7-20-2017

Optimizing an Electron's Path to Ionization Using a Genetic Algorithm

Jason Bennett
Ursinus College, jabennett@ursinus.edu

Kevin Choice
Ursinus College, kechoice@ursinus.edu

Optimizing an electron's path to ionization using a genetic algorithm

Jason Bennett and Kevin Choice
Mentor: Thomas Carroll
Ursinus College

July 2017

Abstract

A Rydberg atom is an atom with a highly excited and weakly bound valence electron. A widespread method of studying quantum mechanics with Rydberg atoms is to ionize the electron and measure its arrival time. We use a Genetic Algorithm(GA) to control the electron’s path to ionization. The Rydberg electron’s energy levels are strongly shifted by the presence of an electric field. The energy levels shift and curve, but never cross. At an avoided crossing the electron can jump from one level to the next. By engineering the electric field’s time dependence, we thereby control the path to ionization.

A GA is an optimization method modeled on natural selection. We use a GA to evolve electric field pulses to achieve a target path to ionization. Our algorithm initially generates random members of a population and then assigns each member a fitness score based on how well they achieve our target solution. We use elitism to pass the best members of the population directly into the next generation. We use tournament style selection to choose fit members to mate and pass their genes to the next generation. We then mutate the offspring to provide genetic diversity to our population. We present our results on the effects of varying GA parameters and modifying the GA to better model the experiment.

Introduction

A proven method of studying the phenomena that occur in the quantum mechanical realm is attempting to control the system of interest. The quantum control in this experiment involves controlling the ionization of electrons from a group of Rydberg atoms. A Rydberg atom is an atom excited to a Rydberg state, with its valence electron weakly bound to the core. This valence electron is easily ionized with an electric field. The higher the energy of the state, the lower the electric field needed for ionization. A common way to study the initial state of the Rydberg electron is to slowly increase the electric field applied and measure the arrival time at a detector. With the presence of an electric field, the energy levels are shifted in what is known as the Stark Effect. The energy levels shift and curve, but never cross. The Stark Effect causes hundreds of these avoided crossings that must be passed through before ionization. At an avoided crossing the electron’s amplitude can traverse the crossing adiabatically or diabatically. In an adiabatic traversal, the probability amplitude remains in the current energy level, whereas in a diabatic traversal the electron’s amplitude can jump from one level to the next. The probability of this jump is calculated using the Landau-Zener approximation and is determined to depend on the rate of change of the electric field applied. In order to construct an electric field pulse with which we can control the probability of traversing the avoided crossings by jumping or maintaining course, we utilize a Genetic Algorithm [1].

A Genetic Algorithm (GA) is an optimization method based on the Darwinian principles of natural selection. Because of the hundreds of avoided crossings that must be traversed, there are upwards of 10^1260 possible electric field pulses that produce the desired path. GAs excel at looking for optimal solutions among a large amount of possible candidates because they only require a target to evolve towards as opposed to finding an exact solution. Our algorithm begins by generating a set number of random members of a population and then immediately evaluates them against the target to determine their fitness score. The population is then ranked in terms of highest performers to worst performers. We use the technique known as elitism to immediately pass along the best performing members of the current generation into the next generation. This guarantees that optimal solutions are not lost in the evolution process. Our next step is to select fit members of the population to mate and propagate their genes into the next generation. We utilize the method of tournament selection to find the best fit parents. We partition the remaining population into sets of a given size and find the best members of the sets. We then allow the winner of one tournament to mate with a winner of another tournament through crossover of their pulses to create offspring for the next generation. The last step is to mutate the population. This is done by randomly selecting members that are being sent to the next generation and assigning them a new random pulse [1].

The various parameters involved in our GA vary the ultimate performance greatly. We present the results of varying select parameters to determine what features an optimal GA would contain. In order to test the various parameters with different sizes and run through 25 generations, we utilize a partial run to ionization. We increase the electric field from 0 to 150 V/cm as opposed to the 429 V/cm used in a full ionization run. By doing this we save an immense amount of computation time, but must change the way we test fitness score. To evaluate fitness score, we determine the state of the electron and designate a higher state that will be act as the target. The fitness score is determined by how much of the 100% of the electrons amplitude is in the target state. While testing the various parameters we keep the following features constant: the minimum and maximum principal quantum numbers (n) of 29 and 33 respectively, starting and ending electric field values of 0 and 150 V/cm respectively, a time resolution of 0.1 ns (the time between snapshots taken in the experiment), and a secondary total angular momentum quantum number (mj) of 1.5. The standard values of parameters that we maintain while testing the other parameters are: a population size of 32, a tournament size of 4, and elitism size of 4, and a mutation rate of 0.02. We test elitism sizes of 1, 2, 4, and 8, tournament sizes of 2, 4, 6, 8, 12, and 16, mutation rates of 0, 0.005, 0.01, 0.02, and 0.04, and population sizes of 16, 32, 48, and 64. We presents results and draw conclusions from these tests.

As mentioned above, with these parameter tests we are not applying an electric field strong enough to fully ionize the sample. To study whether we can draw the same conclusions from a partial ionization run as with full ionization, we compare the results of the tournament size of 4 parameter test and a full path ionization test with the same parameters. Despite only being taken to partial ionization, we draw similar conclusion about tournament size. The results seen in terms of amount of improvement the algorithm provides are in line with the experiment.

We examine the effects of assigning a difficult gate for the algorithm to achieve solutions for. The difficulty of the gate is determined by how much signal naturally occurring in the target gate region. An easy gate would be one in which some of the pulse is naturally within the target area, as opposed to a hard gate where initially few little or no signal is within the target gate.

The apparatus that produces the electric field pulses is called an arbitrary wave form generator. Our particular device has a 14-bit resolution that is capable of producing more field values than our simulation currently accounts for. We write a function in Mathematica and implement it in C++ within the algorithm that would allow for discrete electric field values that are more closely modeled with the experiment's arbitrary wave form generator. This increases the accuracy to which our simulation can produce required electric field pulses.

