Continuous Action Games: An Analysis of the Parameters and Convergent Properties of Best-Response (BR) and Gradient play (GP) Algorithms

Frank-Edward Nemeth

Abstract

A continuous action game involves a situation where a player's set of actions is infinite. In such scenarios, strategies can revolve around certain rules to select actions for the players involved. In order to find the convergence of actions towards a Nash Equilibrium, different algorithms, such as Best-Response and Gradient Play, can be considered.

The objective of this project is to simulate continuous action games in order to compare and contrast the two algorithms, Best-Response and Gradient Play. More specifically, this project aims to analyse each algorithm in terms of efficiency and effectiveness, as well as determine the situations where one may be preferred over the other.

Different continuous game simulations will also be constructed with various degrees of complexity. These simulations will be constructed with Python, using libraries such as Numpy and Sympy. Algorithms will be tested based on model measures, (such as iterations to converge and accuracy), hyperparameters (such as learning rate), as well as coding efficiency.

The results indicated that Gradient Play outperforms Best-Response play in most situations. The algorithm found solutions in less iterations and to a high degree of accuracy. Several plots comparing hyperparameters and their effects on convergence were also created.

In a broader sense, the discussions that come from this project serve as a baseline exploration into the topic of continuous games. It works as an additive, practical learning experience that applies the concepts covered in lecture. Furthermore, the simulation framework represents a structure by which to create, and test games for individual study as well as research.

Keywords: Continuous Action Games, Repeated Games, Gradient Play, Best-Response Play, Learning Rate, Convergence

Introduction

Background of relevant Concepts

All subparts of this section are taken from the Game Theory and Evolutionary Games accompanying notes for the course ECE1657 at The University of Toronto[1]

Continuous Kernel Games

Continuous Kernel games, also referred to as Infinite Action-set Games involve some predefined game, $\mathcal{g}(I,\ \Omega_i, J_i)$, where the following holds true:

- Ω_i: a convex, nonempty, compact set, and is continuous in its elements.
- $J_i : \Omega \ \rightarrow\ \mathbb{R}$ is a continuous function in all its arguments
- $u_i \in \Omega_i$ represents the action of player P_i

With notation $u = (u_i,\ u_{-i})$ representing the action profile of all players.

For a continuous game, there exists a Nash Equilibrium NE if for some optimal action $u^* \in \Omega$:

$$J_i(u_i^*,\ u_{-i}^*) \leq J_i(u_i,\ u_{-i}^*) \qquad (1)$$

Equivalently for any player P_i the action u_i^* belongs to his or her Best-Response strategy BR,

$$u_i^* \in BR_i(u_{-i}^*),\ \forall i \in I$$

$$u^* \in BR(u^*)$$

Where the Best-Response strategy BR is defined as:

$$BR_i(u_{-i}) = \operatorname*{argmin}_{u_i \in \Omega} J_i(u_i,\, u_{-i}) \qquad (2)$$

Debreu-Fan-Glicksberg NE Theorem

An important theorem when considering the convergence of continuous games to an NE is the Debreu-Fan-Glicksberg NE Theorem:

Consider a Game, defined as $\mathcal{G}(I,\ \Omega_i, J_i)$. Ω_i represents a <u>convex, nonempty, compact</u> set and J_i is a continuous function in u and is <u>convex</u> in u_i. Then $\mathcal{G}$ admits at least a Nash Equilibrium in "pure" strategies.

The use of this theorem allows for a guaranteed NE solution, so long as our game is convex it its set and cost function.

Repeated games and iterative algorithms

Consider a game, $\mathcal{G}(I,\ \Omega_i, J_i)$ where the players repeatedly play the game. As a result. It is iterated some number k times where each Player P_i uses the information of P_{-i} to update his or her strategy. There are mathematical proofs that indicate after some k iterations, the Players will gradually converge towards a NE solution, if one exists.

In order to find such a convergence, iterative algorithms can be used.

Best-Response (BR) Play:

$$\boxed{u_i^{k+1} = u_i^k + \alpha\left[BR_i\left(u_{-i}^k\right) - u_i^k\right]} \qquad (3)$$

Gradient Play:

$$\boxed{u_i^{k+1} = u_i^k - \alpha\, \nabla_{u_i} J_i(u_i^k, u_{-i}^k)} \qquad (4)$$

Where u_i^k represents Player P_i's strategy at iteration k of the game, u_i^{k+1} represents Player P_i's strategy at iteration k+1 of the game, and α represents some learning rate. In the Gradient Play algorithm, ∇_{u_i} represents a pseudo-gradient operator:

$$\nabla_{u_i} J_i = \frac{d}{du_i} J_i$$

Objective

The two algorithms mentioned were studied in the course ECE1657 at the University of Toronto. An interesting next step would be to analyze their specific properties in greater detail. This paper will serve as an extension of the lecture material using simulations.

The objective of this paper is to explore the two iterative algorithms, BR Play and Gradient Play in order to identify their characteristics and use cases in solving continuous games.

Specifically the following questions will be explored through experimenting with the cost functions:

- What are the limitations for BR Play and Gradient Play? When should one algorithm be used over the other?
- What is the efficiency of each algorithm? Which converges to a NE solution quicker in terms of time and number of iterations? When does one converge faster over the other?
- How does changing different parameters affect the performance of each algorithm?

The goal at the end of the project is to have some deliverable to encompass the results of this finding. Furthermore, the programming steps taken to achieve the simulations should be done in a repeatable and adaptable way to promote further testing and research into the topic.

Methods

Simulations

Repeated games were constructed and simulated using the programming language of Python. To construct these games, different Python libraries were utalized. Since this project will be dealing with a large quantity of numerical data, the Numpy[2] library was used to efficiently conduct array operations.

Research was done into libraries to handle the manipulation of equations. Specifically, taking the derivative in the Gradient Play algorithm posed an issue. Typically in a programming language, in order to calculate a derivative from a predetermined function, numerical approximations must be done. However, decimal values in Python sometimes carry errors that eventually can become prevalent after many iterations of the repeated algorithms. Another alternative is to compute the derivative outside of the programming environment, and pass it in as a function. This method would solve the issue of numerical errors showing up when taking the gradient, but it would also force a user to determine the derivatives on paper beforehand.

Instead, the Sympy[3] library was discovered, and was used to create symbolic equations for the various cost functions used. These functions were later automatically converted into Python functions to calculate specific values. This method also consequently allows for new cost functions to be easily defined and tested, making the simulations robust and scalable. An example code snippet illustrating symbolic equations with some comments appears as Figure C1 in Appendix C.

Definition of scope

To focus the problem into a more reasonable simulation, the games were limited to convex, 2-player games. This would guarantee a NE solution through the Debreu-Fan-Glicksberg NE Theorem, as mentioned in the Introduction. Furthermore, limiting the game to two players helps with the feasibility of programming. In terms of programming the functions, a stop condition of 300 000 iterations was used. With some parameters the iterative algorithms would not converge in a reasonable time, so the stop case was provided to stop long running code and Python crashes.

