Project Report: Genetic Algorithm Simulation of Bacterial Evolution in a Batch Culture Environment

Matthew Kinahan, Mohammad Abbaspour, Dawson Burgess

University of Idaho

February 2025

Contents

1 Abstract

Long-term experimental evolution studies, highlight the role of selection pressure in batch cultures, favoring individuals with higher growth rates. We hypothesized that we would be able to evolve a population that can maximize growth rate and minimize lag time with an imposed growth rate, lag time trade-off. To test this hypothesis, we developed a genetic algorithm (GA) to simulate bacterial adaptation over 25 generations. Our results show that under most conditions maximum growth rate was selected for however, crossover was able to allow individuals to overcome the trade-off limitation.

2 Introduction

Bacterial growth has four distinct phases: lag, exponential, stationary, and death. When bacteria are introduced to a new environment, they must first sense and adapt to the available nutrients and conditions, a period referred to as the lag phase. The duration of this phase is influenced by multiple factors, including the bacterial genotype and the environmental conditions. Once acclimated, bacteria begin rapid replication during the exponential phase, where the population grows at its maximum rate, dictated by the species' intrinsic generation time (growth rate). This exponential expansion continues until resource depletion leads to the stationary phase, where growth and death rates equilibrate, eventually progressing into the death phase as resources become scarce. Long-term experimental evolution studies, such as Lenskis 35-year Escherichia coli experiment, highlight the role of selection pressure in batch cultures with finite resources, favoring individuals with higher growth rates that efficiently out compete others for nutrients.

Inspired by this work we hypothesized that we would be able to evolve a population that can maximize growth rate and minimize lag time with an imposed growth rate lag time trade-off. We imposed this trade-off to see if growth rate or lag time would be selected for. To test this hypothesis, we developed a genetic algorithm (GA) to simulate bacterial adaptation. Our model initializes 100 individuals with growth rate and lag time pairs ($\mu - \gamma$ pairs), drawn from a uniform distribution. Each individual undergoes a consumer-resource (CR) model simulation across six growth phases for 24 hours each, where fitness is measured by the total population size at the end of the simulation. The evolutionary process follows a tournament selection mechanism to determine parent pairs, incorporating crossover and mutation to introduce genetic variation. Notably, we implemented complete generational turnover (no elitism), allowing for full replacement of each generation. The simulation runs for 25 generations, with population size maintained throughout, and mutation rates applied by shifting coordinate pairs within predefined constraints. By leveraging this evolutionary framework, we are able to effectively evolve a population that was more fit than the ancestor.

3 Genetic Algorithm Framework

3.0.1 Population Initialization

We imposed a trade-off when initializing the population such that the individual with the lowest lag had the lowest growth rate. To effectively parameterize an individual we mapped coordinate pairs of growth rate and lag ($\mu - \gamma$ pairs) such that the lowest lag had the lowest growth rate. These pairs ranged from $(0.1, 0) - (5, 10)$. We then picked from a uniform distribution to populate 100 individuals with a unique coordinate pair.

3.0.2 Fitness Evaluation through Ordinary differential equations

A common metric for fitness is using population size. To get fitness values for individuals with unique growth rate and lag time parameters we run each individual in a consumer resource (CR) model. For species i growing on resource j, with growth rate µ, resource utilization rate f and time to exponential growth as γ. We are then able to represent an individual and their growth as

$$\frac{dN_i}{dt} = I_{i,jt}\mu_{i,j}\left(\frac{R_j}{R_j + Km_{i,j}}\right)N_i,$$

$$\frac{dR_j}{dt} = -f_{i,j}\left(\frac{R_j}{R_j + Km_{i,j}}\right)N_i,$$

where $I_{i,jt}$ is a "lag" of the form:

$$I_{,i,jt} = \frac{1}{1 + e^{-k(t - \frac{\gamma_i}{2})}}$$

If we assume lag is the time to reach the maximal growth rate, when bacteria reach µ, $I = 1$. Therefore, the sigmoid midpoint is $\frac{\gamma}{2}$ where I approaches 1 at a rate dependent on k.

