Czech Technical University in Prague
Faculty of Electrical Engineering
Technická 1902/2, 166 27 Prague 6 - Dejvice-Prague 6

BACHELOR THESIS

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

2021 Stanislav Kubiš

Czech Technical University in Prague
Faculty of Electrical Engineering

BACHELOR THESIS

Focus: Open Informatics – Informatics and Computer Science

Topic: Games with Piecewise Affine Utility Functions

Author: Stanislav Kubiš
Supervisor: doc. Ing. Tomáš Kroupa, Ph.D.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kubiš Stanislav**	Personal ID number:	**474728**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Games with Piecewise Affine Utility Functions

Bachelor’s thesis title in Czech:

Hry s po částech afinními užitkovými funkcemi

Guidelines:

1. Learn the basics of zero-sum strategic games. In particular, pay attention to the algorithm for computation of equilibrium based on linear programming [1].
2. The goal of this work is to experimentally verify whether infinite two-player zero-sum games with payoff functions in the form of piecewise affine functions [3] have finite equilibria. This property is known to hold for polynomial games [2].
3. The experimental work is based on the generation of random triangulations together with piecewise affine functions arising from them. The next step is to approximate such functions over a finite grid and determine the equilibrium of the respective finite game. Use experiments to assess the convergence of such a solution.

Bibliography / sources:

[1] Y. Shoham and K. Leyton-Brown. Multiagent systems: Algorithmic, game-theoretic, and logical foundations. Cambridge University Press, 2008.
[2] P. Parrilo. Polynomial games and sum of squares optimization. In Decision and Control, 2006 45th IEEE Conference on, pages 2855–2860, 2006.
[3] T. Kroupa and O. Majer. Optimal strategic reasoning with McNaughton functions. International Journal of Approximate Reasoning, 55(6):1458–1468, 2014.

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Tomáš Kroupa, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **05.01.2021**

Assignment valid until: **30.09.2021**

doc. Ing. Tomáš Kroupa, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt   Student’s signature

Sincere thank you belongs to the supervisor of my thesis, who was always there to help me with anything from explaining theoretical content to working with me on building ideas how to approach the thesis.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ………………………. ……………………………

Podpis autora práce

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date …………………………… ……………………………

Signature

Abstract

This bachelor thesis aims to experimentally verify whether infinite two-player zero-sum games [9, p.89] with payoff functions in the form of piecewise affine functions have finite equilibria. We define piecewise affine functions over a domain split by a set of continuous affine functions. This domain for the experiment is limitary to a square $[0,1] \times [0,1]$, this is due to zero-sum games using the interval of $[0,1]$ to represent possible strategies. To verify equilibrium finiteness assumptions, we built an algorithm creating a new program. Python programming language was chosen for the program creation. The algorithm's experimental function is based on random triangulations with piecewise affine functions arising from them. The algorithm's next step is to approximate such functions over a finite grid and determine the respective finite game's equilibrium. An iterative approach was used in experiments to create larger zero-sum games to assess such solutions' convergence in single steps visually. An iterative approach was used to generate larger zero-sum games in the experiments so that it was possible to visually assess the convergence of the behavior of the program's partial outputs in individual steps. To fully understand the program's functioning, we need to master zero-sum strategic games' basics theory. Next problematics, which was necessary to analyze in detail for program creation, was an algorithm for equilibria computing (based on linear programming). The last area addressed during this bachelor thesis was the problems arising from our games' infinite strategy space.

Keywords

Game Theory, Piecewise Affine Games, Zero-Sum Games, Two-player Games, Nash Equilibrium, Linear Programming

Abstrakt

Cílem této bakalářské práce je experimentálně ověřit, zda nekonečná hra o dvou hráčích s nulovým součtem [9, p.89] a s výplatními funkcemi, které jsou reprezentovány ve formě po částech afinních funkcí, má konečnou rovnováhu. Po částech afinní funkce definujeme přes doménu rozdělenou na sadu spojitých afinních funkcí. Tato doména je v experimentu ohraničená čtvercem $[0,1] \times [0,1]$ z toho důvodu, že hry s nulovým součtem využívají interval $[0,1]$ pro reprezentaci možných strategií. K tomu, abychom ověřili předpoklad ohledně konečnosti rovnováhy, byl navržen algoritmus, ze kterého vychází nově vytvořený program. Pro tvorbu programu byl zvolen programovací jazyk Python. Experimentální funkce algoritmu je založena na generovaní náhodných triangulací společně s jejich vznikajícími po částech afinními funkcemi. Dalším krokem algoritmu je aproximace těchto funkcí nad konečnou mřížku a určení rovnováhy dané konečné hry. K vytváření větších her s nulovým součtem byl v experimentech použit iterační postup, aby bylo možné v jednotlivých krocích vizuálně posoudit konvergenci chování dílčích výstupů programu. Abychom plně rozuměli funkčnosti programu, musíme ovládat základy teorie strategických her s nulovým součtem. Další problematikou, kterou bylo nutné detailně analyzovat pro tvorbu programu, byl algoritmus pro počítání rovnováhy (založený na principu lineárního programování). Poslední z oblastí, kterou se bylo nutné v průběhu bakalářské práce zabývat, byly problémy vznikající z nekonečného strategického prostoru naší hry.

Klíčová slova

Teorie her, Po částech affinní hry, Hry s nulovým součtem, Hry o dvou hráčích, Nashova rovnováha, Lineární programování

Table of Contents

1 Introduction

1.1 History

Recently we encountered a vast space of mathematical and algorithmic tools, which come from mathematical optimization and numerical methodology to the quickly evolving field of artificial intelligence. This would not have been possible without a shared work brought to us by John von Neumann and Oskar Morgenstern. They introduced us to a game-theoretical methodology [2], which is now being solved for some highly complex games. The possibility of the game-theory area having such a big impact nowadays is attributed to a rapidly improving infrastructure, which provides almost exponentially improving computational power[1] needed for solving a vast range of complex problems. The rise of game theory paved the way for it to intertwine with other science and economics fields. As the rise of technology is relatively recent and Neumann-Morgenstern popularized game-theory only in 1944, another important discovery was attributed for problems to be solved and expanded. The discovery mentioned is the one of Nash equilibrium by John Forbes Nash, Jr. We describe its properties more in section 3.

1.2 Strategy space

It is important to note that searching for Nash equilibria in the space of large or even infinite game is infamously difficult to solve [3]. To minimize this, it helps to identify the problem’s key components once it is formulated. The construction of a game involves a realization of who participates in a game. In our case, the game involves agents (called players in the thesis) whose objectives may discord. Specifically, we have two players who evaluate their strategies depending on utility function values keeping in mind that a number of strategies may be infinite. However, many definitions in use are based on games with only finitely many actions, since theory about them has been expanded to a great extent [4] even though many naturally appearing games where strategy sets are uncountable. Still, many obstacles arise by switching to games with infinite action spaces. This results in the game’s mixed strategies unable to be represented as finite-dimensional probability vectors. Instead, they become probability measures supported by possibly infinite sets.

1.3 Solution

Generalization of the Nash theorem, Glicksberg’s theorem [6], guarantees Nash equilibria in continuous games. When finding Nash equilibrium in our Zero-Sum games, we use a method of solving with linear programming. The algorithm at use in the experiment is based on a dual form of linear programming introduced by Koller, Megiddo, and von Stengel [24], more described in section 2 because there is a need to search for standard Nash equilibrium with a single solution of linear programming. Its extended option form, which searched for a proper normal form Nash equilibrium with iterative solving [26] cannot be used since a different type of iterative approach is used as our iteration consists of changing the game set to create a new game and not iterating over the same. This helps us to get the solution without any computational barriers quickly. The algorithm used to verify equilibria's finiteness has been only recently discovered for polynomial games by Dresher, Karlin, and Shapley [25]. Polynomial games are part

[1] http://www.singularity.com/charts/page70.html

of the family called separable games. Every utility function is a finite sum of products, where a product component is a function of actions for each player separately. It is known that separable games also include the zero-sum games [8]. Putting it together, we arrive at our case of zero-sum games with piecewise-affine functions representing our utility functions. Ultimately, we want to check whether these similar characteristics will yield a finite Nash equilibrium as expected. Before explaining the algorithm, we explain key components of our game.

1.4 Algorithm overview

We have mentioned our algorithm is based around random triangulations together with piecewise affine functions arising from them. Edges of these triangles are used to create new points. New points are created by checking vertical, and a horizontal straight line passing through an edge and see whether it intersects with any line segment from the triangulation. New points and the original edges create a grid to approximate our functions and calculate equilibrium for the respective finite game. Before calculation, all points were assigned a value of interpolated functions, called heights throughout the thesis. Iterations of the algorithm, together with detailed methods of how finite equilibrium is checked, are described in section 5.

2 Strategic games

In our experiment, we encounter many terms that need to be understood to solve games with piecewise-affine utility functions. First, we introduce types of games (finite and continuous) and their properties. One of these properties describes players playing the game. In our case, our game consists of two self-interested players. This does not necessarily describe that they want to cause harm to one another or that they only care about themselves. Instead, it means that each player has his description of which states of the world he likes. [9, p.47]

2.1 Finite games

Let us have a strategic game G = (N, A, u), where

- N is a finite set of n;
- $A = A_1 \times \ldots \times A_n$, where A_i is a finite set of actions available to player $i \in N$. Each vector $a = (a_1 \ldots a_n) \in A$ is called an action profile;
- $u = (u_1 \ldots u_n)$ where $u_i: A \to R$ is a real-valued utility (or payoff) function for player $i \in N$.

The game is called finite because of the finite size of A.[2] A natural way to represent games with an n-dimensional matrix. Therefore, for our two-player game, it utilizes a two-dimensional matrix. Here, each row corresponds to a possible action that player 1 can choose, each column corresponding to a possible action of player 2. [9, p.56] These players decide how to play based on utility values stored in the matrix. This decision is referred to as a strategy.

2.2 Continuous games

Now we consider strategic games in which players may have infinitely many pure strategies. A strategy is called pure if a player decides to play it with probability one when exposed to his strategy set. We want to include a possibility that the real-valued interval $[0, 1]$ is the pure strategy set. A continuous game is G = (N, S, u), where

- N is a finite set of n players indexed by i;
- $S = S_1 \times \ldots \times S_n$, where S_i is a nonempty compact metric space;
- $u = (u_1 \ldots u_n)$ where $u_i: S \to R$ is a continuous utility (or payoff) function for player $i \in N$.

A compact metric space is a general mathematical structure used to represent infinite sets that can be approximated by large finite sets. Moreover, suppose there is any close bounded subset of a finite-dimensional Euclidean space or any closed bounded interval of the real line. In that case, we are talking about a compact metric space, where the distance between two points x and y is given by $||x - y||_2$. In such metric space, any infinite sequence has a convergent subsequence. As we are trying to verify equilibria finiteness in this game type, it is useful to mention Glickberg's theorem, which guarantees Nash equilibria for every continuous game. [23]

[2] http://gki.informatik.uni-freiburg.de/teaching/ws0607/advanced/recordings/aait-03-strategic-games.pdf

2.3 Utility and strategies

The utility contains important information about a player's decision-making. Utility characteristics and mixed strategies are introduced here.

2.3.1 Utility theory

Utility theory is the leading approach to model player's desires. It aims to describe its preferences across a set of available options. It does so by focusing on understanding how preferences change when a player deals with uncertainty about alternatives it may receive. [9, p.47]

2.3.1.1 Preferences and utility

How do we express preferences? The utility is deeply intertwined with game solving and sometimes hard to grasp. It claims to provide a sensible formal model for reasoning about an agent's happiness in a variety of situations. Why should a game with a player's uncertainty presented in the form of the expected value of utility function, or expected utility, be enough to justify his response and not also depend on other properties of the distribution such as its standard deviation? Theorists researching utility properties ground relative questions in a more basic concept of preferences. The most influential of these theories is the one developed by John von Neumann and Oscar Morgenstern in their book *Theory of Games and Economic Behavior*. [10]. Let O denote a finite set of outcomes, then when we take $o_1, o_2 \in O$ let $o_1 \geq \ o_2$ denote the fact that the agent weakly prefers o_1 compared to o_2. Let $o_1 \sim o_2$ denote that the agent is indifferent between both o_1 and o_1. Lastly, with $o_1 > o_2$ we describe that agents strictly prefers o_1 to o_2. [9, p.49] More rules that also include transitivity rules, but also completeness, and others can be reviewed in [11] or [9, p.50].

2.3.2 Utility function

When we refer to utility functions, as will be done throughout the text, we will be trying
to make a specific assumption that our player's desires how to behave are consistent with utility-theoretic assumptions. Moreover, the utility function is mapping states of the world to real numbers. These values can be interpreted as measurements of a player's level of happiness in the given states. When a player is uncertain, then his utility is defined as the expected utility with respect to the appropriate probability distribution over states in the specific game. [9, p.47,48]

2.3.3 Mixed strategies

We already came across pure strategies, but there is a second type called mixed strategies.
Due to an uncertain game environment, they are encountered more often. For each player, they consist of randomizing over a set of available options according to some probability distribution. Formally this is written as follows:
The set of mixed strategies for player i is $S_i := \Delta(A_i)$, where $\Delta(A_i)$ is the set of all probability distributions with each $p_i \in S_i$ representing one such distribution over A_i.
If $p_i \in S_i$ is a mixed strategy such that $p_i(a_i) = 1$ for some $a_i \in A_i$, then p_i is called a pure strategy [9, p.60]. The strategy describes how a player is trying to achieve the best utility (payoff) from the game. Remember that there exists an expected value that a player will reach.

2.3.4 Expected utility

Expected utility calculates the probability for each strategy in our set of strategies. These are used for measuring average payoff, which is then weighted by each probability. This can be formally defined as follows:

Given a normal-form game (N, A, u), the expected utility u_i for player i of the mixed-strategy profile $s = (s_1 \dots s_n)$ is defined as [9, p.60]

$$u_i(s) = \sum_{a \in A} u_i(a) \prod_{j \in N} s_j(a_j)$$

2.4 Why is finding solutions hard?

Let us look at some properties that make continuous games, especially ones with infinite strategy spaces, hard to solve. Firstly, global minimization and maximization of a polynomial is hard as these optimization problems are typically non-convex and highly nonlinear. Complexity usually has a non-deterministic polynomial-time hardness, even for special cases such as maximizing a quadratic form in binary variables. It is also difficult to find a solution due to its complex content (a combination of heuristics, a need for an insight into the special structure of the game, and also the existence of different types of games with a special equilibrium). We show convex-concave games as an example for special equilibria, where the first player minimizes and the second maximizes to find a saddle-point. Another special example is games of timing. Here, a game starts at a time equal to zero with players' probabilities increasing over time together with a priori probabilities remembered for the past. Another example is games with bell-shaped utility functions or invariants under symmetries[3].

