Josh Engelsma

Adam Terwilliger

April 26, 2016

CIS 678 – Machine Learning

Project 5

Abstract

With our final project in CIS 678 - Machine Learning, we implement a genetic algorithm (GA) to find the global min of two complex functions: Rosenbrock's Banana function and Goldstein-Price function. Using concepts similar to natural biology, we construct an algorithm that pinpoints with high accuracy the minimum values of various functions. Modeling the inputs to the functions as chromosomes, we select the most fit chromosomes. Using the most fit chromosomes, we breed a new population of children organisms using crossover. We then recursively repeat this strategy of choosing the most fit organisms, and breeding until our population converges to the most fit.

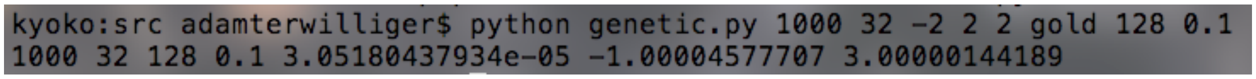
Implementation

Our program is written in Python 2.7 and bash scripting in Unix. These programs were executed locally on each member's respective Macbook Pro (2012), testing on eos23 and okami.

We implemented our program as a modular and highly parameterized Genetic Algorithm class. The genetic algorithm can be modified from the command line to any population size, number of chromosome bits, range values of the function of interest, number of inputs the function will receive, number of generations to evolve for, percent of parents to keep each generation, and any function. Using these inputs, we implement the following methods synonymous with natural evolution: populationEvaluation, populationSelection, populationVariation, populationUpdate, termination, evolveUntilTermination. Each organism in our initial population is a list of bits where the first number of bits represent the first input, the second portion of bits represent the second input etc. In order to encode each real number, input as a bit string, we first shift our real numbers to be all positive. Next, we scale our inputs based on the number of bits we have. Using this process, we can approximately represent the real numbers in our function range. Once we have encoded our initial population, we use the function passed to evaluate our population. Using the roulette wheel methodology, we select probabilistically choose the most fit chromosomes (bit strings). We then use crossover at random locations on all the encoded inputs of the bit string to produce children. Finally, we update our population with a percent of children and a percent of parents - then repeat a new generation. We continue to evolve until the number of generations we specified to evolve for has been reached.

Results

An example of our output can be seen in Figure 1. The parameters to the program include in order: population size, number of chromosome bits, function range min, function range max, number of function inputs, function of choice, number of generations, and percent of parents to keep.



```
kyoko:src adamterwilliger$ python genetic.py 1000 32 -2 2 2 gold 128 0.1
1000 32 128 0.1 3.05180437934e-05 -1.00004577707 3.00000144189
```

Figure 1. Example output of GA.

In Figure 1, we find with 1000 sized population, 32 chromosome bits, 128 number of generations, and 10% of the parents kept around after each generation, our GA finds the global minimum with accuracy to the fifth decimal place. In Figures 2 and 3, we observe the effect population size has on GA convergence, as not until log base 2 of 8 (256) sized population shows convergence. In Figures 4 and 5, we see that GAs do not converge well given under 32 chromosome bits. We find with Figures 6 and 7, that at least 32 generations are needed for convergence for Goldstein and 64-128 generations were needed for Rosenbrock. Inconsequentially, Figure 8 shows us that the percentage of parents kept after each generation from 1% to 99% is negligible with 50 generations and 1000 sized population. In Figure 9, we can see an example of how f(x,y) converges to 3 for the Goldstein function as each generation passes: once generation 24/25 is reached the GA has roughly converged upon 3.00.

