food and bacteria. If consumption or bacterial growth outstrips the net food replenishment in the mean sense, the system collapses.

In the real simulation, local mechanisms soften or shift these thresholds, but broadly the same idea holds: above some minimal g_f, the population surges; below that, it crashes. Our partial-starvation equations capture some aspects of these transitions - highlighting how a small difference in g_f can cause a qualitative change from near-desert states to thriving coexistence.

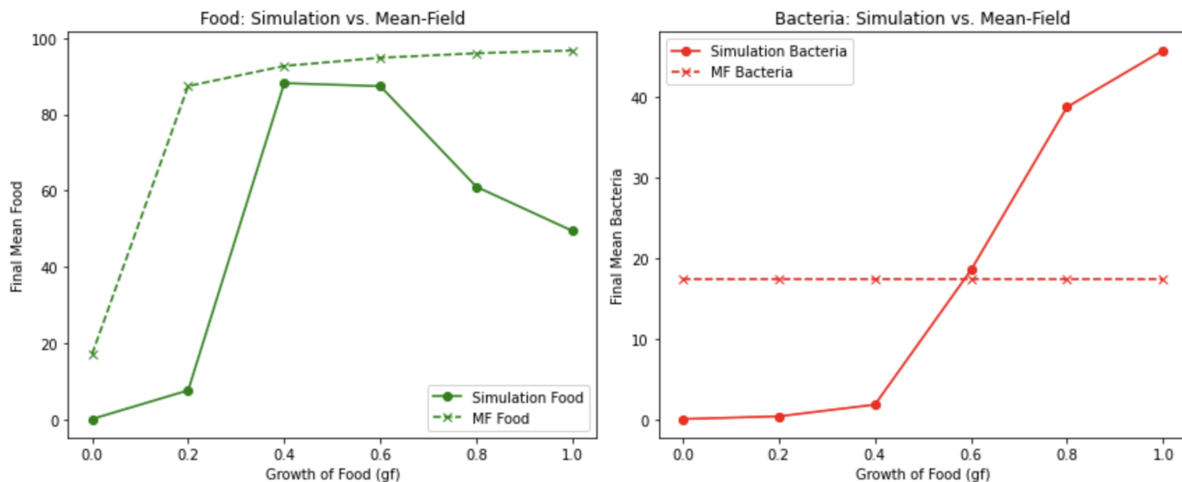


Figure 11: I adjusted the parameters 4 times to see the difference (this exact figure is of the first parameters. I tested other 3 parameter changes as well but did not include all the figures). I first varied the food growth rate g_f at fixed c_b and g_b, observing that both the MF and the actual simulation recognized a threshold beyond which bacteria thrived. However, the MF often collapsed or jumped to near‑capacity, while the simulation maintained a more moderate outcome. When we increased the bacteria growth rate g_b, the MF would sometimes starve everything at once, whereas the detailed simulation formed patchy or wave‑like patterns allowing some bacteria to persist. In a third sweep, raising the consumption rate c_b again drove the MF to $(F^*, B^*) = (0, 0)$ for most settings, while the simulation occasionally found localized pockets of survival. Finally, using shorter or longer max iterations in the MF approach shifted whether it converged to a very high equilibrium or zero, but rarely produced the same moderate results seen in the simulation.

Overall, the MF equations reveal how a balance of logistic food growth, consumption, and reproduction can tip the system into crash or coexistence, yet the real simulation's local diffusion and partial starvation smooth out these extremes - leading to less abrupt but still parameter‑sensitive outcomes.

Examining the four parameter sweeps we performed, we indeed see that the simulation typically has a lower or more moderate final food and bacteria than the fully "optimistic" mean‑field would predict, or else it avoids the harsh all‑or‑nothing crash predicted by the MF code. This underscores how ignoring spatial granularity can lead to more extreme equilibria. Nonetheless, we can glean phase‑transition‑like insights from the mean‑field: for instance, once g_f passes $\sim 0.4 - 0.5$ in our runs, the simulation's bacteria population begins to grow significantly, roughly mirroring the mean‑field's statement that "enough food growth triggers stable coexistence."

Throughout this report, we have progressed from the detailed implementation and testing of our bacteria-food simulation, through dynamic time-series and spatial distribution analyses, to systematic parameter sweeps that reveal the long-term equilibria of the system. These empirical sections illustrate how local interactions, diffusion, and random reseeding produce complex patterns and oscillatory behaviors in a spatially extended system. In parallel, our theoretical analysis - with its simplified mean-field model - provides a higher-level perspective by distilling these dynamics into key thresholds and equilibrium conditions. Together, the empirical and theoretical approaches form a coherent narrative, demonstrating that while local effects moderate the extremes predicted by a global average, the balance between logistic food growth and bacterial consumption/reproduction ultimately governs whether the system collapses or sustains stable coexistence.

AI Statement: I used an AI tool to assist in debugging my code and refining my approach. For example, when my mean-field model was always returning zero, the AI helped me identify that I needed to store the iterative values in lists to capture the full evolution of the system, rather than just the final step. Additionally, the tool provided suggestions on how to better compare the theoretical predictions with the empirical simulation results, clarifying concepts like partial starvation and threshold effects. I used the AI in a similar way for aspects of the empirical simulation, treating it as a research resource to inspire and validate my own coding and analytical decisions.

okiecfzkt

February 27, 2025

0.1 1. Implementation

[3]:
```python
import numpy as np

class BacteriaGrowthSimulation:
    def __init__(self,
                 N=100,
                 gf=0.1,    # food growth rate
                 df=0.05,   # food diffusion rate
                 cb=0.2,    # bacteria consumption rate
                 gb=0.5,    # bacteria growth rate
                 db=0.05):  # bacteria diffusion rate
        """
        Initializes a simulation object with grid size N and model parameters:
        gf (food growth), df (food diffusion), cb (bacteria consumption),
        gb (bacteria growth), db (bacteria diffusion).
        """
        self.N = N
        self.gf = gf
        self.df = df
        self.cb = cb
        self.gb = gb
        self.db = db

        # Food capacity is fixed
        self.food_capacity = 100

        # Arrays for storing the state of the system
        self.food_grid = None
        self.bacteria_grid = None

    def initialize(self,
                   initial_food_range=(0, 50),
                   initial_bacteria_fraction=0.1):
        """
        Randomly initializes the grids:
          - food_grid in [initial_food_range[0], initial_food_range[1]]
          - a fraction of cells get a small random bacteria population
```

```python
        """
        # Food grid: random uniform distribution
        self.food_grid = np.random.uniform(initial_food_range[0],
                                           initial_food_range[1],
                                           size=(self.N, self.N))

        # Bacteria grid: mostly zeros, except for a fraction of cells
        self.bacteria_grid = np.zeros((self.N, self.N))
        total_cells = self.N * self.N
        num_bacteria_cells = int(total_cells * initial_bacteria_fraction)
        chosen_cells = np.random.choice(total_cells,
                                        size=num_bacteria_cells,
                                        replace=False)

        for cell_index in chosen_cells:
            row = cell_index // self.N
            col = cell_index % self.N
            self.bacteria_grid[row, col] = np.random.uniform(0.1, 5.0)

    def step(self):
        """
        Executes a single time-step of the simulation in the following order:
          1. Food logistic growth
          2. Food reseeding
          3. Food diffusion
          4. Bacteria consumption (and starvation if insufficient food)
          5. Bacteria reproduction
          6. Bacteria diffusion
        """
        # 1. Food growth (logistic, capped at self.food_capacity)
        self.food_grid *= (1.0 + self.gf *
                           (1.0 - self.food_grid / self.food_capacity))
        self.food_grid = np.clip(self.food_grid, 0, self.food_capacity)

        # 2. Food reseeding (probability 0.01 to gain +1 food)
        reseed_matrix = np.random.random((self.N, self.N))
        self.food_grid[reseed_matrix < 0.01] += 1
        self.food_grid = np.clip(self.food_grid, 0, self.food_capacity)

        # 3. Food diffusion (fraction df flows to neighbors)
        food_out = self.df * self.food_grid
        self.food_grid -= food_out
        incoming_food = (np.roll(food_out, 1, axis=0) +
                         np.roll(food_out, -1, axis=0) +
                         np.roll(food_out, 1, axis=1) +
                         np.roll(food_out, -1, axis=1)) * 0.25
        self.food_grid += incoming_food
```

```
        # 4. Consumption
        food_needed = self.cb * self.bacteria_grid
        enough_food_mask = (self.food_grid >= food_needed)

        # If enough food is available
        self.food_grid[enough_food_mask] -= food_needed[enough_food_mask]

        # If not enough food is available
        not_enough_food_mask = ~enough_food_mask
        self.bacteria_grid[not_enough_food_mask] = (
            self.food_grid[not_enough_food_mask] / self.cb
        )
        self.food_grid[not_enough_food_mask] = 0.0

        # 5. Bacteria reproduction
        self.bacteria_grid *= (1.0 + self.gb)

        # 6. Bacteria diffusion
        bacteria_out = self.db * self.bacteria_grid
        self.bacteria_grid -= bacteria_out
        incoming_bacteria = (np.roll(bacteria_out, 1, axis=0) +
                             np.roll(bacteria_out, -1, axis=0) +
                             np.roll(bacteria_out, 1, axis=1) +
                             np.roll(bacteria_out, -1, axis=1)) * 0.25
        self.bacteria_grid += incoming_bacteria
```