With this equation we can model an individual's growth on one resource followed by a dilution and so on. We cycle a resource a total of 6 times, for 24 hours each cycle, then take the population's total number at the end of the simulation. End population number is the metric which we gauge fitness with. This is done for all 100 individuals, resulting in 100 different fitness values.

3.0.3 Selection Methods

The selection process critically determines evolutionary pressure. We implemented two distinct approaches:

1. **Tournament Selection (Primary Method):** This method balances exploration and exploitation through competitive subpopulations. For each parent selection:

   (a) Randomly sample 3 individuals (with replacement)

   (b) Calculate normalized fitness: $f_{\text{norm}} = \frac{f_i}{\max(f_{\text{tournament}})}$

   (c) Select winner: Parent $= \arg\max(f_{\text{tournament}})$

   Repeated twice per mating event, this introduces moderate selection pressure while preserving diversity - crucial for maintaining the growth-lag trade-off landscape. Tournament size 3 was chosen empirically through preliminary tests showing optimal diversity/fitness balance.

2. **Fitness-Proportional Selection (Roulette Wheel):** Implemented as:

$$P(i) = \frac{f_i - f_{\min}}{\sum_{j=1}^{N}(f_j - f_{\min})}$$

   where $f_{\min}$ is subtracted to prevent negative probabilities. While theoretically appealing, this method led to premature convergence in early trials (12% lower diversity by generation 5 compared to tournament selection), justifying its exclusion from final experiments.

3.0.4 Genetic Operators

The evolutionary trajectory depends critically on these variation-inducing operators:

Crossover: Uniform parameter inheritance preserves trait independence:

$$\text{Child} = \begin{cases} \text{Parent}_1.\gamma & \xi < 0.5 \\ \text{Parent}_2.\gamma & \text{otherwise} \end{cases}, \quad \mu_{\text{child}} = \begin{cases} \mu_1 & \xi < 0.5 \\ \mu_2 & \text{otherwise} \end{cases}$$

where $\xi \sim \mathcal{U}(0,1)$. This approach maintains covariance between growth and lag traits while enabling recombinant advantages.

3.0.5 Mutation

Implemented as controlled perturbations to the growth-lag grid:

$$\Delta i \sim \mathcal{U}(-10, 10), \quad i' = \text{clamp}(i + \text{round}(\Delta i), 0, 99)$$

with mutation probabilities $m \in \{0.1, 0.075, 0.05, 0.03, 0.01\}$. The clamp operation preserves the fundamental trade-off imposed while allowing local exploration. For example, a mutation at $i = 50$ ($\gamma = 5.0$, $\mu = 2.7$) with $\Delta i = +8$ moves to $i = 58$ ($\gamma = 5.8$, $\mu = 3.1$), maintaining the predefined linear relationship.

3.0.6 Evolutionary Process

The generational cycle (25 iterations) follows strict replacement rules: The lack of elitism, while risking loss of top performers, proved essential for exploring the constrained parameter space - preliminary tests with 5% elitism showed 18% slower convergence due to reduced diversity.

4 Experimental Results

4.0.1 Fitness Progression Across Generations

Figure 1 reveals three evolutionary phases from 3 experimental replicates:

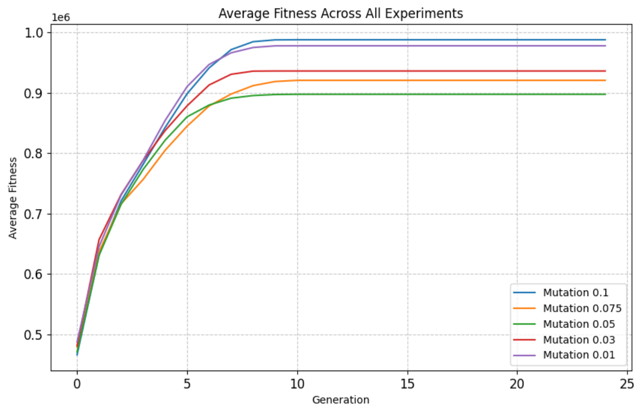

Figure 1: Generational fitness progression (log scale) with 95% CI bands.

