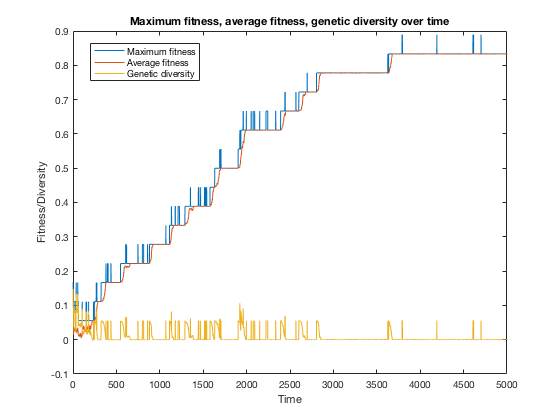
**Task 1.7. String Evolution Experiments**

**Try to answer the questions below and motivate your choice for the most suitable value for each parameter:**

We start with the following set of parameters (random seed = 42):

* Target phrase = ‘To be or not to be’
* Target phrase length = 18
* DNA bits ASCII = [32, 65:90, 97:122]
* Population size = 200
* Mutation rate = 0.01
* Max generations = 5000
* Mating factor = 10
* Breeding method = 0
* Kill factor = 0.1 (kill 10% of lowest fitness members)

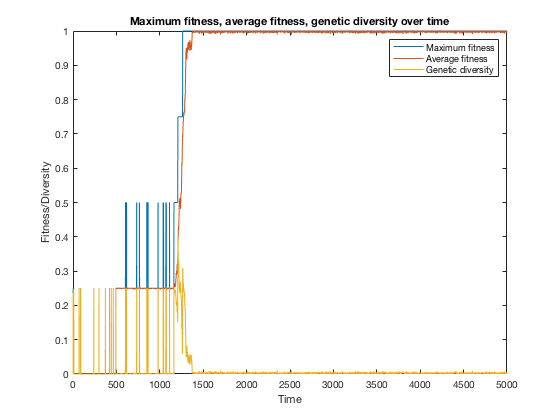
The plot for maximum and average fitness and genetic diversity over generations for the above parameters:



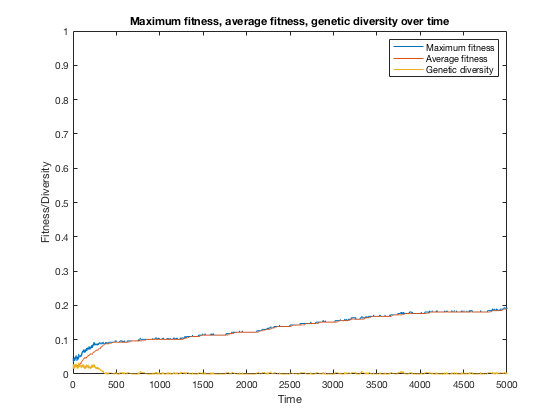
By the last generation, the maximum fitness is 0.8333 and average fitness is 0.8331 with a genetic diversity of 2.78e-4. The fittest string is “To be or noW tonZe”.

**1) Change the length of the target phrase**

- Target phrase = ‘Meow; Target phrase length = 4

As a result, we converge to the fittest string much quicker with a short phrase. We obtained the max fitness of 1.0 and the fittest string is an exact match by generation 1500.

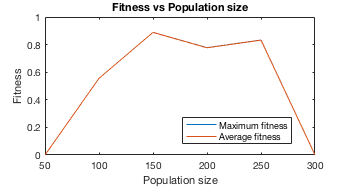
- Target phrase = ‘The universe we observe has precisely the properties we should expect if there is at bottom no design no purpose no evil and no good nothing but blind pitiless indifference DNA neither knows nor cares DNA just is And we dance to its music’; Target phrase length = 238 (Note: Punctuations are omitted to keep the DNA bits the same.)



As expected, longer phrase lengths like this one takes longer to evolve to the target phrase. After 5000 generations, we only have a max fitness of 0.1891 and average fitness of 0.1891. For the remaining experiments, we will go with the phrase ‘To be or not to be’ as the target.

**2) Change the number of population members**

We test the following population size array: [50, 100, 150, 200, 250, 300]. Fitness vs population size (at end of time) plot below shows that the higher the population the higher fitness we’ll get at the end of the generations. However, we see a decrease in fitness when the population size is above 250. Thus, the best range for this parameter would be around 150-250 members in the population.