Pseudocode

Figures 1, 2 and 3 display pseudocode of the BR Play and Gradient Play.

Figure 1: Best Response Play Pseudocode

```
BR_play($i_1$, $i_2$, $u_1$, $u_2$, $J_1$, $J_2$, alpha, error):
    $u_1^k$ = i1
    $u_2^k$ = i2
    while(True):
        $u_1^{k+1}$ = $u_1^k$ + alpha*(BR($u_1$, $u_2^k$, $J_1$)- $u_1^k$)
        $u_2^{k+1}$ = $u_2^k$ + alpha*(BR($u_1^k$, $u_2$, $J_2$)- $u_2^k$)
        if (abs($u_1^{k+1}$ - $u_1^k$) < error) and (abs($u_2^{k+1}$ - $u_2^k$) < error):
            return ($u_1^{k+1}$, $u_2^{k+1}$)
        else:
            $u_1^k$ , $u_2^k$ = $u_1^{k+1}$, $u_2^{k+1}$
```

Figure 2: Best Response Strategy Pseudocode

```
BR($u_i$, $u_1$, $u_2$, J):
    J = J($u_1$, $u_2$)
    return $u_i$[J.argmin()]
```

Figure 3: Gradient Play Pseudocode

```
Gradient_play($i_1$, $i_2$, $u_1$, $u_2$, $J_1$, $J_2$, alpha, error):
    $u_1^k$ = i1
    $u_2^k$ = i2
    while(True):
        $u_1^{k+1}$ = $u_1^k$ - alpha*grad($u_1^k$, $u_2^k$, $J_1$)
        $u_2^{k+1}$ = $u_2^k$ - alpha*grad($u_2^k$, $u_1^k$, $J_2$)
        if (abs($u_1^{k+1}$ - $u_1^k$) < error) and (abs($u_2^{k+1}$ - $u_2^k$) < error):
            return ($u_1^{k+1}$, $u_2^{k+1}$)
        else:
            $u_1^k$ , $u_2^k$ = $u_1^{k+1}$, $u_2^{k+1}$
```

Each function makes use of the same variables:

- i_1 : initial condition P_1 (float)
- i_2 : initial condition P_2 (float)
- u_1 : matrix of actions for P_1 (array of floats)
- u_2 : matrix of actions for P_2 (array of floats)

- J_1 : cost function for P_1 (Python function)
- J_2 : cost function for P_2 (Python function)
- alpha : Learning Rate (float)
- e : Convergence Error (float)

The Convergence Error is defined as $\left|u_i^{k+1} - u_i^k\right| < e$, and is used as a stop condition for the algorithm. When the difference between successive steps drops below the convergence error, the algorithm is considered to have converged on a value. Another way to view the convergence error is the value $u_i^{k+1} - u_i^k = \alpha[function]$, from rearraging either (3) or (4). Ideally the term $\alpha[function]$ should equal zero. The error e defines an upper acceptable bound for this measure.

Figure 2 shows the BR strategy helper function, which is based on (2). In Figure 3, grad is a symbolic function, calculated similar to the example in Figure C1 in Appendix C.

The code itself was attempted to work as efficiently as possible. The while loops had the minimum amount of steps to prevent harsh time scaling for each function. Using the Numpy library where possible also helped to speed up repeated calculations in matrices, as per its documentation[2].

Testing procedure

Table 1 summarizes the different examples that were tested.

Example1 was taken from the Game Theory and Evolutionary Games notes[1] and was chosen as a preliminary testing game to ensure the functionality of all the code. Set space was the same as defined in the text. Initial conditions of (0,0) were chosen arbitrarily. The goal was to also analyze this function in depth.

Example2, Example3, and Example4 were specifically and meticulously constructed for this report. Each has a different polynomial order cost function (squared, cubed, and quartic order functions respectively). For the cost functions, the coefficient of the dominant terms, as well as the NE solution were kept the same across all the examples. This was done in an attempt to make the most distinguishing factor when comparing the functions the polynomial order of the dominant terms. The set space was an arbitrary range that included the NE solution. Initial conditions were also an arbitrary point within the set space.

Where possible, the default values for Learning Rate α and Convergence Error e were 0.01 and 0.00001 respectively. These values were chosen after experimentation with the functions when originally designing the simulations as well as consulting literature for common rates[4]. When varying Learning Rates in analysis, values between 0.0001 and 0.03 were chosen. When varying Convergence Error in analysis, values between 0.0000001 and 0.009 were chosen. These values were found to be optimal without causing too much numerical error when small, and still allowing the iterative algorithms to converge when large.

Table 1: Example Continuous Games

Label	Cost Functions, (J_1, J_2)	Solution, (u_1^*, u_2^*)	Set	Initial conditions, (u_1^1, u_2^1)
Example1	$J_1 = 2(u1)^2 - 2u1 - u1u2$ $J_2 = (u2)^2 - \frac{1}{2}u2 - u1u2$	$\left(\frac{9}{14}, \frac{5}{7}\right)$	[0,1]	$(0, 0)$
Example2	$J_1 = 2(u1)^2 - \frac{1}{4}u1 - u1u2$ $J_2 = \frac{2}{3}(u2)^2 - \frac{3}{4}u2 - u1u2$	$\left(\frac{1}{4}, \frac{3}{4}\right)$	[0.2,0.8]	$\left(\frac{1}{2}, \frac{1}{2}\right)$
Example3	$J_1 = 2(u1)^3 + \frac{3}{8}u1 - u1u2$ $J_2 = \frac{2}{3}(u2)^3 - \frac{7}{8}u2 - u1u2$	$\left(\frac{1}{4}, \frac{3}{4}\right)$	[0.2,0.8]	$\left(\frac{1}{2}, \frac{1}{2}\right)$
Example4	$J_1 = 2(u1)^4 + \frac{5}{8}u1 - u1u2$ $J_2 = (u2)^4 - \frac{7}{8}u2 - u1u2$	$\left(\frac{1}{4}, \frac{3}{4}\right)$	[0.2,0.8]	$\left(\frac{1}{2}, \frac{1}{2}\right)$

Results

The results from the simulations will be split into two sections. The first will use Example1 to highlight the differences between the two iterative algorithms by adjusting the hyperparameters. The same analysis was conducted on Example2, Example3 and Example4, but these graphs appear in Appendix A to save space. The trends with the hyperparameters appeared similar in all the graphs, and will be discussed holistically in the discussion.

The second part will show some of the differences between Example2, Example3 and Example4 to highlight how the polynomial order of the cost functions can affect the iterative algorithms.