We write a function to verify the fitness score calculation performed by the algorithm to confirm the results we see.

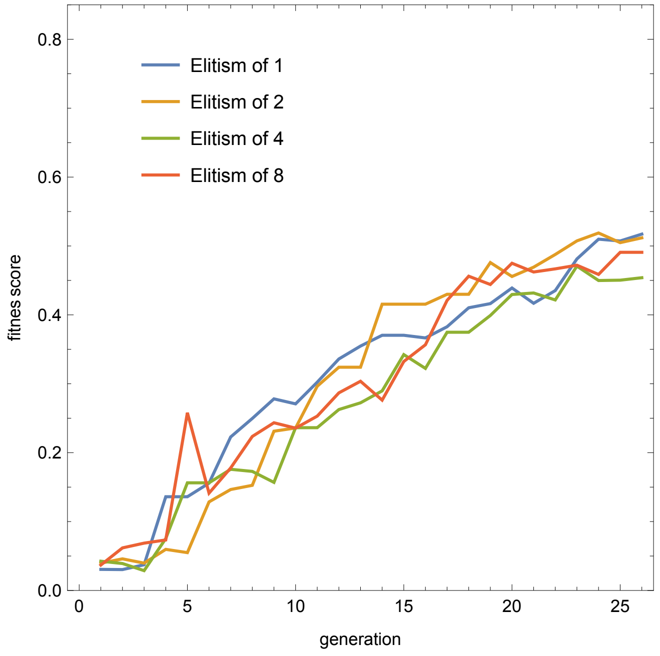
In order to visualize the increase in signal within the target gate as the algorithm evolves across generations, we write a function to calculate the fittest member of a given generation and plot the pulse produced by this member of the population across multiple generations.

Elitism Parameter Testing

We analyze the effect of elitism size on a partial ionization run. We cannot conclude that any elitism size is better than another, but we see that none of the runs are plateauing at generation 25, so had we run 50 or 100 generations we may have been able to make some conclusions about optimal elitism size.

```
elit1fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/elit1/elit1_score.dat"][[1, 1]];
elit2fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/elit2/elit2_score.dat"][[1, 1]];
elit4fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/elit4/elit4_score.dat"][[1, 1]];
elit8fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/elit8/elit8_score.dat"][[1, 1]];
```

```
elitParamTest =
 ListLinePlot[{elit1fitnessScores, elit2fitnessScores, elit4fitnessScores,
   elit8fitnessScores}, Frame → True, FrameLabel → {generation, fitnes score},
  PlotRange → {0, 0.85}, AspectRatio → 1, PlotLegends → Placed[
    {"Elitism of 1", "Elitism of 2", "Elitism of 4", "Elitism of 8"}, {0.25, 0.8}]]
```

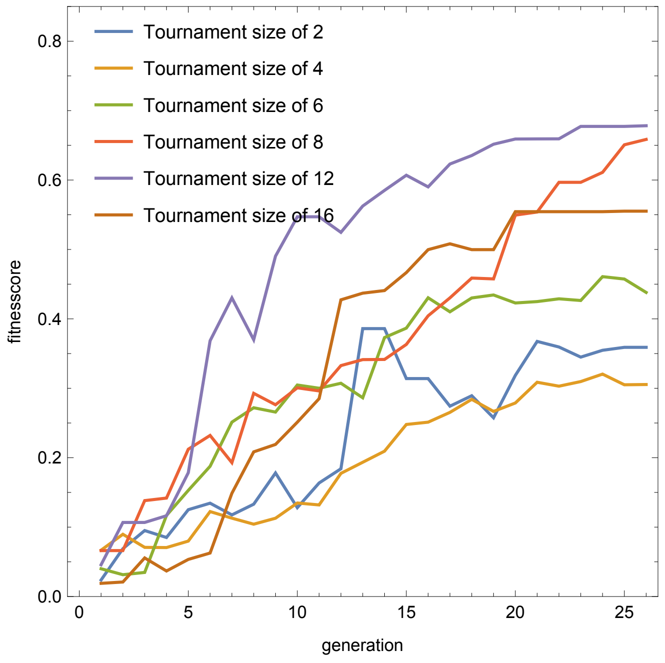


Tournament Size Parameter Testing

We analyze the effect of tournament size on a partial ionization run. We see that intermediate tournament sizes are better than the extremes. We see that a tournament size too small allows unfit parents to mate and pass on genes that should not be allowed to propagate, where as a tournament size too large allows only the best parents to mate, restricting genetic diversity. A tournament size of 8 and 12 perform the strongest, however a size of 8 may be argued to be the stronger of the two because of its continued increase at generation 25 whereas a size of 12 seems to be plateauing. In this study the population size is 32, a constant we held through the elitism, mutation rate, and tournament size studies. It could be that a ratio of tournament size to population is what determines the performance. A ratio of 1/4, corresponding to a tournament size of 8 and a population of 32, could be the optimal choice.

```
tourn2fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/tourn2/tourn2_score.dat"][[1, 1]];
tourn4fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/tourn4/tourn4_score.dat"][[1, 1]];
tourn6fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/tourn6/tourn6_score.dat"][[1, 1]];
tourn8fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/tourn8/tourn8_score.dat"][[1, 1]];
tourn12fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/tourn12/tourn12_score.dat"][[1, 1]];
tourn16fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/tourn16/tourn16_score.dat"][[1, 1]];
```