0.2 2. Tests

[76]:
```
def test_default_small_grid():
    """
    Test 0 (original test):
    Creates a small 5x5 simulation, runs it for a few steps, and prints
    partial snapshots of the food and bacteria grids for debugging.
    """
    print("\n=== TEST: Default Small Grid ===")
    sim = BacteriaGrowthSimulation(
        N=5,
        gf=0.1,   # food growth rate
        df=0.05,  # food diffusion rate
        cb=0.2,   # bacteria consumption rate
        gb=0.5,   # bacteria growth rate
        db=0.05   # bacteria diffusion rate
    )
    sim.initialize(initial_food_range=(0, 10), initial_bacteria_fraction=0.2)

    print("Initial food grid:\n", sim.food_grid)
```

```
    print("Initial bacteria grid:\n", sim.bacteria_grid)
    print("-"*50)

    for step_number in range(5):
        sim.step()
        print(f"After step {step_number+1} - Food:\n{sim.food_grid}")
        print(f"After step {step_number+1} - Bacteria:\n{sim.bacteria_grid}")
        print("-"*50)

def test_minimal_food():
    """
    Test 1: Minimal initial food, ensures bacteria starve quickly.
    """
    print("\n=== TEST: Minimal Food ===")
    sim = BacteriaGrowthSimulation(
        N=5,
        gf=0.1,
        df=0.05,
        cb=0.2,
        gb=0.5,
        db=0.05
    )
    # Initialize with almost zero food, but some bacteria
    sim.initialize(initial_food_range=(0, 1), initial_bacteria_fraction=0.4)

    for step_number in range(5):
        sim.step()
        print(f"After step {step_number+1} - Food:\n{sim.food_grid}")
        print(f"After step {step_number+1} - Bacteria:\n{sim.bacteria_grid}")
        print("-"*50)

def test_plentiful_food():
    """
    Test 2: Plenty of initial food, checks that the bacteria expand but
    do not exceed the max food capacity (100) in each cell.
    """
    print("\n=== TEST: Plentiful Food ===")
    sim = BacteriaGrowthSimulation(
        N=5,
        gf=0.1,
        df=0.05,
        cb=0.2,
        gb=0.5,
        db=0.05
    )
    # Initialize with high food values, moderate bacteria
    sim.initialize(initial_food_range=(30, 50), initial_bacteria_fraction=0.3)
```

```
    for step_number in range(5):
        sim.step()
        print(f"After step {step_number+1} - Food:\n{sim.food_grid}")
        print(f"After step {step_number+1} - Bacteria:\n{sim.bacteria_grid}")
        print("-"*50)

def test_no_bacteria():
    """
    Test 3: Start with no bacteria at all. Food should just grow logistically
    (and occasionally reseed), but remain under or at capacity.
    """
    print("\n=== TEST: No Bacteria ===")
    sim = BacteriaGrowthSimulation(
        N=5,
        gf=0.1,
        df=0.05,
        cb=0.2,
        gb=0.5,
        db=0.05
    )
    # Initialize with some food but zero bacteria fraction
    sim.initialize(initial_food_range=(5, 10), initial_bacteria_fraction=0.0)

    for step_number in range(5):
        sim.step()
        print(f"After step {step_number+1} - Food:\n{sim.food_grid}")
        print(f"After step {step_number+1} - Bacteria:\n{sim.bacteria_grid}")
        print("-"*50)

def run_all_small_grid_tests():
    """
    Runs all four small-grid tests (default test + minimal food + plentiful␣
 ↪food + no bacteria).
    """
    test_default_small_grid()
    test_minimal_food()
    test_plentiful_food()
    test_no_bacteria()

run_all_small_grid_tests()
```

```
=== TEST: Default Small Grid ===
Initial food grid:
 [[4.37263326 9.84662145 2.36798286 3.53301865 1.8366741 ]
 [4.20788149 1.14458648 1.99166776 5.57878099 3.83825415]
```

```
 [1.48446092 8.67146843 1.88692287 4.00343366 7.07198275]
 [1.09986713 6.33150154 7.80253898 5.68980118 3.37724686]
 [6.80579719 6.25089011 0.67980986 7.69256253 8.67964826]]
Initial bacteria grid:
 [[0.         4.42706919 0.         0.         0.        ]
 [0.         0.         0.         0.         0.        ]
 [4.19491299 3.16551041 3.24426023 0.         0.        ]
 [0.         0.         0.         0.         0.        ]
 [0.         0.         4.59994363 0.         0.        ]]
--------------------------------------------------
After step 1 - Food:
[[4.87376679 9.5057548  2.68849464 3.9317004  3.14542272]
 [4.52899724 1.53229304 2.22795565 5.98345702 4.26526295]
 [0.99783808 8.505711   1.6265993  4.44502665 7.51682675]
 [1.39444806 6.90372164 8.29544703 6.22778548 3.82634367]
 [7.34691267 6.8181387  0.11948149 8.23650924 9.28070339]]
After step 1 - Bacteria:
[[0.08300755 6.30857359 0.16925649 0.         0.        ]
 [0.07865462 0.14236087 0.06082988 0.         0.        ]
 [6.03710433 4.65033684 4.68242415 0.06082988 0.07865462]
 [0.07865462 0.05935332 0.14707882 0.         0.        ]
 [0.         0.16925649 6.55491968 0.08624894 0.        ]]
--------------------------------------------------
After step 2 - Food:
[[5.38898204 8.80375947 2.98441632 4.36817292 3.58310576]
 [4.85746981 1.90875701 2.47340589 6.42243301 4.72953524]
 [0.13423997 8.04108144 1.08149378 4.90607152 7.97110462]
 [1.70043282 7.49880922 8.77740567 6.81009735 4.31809998]
 [7.93181355 7.37283405 0.         8.79264489 9.92481461]]
After step 2 - Bacteria:
[[2.38046284e-01 9.00029014e+00 4.05663275e-01 4.79072688e-03
  1.55639151e-03]
 [2.29504195e-01 4.10959141e-01 1.80320856e-01 2.28112048e-03
  2.94954820e-03]
 [8.69449181e+00 6.83150329e+00 6.76468708e+00 1.75952805e-01
  2.26419098e-01]
 [2.26391412e-01 1.79178358e-01 3.43542110e-01 5.51545584e-03
  2.94954820e-03]
 [6.20472480e-03 4.05635589e-01 3.43425304e+00 1.67951205e-01
  1.61716768e-03]]
--------------------------------------------------
After step 3 - Food:
[[ 5.90024928  7.56779648  3.24296059  4.84341262  4.06538331]
 [ 5.18677228  2.23926569  2.71260617  6.89746685  5.23272814]
 [ 0.          7.13301049  0.10668584  5.37330447  8.42955442]
 [ 2.01309524  8.10052415  9.24081056  7.43734728  4.85455819]
 [ 8.56074844  7.90260832  0.          9.36939651 10.6139875 ]]
After step 3 - Bacteria:
```

```
[[5.12420119e-01 1.28527942e+01 7.86041463e-01 1.76540106e-02
  6.85667777e-03]
 [4.70392836e-01 8.90147123e-01 3.99149544e-01 1.00758580e-02
  1.28236213e-02]
 [1.01063994e+01 1.00039204e+01 9.78089132e+00 3.81962174e-01
  4.57182139e-01]
 [4.57264201e-01 4.01711768e-01 6.55593413e-01 2.08044434e-02
  1.28270399e-02]
 [2.51859288e-02 7.86007102e-01 2.74142344e+00 2.75299044e-01
  5.65437400e-03]]
--------------------------------------------------
After step 4 - Food:
[[ 6.37701396  5.54714049  3.42854592  5.35767872  4.59439207]
 [ 5.48356922  2.46621202  2.91875196  7.40928121  5.77556172]
 [ 0.          5.54763203  0.          5.82358448  8.86152879]
 [ 2.30219353  8.68131408  9.65800626  8.10902183  5.43681328]
 [ 9.23205985  8.37239311  0.          9.96497314 11.35013628]]
After step 4 - Bacteria:
[[9.80609224e-01 1.83710057e+01 1.40698264e+00 4.53745847e-02
  2.00561182e-02]
 [7.26011397e-01 1.71332697e+00 6.42184394e-01 2.95753979e-02
  3.59831762e-02]
 [2.42992931e+00 1.43507507e+01 3.38970323e+00 5.95226104e-01
  6.88290316e-01]
 [6.89009353e-01 7.95616490e-01 1.02199023e+00 5.45028631e-02
  3.59205036e-02]
 [6.89151823e-02 1.40712294e+00 2.94014393e+00 4.31196855e-01
  1.40606459e-02]]
--------------------------------------------------
After step 5 - Food:
[[ 6.76892098  2.3755782   3.47949095  5.90930298  5.17101142]
 [ 5.75656851  2.49558142  3.09558712  7.9566142   6.35656418]
 [ 0.          3.05014831  0.          6.26711506  9.28107323]
 [ 2.58164342  9.19461292 10.02870837  8.82200462  6.06378928]
 [ 9.9402129   8.72132408  0.         10.57126233 12.13464426]]
After step 5 - Bacteria:
[[ 1.75710543 26.28195893  2.40248415  0.10005524  0.04875549]
 [ 1.11419333  3.08067753  1.00484844  0.06687185  0.07872477]
 [ 2.47003768 20.55597891  2.64276578  0.89335444  1.02176349]
 [ 1.02716291  1.46129487  1.54313695  0.11674784  0.07829665]
 [ 0.15615668  2.40600232  3.13413235  0.65677766  0.03046308]]
--------------------------------------------------

=== TEST: Minimal Food ===
After step 1 - Food:
[[0.58304773 0.40657847 0.27729044 0.35549933 0.        ]
 [0.41044435 0.97035907 0.16091737 0.05161647 0.        ]
 [0.13608021 0.74576989 0.99243244 0.76151766 0.16329315]
```

```
 [0.         0.63004762 0.92474984 0.70347595 0.35426336]
 [0.13323387 1.05612585 0.         0.50016123 0.60871426]]
After step 1 - Bacteria:
[[0.19371849 1.90143336 0.04868652 0.08235449 6.28056418]
 [2.95038547 0.06369569 0.         0.0246129  1.00159592]
 [0.10680823 0.         0.0136666  1.0873079  3.72170354]
 [1.52856422 0.01948547 0.02366766 0.0136666  0.07880812]
 [3.65295407 0.09635484 1.79874241 0.03434438 0.94145315]]
--------------------------------------------------
After step 2 - Food:
[[0.58327986 0.08409369 0.29264538 0.36630141 0.        ]
 [0.         1.02408114 0.19961983 0.06657654 0.        ]
 [0.13895842 0.81629454 1.06905081 0.60402834 0.        ]