**3) Change the mutation rate**

We test the following mutation rate array with varying magnitudes: [0, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5]. Fitness vs mutation rate (at end of time) plot below shows that up until the value of around 0.05, the higher the mutation rate the higher fitness we’ll get at the end of the generations. Interestingly, the peak fitness is at around mutation rate of around 0.025 for average fitness and 0.05 for max fitness, and increasing the rate from 0.05 on shows a decrease in fitness.

|  |  |  |
| --- | --- | --- |
| **Mutation rate** | **Maximum fitness at gen=5000** | **Average fitness at gen=5000** |
| **0** | **0.0000** | **0.0000** |
| **0.001** | **0.0000** | **0.0000** |
| **0.0025** | **0.0000** | **0.0000** |
| **0.005** | **0.0000** | **0.0000** |
| **0.0075** | **0.7222** | **0.7053** |
| **0.01** | **0.8333** | **0.8331** |
| **0.025** | **0.9444** | **0.9314** |
| **0.05** | **1.000** | **0.9183** |
| **0.075** | **0.5556** | **0.4119** |
| **0.1** | **0.6111** | **0.4269** |
| **0.25** | **0.0556** | **0.0011** |
| **0.5** | **0.1111** | **0.0197** |

**4) Change the range of possible characters being considered (include numbers, etc)**

We test the ‘alphabet’ (aka range of DNA bits) to varying ASCII ranges:

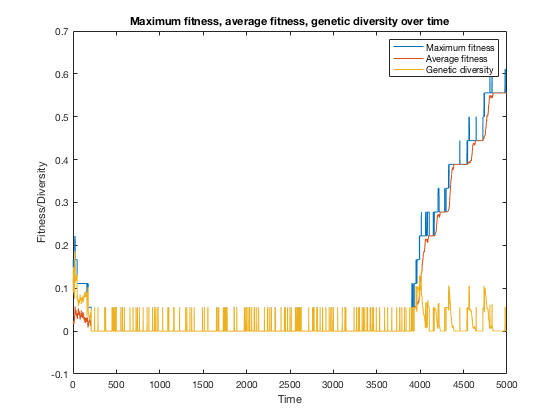
* ASCII vector [32:126] for space, special characters/punctuations, numbers, upper and lowercase characters: At the end of the generations, the max fitness is 0.0556 and the average fitness is 2.78e-4.
* ASCII vector [32, 48:57, 65:90, 97:122] for space, numbers, upper and lowercase characters: At the end of the generations, the max fitness is 0.8889 and the average fitness is 0.8886.
* ASCII vector [32, 65:90] for only space and uppercase characters: At the end of the generations, the max fitness is 0.0656 and the average fitness is 2.88e-4.
* ASCII vector [32, 97:122] for only space and uppercase characters: At the end of the generations, the max fitness is 0.9444 and the average fitness is 0.9444. Given the target phrase we have, this is the best range of possible characters since most of the target characters are lowercase and spaces.

**5) Try changing between the two breeding methods ‑ which one works better?**

Using the original set of parameters.

Method 0: See the plot at the beginning of this report. As above-mentioned, by the last generation the maximum fitness is 0.8333 and average fitness is 0.8331 with a genetic diversity of 2.78e-4. The fittest string is “To be or noW tonZe”.

Method 1: At the last generation, the maximum fitness is 0.5556 and average fitness is 0.5556 with a genetic diversity of 7.77e-16. The fittest string is “BQ zi oI not Fi bB”. This method probably needs even more generations to evolve.



**6) Change the mating factor ‑ what benefit might we get from increasing this?**

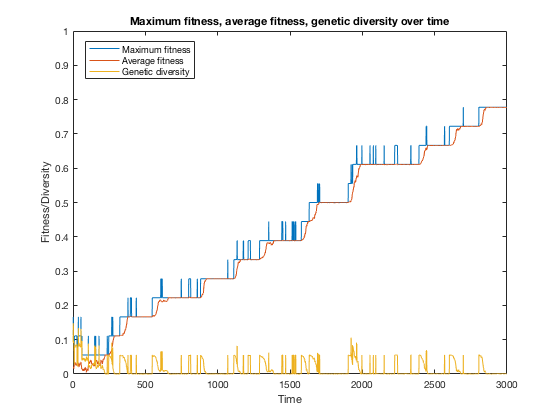
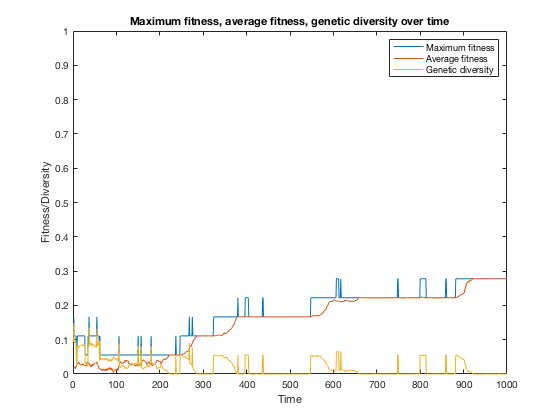
**What’s a reasonable value for it?**

