Final Report on Solving the 1D Schroedinger Equation using the Genetic Algorithm and a Simple Neural Network.

Joshua Klukas

Submitted December 17, 2020

0.1 ALGORITHM AND CODE

Following a paper by Sugawara[1], the aim of this project was to pre-optimize the parameters of a neural network using the genetic algorithm and subsequently use back-propagation to find an accurate solution of the 1D Schroedinger equation for some known energy state. This was done in C++ with the help of two classes, GeneticArray and NeuralNetwork.

0.1.1 NeuralNetwork Class

The NeuralNetwork class initializes a simple neural network with one input neuron, 6 hidden neurons, and two output neurons by storing their corresponding weights and biases. These are initialized with the setParameters() function, which accepts a double array from the GeneticArray class. The main purpose of this class is to evaluate the two output neurons, o_1 and o_2, in order to calculate the wave function

$$\Psi = o_1 sin(o_2) \tag{1}$$

at some point, x. There is also a method to perform back-propagation on each of the network's parameters, θ, as

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial R}{\partial \theta} \tag{2}$$

where

$$R = \frac{\sum_{x_i}^{M} |(\hat{H} - E)\Psi(x_i)|^2}{\sum_{x_i}^{M} |\Psi(x_i)|^2}. \tag{3}$$

is the fitness function evaluated at M random points on some symmetric interval. The private variable, E, represents the energy of one of the eigenstates of the harmonic oscillator potential, which was hard-coded into the NeuralNetwork class.

0.1.2 GeneticArray Class

The GeneticArray class generates and stores the weights and biases of a private NeuralNetwork object in floating-point binary as shown below

$$[w_j, \ldots, b_j, \ldots w_{jk}, \ldots, b_k, \ldots] \tag{4}$$

where w_j and b_j are the weights and biases from the input layer to the hidden layer, w_{jk} are the weights from the hidden layer to the output layer, and b_k are the biases of the two output neurons. The constructor randomly initializes binaryArray[], stores the corresponding values in doubleArray[], and passes the doubleArray[] to the private NeuralNetwork object, NN, using setParameters(). For every time that the genetic algorithm resets the population, the randomize() function is called, which does the same thing as the constructor. The + operator was overloaded to mate two parent arrays by looping through the left object's binary array and switching bits with the right object's binary array with some fixed probability. In calculating the convergence of a population, the - minus operator was overloaded to return the total number of different bits between two GeneticArray objects.

