Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

CT421

2025–03–17

Project 2: Evolutionary Game Theory

1 Part 1: Evolution Against Fixed Strategies

1.1 Implementation

To implement the genetic algorithm for this assignment, I largely re-used the general framework I developed in the previous assignment, making appropriate changes and removing unnecessary features. The genetic algorithm can be tuned by providing command-line flags & arguments, the possible options for which can be displayed by running the program with the `-h` flag, i.e., `python3 ipd.py -h`, which gives the following output:

```
usage: ipd.py [-h] [-s SIZE] [-g NUM_GENERATIONS] [-a GIVE_UP_AFTER]
              [-i NUM_ITERATIONS] [-p SELECTION_PROPORTION]
              [-c CROSSOVER_RATE] [-m MUTATION_RATE] [-o OUTPUT_FILE]

Program to evolve strategies for the Iterated Prisoner's Dilemma

options:
  -h, --help            show this help message and exit
  -s, --size SIZE       Initial population size
  -g, --num-generations NUM_GENERATIONS
                        Number of generations
  -a, --give-up-after GIVE_UP_AFTER
                        Number of generations to give up after if best
                        solution has remained unchanged
  -i, --num-iterations NUM_ITERATIONS
                        Number of iterations of the dilemma between two agents
  -p, --selection-proportion SELECTION_PROPORTION
                        The proportion of the population to be selected
                        (survive) on each generation
  -c, --crossover-rate CROSSOVER_RATE
                        Probability of a selected pair of solutions to
                        sexually reproduce
  -m, --mutation-rate MUTATION_RATE
                        Probability of a selected offspring to undergo
                        mutation
  -o, --output-file OUTPUT_FILE
                        File to write TSV results to
```

Listing 1: Output of `python3 ipd.py -h`

I chose to represent each strategy as a 3-bit string, where `0` represents defection and `1` represents co-operation; the first bit of the string represents the strategy's first move, the second bit represents the strategy's reaction to a defection by its opponent, and the third bit represents the strategy's reaction to a co-operation by its opponent. For that reason, there are only eight possible strategies in the search space:

- `[0, 0, 0]`: always defect.
- `[0, 0, 1]`: grim tit-for-tat.
- `[0, 1, 0]`: defect at first, then do opposite of what opponent did last.
- `[0, 1, 1]`: defect at first, then always co-operate.
- `[1, 0, 0]`: feint co-operation, then always defect.
- `[1, 0, 1]`: tit-for-tat.
- `[1, 1, 0]`: co-operate at first, then do opposite of what opponent did last.

- `[1, 1, 1]`: always co-operate.

Because there are only 8 possibilities in the search space, and the recommended population size in the assignment specification was 50 – 100, any random initialisation of the population was almost guaranteed to find not only the optimal solution, but every possible solution in the search space (assuming there is one optimal solution out of the eight, and that we randomly initialise 100 individuals, the chances of *not* finding the optimal solution immediately are $\frac{7}{8}^{100} \approx 0.0000015878$). Therefore, I took the assignment not to focus on finding the optimal solution, but exploring how the population converges on the optimal solution for a given fixed fitness landscape. This small search space also meant that the population converges very quickly, and quickly sheds diversity in the population; in an attempt to to mitigate this, I set the mutation rate to be relatively high (0.1), and allowed the number of generations to be longer than necessary so as to observe the population dynamics over time. My crossover & mutation operators were also relatively simple to reflect this small search space: I implemented one crossover operator, that being single-point crossover, and one mutation operator, that being a simple bit-flip mutation.

1.2 Exploring Convergence with different Fitness Evaluations

1.2.1 Equally-Proportioned Always Co-Operate, Always Defect, & Tit-for-Tat

When the fitness function consisted of an always co-operate strategy, an always defect strategy, & a tit-for-tat strategy, the best-performing evolved strategy was `[0,1,0]`: defect on the first move, then do the opposite of what the opponent did last time. It achieved a fitness of 75, narrowly outperforming its more-polite sibling of `[1,1,0]` (co-operate at first, then do the opposite of what the opponent did) and `[0,0,0]` (always defect), both with fitnesses of 74. This surprised me at first, but it makes sense: the strategy of doing the opposite of what the opponent last did performs poorly against Always Defect, as it will lose each time, but exploits Always Co-Operate efficiently, allowing it to gain a high fitness score. It also does reasonably well against Tit-for-Tat: every second iteration, it will successfully exploit Tit-for-Tat, but lose every other iteration. The strategy of defecting then doing the opposite of the opponent out-performs its sibling co-operate then do the opposite because it doesn't miss the opportunity to exploit Tit-for-Tat & Always Co-Operate on its first move.

