Teaching Nash Equilibrium with Python

Allison Oldham Luedtke

Department of Economics, St. Olaf College, Northfield, Minnesota, United States

Dr. Luedtke can be reached by phone at (540) 905-2622 or by email at luedtk2@stolaf.edu.

Acknowledgments: I want to express my appreciation to my first ever Game Theory students for being so willing to try something new. Additionally, I am very grateful to Sarah Jacobson for the opportunity to contribute to this special issue, and for her comments that helped shape this paper. I am also grateful to Erin Cottle Hunt and Melissa Spencer for their comments, which significantly improved this paper's clarity and focus.

Teaching Nash Equilibrium with Python

This paper describes an assignment in an undergraduate Game Theory course in which students work together in class to develop a computer algorithm to identify Nash equilibria. This assignment builds basic computer science skills while applying game theory knowledge to real world situations. Students work as a team to delineate the steps and write a program to identify all of the pure strategy Nash equilibria of the game. They then test this program by creating and solving their own game. This assignment represents an efficient way for undergraduate economics students to gain valuable computer science skills without assuming any pre-existing computer science knowledge, without having to take classes outside of the economics major, and without economics faculty having to restructure entire courses or curricula.

Keywords: games in the classroom; game theory; Nash equilibrium; computer science

JEL codes: A22, C70, C63

The assignment described in this paper combines the hands-on experience of playing and solving a game in the classroom with the development of broadly applicable computer programming skills. This assignment can be implemented by instructors with any level of experience with Python, including none. The concept of a Nash equilibrium is one of the most important definitions that students learn in undergraduate Game Theory and economics as a whole. At the same time, the definition of a Nash equilibrium is complex and students often struggle with it. In "Games of Strategy," by Avinash Dixit, Susan Skeath, and David Reiley Jr., a textbook commonly used in undergraduate Game Theory courses, the authors provide the following definition of a Nash equilibrium: "A configuration of strategies (one for each player) such that each player's strategy is best for him, given those of the other players" (Dixit, Skeath, and Reiley 2015, 766). A deep, intuitive understanding that a Nash equilibrium is a set of choices for each player that cannot be improved upon given what other players have

chosen can be helpful for students as they learn more about economics. This paper presents an assignment that uses in-class teamwork and computer programming to help students internalize this definition. Instructors have designed many creative and interactive assignments to help students engage with the concepts of Game Theory. O'Roark and Grant (2018) use comic books (O'Roark and Grant 2018). Alba-Fernández, Brañas-Garza, Jiménez-Jiménez, and Rodero-Cosano (2010) use a beauty contest game (Alba-Fernández et al. 2010). McCannon (2010) uses the bible (McCannon 2010). The exercise described here uses computer programming.

Experiential learning and classroom activities can be used to solidify students' understanding of key concepts throughout economics (Henderson 2018; Truscott, Rustogi, and Young 2000). These activities range from service learning (McGoldrick 1998), to writing as macroeconomic journalists (Aguilar and Soques 2013), to competing to create the best Fiscal Policy (Aguilar and Soques 2015). The assignment presented in this paper asks students to act as a programming team to solve a computational problem. This type of work takes place every day at social media companies, research labs, financial consulting firms, and a number of other jobs which economics majors may eventually pursue. This assignment provides students with the opportunity to experience a type of work that their eventual career may feature.

Experience with computer science is not only helpful for students because of the increasing digitalization of jobs, but also because of the skills that it develops. Students with degrees in economics are already sought after because of their problem-solving skills. Computer programming requires these same skills as well as communication and abstraction. The ability to break down a problem into lines of code for a computer to process is an excellent complement to the existing economics major skill set. These problem-solving, communication, and abstraction skills are particularly honed by

working with a definition as complex as that of a Nash equilibrium. Students break down the process of identifying a Nash equilibrium into its foundational components and must explain to a computer what they mean.

Whether economics undergraduates become financial consultants, political advocates, or go on to graduate school, a basic knowledge of computer science is helpful for success (Takemura 2017). These skills are increasingly important in economics, as computation, technology, and mathematics become a greater part of the discipline (Marshall and Underwood 2020). But for most economics majors, it is not feasible to add a computer science major, both because of their already full academic plates and because demand for computer science classes is increasing (Soper 2014). Nor is it practical for most economics departments to devote faculty resources to a "Computer Science for Economics" elective. As such, if faculty want to ensure that economics majors are exposed to computer programming, this material needs to be incorporated into our existing curriculum. This paper describes an assignment that can be used in a variety of undergraduate or Masters level microeconomic courses to cement students' understanding of the concept of a Nash equilibrium. This assignment could be used in undergraduate introductory and intermediate microeconomic theory courses, and game theory courses, as well as in Masters and MBA courses on microeconomics and game theory.