0.1.3 Main Program

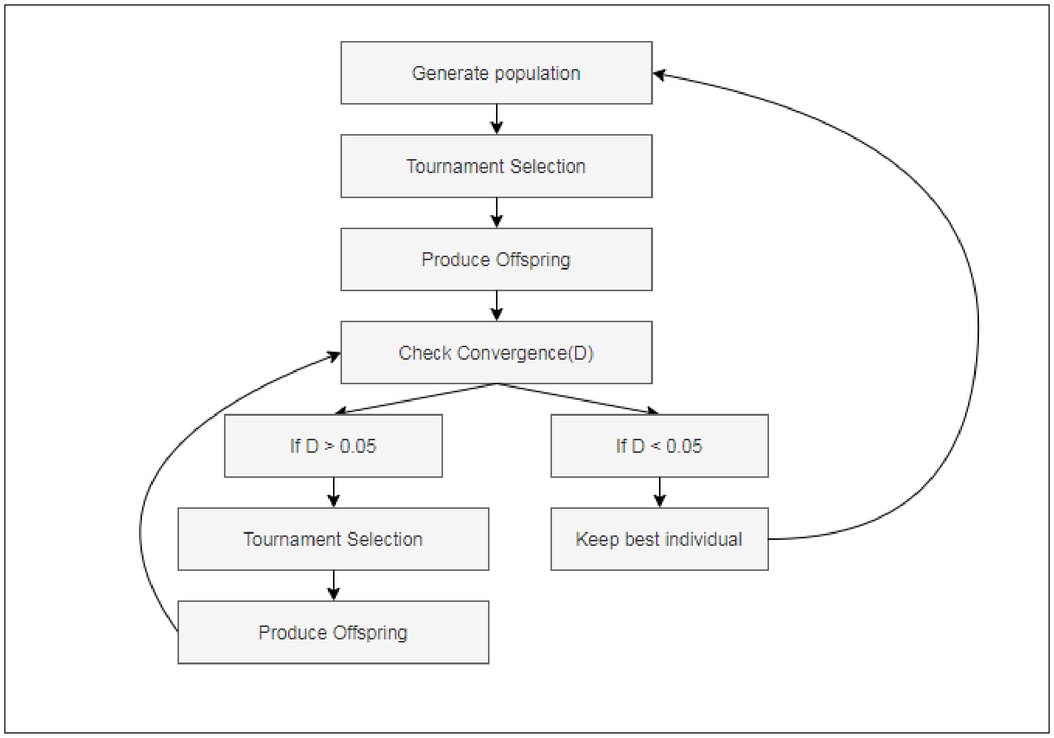

Figure 1: *Algorithm performed by one iteration of the main loop.*

In the main program, a population of N=50 genetic arrays is used and R is calculated using M=100 evaluations on the interval $-5 < x < 5$. The two arrays, isParent[] and isAlive[], keep track of the indices of surviving arrays during the competition stage. Each iteration of the main loop produces one converged population as shown in figure 1.

Population generation is carried out by the generatePopulation() function by calling the randomize() class method on each array in population[] except the best array. Tournament selection is carried out by the compete() function, which selects 25 unique pairs of arrays, determines the fittest array for each pair, and stores the winners in parents[]. The mate() function is then called to replace the N/2 losers from the competition stage with N/2 offspring from randomly selected pairs of parents. The fittest array is then found using findFittestIdx() and the convergence is calculated using calculateD(). The compete() and mate() functions are called until the population has converged. Once a satisfactory array is obtained, the optimize() class method is called to perform back-propagation on the neural network.

0.2 RESULTS

0.2.1 Pre-optimized Wave Function using Genetic Algorithm

The algorithm was applied to the 1D harmonic oscillator potential $V = \frac{1}{2}x^2$ with known energies, $E = n + \frac{1}{2}$, where n is an integer. Given that the R value was not very sensitive to the shape of the wave function, it was often the case that the algorithm would get stuck on a wave function with the wrong symmetry properties such as shown in figure 2 for the zeroth energy state.