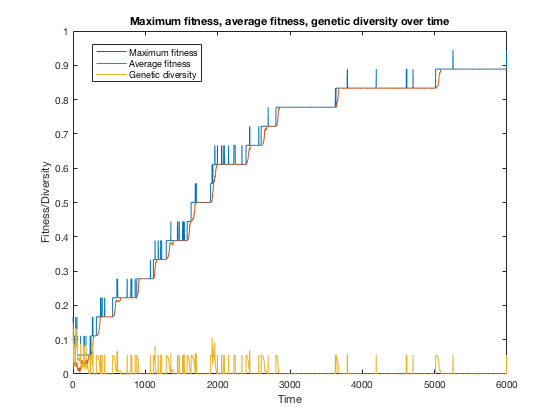
Using the original set of parameters.

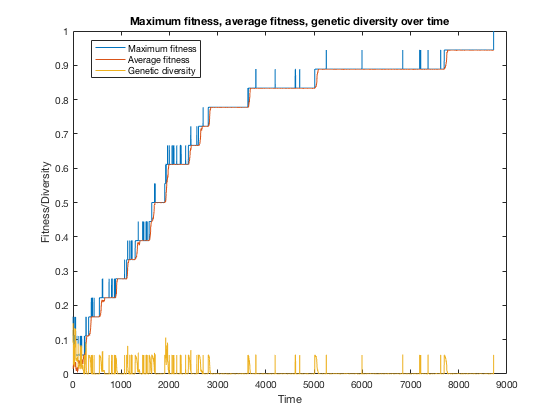
|  |  |  |
| --- | --- | --- |
| **Mating factor** | **Maximum fitness at gen=5000** | **Average fitness at gen=5000** |
| 10 | 0.8333 | 0.8333 |
| 20 | 0.8333 | 0.8333 |
| 50 | 0.8333 | 0.8333 |
| 100 | 0.8333 | 0.8333 |

**7) Change the maximum generations ‑ what happens to fitness over time?**

Using the original set of parameters.







For 10,000 generations, we have evolved the target string by generation 8722 and the algorithm is stopped early.

**8) Which function takes the longest to run? Can you improve its runtime at all?**

At first, calculateFitness() took 0.000898 seconds each time it is called because I used a for loop within for loop to compare each character in the organism and target strings to calculate the fitness score. I have replaced this with fitness\_score = sum(organism\_str == target\_str) / target\_len;.

Now,

buildPopulation() took 0.000463 seconds.

calculateFitness() took 0.000136 seconds.

buildMatingPool() took 0.000246 seconds.

breed() took 0.000478 seconds.

causeMutation() took 0.000294 seconds.

Plotting took 0.144744 seconds.

Writing data to text file took 0.029518 seconds.

After modifications, out of the functions, buildPopulation() and breed() took longest. Overall, plotting took longest.

**To make our code much more efficient at selecting the best parents, we can force our population to favor slightly fitter members much more than everyone else. After calculating our fitness, which are numbers from 0 to 1, we can raise it to some power, let’s say 3. Now values which are close in fitness will be much easier to differentiate between.**

**Although the fitness values themselves got smaller, the relative difference between them became larger. Remembering that the mating pool normalizes fitnesses to the maximum fitness, this will now award 0.33 even more tickets in the raffle compared to 0.30 or 0.27. In your script, try raising your entire fitness vector to different values before the mating pool is built (we’ll call this the ‘exponential factor’). How high can you make this value before it stops becoming beneficial? How might you want to adjust your mating factor (from Task 1.3) after introducing this exponential factor? Write your observations in the report.**

Mating factor = 10

|  |  |  |
| --- | --- | --- |
| **Exponential**  **factor** | **Maximum fitness at gen=5000** | **Average fitness at gen=5000** |
| 1 | 0.8333 | 0.8331 |
| 2 | 0.8333 | 0.8333 |
| 3 | 0.9444 | 0.9442 |
| 4 | 0.9444 | 0.9442 |
| 5 | 0.7778 | 0.7775 |
| 6 | 0.6111 | 0.6108 |

Overall, the best parameters from these isolated experiments are:

Target phrase = ‘To be or not to be’

Population size = 150 - 250

Mutation rate = 0.025 - 0.05

ASCII range = [32, 97:122]

Breeding method = 0

Mating factor = 10

Max generations = 10,000

Exponential factor = 3, 4

**Task 2.5. Image Evolution Experiments**

pop\_size = numel(target);

channel\_size = target\_x \* target\_y;

