# A Robust Optimization Method Based on Biological Evolution

Andrew Elias

13 May 2011

## Motivation

One usually cannot find analytic, exact solutions to applied optimization problems, so iterative, numerical methods are typically used. But all currently used numerical optimization methods have weaknesses. Newton's method, for example, is useless without access to a function's derivative (sometimes even requiring access to higher-order derivatives), and for some functions, convergence is rare. Newton-like methods have similar issues. The Bisection method (and similar methods) is slower, and it can only find solutions on a finite domain, which must be specified in advance. Furthermore, multidimensional problems throw a few kinks into the Bisection method: the midpoint of a multidimensional region is not well-defined, and it is hard to have a user specify a multidimensional domain. Most other methods have problems with convergence and require the function to be defined everywhere, and almost all of them require the function to be continuous in both directions. Furthermore, most methods only find a single solution, and the particular solution found is highly dependent on the initial guess.

Coming up with a more robust, effective, and useful optimization method requires a glance at the overarching philosophy of iterative optimization. The main strategy of any optimization method is to take some initial information, repetitively use it to reach new, more useful information, and, over much iteration, approach the desired solution. For example, Newton's method takes as its initial information the value of a function at a point, and the slope (gradient in multivariable functions) at that point. From that information, a new point is found and the process is repeated, producing a sequence of points approaching the solution. For any given initial point, that sequence of points is unique, and if it doesn't converge, Newton's method fails. A question arises: given such limited initial information, how can one leverage it to find more useful information, while moving in the direction (a philosophical direction, not necessarily a geometrical one) of a solution?

## The Solution - Randomness

With such limited initial information, there is only a single logical, algorithmic direction to move. So, in order to break from that path, one needs to use something other than concrete algorithms. Random perturbation is the necessary factor. By finding new seeds near the old ones, a program gains access to more information about the function, and can therefore extrapolate more accurately to find even more seeds.

But perfect randomness will not do; there must be some rule to help push the method in the direction of the intended solution. Randomly walking around the domain looking for the solution would certainly take a long time! Taking a tip from the Theory of Evolution, this 'pushing force' will be a practice of Survival-of-the-Fittest, in other words, a practice of giving preferential treatment to those seeds closer to the objective. Thereby there will be a sort of natural selection amongst the seeds, eventually finding the extremum desired. With a little tweaking, one can hope to find global extrema instead of simply finding local ones.

## The Method and the Program

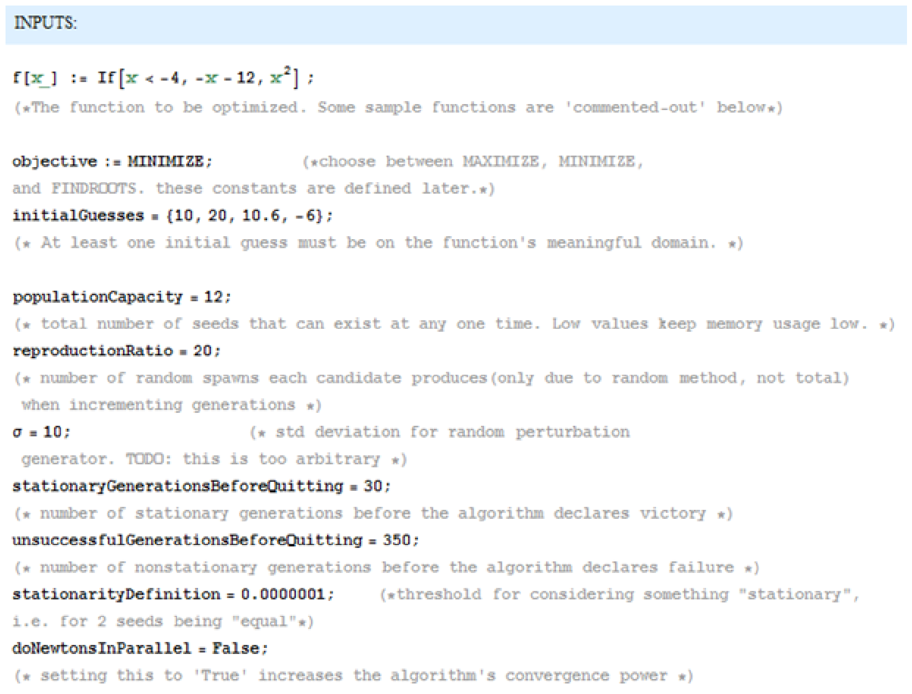
This new 'randomness' method will have the same structure as biological evolution, albeit more structured:

1. **Input.** Specify initial input seeds
2. **Reproduction** Each seed spawns a group of 'child' seeds based on random perturbations (similar to genetic mutations in organisms).
3. All new seeds make up a 'generation' of seeds, and those seeds are ranked according to the objective function.
4. **Survival of the Fittest.** Seeds are eliminated probabilistically, with the best-ranking seeds having the highest probability of surviving.
5. Repeat steps 2–4 until a certain evaluation parameter is met, and return the highest-ranking seed as the output.