2.4.1 Convex/concave games special equilibria

Convex-concave games are built on a similar principle compared to our games, where one player minimizes, and the other maximizes their payments. It is a type of two-player, zero-sum game of $R^p \times R^q$ with payoff function $f: R^{p+q} \to R$. If we mark one payment as u and other as v, we end up with $f(u, v)$. Lastly, a solution to the game is defined as (u^*, v^*) if

$$f(u^*, v) \leq f(u^*, v^*) \leq f(u, v^*), \forall u, v \in R^p \times R^q$$

At this saddle point, neither player wants to deviate since it would only worsen his standings. The name convex-concave has to do with the function graphs of u and v. Therefore, we need for each v, $f(u, v)$ to be a convex function of u, and for each u, $f(u, v)$ to be a concave function of v. When f is differentiable, our saddle-point will be characterized by a gradient: $\nabla(u^*, v^*) = 0$ [4].

2.4.2 An infinite number of strategies

This subsection will analyze a particular class of infinite strategic games where each player makes his choice from the real unit interval [0,1] (as is the case in our experiment). In the algorithm, each iteration consists of a finite set, as we only add a countable number of strategies, but it inevitably approaches the mentioned infinity. The matrix can hold any number of strategies in the interval [0,1]. This part is described throughout in section 5. We now go more in-depth on why infinite strategies cause difficulty.

Finding Nash equilibrium (defined in section 3) in an infinite or a very large game is known to be difficult to solve. [28] In this subsection, we reinstitute why a message from pioneers Kuhn and Tucker, who wrote in the preface of [29] that finding constructive methods for solving games with infinite strategy spaces "would constitute a considerable contribution" still holds today where the difficulty is based.

[3]https://books.google.cz/books?id=NWIdlT9Z67wC&dq=Global+maximization+of+a+polynomial+is+hard&source=gbs_navlinks_s page 6

Let us have a strategic game with infinite strategy sets. In general, mixed strategies in such a game cannot be represented as finite-dimensional probability vectors. What happens instead is that they become probability measures supported by a possibly infinite set. Hence, a purely mathematical concept that is not a priori computable. Therefore, we need to mention Glicksberg’s theorem [6], which is a generalization of the Nash theorem guaranteeing Nash equilibria in continuous games. This, however, may not be used in computations directly as it is a fully general model of continuous games due to a number of reasons, in particular:

- Glicksbergs’s theorem is a purely existential statement proved in a non-constructive manner, which provides no information about the specific equilibrium strategies.
- Mixed strategies cannot be directly represented in any computer since they may pose complicated probability measures.
- Players may be forced to randomize over an infinitely large set of pure strategies. This happens due to relatively simple games where no single finitely supported mixed strategy equilibrium is present.

After Karlin’s book [29], continuous games' research has been pursued in several directions. Among the most studied continuous games classes were the strategy spaces with real one-dimensional compact intervals, particularly the interval of [0,1]. On the one hand, some carefully crafted solutions to particular examples of these games were developed, such as for games with bell-shaped kernels or games of timing. On the other hand, we may identify efforts to single out entire classes of games where equilibria with finite supports exist and have efficient solution methods for their computation. In the paper [1] Parillo showed that finding an equilibrium of a two-player zero-sum polynomial game over $[-1,1]$ can be obtained by solving a single semidefinite programming problem. Parillo’s result was further expanded to include polynomial games with basic semi-algebraic strategy sets by Laraki and Lasserre [30]. Their method consists of solving a hierarchy of semidefinite relaxations with a possibly high number of decision variables.

Since polynomial games are part of the family called separable games, every utility function is a finite sum of products with a product component being a function of each player's actions separately. We remember that separable games also include the zero-sum games [8] and should therefore yield the same results. The class of separable games allows us to have finite mixed equilibria and algorithms for computing its approximate equilibria of two-player separable games in polynomial time in the game's rank.

2.5 Zero-Sum Games and Their Uses

2.5.1 Description

As mentioned previously, we are experimenting with two-player games. Furthermore, our players' game belongs to the group of games referred to as constant sum games since our player’s utilities always add up to zero. These zero-sum games refer to games of pure conflict, where the payoff of one player is equal to a negative value of the other player. In other words, one player’s gain is another’s player loss. [12; 13] Unlike common-payoff games, where each payoff for each action is the same for both players, constant-sum games are primarily useful in the context of two-player games. [9, p.57] This needs to be mentioned as the experiment is built on a two-player zero-sum game. As said, every iteration adds a countable number of actions available to players into a matrix representing their possibly infinite set in its interval, in our case [0,1].

Each value in this matrix represents a utility. We mark the matrix M.

$$M = [m_{ij}] \in R^{m*n}$$

Formally, we can then define it as follows. Let there be a finite number of strategies denoted by I and J. There is always an action player chooses, which is called a play. Players choose what they play simultaneously, with one choosing from $i \in I$ and the other $j \in J$. Value m_{ij} is called a gain for player A, which equals to a loss for player B. Note that a rational player $i \in N$ chooses a strategy that maximizes u_i gain[4]. Specifically,

$$0 = m_{ij} + (-m_{ij}), \forall i, j \in I, J$$

Here we see the reason why it is called zero-sum games.

2.5.2 Matching pennies

To strengthen our environment's understanding, we show an example of a zero-sum game, which is called Matching Pennies. This game consists of two players, each having one personal coin, who simultaneously choose to display either heads or tails, then the two players compare what they have chosen. If coins are the same, then player 1 takes them both, and otherwise, player 2 receives them. The payoff of all possible outcomes is displayed below:

	Heads	Tails
Heads	1,-1	-1,1
Tails	-1,1	1,-1

Another popular game, which can be used as an example for this game type, is Rock, Paper, Scissors. It is widely regarded as a three-strategy generalization of the above-explained Matching Pennies game and needs to be acknowledged. Here if two players choose the same option, then the utilities are zero. Otherwise, each action wins only against one of the remaining two actions and loses to the other. [9, p.58]

2.5.3 Other uses

So far, we have covered some specific uses of game theory in mathematical problems, but this field has already expanded elsewhere. For example, it is extensively used in economics, sociology, political science, and others because of the versatile nature and applications in many conflicts and problems. Another property is robustness, causing game theory's extensive use in computer science fields, which we show examples of. The two uses which we will mention are fields of cyber security and cloud computing. [14]

2.5.3.1 Cloud computing

The NIST, or National Institute of Standards and Technology, defines cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computer resources (we regard these as mostly storage, applications, and services, but resources also include networks or servers). Remaining available on-demand provides a rapid speed of provisioning and releasing with minimal management overhead or service provider interaction. [15] As cloud computing is becoming more popular, more challenges rise along. Here are two problems that are being solved with games theory.

2.5.3.1.1 Cloud Cyber Space Security

Cloud cyber space has expanded into a multi-dimensional space that extends over various areas. It is due to this reason that conventional methods cannot be used for their security.

[4] https://cw.fel.cvut.cz/b192/_media/courses/b0b33opt/13games.pdf

The first approach is called secure virtual machines, which use Nash Equilibria to analyze cause-effect interdependencies in the public cloud. [16] Another is called the scalable security risk assessment model. This model was created to respond to a vast number of attacks such as data breaches, data loss, hacked interfaces, insecure APIs and DDOS attacks. It works by evaluating the risk and deciding whether the provider or the client causes it. [17]

The last problem is cloud security transparency problem. [16] It can be modeled as a non-cooperative stochastic problem where the client and cloud provider are considered players. This is also a Nash Equilibrium problem with the client deciding whether to choose the provider or not based solely on the level of transparency provided.

2.5.3.1.2 Pricing Strategies

Cloud beneficiaries compete with each other for maximum financial gain advantage. The market model depicts the provider's and the client's potential behavior and rewards involved. We will now describe one of these models.

An extensive form game [18] is a model where a provider makes an offer to the client, who is free to accept it or not. We arrive at Nash equilibria with two players, each viewing the situation differently. On the one hand, the client wants an offer cheaper than making his data center or at least at the same price. On the other hand, the provider wants to make the most significant possible profit.

Other models include Discriminatory pricing policy, Uniform Pricing Policy, or Resource pricing. [14]

2.5.3.2 Cyber security

An interesting approach in solving cyber security attacks like Denial of Service, Brute force, or SQL injection [19] is with game theory in which the ubiquitous attacker is considered a player system administrators are considered as a player on the opposite side.

Two models portray the approach, one of which is with static games, where players make decisions based on prior knowledge about the opponent's behavior. In [16], they mention various economic problems in cyber security for resource allocation or overall investment in security protection for divergent defense mechanisms, which can be solved using these static game models.

The second model is with imperfect information stochastic model. We have two functions for portraying this model [16]. The first is Min-max Q, which wants to improve decision-making for a player in multi-player games with players described previously using zero-sum equilibrium. [20] Nash-Q is the second one, built on the fact that Nash equilibrium is a baseline answer to all general sum games presented in [21]. Every player has some correct expectation for the behavior of other players. It adopts the Markov decision process and has many applications in multiagent environments, for example, robotic soccer games.

Overall, there are five models. The remaining two are the cooperative model and the static prisoner's problem. [14]

To summarize, we need to note that even with these computer science uses. We stumble upon some limitations. Precisely, cyber security can't quantify the parameters of cyber space, affecting the decision-making process. [19] We work with equilibrium created from a very small number of cloud service providers and clients for cloud services. Therefore, we need to make these models more scalable for their more practical usage. [22]

3 Nash Equilibrium

As we have now discussed games in game theory, there is a need to define the term Nash Equilibrium formally. What makes Nash's theorem so crucial is its wide usage in many real-life examples usually presented in a famous Prisoner's dilemma. It is also essential to understand it before explaining what we are trying to find in our experiment. Now we consider a game from a player's point of view rather than from an outside observer.

3.1 Definition

A Nash equilibrium describes a set of strategies, one for each player i, where none of the players has the incentive to deviate since it would result in his loss of payoff. It can be written formally as follows:

The best response of player i to the strategy profile s_{-i} is a mixed strategy $s_i^* \in S_i$ such that $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$ for all strategies $s_i \in S_i$.

A strategy profile $s = (s_1 \dots s_n)$, where if for every player i is s_i the best response to s_{-i} can be called Nash Equilibrium. [9, p.62]

3.2 Linear program

Our program uses an algorithm that utilizes finding Nash equilibria with linear programming (LP). This method results in solving equilibria in polynomial time.

Let us consider a two-player, zero-sum game $G = (\{1, 2\}, A_1 \times A_2, (u_1, u_2))$. We set $U^*{}_i$ to be the expected utility for player i in equilibrium, it is also known as the game's value. As we pointed out in our zero-sum game definition $U^*{}_2$, or the expected utility for the second player, needs to be a negative of $U^*{}_1$, resulting in their combined sum is zero. The min-max theorem (in Section 3.4.1 and Theorem 3.4.4. in [9]) tells us that our expected utility remains constant in all equilibria. It also explains why player 1 achieves the same value as under a min-max strategy by player 2. Using this, we construct the linear program as follows.

$$\min U_1^*$$
$$subject\ to \sum_{k \in A_2} u_1(a_1^j, a_2^k) * s_2^k \leq U_1^* \quad \forall j \in A_1$$
$$\sum_{k \in A_2} s_2^k = 1$$
$$s_2^k \geq 0 \quad \forall k \in A_2$$

Having minimization for one player, we can transform it into its dual program form and create a maximization program for the other player [9, p.89-90].

$$\max U_1^*$$
$$subject\ to \sum_{j \in A_1} u_1(a_1^j, a_2^k) * s_1^j \geq U_1^* \quad \forall k \in A_2$$
$$\sum_{j \in A_1} s_1^j = 1$$
$$s_1^j \geq 0 \quad \forall j \in A_1$$

4 How to Deal with Games Having Piecewise-Affine (PA) Utility Functions

We already mentioned what piecewise-affine functions are. This section introduces them more in section 4.4, along with methods to solve our strategic zero-sum game. [27, p.79-80]

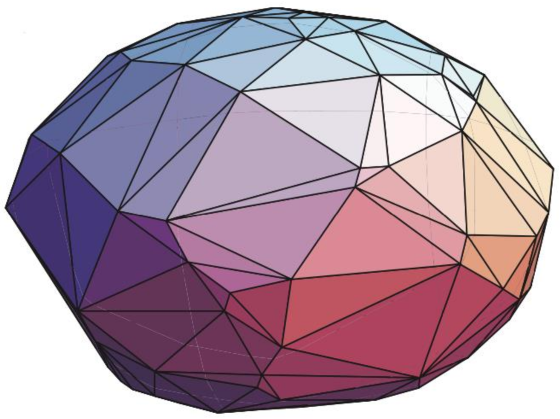
4.1 Domains of linearity are polyhedra

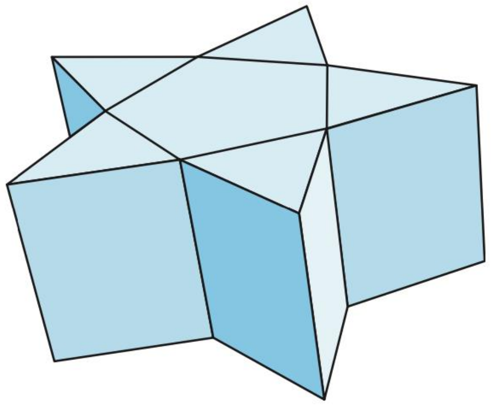
Before we define polyhedra, let us realize that our game shape has constraints in the form of affine functions. A polyhedron is defined as an intersection of half-spaces (in our case represented by affine functions), creating a set of linear inequalities. Formally we define it as follows:

$$\{x \in \mathbb{R}^n \mid Ax \leq b\}$$

$$\text{where } A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$$

A representing a matrix of m rows and n columns, and b is a vector of m entries. Here is an example of convex polyhedron on the left and non-convex on the right.