DNA\_bits = [0 255];

max\_gen = 501;

mating\_factor = 10;

exponential\_factor = 2;

breeding\_method = 1;

mutation\_rate = 0.01;

kill\_factor = 0.98;   % Only breed top x% to save time

tolerance\_1 = 10;

tolerance\_2 = 50;

mutation\_range = [0 30]; % Brighten/darken a pixel by an amount

random\_mutation\_rate = 0.25;

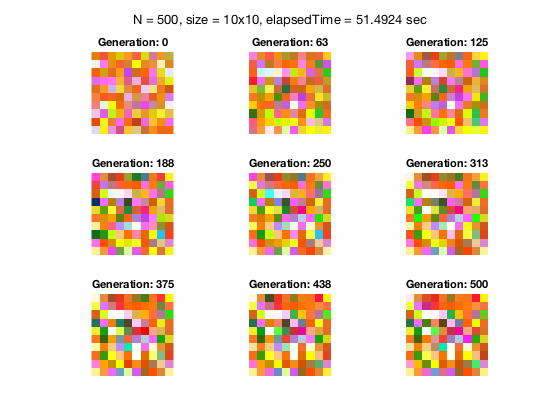
Fitness measure = fit1 (pixel to pixel comparison with tolerance)

Note: For memory, DNA bits are made to be weighted based on target image.

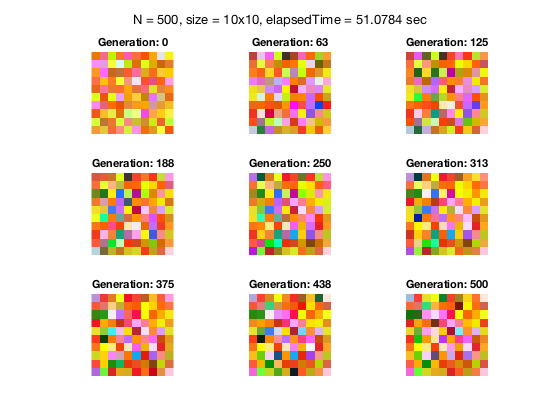
Note: These parameters remain constant for each test below, except when a better value is found (denoted by **Best**) then the rest of the tests will use that value with the rest of the parameters above.

1. Testing various combinations of fitness measures

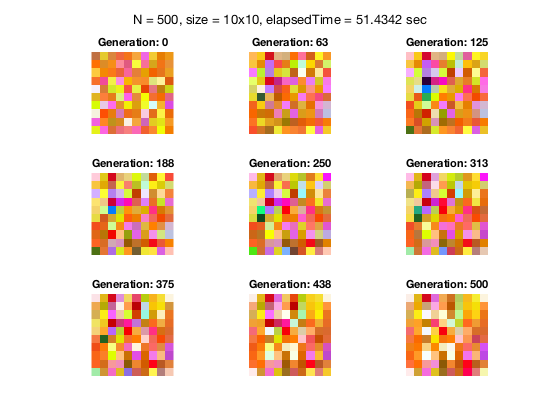
Original fitness measure (pixel by pixel): Gen- 500, MaxRed- 0.24, AvgRed- 0.23, MaxGreen- 0.24, AvgGreen- 0.23, MaxBlue- 0.22, AvgBlue- 0.21



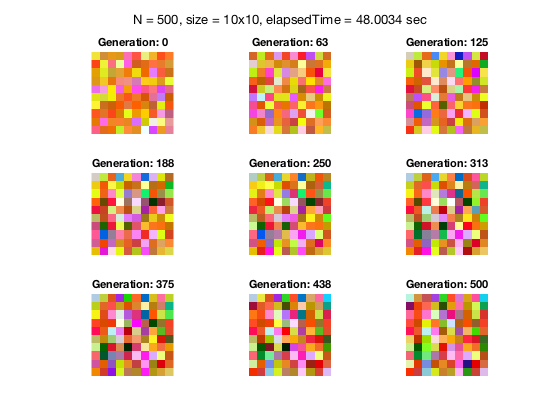
Fitness measure 1 and 2 (mean filter): Gen- 500, MaxRed- 0.25, AvgRed- 0.23, MaxGreen- 0.21, AvgGreen- 0.17, MaxBlue- 0.47, AvgBlue- 0.46



