# Comparing Genetic Algorithms
# CMP_SC 4770

**Jonas Ferguson**
**JFR48**

University of Missouri

# 1 Introduction

## 1.1 Evolutionary Algorithms Overview

Evolutionary algorithms are an attempt to take biological concepts and apply them to optimize solving problems. Evolutionary algorithm take two key concepts from biological evolution: heredity and natural selection. Any given individual inherits genes from its parent(s). Natural selection posits that individuals with traits that are more 'fit' for the given situation will live on, while less fit traits will die out. Fitness is heavily determined by environment, and traits are influenced by the inherited genes of an individual. Conclusively, this means that if we leave a population for long enough, it will, theoretically, work to find the best genes to survive in its environment.

On the technological side, when we deal with functions and optimization problems at a grand scale—several dimensions across vast ranges–it quickly becomes impractical to search all possible solutions, and we must rely on specific techniques to find these optimal solutions. The biological concept of evolution offers a useful tactic. By simulating a population of individuals over multiple generations, allowing individuals to inherit genes from parent(s), evaluating fitness to determine which individuals will get to pass along genes, and creating a specific environment tailored to the problem, we can exploit these evolutionary concepts to find answers to a wide array of problems.

Evolutionary algorithms are the result of combining evolution concepts with optimization problems. Evolutionary algorithms simulate a population over many generations. Each individual has chromosomes which hold information about potential solutions. As generations play out, these individuals are evaluated for their fitness, some individuals are allowed to pass on their chromosomes, various mutation and crossover parameters are used to mix up the gene pool, and the cycle is continued. This process leaves lots of room to vary the mechanics to better suit a problem. Many such possibilities will now be viewed in greater detail.

## 1.2 Initialization

The first issue presented when writing an evolutionary algorithm is determining where to place the individuals at the start. Many different strategies have been formed to solve this issue, involving different perspectives on randomness and how to incorporate known data.

When not using any external data and given a known range of a function, we have at least three different strategies that we can use: bounded uniform, Poisson, and grid.

Bounded uniform will generate uniformly random points along the range of a function. In my project, this is implemented with the following code.

```
for i in range(size):
    chromosome = (np.random.rand(2) - 0.5) * 2.0 * functionRange
    population.append(chromosome)
```

For a function with range (-5, 5), functionRange is set to 5, which results in chromosome creating coordinates uniformly random between -5 and 5.

Critics of the bounded uniform strategy cite a theoretical case in which too many points are concentrated together, and other spaces are left wide open. The Poisson initialization aims to solve this be swapping a uniform random distribution for a Poisson random distribution, and thus better filling in the gaps and spacing points around the function range.

Finally, the grid approach takes the random element away entirely, and spreads all the individuals evenly around the full range of the function in a grid, in an attempt to best cover the possible

range of solutions.

```
dimension = int(np.sqrt(size))
for i in range(dimension):
    for j in range(dimension):
        chromosome = (((-1.0 * functionRange) +
            ((functionRange / (dimension/2)) * i) +
            (functionRange / dimension)),
            ((-1.0 * functionRange) +
            ((functionRange / (dimension/2)) * j) +
            (functionRange / dimension)))
        population.append(chromosome)
```

This code will create coordinates that span the XY plane with a grid set by first finding the lower bound, adding even segments to it, centering this grid around the origin. This will evenly distribute points around the solution space in the most efficient manner possible.

While these initialization methods can work quite effectively, with problems where there is existing data, we can use this data to provide a head start to the genetic algorithm. Although I did not explore these methods for this project, using existing data to thoughtfully initialize points can be very effective.

One common method for initialization using existing data is the exemplars method. This method takes pairs of points from an existing data set. The points taken will be the two points most different from each other. This continues until all needed individuals are selected. This can help initialize points that are more varied with the extra information that previous data can provide.

## 1.3 Fitness Scoring

In order to have an evolutionary algorithm at all, the individuals must be working towards optimizing some problem. The level to which a given individual optimizes a problem is its fitness. In order to run an evolutionary algorithm, at each generation we much test the fitness of the individuals.

I have implemented genetic algorithms which find the minimum of functions in the XY plane. This can also be easily attempted for finding a maximum by taking the inverse of the function result. The three functions that I have used are the Himmelblau's Function, the Ackley Function, and the Beale function.

Himmelblau's function is the equation:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

It has a range of $-5 <= x, y <= 5$ and has 4 global minimum points, all at 0. This function is a way to see how the algorithm performs when there are multiple different minima, which aren't in coordinate alignment with each other.

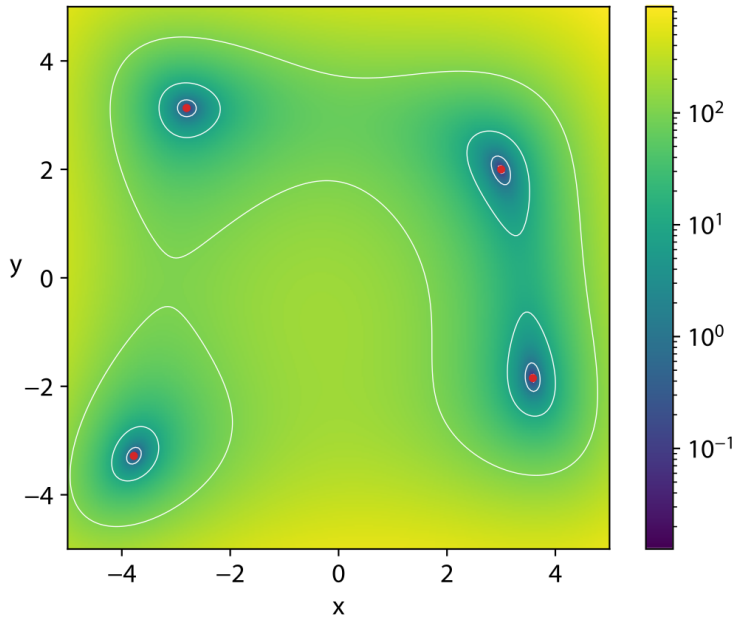



Figure 1: Himmelblau's Function

The Ackley function is the equation:

$$f(x, y) = -20 \exp[-0.2\sqrt{0.5(x^2 + y^2)}] - \exp[0.5(\cos 2\pi x + \cos 2\pi y] + e + 20$$

In contrast to Himmelblau's function, the Ackley function has only one global minimum point but has many local minima, with a minimum value of 0. The range is also $-5 <= x, y <= 5$. Ackley's function can be helpful for testing to see if a genetic algorithm gets stuck in the local minima, or if it explores enough to find the central minimum point.

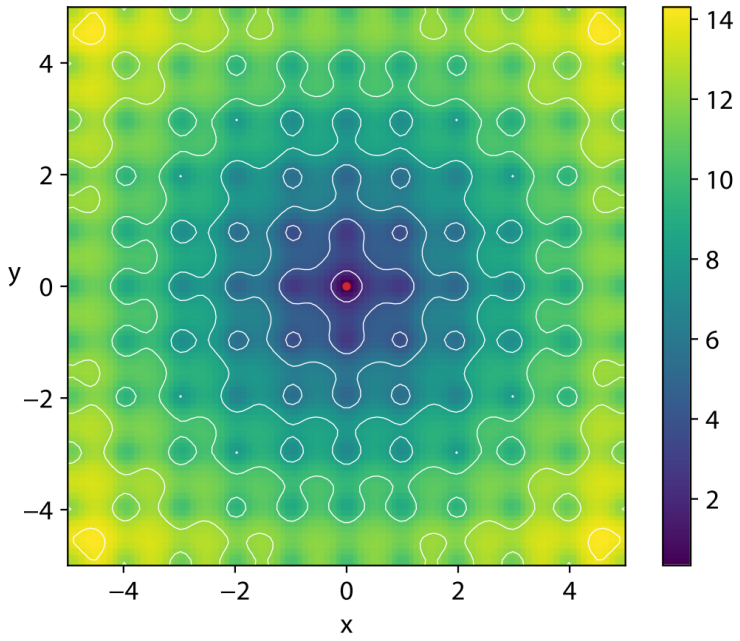



Figure 2: Ackley Function

Finally, I implemented the Beale function, which is shown below:

$$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$$

The Beale function 2 local minima, but one is a global minimum. Neither are aligned with each other or the coordinate grid, making the Beale function a good test function for something that is not aligned with the grid. It has a range of $-4.5 <= x, y <= 4.5$ and 1 global minimum at 0.