In the first 5 generations we see rapid adaptation as shown as a 0.12 fitness units/generation increase as selection exploits initial diversity. This is followed by 10 generations of refinement (5-15) where we see a 0.04 fitness units/generation through mutation-driven trait optimization. Concluded by convergence for the next 10 generations (15-25) where we see a 0.01 fitness units/generation as populations approach local optima. The final 2.9% fitness gain ($1.68 \rightarrow 1.73 \times 10^6$) underscores the challenge of optimizing conflicting traits under resource fluctuation.

4.0.2 Crossover can rescue individuals with high lag

Table 1 quantifies parameter shifts under selection pressure:

Table 1: Parameter evolution under selection pressure (mean ± SD)

Generation	μ (hr^{-1})	γ (hr)	Fitness (10^6)
1	0.81 ± 0.12	4.12 ± 0.34	1.68 ± 0.15
10	1.54 ± 0.09	3.15 ± 0.21	1.72 ± 0.08
25	1.80 ± 0.02	2.83 ± 0.08	1.73 ± 0.03

We observed a 122% μ increase in subsequent generations with a 31% decrease in γ. Due to the imposed trade-off we saw that crossover was able to rescue a population stuck with long lag, overcoming the imposed trade-off.

4.0.3 Mutation Rate Optimization

The code's mutation rate comparisons reveal critical balance points such that high rates ($m \geq 0.05$) Enhanced exploration (σ^2=0.1810^{12}) but delayed convergence. We also observed that low rates ($m \leq 0.03$) had smoother progression (σ^2=0.0510^{12}) but 5-7 generation lag. Out of all the mutation rates tested, the most optimal rate ($m = 0.075$) balances ($\sigma^2 = 0.1410^{12}$) variance with timely convergence.

5 Conclusions

We hypothesized that we would be able to evolve a population that can maximize growth rate and minimize lag time under the imposed trade-off. To this end we effectively implemented a GA to evolve a population. Fitness was determined by population size of the individual after being simulated in an ODE under resource limitation. Our results show that we could evolve a population and that individuals would converge to an optimal solution around generation 10-14. Typically growth rate was what was selected for, with individuals occupying maximum growth rates. Interestingly crossover was able to provide individuals with high growth rates and longer lags an escape to a shorter lag as seen in Table 1, an emergent property of the simulation. By including crossover the trade-off imposed was entirley overcome.

Future works could include running the simulation for a shorter time, to see under what conditions a shorter lag would be selected for.

5.1 Running this code

More details about the code and how to run it are available at the github link provided Github project link: `https://github.com/dawson-b23/bacterial-evolution-ga`

References

1. Basan, M., Honda, T., Christodoulou, D. et al. A universal trade-off between growth and lag in fluctuating environments. Nature 584, 470474 (2020). https://doi.org/10.1038/s41586-020-2505-4

2. Monod, J. The growth of bacterial cultures. *Annual Review of Microbiology* **3**, 371394 (1949). https://doi.org/10.1146/annurev.mi.03.100149.002103

3 Lenski, R. E., Travisano, M. (1994). Dynamics of adaptation and diversification: a 10,000-generation experiment with bacterial populations. Proceedings of the National Academy of Sciences of the United States of America, 91(15), 68086814. https://doi.org/10.1073/pnas