[27, p.157]

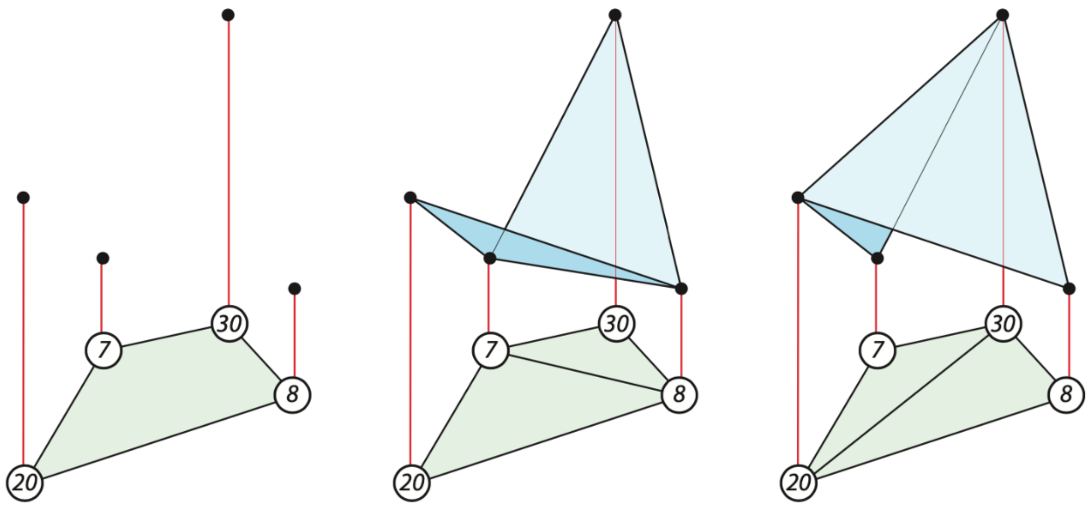
In our experiments, we have a square of size $[0,1] \times [0,1]$ where we have random points with random heights placed. Therefore, by connecting our vertices, they form half of possibly non-convex polyhedra with triangular faces. [29, p.156-160] We have become more familiar with this shape as it is used to calculate the probability distribution for our players.

4.2 Triangulations

We describe a triangulation for a set of points P. The term edge is used when discussing a line segment containing exactly two points from P as endpoints. A triangulation of P is its subdivision into a maximal possible set of non-intersecting edges, where the set of these vertices are points from P. Maximal meaning that for every other possible connection of points from P an intersection informed with other already existing lines. [29, p.59].

4.2.1 Delaunay triangulation

There are many different algorithms to triangulate a specific space. However, for our space decomposition Delaunay algorithm is the best as it is mainly used for terrain reconstruction, which our program is also technically representing. The question to ask is which of the possible triangulations is the most suitable from sampled heights. Even in real life, we do not know the Earth's exact shape, but only at the sample points presented. The choice will, therefore, have a major impact on what will be the terrain's appearance. The example below shows a different way of connecting points with different heights. We can either connect high or low positioned points, and therefore if we imagine this layout, in reality, get either a hill or a valley.



[27, p.97]

For a triangulation to be Delaunay, it needs to meet some conditions. One of which says that no four points are cocircular. Cocircularity describes a case where if we make a circle passing through all three vertices constructing one triangle, the circle does not pass through any other point. [6, p.81] In our case as we have a square $[0,1] \times [0,1]$ this requirement is mostly satisfied since we experiment with a smaller number of points in a grid with many free spaces. This results in no square present unless the chosen number of points fills most of these spaces in a setting creating it. If that is the case, then our triangulation cannot be rightfully called Delaunay. Still, the algorithm in use will create a triangulation for every setting due to it always using the same heuristics for a specific set of points. However, all experiments verifying the finite equilibria will be triangulated using the Delaunay triangulation.

Now, we will go over the steps that make the definite shape of our triangulation. Let us have triangulation T of our point set P, suppose T has n triangles. Therefore there are $3n$ angles which create a sorted angle sequence $(\alpha_1 ... \alpha_n)$, where the first being the smallest angle and the last being the largest one. Why do we want to have angles for a specific triangulation sorted? As seen in the picture above, we have two triangulations, if points B (height 7) and D (height 8) were positioned lower, then middle triangulations would appear less natural terrain, and therefore larger angles result in a more realistic generation. When a triangulation has a larger sorted angle sequence, then it is called a fatter triangulation. For the two triangulations $T1$ and $T2$ of P, we say $T1$ is fatter than $T2$ if $T1$ has a lexicographically greater angle sequence than $T2$. We will present two sequences with $T1$ being $(10,20,30)$ and $T2$ being $(10,25,30)$. We see that a second angle in the second sorted list is greater than the one in $T1$. Therefore, we would mark $T2$ as a fatter triangulation. When the fattest triangulation is what we seek, how do we go about finding it? Edge flipping is one elegant way of finding such desired triangulation.

“Definition: Let e be an edge of a triangulation $T1$, and let Q be the quadrilateral in $T1$ formed by the two triangles having e as their common edge. If Q is convex, let $T2$ be the triangulation after flipping edge e in $T1$. We say e is a legal edge if $T1 \geq T2$ and e is an illegal edge if $T1 < T2$.”[6, p.82]

Flipping of one e changes six angles in $T1$ angle sequence and replaces them with their counterparts in $T2$. However, this counts on the lexicographical ordering of angles. It helps complete the definition mentioned above with a declaration that all hull edges of triangulation are legal. We are looking for the fattest one. We tend to avoid illegal edges. Therefore, a Delaunay triangulation of P, noted as $Del(P)$, has only legal edges. [29, p.81-82]

4.3 Equilibrium existence

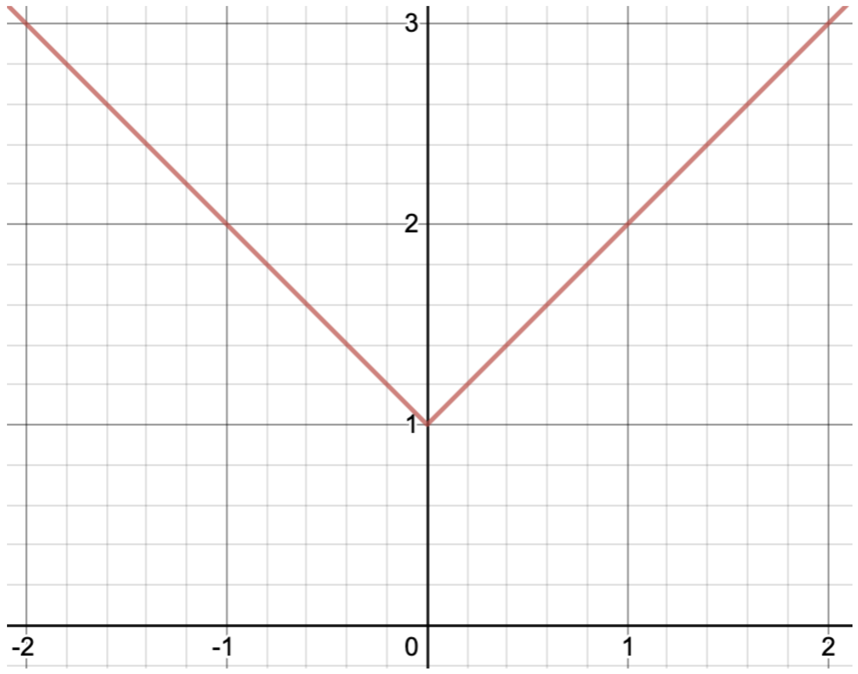
To explain the process, we need to mention the type of games we are using are nondegenerate games. Our two-player game is called nondegenerate if there exists no mixed strategy of specific size k with more than k pure strategies. Here we further explain piecewise-affine functions and ideas behind solving.

4.3.1 Piecewise-affine functions

For our strategic zero-sum game to be piecewise affine, it needs strategy sets that are $[0,1]$ and $u: [0,1]^2 \rightarrow R$ to be a piecewise affine function. Piecewise affine functions are sometimes generalized under the common, and less accurate, name of piecewise linear functions. The name linear only applies when there is no offset to the function. Let us recall the term piecewise. It refers to a case where a function is represented by a combination of equations that create a full domain rather than a typical single equation, which might not represent real-world examples.[5] Formally:
“A continuous function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is called piecewise affine if there exists a finite set of affine functions $f_i(x) = A_i x + b_i$”[30], $i = 1, \ldots, k$, such that the inclusion $f(x) \in \{f_1(x), \ldots f_k(x)\}$ holds for every $x \in \mathbb{R}^n$. The affine functions $f_i(x)$, are called selection functions, the set of pairs (A_i, b_i), is called a collection of matrix-vector pairs corresponding to f. The function f is called piecewise affine if there exists a corresponding set of linear selection functions.[30]

To visualize piecewise-affine functions, we show an example for a function $y = |x + 1|$ (a function split along y axis). Absolute value sets every x smaller than zero to $-x$, and every larger value is kept as x. Though we only use the interval $[0,1]$, we introduce the formal definition applicable to the whole of $\mathbb{R}$:
$f: y = |x| + 1$ is $-x + 1 \; if \; x \leq 0$ and $x + 1 \; if \; x > 0$
This fact is depicted in the graph below:

[5] https://www.classzone.com/eservices/home/pdf/student/LA202GAD.pdf



4.3.2 Existence and computation of finitely supported mixed strategy equilibria

Let us have $f \in PA_2$ as a two-variable piecewise-affine function. Such strategic game with piecewise-affine payoff functions f is the strategic game Γ_f with the player set $N = \{1,2\}$, the strategy spaces $\mathrm{A}_1 = \mathrm{A}_2 = [0,1]$, and the payoff functions $\mathrm{f}_1 = -\mathrm{f}_2$. We want to prove that f be such that every line segment in $G(f)$ is part of some polytope $P(f)$. Then the game Γ_f has to have a Nash equilibrium consisting of finitely supported mixed strategies. To prove a two-player constant-sum game is used with the strategy spaces of X_f and Y_f. The payoff function of player 1 is the restriction f_0 of f to $X_f \times Y_f$, and the payoff function for the second player is it's negative noted as $\neg f_0$. Here each strategy set is finite, making this game solvable in mixed strategies by the Nash Theorem [29]. Next, assume that the probability vectors give an optimal pair of strategies of players 1 and 2 $(\alpha_1 \dots \alpha_p)$ and $(\beta_1 \dots \beta_q)$ respectively and δ as the symbol for Dirac measure. Put

(6)

$$\mu^* = \sum_{i=1}^{p} \alpha_i \, \delta_{x_i} \; AND \; \upsilon^* = \sum_{j=1}^{q} \beta_j \, \delta_{y_j}$$

Where $x_i \in x$ and $y_j \in y$ with x, y defined as follows:

$0 = x_1 \leq \cdots \leq x_p \leq 1$ and $0 = y_1 \leq \cdots \leq y_q \leq 1$

We will show that $(\mu^*, \upsilon^*) \in \Delta^2$ is a finitely supported Nash equilibrium in the game Γ_f.

Firstly, the inequality for expected payoffs is:

(7)

$$E_f(\mu^*, \upsilon^*) \geq E_f(x, \upsilon^*)$$

This holds for every choice of pure strategy $x \in [0,1]$. Moreover, inequality

(8)

$$E_f(\mu^*, \upsilon^*) \geq E_f(x_i, \upsilon^*)$$

Will be true for every $i = 1, \ldots, p$ since (μ^*, υ^*) is an equilibrium of the game associated with f_0 and the value of this game is $E_f(\mu^*, \upsilon^*)$. For every x, there exist some $\gamma \in [0,1]$ and some $i = 1, \ldots, p-1$ such that $x = \gamma x_i + (1-\gamma)x_{i+1}$. Hence

(9)

$$E_f(x, \upsilon^*) = \sum_{j=1}^{q} \beta_j f(x, y_j) = \sum_{j=1}^{q} \beta_j f(\gamma x_i + (1-\gamma)x_{i+1}, y_j)$$

For every $j = 1, \ldots, q$, the line segment with endpoints (x_i, y_j) and (x_{i+1}, y_j) is included in some polytope $A_j \in P(f)$ As the function f is linear over A_j, the sum defined earlier for E_f becomes

(10)

$$\sum_{j=1}^{q} \beta_j \left(\gamma f(x_i, y_j) + (1-\gamma) f(x_{i+1}, y_j)\right) = \gamma \sum_{j=1}^{q} \beta_j f(x_i, y_j) + (1-\gamma) \sum_{j=1}^{q} \beta_j f(x_{i+1}, y_j)$$
$$= \gamma E_f(x_i, \upsilon^*) + (1-\gamma) E_f(x_{i+1}, \upsilon^*) \leq \gamma E_f(\mu^*, \upsilon^*) + (1-\gamma) E_f(\mu^*, \upsilon^*) = E_f(\mu^*, \upsilon^*)$$

Where the inequality follows from (8), proving (7). The proof that υ^* is an optimal strategy for the second player can be proven in the same matter.

As we have gone through this proof, we have not assumed the integer values of linear coefficients of our function f. Therefore, this theory holds for piecewise-affine continuous functions with real coefficients. Also, there were only two players, with their payoffs always remaining constant. The task specified is standardly solved with linear programming as introduced in section 3.

5 Program Explanation

The following program is built to confirm whether infinite two-player zero-sum games have finite equilibria. We chose a Python programming language to program it. The decision to code in Python was mainly for language readability, even for users unfamiliar with it. The user's input is used to build the experiment and specify the base game's appearance. The algorithm has four inputs (`points_number`, `heights_number`, `kernel_sizer`, `steps`), which are more specified in section 5.1. Experimental work is based on the generation of random triangulations, together with their piecewise affine functions. Their shapes are directly dependent on how inputs are set (specifically `points_number`, `kernel_sizer`). The following approximation of these functions over a finite grid and determination of the game's equilibria are iteratively checked. Iterations are used to see if game results are stabilizing to verify the finiteness of such equilibria. There are four examples shown—one described in detail to visualize the algorithm's process. The rest is used as follows: first to show five rounds of stabilization to support the detailed example; second mainly to show all points generated in later iterations, as they cannot typically be visualized properly due to their large number; third shows iterations until testing computer's memory runs out.

5.1 Summary

We have explained the theory used in our experiment. Now it is high time to dismantle the algorithm into a detailed description to see how it solves the problem. Altogether, the algorithm consists of two main parts. In the first part, we have to create an environment where the game is to be played. This means creating a square $[0,1] \times [0,1]$, which will hold the points building strategies for the players. The second part focuses on using these points in a game and solving it for both players. The goal is to monitor whether the results stabilize and approach equilibria for players whilst we perform more iterations. Let us introduce variables required for starting the run of the algorithm. The program needs four parameters as input:

`points_number` – This parameter is used to set a number of points to populate the square $[0,1] \times [0,1]$. The minimum required number of points is four as we always put four points into corners.

`heights_number` – The number changing the possible value of interpolated functions of our points is called `heights_number`. These values are called height, as mentioned. This value is assigned to each point. Note that `heights_number` must be greater than 0.

`kernel_sizer` – As we work with a square $[0,1] \times [0,1]$ we need to specify the sizes of our bins to set points on. Moreover, points could be placed anywhere inside of the square. However, it is better to have them placed in an orderly manner as we plan to make a finer grid out of them. `Kernel_sizer` specifies a grid's dimensions between the numbers 0 and 1, where the points will be placed. This number needs to be a divider of 1, since we want these dimensions to divide the game space evenly. If this is not satisfied, then it is changed to the default value of 0.2.

`steps` – Lastly, we use steps to indicate how many iterations we want our algorithm to run. Each iteration consists of the second part of the algorithm. As our program needs to compare some values for our verification, it is suggested to choose a value of 2 or greater, but no smaller than 1.

