Methods

Simulations used

Different simulated repeated games were constructed using the programming language of Python. These games were constructed in an attempt to find connections between the different variables used in the repeated games.

Python Implemenation

To construct these games, different Python libraries were taken advantage of. Since this project will be dealing with a large quantity of numerical data, the Numpy 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, small change 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 it 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 library was discovered, and was used to create symbolic equations for the various cost functions used, and later convert them into Python functions to calculate at specific values. This method also consequently allows for new cost functions to be easily defined and tested, making the simulations robust and scalable. <<<<Figure XXX shows an example of the use of the function, with comments to elaborate on the steps>>>

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 \*\*\*\*gelensburg, refrence with number). Furthermore, Limiting the game to two players helps with the programming. Most other parameters will be variables, as the goal of this project is to analyse the hyperparameters and how they affect each of the iterative algorithms.

\*\*\*also say that certain number of iterations necessary

\*\*\*edge case accounting for not going out of the range of error values

Pseudocode

Figures \*\*\*\* \*\*\*\* \*\*\*\* show pseudocode representations of the BR Play and Gradient Play. Each function makes use of the same variables:

* i1 : initial condition P1 (float)
* i2 : initial condition P2 (float)
* u1 : matrix of actions for P1 (array of floats)
* u2 : matrix of actions for P2 (array of floats)
* J1 : cost function for P1 (Python function)
* J2 : cost function for P2 (Python function)
* alpha : learning rate (float)
* e : convergence error (float)

the convergence error is defined as , 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 \*\*\*talk about with equation, ideally the second portion should be zero, but reaching zero might not be practical, this just defines an error bound on it\*\*\*

Figures \*\*\*\*\* \*\*\*\*\* show the helper functions for each. Specifically the BR function take advantage of min\*\*\*\*\* Grad is a symbolic fnction, so the derivative is calculated once using sympy and then the iterations ensue

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

Testing procedure

The following table \*\*\*\* summarizes the different examples that were chosen to be tested

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Label | Cost Functions, | Solution, | Set | Initial conditions, | Default Learning Rate | Default Convergence Error |
| Example1 |  |  |  |  |  |  |
| Example2 |  |  |  |  |  |  |
| Example3 |  |  |  |  |  |  |
| Example4 |  |  |  |  |  |  |

Example1 was taken from \*\*\*\*\*<<insert name and citation>>\*\*\*\*, 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 are games constructed with different polynomial orders on each (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. This was done in an attempt to make the most distinguishing factor when comparing the functions the order of the dominant terms. The set space was an arbitrary range that included the NE solution. Initial conditions were also an arbitrary point withing the set space.

Where possible, the default values for Learning Rate and Convergence Error were 0.01 and 0.00001 respectively. These values were chosen after some experimentation with the functions. Typical Learning Rates are between \*\*\*CITATION\*\*\*. Values were chosen as analysis of their convergent properties \*\*\*mention results briefly. The error should be smaller than the alpha. This causes the dunction to numerically work \*\*\*\*etc