Figure 4: Convergence for u Vs. Learning rate for Example1

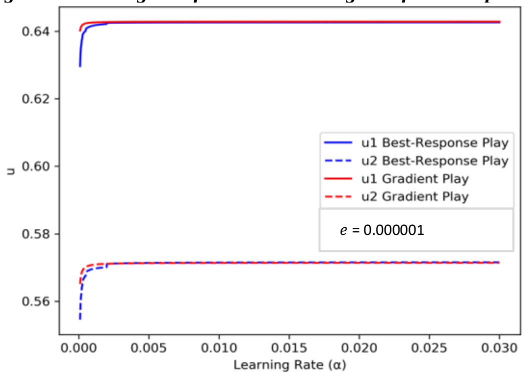


Figure 5: Convergence for u Vs. Convergence Error for Example1

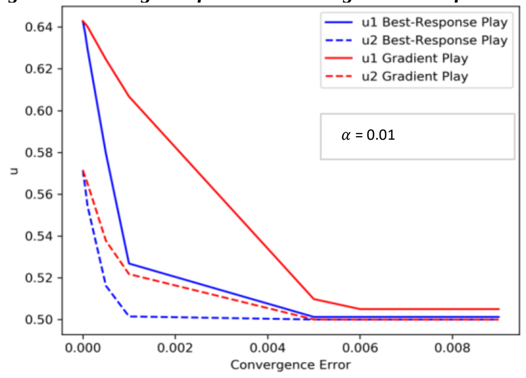


Figure 6: Convergence for u Vs. Iteration for Example1

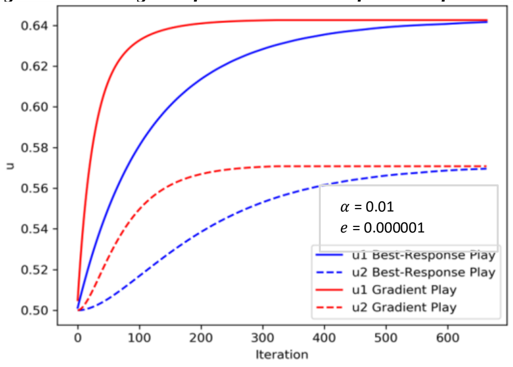


Figure 7: Convergence for J Vs. Iteration for Example1

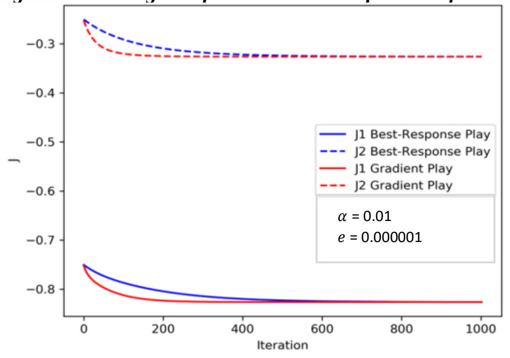


Figure 8: Iterations Vs. Learning Rate for Example1

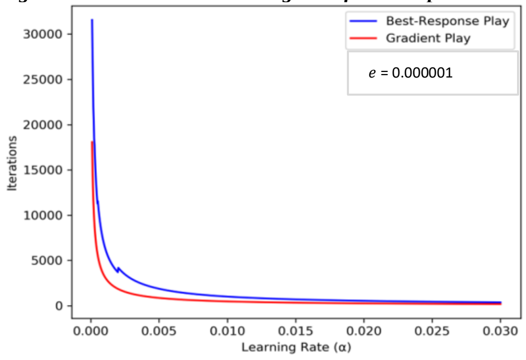


Figure 9: Iterations Vs. Convergence Error for Example1

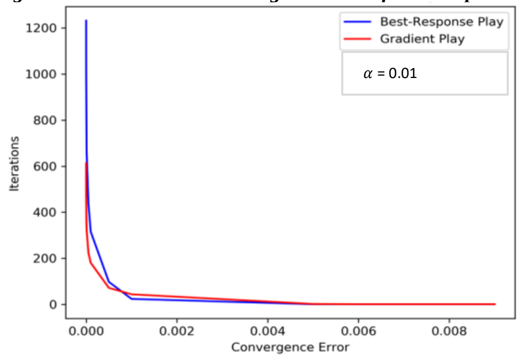


Figure 10: Execution Time Vs. Learning Rate for Example1

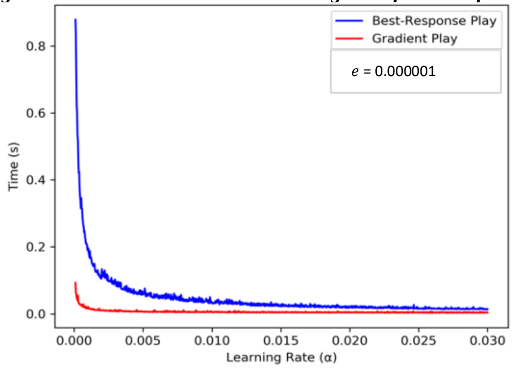


Figure 11: Execution Time Vs. Convergence Error for Example1

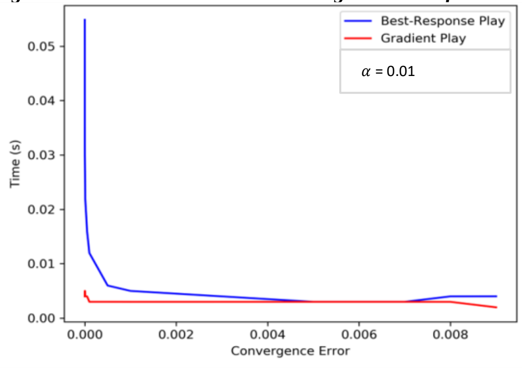


Figure 12: Iterations Vs Initial Conditions for Example1

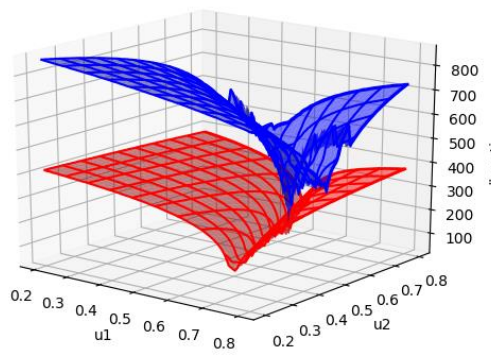
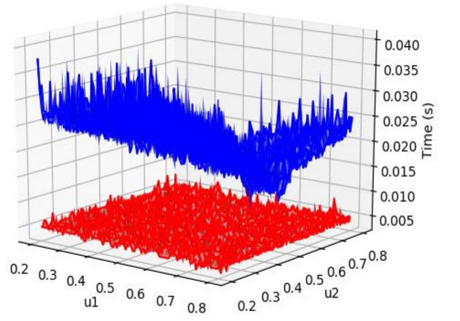