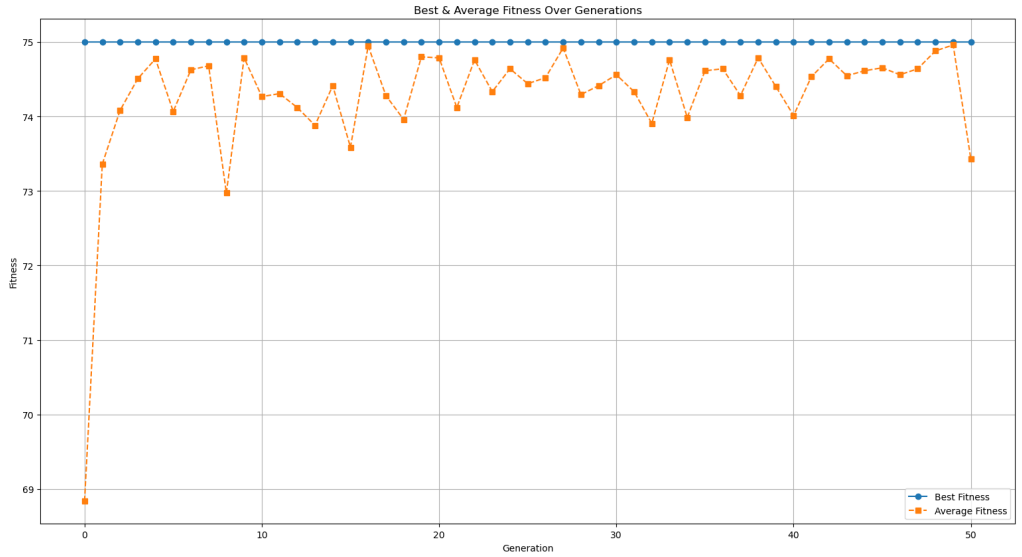

Figure 1: Fitness over generations

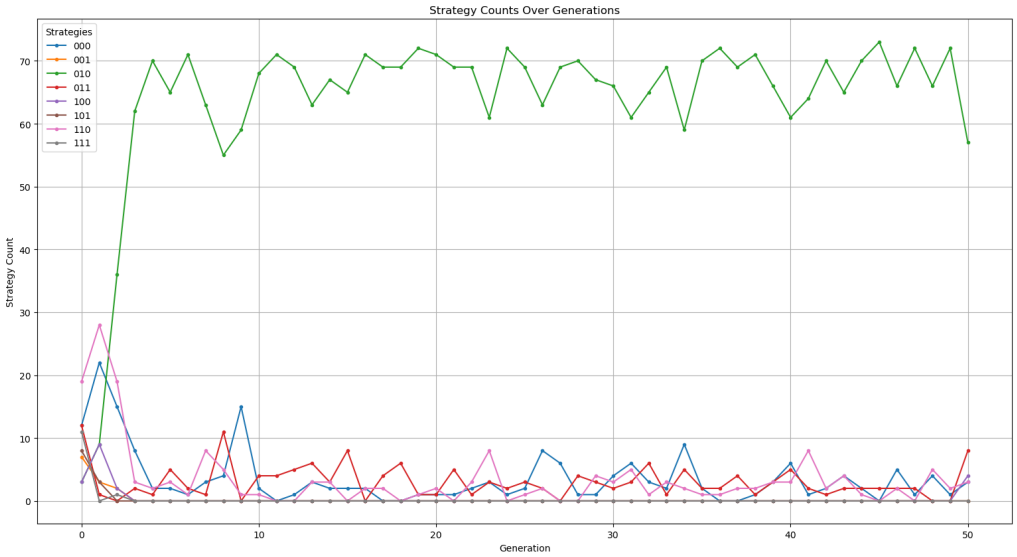

Figure 2: Diversity of the strategy population over generations

1.2.2 2× Always Defect, 1× Always Co-Operate, & 1× Tit-for-Tat

```
Best strategy: [0, 0, 0]
Fitness: 84
Generation: 0
[0, 0, 0]: 84
[0, 0, 1]: 77
[0, 1, 0]: 76
[0, 1, 1]: 63
[1, 0, 0]: 82
[1, 0, 1]: 78
[1, 1, 0]: 74
[1, 1, 1]: 60
```

Listing 2: Output of 2× always defect, 1× always co-operate, & 1× tit-for-tat

When a second Always Defect is added to the mix, it becomes a better strategy to also always defect: while this means it will do poorly against Always Defect and Tit-for-Tat, it can efficiently & ruthlessly exploit the Always Co-Operate to gain a high fitness regardless.