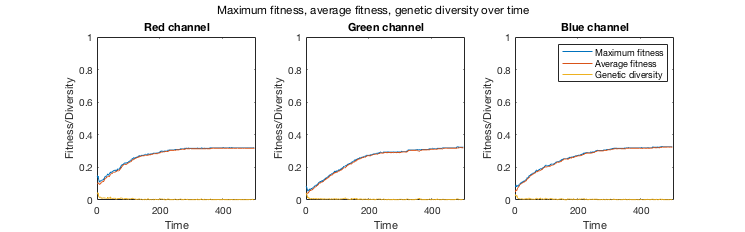
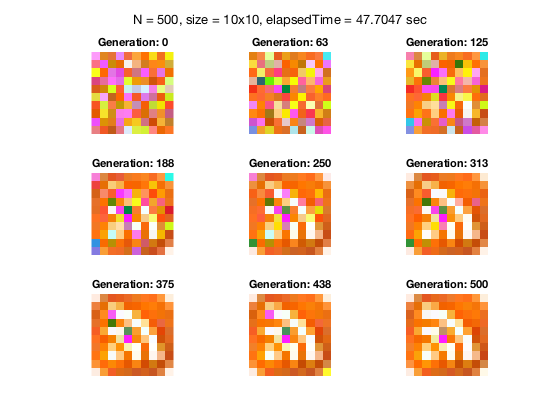
Fitness measures 1, 3, and 4 (rate of change up/down and left/right): Gen- 500, MaxRed- 0.50, AvgRed- 0.50, MaxGreen- 0.46, AvgGreen- 0.45, MaxBlue- 0.38, AvgBlue- 0.38



Fitness measures 1-4: Gen- 500, MaxRed- 0.37, AvgRed- 0.36, MaxGreen- 0.23, AvgGreen- 0.22, MaxBlue- 0.51, AvgBlue- 0.50



Fitness measure 1 (pixel by pixel with tolerance): Gen- 500, MaxRed- 0.33, AvgRed- 0.33, MaxGreen- 0.33, AvgGreen- 0.33, MaxBlue- 0.33, AvgBlue- 0.33 (**Best outcome despite lower fitness values**)

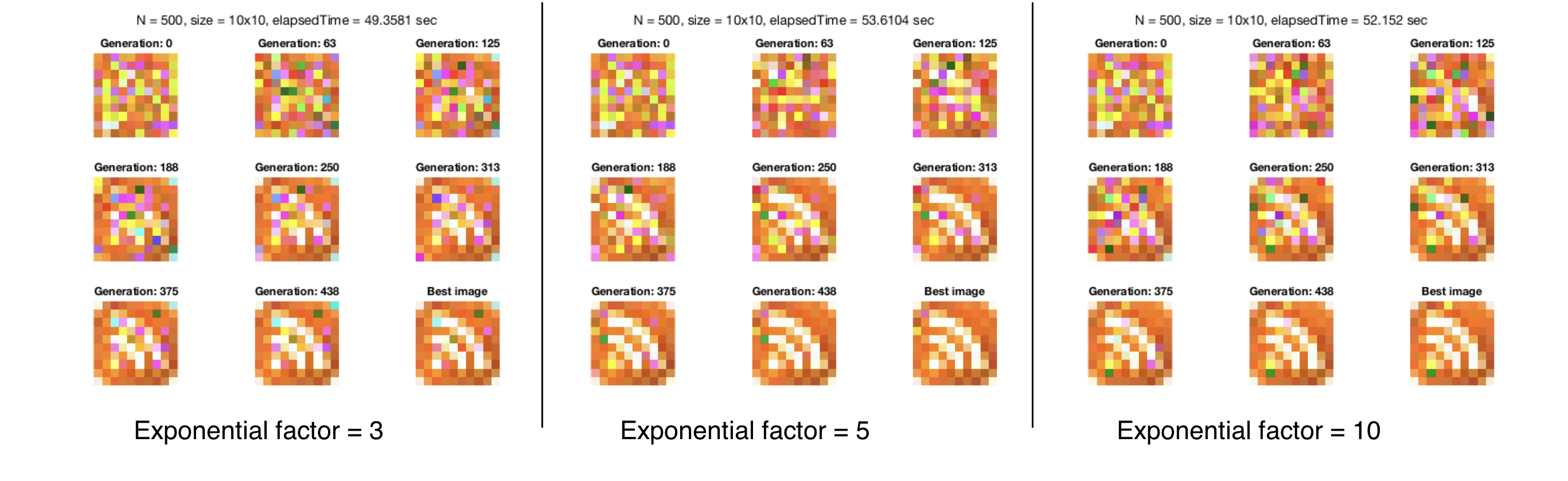


1. Exponential factor

Exponential factor = 3: Gen: 500, MaxRed: 0.32, AvgRed: 0.32, MaxGreen: 0.32, AvgGreen: 0.32, MaxBlue: 0.33, AvgBlue: 0.32

Exponential factor = 5: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33 (**Best)**

Exponential factor = 10: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33



1. Breeding method = 0

Gen: 500, MaxRed: 0.33, AvgRed: 0.32, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33. No significant improvement from using method 1.