Figure 13: Time vs Initial Conditions for Example1



Example1 Plots

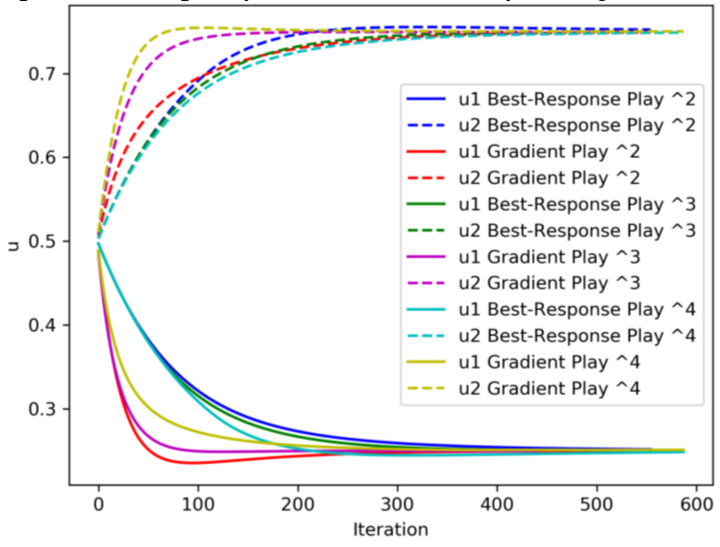
Parameter Evaluation

Plots appear on the previous page as Figures 4-11. Figures 4-6 show the converged Nash equilibrium solution ($\boldsymbol{u_1^*}, \boldsymbol{u_2^*}$), and how that changes based on Learning Rate, Convergence Error, and Iteration. Figure 7 shows the $\boldsymbol{J}^*$ convergence based on Iterations. Figures 8-11 attempt to show Iterations and Execution Time against the Learning Rate and Convergence Error in order to see how changing these hyperparameters would affect the speed and efficiency of each iterative algorithm. The corresponding plots for the remaining examples are in the Appendix.

Initial condition Analysis

Figures 12 and 13 show 3-Dimensional plots of the initial conditions for u_1 and u_2 and how that affects the number of iterations and time it takes to converge. The corresponding plots for the remaining examples are in the Appendix A.

Figure 14: Convergence for u Vs Iteration number for Examples 2, 3, and 4



Joint Plots for Example2, Example3, Example4

Figure 14 plots the NE convergences for Example2, Example3, and Example4. Table 2 calculates the error between the algorithms.

Table 2: Difference between determined NE and real NE (Learning Rate = 0.01, e = 0.000001

Label	NE (row1 = u1*, row2 = u2*)	BR Algorithm	$error = abs(NE - BR)$	GP Algorithm	$error = abs(NE - GP)$
Example1	0.64286	0.64276	0.00010	0.64283	0.00003
	0.57143	0.57129	0.00014	0.57137	0.00006
Example2	0.25000	0.25010	0.00010	0.25000	0.00000
	0.75000	0.75017	0.00017	0.74990	0.00010
Example3	0.25000	0.25002	0.00002	0.24995	0.00005
	0.75000	0.74992	0.00008	0.74995	0.00005
Example4	0.25000	0.24983	0.00017	0.25008	0.00008
	0.75000	0.74990	0.00010	0.75002	0.00002

Discussion

Analysis from Graphs

From looking at Figures 4-13, Gradient play was found in overall to converge faster in time, in iterations, and more accurately in some cases. Specifically Figures 4 – 6 show Gradient play achieving the equilibrium sooner (with a steeper slope) than the BR play. Furthermore the associated Figures for the other examples in the appendix show the same trend with little difference.

Table 2 shows that given the iterations, both the BR and Gradient Play end up converging very close to the NE (within 4 decimal places). However, even in these close settings, the Gradient Play outperforms BR (albeit small for this particular Learning Rate and Convergence Error)

Parameter anomalies

In Figure 4, for convergence with respect to α, it was expected that with a small α, the iterations would be more precise. Thinking of the Learning Rate α as a step size, some ituition indicates that small steps will allow the algorithms to achieve a precise solution. However, the small values ended up being the most inaccurate portions. This may be due to numerical approximations having more significant errors when dealing with smaller values, especially in a programming simulation. The smaller the α, the more significant rounding errors become.

It is important to note as well that these graphs had Learning Rate ranges selected to ensure convergence. High learning rates risk the iterative algorithms never converging. As shown with this project, a small Learning rate can carry its own errors as well.

Some of the other graphs intuitively made sense with one another (for instance, if Gradient Play performed better on iterations, it made sense that It would do better on timing as well).

Some important observations were in some of the convergence error graphs. Figure 5 indicated how important a small convergence error is, as many values further from zero end up with very inaccurate converged solutions.

Complexity Analysis

BR Play

We can refer to Figure 1 and 2 to determine a rough estimate of the complexity of the BR Play function. The while loop will run as long as the convergence is not met. We can call this at worst case k iterations. Inside the while loop, there is the BR function, depicted in Figure 2. For every iteration of the while loop, J needs to be updated with all the values of either u_1 or u_2. In a worst case scenario, this would be n calculations, where n is the length of the u_1 or u_2 vector. The J.argmin() function also runs each time the loop runs, however, minimizing functions can be of order log(n), so it is superseded by the prior calculation. As a result, the BR Play ends up with an approximate O(kn) complexity in this case.

Gradient Play

Let us assume the while loop for Gradient play in Figure 4 runs the same iterations as BR Play, making it run k times. In each while loop we utalize the ‘grad’ function. The grad function is generated before hand, and needs only be generated once. This function only runs one time, to update the equation based on the single dimensional parameters. As a result, it can be considered of complexity O(1). Thus we find the overall complexity to be an approximate order of O(k).

Complexity results

In terms of strictly runtime, the complexity analysis seems to coincide with the results of the simulations. The Gradient Play algorithm consistently ran faster than the BR play by a considerable margin. Consider the Example1 in Figures 10, 11, and 13, and the other examples in the Appendix as supplements. The only times the BR performed better is in some higher error settings (as seen on Figure 11). This seems very unusable however, as other Figures, such as 5, show that this regime is very erroneous and does not converge to the correct solution.

The complexity analysis between the two algorithms is potentially grimmer, as the BR Play consistently took more iterations to run as well. This would only magnify the difference in efficiency between the two algorithms.

A lot of the difference in performance might be accredited to this disparity in complexity. Both algorithms were attempted to be written in the most efficient way. One of the large advantages Gradient Play had over Best Response Play is the use of symbolic equations to calculate the gradient. This greatly reduced the runtime and number of calculations, as otherwise there would be large matrix multiplications to numerically predict the derivative. However, one such library for BR play was not found to exist during this project.

Limitations of Each algorithm

BR Play

In most the test cases when altering the hyperparameters, the BR algorithm performed worse than the Gradient Play algorithm. This included being less accurate it its convergence, as well as taking more time and iterations to converge.

Originally it was anticipated that the advantage of the BR play would be in large Learning Rate α cases, where Gradient play can sometimes end up diverging from the large step size. However, BR play ends up taking so many iterations that it is often infeasible to use anyways. In addition to taking many iterations, BR Play, as programmed in this project, had a higher runtime complexity than Gradient Play, making its clock speed much slower. This was especially prevalent in cases with many iterations. To prevent long running code and crashes, the BR Play usually was cut off before convergence in high Learning Rate cases (with these data points excluded).