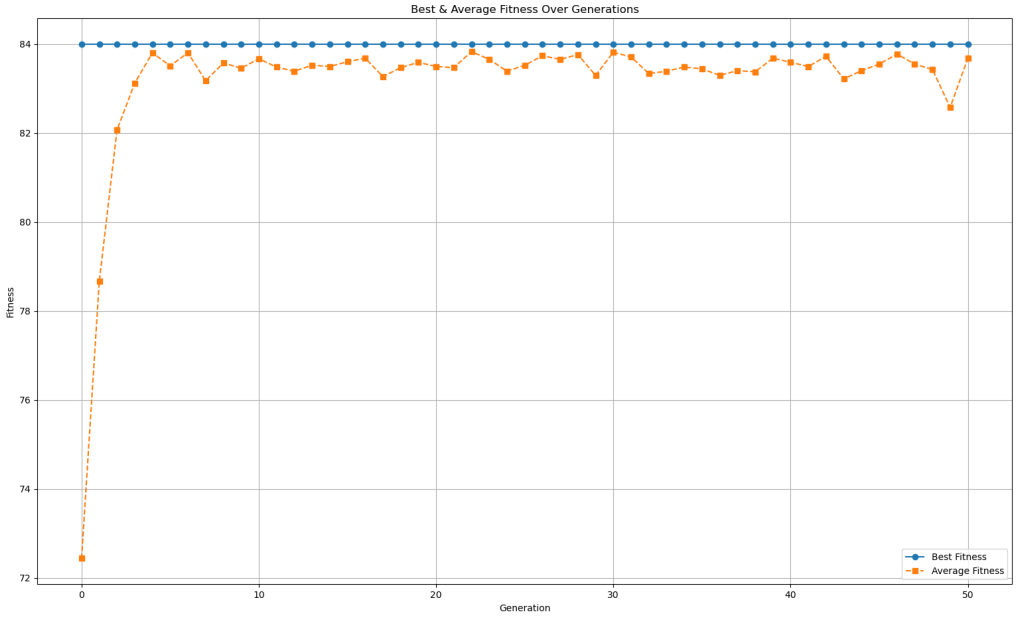

Figure 3: Fitness over generations

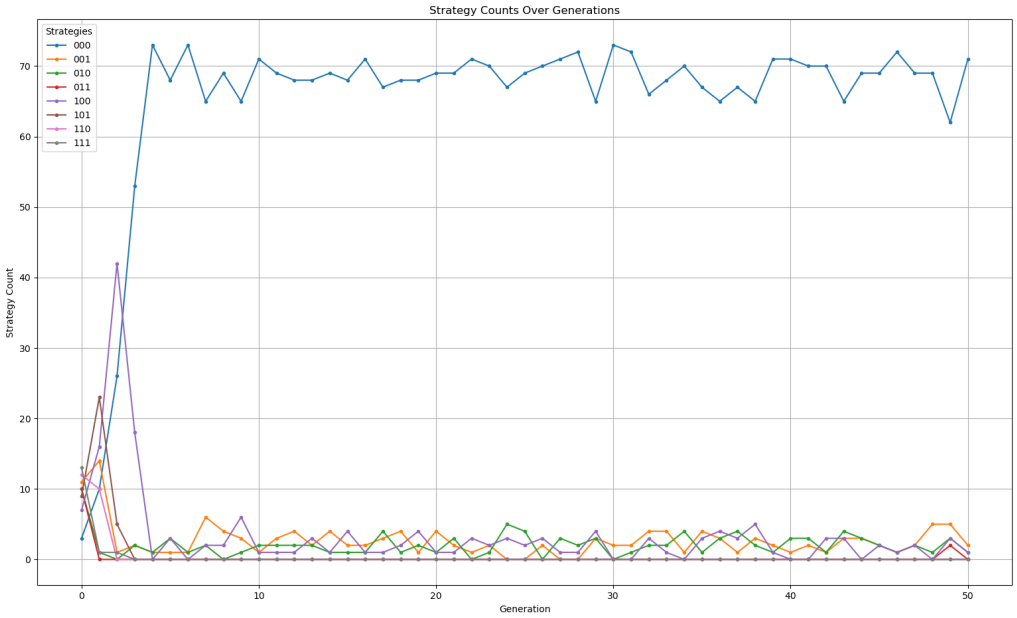

Figure 4: Diversity of the strategy population over generations

1.2.3 1× Always Defect, 2× Always Co-Operate, & 1× Tit-for-Tat

```
Best strategy: [0, 1, 0]
Fitness: 125
Generation: 0
[0, 0, 0]: 124
[0, 0, 1]: 99
[0, 1, 0]: 125
[0, 1, 1]: 94
```

```
[1, 0, 0]: 121
[1, 0, 1]: 99
[1, 1, 0]: 122
[1, 1, 1]: 90
```

Listing 3: 1× always defect, 2× always co-operate, & 1× tit-for-tat

When a second Always Co-Operate is added to the mix, the best strategy reverts to being defect on the first move, then do the opposite of what the opponent did, for the same reasons as previously explored.