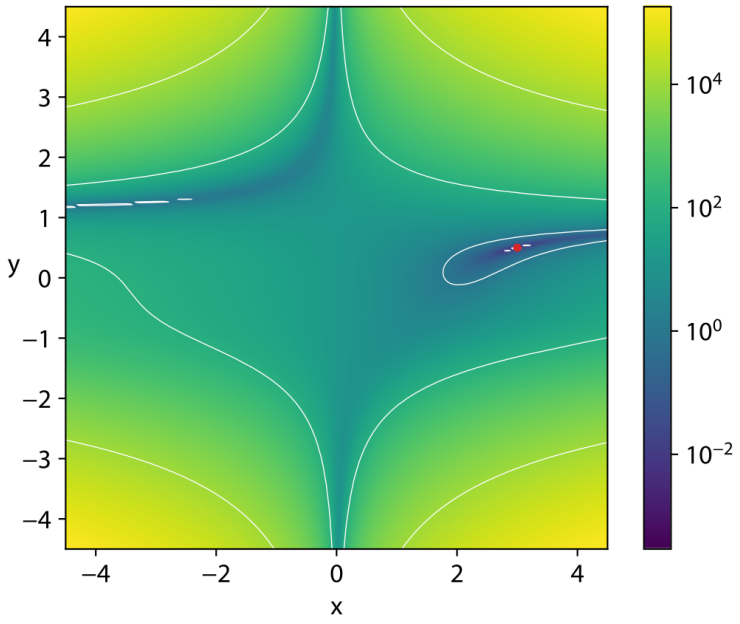



Figure 3: Beale Function

## 1.4 Selection

Once the fitness of all individuals has been determined, we must next select which individuals will continue on to the next generation. In all of my implementation, I first made a duplicate of every individual and then used various methods to narrow that back down to the original number of individuals.

### 1.4.1 Roulette Wheel Selection

The first selection method that I implemented was Roulette Wheel Selection, or RWS. RWS divides up a wheel into segments. Each segment represents one individual, and the size of the segment is proportional to its calculated fitness. Then the wheel is spun and a random spot along it determined. Thus leaving all individuals some chance of being selected, but still giving greater weight to the more fit.

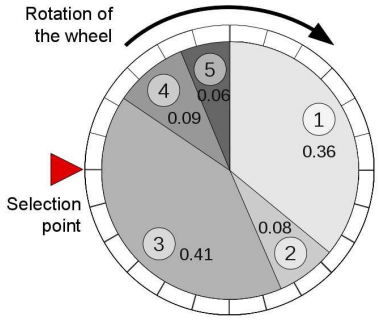



Figure 4: Visualization of Roulette Wheel Selection

This diagram provides an example of RWS with 5 individuals of the shown fitness values. The fitness values are normalized so that they total to 1, allowing a random point from 0 to 1 to be chosen as the selection point.

My genetic algorithm uses the following implementation of RWS:

```
max_val = np.sum(scores)
for i in range(1,len(scores)):
    for k in range(2):
        pick = random.uniform(0, max_val)
        current = 0
        for j in range(len(scores)):
            current += scores[j]
            if current > pick:
                break
        population_nextgen.append(pop_after_fit[j].copy())
```

This code implements the same concept, but instead of normalizing the fitness values, it instead calculates the maximum fitness value and then iterates through each individual to find an individual that matches a randomly selected fitness value.

### 1.4.2 Tournament Selection

An alternative method of selection is tournament selection. Tournament selection selects n individuals and then has them 'compete' by determining the fittest of the selected individuals. This process is repeated until all of the next generation has been selected. Tournament selection aims to keep individuals with high fitness levels, while maintaining some diversity, and rejecting the least fit individuals.

Selection of the n value has substantial impact on tournament selection. An n value of 1 is equivalent to pure random, whereas an n value infinitely large is a purely elitist strategy. For my implementation, I chose a value of 5, although further exploration could be done to determine the optimum value for this parameter.

My implementation of tournament selection is as follows:

```
n = 5
for i in range(1,2 * len(scores)):
    best = random.randint(0, len(scores)-1)
    for j in range(n-1):
        pick = random.randint(0, len(scores)-1)
```

```
        if scores[pick] > scores[best]:
            best = pick
    population_nextgen.append(pop_after_fit[best].copy())
```

This implementation will run a tournament for each individual, selecting 5 values from the pool of individuals (any given individual may be selected multiple times), and evaluating the fittest of the 5 individuals to continue.

### 1.4.3 Elitism

While not an entire strategy by itself, elitism is an important concept when it comes to selection in genetic algorithms that can be used in tandem with other strategies. Elitism is the concept of preserving the best performing member of a generation regardless of general selection strategy, to make sure that the best solution is never lost. I have implemented elitism with both of my selection types in order to ensure that I am keeping my best result.

## 1.5 Crossover

### 1.5.1 Crossover Methods

Once the parents have been selected, we perform crossover and mutation in order to modify the individuals and continue working towards an improved answer. Crossover uses multiple parents to create offspring, whereas mutation modifies individuals directly.

The simplest method of crossover is known as one-point crossover. A random point is selected along the chromosome of two parents, and the genes on one side of the point are swapped between the individuals. The downside of this method is that it can often leave the ends of the chromosomes untouched.

This can be extended to two-point crossover, which selects two random points along the chromosome, and swaps the inside or outside of the two points with another individual. This helps balance out which genes will get swapped.

Finally, we can continue to multipoint crossover which continues to divide chromosomes into more pieces, swapping sections between them from individual to individual.

An entirely different method, uniform crossover, crosses over genes without any regard to nearby genes. First, a random binary mask is created to match up with all the genes. Where a 1 is present in the mask, the genes are swapped. Where a 0 is present, the genes remain as they are. This also provides the opportunity to more finely adjust how many genes get crossed over and takes away the position of a gene from being connected to how it can be crossed over.

Shifting away from gene swapping, instead we can use multiple parents to find a new individual that is placed along the middle. One such technique like this is that of floating point. Floating point chooses a random spot on a line between 2 points, and can be represented with the equation:

$$x_{ij} = U(0,1)(x_{2j}(t) - x_{1j}(t)) + x_{2j}(t)$$

### 1.5.2 Crossover Implementation

For this project, as there are only 2 dimensions, the potential for crossover is significantly limited. Most of the crossover strategies cannot really be applied as such a small scale. Instead, I have implemented modified versions of one-point crossover, and of uniform crossover.

The one-point crossover method as it has been implemented chooses either the x-axis or the y-axis randomly, and then swaps the gene for this coordinate between two individuals. It will always result in a crossover along one of the axis for each individual. The code for this implementation is shown below:

```
for i in range(1,sz):
    child = pop_after_sel[2*i+0].copy()
    parent2 = pop_after_sel[2*i+1].copy()
    loc = np.random.randint(0,1)
    child[loc] = parent2[loc]
    population_nextgen.append(child)
```

The other crossover method that I implemented is uniform crossover. It will randomly generate the binary mask for the 2 coordinates, and then swap them at a 50% rate. This allows for more variation in crossover. Sometimes no axes are crossed, sometimes it is just one, and sometimes both are crossed. The code for this is shown below:

```
for i in range(1,sz):
    child = pop_after_sel[2*i+0].copy()
    parent2 = pop_after_sel[2*i+1].copy()
    for j in range(2):
        if (np.random.randint(0,1) == 1):
            child[j] = parent2[j]
    population_nextgen.append(child)
```

## 1.6 Mutation

In addition to crossover, we can modify the individual on its own through mutation. While crossover does a great job of connecting pieces of existing individuals together, in order to make sure we are exploring the entire space, some random adjustments within the chromosomes of individuals are necessary.

In order to make sure that solutions do not change wildly away from what has been shown to work already, mutation is only done at a specified rate, between 0 and 1. For my project, I kept this rate at 0.2, meaning that each individual has a 20% chance of being mutated, although further work could be done to optimize this parameter.

Individuals can be mutated in many ways. One simple way is uniform mutation. This is mutating an individual anywhere within the range of the function with equal probability. It is essentially disregarding a section of the population completely and replacing them with random points, completely independent of existing data. While this method can be helpful in exploring outside of where the rest of the population is, it is uninformed by any existing data, thus wasting information that could help optimize the mutation better.

Below is the implementation which I used for uniform mutation:

```
for i in range(1,sz):
    chromosome = pop_after_cross[i].copy()
    for j in range(len(chromosome)):
        if random.random() < mutation_rate:
            chromosome[j]= ((( np.random.rand() - 0.5 ) * 2.0 ) * functionRange )
    population_nextgen.append(chromosome.copy())
```

This program randomly selects if a chromosome will get mutated (based on the mutation rate), and then picks a random location within the function range (as specified by a pre-defined functionRange parameter which is specific to each function). One important thing to notice about this code is that mutation is done for each axis independently, meaning that any individual has a 20% chance of having its x coordinate changed and a 20% chance of having its y coordinate changed. Swapping this order tended to produce substantially worse results.

In contrast to uniform mutation is Gaussian mutation. Instead of mutating randomly across the entire range of the function, Gaussian mutation uses a normal distribution to create random mutations that remain informed by existing knowledge.

In my implementation of Gaussian mutation, I ran into some issues early on. When basing the mutation on the standard deviation of the population, the entire population quickly centered at one point, never exploring further. As a result, I decided to use a static standard deviation instead. Setting this value too high would produce lots of solutions that left the bounds of the function and had to be clipped back into the bounds. Setting it too low meant not getting to explore the full range of the function. I settled on balancing this by deciding the standard deviation for each mutation by dividing the function range by a random integer from 1 to 10. This means that some mutations will explore all over the function range, but that others will move only slightly, attempting to exploit already known information.