Gradient Play

Gradient Play performed better in all the examples. This included using fewer iterations, less time, and converging more accurately in all the examples. The main area of struggle for Gradient Play is when the Learning Rate α is too high. This was typically found to be $\alpha > 0.03$ when the Convergence Error is set to 0.00001. As stated prior, these values are approximate, and were found while designing the simulation. When the learning rate is too large, the Gradient Play starts to exhibit divergent properties, and would often propagate errors until a Python overflow occurred.

Higher Order Polynomials Examples

Considering Figure 14, we see that despite the different order of polynomials, the NE solution is eventually converged to with both algorithms. Somewhat different than originally expected is that the higher order terms tend to converge faster to the equilibrium.

The intuition when making the higher order polynomial cost functions was to see if the increased ‘steepness’ would cause the algorithms to have difficult converging. A similar concept occurs with large Learning Rates, where the step size becomes too big. Instead it appeared that the polynomials were well within reason, converging faster and more efficiently (possibly because of their steepness). The algorithms likely moves faster down the gradient but somehow with the parameters tested did not end up overshooting.

Surprising as well, most other trends (Appendix A) appeared the same as with the original quadratic example. Some key differences include much more steepness on many of the curves. The 3D plots for the initial conditions also had a larger degree of separation between the BR play and Gradient Descent. The higher order terms likely exacerbated the differences in complexity between the two algorithms.

Initial condition Plots

The 3-Dimensional Plots were very interesting in terms of results, but also visually displayed some intuitive outcomes. In Figure 12 and 13, there are noticeable pin-like drops at the Nash Equilibrium solution, which makes sense that both algorithms would have an easier time converging when the initial conditions are the solution.

Something interesting about the dips and contours on the plots is that there are some times it is visible that the BR Play performs better than the Gradient Play. This is not applicable for all the examples, but there are some such points in Figure 12. For most the other plots in the Appendix, this is not observed. In general it holds the Gradient Play is significantly bounded by the BR Play algorithm.

The timing graph (Figure 13) is a lot more pliable, with many sharp peaks and valleys. This makes sense in terms of a computer program, as not only can the conditions and parameters themselves alter the calculations, but multiple computer tasks can spread the resources thin, making precise timing ineffective. However, from the 2D and 3D timing plots, it still holds that the Gradient Play performed better.

Possibly the most interesting trait that appears on all the initial condition 3D plots are very clear valleys or ridges. These ridges define certain combinations of initial values that perform much better than normally. A further investigation to try to quantize, or determine the actual relationship between the initial conditions would be interesting.

Conclusion

Overall, it was found that the Gradient Play Algorithm outperformed the Best Response Play Algorithm when trying to solve continuous action games.

Referring back to the original questions, both algorithms were found to have few limitations. In almost all cases the Gradient Play performed better. In cases with high learning rates, Best Response could converge, but in a less reasonable time, while Gradient Play usually would not. This would be one of the use cases for Best Response Play over Gradient Play. Furthermore, if the use of symbolic equations to calculate the gradient was not available, it would have likely seen a much more competitive performance between the two.

Future steps

Different cost functions

Originally, the scope of this project was limited to 2 player games. Making support of i>2 player can greatly impact the performance and characteristics of the algorithms.

Furthermore, different functions such as logarithms, exponentials, negative exponents, or even periodical functions can be interesting to test for convergent properties

Single Parameter Evaluations

This project focuses heavily on breadth of experimentation, looking at many different hyperparameters. Something that would likely be more useful would be to focus on one parameter to derive some sort of rule or model.

For instance, determining a method to pick an optimal Learning Rate would be highly applicable in many situations. Further analysis into the initial conditions, the apparent ridges that showed up in the 3D plot simulations, and how to best pick initial conditions would also be a fruitful future topic.

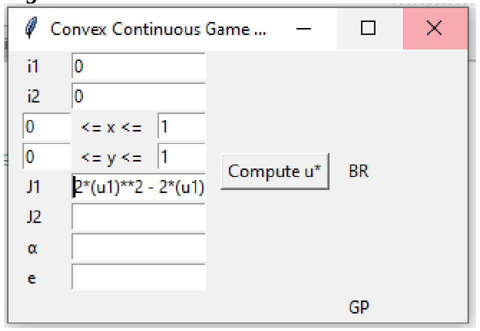
Graphical User Interface

One future step would be to include the results into some sort of product or tool. During this project, an original stretch goal was to create a Graphical User Interface (GUI) solver that would parse any particular cost functions and then run the simulations to find the NE. Some progress was made, but the graphing portions still contain bugs. Figure 15 shows the current state of the Graphical User Interface. An interesting and enjoyable future step would be to finish the adaptation of this, such that it can be used a teaching and experimental

Acknowledgements

I would like to thank Professor Lacra Pavel for presenting such an interesting course this semester. Credit for this project is in a large part to due to the extremely high quality of instruction and enthusiasm for Game Theory from Professor Pavel and her teaching team.

Figure 15: Continuous Game Solver GUI


Convex Continuous Game ...

i1	0	
i2	0	
0	<= x <=	1
0	<= y <=	1
J1	2*(u1)**2 - 2*(u1)	
J2		
α		
e		

References

[1] Lacra, P., 2016. *Game Theory And Evolutionary Games*. Toronto: Dept. of Electrical and Computer Engineering, University of Toronto.

[2] Numpy.org. 2020. *Overview — Numpy V1.19 Manual*. [online] Available: https://numpy.org/doc/stable/ [Accessed 12 December 2020].

[3] Docs.sympy.org. 2020. *Sympy 1.7 Documentation*. [online] Available: https://docs.sympy.org/latest/index.html [Accessed 12 December 2020].

[4]Y. Bengio, "Practical recommendations for gradient-based training of deep architectures", *CoRR*, vol. 12065533, 2012. Available: http://arxiv.org/abs/1206.5533. [Accessed 12 December 2020].