 [0.         0.69158535 0.99277608 0.7668408  0.3746011 ]
 [0.         1.09939924 0.         0.53848029 0.46103915]]
After step 2 - Bacteria:
[[3.72013015e-01 2.71708858e+00 1.10127282e-01 1.21367004e-01
  1.75143228e-01]
 [3.22791062e+00 1.68790282e-01 2.82490730e-03 5.78104454e-02
  1.23682735e-01]
 [2.13797255e-01 3.81855003e-03 4.03057037e-02 1.56799588e+00
  1.36289109e+00]
 [1.42558679e-01 3.16325664e-02 3.81572343e-02 4.24273130e-02
  1.49433538e-01]
 [1.23665588e+00 1.92822538e-01 2.73832503e-01 7.19463449e-02
  1.36163258e+00]]
--------------------------------------------------
After step 3 - Food:
[[0.53595785 0.         0.29264305 0.37072276 0.        ]
 [0.         1.05052662 0.24169123 0.07408626 0.        ]
 [0.11364947 0.89179799 1.14391041 0.34340108 0.        ]
 [0.         0.7683815  1.06360842 0.8267932  0.37831148]
 [0.         2.06981897 0.         0.56995548 0.22180696]]
After step 3 - Bacteria:
[[0.55139186 1.0245741  0.17747023 0.17926338 0.17300284]
 [0.18519388 0.25617326 0.01109488 0.11419422 0.01311304]
 [0.30997494 0.01396379 0.08767556 2.23846934 0.14573829]
 [0.12934661 0.05103164 0.06145559 0.09472516 0.2422617 ]
 [0.33659706 0.29750868 0.38297051 0.13606248 1.95022807]]
--------------------------------------------------
After step 4 - Food:
[[0.44952469 0.         0.27865482 0.36429626 0.        ]
 [0.         1.06109205 0.28547761 0.06771462 0.        ]
 [0.06900872 0.97064435 1.21151373 0.         0.        ]
 [0.         0.84746067 1.13573512 0.87676804 0.36115628]
 [0.         2.10994701 0.         0.58758719 0.        ]]
After step 4 - Bacteria:
[[0.79810067 0.41040799 0.26669228 0.26492433 0.14725719]
```

```
 [0.18748434 0.37279036 0.02772607 0.20265707 0.01411937]
 [0.44686634 0.03311444 0.16282541 2.76282272 0.1285305 ]
 [0.13835849 0.08133627 0.09709538 0.17949482 0.37264691]
 [0.31709159 0.43877422 0.40357611 0.22711821 1.75620452]]
--------------------------------------------------
After step 5 - Food:
[[0.3099412  0.         0.24671378 0.34041521 0.        ]
 [0.         1.05047238 0.32867893 0.03915772 0.        ]
 [0.         1.05061373 1.2649315  0.         0.        ]
 [0.         0.9265378  1.20652825 0.90823286 0.31480097]
 [0.         2.12455411 0.         0.58533034 0.        ]]
After step 5 - Bacteria:
[[1.14869014 0.41666159 0.39583648 0.39162461 0.10094064]
 [0.18468711 0.53942117 0.05835275 0.29713739 0.01407103]
 [0.61364558 0.06676632 0.2377641  0.22179335 0.05999208]
 [0.14490783 0.12821857 0.15159778 0.27162285 0.5378101 ]
 [0.27639353 0.64038496 0.42159753 0.33849204 0.10849298]]
--------------------------------------------------

=== TEST: Plentiful Food ===
After step 1 - Food:
[[35.97882084 35.6658994  38.94621696 43.00791853 43.22963389]
 [34.22224329 47.77473044 47.6343885  46.16577124 46.51779449]
 [51.45989919 36.5521179  35.8172818  50.3169353  46.38489982]
 [41.72756628 31.78272031 37.2624486  50.9786238  47.15764742]
 [48.10023258 35.48351003 45.15846402 38.24223576 45.73751453]]
After step 1 - Bacteria:
[[0.09230274 0.08435618 1.24363586 0.10866637 7.01500855]
 [0.03012334 2.36428146 0.04648697 0.         0.09230274]
 [0.07490741 5.81015115 0.13388638 0.         0.        ]
 [0.08706479 6.78867962 4.56946671 0.05897897 0.        ]
 [0.03786921 2.96512448 0.11321181 0.         0.09230274]]
--------------------------------------------------
After step 2 - Food:
[[38.48358243 38.14115031 41.27389311 45.36763068 44.25868291]
 [37.03755877 49.32635594 49.84256801 48.6861455  48.78292478]
 [53.34959228 37.96936014 38.4512622  52.53995185 48.99611233]
 [44.28569864 32.90061493 38.87104579 53.07536947 49.60118973]
 [50.16463704 37.41790422 47.22189259 41.00867951 48.12159264]]
After step 2 - Bacteria:
[[2.65919359e-01 2.45182765e-01 1.77879462e+00 3.09699165e-01
  1.00036167e+01]
 [9.21219076e-02 3.48105953e+00 1.36402755e-01 4.63980169e-03
  2.63627633e-01]
 [2.17880668e-01 8.45499830e+00 3.86277558e-01 3.61622538e-03
  3.13519036e-03]
 [2.53469630e-01 9.92571484e+00 6.64451676e+00 1.69722537e-01
  4.46899700e-03]
```

```
 [1.14653521e-01 4.35700458e+00 3.25918585e-01 6.99674811e-03
  2.63772868e-01]]
--------------------------------------------------
After step 3 - Food:
[[40.99597773 40.62813304 43.53714443 47.70486995 44.77116876]
 [39.88009122 50.69511036 52.04220999 51.2055661  51.02433053]
 [55.2271566  38.91227286 41.06158714 54.76939378 51.59790161]
 [46.81110219 33.4687274  40.12100672 55.15129097 52.05048692]
 [52.23321948 39.12505852 49.2649369  43.80184856 50.4708964 ]]
After step 3 - Bacteria:
[[5.74977117e-01 5.34687529e-01 2.55385489e+00 6.62459708e-01
  1.42758354e+01]
 [2.10557853e-01 5.12792307e+00 3.00325892e-01 1.99869508e-02
  5.65110258e-01]
 [4.75549797e-01 1.23110776e+01 8.36186784e-01 1.57239041e-02
  1.36475248e-02]
 [5.53620186e-01 1.45137059e+01 9.67107951e+00 3.66722091e-01
  1.93077000e-02]
 [2.59759388e-01 6.40769658e+00 7.04196096e-01 3.00162377e-02
  5.65808887e-01]]
--------------------------------------------------
After step 4 - Food:
[[43.47029013 43.08274487 45.6916154  49.97221462 44.52782539]
 [42.71959125 51.78612814 55.15576982 53.72125431 53.20060726]
 [57.05677299 39.13854786 43.59783788 56.9924934  54.17555195]
 [49.25853484 33.19068695 40.8201043  57.17742016 54.49000315]
 [54.27978506 40.46526982 51.2389502  46.60124952 52.74375132]]
After step 4 - Bacteria:
[[ 1.10585814  1.0368882   3.68052452  1.26049934 20.38747215]
 [ 0.4264867   7.55772754  0.58805099  0.05742428  1.07753264]
 [ 0.92307539 17.93616113  1.60965755  0.04559175  0.03961694]
 [ 1.07518982 21.22472111 14.0891785   0.70513137  0.0556347 ]
 [ 0.52207155  9.43119917  1.35340407  0.08588289  1.07974489]]
--------------------------------------------------
After step 5 - Food:
[[45.84085914 45.44149902 47.64546707 52.10235377 43.1870729 ]
 [45.51555933 52.43038609 57.18912137 56.20099343 55.25321356]
 [58.78580194 38.29800186 45.95494115 59.19415364 56.71242224]
 [51.56312811 31.63514246 40.69156023 59.11282045 56.9012018 ]
 [56.26643352 41.2374249  53.07202543 49.3814335  54.88119914]]
After step 5 - Bacteria:
[[ 1.99534008  1.88585273  5.32422574  2.2501735  29.13696598]
 [ 0.80769717 11.14452899  1.07994767  0.13754849  1.92756526]
 [ 1.68058471 26.14618927  2.90611792  0.11019006  0.09586354]
 [ 1.95824867 31.04269749 20.5438215   1.2724926   0.1336485 ]
 [ 0.98192682 13.89202915  2.44022802  0.20486024  1.93334387]]
--------------------------------------------------
```

```
=== TEST: No Bacteria ===
After step 1 - Food:
[[ 8.97413889  7.21126556 10.10411048 10.52677435  7.35230042]
 [ 6.12383531  8.08402056  9.93000173  8.20275588  7.45512675]
 [ 6.1022419   8.63989119 10.01126575  9.88076153  6.73106428]
 [10.08698679  6.28770994 10.31336286  9.30614492  8.30329481]
 [ 9.14442533  8.70785968  9.4600861   6.30179694  6.13569153]]
After step 1 - Bacteria:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
--------------------------------------------------
After step 2 - Food:
[[ 9.70881459  7.97553698 10.96793239 11.33138078  8.08325457]
 [ 6.78176133  8.83362688 10.77951978  9.02307526  8.1258925 ]
 [ 6.78497669 10.32391362 10.90734819 10.69985947  7.42412252]
 [10.85164253  7.0601524  11.15461538 10.11722839  9.05162774]
 [ 9.93897734  9.46579269 10.28384447  7.03073841  6.80060409]]
After step 2 - Bacteria:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
--------------------------------------------------
After step 3 - Food:
[[10.50346021  8.80515232 11.8971254  12.19805047  8.87528424]
 [ 7.49967644  9.64593905 11.69554085  9.90933358  8.85627367]
 [ 7.52899319 11.14538357 11.87005719 11.58335412  8.17859765]
 [11.67613009  7.8997207  12.06122513 10.99226639  9.86100906]
 [10.79440809 10.28746177 11.1723633   7.82520691  7.52794012]]
After step 3 - Bacteria:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
--------------------------------------------------
After step 4 - Food:
[[11.36184572  9.70418806 12.89516012 13.13033373  9.73234087]
 [ 8.28184958 10.52493371 12.68165657 10.86532289  9.65040759]
 [ 8.33852244 12.03249408 12.90277394 12.53487052  8.99866531]
 [12.5640367   8.81061488 13.03676693 11.93495188 10.73527328]
 [11.71427986 11.17682642 12.12933954  8.68952616  8.32208274]]
After step 4 - Bacteria:
[[0. 0. 0. 0. 0.]
```

```
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
--------------------------------------------------
After step 5 - Food:
[[12.28775708 10.67670845 13.96541577 14.13173859 10.65840433]
 [ 9.13263264 11.47459323 13.74136559 11.89477883 10.51249781]
 [ 9.21785841 12.98904089 14.02126229 13.55797198  9.88855246]
 [13.51894273  9.80951723 15.03473421 12.96143187 11.67826653]
 [12.70213614 12.13783475 13.17091294  9.62803676  9.18748057]]
After step 5 - Bacteria:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
--------------------------------------------------
```