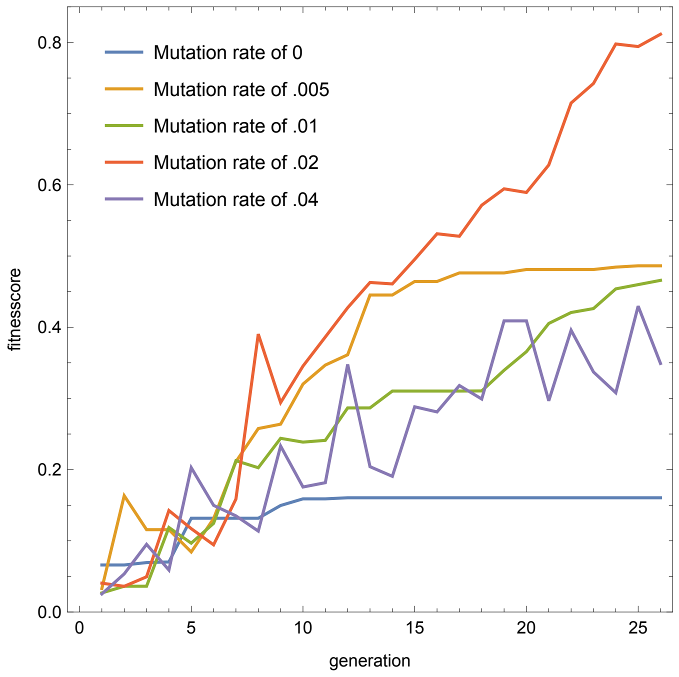
```
tournParamTest =
 ListLinePlot[{tourn2fitnessScores, tourn4fitnessScores, tourn6fitnessScores,
   tourn8fitnessScores, tourn12fitnessScores, tourn16fitnessScores},
  Frame → True, FrameLabel → {generation, fitnesscore}, PlotRange → {0, 0.85},
  AspectRatio → 1, PlotLegends → Placed[{"Tournament size of 2",
     "Tournament size of 4", "Tournament size of 6", "Tournament size of 8",
     "Tournament size of 12", "Tournament size of 16"}, {0.25, 0.8}]]
```



Mutation Rate Parameter Testing

We analyze the effect of mutation rate on a partial ionization run. As with tournament size, we see that intermediate mutation rates are better than the extremes. We see that a mutation rate too low causes the algorithm to converge to a solution far too early; a mutation rate of 0 plateaus around the 10th generation. We see that a mutation rate too high can be harmful to high performing members of the population. There may be good solutions that are mutated and hence lost. Again the intermediate rates are much stronger, with a mutation rate of 0.02 far outperforming the rest. As well as having a very high fitness score, it has not plateaued at generation 25.

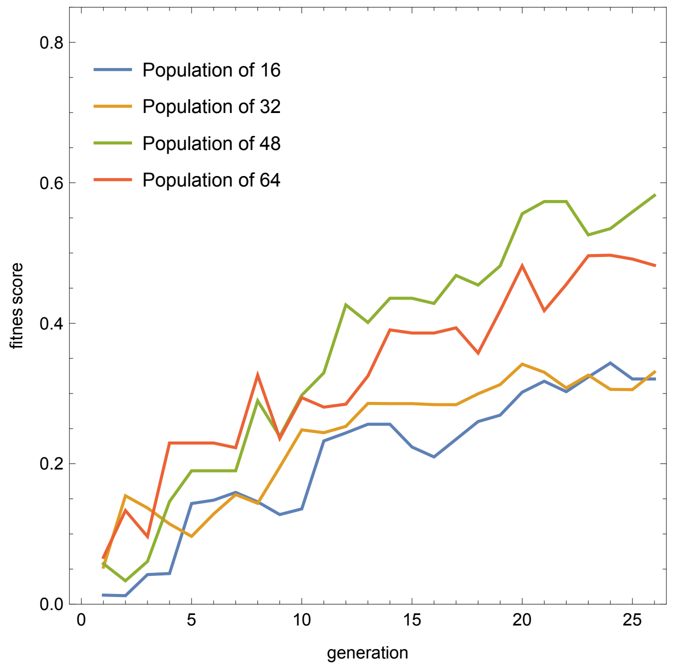
```
mut0fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/mut0/mut0_score.dat"][[1, 1]];
mut005fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/mut005/mut005_score.dat"][[1, 1]];
mut01fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/mut01/mut01_score.dat"][[1, 1]];
mut02fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/mut02/mut02_score.dat"][[1, 1]];
mut04fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/mut04/mut04_score.dat"][[1, 1]];

mutRateParamTest = ListLinePlot[{mut0fitnessScores, mut005fitnessScores,
    mut01fitnessScores, mut02fitnessScores, mut04fitnessScores}, PlotLegends →
    Placed[{"Mutation rate of 0", "Mutation rate of .005", "Mutation rate of .01",
      "Mutation rate of .02", "Mutation rate of .04"}, {0.25, 0.8}], Frame → True,
  FrameLabel → {generation, fitnesscore}, PlotRange → {0, 0.85}, AspectRatio → 1]
```



Population Size Parameter Testing

We analyze the effect of population size on a partial ionization run. Although we do not necessarily see this with the results, intuitively we know that with a higher population size, there is a higher possibility of more genetic diversity in the initial randomly generated population simply because of more initial members. The reason for the discrepancy in the data, notably that a population size of 32 is better performing than a population size of 64, may be due simply to a poorly scoring initial population with a population size of 64, or a highly scoring initial population with a population size of 32.

```
pop16fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/pop16/pop16_score.dat"][[1, 1]];
pop32fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/pop32/pop32_score.dat"][[1, 1]];
pop48fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/pop48/pop48_score.dat"][[1, 1]];
pop64fitnessScores =
  ReadList["/Users/boymommy148/Desktop/param/pop64/pop64_score.dat"][[1, 1]];

popSizeParamTest = ListLinePlot[{pop16fitnessScores,
   pop32fitnessScores, pop48fitnessScores, pop64fitnessScores},
  PlotLegends → Placed[{"Population of 16", "Population of 32",
     "Population of 48", "Population of 64"}, {0.2, 0.8}], Frame → True,
  FrameLabel → {generation, fitnes score}, PlotRange → {0, 0.85}, AspectRatio → 1]
```