Appendix A: Additional Plots

Figure A1: Convergence for u Vs. Learning rate for Example2

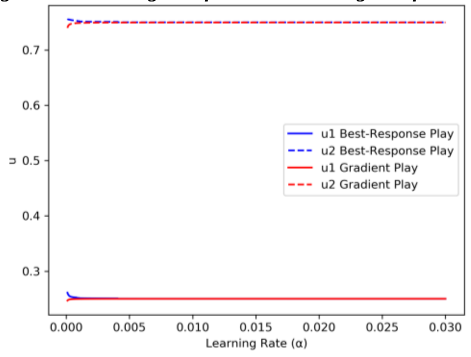


Figure A2: Convergence for u Vs. Convergence Error for Example2

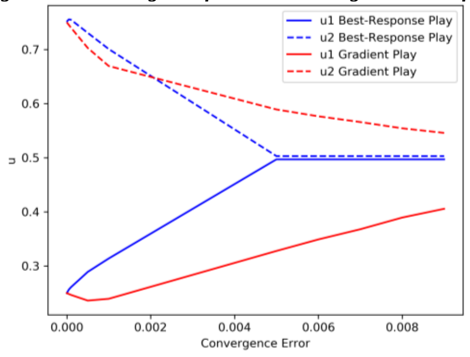


Figure A3: Convergence for u Vs. Iteration for Example2

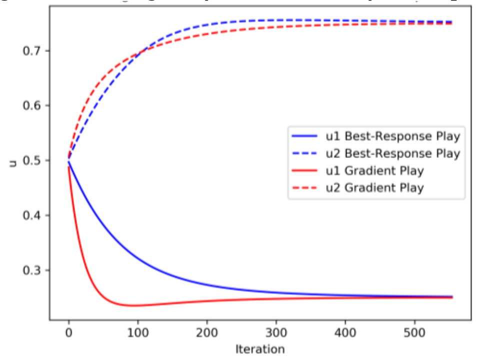


Figure A4: Convergence for J Vs. Iteration for Example2

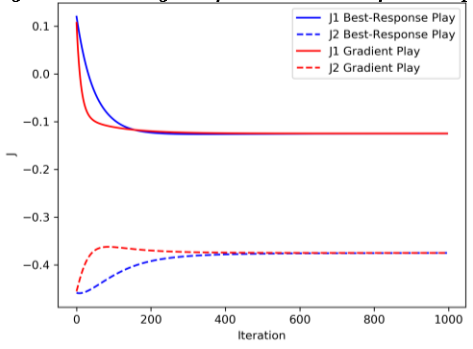


Figure A5: Iterations Vs. Learning Rate for Example2

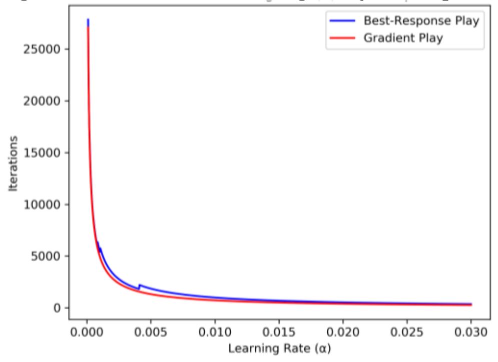


Figure A6: Iterations Vs. Convergence Error for Example2

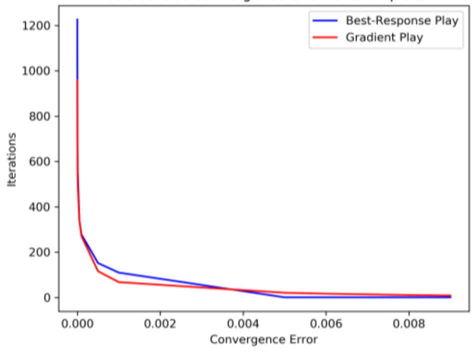


Figure A7: Execution Time Vs. Learning Rate for Example2

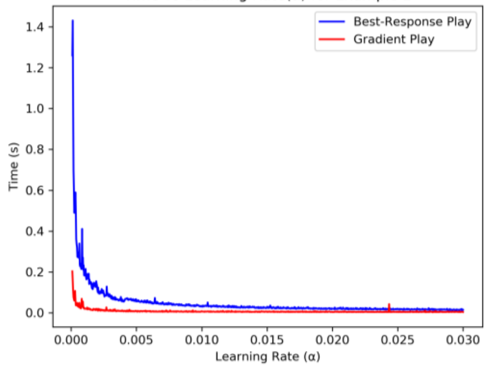


Figure A8: Execution Time Vs. Convergence Error for Example2

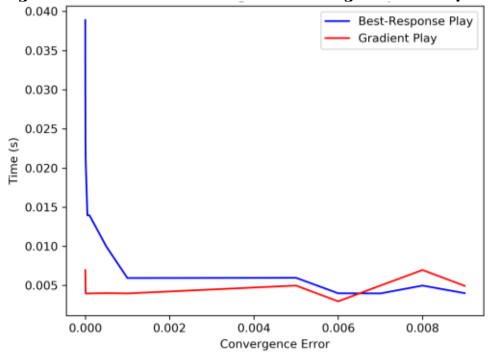


Figure A9: Iterations Vs Initial Conditions for Example2

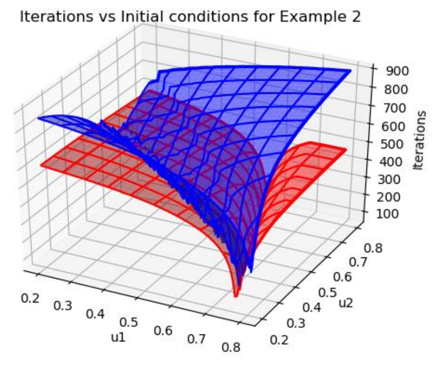


Figure A10: Time vs Initial Conditions for Example2