If a user wants to run the algorithm according to the chosen value, they need to fulfill the requirements specified above. Otherwise, they are changes to the lowest possible or default value. Many pictures are staged for visualization as the default Python output takes an unreasonable amount of space. All of these values are taken from outputs.

In the first part, we use:

Input: `points_number`, `height_number`, `kernel_sizer`

Output: Triangulated square $[0,1] \times [0,1]$ with randomly placed points and heights along each gridline corner

The second part uses an output of the first one; therefore we end up with:

Input: Triangulated square $[0,1] \times [0,1]$ with randomly placed points and heights along each gridline corner

Output: $x = (x_1, \dots, x_n)$ is a mixed strategy of Alice, where we show only those $x_i > 0$

$y = (y_1, \dots, y_n)$ is a mixed strategy of Bob, where we show only those $y_i > 0$

Together with our printable outputs, we show a triangulated space, intersection points for each iteration, and probability distribution over players' mixed strategies. To put the algorithm into perspective, we follow a random example for a generation of 10 points, with `kernel_sizer` set to 0.2 and `height_number` set to 50 to avoid an overwhelming number of decimal places from heights calculation. Code is split into functions, one for each job needed. There is one function arching over all calculations of both parts, which is called `runner`. This prevents having code in the `main` function. Firstly, we describe steps done for the first part of the program titled Triangulation Part.

5.2 Triangulation

The triangulation part starts with taking the first three manual inputs (`points_number`, `heights_number`, and `kernel_sizer`) and passing them into `pa_games_watch`. It returns `tri`, `coords`, `coords_tri`, `threedim_coords`, `heights_orig`, `visual_heights`, and `visual_grid`. We will now explain each function and explain what these variables hold.

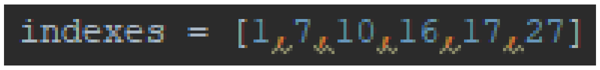
5.2.1 Square creation

5.2.1.1 *Kernel_number*

Firstly, to transform our inputs into a more intuitive form, we use `kernel_sizer` (now called `kernel_size`) to calculate how many boxes there are between 0 and 1. This is done by dividing 1 by the `kernel_size`. For our example with 0.2 we have $\frac{1}{0.2} = 5$ boxes to store into `kernel_number`.

5.2.1.2 *Indexes*

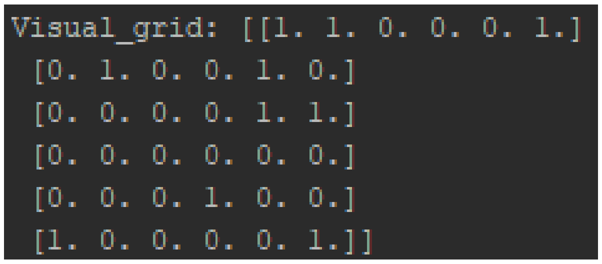
Secondly, we call `make_indexes` with `numvars` (`points_number` renamed for the function) and `kernel_number` from earlier. Indexes begin at zero and go up to the `kernel_number` plus one squared minus one. For our example, this is $(5+1)^2 - 1 = 35$. The function starts with setting corners as chosen points and then uses `np.sort` together with `np.random.choice` to return a list of unique sorted placements for points. This array will have a size of `points_number` minus four as corners are chosen by default.



```
indexes = [1,7,10,16,17,27]
```

5.2.1.3 *Visual_grid*

In the last part of square creation, we call `visualize_grid` with `kernel_number` and `indexes` from previously. This function creates a one-dimensional array populated with zeros using `np.zeros` with an argument size of `kernel_number` plus one. Function substitutes ones for zeros on all indexes from `indexes`. The next step is to create a two-dimensional square from the one-dimensional array and add corners since they are not in `indexes`. This is done with `np.reshape`.



```
Visual_grid: [[1. 1. 0. 0. 0. 1.]
 [0. 1. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1. 1.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 1.]]
```

We have a two-dimensional array/square with dimensions of 6×6 to satisfy $0 - 5$ indexes holding our points on randomly assigned indexes.

Input: `points_number, height_number, kernel_sizer`

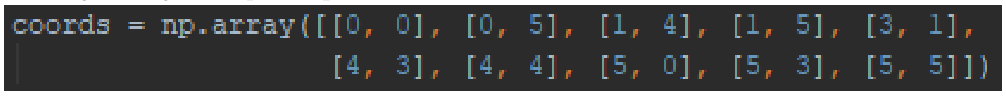
Output: Square [0,1] × [0,1] with size 6 × 6 (corresponding to `1 / kernel_size + 1`) with randomly assigned points (with visualization)

5.2.2 Value assigning

We need to set random interpolated functions values, which are called heights in the program, for our points.

5.2.2.1 Coords

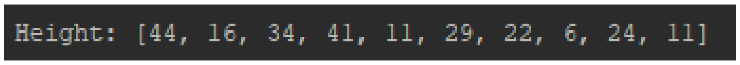
Before we go over the heights, there is a variable called `coords`, which stands for a list holding coordinates of all points. This is done using `np.argwhere` for the two-dimensional variable `visual_grid`, finding where values are greater than zero since we used an array with only zeroes and changed them to ones only for spots where the points were assigned. Afterward, we end up with a list of lists with the size equal to the number of points specified in `points_number`. This list will be used in this part for triangulating our square space.



```
coords = np.array([[0, 0], [0, 5], [1, 4], [1, 5], [3, 1],
                   [4, 3], [4, 4], [5, 0], [5, 3], [5, 5]])
```

5.2.2.2 Heights

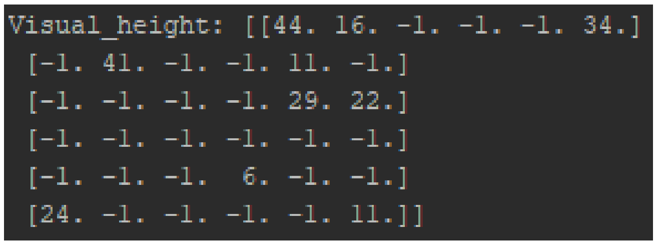
Similarly to how we chose the points, we will choose values for the points' heights—calling a function `make_heights` with `numvars` and `max_height` (known earlier as `heights_number`). Random generation of values is done using `np.random.randint`. It takes in a minimum and maximum value but chooses between them; therefore, we use `max_height` plus one as a maximum. The values are represented in an array which is saved as `heights`. After returning from the function, it is renamed to `heights_orig`.



```
Height: [44, 16, 34, 41, 11, 29, 22, 6, 24, 11]
```

5.2.2.3 Visual heights

To visualize our `heights` we call a function called `visualize_heights` with `heights` and `visual_grid` as parameters. To make it more readable when copying the two-dimensional array using `np.copy` we subtract one from it to have it as a clear indicator of where the points are not present. Using pattern matching, we assign a value from the heights array for each value greater than minus one.



```
Visual_height: [[44. 16. -1. -1. -1. 34.]
 [-1. 41. -1. -1. 11. -1.]
 [-1. -1. -1. -1. 29. 22.]
 [-1. -1. -1. -1. -1. -1.]
 [-1. -1. -1.  6. -1. -1.]
 [24. -1. -1. -1. -1. 11.]]
```

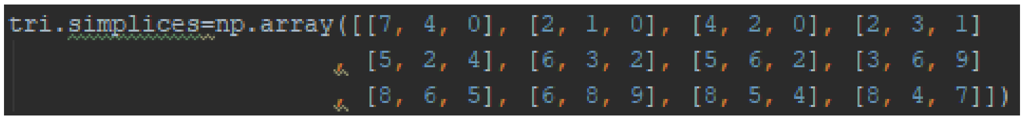
Input: Square [0,1] × [0,1] with size 6 × 6 (corresponding to `1 / kernel_size + 1`)

Output: Square [0,1] × [0,1] with size 6 × 6 (corresponding to `1 / kernel_size + 1`) with randomly assigned points and heights (with visualization)

5.2.3 Triangulation function

5.2.3.1 Tri

Now when we have all points and heights for the triangulation, we use a `scipy.spatial.Delaunay` to create triangles from our points. This library uses Qhull algorithms. The convex hull of a set of points P in n dimensions is R^n. A set $Q \in R^n$ is convex if for all $q_1, q_2 \in Q$ the line $q_1 q_2$ is fully within Q. The Convex hull of the set of points P can be described as the smallest wrapping of such points[6]. When asked how many triangles we have in triangulation, we look at our wrapped points, shaped like a half polyhedra, and mark the number of points it consists of as n corners. The rest of the points which are inside our half polyhedra will be m. We can find a triangulation of a pivot corner p by drawing a line to $n-3$ corners with no edge to p. This, together with half polyhedra edges, gives us $n+(n-3)=2n-3$ edges and $n-2$ triangles. The next step is for us to take each of the inner points m and do the following:
For a point $q \in m$ we find the triangle it lies in and connects it with its edges. This gives us another 3 edges and 2 more triangles. After pursuing this for all the points in m we get $2n-3+3m=2n+3m-3$ edges and $n+2m-2$ triangles[7].
Even though this is a special way of creating triangles, the number of them and edges remains constant for all other triangulation forms. The plane we perform this in is not without special cases. For this triangulation to stay omnipresent, there need to be no four points along a circle circumference. To avoid a square of points, we set a small `kernel_sizer`, creating many options for placements and `points_number`. To clarify, imagine having a square $ABCD$. When you want to triangulate it, you can either connect AC or BD. These dual options for edges are always done in the same way. This is due to points being iterated through in the same order. Therefore, even though not triangulated by Delaunay, the triangulation will always be the same. We call this function with parameter `coords`. Since a library does the calculation is stores more than single information into `tri`. Calling `tri.simplices` returns triangles marked by indices of the points in an array `coords` with coordinates.



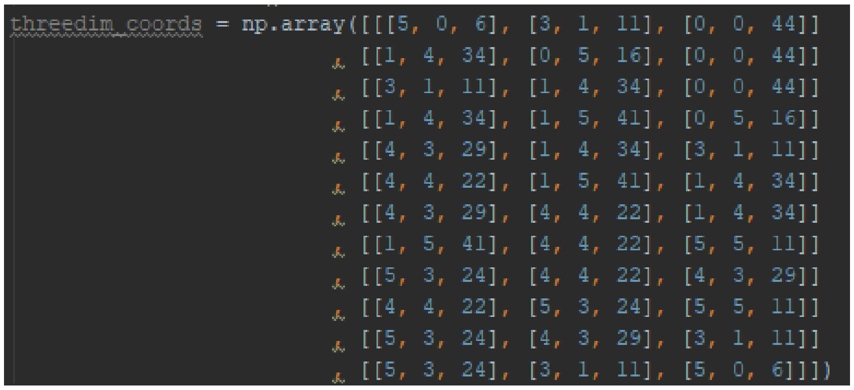
```
tri.simplices=np.array([[7, 4, 0], [2, 1, 0], [4, 2, 0], [2, 3, 1]
                        , [5, 2, 4], [6, 3, 2], [5, 6, 2], [3, 6, 9]
                        , [8, 6, 5], [6, 8, 9], [8, 5, 4], [8, 4, 7]])
```

5.2.3.2 Threedim_coords

To better view how our triangles are subdividing the square space, we use `np.insert`. Calling `coords` with `tri.simplices` we now, instead of having just indices of the coordinates, have triangles with both coordinates and heights. This creates a list with lists of lists. `Threedim_coords` represents three-dimensional information about all our triangles.

[6] http://www.qhull.org/

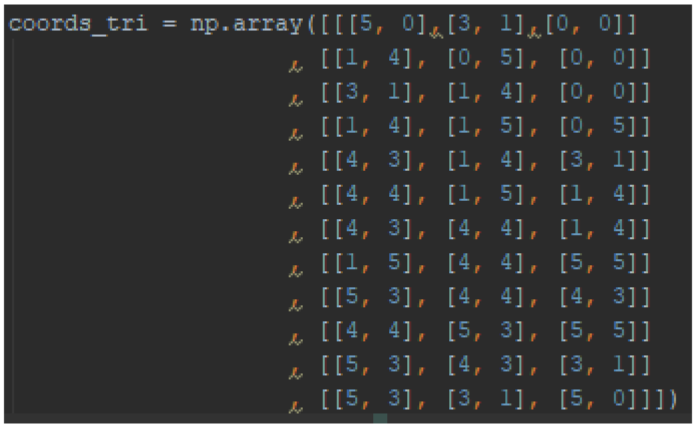
[7] https://www.uio.no/studier/emner/matnat/ifi/INF4130/h18/slides/forelesning-11---triangulering-og-convex-hull.pdf



```
threedim_coords = np.array([[[5, 0, 6], [3, 1, 11], [0, 0, 44]]
                           , [[1, 4, 34], [0, 5, 16], [0, 0, 44]]
                           , [[3, 1, 11], [1, 4, 34], [0, 0, 44]]
                           , [[1, 4, 34], [1, 5, 41], [0, 5, 16]]
                           , [[4, 3, 29], [1, 4, 34], [3, 1, 11]]
                           , [[4, 4, 22], [1, 5, 41], [1, 4, 34]]
                           , [[4, 3, 29], [4, 4, 22], [1, 4, 34]]
                           , [[1, 5, 41], [4, 4, 22], [5, 5, 11]]
                           , [[5, 3, 24], [4, 4, 22], [4, 3, 29]]
                           , [[4, 4, 22], [5, 3, 24], [5, 5, 11]]
                           , [[5, 3, 24], [4, 3, 29], [3, 1, 11]]
                           , [[5, 3, 24], [3, 1, 11], [5, 0, 6]]])
```

5.2.3.3 Coords_tri

`Coords_tri` is very similar to `threedim_coords`, but now we only want two-dimensional coordinates for our triangles. To do this, we call `tri.simplices` on `coords`, and get a list with lists of lists for only coordinates of all vertices for each triangle.



```
coords_tri = np.array([[[5, 0],[3, 1],[0, 0]]
                      , [[1, 4], [0, 5], [0, 0]]
                      , [[3, 1], [1, 4], [0, 0]]
                      , [[1, 4], [1, 5], [0, 5]]
                      , [[4, 3], [1, 4], [3, 1]]
                      , [[4, 4], [1, 5], [1, 4]]
                      , [[4, 3], [4, 4], [1, 4]]
                      , [[1, 5], [4, 4], [5, 5]]
                      , [[5, 3], [4, 4], [4, 3]]
                      , [[4, 4], [5, 3], [5, 5]]
                      , [[5, 3], [4, 3], [3, 1]]
                      , [[5, 3], [3, 1], [5, 0]]])
```

Input: Square [0,1] × [0,1] with size 6 × 6 (corresponding to `1 / kernel_size + 1`) with randomly assigned points and heights (with visualization)