1. Mutation rate

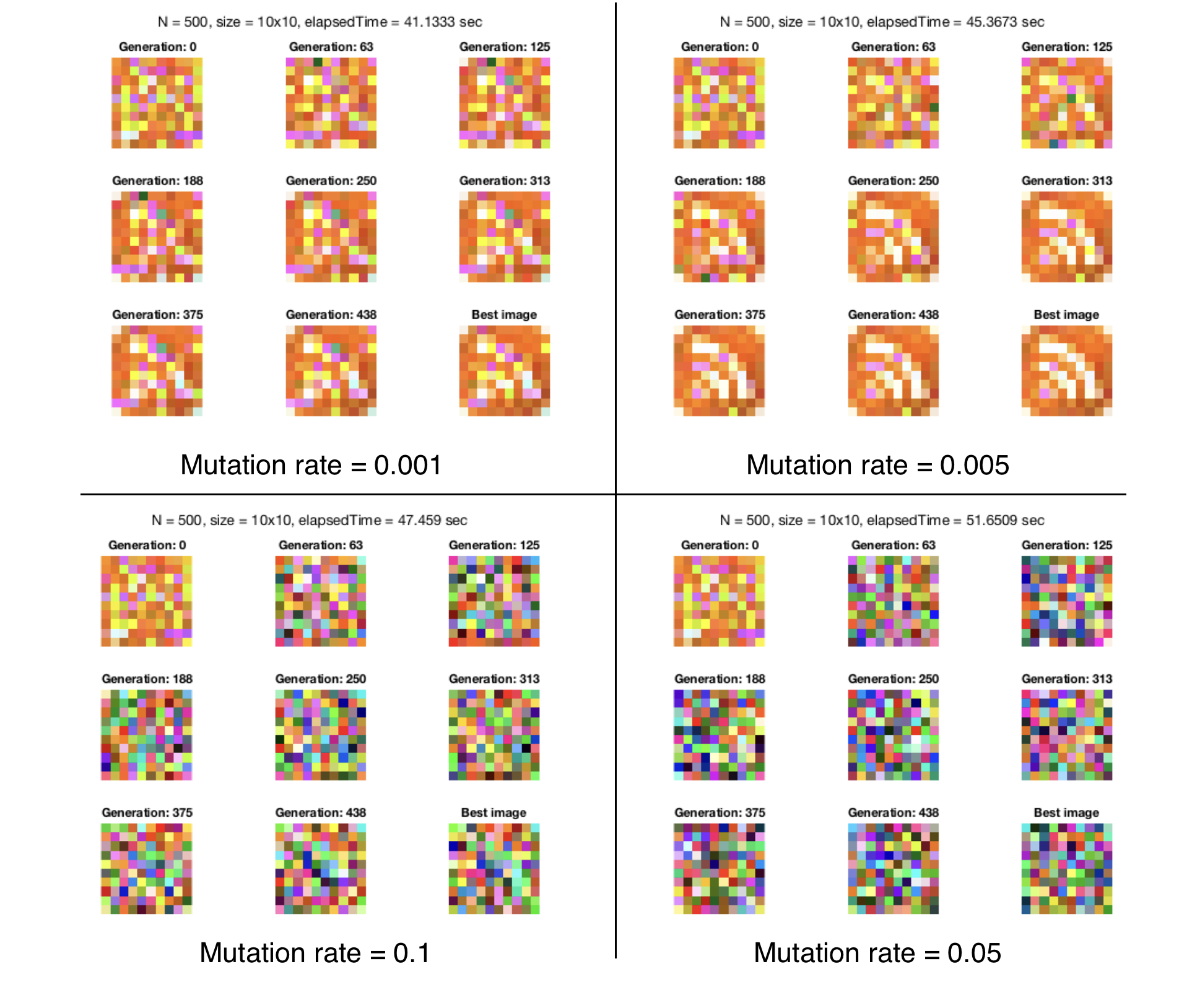
Going below 0.1 makes no big difference and going above 0.01 actually yields worse results, shown below.

Mutation rate = 0.001: Gen: 500, MaxRed: 0.31, AvgRed: 0.31, MaxGreen: 0.30, AvgGreen: 0.30, MaxBlue: 0.29, AvgBlue: 0.29

Mutation rate = 0.005: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33

Mutation rate = 0.05: Gen: 500, MaxRed: 0.11, AvgRed: 0.09, MaxGreen: 0.10, AvgGreen: 0.09, MaxBlue: 0.10, AvgBlue: 0.09

Mutation rate = 0.1: Gen: 500, MaxRed: 0.33, AvgRed: 0.32, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33.