Combining these ideas together into my Gaussian mutation implementation produces the following code:

```
for i in range(1,sz):
    chromosome = pop_after_cross[i].copy()
    for j in range(len(chromosome)):
        if random.random() < mutation_rate:
            chromosome[j] += np.random.normal(loc=0,
                scale=functionRange/random.randint(1, 10))
            chromosome[j] = np.clip(chromosome[j],
                -functionRange, functionRange)
    population_nextgen.append(chromosome.copy())
```

## 1.7 Termination

Deciding when to end an evolutionary algorithm is also a crucial decision. Ending the algorithm too soon will remove the potential to improve the answer, but running it too long risks wasting time and resources on something that is not going to see substantial improvement. Depending on the problem, multiple different termination methods can be used.

The simplest termination method is the N-Generations method. After an algorithm has run N generations, it terminates, regardless of if it has been stagnant, is actively growing, or any other factor. It is crucial to find the right value for N that balances improvement with resource needs.

The code for implementing this is very simple, we just need to run the generation loop N times. I have used this method for my algorithm, varying the number of generations needed based on the function.

An alternate method is fitness-based termination. Instead of terminating after a specific number of generations, we evaluate at every generation if the best individual is within a certain acceptable bound, defined by $\epsilon$. If so, we terminate the function. The danger with this method is that an algorithm may never reach the requirement and run on forever. Thus, it may be smart to combine this method with a cap of N-generations, so that it does not run infinitely. My other termination

implementation was this fitness-based termination combined with a large N-generations cap to make sure it stops eventually.

Finally, an algorithm can use an improvement-based termination. With this type of termination, an algorithm will stop when it stops noticing improvement. One must be very careful when setting parameters for this termination, because there may be gaps where no improvement is noticed, but if it runs too long, resources may be wasted while no further improvement will be seen.

# 2 Experiments and Results

## 2.1 Experiment 1

### 2.1.1 Description

Experiment 1 was conducted to be a broad overview of the performances of all the possible genetic algorithms (GAs) compared to each other across all the test functions. With 5 parameters to vary: initialization, selection, crossover, mutation, and termination, along with 3 possible test functions, this means running 32 GAs across 3 test functions, a total of 96 tests.

Across all tests, 200 samples were run, with results being averaged. For the GAs using n-generation based termination, I have recorded the mean and standard deviation (in parentheses) of the fitness value of the best-performing individual at the 500th generation. For the GAs using fitness-based termination, I have recorded the mean and standard deviation (in parentheses) of the number of generations that the function ran, given stop bounds that were varied for each function.

In order to save space, I have created a shorthand for describing the genetic algorithms. GAs will be written along with a set of 5 ordered values, matching the 5 variable parameters, each either 1 or 0, representing one of two possibilities for that parameter (ex. GA13 (0,0,1,1,0)). These parameter listings are as follows:

Initialization: 0 - random initialization, 1 - grid initialization

Selection: 0 - roulette wheel selection, 1 - tournament selection (n=5)

Crossover: 0 - 1-point crossover, 1 - uniform crossover

Mutation: 0 - uniform mutation, 1 - Gaussian mutation

Termination: 0 - n-generations termination, 1 - fitness-based termination

In addition to the variations in GA structure, specific parameters were used within the GA implementation. For tournament selection, a parameter of $n = 5$ was used for running each tournament. A global mutation rate of 0.2 was used for both mutation algorithms. For N-generation termination, an N value of 500 generations was used. For fitness-based termination, values for the stop bound of 0.0001, 0.004, and 0.003 were used for the Himmelblau, Ackley, and Beale functions respectively.

### 2.1.2 Results

The following tables show the performance of all the GAs on all the test functions. The first table will show the mean and standard deviation of 200 trials of the fitness of the best-performing individual at the end of 500 generations across the 16 n-generations termination GAs. The second table will show the mean and standard deviation of 200 trials of the number of generations run for the 16 fitness-based termination algorithms.

The second pair of tables has derived results, which break down the totals of each GA by parameter ran. Because the same number of trials was run for each parameter in combination with all other parameters, we can compare these values to get an idea of performance depending on parameters chosen.

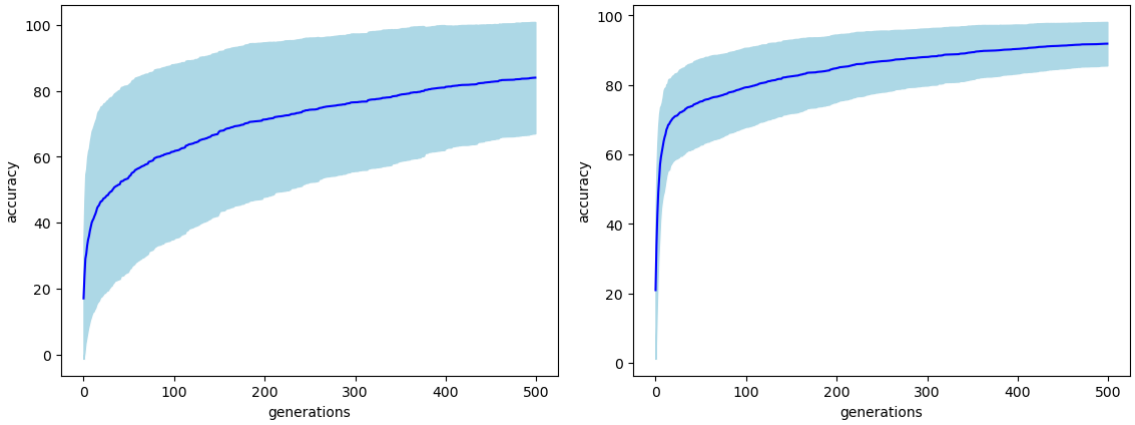
| Experiment 1 | | | |
|---|---|---|---|
| Fitness of Best Individual After 500 Generations | Himmelblau | Ackley | Beale |
| Average Best of 200 Trials | mean (std) | mean (std) | mean (std) |
| GA1 (0,0,0,0,0) | 99.259 (1.166) | 80.535 (10.655) | 85.914 (14.96) |
| GA2 (1,0,0,0,0) | 99.302 (0.887) | 76.677 (11.623) | 90.937 (7.357) |
| GA3 (0,1,0,0,0) | 99.381 (1.036) | 79.017 (11.144) | 87.466 (11.613) |
| GA4 (1,1,0,0,0) | 99.37 (0.913) | 78.179 (11.62) | 90.694 (6.487) |
| GA5 (0,0,1,0,0) | 98.997 (1.615) | 76.965 (11.535) | 84.542 (16.933) |
| GA6 (1,0,1,0,0) | 99.309 (1.072) | 75.843 (12.2) | 88.938 (7.714) |
| GA7 (0,1,1,0,0) | 99.35 (0.985) | 78.656 (12.174) | 84.242 (15.376) |
| GA8 (1,1,1,0,0) | 99.438 (0.832) | 77.459 (11.091) | 88.837 (6.928) |
| GA9 (0,0,0,1,0) | 99.952 (0.066) | 93.386 (4.095) | 99.482 (0.95) |
| GA10 (1,0,0,1,0) | 99.956 (0.071) | 93.071 (4.156) | 99.622 (0.761) |
| GA11 (0,1,0,1,0) | 99.965 (0.054) | 94.319 (3.412) | 98.465 (1.551) |
| GA12 (1,1,0,1,0) | 99.969 (0.05) | 94.332 (3.71) | 98.443 (1.495) |
| GA13 (0,0,1,1,0) | 99.943 (0.105) | 93.031 (4.439) | 99.555 (0.914) |
| GA14 (1,0,1,1,0) | 99.955 (0.076) | 93.278 (4.548) | 99.63 (0.612) |
| GA15 (0,1,1,1,0) | 99.96 (0.065) | 94.042 (3.964) | 98.698 (1.432) |
| GA16 (1,1,1,1,0) | 99.97 (0.038) | 93.754 (4.095) | 98.391 (1.671) |
| Generations ran to reach stop bound $\epsilon$ | $\epsilon = 0.0001$ | $\epsilon = 0.004$ | $\epsilon = 0.003$ |
| GA17 (0,0,0,0,1) | 362.795 (247.064) | 375.7 (278.493) | 312.12 (314.31) |
| GA18 (1,0,0,0,1) | 302.575 (191.603) | 360.16 (238.556) | 139.59 (136.356) |
| GA19 (0,1,0,0,1) | 312.815 (220.899) | 348.83 (250.784) | 254.76 (314.742) |
| GA20 (1,1,0,0,1) | 296.615 (220.899) | 359.24 (236.624) | 137.04 (127.178) |
| GA21 (0,0,1,0,1) | 345.245 (251.858) | 438.98 (279.291) | 342.65 (368.692) |
| GA22 (1,0,1,0,1) | 363.25 (238.208) | 428.76 (273.016) | 182.495 (166.597) |
| GA23 (0,1,1,0,1) | 322.89 (202.024) | 370.77 (246.862) | 313.935 (357.278) |
| GA24 (1,1,1,0,1) | 338.045 (228.358) | 394.63 (264.526) | 183.665 (166.917) |
| GA25 (0,0,0,1,1) | 87.195 (62.314) | 95.265 (66.157) | 37.28 (48.527) |
| GA26 (1,0,0,1,1) | 83.9 (53.528) | 98.44 (64.511) | 17.42 (20.029) |
| GA27 (0,1,0,1,1) | 77.945 (52.258) | 89.99 (56.914) | 36.395 (46.64) |
| GA28 (1,1,0,1,1) | 72.6 (50.651) | 82.71 (54.687) | 24.945 (26.061) |
| GA29 (0,0,1,1,1) | 111.395 (72.625) | 116.54 (67.886) | 36.195 (45.5) |
| GA30 (1,0,1,1,1) | 104.66 (65.694) | 121.945 (67.453) | 27.81 (27.016) |
| GA31 (0,1,1,1,1) | 87.645 (58.091) | 103.665 (62.247) | 46.525 (79.062) |
| GA32 (1,1,1,1,1) | 85.06 (61.456) | 91.765 (65.577) | 28.895 (33.256) |