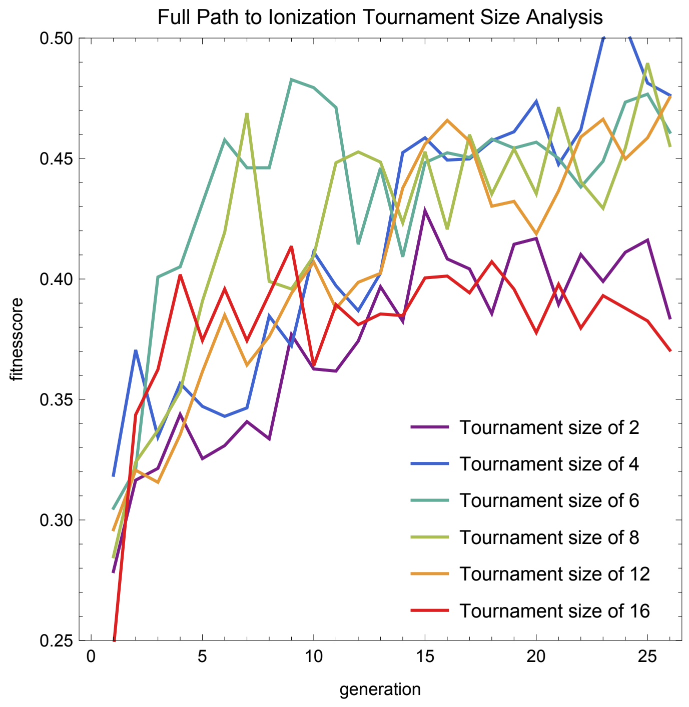


Full Ionization Path Parameter Testing comparing with Tournament Size

We compare runs with identical parameters in an attempt to see if similar conclusions can be drawn from a partial ionization run. A partial ionization run takes a fraction of the time a full ionization run does, so seeing similar results would allow for easier parameter analysis. We find that the same conclusions can be drawn from a tournament size parameter analysis using a partial run as seen above as tournParamTest. The results seen in terms of amount of improvement the algorithm provides are in line with the experiment.

```
tourn2FullPathFitnessScores =
  ReadList["/Users/boymommy148/Desktop/tourn2FullPath/tourn2fullPath_score.dat"][[
   1, 1]];
tourn4FullPathFitnessScores =
  ReadList["/Users/boymommy148/Desktop/tourn4FullPath/tourn4fullPath_score.dat"][[
   1, 1]];
tourn6FullPathFitnessScores =
  ReadList["/Users/boymommy148/Desktop/tourn6FullPath/tourn6fullPath_score.dat"][[
   1, 1]];
tourn8FullPathFitnessScores =
  ReadList["/Users/boymommy148/Desktop/tourn8FullPath/tourn8fullPath_score.dat"][[
   1, 1]];
tourn12FullPathFitnessScores = ReadList[
    "/Users/boymommy148/Desktop/tourn12FullPath/tourn12fullPath_score.dat"][[
   1, 1]];
tourn16FullPathFitnessScores = ReadList[
    "/Users/boymommy148/Desktop/tourn16FullPath/tourn16fullPath_score.dat"][[
   1, 1]];
```

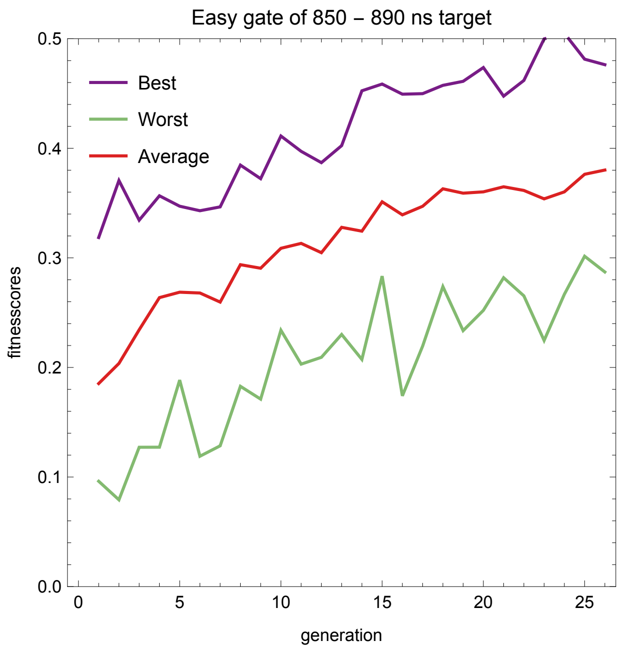
```
tournFullPathParamTest =
 ListLinePlot[{tourn2FullPathFitnessScores, tourn4FullPathFitnessScores,
   tourn6FullPathFitnessScores, tourn8FullPathFitnessScores,
   tourn12FullPathFitnessScores, tourn16FullPathFitnessScores},
  PlotLegends → Placed[{"Tournament size of 2", "Tournament size of 4",
     "Tournament size of 6", "Tournament size of 8",
     "Tournament size of 12", "Tournament size of 16"}, {0.75, 0.2}],
  Frame → True, FrameLabel → {generation, fitnesscore}, PlotStyle → "Rainbow",
  PlotRange → {0.25, 0.5},
  PlotLabel → "Full Path to Ionization Tournament Size Analysis", AspectRatio → 1]
```



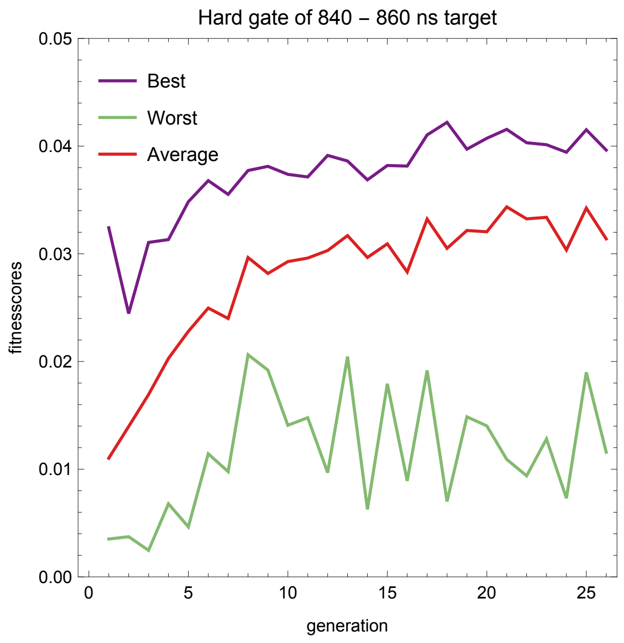
Comparing fitness scores with gates of 850 - 890 (easy) and gates of 840 - 860 (hard)