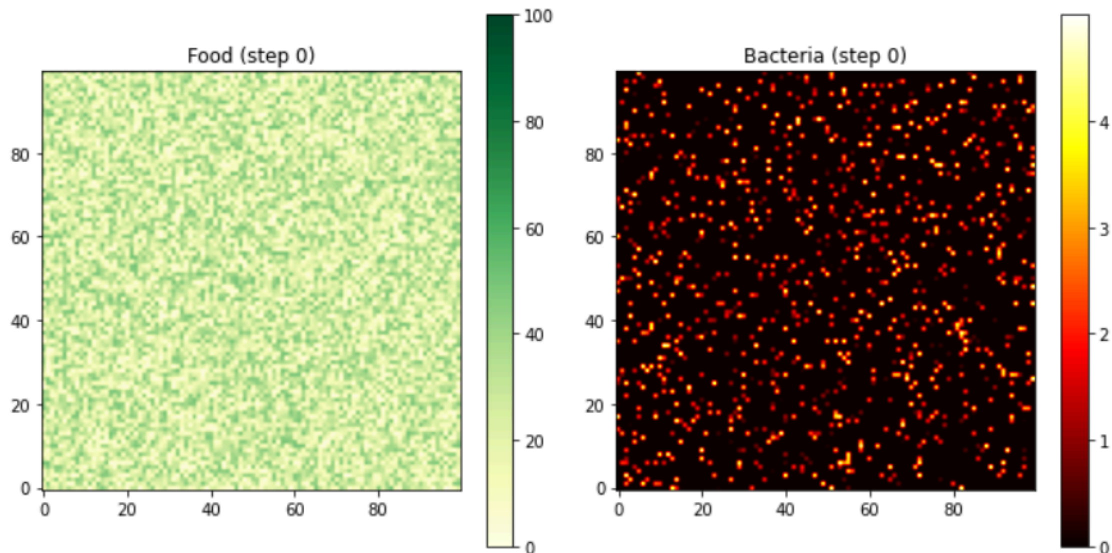
1 3. Simulations

[20]:

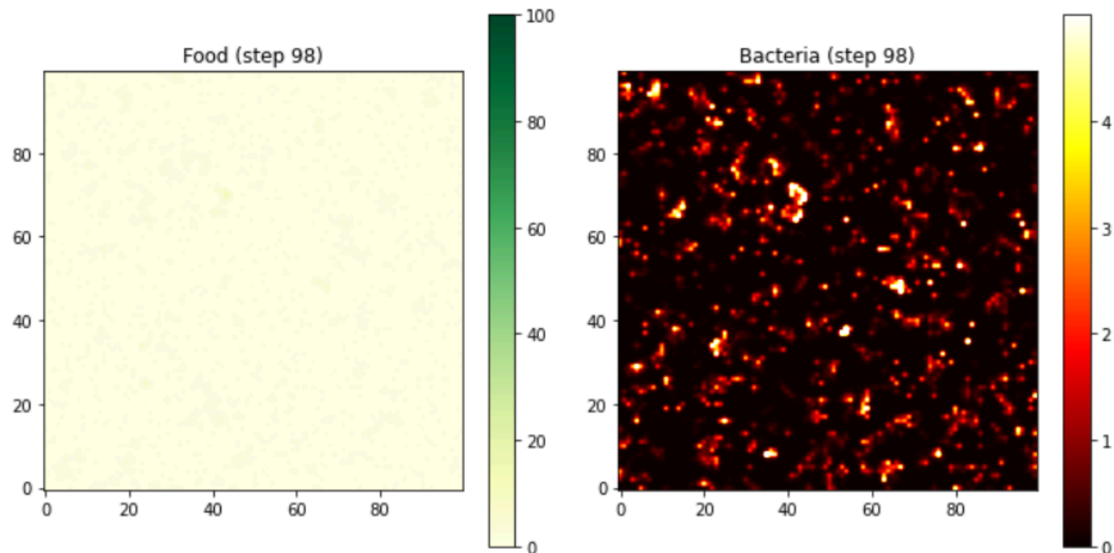
```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

import matplotlib
matplotlib.rcParams['animation.html'] = 'jshtml'

def visualize_grids_static(sim, step_number=0):
    """
    Shows a static, side-by-side view of the current food_grid and␣
    ↪bacteria_grid.
    """
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    # Food
    im1 = axes[0].imshow(
        sim.food_grid,
        cmap='YlGn',
        vmin=0,
        vmax=sim.food_capacity,
        origin='lower')
    axes[0].set_title(f'Food (step {step_number})')
    plt.colorbar(im1, ax=axes[0], orientation='vertical')

    # Bacteria
    im2 = axes[1].imshow(
```

```
        sim.bacteria_grid,
        cmap='hot',
        origin='lower')
    axes[1].set_title(f'Bacteria (step {step_number})')
    plt.colorbar(im2, ax=axes[1], orientation='vertical')

    plt.tight_layout()
    plt.show()

def animate_simulation_jshtml(sim, total_steps=100, steps_per_frame=1,␣
↪interval=200):
    """
    Creates an animation that updates the simulation a few steps at a time,
    then displays the frames inline in a Jupyter notebook using a JS-based
    animation (so we don't need ffmpeg).
    """

    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    ax_food, ax_bacteria = axes

    # Display initial data
    food_img = ax_food.imshow(sim.food_grid, cmap='YlGn',
                              vmin=0, vmax=sim.food_capacity, origin='lower')
    bact_img = ax_bacteria.imshow(sim.bacteria_grid, cmap='hot', origin='lower')
    ax_food.set_title('Food (step 0)')
    ax_bacteria.set_title('Bacteria (step 0)')
    plt.colorbar(food_img, ax=ax_food)
    plt.colorbar(bact_img, ax=ax_bacteria)
    plt.tight_layout()

    def update(frame):
        # Advance the model 'steps_per_frame' times
        for _ in range(steps_per_frame):
            sim.step()

        # Update images
        food_img.set_data(sim.food_grid)
        ax_food.set_title(f'Food (step {frame * steps_per_frame})')
        bact_img.set_data(sim.bacteria_grid)
        ax_bacteria.set_title(f'Bacteria (step {frame * steps_per_frame})')

        return [food_img, bact_img]

    # Create FuncAnimation
    anim = FuncAnimation(fig, update, frames=total_steps, interval=interval,␣
↪blit=False)
    return anim
```

[32]:
```
# 1) Create an instance of simulation
sim = BacteriaGrowthSimulation()

# 2) Initialize it
sim.initialize(
    initial_food_range=(0, 50),
    initial_bacteria_fraction=0.1
)

# 3) Visualize the current state as a single, static image
visualize_grids_static(sim, step_number=0)

# 4) Create and display the animation
anim = animate_simulation_jshtml(sim, total_steps=50, steps_per_frame=2,␣
 ↪interval=200)
HTML(anim.to_jshtml())  # ensures the animation actually renders


# 5) Show the JS-based animation right here in the notebook
from IPython.display import HTML
HTML(anim.to_jshtml())
```



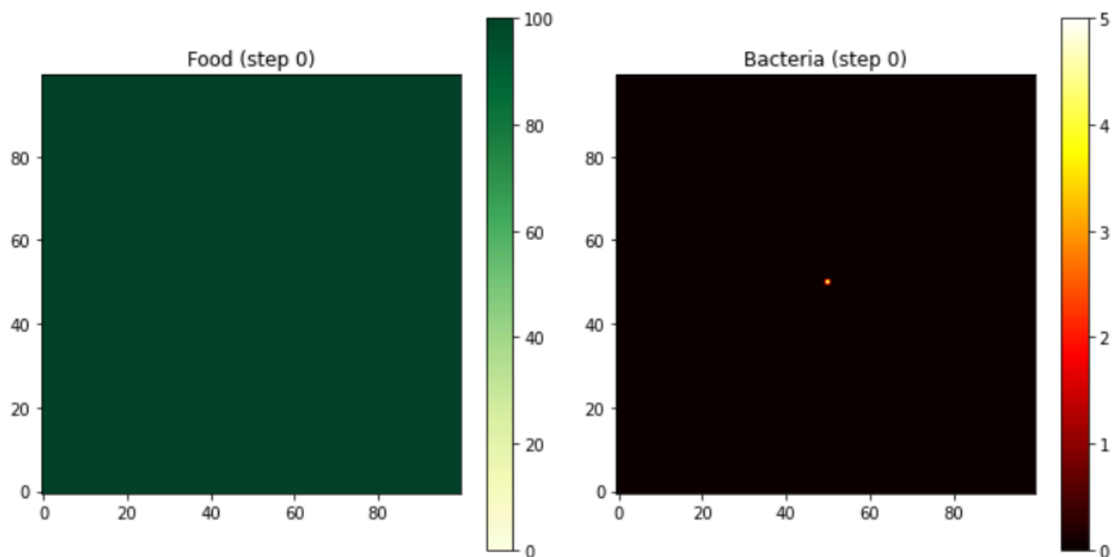
[32]:
```
<IPython.core.display.HTML object>
```