In the assignment, students first work as a group to identify the pure strategy Nash equilibrium in a simultaneous discrete-choice game, often referred to as a matrix game. Following this, students break down the process into discrete steps, designing an algorithm that will eventually become a computer program. Finally, students test the algorithm using a game which they create themselves. The majority of students in the class in which I implemented this assignment had no prior computer science experience,

but after working through the game and algorithm as a group in class, were able to write a program in Python to correctly identify the Nash equilibria in any matrix game with two players.

To begin the assignment, the instructor presents the class with a standard matrix game with two players. In this particular game, the two players are planning a Thanksgiving meal. The first player is bringing the main dish and the second player is bringing a side dish. The available strategies and payoffs of this game are described in Table 1. A Nash equilibrium in this game is a choice of main dish and side dish such that neither player wishes to bring a different dish, given the other player's choice. The class proceeds to check each cell of the matrix for the characteristics of a Nash equilibrium. As they move through this process, the instructor asks students to identify the step-by-step process used and takes notes on the board as the students name the steps. The instructor and class then use these steps to draft an algorithm (in words) for identifying a Nash equilibrium using cell-by-cell inspection. The students then use this *pseudocode* to write a computer program in the programming language, Python, in order to find any Nash equilibria in pure strategies in a two-player matrix game. The instructor and the class work together to outline the algorithm and then students independently implement their own program. To test their algorithm, students design their own matrix game. The only requirement for their game is that it has two players and each player has enough choices so as to make the game tedious – but possible – to solve by hand. The assignment introduces as little new terminology as possible so as to require as little computer science overhead as possible. Students turn in a description of their game, their Python code, the output that the code creates, and a discussion of the results.

This assignment can be effectively implemented by instructors with little to no experience with Python. The use of pseudocode to describe the steps of the algorithm is more important than the specific programming language and it is a concept that instructors can introduce without any specific programming experience. The Python code needed to implement the assignment can be found on Dr. Luedtke's website, www.aoluedtke.com. Additionally, Figure 1 provides a fully annotated version of the Python code that directly matches each section of the code with the pseudocode described in the "Assignment" Section of this paper.

The assignment is designed to require as little time introducing computer science topics as possible. No additional Python packages are required, so there is no complicated header needed at the top of students' code. The class in which this assignment was first introduced was able to complete this assignment with three lecture hours devoted to background and outlining the algorithms. It could be done with less time in class if the instructor wants to leave more for the students to complete independently. Additionally, this assignment is easier to grade than a traditional Game Theory problem set. Grading is discussed in more detail in later sections. Resources to assist instructors with the implementation of this assignment are discussed in Section 3, "Tips and Resources for Instructors."

Not only did students in the undergraduate Game Theory class in which this assignment was implemented perform well on this assignment – all but two students received a B or better – they also felt that it improved their resumes and job prospects. Multiple students reported discussing this project with potential employers in interviews for jobs and internships. Students also seemed to legitimately enjoy the assignment. At the end of the semester, students in this course had several options for their final project, one of which was an extension of the assignment described here. The final project

assignment required them to write a new computer program that built on what they had produced previously. The vast majority of students chose to complete this option for their final project. One student even mentioned it as her favorite aspect of the class on my teaching evaluation for the course.

To best prepare students for life after college – whatever form that may take – we should expose them to computer programming at some point throughout their economics major. This paper describes an assignment that accomplishes this by having students work together and rely on one another without requiring any prior computer science experience on the students' part and with little extra work for instructors. Assignments of this type will become increasingly necessary as departments are doing more with less.

The next section describes the assignment in detail, including the game the students solve as a group, the process for creating an algorithm from the steps students suggest, and an outline of the Python code that identifies the Nash equilibria in a two-player matrix game. The following section describes some tips and resources for instructors, including potential Python compilers that allow students to begin writing and running code quickly. The final section concludes.