The effect of an assigned gate in a difficult region is shown with differing fitness scores for otherwise identical algorithms. For a much easier gate of 850 - 890 ns, the algorithm is able to achieve upwards of 0.5 fitness score, with a maximum of 1.0. For a much harder target gate of 840 - 860 ns, the algorithm does not even achieve scores of 0.05. We see however that the amount of improvement is higher in the hard gate results than the easy gate results. This is most likely due to the fact that some of the signal was initially in the target gate with the easy results. A key feature of the results is the continued increase in fitness score at generation 25, indicating that the algorithm could have improved further with additional allowed evolution.

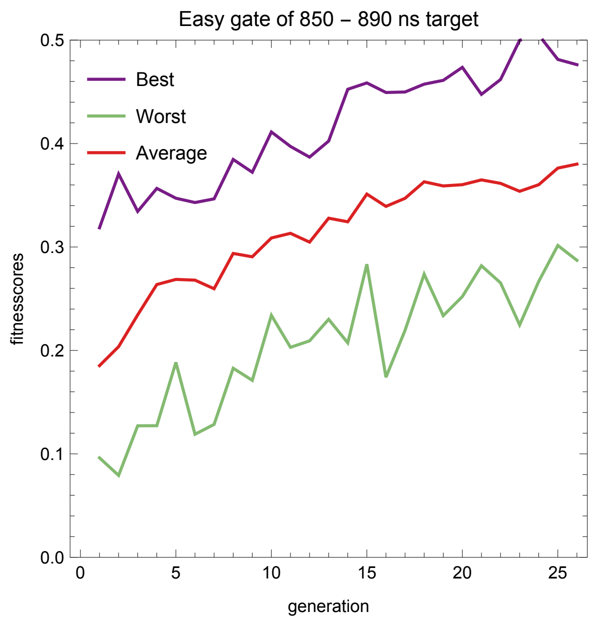
```
easierGateFitnessScores = ReadList[
    "/Users/boymommy148/Desktop/tourn4FullPath/tourn4fullPath_score.dat"][[1]];
easy = ListLinePlot[easierGateFitnessScores, Frame → True,
  FrameLabel → {generation, fitnesscores},
  PlotRange → {0, 0.5}, PlotStyle → "Rainbow", AspectRatio → 1,
  PlotLegends → Placed[{"Best", "Worst", "Average"}, {0.15, 0.85}],
  PlotLabel → "Easy gate of 850 - 890 ns target"]
```

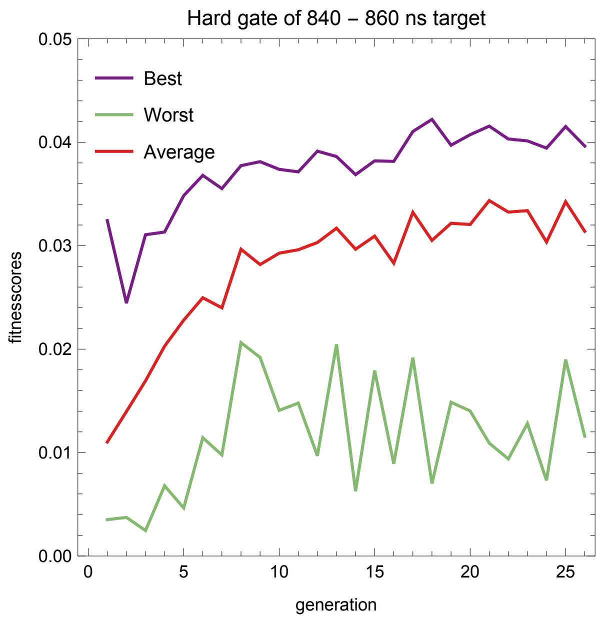


```
harderGateFitnessScores = ReadList[
    "/Users/boymommy148/Desktop/harderGate/fullPathNewGate2_score.dat"][[1]];
hard = ListLinePlot[harderGateFitnessScores, Frame → True,
  FrameLabel → {generation, fitnesscores},
  PlotRange → {0, 0.05}, PlotStyle → "Rainbow", AspectRatio → 1,
  PlotLegends → Placed[{"Best", "Worst", "Average"}, {0.15, 0.85}],
  PlotLabel → "Hard gate of 840 - 860 ns target"]
```



```
comparingHardGateToEasyGate = GraphicsColumn[{easy, hard}]
```





Function to calculate available electric field values

In an effort to allow for discrete electric field values that are more closely modeled with the experiment's arbitrary wave form generator, we have written pseudocode in Mathematica that we then implemented in C++ to be included in the algorithm. Our code allows for user input of only one value, the maximum electric field value, from which we determine the minimum step size the arbitrary wave form generator is experimentally capable of producing. With a minimum step size, we can determine the available field values by multiplying by a random integer in the given resolution range to emulate the random initial population.