| Experiment 1 - Derived Results | | | | |
|---|---|---|---|---|
| Fitness Totals | By Type | Himmelblau | Ackley | Beale |
| Initialization | Random | 796.807 | 689.951 | 738.364 |
| | Grid | 797.269 | 682.593 | 755.492 |
| Selection | RWS | 796.673 | 682.786 | 748.62 |
| | Tournament | 797.403 | 689.758 | 745.236 |
| Crossover | 1-point | 797.154 | 689.516 | 751.023 |
| | Uniform | 796.922 | 683.028 | 742.833 |
| Mutation | Uniform | 794.406 | 623.331 | 701.57 |
| | Gaussian | 799.67 | 749.213 | 792.286 |

| Experiment 1 - Derived Results | | | | |
|---|---|---|---|---|
| Generations Ran | By Type | Himmelblau | Ackley | Beale |
| Initialization | Random | 1707.925 | 1939.74 | 1379.86 |
| | Grid | 1646.705 | 1937.65 | 741.86 |
| Selection | RWS | 1761.015 | 2035.79 | 1095.56 |
| | Tournament | 1593.615 | 1841.6 | 1026.16 |
| Crossover | 1-point | 1596.44 | 1810.335 | 959.55 |
| | Uniform | 1758.19 | 2067.055 | 1162.17 |
| Mutation | Uniform | 2644.23 | 3077.07 | 1866.255 |
| | Gaussian | 710.4 | 800.32 | 255.465 |

### 2.1.3 Findings

**Initialization** Initialization was run following a random uniform distribution method and a grid method. There was no substantial difference in results between these two methods. The difference is largely insignificant when comparing the fitness performance, however grid initialization did perform better across all three functions with generations ran. This was extremely noticeable with the Beale function specifically, where grid initialization reached the stop bound almost twice as fast as random initialization (1379.86 vs 741.86 generations run on average).




(a) GA 1 - Using Random Initialization

(b) GA 2 - Using Grid Initialization

Figure 5: Comparing two functions with different initialization types

Figure 5 shows a comparison between the performance of random and grid initialization over 200 runs on the Beale function. The mean of these runs is shown with a blue line, and a shaded area showing standard deviation. We can see that across many runs, grid initialization produces fit results much faster early on, however random initialization largely catches up by the end of 500 generations.

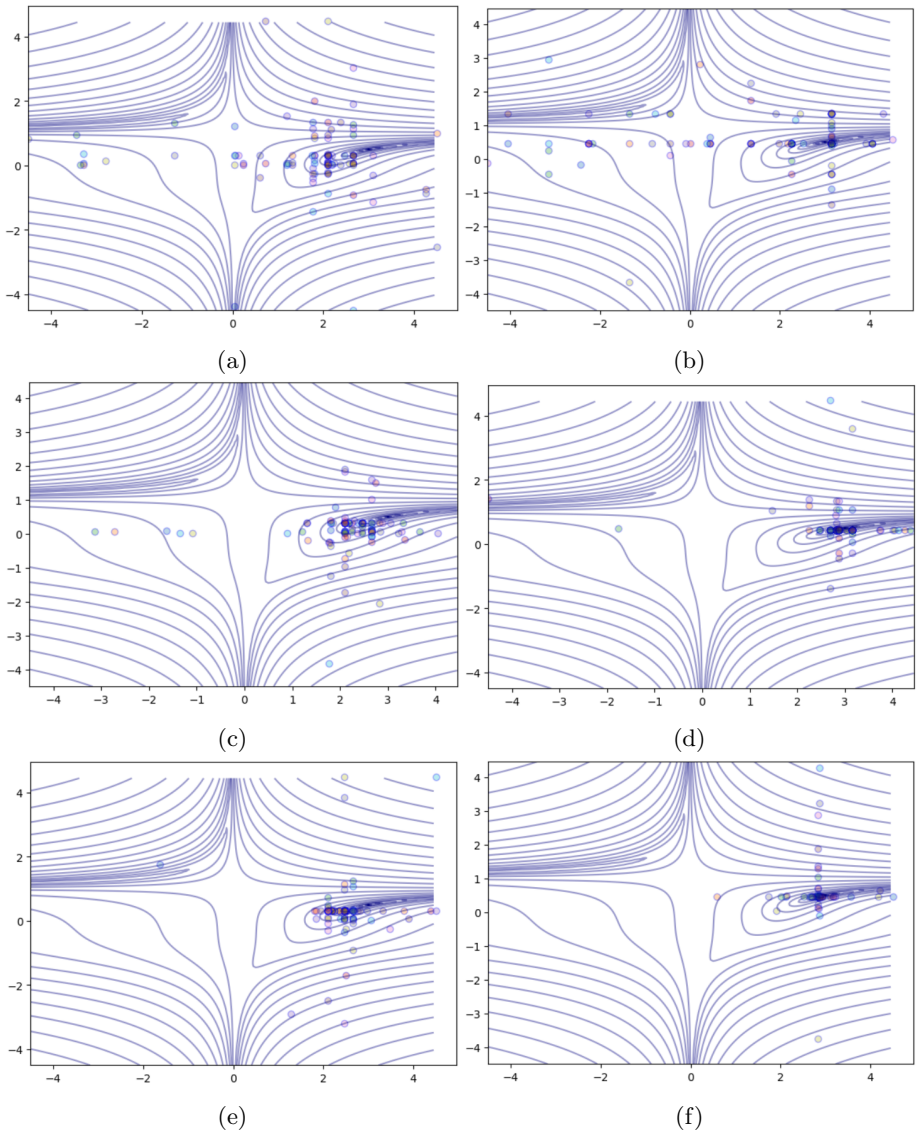



Figure 6: Uniform Initialization (a, c, e) vs. Grid Initialization (b, d, f) Over Time

Figure 6 allows us to watch individuals converge on the global minimum of the Beale function over three time steps. We can see how individuals can much more quickly gravitate to the minimum point when using grid initialization. After looking at these images, it appears that the Beale function experiences such a quick high fitness from grid initialization is because 10 points are placed in the horizontal line $y = 0.5$, which also holds the global minimum of $(3, 0.5)$. As a result of this bias, it is hard to say if grid initialization does perform better than uniform initialization or not.

**Selection** Selection was run with Roulette Wheel Selection (RWS) and with tournament selection (N=5). No substantial difference was observed with fitness totals across all three functions. However, tournament selection does seem more effective in producing fitter results faster, as all three functions ran for fewer generations when using tournament selection.

**Crossover** Crossover used a 1-point crossover and a uniform crossover. Because there are only 2 different values to vary, the effect of this is having either the x or y coordinate swapped with 1-point crossover, or having 0-2 of the coordinates swapped with uniform crossover. There is little noticeable difference between these two methods, with them producing mostly identical results for fitness totals. However, 1-point crossover tends to produce results within the stop bound of fitness-based termination more quickly, resulting in fewer generations run. We can look at how the individuals behave to see how these two crossover methods differ. Figure 7 shows three time steps of GAs running on the Himmelblau function. They are identical, except for the left set uses 1-point crossover, and the right set uses uniform crossover. As can be seen from these images, 1-point crossover tends to form lines in the directions linearly outward from the best individuals (along the x and y axes) much more quickly and much more strongly than uniform crossover does. This 1-point crossover behavior prioritizes an exploitation strategy because an individual will still share one coordinate with its parents, but the uniform crossover behavior shows an exploration strategy because individuals may not share either coordinate with their parent. This results in more varied behavior. On the functions I used, the uniform crossover behavior was not rewarded as well for its exploration strategy, but this may prove helpful for certain functions.