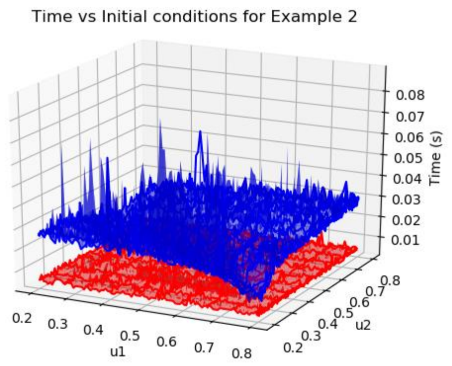


Figure A11: Convergence for u Vs. Learning rate for Example3

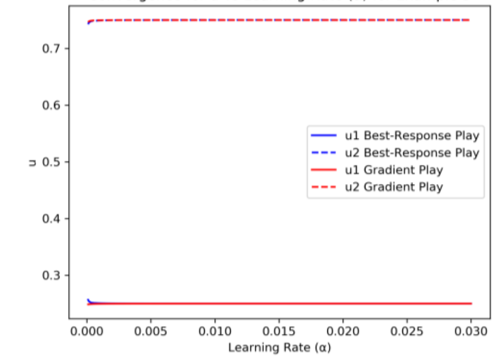


Figure A12: Convergence for u Vs. Convergence Error for Example3

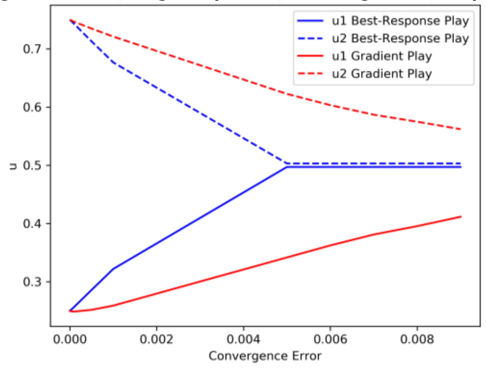


Figure A13: Convergence for u Vs. Iteration for Example3

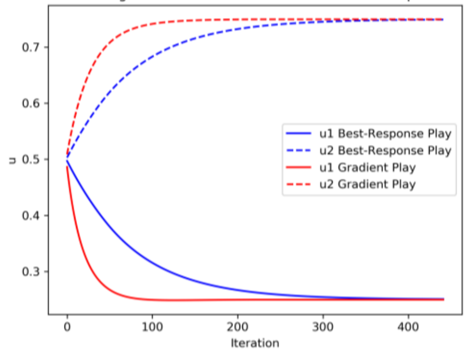


Figure A14: Convergence for J Vs. Iteration for Example3

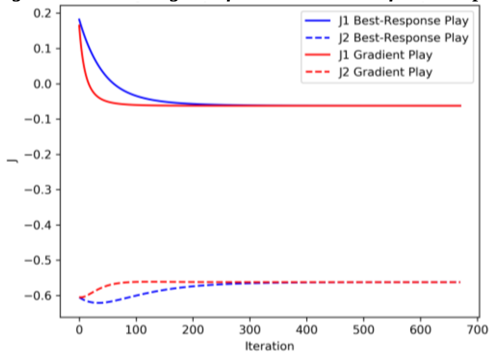


Figure A15: Iterations Vs. Learning Rate for Example3

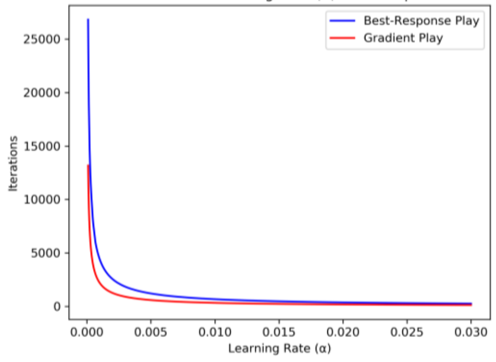


Figure A16: Iterations Vs. Convergence Error for Example3

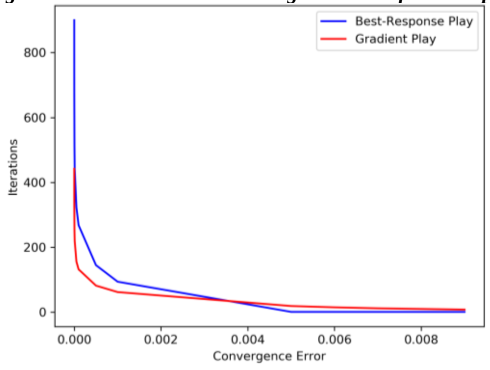


Figure A17: Execution Time Vs. Learning Rate for Example3

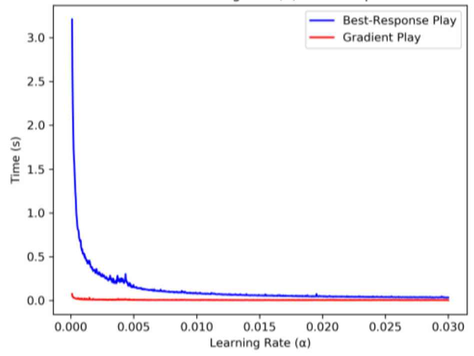


Figure A18: Execution Time Vs. Convergence Error for Example3

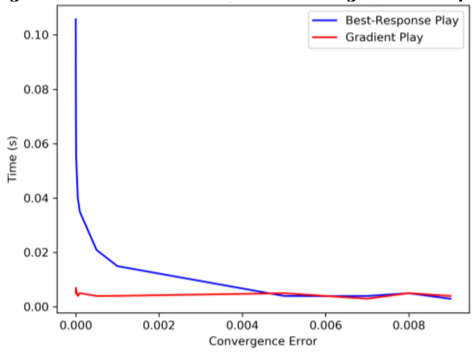
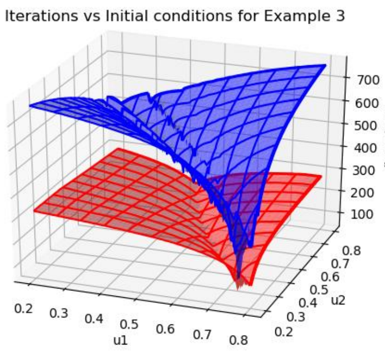


Figure A19: Iterations Vs Initial Conditions for Example3 *Figure A20: Time vs Initial Conditions for Example3*