The following inputs are necessary: a list of initial seeds, a function to be optimized, an objective (Minimize, Maximize, or Find Roots), a reproduction ratio (how many seeds are spawned by each seed during a generation), a population capacity (the number of surviving seeds to allow, per generation), a parameter for how big the random perturbations/mutations should be, parameters for determining when to stop the process, and options for customizing the algorithm.

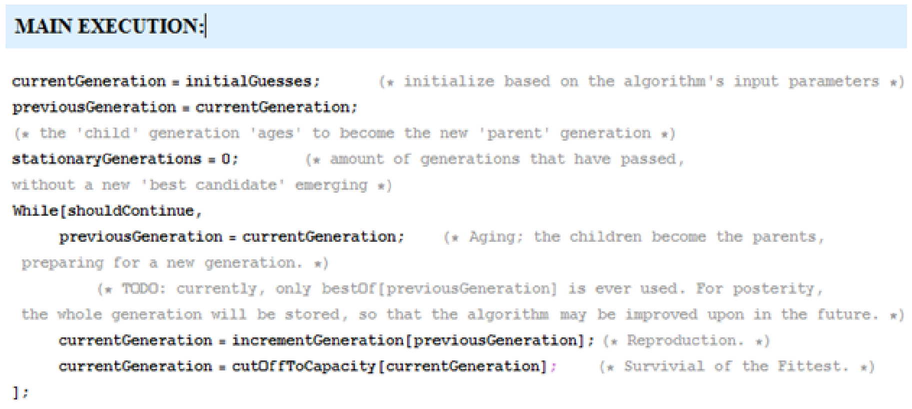
A prototype program/notebook was written in Mathematica to follow this method. The notebook is designed such that filling in the input fields and executing the notebook will return the solution at the bottom of the output.

The aforementioned inputs can be found in the following screenshot from the program:



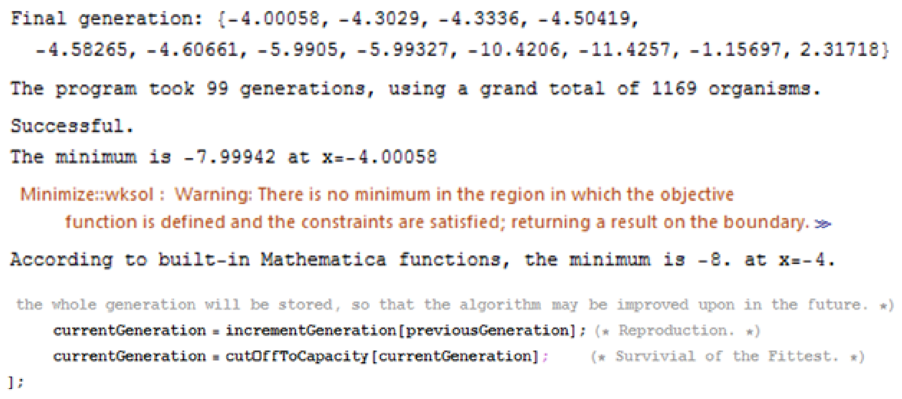
```
INPUTS:

f[x_] := If[x < -4, -x - 12, x^2] ;
(*The function to be optimized. Some sample functions are 'commented-out' below*)

objective := MINIMIZE;          (*choose between MAXIMIZE, MINIMIZE,
and FINDROOTS. these constants are defined later.*)
initialGuesses = {10, 20, 10.6, -6};
(* At least one initial guess must be on the function's meaningful domain. *)

populationCapacity = 12;
(* total number of seeds that can exist at any one time. Low values keep memory usage low. *)
reproductionRatio = 20;
(* number of random spawns each candidate produces(only due to random method, not total)
 when incrementing generations *)
σ = 10;                  (* std deviation for random perturbation
 generator. TODO: this is too arbitrary *)
stationaryGenerationsBeforeQuitting = 30;
(* number of stationary generations before the algorithm declares victory *)
unsuccessfulGenerationsBeforeQuitting = 350;
(* number of nonstationary generations before the algorithm declares failure *)
stationarityDefinition = 0.0000001;    (*threshold for considering something "stationary",
i.e. for 2 seeds being "equal"*)
doNewtonsInParallel = False;
(* setting this to 'True' increases the algorithm's convergence power *)
```