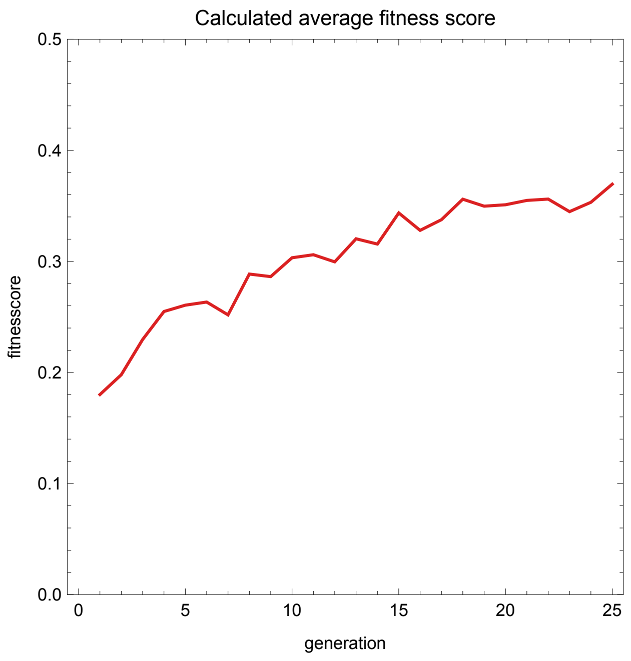
```
(*User inputs only fieldMax, and the negative is automatically the fieldMin*)
fieldMax = 3.8;
fieldMin = -1 * fieldMax;

(*number of bits to create resolution*)
numBits = 2 * 10 ^ 11;

(*11-bit resolution*)
resolution = 1 / numBits;

(*Minimum field step allowed with given resolution*)
minStepSize = fieldMax * resolution;

(*Integer to multiply by the step size that
 keeps the value inside the fieldMin and fieldMax*)
counter = RandomInteger[{-1 / resolution, 1 / resolution}];

(*Takes the minimum size step and multiplies it by a random
 integer in the resolution range to give a possible field value*)
availableFieldVals = minStepSize * counter

3.30567
```

Function to calculate the average fitness score

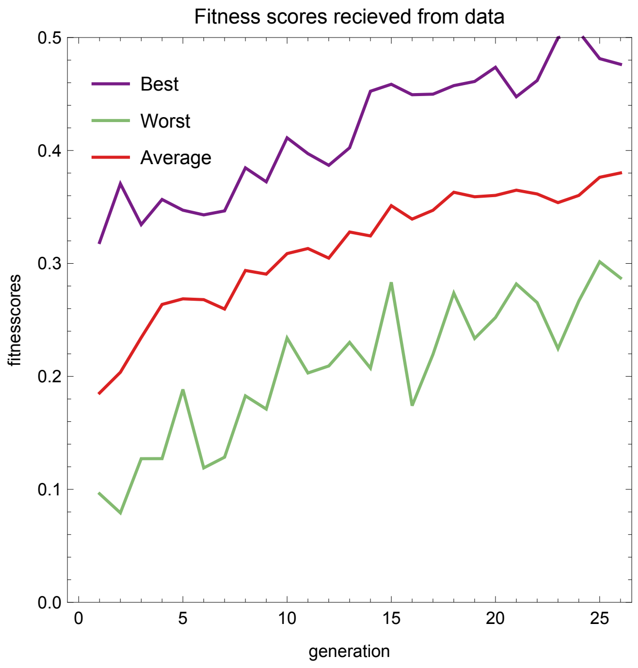
In an effort to check the fitness scores of our data, we have written a function to determine the average fitness score of the run data by looping over the runs, generations, and members. The data is normalized as an input, so we can simply add the data to a list if the pulse occupies the target gate given. We see that the algorithm is properly computing average fitness score.

```
tourn4FullPath = Table[Table[ReadList[
      "/Users/boymommy148/Desktop/tourn4FullPath/tourn4fullPath_mj_15_generation_
         " <> ToString[numGenerations] <>
        "_" <> ToString[numMembers] <> ".dat"][[1]],
    {numMembers, 0, 31}], {numGenerations, 0, 25}];
```

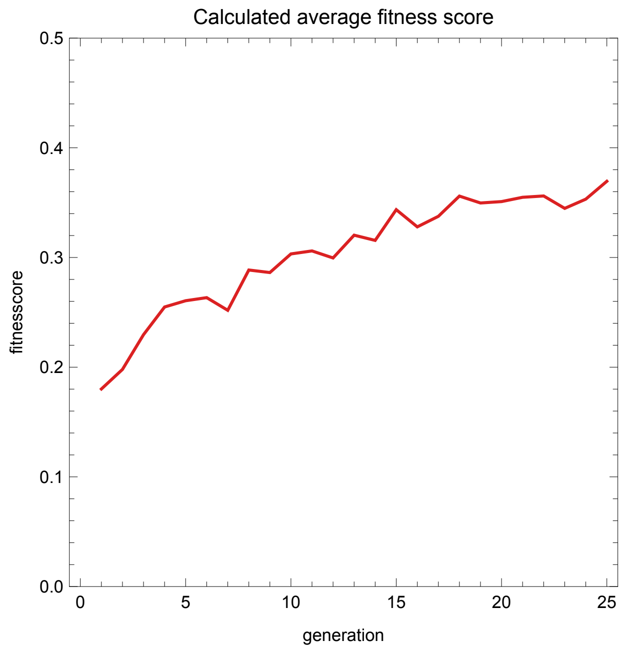
```
tourn4AverageFitScore = {};
For[k = 1, k < Length[tourn4FullPath], k++,
 generationAverageFitScore = 0;
 For[j = 1, j < Length[tourn4FullPath[[k]]], j++,
    For[i = 1, i < Length[tourn4FullPath[[k, j]]], i++,

    If [tourn4FullPath[[k, j]][[i, 1]] > 850 && tourn4FullPath[[k, j]][[i, 1]] < 890,
              generationAverageFitScore += tourn4FullPath[[k, j]][[i, 2]];
              ];
        ];
 ];
 generationAverageFitScore =
  generationAverageFitScore / Length[tourn4FullPath[[k]]];
 AppendTo[tourn4AverageFitScore, generationAverageFitScore];
]
calc = ListLinePlot[tourn4AverageFitScore, Frame → True,
  FrameLabel → {generation, fitnesscore}, PlotRange → {0, 0.5}, PlotStyle → "Rainbow",
  AspectRatio → 1, PlotLabel → "Calculated average fitness score"]
```