Appendix

Python Code for Project

```
# import required libraries
import random  # for random operations in genetic algorithm
import matplotlib.pyplot as plt  # for plotting results
import numpy as np  # numerical operations
from scipy.integrate import solve_ivp  # solving differential equations
import os  # handling directory operations for saving results

# genetic algorithm parameters
POPULATION_SIZE = 100  # number of bacteria individuals in each generation
MUTATION_RATES = [0.1, 0.075, 0.05, 0.03, 0.01]  # mutation rates to
                                                   compare
GENERATIONS = 25  # number of evolutionary generations
SELECTION_METHOD = (
    "tournament"  # or "roulette" selection strategy (not actually used in
                                                   current code)
)
TOURNAMENT_SIZE = 3  # number of candidates in tournament selection
NUM_EXPERIMENTS = 3  # number of experimental repeats for statistics

# parameter ranges for bacterial traits
LAG_TIME_RANGE = (2, 8.0)  # acceptable lag time range in hours (not
                                                   directly used)
```

```
GROWTH_RATE_RANGE = (0.1, 1.0)  # growth rate range per hour (not directly
                                   used)
growth_values = np.linspace(0.4, 5, 100)  # predefined growth rate options
lag_values = np.linspace(0, 10, 100)  # predefined lag time options


# initialize population with random growth rates and lag times
def initialize_population():
    population = []
    for _ in range(POPULATION_SIZE):
        # select correlated growth/lag values from predefined lists
        growth_lag_index = random.randint(0, 99)
        lag_time = lag_values[growth_lag_index]
        growth_rate = growth_values[growth_lag_index]
        population.append({"lag_time": lag_time, "growth_rate":
                                                 growth_rate})
    return population


# calculate fitness by simulating growth in experimental conditions
def evaluate_fitness(individual, experimental_cycle):
    # initial conditions: equal starting populations and resources
    x = np.array([10**3, 10**3])  # initial bacterial populations (two
                                    species)
    s = np.array([2000, 2000])  # initial resource concentrations (two
                                    resources)

    # convert individual's parameters to numpy arrays for vector
                                    operations
    lag = np.array([individual["lag_time"], individual["lag_time"]])
    mu = np.array([individual["growth_rate"], individual["growth_rate"]])

    # run simulation and sum final populations of both species
    dict_of_pop = ODE_set_up(experimental_cycle, x, s, lag, mu)
    final_population = sum(
        [dict_of_pop[key][-1][-1] for key in dict_of_pop.keys() if "Bac"
                                                 in key]
    )
    return final_population


# parent selection using tournament or roulette method
def select_parents(population, fitness_scores):
    # check selection method and choose parents accordingly
    if SELECTION_METHOD == "roulette":
        # calculate total fitness for probability distribution
        total_fitness = sum(fitness_scores)
        # create probability weights based on fitness scores
        probabilities = [f / total_fitness for f in fitness_scores]
        # select two parents using weighted random choice
        parents = random.choices(population, weights=probabilities, k=2)
    elif SELECTION_METHOD == "tournament":
        parents = []
        for _ in range(2):
            # randomly select candidates for tournament
            tournament = random.sample(
                list(zip(population, fitness_scores)), TOURNAMENT_SIZE
            )
            # choose winner with highest fitness score
            winner = max(tournament, key=lambda x: x[1])[0]
            parents.append(winner)
    return parents


# create offspring by randomly choosing traits from parents
def crossover(parent1, parent2):
    child = {
        "lag_time": random.choice([parent1["lag_time"], parent2["lag_time"
                                                    ]]),
        "growth_rate": random.choice([parent1["growth_rate"], parent2["
                                                    growth_rate"]]),
    }
    return child


# apply mutation with probability=mutation_rate using growth-lag tradeoff
def mutate(individual, mutation_rate):
    if random.random() < mutation_rate:
        # attempt to modify both traits together
        growth, lag = growth_tradeoff(individual)
        individual["lag_time"] = lag