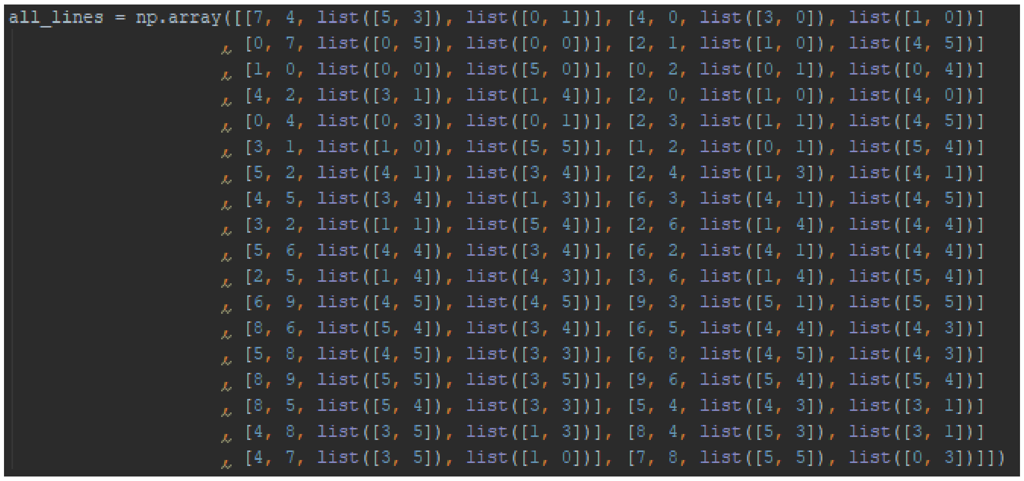
Output: List of coordinates and list of triangles as an array describing indices from the list of coordinates

5.3 Creating and Solving Zero-Sum Game

5.3.1 Line information

To clarify the work, we call a function `line_info` with `coords` and `tri.simplices` as parameters, which iterates through all triangles describing every line from them into the information array. Important to remember is that triangle is marked by indices of the points in `coords` array. Information about a line segment looks as follows:
Let us have a triangle ABC with two-dimensional coordinates. Firstly, for a line segment AB we take the index of A from a triangle and use it as the first value. Now we do the same with an index of point B and place it in the second position. The third index consists of an array of size two with x coordinates for A and B. The fourth index similar to the third instead now with y coordinates. This is also done for BC and CA. This array, called `all_lines` in our program, allows us to search for intersections much better than if we had to search for this information every time.

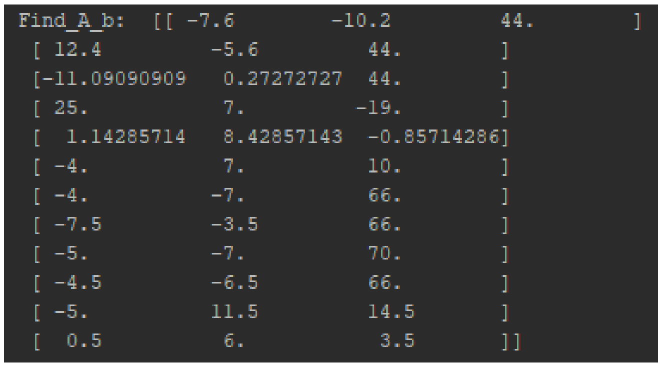
Input: List of coordinates and list of triangles as an array describing indices from the list of coordinates
Output: List describing each line with indices of points and X and Y coordinates together for each line



```
all_lines = np.array([[7, 4, list([5, 3]), list([0, 1])], [4, 0, list([3, 0]), list([1, 0])]
                       [0, 7, list([0, 5]), list([0, 0])], [2, 1, list([1, 0]), list([4, 5])]
                       [1, 0, list([0, 0]), list([5, 0])], [0, 2, list([0, 1]), list([0, 4])]
                       [4, 2, list([3, 1]), list([1, 4])], [2, 0, list([1, 0]), list([4, 0])]
                       [0, 4, list([0, 3]), list([0, 1])], [2, 3, list([1, 1]), list([4, 5])]
                       [3, 1, list([1, 0]), list([5, 5])], [1, 2, list([0, 1]), list([5, 4])]
                       [5, 2, list([4, 1]), list([3, 4])], [2, 4, list([1, 3]), list([4, 1])]
                       [4, 5, list([3, 4]), list([1, 3])], [6, 3, list([4, 1]), list([4, 5])]
                       [3, 2, list([1, 1]), list([5, 4])], [2, 6, list([1, 4]), list([4, 4])]
                       [5, 6, list([4, 4]), list([3, 4])], [6, 2, list([4, 1]), list([4, 4])]
                       [2, 5, list([1, 4]), list([4, 3])], [3, 6, list([1, 4]), list([5, 4])]
                       [6, 9, list([4, 5]), list([4, 5])], [9, 3, list([5, 1]), list([5, 5])]
                       [8, 6, list([5, 4]), list([3, 4])], [6, 5, list([4, 4]), list([4, 3])]
                       [5, 8, list([4, 5]), list([3, 3])], [6, 8, list([4, 5]), list([4, 3])]
                       [8, 9, list([5, 5]), list([3, 5])], [9, 6, list([5, 4]), list([5, 4])]
                       [8, 5, list([5, 4]), list([3, 3])], [5, 4, list([4, 3]), list([3, 1])]
                       [4, 8, list([3, 5]), list([1, 3])], [8, 4, list([5, 3]), list([3, 1])]
                       [4, 7, list([3, 5]), list([1, 0])], [7, 8, list([5, 5]), list([0, 3])]])
```

5.3.2 Height

The next function called is `find_A_b` with `coords(all_coords)`, `tri.simplices(triangles)`, `heights_orig(heights)`. It is used to find a function for each triangle in our triangulated plane. To do this we create an array full of zeros with the size equal to triangles. What we calculate is `y = Ax + b` or `y1, y2, y3 = (x1x, x2x, x3x) * a1 + (x1y, x2y, x3y) * a2 + b`, where y_i stands for a height of vertices in a triangle, x_{ix} and x_{iy} are x and y coordinates for each vertex in a triangle. To solve this equation, we use `np.linalg.solve(x, y)`, which calculates both A and b, and we need to split it with the first two being A (a_1, a_2) and the last being b. When we know A and b for a triangle, if we give it point's coordinates, we can calculate its height very precisely. This array of arrays is returned into `triangle_equations`. Before we can call for `find_height` it is necessary to look for new points to assign them these heights.



```
Find_A_b:  [[ -7.6        -10.2         44.        ]
 [ 12.4        -5.6         44.        ]
 [-11.09090909   0.27272727  44.        ]
 [ 25.          7.         -19.        ]
 [  1.14285714   8.42857143  -0.85714286]
 [ -4.          7.          10.        ]
 [ -4.         -7.          66.        ]
 [ -7.5        -3.5         66.        ]
 [ -5.         -7.          70.        ]
 [ -4.5        -6.5         66.        ]
 [ -5.         11.5         14.5       ]
 [  0.5         6.           3.5       ]]
```

Input: List describing each line with indices of points and X and Y coordinates together for each line
Output: List of $Ax + b = y$ for describing each triangle to return more precise height for future intersections

5.3.3 Intersections

The function `quadratic_direction` is called by two functions `first_round` and `next_rounds`. It is split for the ability to see the process more clearly. `First_round` calls it for initial randomly generated points, while `next_rounds` calls it for all points available in the following iterations, consisting of all points from the grid as explained in 5.3.4.1. It is rerun `steps` minus twice as `first_round` and `next_rounds` did two iterations already. Since these two functions are a little different, let us go over them in more detail. Before that, we explain how the intersections are searched for.

5.3.3.1 Quadratic direction

We use `quadratic_direction` to find intersections of a horizontal and a vertical line from a point. It checks with every line that our triangulation has created. This is very computationally intensive. Therefore as an improvement to the computation time, we added the direction from which the point was found. If the point was found as a horizontal intersection, there is no need to search for horizontal intersections. As mentioned, `quadratic_direction` checks each line segment for possible intersections, and there are a few different types of possible intersections from a point to a line segment. The easiest are the ones where either x or y coordinate is the same for both ends of the line segment. Then we can set new point's coordinates, being x or y, as from one of the points from the line segment and the other from the point that was searching intersections. Another option is when the line segment is scute to the pivot. In that case, we can use analytic geometry to count the slope or the segment and plug it into the equation. If our point were any one of four corners, we would skip it because it does not have any new intersections. The function is split into vertical and horizontal checking. We will go over an example in more detail. For this, we use one of the initial points [4,1,35,0] and look at how it intersects the line from [0,0] to [5,5], where A will be [0,0], and B will be [5,5]. The line will be intersected at coordinates 1 (by a horizontal line) and 4 (by a vertical one), creating two new points, where the missing coordinate will be calculated using the line equation. Since the original points are predetermined, they are the only points that need to check both directions. Others will only check the direction by which they were not made. This is noted on the fourth index of the point's array with initial points having this value set to zero with one meaning horizontal checking is needed and two being used for the vertical one. Therefore, if a point is deemed to be the one to go through the horizontal part, then before calculating, we check if the point has y coordinate between the line edges. Otherwise, it could not intersect this line. If it passes this condition, we check if y coordinates of the points ' coordinates are not the same since that would mean the imaginary line from a point in a horizontal direction towards the line segment does not intersect but instead create a line. Lastly, we check if this potential point is already present in our point collection. If that is not the case, we can add it to the collection. This collection saves all coordinates in a set called `new_candidate_check`, named in the round functions as `candidates`. A similar is done for vertical direction to get all four options mentioned above.

5.3.3.2 *Coords extended*

`Coords_extended` combines values for point coordinates and their respective heights. This is done by creating a list of lists with size for the number of points 3 using `np.zeros` and plugging coordinates to the first and second indexes and heights to the third. These values can be seen in `threedim_coords`, where they are put into triangles.

5.3.3.3 *First round*

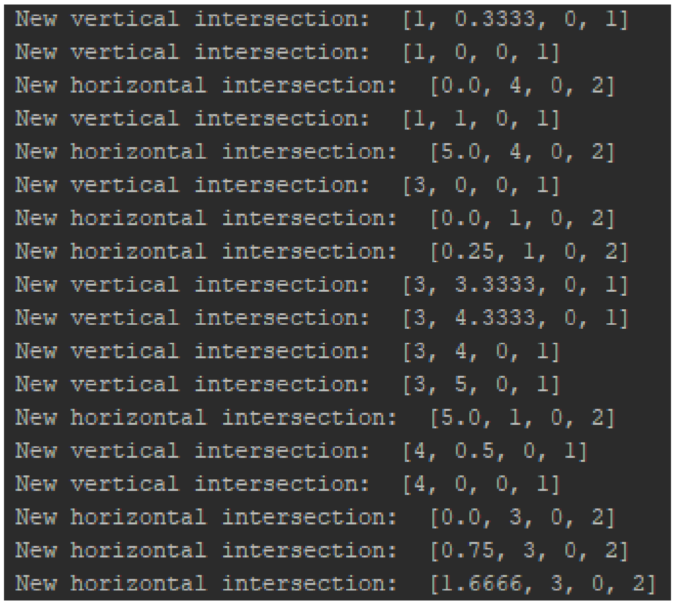
The parameters used are `coords`, `coords_extended`, `all_lines`, `kernel_sizer`, `heights_orig`, `sample_array`, `precision_check`, `intro`, `coords_tri`, `triangle_equations`. We now know all but three of them. `Sample_array` is an array of size four consisting of only zeros, and it is used as a sample for some other array, for example, `intro`, which holds points found in the first round. `Precision_check` describes the precision of calculations. Setting this number to two, we then round to this number of decimal places. `First_round` goes over all the initial points and checks the existence of intersections using `quadratic_directions`. If there is any new point found, then it is added into `intro` points with `np.append`. After all points have been checked, we use `np.unique` to find unique x and y from both `intro` and `coords` coordinates, in case numerical error caused the dual appearance of the same point. Then we make a list of them using `np.unique` and `np.append` to run `grid_maker`. We will explain it in more detail in 5.3.4.1, but to summarize, it creates a grid by calculating the Cartesian product from all points. It uses the heights of these points in linear programming to evaluate probability distribution for the game. The function `first_round` returns `intro`, `candidates`, `A`, and `gridpoints`. In the `runner` function, they keep their name, but `gridpoints` changes to `temp_gridpoints`. `A` is the matrix keeping the values for the mentioned linear programming. `Gridpoints` are all points creating the game grid in that round.

5.3.3.4 *Next rounds*

This function uses `candidates`, `steps`, `coords`, `coords_extended`, `all_lines`, `intro`, `sample_array`, `kernel_sizer`, `heights_orig`, `precision_check`, `coords_tri`, `temp_gridpoints`, `myDict`, `triangle_equations` as parameter. New here is `myDict`, which collects what points exist in each round to get a clearer view of how the game is developing. `Next_rounds` runs the same algorithm as `first_round`, but inside there is a for loop running `steps` minus one time. It returns `newer`, `newest`, `tester`, `A`, `gridpoints`, `myDict`, with `newer` representing all points from all rounds and `newest` represents a grid these points make. `Tester` is a copy of three-dimensional points from the beginning used mainly to check that our code is running properly and have this information available on hand throughout the process. The rest of the returns is already known.