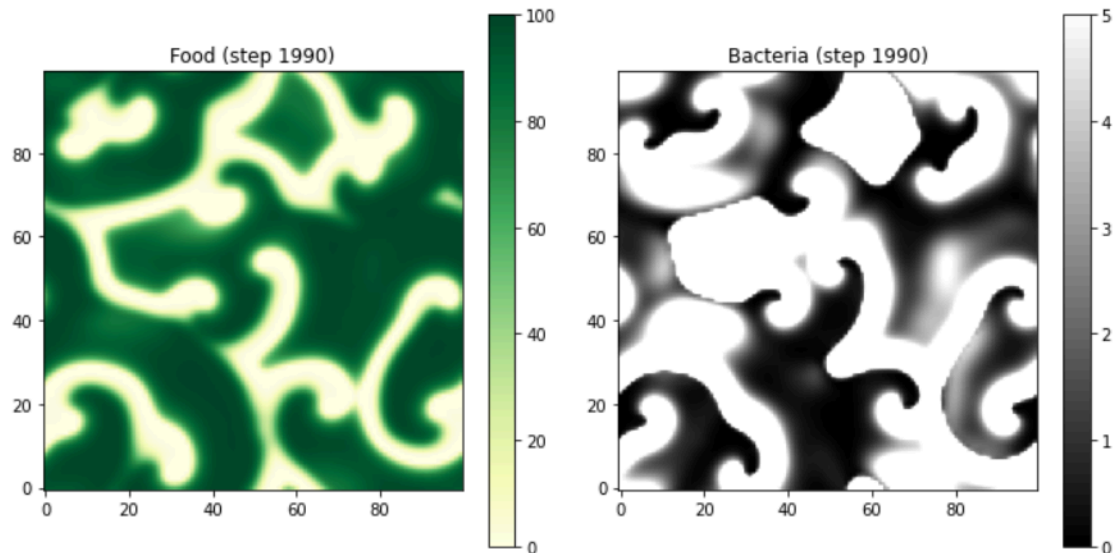
[169]:
```
matplotlib.rcParams["animation.embed_limit"] = 10000  # in MB

def visualize_grids_static(sim, step_number=0):
    """
    Shows a static, side-by-side view of the current food_grid and␣
 ↪bacteria_grid.
    """
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    # Food
    im1 = axes[0].imshow(
        sim.food_grid,
        cmap='YlGn',
        vmin=0,
        vmax=sim.food_capacity,
        origin='lower')
    axes[0].set_title(f'Food (step {step_number})')
    plt.colorbar(im1, ax=axes[0], orientation='vertical')

    # Bacteria
    im2 = axes[1].imshow(
        sim.bacteria_grid,
        cmap='hot',
        origin='lower')
    axes[1].set_title(f'Bacteria (step {step_number})')
    plt.colorbar(im2, ax=axes[1], orientation='vertical')

    plt.tight_layout()
```

```
    plt.show()

def animate_simulation_jshtml(sim, total_steps=100, steps_per_frame=1,␣
 ↪interval=200):
    """
    Creates an animation that updates the simulation a few steps at a time,
    then displays the frames inline in a Jupyter notebook using a JS-based
    animation (so we don't need ffmpeg).
    """

    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    ax_food, ax_bacteria = axes

    # Display initial data
    food_img = ax_food.imshow(sim.food_grid, cmap='YlGn',
                              vmin=0, vmax=sim.food_capacity, origin='lower')
    bact_img = ax_bacteria.imshow(sim.bacteria_grid, cmap='gray',␣
 ↪origin='lower')
    ax_food.set_title('Food (step 0)')
    ax_bacteria.set_title('Bacteria (step 0)')
    plt.colorbar(food_img, ax=ax_food)
    plt.colorbar(bact_img, ax=ax_bacteria)
    plt.tight_layout()

    def update(frame):
        # Advance the model 'steps_per_frame' times
        for _ in range(steps_per_frame):
            sim.step()

        # Update images
        food_img.set_data(sim.food_grid)
        ax_food.set_title(f'Food (step {frame * steps_per_frame})')
        bact_img.set_data(sim.bacteria_grid)
        ax_bacteria.set_title(f'Bacteria (step {frame * steps_per_frame})')

        return [food_img, bact_img]

    # Create FuncAnimation
    anim = FuncAnimation(fig, update, frames=total_steps, interval=interval,␣
 ↪blit=False)
    return anim


# 1) Create an instance of the simulation
sim = BacteriaGrowthSimulation(
    N=100,
    gf=0.5,
```

```
    df=0.5,
    cb=0.5,
    gb=0.1,
    db=0.5
)

# 2) Manually set up the grid:
#    * Uniform food in every cell
#    * One bacterium in the center
sim.food_grid = np.full((sim.N, sim.N), 100.0, dtype=float)
sim.bacteria_grid = np.zeros((sim.N, sim.N), dtype=float)

center = sim.N // 2
sim.bacteria_grid[center, center] = 5.0  # place a single bacterium in the␣
↪middle

# 3) Visualize the current state as a single, static image
visualize_grids_static(sim, step_number=0)

# 4) Create and display the animation for 100 steps
anim = animate_simulation_jshtml(sim, total_steps=200, steps_per_frame=10,␣
↪interval=100)
HTML(anim.to_jshtml())
```



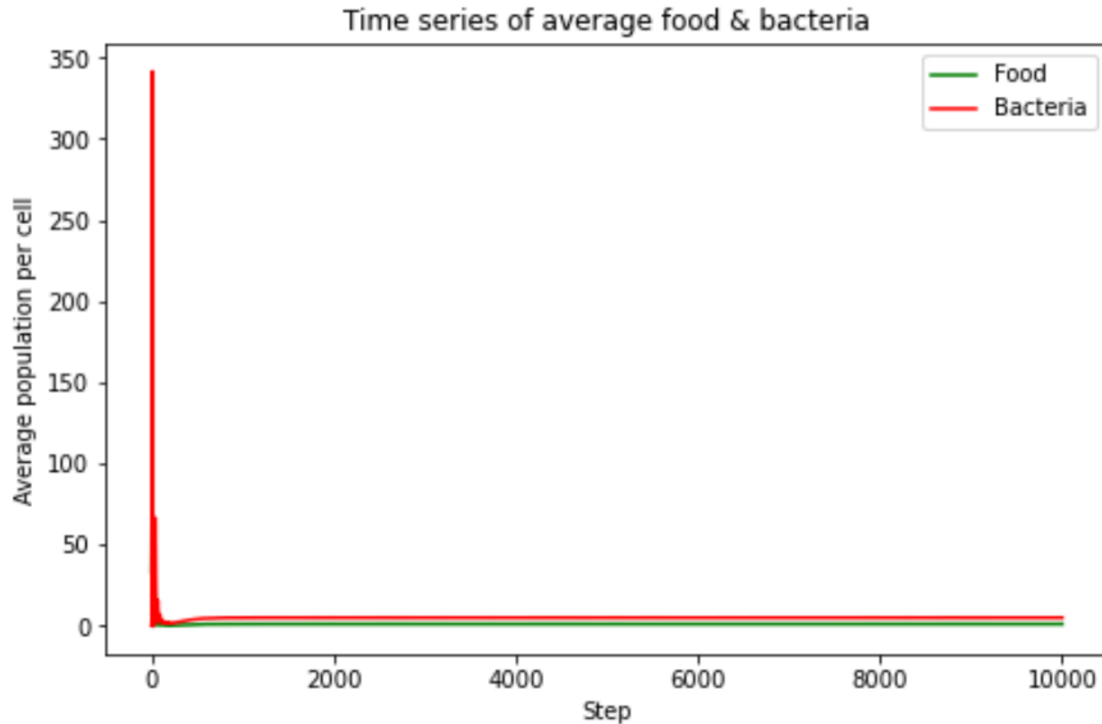

```
[169]: <IPython.core.display.HTML object>
```