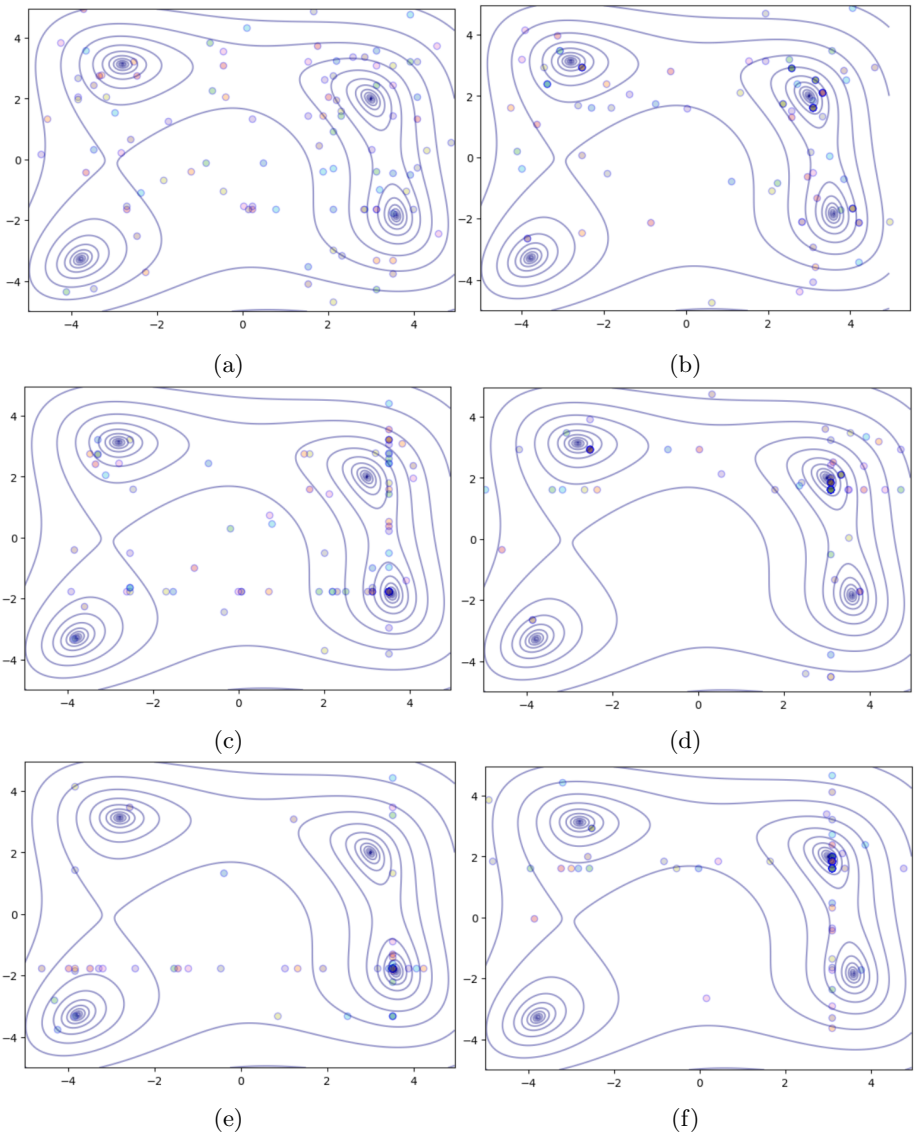
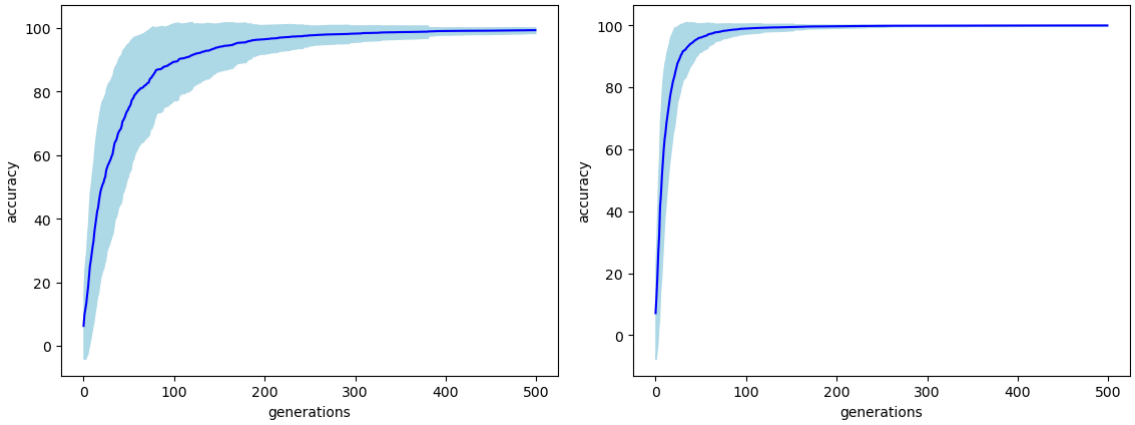



Figure 7: 1-point Crossover (a, c, e) vs. Uniform Crossover (b, d, f) Over Time

**Mutation** Mutation was done using a uniform method and a Gaussian method. It was by far the most differentiating parameter between all the GAs. The Gaussian mutation method showed substantially better results across every test performed. This makes sense: the uniform mutation method is completely uninformed, whereas the Gaussian mutation method uses existing data and attempts to use both exploration and exploitation strategies to achieve better performance.




(a) GA 1 - Using Uniform Mutation

(b) GA 9 - Using Gaussian Mutation

Figure 8: Comparing two functions with different mutation types

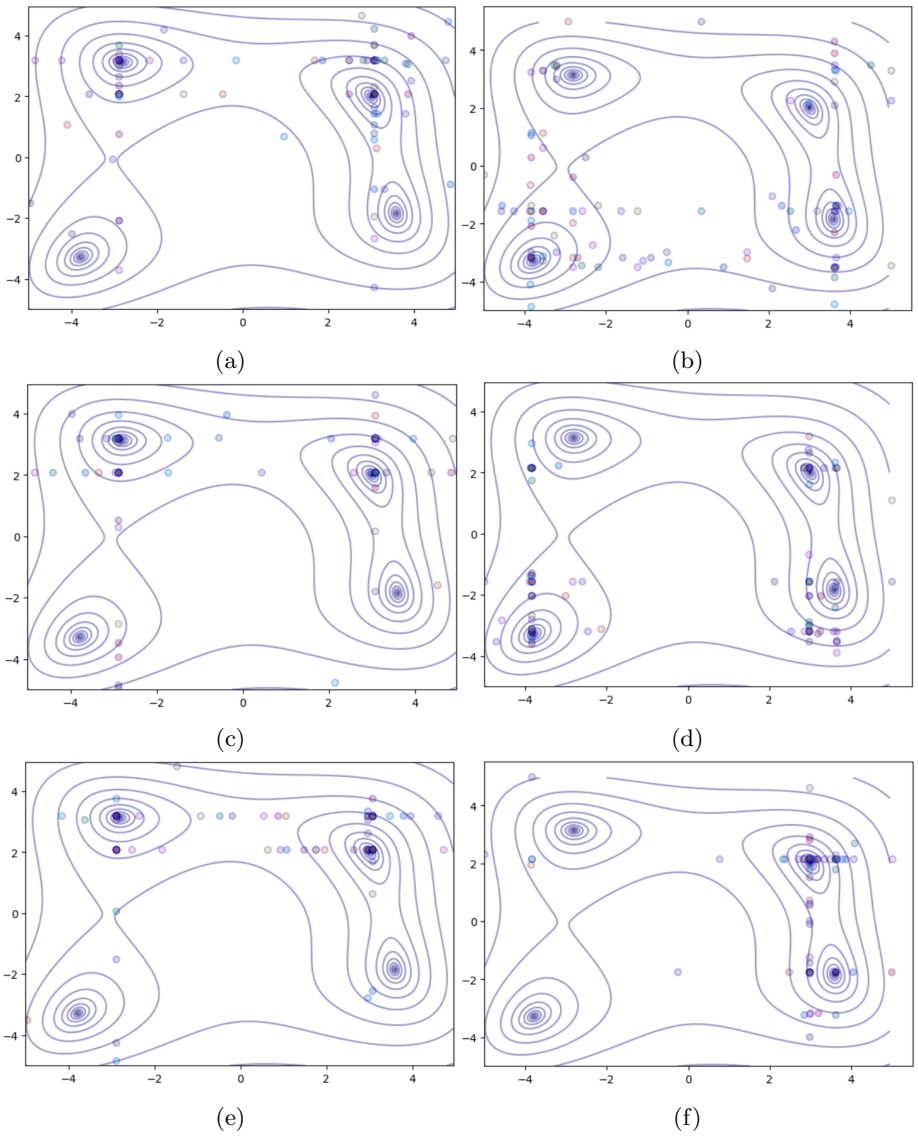



Figure 9: Uniform Mutation (a, c, e) vs. Gaussian Mutation (b, d, f) Over Time

Figure 8 shows the difference between uniform and Gaussian mutation for the best-fit individuals. Not only does Gaussian mutation generate better results, it does so faster, and more consistently.

We can investigate the behavior of Gaussian mutation further by graphing it over time. Figure 9 shows the performance of uniform mutation compared to Gaussian mutation. We can see the difference in behavior between the two mutation types. Uniform mutation will continue to mutate 20% of individuals randomly around the sample space. Most of these will continue to stay along one of the axes because it is much more likely that only one axis will get mutated than both. This is also connected to the crossover behavior, as mutation and crossover are very connected processes.

The Gaussian mutation results, in contrast, will mutate mostly closer to the best individuals, but some nodes will be located farther out, much more sporadically, in an attempt to explore the sample space.

**Summary** Overall, most of the varied parameters remained fairly consistent in their behavior. Across initialization, selection, and crossover, the final fitness of best individuals was nearly identical across these parameters. However, grid initialization, tournament selection, and 1-point crossover all produced fit results faster across all three functions can could be said to have worked better.