Figure 2. Genetic Algorithm -- Rosenbrock's Banana Function
F(x,y) vs. log2(popSize)

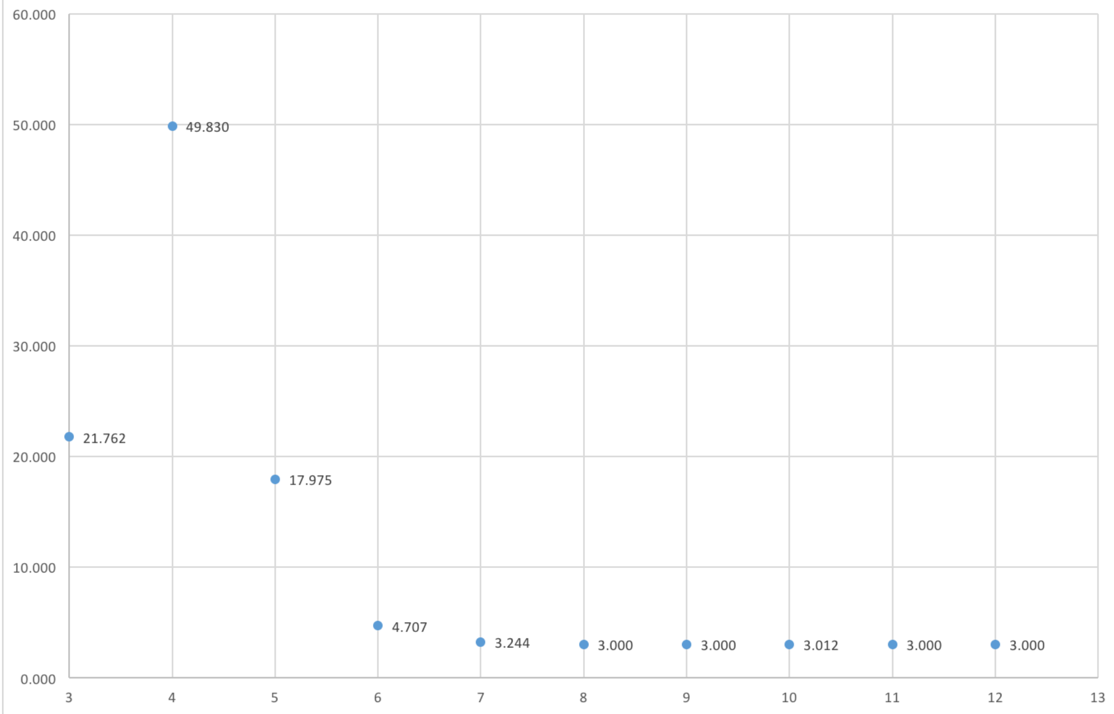
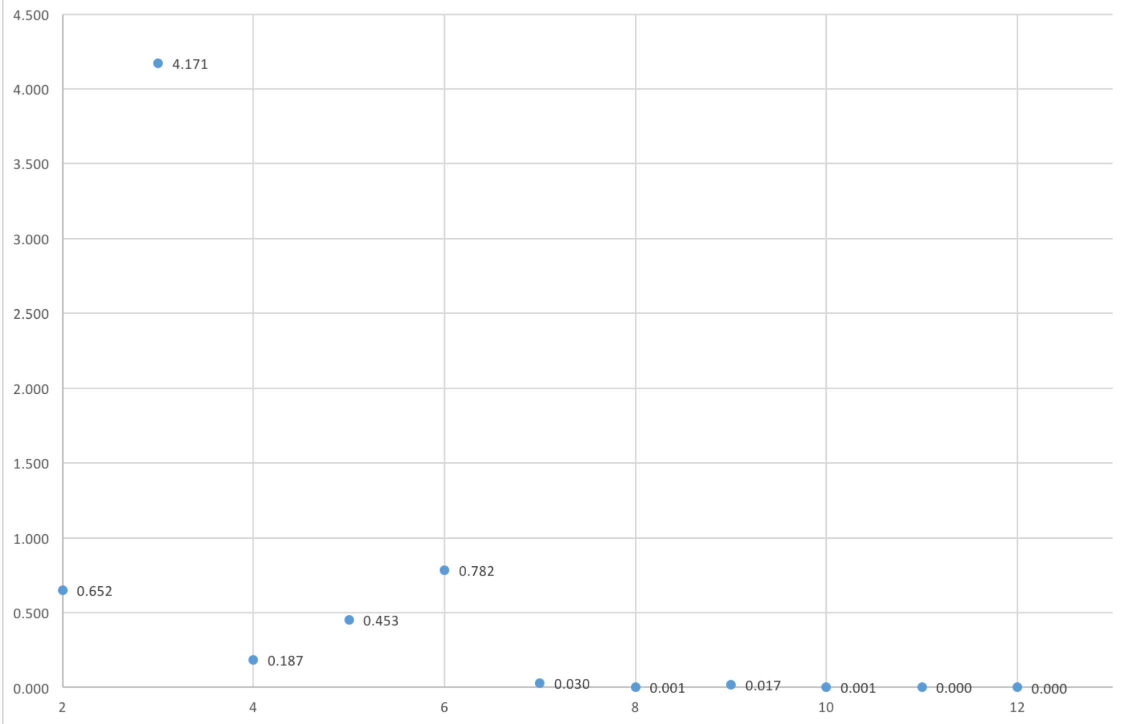