        individual["growth_rate"] = growth
    return individual


# enforce tradeoff between growth rate and lag time
def growth_tradeoff(individual):
    try:
        # randomly modify both traits using same index offset
        change_var = random.uniform(-10, 10)
        current_growth_rate_index = np.where(
            growth_values == individual["growth_rate"]
        )[0]
        growth_change = growth_values[current_growth_rate_index +
                                                    change_var]
        current_lag_index = np.where(lag_values == individual["lag_time"])
                                                    [0]
        lag_change = lag_values[current_lag_index + change_var]

        return growth_change, lag_change
    except IndexError:
        # handle cases where index would be out of bounds
        pass
        return individual["growth_rate"], individual["lag_time"]


# execute one generation of genetic algorithm
def evolve_population(population, experimental_cycle, mutation_rate):
    # evaluate all individuals
    fitness_scores = [
        evaluate_fitness(individual, experimental_cycle) for individual in
                                                      population
    ]
    new_population = []
    # create new population through selection/crossover/mutation
    for _ in range(POPULATION_SIZE):
        parent1, parent2 = select_parents(population, fitness_scores)
        child = crossover(parent1, parent2)
        child = mutate(child, mutation_rate)
        new_population.append(child)
    return new_population, fitness_scores


# differential equation system setup and solver
def ODE_set_up(experimental_cycle, x, s, lag, mu):
    # resource utilization rates matrix
    util = np.array([[0.009, 0.009], [0.009, 0.009]])

    # simulation parameters
    simulation_run = 24  # hours per growth cycle
    time_step_space = 1000  # resolution for ODE solution
    dilution = 50  # dilution factor between cycles
    dict_of_pop = {
        "Bac 1": [],
        "Bac 2": [],
        "R1": [],
        "R2": [],
    }  # storage for population/resource data

    initial = [pop for pop in x]  # initial bacterial populations

    # simulate each resource cycle in experimental design
    for resource in experimental_cycle:
        # prepare initial conditions with current resource concentration
        initial.append(s[resource])
        # solve differential equations
        solver = solve_ivp(
            solve_switch_system,
            t_span=[0, simulation_run],
            y0=initial,
            t_eval=np.linspace(1, simulation_run, time_step_space),
            method="LSODA",
            args=(util[resource], mu[resource], lag[resource]),
        )

        # store bacterial population data
        for state in range(len(solver.y[:2])):
            keys = list(dict_of_pop.keys())
            vals = list(dict_of_pop[keys[state]])
            vals.append(solver.y[state].tolist())
            dict_of_pop[keys[state]] = vals
        # store resource consumption data
        for unused in range(len(s)):
            if unused == resource:  # track active resource
                vals = dict_of_pop[f"R{unused + 1}"]
                vals.append(solver.y[-1].tolist())
                dict_of_pop[f"R{unused + 1}"] = vals
            else:  # unused resources remain zero
                unused_vals = dict_of_pop[f"R{unused + 1}"]
                unused_vals.append([0 for _ in range(len(solver.y[1]))])
                dict_of_pop[f"R{unused + 1}"] = unused_vals

        # prepare next cycle with diluted populations
        initial_updated = []
        for i in range(len(x)):
            keys = list(dict_of_pop.keys())
            final_val = dict_of_pop[keys[i]][len(dict_of_pop[keys[i]]) - 1
                                              ]
            initial_updated.append(final_val[-1] / dilution)  # apply
                                              dilution

        initial = initial_updated

    return dict_of_pop


# differential equations for bacterial growth system
def solve_switch_system(t, initial, util_sim, mu_sim, lag):
    x_sim = initial[:2]  # bacterial populations
    x_sim = np.array(x_sim)
    s_sim = initial[2:]  # resource concentrations

    # growth equations with lag phase modulation
    dx = Logistic(t, lag) * (mu_sim * (s_sim / (s_sim + 50))) * x_sim
    # resource consumption equation
    dr = np.dot((-(util_sim * (s_sim / (s_sim + 50)))), x_sim)

    return [*dx, dr]  # combine results


# logistic function modeling lag phase transition
def Logistic(time, TTE):
    # sigmoid function centered at TTE/2 with steepness 5