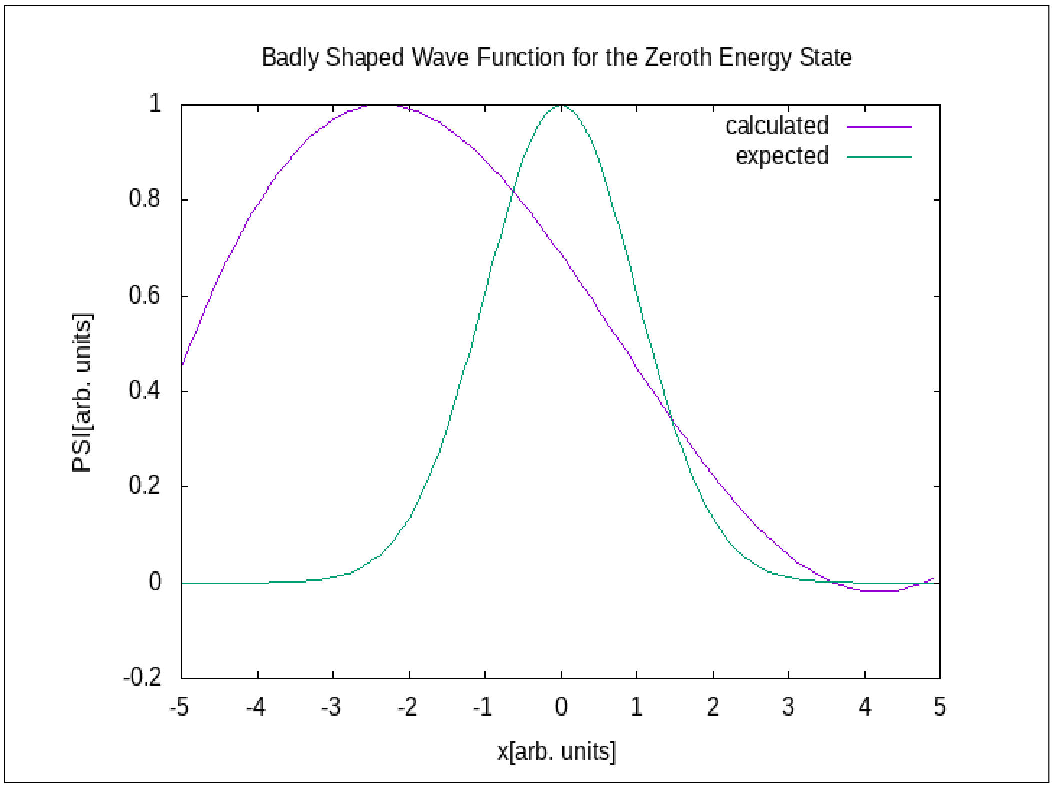

Figure 2: *Badly shaped wave function for the zeroth energy state with an R value close to 0.5.*

As a result, the algorithm converged very slowly, as R values of less than unity were often only obtained after 10000 generations as shown in Figure 3 below.

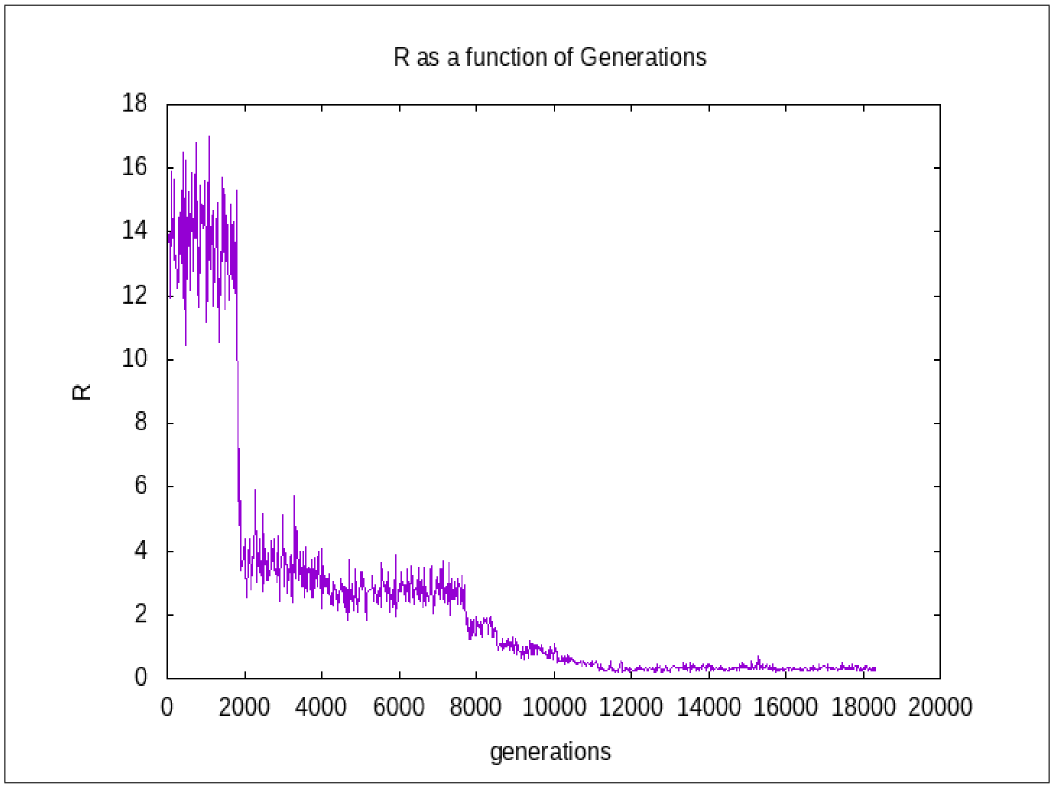

Figure 3: *Typical plot of the fitness of the best array with respect to the number of generations.*