Input: All points in the set and the list, information about lines and heights, storage for points in each round, number of iterations to run (for `next_rounds`, `first_round` only does round 1)

Output: Found intersection and updated grid from them with updated lists and sets of point



```
New vertical intersection:  [1, 0.3333, 0, 1]
New vertical intersection:  [1, 0, 0, 1]
New horizontal intersection:  [0.0, 4, 0, 2]
New vertical intersection:  [1, 1, 0, 1]
New horizontal intersection:  [5.0, 4, 0, 2]
New vertical intersection:  [3, 0, 0, 1]
New horizontal intersection:  [0.0, 1, 0, 2]
New horizontal intersection:  [0.25, 1, 0, 2]
New vertical intersection:  [3, 3.3333, 0, 1]
New vertical intersection:  [3, 4.3333, 0, 1]
New vertical intersection:  [3, 4, 0, 1]
New vertical intersection:  [3, 5, 0, 1]
New horizontal intersection:  [5.0, 1, 0, 2]
New vertical intersection:  [4, 0.5, 0, 1]
New vertical intersection:  [4, 0, 0, 1]
New horizontal intersection:  [0.0, 3, 0, 2]
New horizontal intersection:  [0.75, 3, 0, 2]
New horizontal intersection:  [1.6666, 3, 0, 2]
```

5.3.4 Zero-sum game

5.3.4.1 Grid maker

This function gets called with `xn`, `yn`, `coords`, `coords_tri`, `tester`, `triangle_info` by previously explained rounds functions, but since it consists of calculating the game, it is kept in the second part. X_n and y_n are lists of unique x and y coordinates of all points, respectively, and `triangle_info` is `triangle_equations`. With all-new points, we again use the Cartesian product to create a grid from all unique X and Y. This is done using `itertools.product(`X`, `Y`)`. `Gridsmooth_points` are all grid points, but `new_gridsmooth_points` represent our grided points without the initial ones that have their height values already. `Grid_maker` does two things. At first, it assigns grid points to their triangles to be able to assign them a height using the `find_height` mentioned above. From triangles, we calculate $c1, c2, c3$ for our points. If their coordinates together all sum up to either non-negative or non-positive, we found a triangle for the point.

```
c1 = (x2 - x1) * (yp - y1) - (y2 - y1) * (xp - x1)
c2 = (x3 - x2) * (yp - y2) - (y3 - y2) * (xp - x2)
c3 = (x1 - x3) * (yp - y3) - (y1 - y3) * (xp - x3)
```
[8]

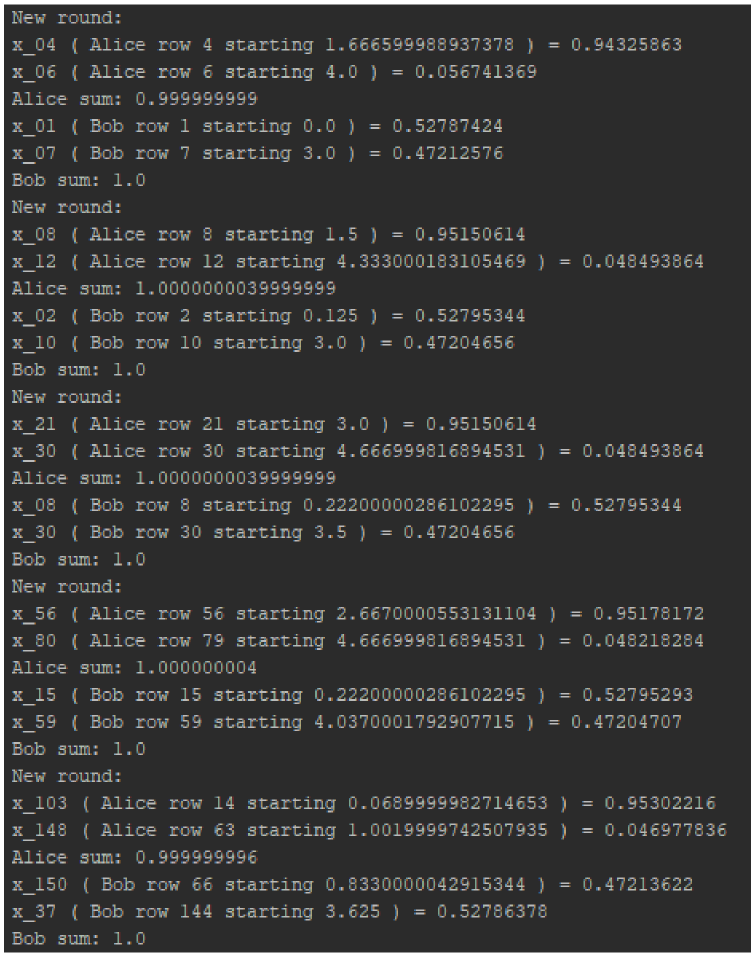
Now we add initial points back into `new_gridsmooth_points.` Because we need all of their heights to represent values for our zero-sum game. Next, the program computes a game for Alice and Bob, finding

[8] https://www.w3resource.com/python-exercises/basic/python-basic-1-exercise-40.php

their probability distribution. Here, we came across an issue with the possibility of solving the problem using Python libraries. Firstly, we set up a `linprog()` library, where one only needs to insert specific values into a function receiving an output. After getting various outputs and some research, we found this is not as robust as its Matlab programming language counterpart. We moved on to `nashpy()`, which was meant to be more stable with our examples. It held up better, but we could not test if there is something wrong or it is simply not robust enough. Lastly, before founding the library used (`pulp()`), we tried several other libraries, which did not work as needed. PuLP is the most stable out of all tested. It is harder to set up but easier to comprehend once all values are properly represented. When we wanted to put our dual program into a `pulp` problem, we needed to do each part manually.

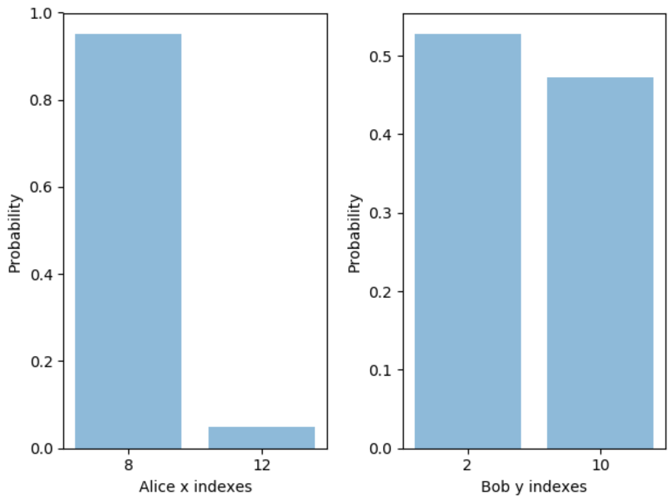
```
Alice = np.rot90(A, -1) # We use a different orientation for dual program
x_names = [format(x, '02d') for x in range(np.shape(Alice)[1] + 1)]
x = pulp.LpVariable.dicts("x", x_names, cat="Continuous")
for i in x.keys():
    if i == np.shape(Alice)[1]:
        continue # x0 has no bounds
    else:
        x[i].lowBound = 0  # BOUNDS x1-xn >= 0

prob = pulp.LpProblem("Alice", LpMaximize)  # MAX
prob += x[x_names[np.shape(Alice)[1]]]  # (max) x0

for i in range(np.shape(Alice)[0]):
        prob += lpSum(Alice[i, k] * x[j] for k, j in enumerate(x_names[:-1])) - 1 *
x[x_names[np.shape(Alice)[1]]] >= 0 # Ax - 1x1 >= 0
prob += lpSum(1 * x[i] for i in x_names[:-1]) == 1 # Sum of xs is 1
prob.solve()
```

For our dual program, we have a maximizing Alice. Therefore we create `LpProblem(“Alice”, LpMaximize)`. In this style, we add more equations to the problem. We can use a `for` cycle to help us add $Ax \leq 1x_0$and constraints. The last thing is to call `solve()` on our problem. The values printed are for the strategies, which received more than 0 probability of being played. Here is one example of the first five rounds. To visualize, we also create a graph from these probabilities.



```
New round:
x_04 ( Alice row 4 starting 1.666599988937378 ) = 0.94325863
x_06 ( Alice row 6 starting 4.0 ) = 0.056741369
Alice sum: 0.999999999
x_01 ( Bob row 1 starting 0.0 ) = 0.52787424
x_07 ( Bob row 7 starting 3.0 ) = 0.47212576
Bob sum: 1.0
New round:
x_08 ( Alice row 8 starting 1.5 ) = 0.95150614
x_12 ( Alice row 12 starting 4.333000183105469 ) = 0.048493864
Alice sum: 1.0000000039999999
x_02 ( Bob row 2 starting 0.125 ) = 0.52795344
x_10 ( Bob row 10 starting 3.0 ) = 0.47204656
Bob sum: 1.0
New round:
x_21 ( Alice row 21 starting 3.0 ) = 0.95150614
x_30 ( Alice row 30 starting 4.666999816894531 ) = 0.048493864
Alice sum: 1.0000000039999999
x_08 ( Bob row 8 starting 0.22200000286102295 ) = 0.52795344
x_30 ( Bob row 30 starting 3.5 ) = 0.47204656
Bob sum: 1.0
New round:
x_56 ( Alice row 56 starting 2.6670000553131104 ) = 0.95178172
x_80 ( Alice row 79 starting 4.666999816894531 ) = 0.048218284
Alice sum: 1.000000004
x_15 ( Bob row 15 starting 0.22200000286102295 ) = 0.52795293
x_59 ( Bob row 59 starting 4.0370001792907715 ) = 0.47204707
Bob sum: 1.0
New round:
x_103 ( Alice row 14 starting 0.0689999982714653 ) = 0.95302216
x_148 ( Alice row 63 starting 1.0019999742507935 ) = 0.046977836
Alice sum: 0.999999996
x_150 ( Bob row 66 starting 0.8330000042915344 ) = 0.47213622
x_37 ( Bob row 144 starting 3.625 ) = 0.52786378
Bob sum: 1.0
```

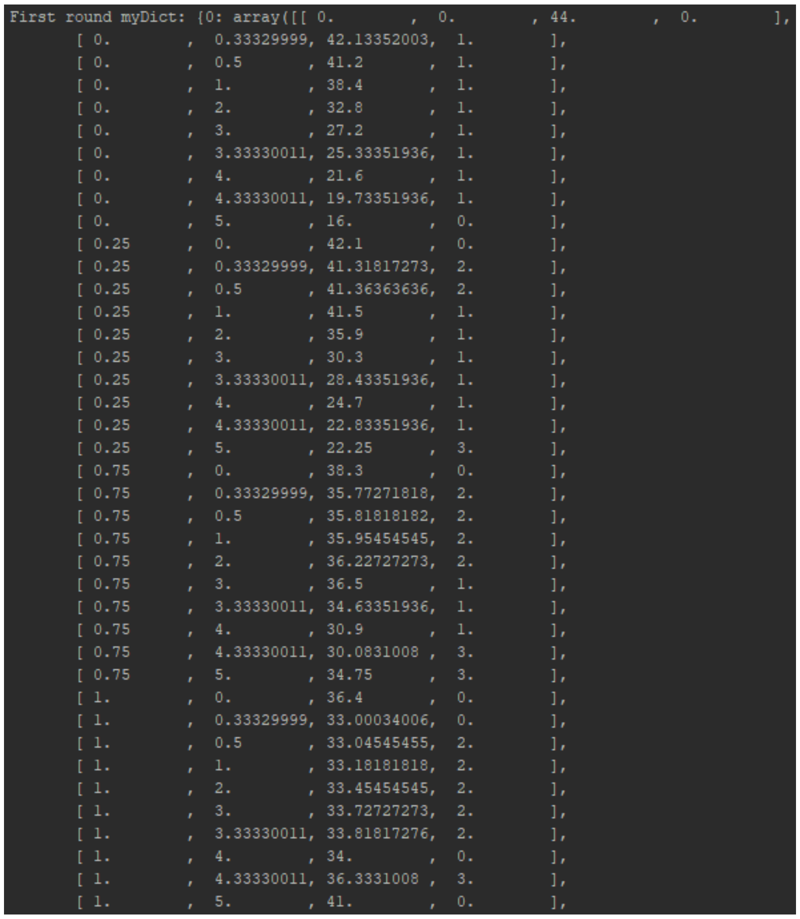
Input: Grid of points with all information for this iteration
Output: Mixed strategies for Alice and Bob in this iteration and visual output showing these values



5.3.5 Point Storing

We already mentioned a dictionary `myDict` storing points for each round. We also use `gridpoints` to get a list of all points with their information about height and destination from where they were created to see them all in one place. The difference between them is that `myDict` as a dictionary remembers the situation for each round separately, but `gridpoints` only all of the points after iterations have been done. The most useful besides `myDict` is `coords`, which `all_lines` uses to access all triangle information at one place and is used throughout the code. Bellow, you can see part of `myDict` after the `first_round` as the number of points is really high.

Input: Coordinates current for iteration
Output: New dictionary key with all values stored