Figure 3. Genetic Algorithm -- Goldstein-Price Function
F(x,y) vs. log2(popSize)



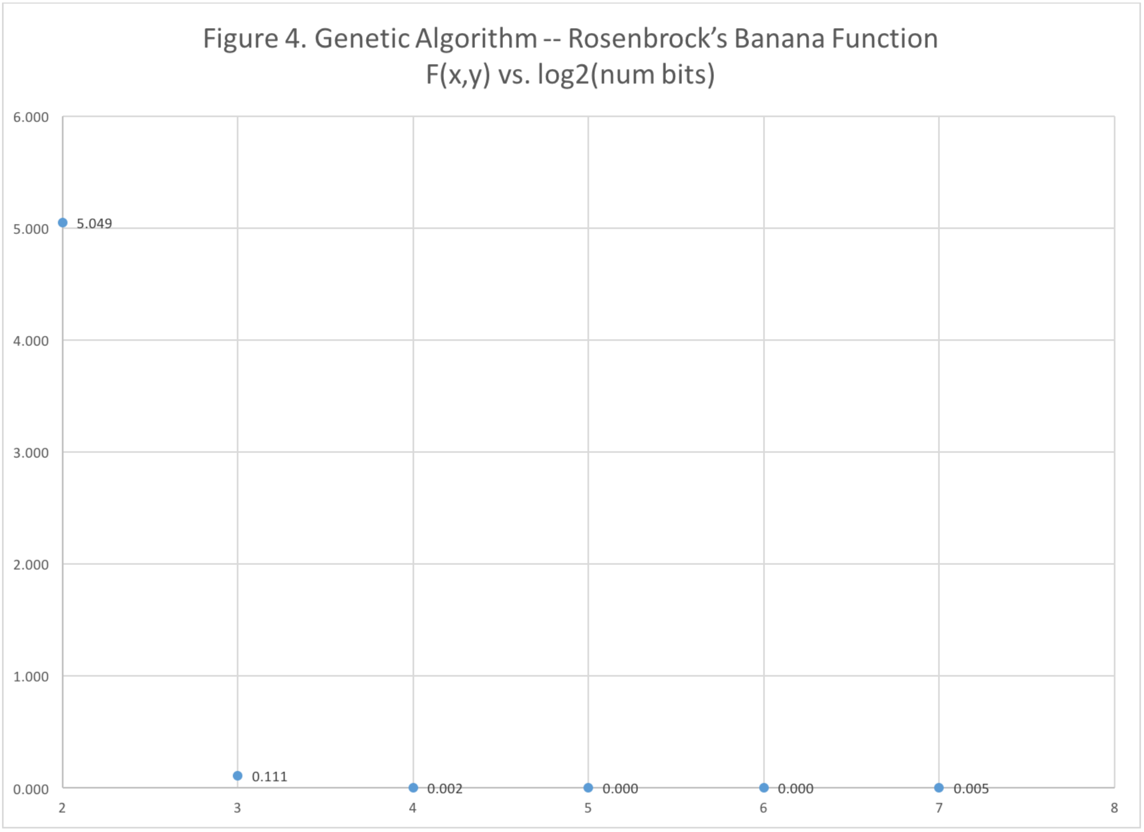


Figure 4. Genetic Algorithm -- Rosenbrock's Banana Function
F(x,y) vs. log2(num bits)
6.000
5.000
4.000
3.000
2.000
1.000
0.000
2
3
4
5
6
7
8
5.049
0.111
0.002
0.000
0.000
0.005