It was proposed by Sugawara[1] that by reducing the number of random point evaluations of R, the algorithm would be more likely to sort out badly shaped wave functions due to the increased likelihood of the points being localized to one region. This was not observed to be the case, and it is suspected by the author that niching may have been implemented by Sugawara[1], whereby a diverse range of candidates is maintained in the population by decreasing the fitness scores of individuals that tend to clump towards one characteristic. Had this been implemented, it is expected that the algorithm would have converged much faster since niching would prevent it from getting stuck on one badly shaped wave function.

Despite the lack of niching, decent solutions were obtained for the zeroth and first energy states shown in figures 4 and 5. It should be noted that solutions were not observed for higher energy states since interestingly, the algorithm would often converge to the solutions for the zeroth and first energy states.

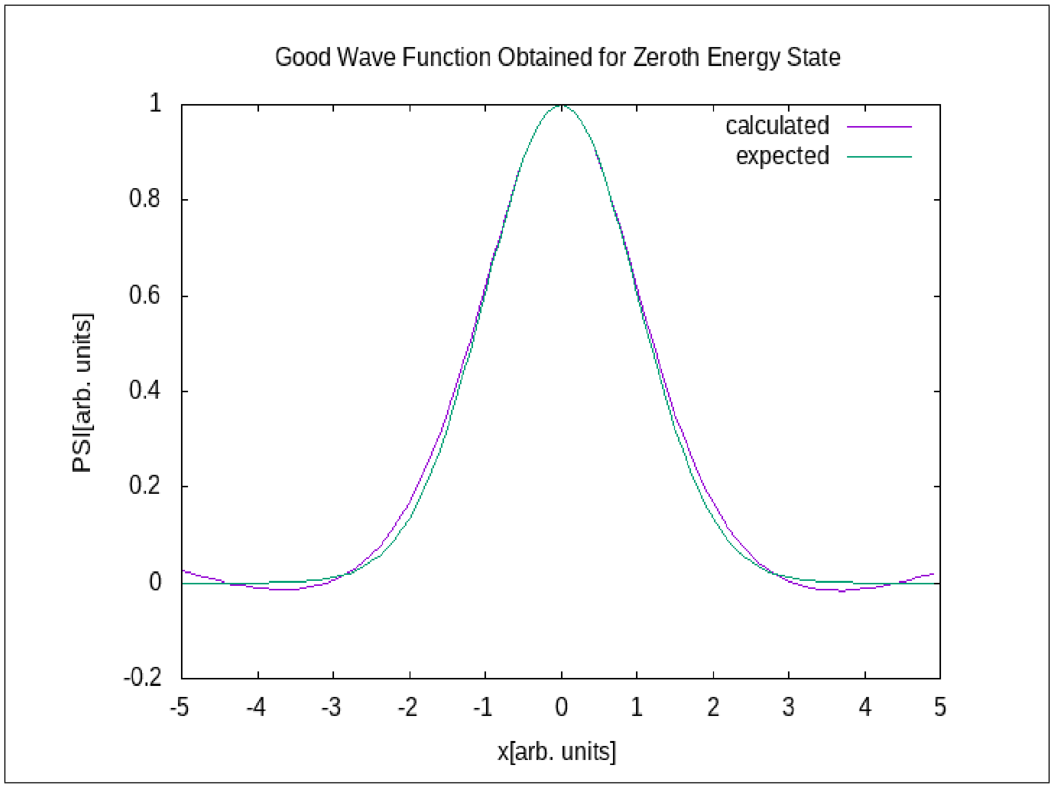

Figure 4: *Wave function obtained for the zeroth energy state using the genetic algorithm.*