Only in varying the mutation algorithm were increases in performance seen across all data points collected. This is in large because one mutation method was completely random, whereas one was an informed mutation method that relied on previous generations.

## 2.2 Experiment 2

### 2.2.1 Description

We can dig further into the performance of all of these genetic algorithms by looking at how they perform across the set of individuals, instead of looking just at the performance of the best individual. I have run the same tests with the same parameters as Experiment 1, except more data was collected, and GAs were tested only on the Beale Function. This is to say, each of the 32 GA algorithms were run 200 times, with $n = 5$ being used for tournament selection, a global mutation rate of 0.2, an N value of 500 generations, and a fitness-based termination stop bound of 0.003 (running for Beale Function only).

For each GA, we take the mean and standard deviation of the number of generations run, the minimum performing individual, the average individual, and the best performing individual. For generation-based termination, best performing individual data is what was collected for Experiment 1, whereas for fitness-based termination the generations data is analogous to Experiment 1 data collection.

### 2.2.2 Results

| Experiment 2 - Generation-based Termination ($N = 500$) | | | | |
|---|---|---|---|---|
| Beale Function Fitness | Generations | Worse Individual | Average Individual | Best Individual |
| Genetic Algorithm | mean (std) | mean (std) | mean (std) | mean (std) |
| GA1 (0,0,0,0,0) | 500 (0) | $1.848\times10^{-5}$ ($1.385\times10^{-5}$) | 52.510 (11.781) | 86.592 (13.596) |
| GA2 (1,0,0,0,0) | 500 (0) | $2.038\times10^{-5}$ ($2.759\times10^{-5}$) | 55.868 (7.965) | 92.638 (6.785) |
| GA3 (0,1,0,0,0) | 500 (0) | $1.789\times10^{-5}$ ($9.280\times10^{-6}$) | 55.015 (10.359) | 86.576 (14.156) |
| GA4 (1,1,0,0,0) | 500 (0) | $2.073\times10^{-5}$ ($1.916\times10^{-5}$) | 58.337 (6.025) | 89.758 (7.529) |
| GA5 (0,0,1,0,0) | 500 (0) | $1.806\times10^{-5}$ ($8.444\times10^{-6}$) | 54.584 (11.085) | 85.051 (15.855) |
| GA6 (1,0,1,0,0) | 500 (0) | $1.884\times10^{-5}$ ($9.629\times10^{-6}$) | 57.897 (6.394) | 90.268 (7.249) |
| GA7 (0,1,1,0,0) | 500 (0) | $1.813\times10^{-5}$ ($8.572\times10^{-6}$) | 54.096 (11.376) | 84.167 (16.367) |
| GA8 (1,1,1,0,0) | 500 (0) | $1.917\times10^{-5}$ ($1.002\times10^{-5}$) | 58.038 (6.129) | 89.846 (7.322) |
| GA9 (0,0,0,1,0) | 500 (0) | 0.000753 (0.00438) | 58.188 (11.114) | 98.744 (7.129) |
| GA10 (1,0,0,1,0) | 500 (0) | 0.000326 (0.00115) | 61.111(7.427) | 99.585 (0.914) |
| GA11 (0,1,0,1,0) | 500 (0) | 0.000251 (0.000772) | 65.785 (4.986) | 98.800 (1.099) |
| GA12 (1,1,0,1,0) | 500 (0) | 0.000360 (0.00138) | 65.613 (4.793) | 98.535 (1.460) |
| GA13 (0,0,1,1,0) | 500 (0) | 0.000430 (0.00182) | 63.467 (7.842) | 98.655 (9.807) |
| GA14 (1,0,1,1,0) | 500 (0) | 0.000451 (0.00225) | 64.154 (4.974) | 99.361 (1.065) |
| GA15 (0,1,1,1,0) | 500 (0) | 0.000264 (0.00128) | 64.507 (7.630) | 97.656 (9.810) |
| GA16 (1,1,1,1,0) | 500 (0) | 0.000325 (0.00107) | 65.360 (4.260) | 9.588 (1.361) |

| Experiment 2 - Fitness-based Termination ($\epsilon = 0.003$) | | | | |
|---|---|---|---|---|
| Beale Function Fitness | Generations | Worse Individual | Average Individual | Best Individual |
| Genetic Algorithm | mean (std) | mean (std) | mean (std) | mean (std) |
| GA17 (0,0,0,0,1) | 325.66 (420.767) | $1.871\times10^{-5}$ ($1.082\times10^{-5}$) | 35.702 (15.147) | 82.883 (6.397) |
| GA18 (1,0,0,0,1) | 140.005 (226.858) | $1.916\times10^{-5}$ ($8.437\times10^{-6}$) | 38.622 (12.329) | 82.330 (5.653) |
| GA19 (0,1,0,0,1) | 243.265 (276.017) | $1.977\times10^{-5}$ ($1.169\times10^{-5}$) | 35.478 (18.157) | 83.498 (7.028) |
| GA20 (1,1,0,0,1) | 169.455 (174.509) | $2.078\times10^{-5}$ ($1.323\times10^{-5}$) | 43.104 (11.993) | 81.286 (5.696) |
| GA21 (0,0,1,0,1) | 350.445 (366.528) | $2.079\times10^{-5}$ ($4.056\times10^{-5}$) | 39.806 (11.783) | 82.865 (5.971) |
| GA22 (1,0,1,0,1) | 165.395 (157.465) | $2.111\times10^{-5}$ ($1.282\times10^{-5}$) | 43.388 (8.934) | 81.300 (4.687) |
| GA23 (0,1,1,0,1) | 347.195 (424.324) | $1.996\times10^{-5}$ ($1.235\times10^{-5}$) | 42.354 (12.533) | 81.652 (5.641) |
| GA24 (1,1,1,0,1) | 173.935 (164.634) | $2.231\times10^{-5}$ ($2.895\times10^{-5}$) | 44.329 (9.426) | 81.082 (5.003) |
| GA25 (0,0,0,1,1) | 38.915 (59.902) | 0.000308 (0.00112) | 26.170 (13.658) | 85.956 (6.361) |
| GA26 (1,0,0,1,1) | 22.495 (23.055) | 0.000253 (0.000761) | 34.403 (10.642) | 84.738 (6.427) |
| GA27 (0,1,0,1,1) | 31.64 (38.869) | 0.000419 (0.00142) | 26.971 (18.041) | 84.901 (6.726) |
| GA28 (1,1,0,1,1) | 27.425 (28.596) | 0.000512 (0.00383) | 35.231 (16.692) | 83.827 (6.429) |
| GA29 (0,0,1,1,1) | 39.18 (43.181) | 0.000236 (0.000809) | 33.547 (13.488) | 85.546 (6.230) |
| GA30 (1,0,1,1,1) | 25.315 (26.184) | 0.000471 (0.00227) | 38.736 (10.401) | 84.260 (6.218) |
| GA31 (0,1,1,1,1) | 47.22 (63.425) | 0.000542 (0.00212) | 35.698 (16.418) | 84.721 (6.765) |
| GA32 (1,1,1,1,1) | 28.85 (29.184) | 0.000238 (0.000747) | 40.922 (13.411) | 82.852 (6.022) |

| Experiment 2 - Derived Results | | | | | |
|---|---|---|---|---|---|
| Fitness Totals | By Type | Generations | Minimum | Average | Maximum |
| Initialization | Random | 4000 | 0.00177056 | 468.152 | 736.241 |
| | Grid | 4000 | 0.00154112 | 486.378 | 758.579 |
| Selection | RWS | 4000 | 0.00203576 | 467.779 | 750.894 |
| | Tournament | 4000 | 0.00127592 | 486.751 | 743.926 |
| Crossover | 1-point | 4000 | 0.00176748 | 472.427 | 751.228 |
| | Uniform | 4000 | 0.0015442 | 482.103 | 743.592 |
| Mutation | Uniform | 4000 | 0.00015168 | 446.345 | 704.896 |
| | Gaussian | 4000 | 0.00316 | 508.185 | 789.924 |