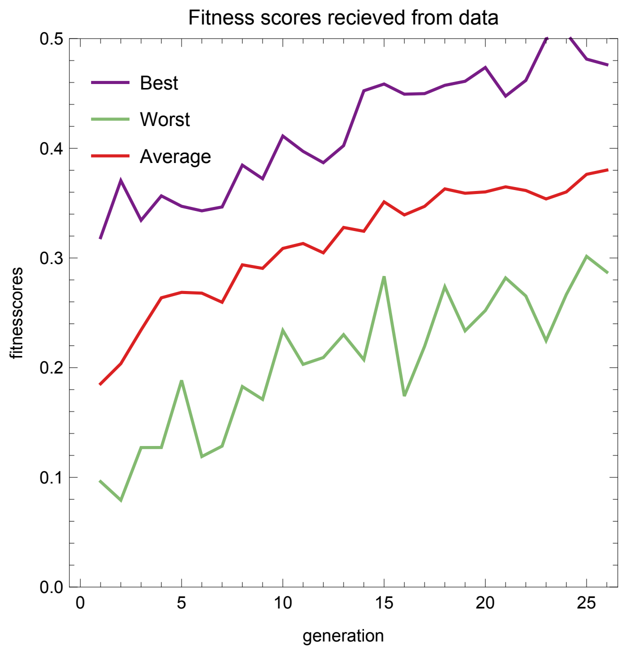


```
data = ListLinePlot[easierGateFitnessScores,
  Frame → True, FrameLabel → {generation, fitnesscores},
  PlotRange → {0, 0.5}, PlotStyle → "Rainbow", AspectRatio → 1,
  PlotLegends → Placed[{"Best", "Worst", "Average"}, {0.15, 0.85}],
  PlotLabel → "Fitness scores recieved from data"]
```



```
calculationVsData = GraphicsColumn[{calc, data}]
```





Function to calculate fittest member of each generation

In an effort to examine the the increase in fitness score as generation increases, we create a function that loops through all the members of a generation and calculates the fitness score. The function outputs the fittest member of the 1st, 5th, 10th, 15th, 20th, and 25th generations.

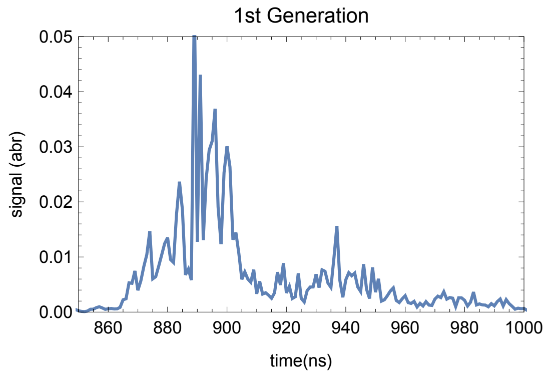
```
tourn12FullPath = Table[Table[ReadList[
      "/Users/boymommy148/Desktop/tourn12FullPath/tourn12fullPath_mj_15
         _generation_" <>
       ToString[numGenerations] <> "_" <> ToString[numMembers] <> ".dat"][[1]],
    {numMembers, 0, 31}], {numGenerations, 0, 25}];

For[k = 1, k < Length[tourn12FullPath], k++,
 temp = {};
 For[j = 1, j < Length[tourn12FullPath[[k]]], j++,
  fitscore = 0;
    For[i =  1,  i < Length[tourn12FullPath[[k, j]]], i++,
       If [
     tourn12FullPath[[k, j]][[i, 1]] > 850 && tourn12FullPath[[k, j]][[i, 1]] < 890,
             fitscore += tourn12FullPath[[k, j]][[i, 2]];
             ];
       ];
  AppendTo[temp, fitscore];
 ];
 If[k == 1 || Mod[k, 5] == 0,
  Print[Position[temp, Last[Sort[temp]]]];
 ];
]
```

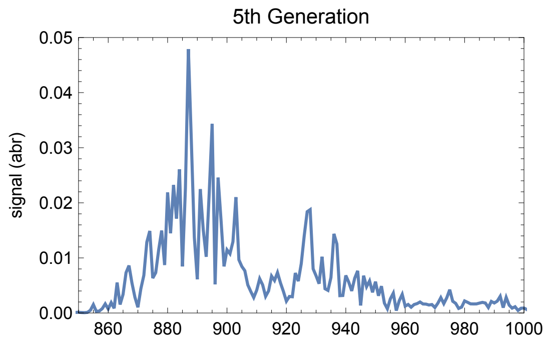
```
{{20}}

{{1}}

{{4}}

{{1}}

{{13}}

{{25}}
```

Examining the increase in fitness score across generations

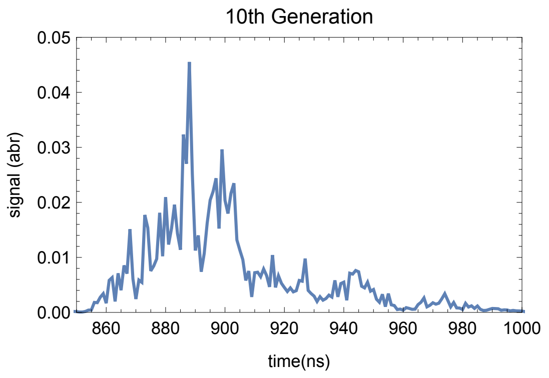
We show the evolution of the electric field pulses across generations. Fitness score is measured by how well the algorithm can allocate signal into the desired gate of 850 - 890 ns. We see an increase in the signal within the gate as generation increases.