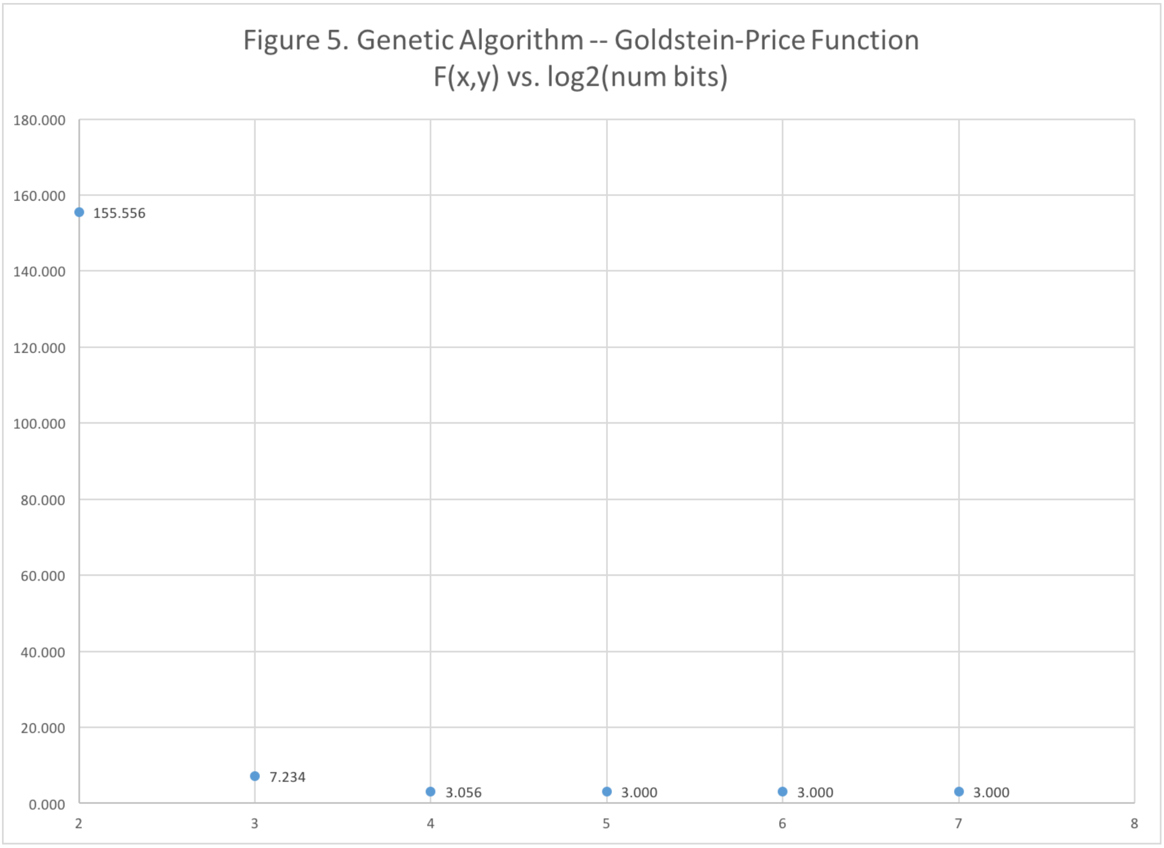


Figure 5. Genetic Algorithm -- Goldstein-Price Function
F(x,y) vs. log2(num bits)
180.000
160.000
140.000
120.000
100.000
80.000
60.000
40.000
20.000
0.000
2
3
4
5
6
7
8
155.556
7.234
3.056
3.000
3.000
3.000

Figure 6. Genetic Algorithm -- Rosenbrock's Banana Function
F(x,y) vs. log2(num generations)