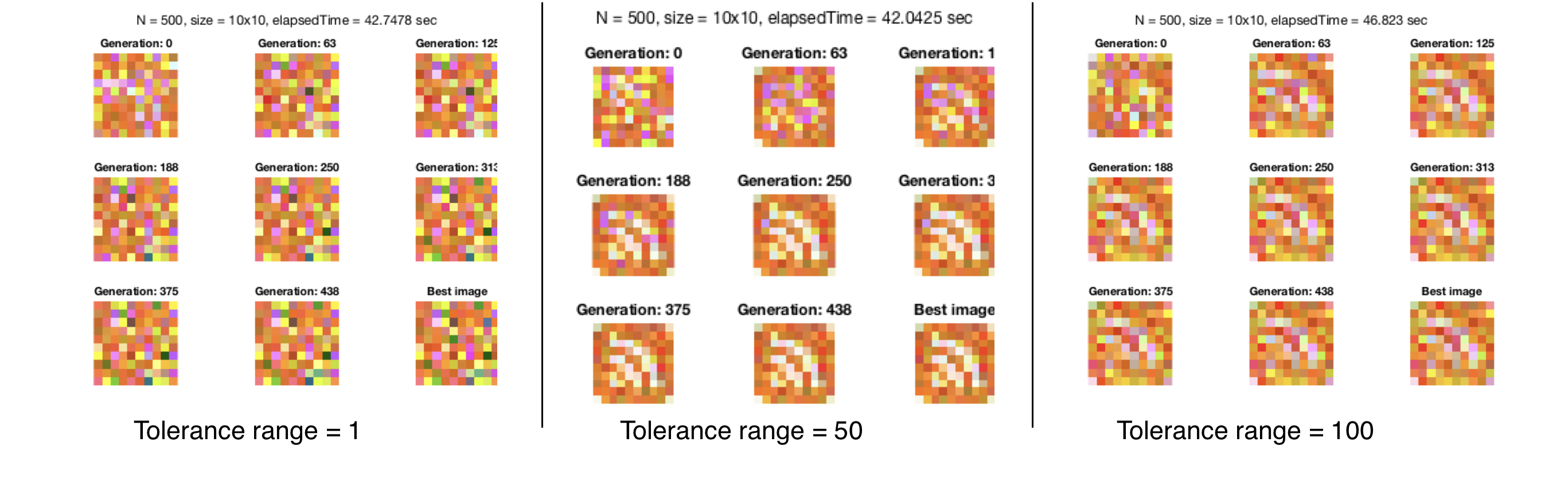


1. Tolerance range

Tolerance range = 1: Gen: 500, MaxRed: 0.33, AvgRed: 0.32, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33.

Tolerance range = 50: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33

Tolerance range = 100: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33. Making tolerance really high yields the same fitness values, but we get worse results in image reconstruction.



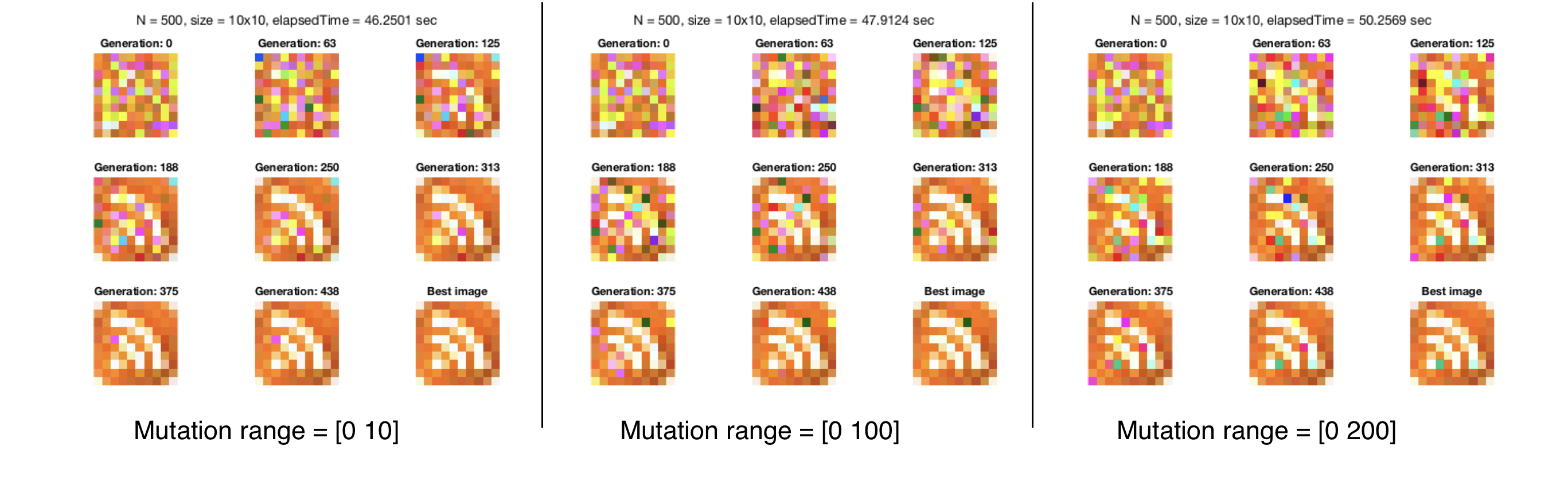
1. Mutation range

Increasing the range lets some random pixels show up, but because the mutation rate is very low this makes no big difference. Setting a smaller range allows the most accurate reconstruction.