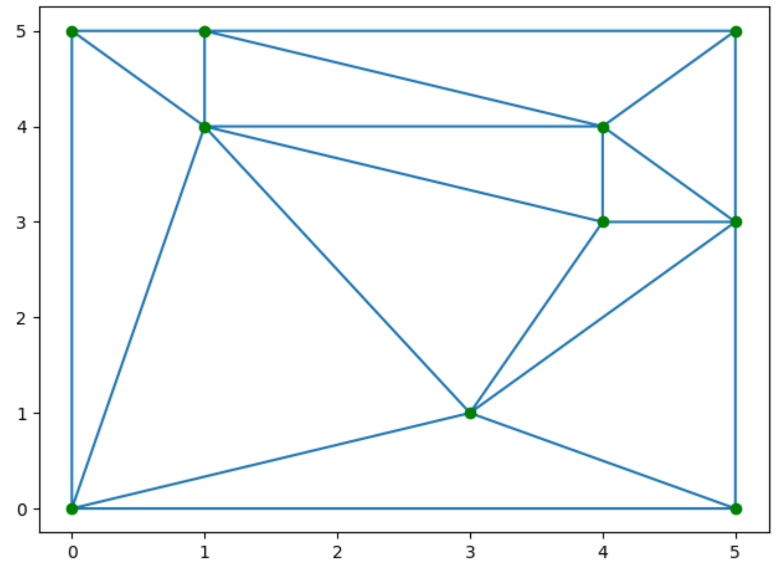


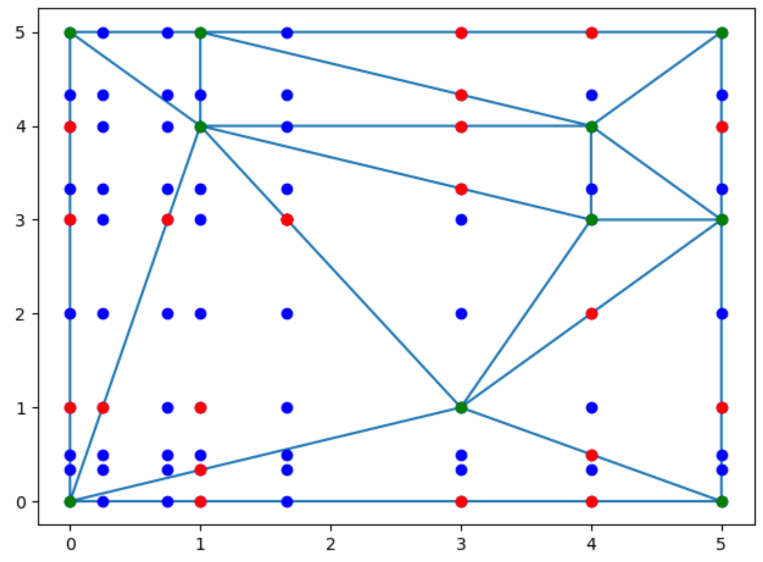
```
First round myDict: {0: array([[ 0.        ,  0.        , 44.        ,  0.        ],
       [ 0.        ,  0.33329999, 42.13352003,  1.        ],
       [ 0.        ,  0.5       , 41.2       ,  1.        ],
       [ 0.        ,  1.        , 38.4       ,  1.        ],
       [ 0.        ,  2.        , 32.8       ,  1.        ],
       [ 0.        ,  3.        , 27.2       ,  1.        ],
       [ 0.        ,  3.33330011, 25.33351936,  1.        ],
       [ 0.        ,  4.        , 21.6       ,  1.        ],
       [ 0.        ,  4.33330011, 19.73351936,  1.        ],
       [ 0.        ,  5.        , 16.        ,  0.        ],
       [ 0.25      ,  0.        , 42.1       ,  0.        ],
       [ 0.25      ,  0.33329999, 41.31817273,  2.        ],
       [ 0.25      ,  0.5       , 41.36363636,  2.        ],
       [ 0.25      ,  1.        , 41.5       ,  1.        ],
       [ 0.25      ,  2.        , 35.9       ,  1.        ],
       [ 0.25      ,  3.        , 30.3       ,  1.        ],
       [ 0.25      ,  3.33330011, 28.43351936,  1.        ],
       [ 0.25      ,  4.        , 24.7       ,  1.        ],
       [ 0.25      ,  4.33330011, 22.83351936,  1.        ],
       [ 0.25      ,  5.        , 22.25      ,  3.        ],
       [ 0.75      ,  0.        , 38.3       ,  0.        ],
       [ 0.75      ,  0.33329999, 35.77271818,  2.        ],
       [ 0.75      ,  0.5       , 35.81818182,  2.        ],
       [ 0.75      ,  1.        , 35.95454545,  2.        ],
       [ 0.75      ,  2.        , 36.22727273,  2.        ],
       [ 0.75      ,  3.        , 36.5       ,  1.        ],
       [ 0.75      ,  3.33330011, 34.63351936,  1.        ],
       [ 0.75      ,  4.        , 30.9       ,  1.        ],
       [ 0.75      ,  4.33330011, 30.0831008 ,  3.        ],
       [ 0.75      ,  5.        , 34.75      ,  3.        ],
       [ 1.        ,  0.        , 36.4       ,  0.        ],
       [ 1.        ,  0.33329999, 33.00034006,  0.        ],
       [ 1.        ,  0.5       , 33.04545455,  2.        ],
       [ 1.        ,  1.        , 33.18181818,  2.        ],
       [ 1.        ,  2.        , 33.45454545,  2.        ],
       [ 1.        ,  3.        , 33.72727273,  2.        ],
       [ 1.        ,  3.33330011, 33.81817276,  2.        ],
       [ 1.        ,  4.        , 34.        ,  0.        ],
       [ 1.        ,  4.33330011, 36.3331008 ,  3.        ],
       [ 1.        ,  5.        , 41.        ,  0.        ],
```

5.3.6 Visual Output

Now when all the steps are finished. The first thing to be drawn is a probability distribution over each player's possible strategies side by side (seen above). The second thing is to show points active in each round with a triangulated original picture in the background. Below are the first two rounds (original points in green, intersections from the first round in red, and points created by Cartesian product in blue).
Input: Triangulated space with a dictionary with points in each iteration

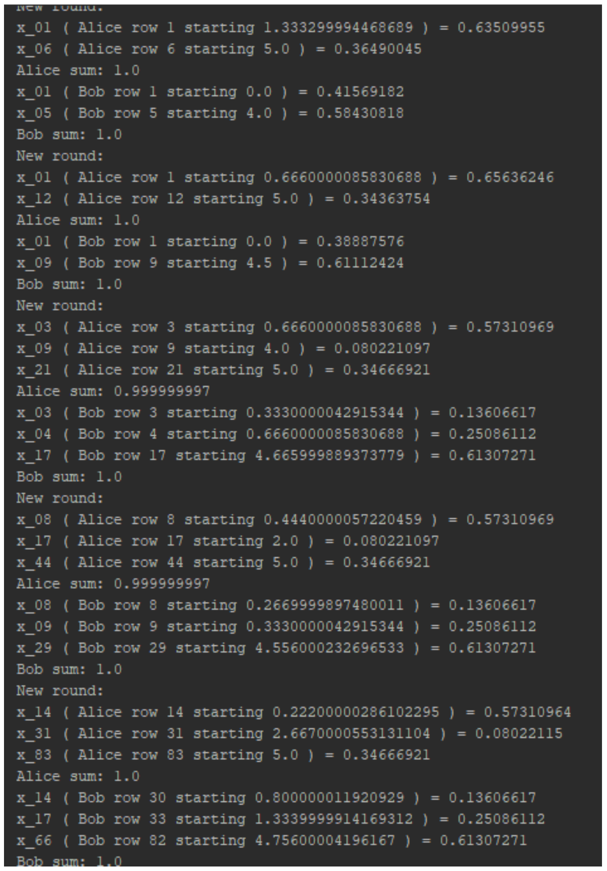
Output: Triangulated space with grid points for each iteration



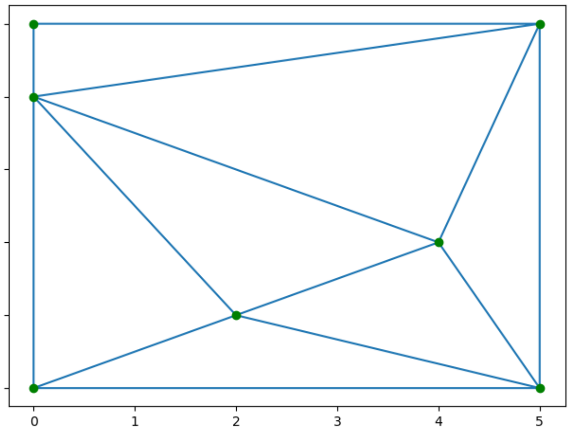


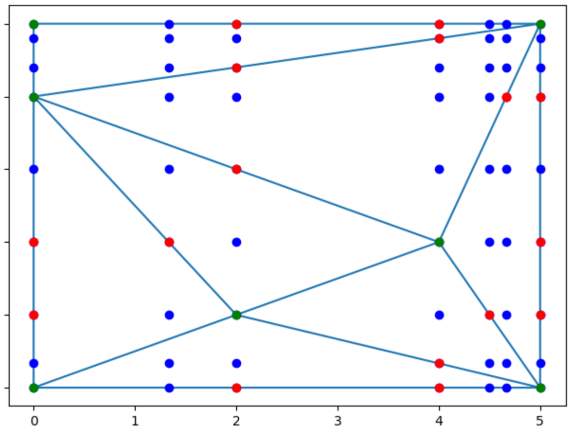
5.3.7 Other examples

The first example is used to verify the results in a detailed visualization above.

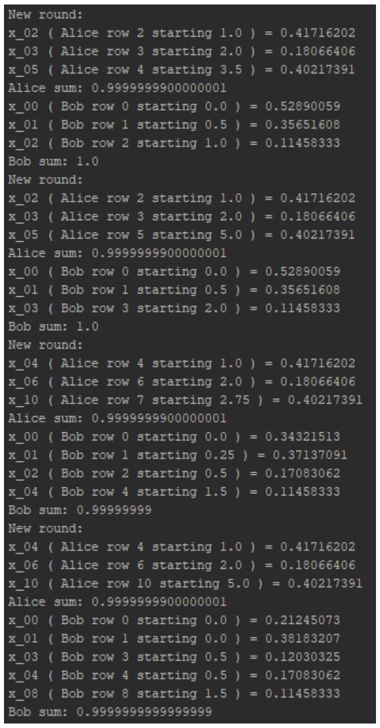


```
New round:
x_01 ( Alice row 1 starting 1.333299994468689 ) = 0.63509955
x_06 ( Alice row 6 starting 5.0 ) = 0.36490045
Alice sum: 1.0
x_01 ( Bob row 1 starting 0.0 ) = 0.41569182
x_05 ( Bob row 5 starting 4.0 ) = 0.58430818
Bob sum: 1.0
New round:
x_01 ( Alice row 1 starting 0.6660000085830688 ) = 0.65636246
x_12 ( Alice row 12 starting 5.0 ) = 0.34363754
Alice sum: 1.0
x_01 ( Bob row 1 starting 0.0 ) = 0.38887576
x_09 ( Bob row 9 starting 4.5 ) = 0.61112424
Bob sum: 1.0
New round:
x_03 ( Alice row 3 starting 0.6660000085830688 ) = 0.57310969
x_09 ( Alice row 9 starting 4.0 ) = 0.080221097
x_21 ( Alice row 21 starting 5.0 ) = 0.34666921
Alice sum: 0.999999997
x_03 ( Bob row 3 starting 0.3330000042915344 ) = 0.13606617
x_04 ( Bob row 4 starting 0.6660000085830688 ) = 0.25086112
x_17 ( Bob row 17 starting 4.665999889373779 ) = 0.61307271
Bob sum: 1.0
New round:
x_08 ( Alice row 8 starting 0.4440000057220459 ) = 0.57310969
x_17 ( Alice row 17 starting 2.0 ) = 0.080221097
x_44 ( Alice row 44 starting 5.0 ) = 0.34666921
Alice sum: 0.999999997
x_08 ( Bob row 8 starting 0.2669999897480011 ) = 0.13606617
x_09 ( Bob row 9 starting 0.3330000042915344 ) = 0.25086112
x_29 ( Bob row 29 starting 4.556000232696533 ) = 0.61307271
Bob sum: 1.0
New round:
x_14 ( Alice row 14 starting 0.22200000286102295 ) = 0.57310964
x_31 ( Alice row 31 starting 2.6670000553131104 ) = 0.08022115
x_83 ( Alice row 83 starting 5.0 ) = 0.34666921
Alice sum: 1.0
x_14 ( Bob row 30 starting 0.800000011920929 ) = 0.13606617
x_17 ( Bob row 33 starting 1.3339999914169312 ) = 0.25086112
x_66 ( Bob row 82 starting 4.75600004196167 ) = 0.61307271
Bob sum: 1.0
```

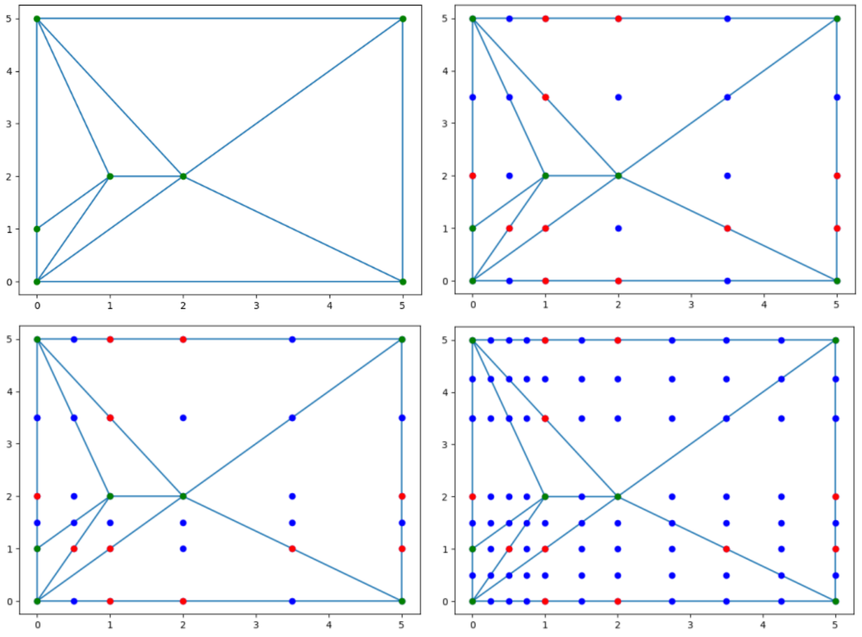


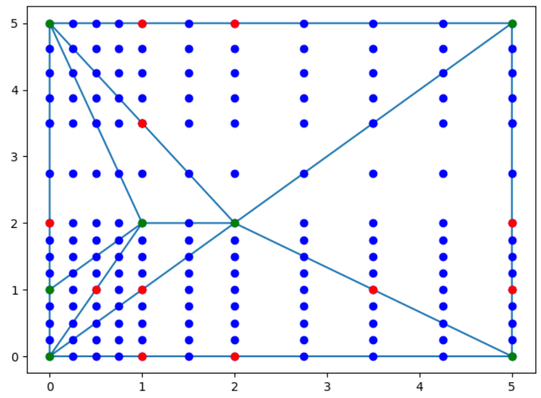


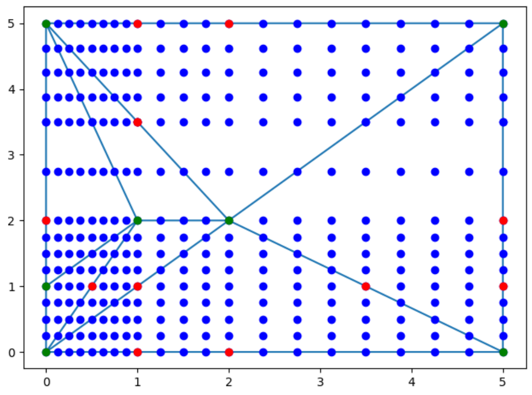
The second example is used to present all points in later iterations since usually such a large number of them is generated they become impossible to visualize.