1.1 4. Time-Series and Spatial Distribution Analysis

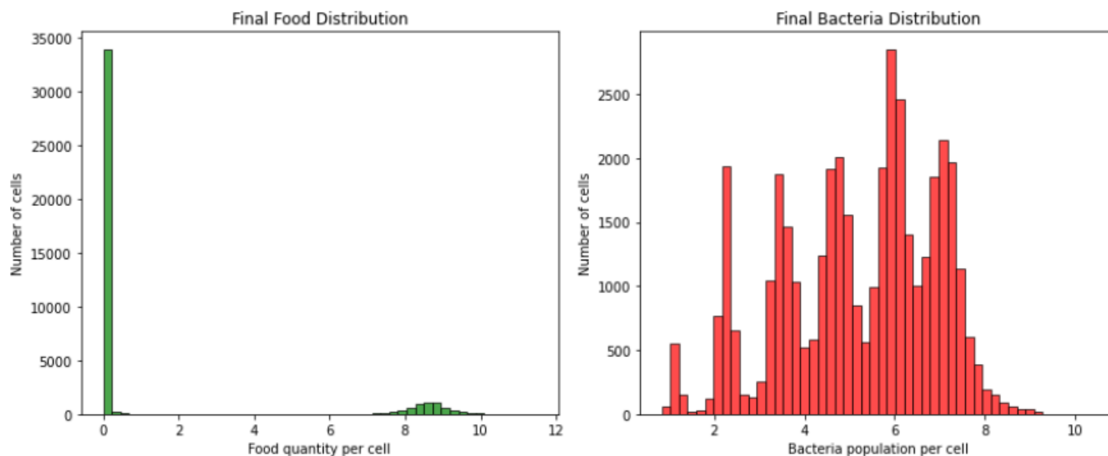
[91]:
```
def run_long_sim_and_record(sim, total_steps=200):
    """
    Runs the simulation for 'total_steps' time steps.
    Records and returns the average food and average bacteria populations per␣
 ↪cell.
    """
    avg_food = []
    avg_bacteria = []

    for step_num in range(total_steps):
        sim.step()
        # Compute mean across the entire grid
        food_mean = np.mean(sim.food_grid)
        bacteria_mean = np.mean(sim.bacteria_grid)
        avg_food.append(food_mean)
        avg_bacteria.append(bacteria_mean)

    return avg_food, avg_bacteria

def plot_time_series(avg_food, avg_bacteria):
    """
    Plots the average food and bacteria populations over time in a single␣
 ↪figure.
    """
    plt.figure(figsize=(8,5))
    plt.plot(avg_food, label='Food', color='green')
```

```
    plt.plot(avg_bacteria, label='Bacteria', color='red')
    plt.xlabel('Step')
    plt.ylabel('Average population per cell')
    plt.title('Time series of average food & bacteria')
    plt.legend()
    plt.show()

# 1) Create a simulation instance (adjusting parameters for interesting␣
↪patterns)
sim = BacteriaGrowthSimulation(
    N=200,
    gf=0.5,
    df=0.2,
    cb=0.2,
    gb=0.8,  # Large bacterial growth
    db=0.8   # Large bacterial diffusion
)



# 2) Initialize with random conditions
sim.initialize(
    initial_food_range=(0, 50),
    initial_bacteria_fraction=0.1
)

# 3) Run for a good number of steps and record data
food_record, bacteria_record = run_long_sim_and_record(sim, total_steps=10000)

# 4) Plot the time series
plot_time_series(food_record, bacteria_record)
```



[73]:
```python
def run_and_histogram_distributions(sim, total_steps=2000):
    """
    Runs the simulation for 'total_steps' steps,
    then plots histograms of the final food and bacteria distributions.
    """
    # 1) Run the model
    for _ in range(total_steps):
        sim.step()

    # 2) Flatten the 2D grids into 1D arrays
    final_food = sim.food_grid.flatten()
    final_bacteria = sim.bacteria_grid.flatten()

    # 3) Create histograms
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Left: Food histogram
    axes[0].hist(final_food, bins=50, color='green', alpha=0.7,␣
↪edgecolor='black')
    axes[0].set_title('Final Food Distribution')
    axes[0].set_xlabel('Food quantity per cell')
    axes[0].set_ylabel('Number of cells')
```

```
    # Right: Bacteria histogram
    axes[1].hist(final_bacteria, bins=50, color='red', alpha=0.7,␣
↪edgecolor='black')
    axes[1].set_title('Final Bacteria Distribution')
    axes[1].set_xlabel('Bacteria population per cell')
    axes[1].set_ylabel('Number of cells')

    plt.tight_layout()
    plt.show()

# Example usage:
# 1) Instantiate simulation with chosen parameters
sim = BacteriaGrowthSimulation(
    N=200,
    gf=0.5,
    df=0.2,
    cb=0.2,
    gb=0.8,   # Large bacterial growth
    db=0.8    # Large bacterial diffusion
)


# 2) Initialize
sim.initialize(
    initial_food_range=(0, 50),
    initial_bacteria_fraction=0.1
)

# 3) Run and create final histograms
run_and_histogram_distributions(sim, total_steps=5000)
```



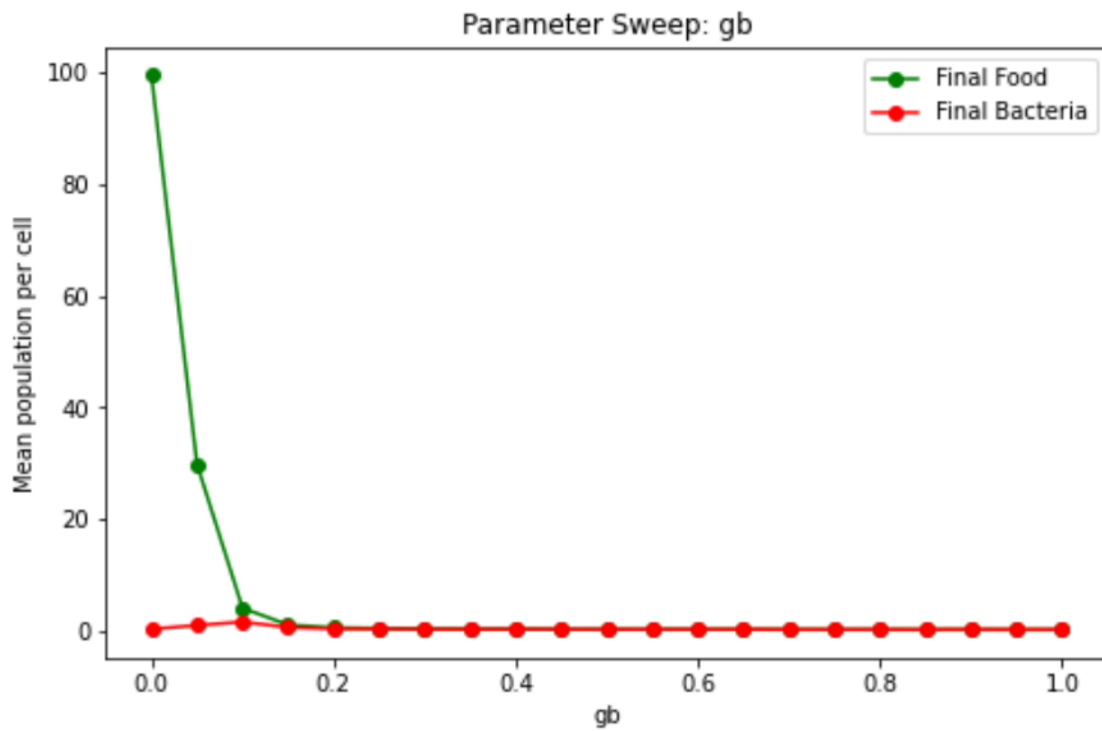
1.2 5. Parameter Sweeps and Final Population Analysis

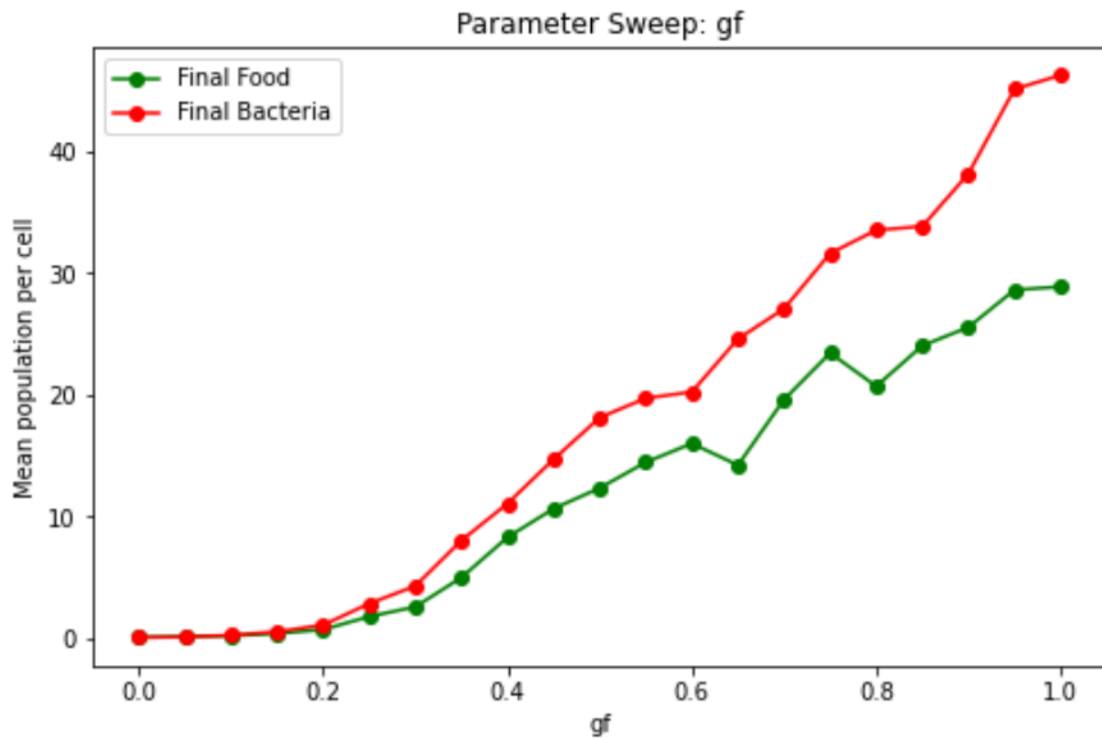
[89]:
```
def param_sweep(
    param_name='gb',
    param_values=np.linspace(0, 1, 11),
    N=100,
    gf=0.1,
    df=0.05,
    cb=0.2,
    gb=0.5,
    db=0.05,
    total_steps=2000
):
    """
    Varies 'param_name' over the given 'param_values' array.
    For each value, runs the simulation for 'total_steps' steps
    and collects the final (mean) food and bacteria populations.

    Returns:
        param_values (array),
        final_food_means (list of floats),
        final_bacteria_means (list of floats).
    """
    final_food_means = []
    final_bacteria_means = []

    for val in param_values:
        # Copy the original parameters, but replace the one we're sweeping
        current_gf = gf
        current_df = df
        current_cb = cb
        current_gb = gb
        current_db = db

        if param_name == 'gf':
            current_gf = val
        elif param_name == 'df':
            current_df = val
        elif param_name == 'cb':
            current_cb = val
        elif param_name == 'gb':
            current_gb = val
        elif param_name == 'db':
            current_db = val

        # Create and initialize the simulation
        sim = BacteriaGrowthSimulation(
```

```
            N=N,
            gf=current_gf,
            df=current_df,
            cb=current_cb,
            gb=current_gb,
            db=current_db
        )
        sim.initialize(initial_food_range=(0, 50), initial_bacteria_fraction=0.
↪1)

        # Run the simulation
        for _ in range(total_steps):
            sim.step()

        # Record final means
        final_food_means.append(np.mean(sim.food_grid))
        final_bacteria_means.append(np.mean(sim.bacteria_grid))

    return param_values, final_food_means, final_bacteria_means


def plot_param_sweep(param_values, food_means, bacteria_means, param_name='gb'):
    """
    Plots the final (mean) food and bacteria populations vs. the parameter␣
↪value.
    """
    plt.figure(figsize=(8,5))
    plt.plot(param_values, food_means, 'g-o', label='Final Food')
    plt.plot(param_values, bacteria_means, 'r-o', label='Final Bacteria')
    plt.xlabel(param_name)
    plt.ylabel('Mean population per cell')
    plt.title(f'Parameter Sweep: {param_name}')
    plt.legend()
    plt.show()
```