Here is the main execution of the program. In the while loop, one can find the Reproduction and Survival-of-the-Fittest stages. It calls subfunctions (defined earlier) to keep the code somewhat clean:



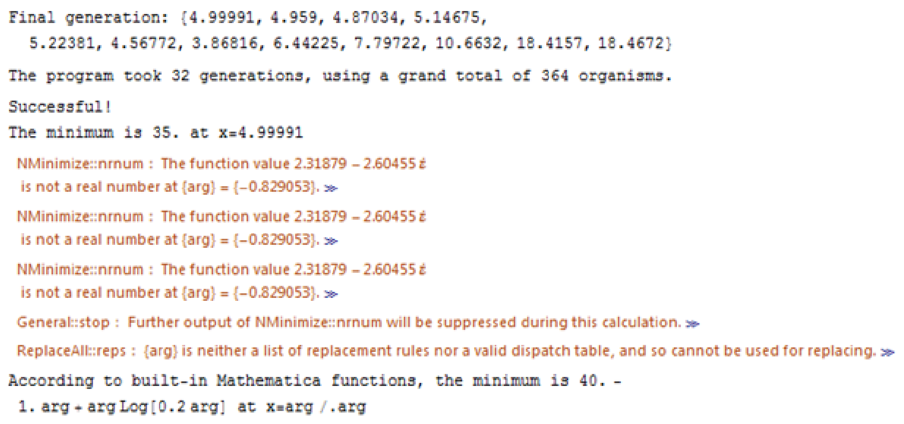
```
MAIN EXECUTION:

currentGeneration = initialGuesses;      (* initialize based on the algorithm's input parameters *)
previousGeneration = currentGeneration;
(* the 'child' generation 'ages' to become the new 'parent' generation *)
stationaryGenerations = 0;       (* amount of generations that have passed,
without a new 'best candidate' emerging *)
While[shouldContinue,
    previousGeneration = currentGeneration;    (* Aging; the children become the parents,
 preparing for a new generation. *)
        (* TODO: currently, only bestOf[previousGeneration] is ever used. For posterity,
 the whole generation will be stored, so that the algorithm may be improved upon in the future. *)
    currentGeneration = incrementGeneration[previousGeneration]; (* Reproduction. *)
    currentGeneration = cutOffToCapacity[currentGeneration];    (* Survivial of the Fittest. *)
];
```

Here is the end of the output of the program. For curiosity's sake, before this section, the program lists all seeds that have existed; this way, one can go back and observe the evolution. Notice that the solution is a numerical approximation, as it always will be:



```
Final generation: {-4.00058, -4.3029, -4.3336, -4.50419,
  -4.58265, -4.60661, -5.9905, -5.99327, -10.4206, -11.4257, -1.15697, 2.31718}
The program took 99 generations, using a grand total of 1169 organisms.
Successful.
The minimum is -7.99942 at x=-4.00058
 Minimize::wksol : Warning: There is no minimum in the region in which the objective
      function is defined and the constraints are satisfied; returning a result on the boundary. ≫
According to built-in Mathematica functions, the minimum is -8. at x=-4.
 the whole generation will be stored, so that the algorithm may be improved upon in the future. *)
    currentGeneration = incrementGeneration[previousGeneration]; (* Reproduction. *)
    currentGeneration = cutOffToCapacity[currentGeneration];    (* Survivial of the Fittest. *)
];
```

Notice that sometimes, this program succeeds where built-in Mathematica functions have trouble, for example, when trying to find the roots of $x\ln(0.2x) - x + 40$:



```
Final generation: {4.99991, 4.959, 4.87034, 5.14675,
  5.22381, 4.56772, 3.86816, 6.44225, 7.79722, 10.6632, 18.4157, 18.4672}
The program took 32 generations, using a grand total of 364 organisms.
Successful!
The minimum is 35. at x=4.99991
 NMinimize::nrnum : The function value 2.31879 - 2.60455 i
  is not a real number at {arg} = {-0.829053}. ≫
 NMinimize::nrnum : The function value 2.31879 - 2.60455 i
  is not a real number at {arg} = {-0.829053}. ≫
 NMinimize::nrnum : The function value 2.31879 - 2.60455 i
  is not a real number at {arg} = {-0.829053}. ≫
 General::stop : Further output of NMinimize::nrnum will be suppressed during this calculation. ≫
 ReplaceAll::reps : {arg} is neither a list of replacement rules nor a valid dispatch table, and so cannot be used for replacing. ≫
According to built-in Mathematica functions, the minimum is 40. -
 1. arg + arg Log[0.2 arg] at x=arg /.arg
```