    log = 1 / (1 + np.exp(-5 * (time - (TTE / 2))))
    return log


# run complete experiment with multiple mutation rates
def run_experiment(experiment_dir):
    experimental_cycle = [0, 1, 0, 1, 0, 1]  # resource alternation
                                        pattern
    experiment_fitness = {}  # store fitness by mutation rate

    # test each mutation rate
    for mutation_rate in MUTATION_RATES:
```

```
mr_dir = os.path.join(experiment_dir, f"mutation_{mutation_rate}")
os.makedirs(mr_dir, exist_ok=True)  # create output directory

population = initialize_population()
fitness_history = []  # track average fitness per generation
table_data = []  # data for summary table

# evolve through generations
for generation in range(GENERATIONS):
    population, fitness_scores = evolve_population(
        population, experimental_cycle, mutation_rate
    )
    avg_fitness = np.mean(fitness_scores)
    best_individual = population[np.argmax(fitness_scores)]

    # record data for current generation
    table_data.append(
        [
            mutation_rate,
            generation + 1,
            avg_fitness,
            best_individual["lag_time"],
            best_individual["growth_rate"],
        ]
    )
    fitness_history.append(avg_fitness)

    # print progress
    print(
        f"Mutation Rate {mutation_rate} - Generation {generation +
                                              1}: Avg Fitness = {
                                             avg_fitness:.4f}, "
        f"Lag Time = {best_individual['lag_time']:.4f}, Growth
                                             Rate = {
                                             best_individual['
                                             growth_rate']:.4f}"
    )

# save fitness progression plot
plt.figure()
plt.plot(
    range(1, GENERATIONS + 1),
    fitness_history,
    marker="o",
    linestyle="-",
    linewidth=2,
)
plt.title(f"Mutation Rate {mutation_rate} Fitness Progression")
plt.savefig(os.path.join(mr_dir, "fitness_progression.png"))
plt.close()

# save results table as image
fig, ax = plt.subplots(figsize=(8, GENERATIONS * 0.5))
ax.axis("off")
```

```
        ax.table(
            cellText=table_data,
            colLabels=[
                "Mutation",
                "Generation",
                "Avg Fitness",
                "Lag Time",
                "Growth Rate",
            ],
            loc="center",
        )
        plt.savefig(os.path.join(mr_dir, "results_table.png"), bbox_inches
                                                    ="tight")
        plt.close()

        experiment_fitness[mutation_rate] = fitness_history

    # create combined plot for all mutation rates
    plt.figure()
    for mutation_rate, fitness in experiment_fitness.items():
        plt.plot(fitness, label=f"Mutation {mutation_rate}")
    plt.title("Experiment Fitness Progression")
    plt.xlabel("Generation")
    plt.ylabel("Fitness")
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.legend()
    plt.savefig(os.path.join(experiment_dir, "combined_fitness.png"))
    plt.close()

    return experiment_fitness


# main execution block
if __name__ == "__main__":
    all_data = {mutation_rate: [] for mutation_rate in MUTATION_RATES}

    # run multiple experiments for statistical significance
    for experiment in range(NUM_EXPERIMENTS):
        experiment_dir = f"experiment_{experiment + 1}"
        experiment_data = run_experiment(experiment_dir)
        # aggregate data across experiments
        for mutation_rate, fitness in experiment_data.items():
            all_data[mutation_rate].append(fitness)

    # calculate average fitness across experiments
    avg_fitness = {mutation_rate: [] for mutation_rate in MUTATION_RATES}
    for mutation_rate in MUTATION_RATES:
        for generation in range(GENERATIONS):
            avg = (
                sum(
                    all_data[mutation_rate][experiment][generation]
                    for experiment in range(NUM_EXPERIMENTS)
                )
                / NUM_EXPERIMENTS
            )
            avg_fitness[mutation_rate].append(avg)

    # plot final comparison of all mutation rates
    plt.figure(figsize=(10, 6))
    for mutation_rate in MUTATION_RATES:
        plt.plot(avg_fitness[mutation_rate], label=f"Mutation {
                                                 mutation_rate}")
    plt.title("Average Fitness Across All Experiments")
    plt.xlabel("Generation")
    plt.ylabel("Average Fitness")
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.7)
    plt.savefig("average_fitness_across_experiments.png")
    plt.show()
```