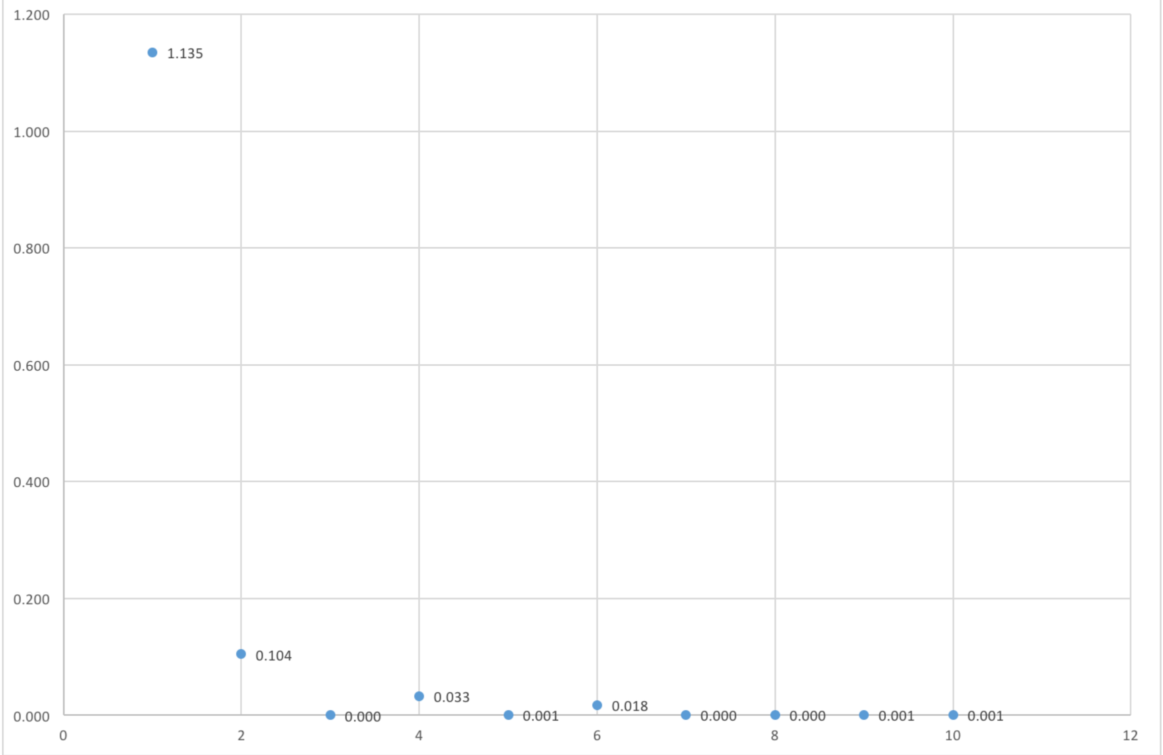


Figure 7. Genetic Algorithm -- Goldstein-Price Function
F(x,y) vs. log2(num generations)