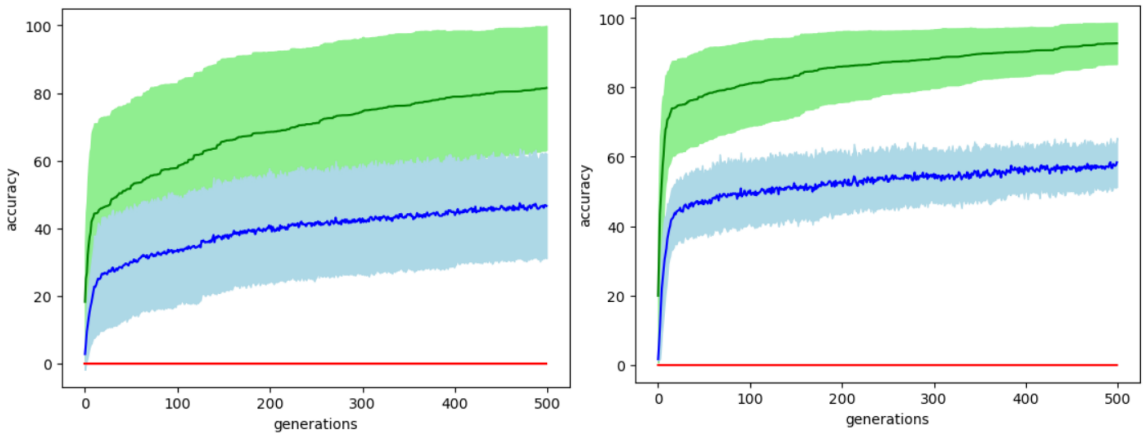
| Experiment 2 - Derived Results | | | | | |
|---|---|---|---|---|---|
| Fitness Totals | By Type | Generations | Minimum | Average | Maximum |
| Initialization | Random | 1423.52 | 0.00158423 | 275.726 | 672.022 |
| | Grid | 752.875 | 0.00155736 | 318.735 | 661.675 |
| Selection | RWS | 1107.41 | 0.00134777 | 290.374 | 669.878 |
| | Tournament | 1068.985 | 0.00179382 | 304.087 | 663.819 |
| Crossover | 1-point | 998.86 | 0.00157042 | 275.681 | 669.419 |
| | Uniform | 1177.535 | 0.00157117 | 318.78 | 664.278 |
| Mutation | Uniform | 1915.355 | 0.00016259 | 322.783 | 656.896 |
| | Gaussian | 261.04 | 0.002979 | 271.678 | 676.801 |

### 2.2.3 Findings

We find many similarities with general trends in Experiment 2. We once again find substantial improvement with Gaussian mutation, now shown across all data found by the experiments. We find varied results for the initialization, selection, and crossover parameters. Now we break down results along each varied parameter in more detail.

**Initialization** Initialization produced varied results and is also very hard to judge because of the bias that grid initialization has with the Beale function specifically. However, generally grid initialization tended to produce better results. We can see random initialization (GA1) compared to grid initialization (GA2) over 50 runs on the Beale function in Figure 10. The green shows the best performing individual, the blue shows the average individual, and the red shows the worst individual, with the shading representing standard deviations. Through visualizing these results, we can see that grid initialization tends to perform much more consistently, which makes sense as the grid is initialized the same way each time, leaving one fewer means of introducing variation.

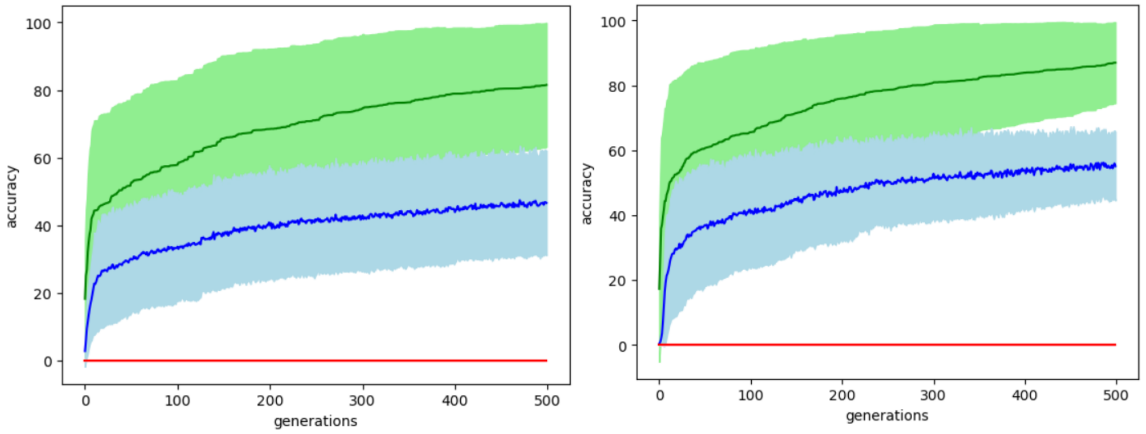
**Selection** Selection produced inconclusive results. Roulette wheel selection produced better maximum and minimum results, but tournament selection produced results quicker and had better average results. A comparison between roulette wheel selection (GA1) vs tournament selection (GA3) is shown in figure 11. We can see that tournament selection has taken a lead on producing fitter results in the first 500 generations, and is also doing so more consistently. This makes sense because although the pool selected from tournament selection varies, we always pick the best individual.




(a) GA 1 - Using Random Initialization (b) GA 2 - Using Grid Initialization

Figure 10: Comparing two functions with different initialization types

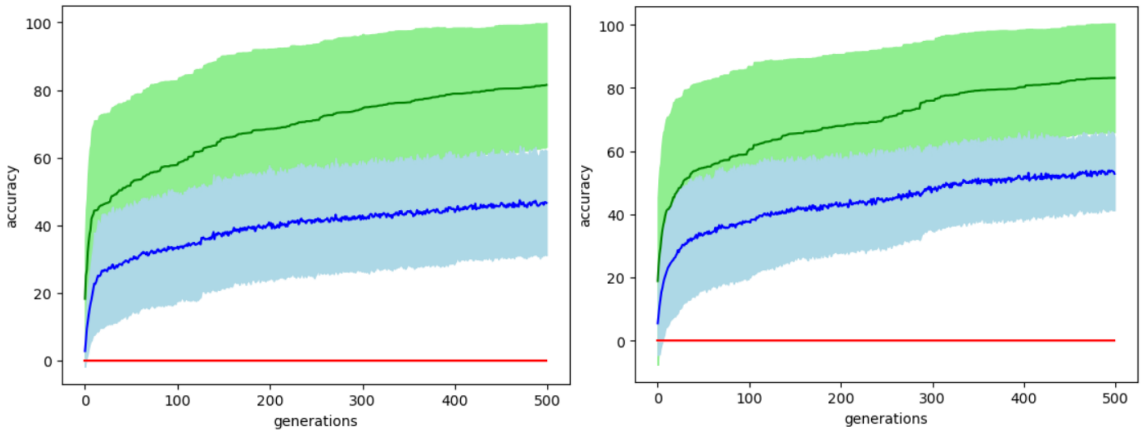
This has less of a randomizing effect than roulette wheel selection, where any individual may be selected (although not with equal probability).




(a) GA 1 - Using Roulette Wheel Selection (b) GA 3 - Using Tournament Selection

Figure 11: Comparing two functions with different selection types

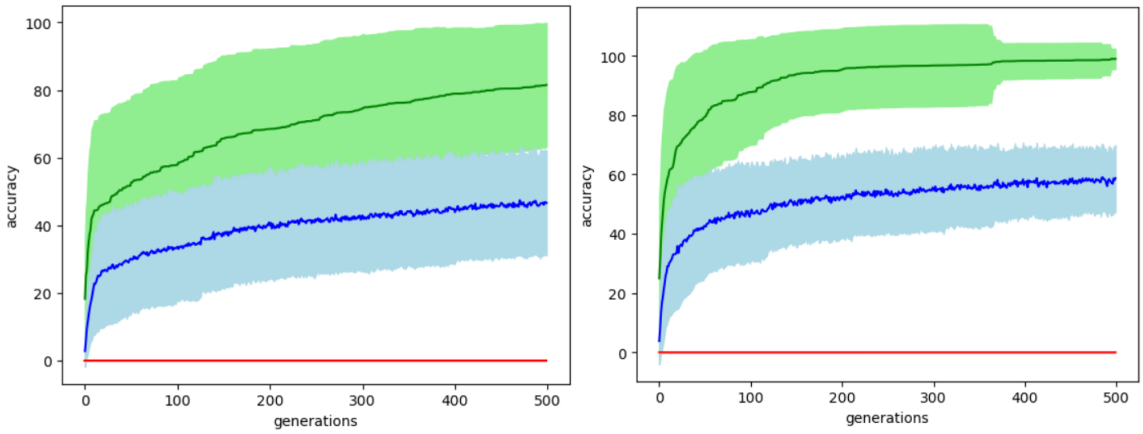
**Crossover** Varying crossover produces very little change in results. Uniform crossover takes longer than 1-point crossover to produce fit results. 1-point crossover tends to produce better maximum results, but has a worse average. We can also visually compare these two crossover algorithms to see their performance compared. This is shown in figure 12. We can see that 1-point crossover tends to produce better results in the first few generations, but that it generally averages out as the functions run.




(a) GA 1 - Using 1-point Crossover (b) GA 5 - Using Uniform Crossover

Figure 12: Comparing two functions with different crossover types

**Mutation** Mutation choice is once again the most significant parameter in genetic algorithm performance, and this can be seen across all data points. Every GA that used Gaussian mutation showed improved results across minimum performance, average performance, maximum performance, and generations ran. Uniform mutation is shown alongside Gaussian mutation in figure 13. We can see that not only does Gaussian mutation have better performance, but it performs more consistently as well.




(a) GA 1 - Using Uniform Mutation (b) GA 5 - Using Gaussian Mutation

Figure 13: Comparing two functions with different mutation types

**Summary** Similar to what was seen in Experiment 1, there were mixed results for different parameter choices for initialization, selection, and crossover. Grid initialization did run much faster

than random initialization, but we must attribute some of this to the bias described earlier for the Beale function specifically. 1-point crossover and tournament selection again produced fit results faster, but 1-point crossover had worse average populations.