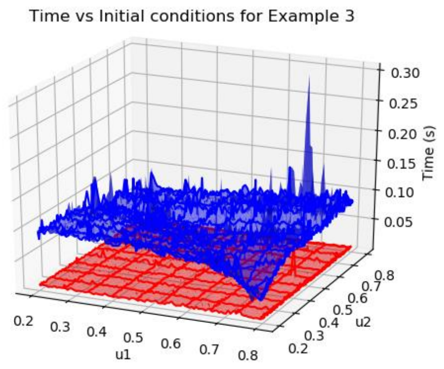


Figure A21: Convergence for u Vs. Learning rate for Example4

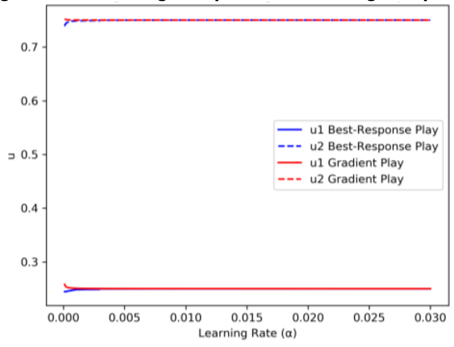


Figure A22: Convergence for u Vs. Convergence Error for Example4

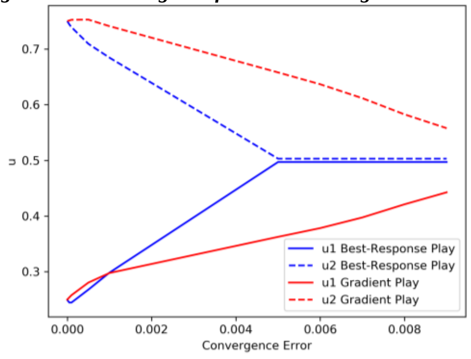


Figure A23: Convergence for u Vs. Iteration for Example4

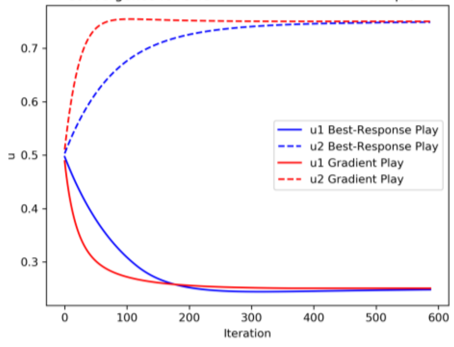


Figure A24: Convergence for J Vs. Iteration for Example4

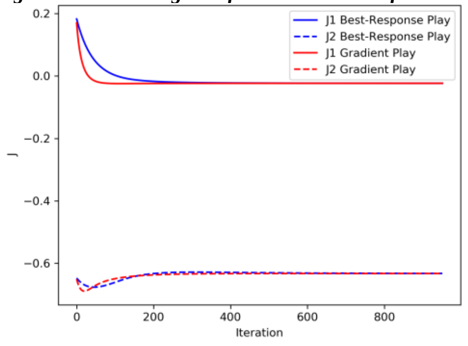


Figure A25: Iterations Vs. Learning Rate for Example4

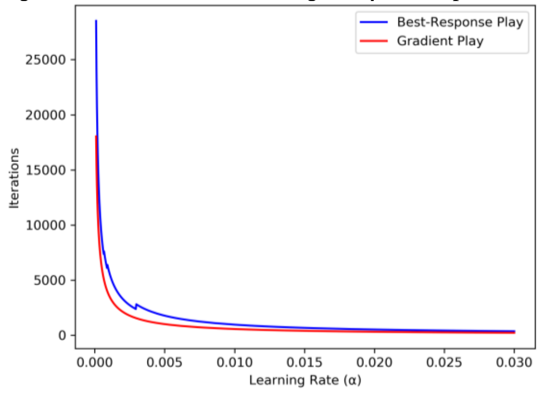


Figure A26: Iterations Vs. Convergence Error for Example4

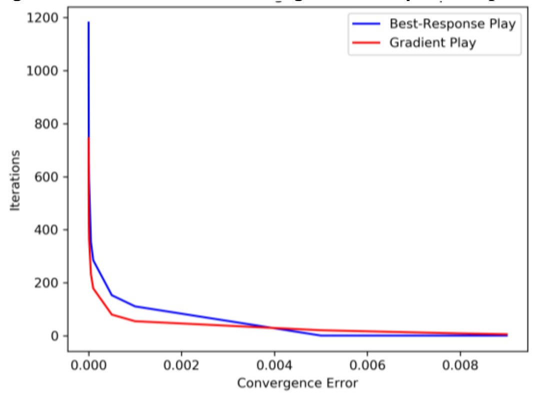


Figure A27: Execution Time Vs. Learning Rate for Example4

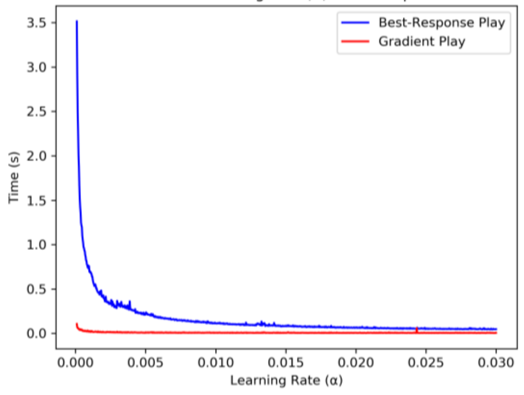


Figure A28: Execution Time Vs. Convergence Error for Example4

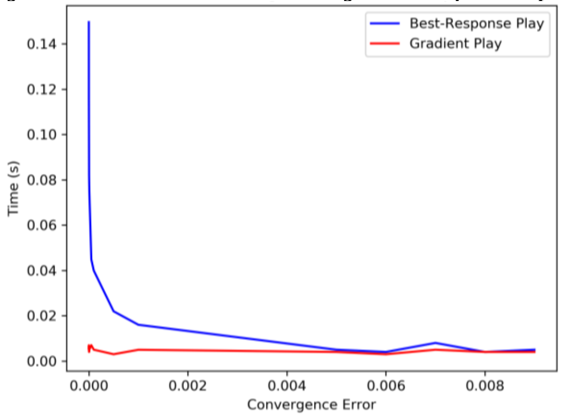
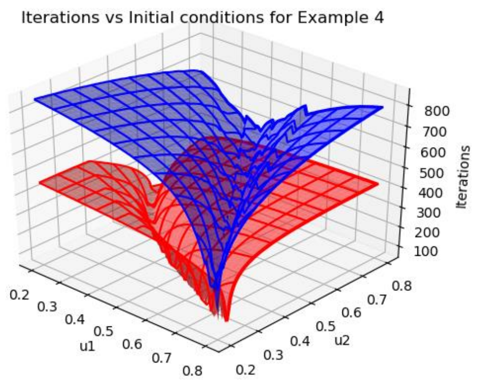
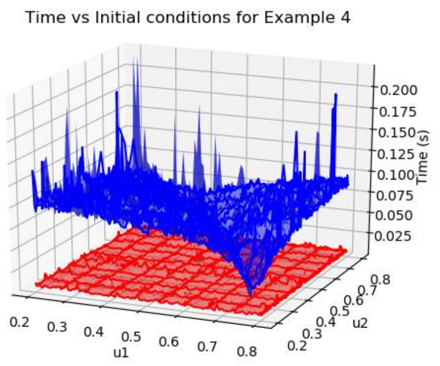


Figure A29: Iterations Vs Initial Conditions for Example4

Figure A30: Time vs Initial Conditions for Example4





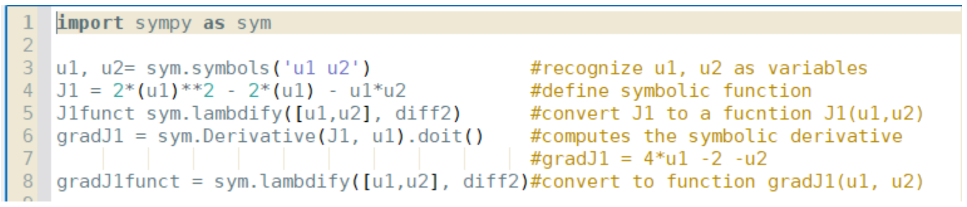
Appendix B: Code Reference

All Code available at:
https://drive.google.com/drive/u/0/folders/1nZ-mKiygOhbBG0t3Y67-CX4zOg9xsYMJ ,
along with a video presentation of the project

Email:
frankedward.nemeth@mail.utoronto.ca if there is any issue

Appendix C: Additional References

Figure C1: Sympy Example Code with Comments



```
import sympy as sym

u1, u2= sym.symbols('u1 u2')              #recognize u1, u2 as variables
J1 = 2*(u1)**2 - 2*(u1) - u1*u2           #define symbolic function
J1funct sym.lambdify([u1,u2], diff2)      #convert J1 to a fucntion J1(u1,u2)
gradJ1 = sym.Derivative(J1, u1).doit()    #computes the symbolic derivative
                                          #gradJ1 = 4*u1 -2 -u2
gradJ1funct = sym.lambdify([u1,u2], diff2)#convert to function gradJ1(u1, u2)
```