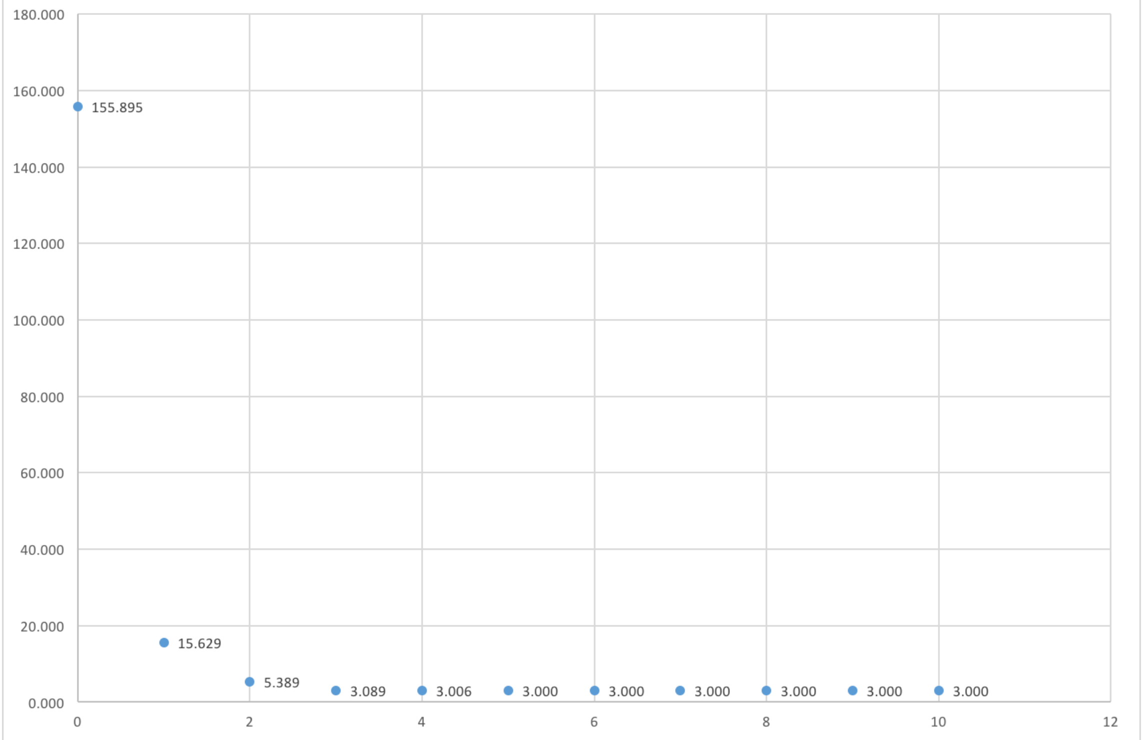


Figure 8. Genetic Algorithm -- F(x,y) vs. % of parents

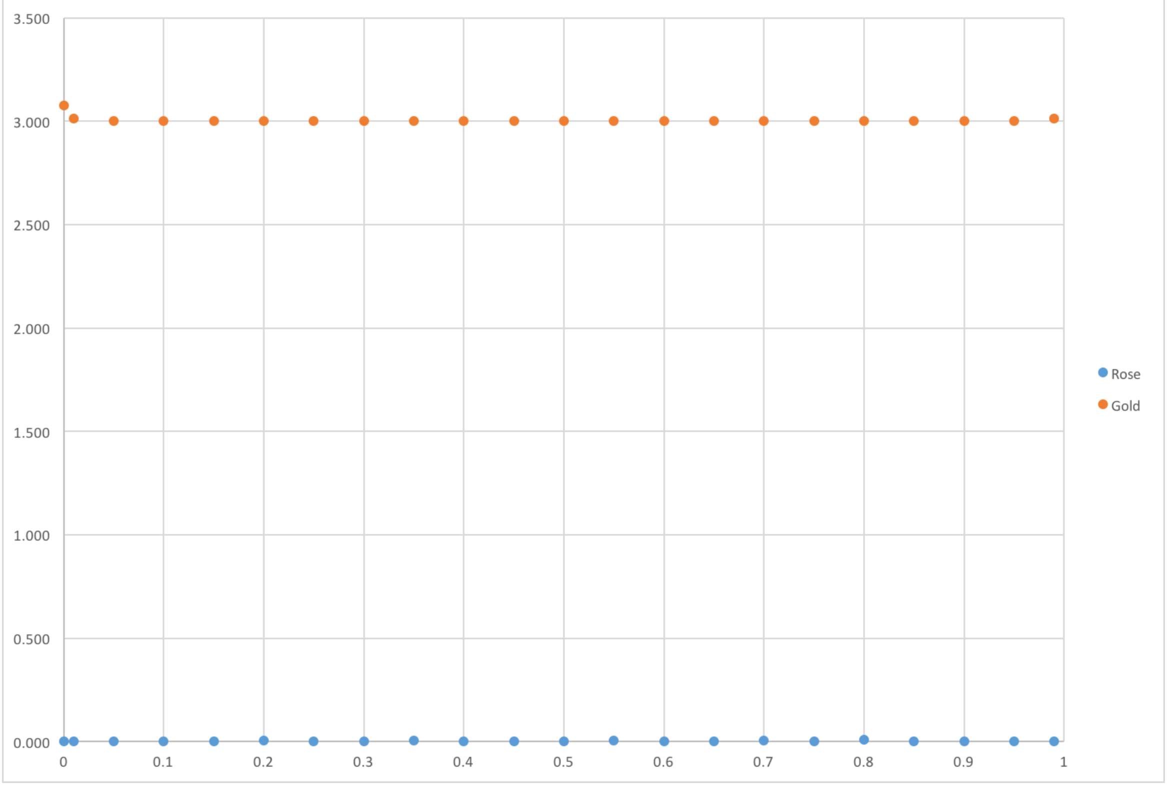
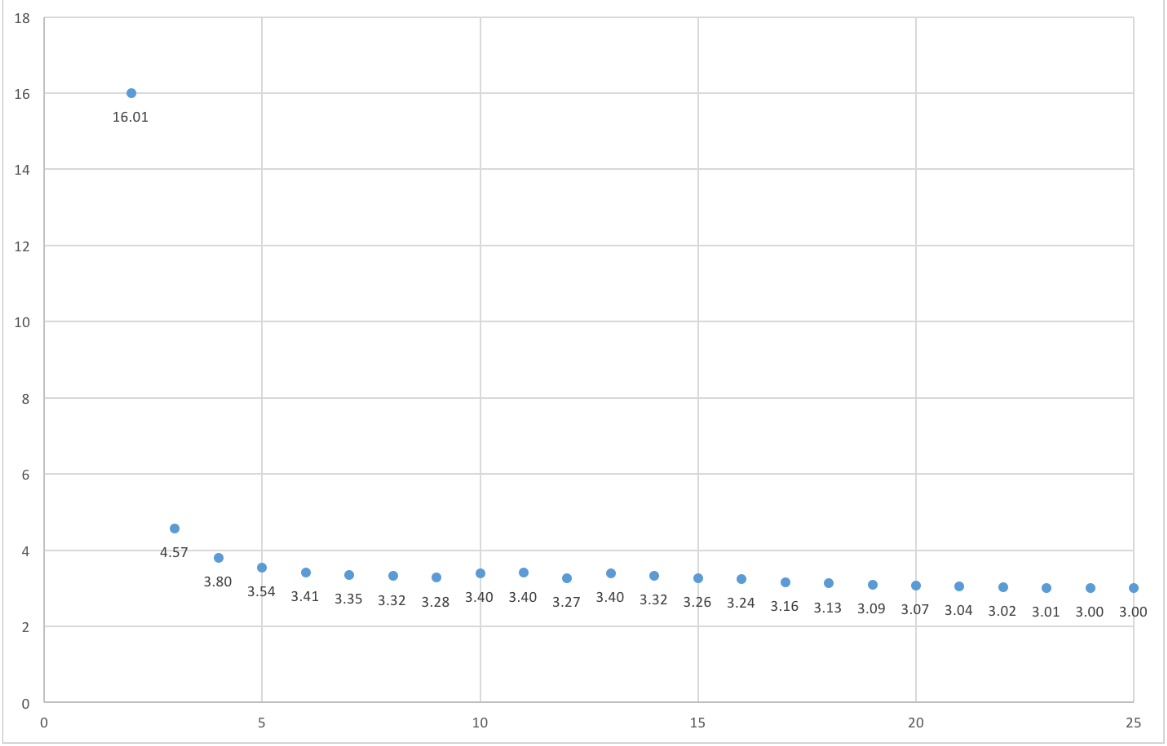


Figure 9. Genetic Algorithm -- Goldstein-Price Function
F(x,y) vs. Generation #


18
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
16.01
4.57
3.80
3.54
3.41
3.35
3.32
3.28
3.40
3.40
3.27
3.40
3.32
3.26
3.24
3.16
3.13
3.09
3.07
3.04
3.02
3.01
3.00
3.00