Mutation once again produced dramatically different results, visible across every data point. Mutation had by far the most varied parameters, so it continues to make sense that it created such different data.

## 2.3 Experiment 3

### 2.3.1 Description

The first two experiments focused on the different GAs and the various strategies that can be used for them. In order to focus on this, I set many parameters to generic values without testing to see if they produce the best performance. One such example is the mutation rate, which I set at 0.2 for both mutation methods across all tests in experiments 1 and 2. However, the mutation rate can have a very significant impact on the performance of a GA and is thus worth looking into further.

### 2.3.2 Results

| Experiment 3 | | |
|---|---|---|
| Mutation Rate | Generations Ran Mean | Generations Ran Std |
| 0.05 | 118.465 | 217.661 |
| 0.15 | 28.44 | 30.632 |
| 0.25 | 14.8 | 14.691 |
| 0.35 | 11.295 | 11.899 |
| 0.45 | 8.225 | 7.751 |
| 0.55 | 6.465 | 4.715 |
| 0.65 | 5.795 | 4.401 |
| 0.75 | 6.49 | 4.918 |
| 0.85 | 7.835 | 5.49 |

### 2.3.3 Findings

I have run 200 trials of the best performing GA on the Beale function (GA26 (1,0,0,1,1) but varied the mutation rate. Everything else was kept consistent. The results of this are shown in the experiment 3 table. As can be seen from this data, values around 0.65 tended to perform the best, with performance decreasing to either side. It is important to note that this is very inconsistent, with the standard deviation nearly meeting and sometimes exceeding the mean value. However, there is a consistent trend showing that 0.65 (or somewhere in the gaps not explored) produces the best results.

Additionally, I tried running tests with mutations rates lower and higher than the ones depicted here. Mutation rate values that were particularly close to 0 or to 1 did not produce results in a sufficient amount of time to be able to record. This suggests that exploring more along the boundaries of the mutation rate possibilities is not worthwhile.

### 2.3.4 Conclusion

While this test would be much more conclusive if I had the computational resources to refine it and run many more tests, there is a clear trend shown for a mutation rate that is somewhere in the middle of 0 and 1. This is balancing individuals staying close to their old locations and exploiting found minima, with individuals mutating large distances and exploring the function range. Balancing these two principles is crucial to any genetic algorithm, so it makes sense that varying the mutation rate will show consistent results.

# 3 Summary and Future Work

## 3.1 Summary

This project let me learn a lot about the behavior of genetic algorithms and all the parameters that can be varied in creating them. My findings can be broken down along the genetic algorithm components.

Initialization did not produce particularly varied results. Grid initialization tended to perform better, but this was sometimes due to the grid lining up with the function. It would be an interesting experiment to program a grid where the grid is placed randomly in the function range, so that the lines of the grid could be anywhere in the range. It would also be very interesting to explore initialization techniques that are informed from previous experimentation. While I did not implement any strategies of this sort, it likely produces different results from the initialization tactics that I implemented Overall, initialization seems to be very function-dependent and require careful selection based on problem.

Selection was among the least varied components in my experimentation. This is likely in part because I picked similar and simple selection methods. Many selection methods exist and would likely have produced varied results from what I found. Of all the GA components, I found selection the most difficult to understand its overall effects and to see its visual impact, which makes it difficult for me to understand which selection algorithms may perform better. Additionally, I only used one N value for my tournament selection. I may have found a more substantial different between tournament selection and roulette wheel selection if I had experimented with different N values.

Crossover also produced pretty similar results, even when varying its algorithm. However, it is much easier to visualize than selection. 1-point crossover sometimes produced better results, but nothing was very conclusive. It was hard to experiment much with crossover algorithms because this project was done over only two dimensions, leaving very few options for varying this parameter. Across larger problems with more axes, there are many more means of implementing crossover that can be explored further.

Mutation did produce quite varied results. I was able to write a mutation algorithm myself that used ideas presented in class (using Gaussian distribution for mutation) and extended it to try to improve performance. My Gaussian mutation algorithm attempted to best balance exploration and exploitation to produce fit results as quickly as possible, which all of my experiments seemed to prove successful. However, I can not give my algorithm all the credit for this increase in performance, the mutation algorithm which I compared it against, a uniform mutation algorithm, was uninformed by the data and thus had little chance to ever perform at the same level as the Gaussian mutation algorithm. I was also able to explore how varying the mutation rate affected performance, and also how far the values I used in experiments 1 and 2 were from what I found to be the optimal value. Although, the tests that I did in experiment 3 to try to find the ideal mutation rate were only run on one GA on one test function, thus I have no way of knowing if this mutation rate will also perform the best with other GAs or on other functions.

Termination was not well tested in my experiments because the two termination algorithms that I used, generation-based termination and fitness-based termination, did not produce easily comparable results. Comparing termination algorithms would make a lot more sense in an experiment where computational performance and time were important. Termination algorithms try to balance computational needs with fitness, and as I was not doing any research into computational needs, this did not apply well to my research. However, the two termination algorithms that I used did

provide valuable insights into the performance of all the other genetic algorithm parameters, as I could see results over constant generations as well as how quickly fitness could be increased with a variable number of generations.

Finally, I was only able to look at all these genetic algorithm components in isolation, and never got to dig deep into how they interacted with each other, which is certainly a large aspect of how they behave.

## 3.2 Future Ideas

If given more time, there are many more elements of genetic algorithms that I would be interested in exploring. I will describe some of these ideas for future research.

Exploring the behavior of informed v. uninformed operators would be very interesting. I was only able to try this with mutation: using an informed Gaussian mutation strategy in comparison to a full random uniform mutation strategy. For all other genetic algorithm components, I tested either two uninformed strategies or two informed strategies. Mutation produced by far the most variation in results, suggesting that informed v. uninformed operators are very important to the performance of a genetic algorithm and are worth further exploration.

In addition to trying different functions for genetic algorithm components, there were several parameters that I did not vary during my testing. I ran a consistent tournament selection without changing the N value, I used fitness-based termination without changing the stop bound (other than setting a different one for each function), I used generation-based termination without changing the number of generations, and I only varied mutation rate in one small experiment. These parameters likely have substantial impact on GA performance and are all worth further exploration into the effects they have on GA behavior and performance.

With regard to crossover, I did not ever set a crossover rate, and instead ran each crossover strategy on every individual. It would be an interesting experiment to see how genetic algorithms would behave if there was a crossover rate implemented, and how performance would change as this crossover rate was varied.

Another crucial realm of exploration would be with the interactions between crossover and mutation. They are very similar components in that they both exist to modify individual's chromosomes by various means. The use of differing crossover and mutation strategies, as well as careful selection of the crossover rate and of the mutation rate, has a significant impact over the performance of a genetic algorithm. As my experiments largely focused on the genetic algorithm components individually, I did not get to explore the interaction between crossover and mutation, which is very important for creating genetic algorithms.

My research was substantially impacted by computational ability in what I could run and test for. As an example, running experiment 1 on my laptop took approximately 12 hours, with my laptop plugged in at its top performance mode. Because of this limitation, I was not able to explore the performance of particularly complicated functions, performance over many generations, performance over many dimensions, or performance of varied populations. For some of my experiments, I was not able to run enough tests to get particularly consistent results, which may have impacted some of the conclusions that I made. The use of greater computational resources would have enabled much more exploration into the behavior of genetic algorithms and would open many opportunities for future exploration.

I was also only able to run my genetic algorithms over three fairly simple test functions. I tried more varied test functions but encountered substantial difficulties with computing power as well

as finding ways to compare results with functions that behave substantially differently. Getting to test these genetic algorithms over vastly different functions could produce completely different results and would be an interesting topic for exploration. Although I often approached this paper from the angle that there is a best way to program each component of a genetic algorithm, this has been shown to not be the case (no free lunch theorem), and there are test functions that exist that would refute my results. To draw substantial conclusions about genetic algorithm performance would require exploration of many more test functions, with vastly different behaviors.

I made no effort to vary the population size of my genetic algorithm programming. Much like with termination, population size variation is about balancing computation performance with fitness, and as I made no effort to monitor computation performance, it did not make sense to vary this. However, any serious application of genetic algorithms would require careful research and selection of a population size to find the best results with the least computational effort.

### 3.2.1 Conclusion

Overall, this project gave me a lot of opportunities to directly work with genetic algorithms. Through programming them, I was able to gain better insights into how various genetic algorithm components behave, and what strategies might work best for certain problems. Although there are many more veins in this topic that I would like to explore, I feel that this project has given me a good basic overview of this topic and been very helpful to my overall understanding.

# A Attributions

All graphs are my own except for the first three figures (graphs of each test function) which were taken from the Wikipedia page on Test Functions for Optimization: https://en.wikipedia.org/wiki/Test_functions_for_optimization and the figure demonstrating roulette wheel selection which was found in this paper https://journals.vilniustech.lt/index.php/MMA/article/view/5526.

The basic structure of the code and some of the programmed GA parameters were taken from class Jupyter pages, specifically GABest. All other code is my own work.

All writing is my own work: no LLM or other AI tools were used in the writing of this paper.