Implementation and Analysis of a Genetic Algorithm

Tarik Tosun, MAE 345 Assignment 5

12/1/2011

Abstract

In this assignment, we implement and analyze a genetic algorithm in MatLab to determine the unknown parameters of a dynamic system. Cost function convergence and noise effects are explored. The effectiveness of the algorithm is then compared with MatLab's gradient-based *fminsearch* function.

1 Algorithm Implementation

1.1 Roulette-Wheel Selection

The program roulette.m was written as a roulette-wheel selection function for gene strings. The effectiveness of this function was tested by calling the function 1000 times with a given probability distribution. This test can be seen in the first cell of a5script.m, "Roulette Wheel Test". As the number of roulette spins was increased, the distribution was found to converge to the intended distribution, confirming the effectiveness of the function

1.2 Binary String Crossover

A random crossover function was implemented in randomCross.m. This function takes any even number of binary strings, randomly pairs them, randomly selects crossover points, and then performs simple crossover. Simple crossover is implemented in cross.m. This function is tested in the second cell of a5script.m, "Crossover Test." In this script, both randomCross.m and cross.m are tested using the four strings given in the assignment specification. Running this script several times with different parameters resulted in tests of many arbitrary pairings and crossover points, and from the results it was determined that the crossing functions worked properly.

1.3 Cost Function

A quadratic cost function was defined as suggested in the assignment specification. This cost function is implemented in fitness.m, which returns the fitness of a system state trajectory when given the trajectory and nominal trajectory as arguments. As indicated in the assignment specification, the function is made positive-definite by taking the negative exponential of the computed quadratic cost function.

1.4 Iterative Solution script

The iterative genetic algorithm solution script is found in genetic.m. We define two genes per chromosome, with one representing ζ and the other representing ω_n. Given the step response plot for the dynamic system, a reasonable range both of these values is (0,2). To reflect this range, the least significant bit in the genes was scaled to represent $2/4096 = 0.000488$.

The genetic algorithm used 32 chromosomes seeded with a random number generator. In order to compute each J (cost function value), a step response was performed or the system. This is done in simulate_sys.m, which also loads and returns values for the nominal response.

2 Results

2.1 Convergence and Optimal Values

The algorithm iterates until the total generation fitness converges to a stable value. For this problem, that value was found to be 0.9832. Attaining this value took 110 generations and 37.17 seconds. The best-estimate values for ζ and ω_n produced by this algorithm were 0.4414 and 1.409, respectively.

Convergence of the cost function to the optimal value was explored. is a plot of J_{total} and J_{best} as they converge to the optimal values. As we can see in the plot, J_{best} converges very fast, after fewer than five iterations. It then changes very little over more than one hundred iterations, as Jtotal converges to a stable value. Jtotal converges to a stable value after about thirty iterations, remaining in much the same position until the stopping criterion is reached (a local deviation of less than 0.00001 from the average of the past fifty generations). Looking at these graphs, it appears that results of similar accuracy could be attained with less stringent stopping criteria. However, the exact results of a genetic algorithm are probabalistic, and after experimenting with many different stopping criteria, it was determined that more stringent criteria gave more consistent results.