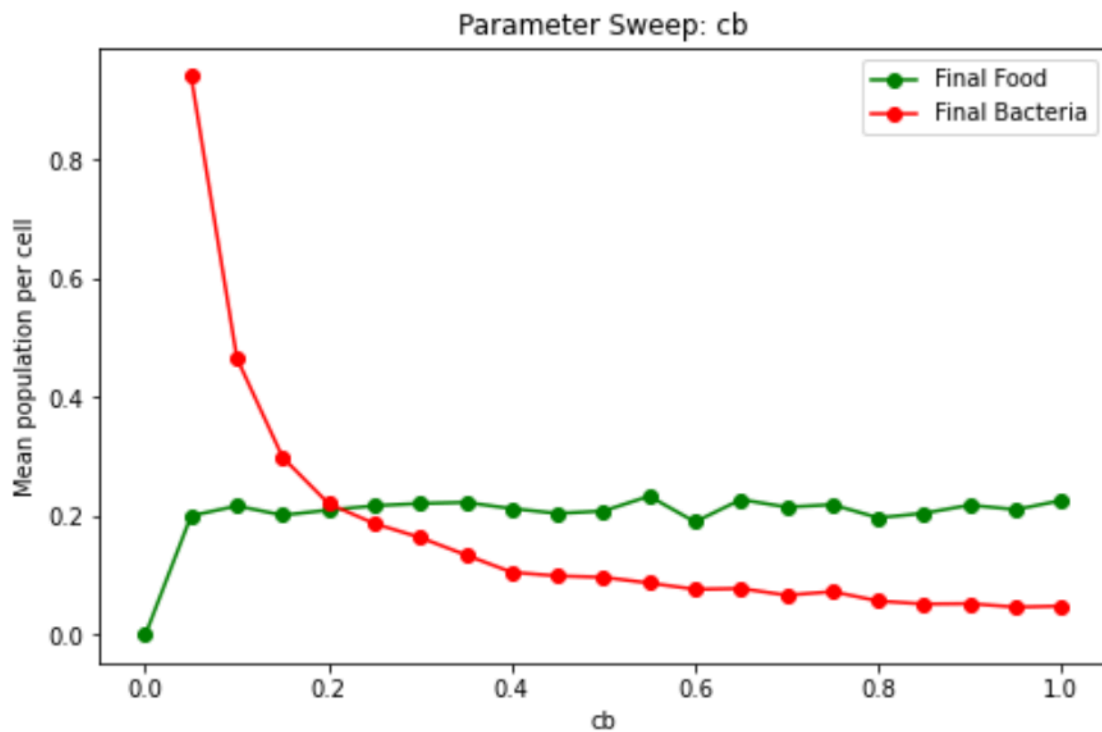
[88]:
```
# Bacteria growth sweep
vals = np.linspace(0, 1, 21)
x, f_means, b_means = param_sweep('gb', vals, total_steps=3000)
plot_param_sweep(x, f_means, b_means, 'gb')

# Food growth sweep
vals = np.linspace(0, 1, 21)
x, f_means, b_means = param_sweep('gf', vals, total_steps=3000)
plot_param_sweep(x, f_means, b_means, 'gf')

# Consumption sweep
vals = np.linspace(0, 1, 21)
```

```
x, f_means, b_means = param_sweep('cb', vals, total_steps=3000)
plot_param_sweep(x, f_means, b_means, 'cb')
```



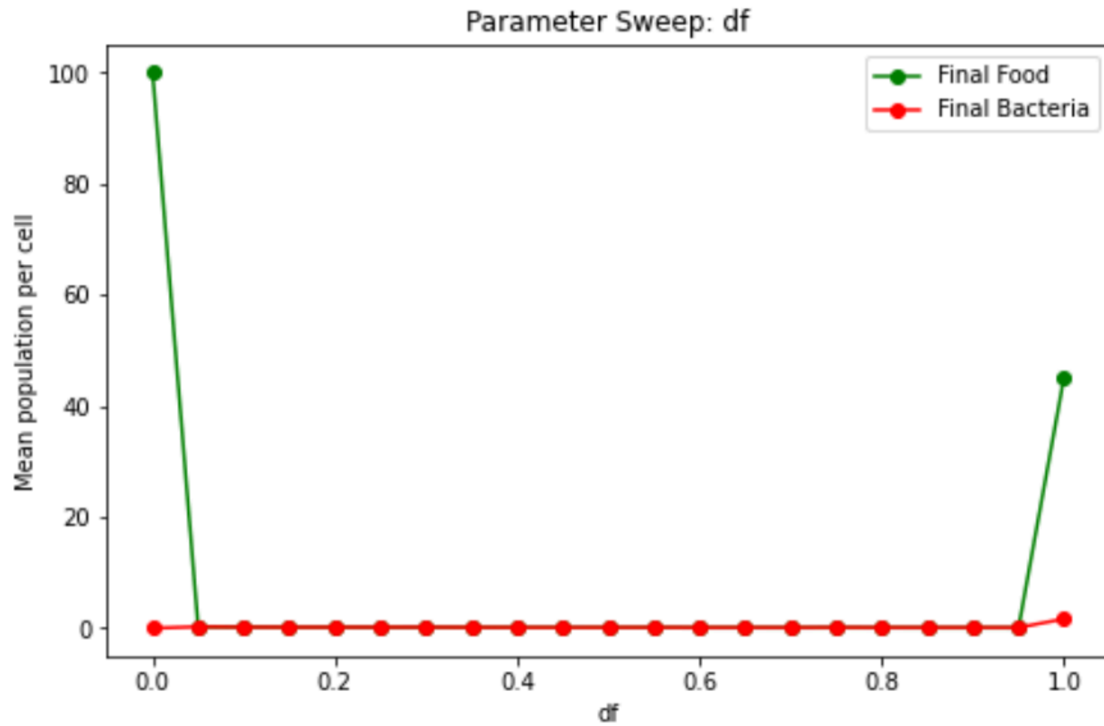


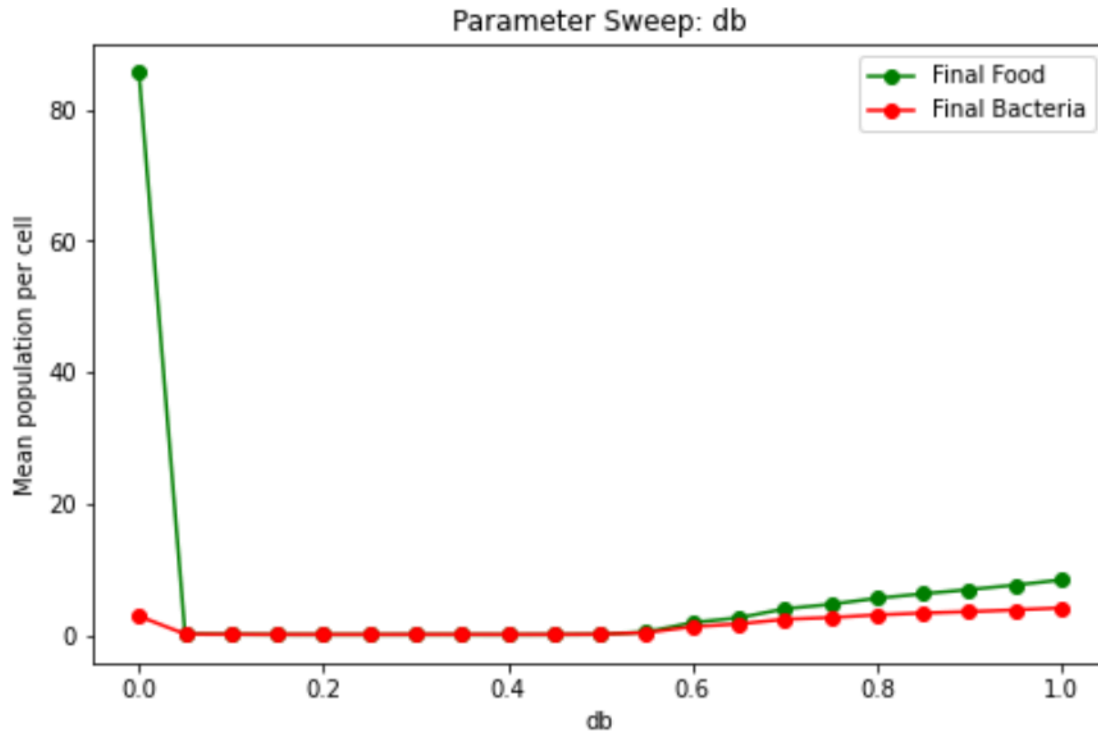
```
/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_14833/1513876596.py:1
00: RuntimeWarning: overflow encountered in multiply
  self.bacteria_grid *= (1.0 + self.gb)
/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_14833/1513876596.py:1
04: RuntimeWarning: invalid value encountered in subtract
  self.bacteria_grid -= bacteria_out
/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_14833/1513876596.py:8
6: RuntimeWarning: invalid value encountered in multiply
  food_needed = self.cb * self.bacteria_grid
/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_14833/1513876596.py:9
5: RuntimeWarning: divide by zero encountered in true_divide
  self.food_grid[not_enough_food_mask] / self.cb
/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_14833/1513876596.py:9
5: RuntimeWarning: invalid value encountered in true_divide
  self.food_grid[not_enough_food_mask] / self.cb
```