Mutation range = [0 10]: Gen: 500, MaxRed: 0.33, AvgRed: 0.32, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33.

Mutation range = [0 100]: Gen: 500, MaxRed: 0.33, AvgRed: 0.32, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33.

Mutation range = [0 200]: Gen: 500, MaxRed: 0.33, AvgRed: 0.32, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33.



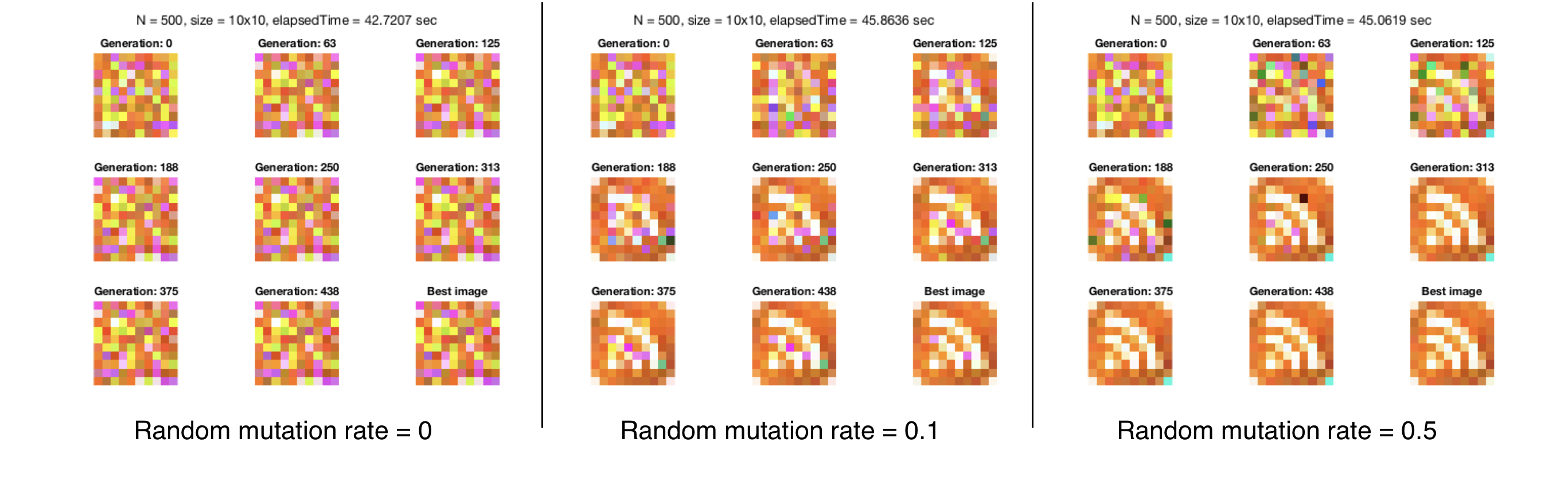
1. Random mutation rate

With no random mutation rate, we obtain poor results in fitness values as well as image reconstruction.

Random mutation rate = 0: Gen: 500, MaxRed: 0.22, AvgRed: 0.22, MaxGreen: 0.13, AvgGreen: 0.13, MaxBlue: 0.15, AvgBlue: 0.15

Random mutation rate = 0.1: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33

Random mutation rate = 0.5: Gen: 500, MaxRed: 0.33, AvgRed: 0.33, MaxGreen: 0.33, AvgGreen: 0.33, MaxBlue: 0.33, AvgBlue: 0.33



**For this 10x10 the best parameters are:**

max\_gen = 501;

mating\_factor = 10;

exponential\_factor = 5;

breeding\_method = 0;

mutation\_rate = 0.01;

kill\_factor = 0.98;   % Only breed top x% to save time

tolerance\_1 = 10;

tolerance\_2 = 50;

mutation\_range = [0 10]; % Brighten/darken a pixel by an amount

random\_mutation\_rate = 0.5;

fitness measure 1

**What I learned…**

* Vectorization is empowering!
* To see a target image reconstructed, fitness values do not have to reach 1 unless we implement the pixel-by-pixel exact matching method as for strings. It’s more informative to observe the images themselves over the generations from the 9-image figure.
* As with all algorithms, the parameter space for this genetic algorithm can be huge.
* I need to upgrade my computer.