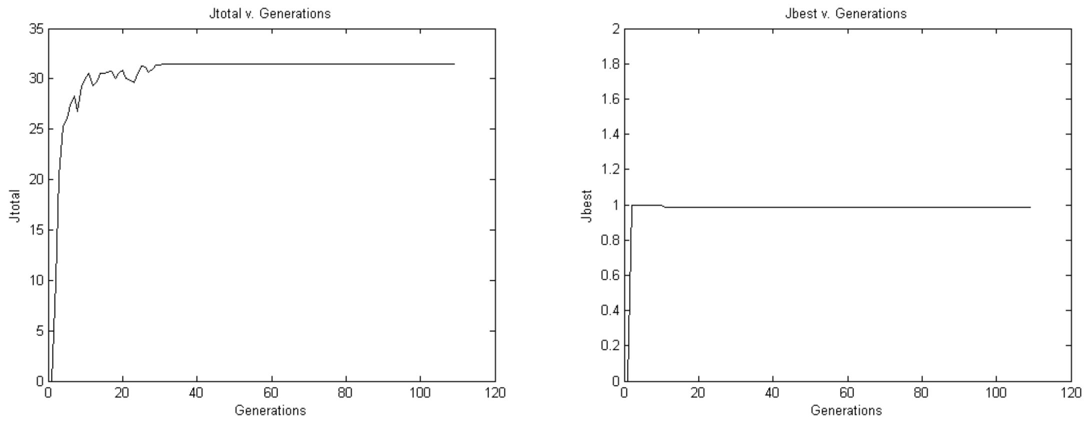

Figure 1: Plots of J_{total} and J_{best} as they converge to the optimal values.

2.2 Mutation Effects

The second argument of genetic.m, "mutation," allows the user to specify how often mutations occur in the algorithm. For this assignment, the effect of mutation every five generations was tested. Adding in mutations increased the runtime of the algorithm, causing it to grow to 152 generations and 48 seconds. However, it also increased the accuracy of the results, resulting in a J_{best} of 0.9955, and ζ and ω_n values of 0.4380 and 1.4961, respectively. Mutation addition, therefore, is something that must be considered on a case-by-case basis in the application of genetic algorithms. In cases where accuracy is valued much more highly than computational efficiency, mutations would certainly be beneficial, but if the genetic algorithm already places the available computational facilities near their practical limit, the addition of mutations is not recommended.

2.3 Effects of Gaussian Random Error

A gaussian random error with 30% standard deviation was introduced into the cost function residual. This drastically increased the runtime of the algorithm, causing it to run for 247 generations and 82 seconds. Unlike the addition of mutations, the addition of gaussian random error did not increase accuracy - it resulted in a J_{best} value of 0.9694 and values of 0.5332 and 1.5103 respectively for ζ and ω_n. Based on these results, the introduction of gaussian random into a genetic algorithm is not recommended.

The combined effects of gaussian random error and mutations were also tested with a 30% gaussian random error and mutations every 5 generations. Unforunately, this proved beyond the computational limits of the computer being used, causing it to run out of physical memory and shut down.

2.4 Comparison to a Gradient-Based Method

To evaluate the effectiveness of the genetic algorithm, its performance was compared with MatLab's *fminsearch* function, which employs a gradient-based search. The cost function had to be slightly modified (made negative) in order to be used with *fminsearch*, as this solver minimizes the objective function rather than maximizing it. Values of 0.4 and 1.5, were found for ζ and ω_n (repsectively) using *fminsearch*. These values produced a cost function value of J=1, indicating a perfect match. The *fminsearch* algorithm was able to produce these results in about one second, a fraction of the time it took the genetic algorithm to get comparable results. For this problem, it is clear that a gradient-based search is a better choice than a genetic algorithm

Conclusions

From the results of this assignment, we conclude that genetic algorithms are powerful but slow optimization tools. For this problem, the genetic algorithm implemented was vastly outperformed by a gradient-based algorithm, which was able to produce better results in less time. We infer from this result that the cost function for this problem has few local minima, and that the *fminsearch* algorithm was able to easily ride the gradient of the function to its optimal value. However, there are many optimization problem that cannot be solved through gradient-based methods. If the cost function has many local minima, a gradient-based algorithm will likely become 'trapped' in one of these minima and be unable to find the global minimum. In these cases, a more cumbersome gradient-free algorithm like a genetic algorithm must be employed.

As an example, part of my thesis research involves configuring a robot arm to minimize the integral of a potential field over its kinematic skeleton (joints connected by straight lines). The objective function (the skeleton integral) has many input variables (the arm's joint angles), and is highly nonlinear with many local minima. Currently, I use MatLab's gradient-based *fmincon* function to optimize the arm's pose. This function works very well (and very quickly) when the arm's initial pose is not far from the optimum, but fails miserably due to local minima when the starting pose is far from optimal. Within the next few weeks, I plan to try a genetic algorithm as a solution to this problem. Based on what I have learned in this class, I expect that it will fare much better than *fmincon*, but that it may also prove very slow.