## Specifics

For a parent seed whose value is x, random mutations in their children were generated according to a normal distribution with $\mu = x$ and $\sigma$ being an arbitrary value input by the user at the start of the program. The use of a normal distribution ensures that, while most 'child' seeds will be near their 'parent' seed, there will, by the Law of Large Numbers, be some deviant seeds. That way, the algorithm can refine its current guesses while simultaneously exploring the domain, looking for new, albeit risky, guesses. The normal distribution also parallels the random mutations in biological evolution, most of which are small, and some of which are large.

For additional convergence power, the user of the program is given an option to do Newton's method in parallel with random mutations. With this Boolean flag set to "True," each parent seed generates a single child seed by way of Newton's method in addition to generating a slew of child seeds via random mutation. The two methods working in parallel allow the program to have both the convergence speed of Newton's method and the robustness of the evolutionary/randomized method.

The Survival-of-the-Fittest stage is done probabilistically, to allow poorly-performing seeds to have a chance to survive. That way, the algorithm has the capacity (but still not the guarantee) to move away from local minima in favor of global minima. First, the sorting function puts seeds in order of how well they suit the objective. Then, seeds of the current generation (here, a list called *out*) are eliminated in a *while* loop, until a certain population capacity (here, *cap*) is met:

```
While[Length[out] > cap,
      (* These lines give positionToDelete a random integer value
   from 1 to Length[out],
  with a probability distribution that is skewed toward Length[out] *)
      positionToDelete = (Length[out] - 1) * (RandomReal[]^0.7);
      positionToDelete = Ceiling[positionToDelete] + 1;

      (* Delete the seed located at positionToDelete *)
      out = Delete[out, positionToDelete];
 ];
```

Raising the randomly generated number (which is on the interval [0,1] by default) to a power slightly less than one (here, *0.7*) produces a skewed probability density function, so that the lower-ranking seeds have a higher probability of being eliminated. From a mathematical perspective, this skewing method is arbitrary and should be replaced by something more rational, but from an engineering perspective, it works well, especially considering that this is a prototype.

## Advantages and Disadvantages

Just like every optimization method preceding it, this random-perturbation method has its advantages and disadvantages. Knowing the strengths and weaknesses of the method is important in deciding whether or not it suits a given problem. Here some of these are discussed.

Speed is a major disadvantage. The program relies heavily on time-costly random number generation, both for random-perturbation generation and for the probabilistic Survival-of-the-Fittest stage. Currently, on a dual-core laptop, execution takes about a minute, depending highly upon the inputs. For a single function, this is not much of an issue, but when running this method for a large quantity of functions, speed is a real concern. To improve speed, the code can be written in a compiled language like C, C++ or FORTRAN, and it can be optimized for efficiency. The current Mathematica incarnation serves merely as a prototype.

Another disadvantage is post-run confidence. Because of the probabilistic nature of the program, it can never be guaranteed that the output solution is actually the optimal solution. Note that post-run confidence is different from pre-run confidence, which is a measure of how confident someone is that the function will not cause convergence issues or other similar issues. This new random-perturbation method is projected to create high levels of pre-run confidence in its users. The best way to improve post-run confidence is to adjust the evaluation parameters. These will tell the program to spend more time looking for the solution, which increases the probability of finding it. However, this also increases the running time of the program, which is already established as being too long.

Interestingly enough, while the probabilistic nature of the program causes the previous two disadvantages, it also leads to most of its biggest advantages. The method does not require thorough understanding of the function to be optimized, and it does not require verification that the function is continuous or differentiable. By this merit, the program can be used to optimize noisy functions, extremely complicated functions, or those that are stochastic in nature. In fact, the method only requires a function that is loosely approximable by continuous functions, and that the function is not double-discontinuous (discontinuous from both sides) at the desired solution point. A single discontinuity at the solution point causes no problems, and neither does a double-discontinuity at any other point.