ASSIGNMENT

To begin the assignment, the instructor describes to the class a simultaneous-choice game with two players. Such games can be represented as a table, or matrix, that describes the payoffs to each player associated with each of their possible choices. The two players in this game are organizing a Thanksgiving dinner and choosing a main course and a side dish. Player 1 is bringing the main course and Player 2 is bringing a side dish. The payoffs to each player for each combination of strategies are enumerated in Table 1. The instructor and the class then check each cell of the matrix for the

characteristics of a Nash equilibrium, by asking questions like, “Is Player 1 happy with his choice to bring Roast Beef if Player 2 brings Mashed Potatoes?” This procedure for identifying any Nash equilibrium of the game is often referred to as “cell-by-cell inspection” (Dixit, Skeath, and Reiley 2015, 95). As the class moves through this process, the instructor writes each step of the process on the board, e.g., “Starting with the first row, one column at a time, check each cell.” After the class solves the Thanksgiving game, they use these steps to outline an algorithm that could be written in computer code. They do this by first outlining the steps of the algorithm in pseudocode. Table 2 contains examples of student descriptions and the corresponding pseudocode. The purpose of pseudocode is to allow the writer to outline code without the specific syntax of a programming language. As such, the examples of pseudocode presented in Table 2 are merely suggestions and instructors can adapt it to their classes.

A pure strategy Nash equilibrium in this type of game consists of a choice by each player such that, given one player’s choice, the other player cannot improve his payoff by changing his choice. Consider the first cell, with payoffs [3, 1]. Given that Player 1 is choosing to bring Roast Beef, Player 2 can improve his payoff from 1 to 3 by switching to bringing Macaroni & Cheese rather than Mashed Potatoes. As such, the first cell is not a Nash equilibrium. The middle cell in the third row, with payoffs [5, 4], represents the only pure strategy Nash equilibrium in this game: Player 1 brings Meat Loaf and Player 2 brings Macaroni and Cheese. There are several ways to identify the pure strategy Nash equilibria in a matrix game, including cell-by-cell inspection, iterated removal of dominated strategies (in certain circumstances), and best-response analysis. Of these, the method that students struggle with the most is cell-by-cell inspection. This method requires some of the deepest knowledge of the Nash equilibrium definition and, at the same time, is well suited for computers. To find all

Nash equilibria using cell-by-cell inspection, you check each cell of the matrix to see if it meets the definition of a Nash equilibrium. The algorithm that the students write visits each cell of their game matrix and checks to see if either player can be made better off, given the other player's choice. If neither player can be made better off, the algorithm records that cell as an equilibrium. If either player can be made better off, the algorithm records that cell as not an equilibrium. Every cell of the matrix is visited and labelled as "equilibrium" or "not." In order to achieve this, students must write an algorithm that (1) visits each cell, (2) queries the payoffs for each player, (3) compares them to the relevant other payoffs, and (4) makes the correct determination about the equilibrium status of that cell. The pseudocode for this is as follows:

```
for each row:
    for each column:
        check if row player can be made better off
        check if column player can be made better off
        if neither can be:
            label as equilibrium
        if any can be:
            label as not an equilibrium
```

Figure 1 provides a fully annotated version of the Python code that implements this assignment. Each line of the pseudocode above is matched to the specific Python language that accomplishes that step.

The Python code to find all pure-strategy Nash equilibria consists of three main components: reading in a matrix of payoffs, checking each cell for the characteristics of a Nash equilibrium, and printing out the cells that are equilibria. Taken together, these components force students to break down their real-world scenario into the foundational elements of a game, payoffs, and equilibrium. The class and the instructor work collectively to come up with pseudocode, which the instructor writes on the board, and then he or she types and explains the actual Python code on the classroom computer display. Because the students had no prior experience with programming, with the

correct code displayed on the projection screen, the translation onto their own computers was sufficiently difficult to present a challenge, but students felt comfortable enough to try and fail a few times before they got it entirely correct.

Instructors who wish to introduce as little Python terminology as possible to complete this assignment may choose to use Python's innate matrix structure, rather than to import a package like Numpy, which might have more elegant and complex matrix operations. While the innate matrix structure requires the use of a cumbersome number of brackets and parentheses, it saves at least an extra day of explaining how to import Python packages, which would not add anything to the students' understanding of Game Theory. If, however, instructors wanted to include the use of packages in this assignment, Numpy would represent a good introductory example for students. In this paper, I provide code for using Python's innate matrix structure and no imported packages. The code to input the game matrix for the game represented in Table 1 can be found at the top of Figure 1.

Each pair of numbers represents the payoffs to Player 1 and Player 2, respectively. The instructor can provide students with this code to input the matrix for the game they solve as a class. When students write their own matrix game to test the code, students can then use this code as a template for inputting their own game matrix independently.