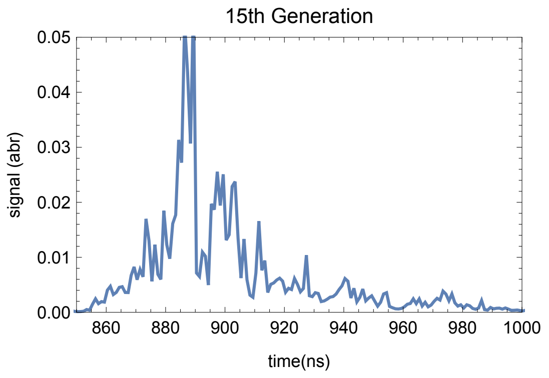
```
gen1 = ListLinePlot[tourn12FullPath[[1, 20]],
   PlotRange → {{850, 1000}, {0, 0.05}}, PlotLabel → "1st Generation",
   Frame → True, FrameLabel → {"time(ns)", "signal (abr)"}];
gen5 = ListLinePlot[tourn12FullPath[[5, 1]], PlotRange → {{850, 1000}, {0, 0.05}},
   PlotLabel → "5th Generation", Frame → True,
   FrameLabel → {"time(ns)", "signal (abr)"}];
gen10 = ListLinePlot[tourn12FullPath[[10, 4]],
   PlotRange → {{850, 1000}, {0, 0.05}}, PlotLabel → "10th Generation",
   Frame → True, FrameLabel → {"time(ns)", "signal (abr)"}];
gen15 = ListLinePlot[tourn12FullPath[[15, 1]],
   PlotRange → {{850, 1000}, {0, 0.05}}, PlotLabel → "15th Generation",
   Frame → True, FrameLabel → {"time(ns)", "signal (abr)"}];
gen20 = ListLinePlot[tourn12FullPath[[20, 13]],
   PlotRange → {{850, 1000}, {0, 0.05}}, PlotLabel → "20th Generation",
   Frame → True, FrameLabel → {"time(ns)", "signal (abr)"}];
gen25 = ListLinePlot[tourn12FullPath[[25, 25]],
   PlotRange → {{850, 1000}, {0, 0.05}}, PlotLabel → "25th Generation",
   Frame → True, FrameLabel → {"time(ns)", "signal (abr)"}];

gateEvolution = GraphicsColumn[{gen1, gen5, gen10, gen15, gen20, gen25}]
```

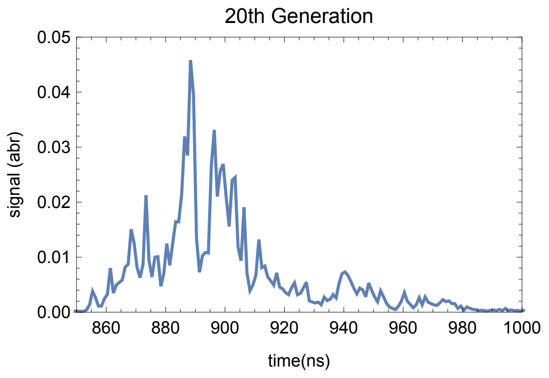


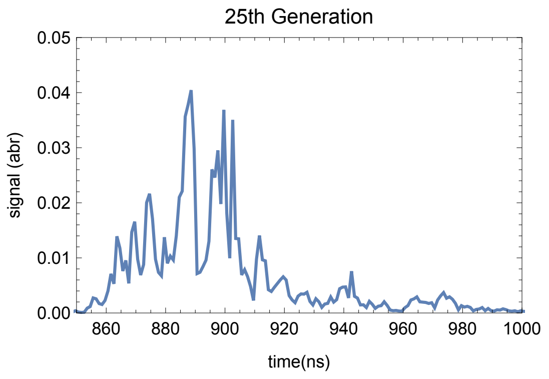


time(ns)


10th Generation
signal (abr)
0.05
0.04
0.03
0.02
0.01
0.00
860
880
900
920
940
960
980
1000
time(ns)


15th Generation
signal (abr)
0.05
0.04
0.03
0.02
0.01
0.00
860
880
900
920
940
960
980
1000
time(ns)


20th Generation
signal (abr)
0.05
0.04
0.03
0.02
0.01
0.00
860
880
900
920
940
960
980
1000
time(ns)


25th Generation
signal (abr)
0.05
0.04
0.03
0.02
0.01
0.00
860
880
900
920
940
960
980
1000
time(ns)

Conclusion

Though our analysis of the various parameters involved with our algorithm, we can draw conclusions about what features the optimal genetic algorithm would have. From the study on elitism we cannot conclude much due to the limiting generation of 25. This study could be continued with 50 or 100 generations, although the computation would take much longer even with the current state of the supercomputer used for these simulations. From the study on tournament size, and the subsequent confirmation with a full path to ionization analysis, we can conclude that an intermediate tournament size is key in a high performing algorithm. Similarly, our study on mutation rate allows us to state that an intermediate mutation rate is also optimal. Our study on population size seems to slightly contradict our intuition, but we are confident in the logic that a larger population would allow for higher fitness scores. Our study regarding the fitness scores of an easy gate versus a hard gate provides insight into just how far we can push the algorithm, but also leaves room for further study with the continued increases we see at generation 25. Our function to discretize the electric field values was implemented in the algorithm and will provide an opportunity to stimulate results that better model the experiment's arbitrary waveform generator's capability.

In the near future we plan, evolve a pulse that simultaneously sends the amplitude of one state adiabatically across a given avoided crossing, and the amplitude of a state that was previously indistinguishable from the latter, diabatically across a given crossing. The Stark effect for a single secondary total angular momentum quantum number (mj) value is complicated as it stands, we aim to use the engineered pulse to traverse all the avoided crossings present for the mj=1/2 and mj=3/2 states simultaneously.

References and Acknowledgments

[1] Vincent C. Gregoric, Xinyue Kang, Zhimin Cheryl Liu, Zoe A. Rowley, Thomas J. Carroll, and Michael W. Noel, "Quantum control via a genetic algorithm of the field ionization pathway of a Rydberg electron," arXiv:1704.01455 (2017).

This work was supported by the National Science Foundation under Grants No.1607335 and No.1607377.