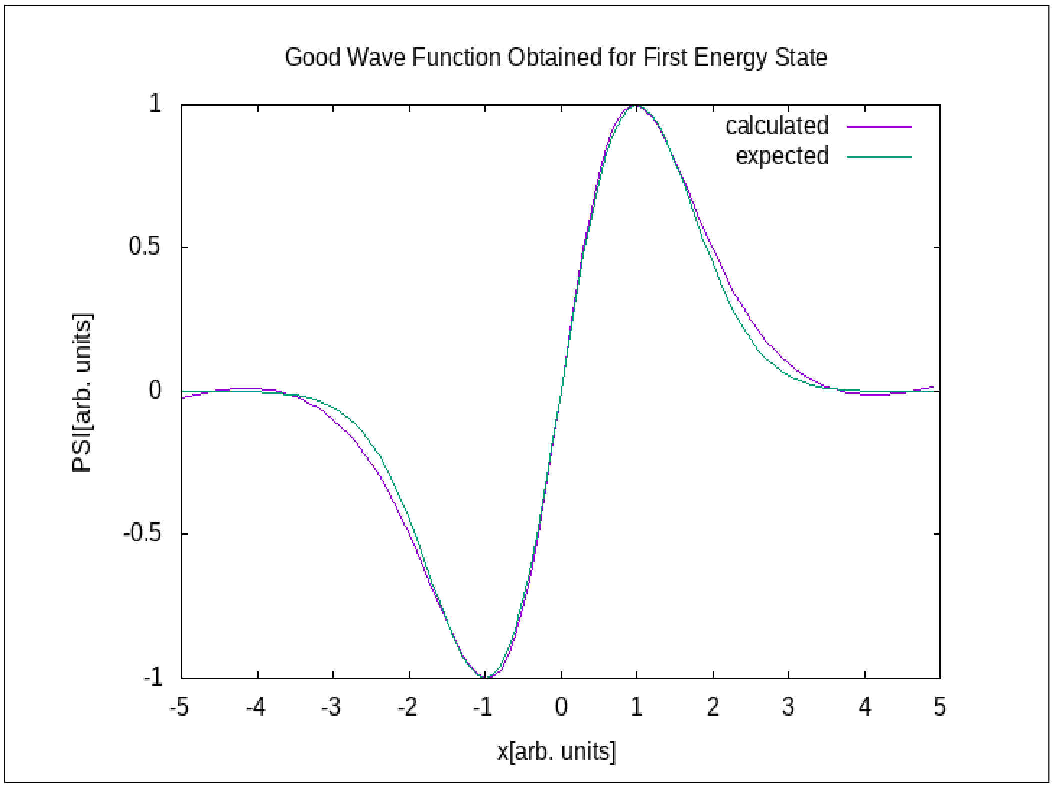

Figure 5: *Wave function obtained for the first energy state using the genetic algorithm.*

0.2.2 Optimized Wave Function using Back-Propagation

An attempt was made to apply the method of back-propagation to the two solutions above, but it was observed that after improving the R value for some time, the algorithm would start to fail and R would either approach infinity or stay the same. When applied to wave functions with worse R values, the algorithm immediately failed since as noted by Sugawara[1], there must be sufficient decomposition of the amplitude and phase components of the wave function.

It is possible that the algorithm cannot deal with the large number of local minima in the problem, even when the solution is very close. It is however just as likely that an error was made in calculating $\frac{\partial R}{\partial \theta}$, since the expression turned out to be quite complicated.

Bibliography

[1] Sugawara, M. “Numerical solution of the Schrödinger equation by neural network and genetic algorithm.” Computer Physics Communications, 140 (2001) 366–380.