The final task of the computer program is to print in an output window any cells that are Nash equilibria. I chose to make the output process a separate task of the algorithm, rather than to have it print the Nash equilibria as it found them. This serves two purposes: (1) it makes the algorithm more modular, so that students could use portions of it for other projects; and (2) it allows each portion of the algorithm to be introduced separately. This method requires the use of a second matrix, the same size

as the game matrix, that contains the equilibrium "label" for each cell. As the algorithm determines the equilibrium status of each cell, it stores the appropriate label in the label matrix. When the equilibrium-identification portion of the algorithm is completed, the label matrix contains the equilibrium status of each cell. To print out any Nash equilibria, the algorithm must now parse the label matrix and print out any cells of the game matrix that correspond to a label of "equilibrium." Every label starts out as a "0." If it is a Nash equilibrium, the label is switched to a "1," and if it is not a Nash equilibrium, the label is switched to a "2." This allowed students to practice the critical computational skill of converting qualitative information ("is a Nash equilibrium") into numerical data ("1"). The pseudocode for printing out any Nash equilibrium cells is as follows:

```
for each row:
        for each column:
                if label(row,col) == 2:
                        print cell(row,col)
```

To test the code that they produce, students create their own two-player simultaneous-choice game and then run the code that the class builds on this game. I encouraged students to be creative in the design of their games because, when they delineate each step of the solution process using their own terms instead of mine or the textbook's, they gain a deeper grasp of the economic meaning of them. To complete the assignment, students turn in the following elements: (1) a description of their game, (2) their Python code, (3) the output when their code is run, and (4) a discussion of the solution of the game. It may be the case that the game that students create does not have any Nash equilibria in pure strategies. In the description of the game, students specify who the players are, what choices are available to them, and what the payoffs for each possible outcome are. In this description, they also include the matrix that represents their game. Students turn in their Python code as a Python file so that the

instructor can run it to ensure that it works correctly. In their discussion of the solution found by their code, students explain whether any of the Nash equilibria are socially optimal, whether they make sense in the context that the students originally envisioned, what would need to change about the game to achieve a different outcome, and any other observations that the students wish to share. Students turn in two files: one is a typed document containing the description of the game, the output of the Python code, and the discussion, and the second is their Python code.

In addition to building teamwork, communication, computation, and abstraction skills, this assignment forces students to consider carefully what constitutes a good example of a simultaneous discrete choice game, and thus what would make a good test question about simultaneous discrete choice games. No matter how many times I tell my students that the best way to prepare for a test is to create their own practice questions, they do not believe it until they are forced to do it. This is one way to accomplish that.

TIPS AND RESOURCES FOR INSTRUCTORS

This assignment can be adapted to fit into most assignment types and grading categories. I used it as one of five homework assignments throughout the semester. Depending on how much you cover together in class and how much you leave to the students to complete independently, in conjunction with the level of programming experience of your students, this assignment can be a small classwork assignment, a homework assignment, a large term project, or even an extra credit assignment. In order to receive a perfect score on this assignment, students had to turn in all components, explain all components of their game, correctly identify any Nash equilibria of their game – both in their description of the game and computationally – and discuss the appropriate outcomes. It is easy for the grader to identify the

components of the game in the description (e.g., “players,” “choices,” “payoffs,” “equilibria”) and the discussion (e.g., “social optima,” “potential changes”). The only numerical component that requires checking is the correct identification of the Nash equilibria. This can be done very quickly by copying their game matrix and label matrix into your code and running it. If their list of equilibria matches yours, they correctly identified them.

To easily run Python code on a classroom computer and on your students’ computers without having to install many new software packages, I recommend making use of online Python editors and compilers, such as Jupyter Notebook or Google Colaboratory. Alternatively, if you would prefer to use an application that does not require an internet connection, I recommend the Python editor and compiler, Thonny. It is easy for students to install and use without complicated installation procedures, and works well for small, pedagogical applications such as this one.

The actual Python code for the example represented in Table 1 is replicated in its entirety in Figure 1. Additionally, Figure 1 matches each line of the pseudocode presented in the “Assignment” Section with the lines of Python code that implement them.

The assignment was designed to allow for expansions and further applications later in the semester, if the students expressed interest. Students were given a slate of choices for their final project in the class and one of the options was to complete another programming assignment. A majority of students in the class chose this option. Specifically, I asked students to modify their code to randomly generate a game matrix and use their existing code to find any Nash equilibria. There are many possible expansions to this assignment and I encourage you to come up with your own.