[85]:
```
# Food diffusion sweep
vals = np.linspace(0, 1, 21)
x, f_means, b_means = param_sweep('df', vals, total_steps=3000)
plot_param_sweep(x, f_means, b_means, 'df')
```

```
# Bacteria diffusion sweep
vals = np.linspace(0, 1, 21)
x, f_means, b_means = param_sweep('db', vals, total_steps=3000)
plot_param_sweep(x, f_means, b_means, 'db')
```





1.3 6. Theoretical Analysis

[168]:
```
##############################
# 1) Mean-Field Single-Step
##############################
def mean_field_step(F, B, gf, cb, gb, capacity=100):
    """
    Performs one mean-field update step for the bacteria-food model,
    using a partial-starvation approach.

    Parameters:
      F (float): average food
      B (float): average bacteria
      gf (float): food growth rate
      cb (float): bacteria consumption rate
      gb (float): bacteria growth rate
      capacity (float): logistic capacity for food (default=100)
    Returns:
      (F_new, B_new)
    """
    # 1) Food logistic growth
    F_new = F + gf * F * (1 - F/capacity)
```

```
    # 2) Consumption
    needed = cb * B
    if needed <= F_new:
        # enough food
        F_new -= needed
        # 3) reproduction
        B_new = B + gb * B
    else:
        # partial starvation
        fraction_eaten = F_new / needed if needed > 0 else 0
        B_surv = fraction_eaten * B
        B_new = B_surv + gb * B_surv
        F_new = 0

    # clamp negatives
    if F_new < 0:
        F_new = 0
    if B_new < 0:
        B_new = 0
    return F_new, B_new


##############################
# 2) Mean-Field Equilibrium
##############################
def mean_field_equilibrium(gf, cb, gb,
                           capacity=100,
                           max_iter=30,
                           tol=1e-4,
                           F_init=50.0,
                           B_init=1.0,
                           debug=False):
    """
    Iterates mean_field_step() until changes are below 'tol'
    or we hit 'max_iter'.

    Returns: (F_star, B_star, F_series, B_series)
    """
    F = F_init
    B = B_init

    F_series = [F]
    B_series = [B]

    for i in range(1, max_iter+1):
        F_new, B_new = mean_field_step(F, B, gf, cb, gb, capacity)
        F_series.append(F_new)
```

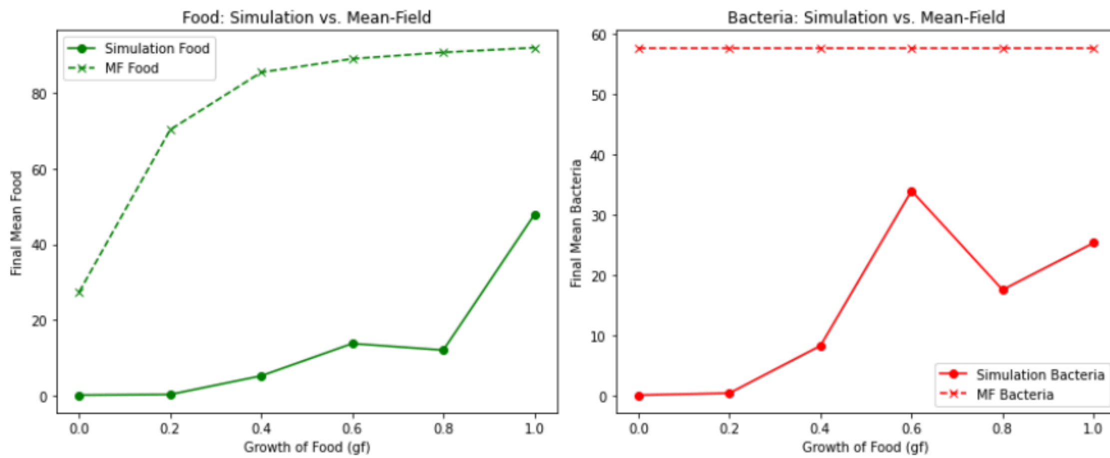
```
        B_series.append(B_new)

        if debug:
            print(f"[Iteration {i}] F={F_new:.4f}, B={B_new:.4f}")

        # check for convergence
        if abs(F_new - F) < tol and abs(B_new - B) < tol:
            return F_new, B_new, F_series, B_series

        F, B = F_new, B_new

    # not converged, return final anyway
    return F, B, F_series, B_series


##############################
# 3) Compare Mean-Field vs Simulation
##############################
def mean_field_vs_simulation(param_name='gf',
                             param_values=np.linspace(0,1,6),
                             total_steps=200,
                             # default parameters for both simulation + MF
                             N=200, gf=0.9, df=0.5,
                             cb=0.2, gb=0.1, db=0.05):
    """
    Sweeps 'param_name' over 'param_values', comparing the final average
    Food + Bacteria from the simulation vs. the mean-field equilibrium.

    param_name: which parameter to vary (gf, df, cb, gb, db)
    param_values: array of values to try
    total_steps: how many steps to run the simulation
    N, gf, df, cb, gb, db: default param settings
    """
    # --- 3A) Simulation (was called "CA" in older code)
    sim_food_means = []
    sim_bact_means = []

    for val in param_values:
        # Collect default parameters in a dict for clarity
        params = dict(N=N, gf=gf, df=df, cb=cb, gb=gb, db=db)
        # Override the one we are sweeping
        params[param_name] = val

        # Create + run the simulation
        sim = BacteriaGrowthSimulation(
            N=params["N"],
            gf=params["gf"],
```

```
            df=params["df"],
            cb=params["cb"],
            gb=params["gb"],
            db=params["db"]
        )
        sim.initialize(initial_food_range=(0, 50), initial_bacteria_fraction=0.
↪1)

        for _ in range(total_steps):
            sim.step()

        sim_food_means.append(np.mean(sim.food_grid))
        sim_bact_means.append(np.mean(sim.bacteria_grid))

    print("Simulation final FOOD:", sim_food_means)
    print("Simulation final BACTERIA:", sim_bact_means)

    # --- 3B) Mean-Field results
    mf_food_means = []
    mf_bact_means = []

    for val in param_values:
        # again use a dict approach
        params = dict(N=N, gf=gf, df=df, cb=cb, gb=gb, db=db)
        params[param_name] = val

        # run mean-field equilibrium
        F_star, B_star, _, _ = mean_field_equilibrium(
            gf=params["gf"],
            cb=params["cb"],
            gb=params["gb"],
            capacity=100,
            max_iter=10,
            tol=1e-4,
            F_init=50.0, # might adjust to avoid immediate crash
            B_init=1.0,  # initial guesses
            debug=False
        )
        mf_food_means.append(F_star)
        mf_bact_means.append(B_star)

    print("Mean-field final FOOD:", mf_food_means)
    print("Mean-field final BACTERIA:", mf_bact_means)

    # --- 3C) Plot
    fig, axes = plt.subplots(1, 2, figsize=(12,5))
```

```
    # X-axis label logic
    if param_name == 'gf':
        x_label = "Growth of Food (gf)"
    elif param_name == 'cb':
        x_label = "Consumption rate (cb)"
    elif param_name == 'gb':
        x_label = "Bacteria growth (gb)"
    elif param_name == 'df':
        x_label = "Food diffusion (df)"
    elif param_name == 'db':
        x_label = "Bacteria diffusion (db)"
    else:
        x_label = param_name

    # Food subplot
    axes[0].plot(param_values, sim_food_means, 'go-', label='Simulation Food')
    axes[0].plot(param_values, mf_food_means, 'gx--', label='MF Food')
    axes[0].set_xlabel(x_label)
    axes[0].set_ylabel('Final Mean Food')
    axes[0].set_title('Food: Simulation vs. Mean-Field')
    axes[0].legend()

    # Bacteria subplot
    axes[1].plot(param_values, sim_bact_means, 'ro-', label='Simulation␣
↪Bacteria')
    axes[1].plot(param_values, mf_bact_means, 'rx--', label='MF Bacteria')
    axes[1].set_xlabel(x_label)
    axes[1].set_ylabel('Final Mean Bacteria')
    axes[1].set_title('Bacteria: Simulation vs. Mean-Field')
    axes[1].legend()

    plt.tight_layout()
    plt.show()


if __name__ == '__main__':
    param_vals = np.linspace(0, 1, 6)
    mean_field_vs_simulation(
        param_name='gf',
        param_values=param_vals,
        total_steps=200,
        N=200,
        gf=0.9,    # default gf
        df=0.5,    # default df
        cb=0.2,    # default cb
        gb=0.5,    # default gb
        db=0.05    # default db
```

```
    )
```

```
Simulation final FOOD: [0.05186882240454924, 0.21040448099482237,
5.144619393538789, 13.699107260604423, 11.95959922366965, 47.84366770503874]
Simulation final BACTERIA: [0.07304678972182342, 0.3937429188948026,
8.257424802622333, 33.94514954706446, 17.53231067821951, 25.30702673073957]
Mean-field final FOOD: [27.333984374999996, 70.31656067216144,
85.51848527636682, 89.09931140107633, 90.80679740256309, 92.03610186169993]
Mean-field final BACTERIA: [57.6650390625, 57.6650390625, 57.6650390625,
57.6650390625, 57.6650390625, 57.6650390625]
```



[ ]:

```

```