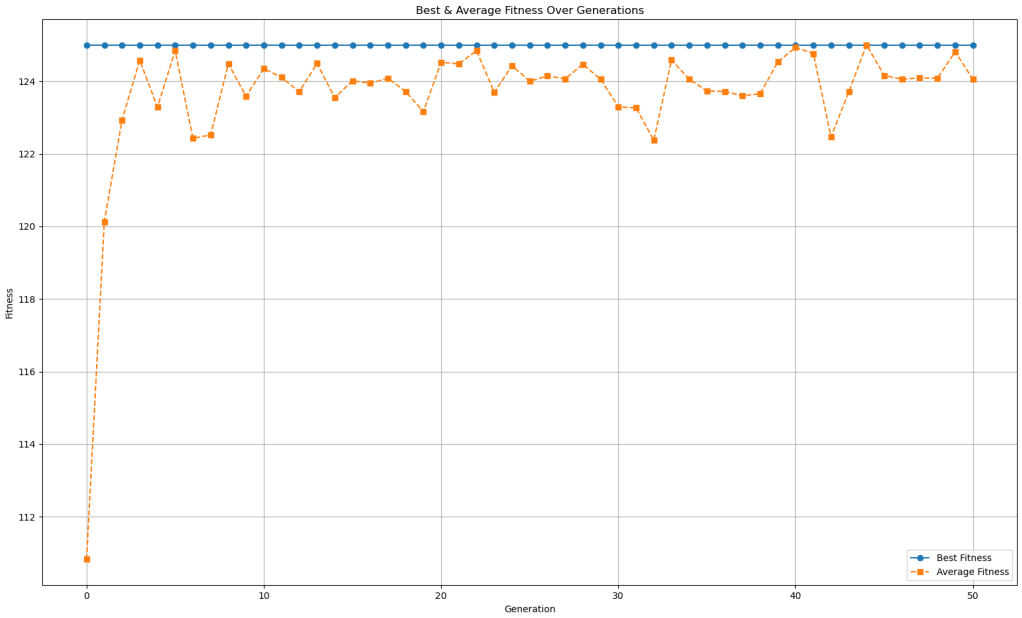

Figure 5: Fitness over generations

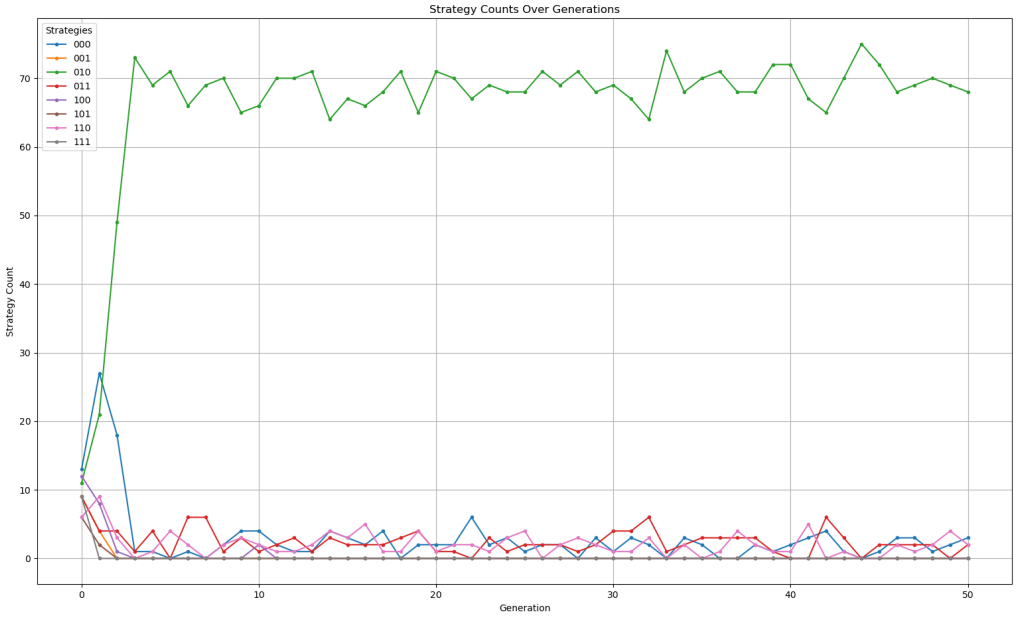

Figure 6: Diversity of the strategy population over generations

1.2.4 1× Always Defect, 1× Always Co-Operate, & 3× Tit-for-Tat

```
Best strategy: [1, 0, 1]
Fitness: 129
Generation: 0
[0, 0, 0]: 102
[0, 0, 1]: 117
[0, 1, 0]: 123
[0, 1, 1]: 120
[1, 0, 0]: 105
[1, 0, 1]: 129
[1, 1, 0]: 126
[1, 1, 1]: 120
```

Listing 4: 1× always defect, 1× always co-operate, & 3× tit-for-tat

When three Tit-for-Tats are added into the mix, we see Tit-for-Tat emerge as the dominant strategy, which is to be expected; it achieves a steady middle-ground approach via co-operation, and it has enough fellow co-operators to make up for the defection it suffers. The defector strategies get punished frequently enough to reduce their winnings from defection, allowing co-operation to become a dominant & winning strategy.