CONCLUSION

This paper presents an assignment for undergraduate Game Theory students. In the assignment, students work together as a class to identify Nash equilibria in a game and outline an algorithm for replicating this process in any similar matrix game. They then write a computer program in Python that finds all of the pure strategy Nash equilibria of their game and test this code by creating their own game based on their lives. The assignment is designed to take only three to six hours of instruction, so that it can be added to most undergraduate Game Theory courses without faculty needing to dramatically restructure their courses. Experience with computer programming is swiftly becoming necessary for most careers that an economics graduate would pursue. This assignment efficiently provides students with workable knowledge of programming in Python.

REFERENCES:

Aguilar, Mike, and Daniel Soques. 2013. "MacroJournal—Turning Students into Practitioners." *The Journal of Economic Education* 44 (3): 230–37. https://doi.org/10.1080/00220485.2013.795453.

———. 2015. "Fiscal Challenge: An Experiential Exercise in Policy Making." *The Journal of Economic Education* 46 (3): 285–99. https://doi.org/10.1080/00220485.2015.1040179.

Alba-Fernández, Virtudes, Pablo Brañas-Garza, Francisca Jiménez-Jiménez, and Javier Rodero-Cosano. 2010. "Teaching Nash Equilibrium and Dominance: A Classroom Experiment on the Beauty Contest." *The Journal of Economic Education* 37 (3): 305–22. https://doi.org/10.3200/JECE.37.3.305-322.

Dixit, Avinash K., Susan Skeath, and David H. Reiley. 2015. *Games of Strategy*. Fourth edition. New York: W.W. Norton & Company.

Henderson, Amy. 2018. "Leveraging the Power of Experiential Learning to Achieve Higher-Order Proficiencies." *The Journal of Economic Education* 49 (1): 59–71. https://doi.org/10.1080/00220485.2017.1397576.

Marshall, Emily C., and Anthony Underwood. 2020. "Is Economics STEM? Trends in the Discipline from 1997 to 2018." *The Journal of Economic Education* 51 (2): 167–74. https://doi.org/10.1080/00220485.2020.1731387.

McCannon, Bryan C. 2010. "Using Game Theory and the Bible to Build Critical Thinking Skills." *The Journal of Economic Education* 38 (2): 160–64. https://doi.org/10.3200/JECE.38.2.160-164.

McGoldrick, Kimmarie. 1998. "Service-Learning in Economics: A Detailed Application." *The Journal of Economic Education* 29 (4): 365–76. https://doi.org/10.1080/00220489809595929.

O'Roark, Brian, and William Grant. 2018. "Games Superheroes Play: Teaching Game Theory with Comic Book Favorites." *The Journal of Economic Education* 49 (2): 180–93. https://doi.org/10.1080/00220485.2018.1438861.

Soper, Taylor. 2014. "Analysis: The Exploding Demand for Computer Science Education, and Why America Needs to Keep Up." GeekWire. June 6, 2014. https://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/.

Takemura, Alison. 2017. "Two Sciences Tie the Knot." MIT News. 2017. http://news.mit.edu/2017/mit-creates-new-major-computer-science-economics-data-science-0904.

Truscott, Michael H., Hemant Rustogi, and Corinne B. Young. 2000. "Enhancing the Macroeconomics Course: An Experiential Learning Approach." *The Journal of Economic Education* 31 (1): 60–65. https://doi.org/10.1080/00220480009596762.

TABLES:

Table 1. In-Class Game Matrix.

		Player 2 (2nd Friend)		
		Mashed Potatoes	Macaroni & Cheese	Broccoli
Player 1 (1st Friend)	Roast Beef	3, 1	2, 3	10, 2
	Turkey	4, 5	3, 0	6, 4
	Meatloaf	2, 2	5, 4	12, 3
	Lasagna	5, 6	4, 5	9, 7

Table 2. Student Descriptions and Pseudocode.

Student Descriptions	**Pseudocode**
"Check each row" "Check each column"	`for each row:` `  for each column:`
"Can Player 1 be made better off (given Player 2's choice)?"	`check if row player can be made better off`
"Can Player 2 be made better off (given Player 1's choice)?"	`check if column player can be made better off`
"If neither has a better option (given the other's choice), this is a Nash equilibrium."	`if neither can be:` `label as equilibrium`
"If either has a better option (given the other's choice), this is not a Nash equilibrium."	`if any can be:` `label as not an equilibrium`

FIGURES:

Figure 1. Annotated Code with Pseudocode.