The method's probabilistic nature also gives it power to find global (not just local) extrema. While it is not guaranteed to jump away from a local extremum, it certainly has the capacity to do so, and changing the evaluation parameters can increase this capacity. Another feature aiding the search for global extrema is the ability to input multiple initial guesses. This gives the user more freedom to customize his search.

Because this method supports multiple seeds existing at every stage of the program, it is ideal for amalgamating other optimization methods. The prototype currently has the ability to do Newton's method in parallel, but it could be extended to allow the Secant method, Method of Steepest Descent, Line Search, and others. And because the method is designed to store so many seeds at each iteration, some more complex methods could be included which utilize multiple seeds. For example, there could be a Secant-method-inspired method which, instead of making a secant *line*, makes a secant *polynomial*, the order of which would be determined by the number of points available. Notably, Bisection method would be a bad method to include in the blend, because its philosophy is so different from that of this method: bisection requires the function to be continuous and two of the input point to straddle the intended solution.

The final advantage of the random-perturbation method is that it is easily extended to do other, more useful things. A later section of this paper will discuss that in detail.

## Issues and Solutions

The only surprising issue encountered while writing the program was that some seeds fell outside of the domain of the function. While the program is designed to search across the set of reals (because computers work numerically, in practice they only use a subset of the rationals), not all functions are defined for all real numbers. And when a seed is spawned outside of the function's domain, the result is usually either a complex number or a complaint from the Mathematica kernel. The solution was to use a subfunction to filter out the non-real seeds and the seeds producing non-real function values. Because complex numbers do not cover all such errors, when moving to a production version of this program, some more work will need to be done in filtering out these error-producing seeds.

A less surprising issue was that, although seeds would come very close to a correct solution, sometimes, over the following few generations, those very seeds would be lost due to "bad" random mutations. For continuous functions, the process would quickly correct itself with "good" random mutations, returning to the correct solution, but at discontinuities, the success of previous generations would sometimes just die off. Solving this problem was simple; each seed was guaranteed that, of all of its children, one would be an exact clone of itself. Therefore, once a correct solution was reached or approached, it would be better protected from 'extinction.' After this augmentation was made, the only way for seeds to die off would be during the Survival-of-the-Fittest stage, which is exactly where bad seeds should die off.

Although not an algorithmic issue, one of the biggest problems encountered was that the programmer was not initially familiar with functional programming practices. The first version of the code attempted to make use of Mathematica's support for procedural and imperative programming paradigms, which resulted in long-winded, buggy code. Some of the code was rewritten in functional style, which was much shorter and more robust. The project has certainly served as a valuable learning experience.

## Extension and Suggestions for Improvement

While the program does a great job at what it is designed to do, there are a few things that can be done to make it more useful and efficient.

First, the program can be extended to functions on a vector (multivariable) domain. The function would take a vector argument, and at each reproduction stage, random mutations would occur along all axes. In this way, the program can approach the solution through the multidimensional space. There may have to be multiple standard deviations for the random perturbations, with one corresponding to each spatial dimension. And if there is an appropriate norm defined for the given vector space, it may be appropriate to apply the normally-distributed random perturbations to the norm and direction instead of the individual components. This will prevent the multidimensional distribution from being similar to a rectangular prism, a shape whose arbitrariness could cause slightly slower convergence.

The next crucial improvement would be to add constraints to the mix. The user would define a list of equations or inequalities (or even constraints taking unusual forms, like "$x < 0$ OR $x > 6$" or "$x \neq 0$"), and for each seed, the program would check to see if the constraints were satisfied. If not, the seed would be eliminated. It is crucial that these seeds be eliminated before the population is reduced to meet the maximum population capacity. Otherwise, the reduced population could, by accident, be composed entirely of infeasible seeds, at which point, the program would be forced to quit and to return no solution. It also may be advantageous to give special consideration to areas where seeds are being eliminated due to constraints. Because most optimal solutions exist near the boundary of a constraint, these areas may be more important than other areas of the domain. Perhaps when a seed dies due to a constraint, its 'parent' or 'sibling' seeds could be given slightly preferential treatment in the Survival-of-the-Fittest ranking stage.