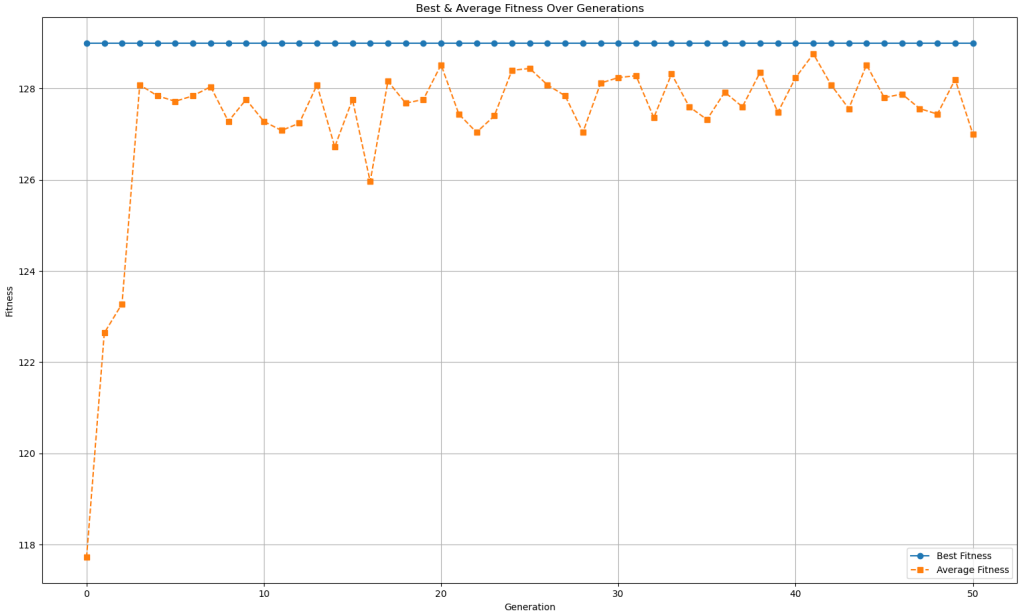

Figure 7: Fitness over generations

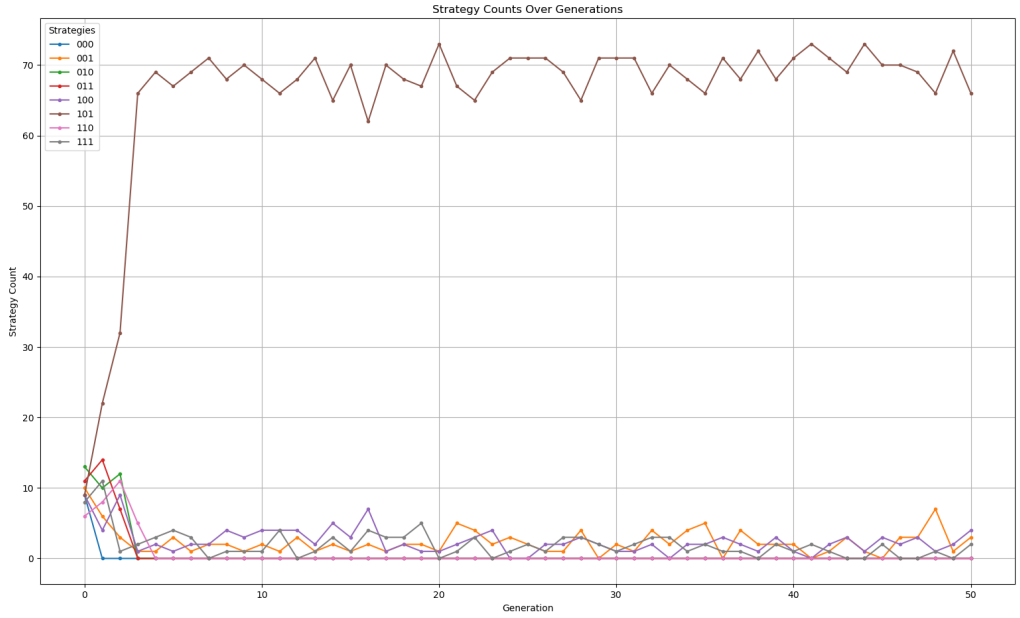

Figure 8: Diversity of the strategy population over generations

2 Part 2: Extension

To extend the genetic algorithm implementation, I chose to add a variable level of noise to the program, supplied via command-line argument `-n, --noise-level NOISE_LEVEL`. I then re-ran the same experiments as before with varying noise levels.

2.1 *NOISE_LEVEL* = 0.1

The results that I got for each of the previously-attempted experiments when I set the noise level to 0.1 were as follows:

- **Equally proportioned:** `[0,0,0]`.
- 2× **always defect:** `[0,0,0]`.
- 2× **always co-operate:** `[0,0,0]`.
- 3× **tit-for-tat:** `[1,0,1]`.

2.2 *NOISE_LEVEL* = 0.2

The results that I got for each of the previously-attempted experiments when I set the noise level to 0.2 were as follows:

- **Equally proportioned:** `[0,0,0]`.
- 2× **always defect:** `[0,0,0]`.
- 2× **always co-operate:** `[0,0,0]`.
- 3× **tit-for-tat:** `[0,0,0]`.

2.3 *NOISE_LEVEL* = 0.5

The results that I got for each of the previously-attempted experiments when I set the noise level to 0.5 were as follows:

- **Equally proportioned:** `[0,0,0]`.
- 2× **always defect:** `[0,0,0]`.
- 2× **always co-operate:** `[0,0,0]`.
- 3× **tit-for-tat:** `[0,0,0]`.

2.4 *NOISE_LEVEL* = 0.8

The results that I got for each of the previously-attempted experiments when I set the noise level to 0.8 were as follows:

- **Equally proportioned:** `[0, 0, 0]`.
- $2\times$ **always defect:** `[0,0,0]`.
- $2\times$ **always co-operate:** `[0,0,0]`.
- $3\times$ **tit-for-tat:** `[0,0,0]`.

As can be seen from the above outputs, the introduction of even just a little noise to each evolution immediately broke any possibility for co-operation, the one exception being that tit-for-tat still performed well against three other tit-for-tats at the lowest noise level, most likely because it got some co-operation in before noise disrupted the chain of co-operation. Noise makes co-operation more difficult, and is highly detrimental to these simple strategies defined by short bitstrings: these genomes don't have the necessary complexity to express a level of forgiveness, so one one bit of noise can destroy all chances of co-operation for the rest of the game. For a noisy environment, error-tolerant strategies are required, like generous tit-for-tat, which can avoid falling into the defection loops that overly rigid and/or grudging strategies fall into.