The third big improvement in the program regards the arbitrariness of the user's input value for the standard deviation $\sigma$ of the random perturbations. Currently, the user inputs a single value which is always used. This works when the user knows the scale of the solution, but when there is a wide range of possible answers, choosing a standard deviation becomes difficult. A better system would be to give each seed its own$\sigma$. This allows the user to specify varying degrees of certainty about his initial guesses. When the user knows beforehand that the answer is either exactly 145 or $290 \pm 20$, he can input a list like "$145 \pm 1$, $290 \pm 20$". More importantly, it allows the standard deviation to evolve alongside the seed values themselves. At each reproduction stage, the standard deviation belonging to a certain seed can be geometrically randomly perturbed (in other words, its value is halved or doubled with equal probability), so that some seeds become oriented toward small-scale improvements, while other seeds are oriented toward large-scale improvements. This phenomenon directly parallels the biological concept of a niche. Another option would be to assign$\sigma$ directly corresponding to the random perturbation size. That way, when the proverbial apple falls far from the tree, future apples will be more inclined to fall far from future trees. Either of these solutions would increase the convergence speed of the algorithm (because$\sigma$ would decrease when approaching a solution) while simultaneously increasing the span/reach of the algorithm on a large scale (because$\sigma$ would increase when hunting for a new solution).

None of these big three improvements were made, because Mathematica lacks a good support for the object-oriented programming style. If seeds were able to be represented as objects with properties and functions, implementing these three extensions would have been considerably easier and more appropriate. This is further motivation for rewriting the program in Java, C++, or even MATLAB. As stated previously, adding parallel optimization methods can increase convergence speed. While Newton's method can already be performed in parallel to the random-perturbation method, secant method would be a nice addition. The power of the algorithm grows with each method added, especially since the user will always have the option of turning off the extra methods.

Another convergence-related suggestion is that of skewed probability distributions for the random perturbations. Using the slope/gradient information when available, the program can make an educated guess about the direction of the optimal solution, similar to the way Newton's method works, and generate the new random 'child' seeds with a bias toward that direction.

To some extent, it would be beneficial to research optimal values for input parameters such as population capacity and reproduction ratio. The current default values have been found by crude trial-and-error. The optimal values for these fields are expected to depend highly upon the other input parameters, so to an extent, research here may be futile.

In light of the lack of post-run confidence provided by this random-perturbation method, it may be beneficial to add a test at the end of the program to see if the output makes sense as an optimal solution. The test would probe the output point, comparing it to nearby points, and verifying that the output point is indeed more optimal than its nearby points.

As stated previously, writing this program in a compiled language like C, C++, or FORTRAN will result in significant speed increases, especially since their random number generators are expected to be much more efficient than those in Mathematica. Again, the current incarnation is only a prototype. Speeding up the process will be a big step to making the algorithm feasible for real-world use.

A few suggested improvements are based on the evolutionary nature of the program. Whereas currently the successful seeds are differentiated from the unsuccessful ones only during the Survival-of-the-Fittest stage, it may be more effective and appropriate to also do so during the reproduction phase. High-ranking seeds would have more children than low-ranking seeds, making the probabilistic nature of the Natural Selection feature a bit less arbitrary and a bit more reliable.

Another evolution-inspired improvement would be to distribute virtual 'resources' to each seed based on how many other seeds are near it. Seeds in a crowded area would obtain less resources, while seeds with their own territory would have plenty resources. The amount of resources allocated to each seed would contribute to the quantity of its offspring or its ranking during the Survival-of-the-Fittest phase. This would prevent seeds from becoming overcrowded near a single point, which is a problem when it doesn't leave any room in the population capacity for 'adventurer' seeds, which roam the domain looking for alternatives to the current best-performing seed. Implementing this change would be tricky. Perhaps each seed's individual $\sigma$ (assuming the earlier improvements have been made) could be used as an aid in comparing the quantity of neighboring seeds to the expected quantity thereof.

Future researchers could also include sexual selection into seed survival rates. In other words, traits and features from multiple seeds could be blended during the reproduction phase. The benefits of this directly parallel those of biological sexual selection; the seeds will be given yet another method of randomizing the properties of their children. While this may be advantageous, this may be the most difficult and subjective improvement of all the ones listed here.

## Epilogue

Because the algorithm works similarly to biological evolution, the program serves a secondary purpose; it can be used as a teaching tool in grade school biology classes. A slight modification can allow the seeds to be visualized along the curve of a simple function like a parabola. Students can see how, as time progresses, order arises from the randomness, leading to an optimal type of organism. Furthermore, students can see how populations can stabilize at a local extremum for a long period and suddenly jump toward a distant location that is a better candidate for a global extremum. This parallels the real-world phenomenon wherein species often change in leaps and bounds rather than continuous, incremental improvement. This is just one of many lessons that can be learned in this manner. Needless to say, the metaphor is limited and currently cannot explain phenomena such as symbiosis, parasitic relationships, kin selection, etc.