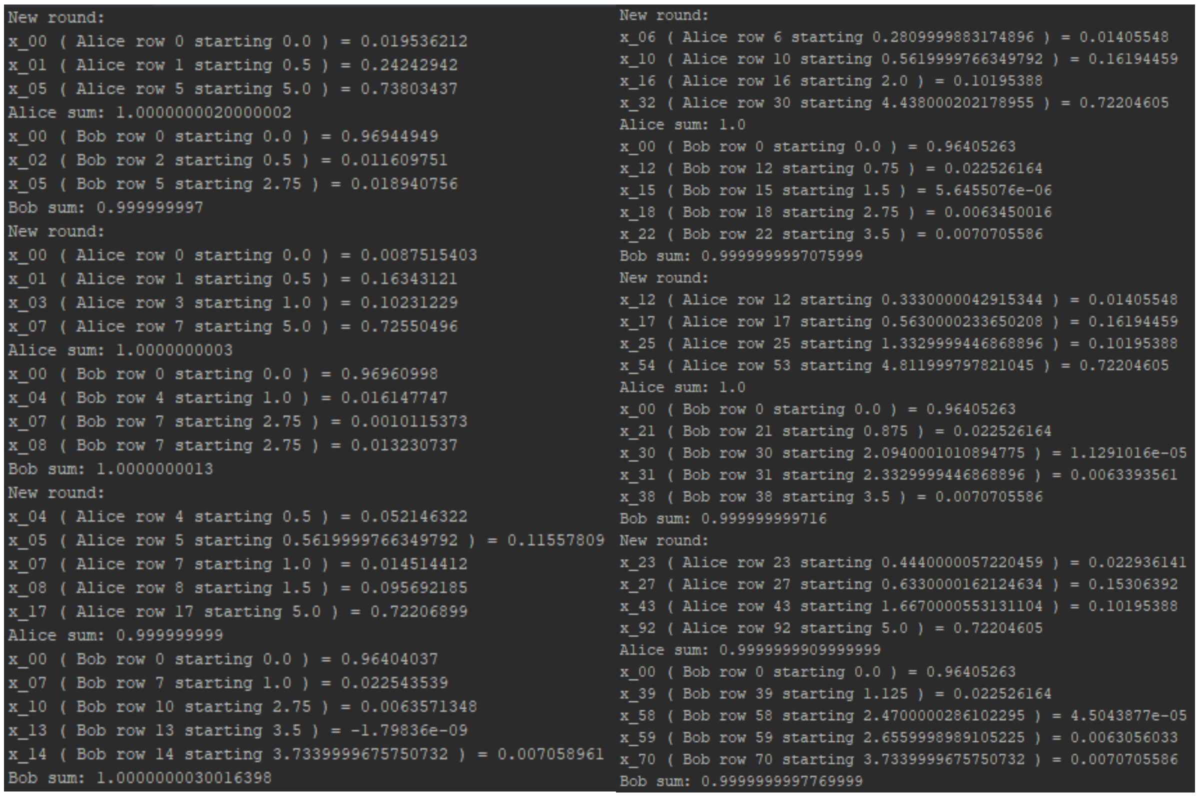
```
New round:
x_02 ( Alice row 2 starting 1.0 ) = 0.41716202
x_03 ( Alice row 3 starting 2.0 ) = 0.18066406
x_05 ( Alice row 4 starting 3.5 ) = 0.40217391
Alice sum: 0.9999999900000001
x_00 ( Bob row 0 starting 0.0 ) = 0.52890059
x_01 ( Bob row 1 starting 0.5 ) = 0.35651608
x_02 ( Bob row 2 starting 1.0 ) = 0.11458333
Bob sum: 1.0
New round:
x_02 ( Alice row 2 starting 1.0 ) = 0.41716202
x_03 ( Alice row 3 starting 2.0 ) = 0.18066406
x_05 ( Alice row 5 starting 5.0 ) = 0.40217391
Alice sum: 0.9999999900000001
x_00 ( Bob row 0 starting 0.0 ) = 0.52890059
x_01 ( Bob row 1 starting 0.5 ) = 0.35651608
x_03 ( Bob row 3 starting 2.0 ) = 0.11458333
Bob sum: 1.0
New round:
x_04 ( Alice row 4 starting 1.0 ) = 0.41716202
x_06 ( Alice row 6 starting 2.0 ) = 0.18066406
x_10 ( Alice row 7 starting 2.75 ) = 0.40217391
Alice sum: 0.9999999900000001
x_00 ( Bob row 0 starting 0.0 ) = 0.34321513
x_01 ( Bob row 1 starting 0.25 ) = 0.37137091
x_02 ( Bob row 2 starting 0.5 ) = 0.17083062
x_04 ( Bob row 4 starting 1.5 ) = 0.11458333
Bob sum: 0.99999999
New round:
x_04 ( Alice row 4 starting 1.0 ) = 0.41716202
x_06 ( Alice row 6 starting 2.0 ) = 0.18066406
x_10 ( Alice row 10 starting 5.0 ) = 0.40217391
Alice sum: 0.9999999900000001
x_00 ( Bob row 0 starting 0.0 ) = 0.21245073
x_01 ( Bob row 1 starting 0.0 ) = 0.38183207
x_03 ( Bob row 3 starting 0.5 ) = 0.12030325
x_04 ( Bob row 4 starting 0.5 ) = 0.17083062
x_08 ( Bob row 8 starting 1.5 ) = 0.11458333
Bob sum: 0.9999999999999999
```





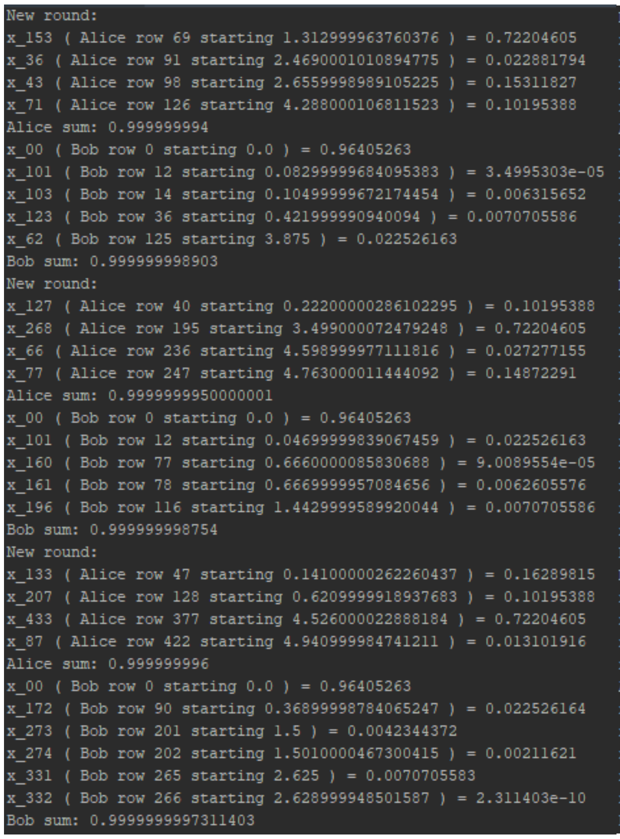


For the last example, we decided to let the algorithm run until our computer runs out of memory. Here we can observe some probabilities becoming so small algorithm starts putting minus before them. However, we see our results being stable.

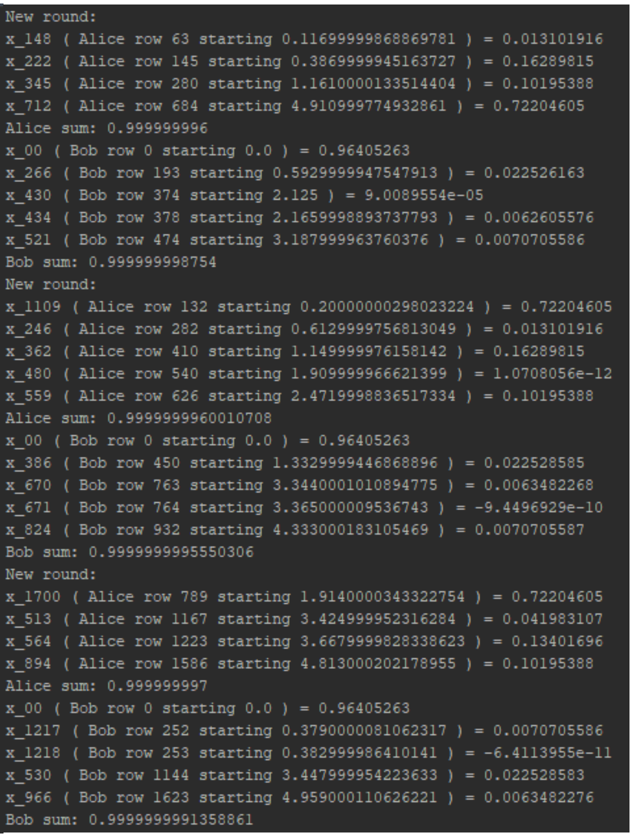


```
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.019536212
x_01 ( Alice row 1 starting 0.5 ) = 0.24242942
x_05 ( Alice row 5 starting 5.0 ) = 0.73803437
Alice sum: 1.0000000020000002
x_00 ( Bob row 0 starting 0.0 ) = 0.96944949
x_02 ( Bob row 2 starting 0.5 ) = 0.011609751
x_05 ( Bob row 5 starting 2.75 ) = 0.018940756
Bob sum: 0.999999997
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.0087515403
x_01 ( Alice row 1 starting 0.5 ) = 0.16343121
x_03 ( Alice row 3 starting 1.0 ) = 0.10231229
x_07 ( Alice row 7 starting 5.0 ) = 0.72550496
Alice sum: 1.0000000003
x_00 ( Bob row 0 starting 0.0 ) = 0.96960998
x_04 ( Bob row 4 starting 1.0 ) = 0.016147747
x_07 ( Bob row 7 starting 2.75 ) = 0.0010115373
x_08 ( Bob row 7 starting 2.75 ) = 0.013230737
Bob sum: 1.0000000013
New round:
x_04 ( Alice row 4 starting 0.5 ) = 0.052146322
x_05 ( Alice row 5 starting 0.5619999766349792 ) = 0.11557809
x_07 ( Alice row 7 starting 1.0 ) = 0.014514412
x_08 ( Alice row 8 starting 1.5 ) = 0.095692185
x_17 ( Alice row 17 starting 5.0 ) = 0.72206899
Alice sum: 0.999999999
x_00 ( Bob row 0 starting 0.0 ) = 0.96404037
x_07 ( Bob row 7 starting 1.0 ) = 0.022543539
x_10 ( Bob row 10 starting 2.75 ) = 0.0063571348
x_13 ( Bob row 13 starting 3.5 ) = -1.79836e-09
x_14 ( Bob row 14 starting 3.7339999675750732 ) = 0.007058961
Bob sum: 1.0000000030016398
```

```
New round:
x_06 ( Alice row 6 starting 0.2809999883174896 ) = 0.01405548
x_10 ( Alice row 10 starting 0.5619999766349792 ) = 0.16194459
x_16 ( Alice row 16 starting 2.0 ) = 0.10195388
x_32 ( Alice row 30 starting 4.438000202178955 ) = 0.72204605
Alice sum: 1.0
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_12 ( Bob row 12 starting 0.75 ) = 0.022526164
x_15 ( Bob row 15 starting 1.5 ) = 5.6455076e-06
x_18 ( Bob row 18 starting 2.75 ) = 0.0063450016
x_22 ( Bob row 22 starting 3.5 ) = 0.0070705586
Bob sum: 0.9999999997075999
New round:
x_12 ( Alice row 12 starting 0.3330000042915344 ) = 0.01405548
x_17 ( Alice row 17 starting 0.5630000233650208 ) = 0.16194459
x_25 ( Alice row 25 starting 1.3329999446868896 ) = 0.10195388
x_54 ( Alice row 53 starting 4.811999797821045 ) = 0.72204605
Alice sum: 1.0
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_21 ( Bob row 21 starting 0.875 ) = 0.022526164
x_30 ( Bob row 30 starting 2.0940001010894775 ) = 1.1291016e-05
x_31 ( Bob row 31 starting 2.3329999446868896 ) = 0.0063393561
x_38 ( Bob row 38 starting 3.5 ) = 0.0070705586
Bob sum: 0.999999999716
New round:
x_23 ( Alice row 23 starting 0.4440000057220459 ) = 0.022936141
x_27 ( Alice row 27 starting 0.6330000162124634 ) = 0.15306392
x_43 ( Alice row 43 starting 1.6670000553131104 ) = 0.10195388
x_92 ( Alice row 92 starting 5.0 ) = 0.72204605
Alice sum: 0.9999999909999999
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_39 ( Bob row 39 starting 1.125 ) = 0.022526164
x_58 ( Bob row 58 starting 2.4700000286102295 ) = 4.5043877e-05
x_59 ( Bob row 59 starting 2.6559998989105225 ) = 0.0063056033
x_70 ( Bob row 70 starting 3.7339999675750732 ) = 0.0070705586
Bob sum: 0.9999999997769999
```



```
New round:
x_153 ( Alice row 69 starting 1.312999963760376 ) = 0.72204605
x_36 ( Alice row 91 starting 2.4690001010894775 ) = 0.022881794
x_43 ( Alice row 98 starting 2.6559998989105225 ) = 0.15311827
x_71 ( Alice row 126 starting 4.288000106811523 ) = 0.10195388
Alice sum: 0.999999994
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_101 ( Bob row 12 starting 0.08299999684095383 ) = 3.4995303e-05
x_103 ( Bob row 14 starting 0.10499999672174454 ) = 0.006315652
x_123 ( Bob row 36 starting 0.421999990940094 ) = 0.0070705586
x_62 ( Bob row 125 starting 3.875 ) = 0.022526163
Bob sum: 0.999999998903
New round:
x_127 ( Alice row 40 starting 0.22200000286102295 ) = 0.10195388
x_268 ( Alice row 195 starting 3.499000072479248 ) = 0.72204605
x_66 ( Alice row 236 starting 4.598999977111816 ) = 0.027277155
x_77 ( Alice row 247 starting 4.763000011444092 ) = 0.14872291
Alice sum: 0.9999999950000001
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_101 ( Bob row 12 starting 0.04699999839067459 ) = 0.022526163
x_160 ( Bob row 77 starting 0.6660000085830688 ) = 9.0089554e-05
x_161 ( Bob row 78 starting 0.6669999957084656 ) = 0.0062605576
x_196 ( Bob row 116 starting 1.4429999589920044 ) = 0.0070705586
Bob sum: 0.999999998754
New round:
x_133 ( Alice row 47 starting 0.14100000262260437 ) = 0.16289815
x_207 ( Alice row 128 starting 0.6209999918937683 ) = 0.10195388
x_433 ( Alice row 377 starting 4.526000022888184 ) = 0.72204605
x_87 ( Alice row 422 starting 4.940999984741211 ) = 0.013101916
Alice sum: 0.999999996
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_172 ( Bob row 90 starting 0.36899998784065247 ) = 0.022526164
x_273 ( Bob row 201 starting 1.5 ) = 0.0042344372
x_274 ( Bob row 202 starting 1.5010000467300415 ) = 0.00211621
x_331 ( Bob row 265 starting 2.625 ) = 0.0070705583
x_332 ( Bob row 266 starting 2.628999948501587 ) = 2.311403e-10
Bob sum: 0.9999999997311403
```



```
New round:
x_148 ( Alice row 63 starting 0.11699999868869781 ) = 0.013101916
x_222 ( Alice row 145 starting 0.3869999945163727 ) = 0.16289815
x_345 ( Alice row 280 starting 1.1610000133514404 ) = 0.10195388
x_712 ( Alice row 684 starting 4.910999774932861 ) = 0.72204605
Alice sum: 0.999999996
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_266 ( Bob row 193 starting 0.5929999947547913 ) = 0.022526163
x_430 ( Bob row 374 starting 2.125 ) = 9.0089554e-05
x_434 ( Bob row 378 starting 2.1659998893737793 ) = 0.0062605576
x_521 ( Bob row 474 starting 3.187999963760376 ) = 0.0070705586
Bob sum: 0.999999998754
New round:
x_1109 ( Alice row 132 starting 0.20000000298023224 ) = 0.72204605
x_246 ( Alice row 282 starting 0.6129999756813049 ) = 0.013101916
x_362 ( Alice row 410 starting 1.149999976158142 ) = 0.16289815
x_480 ( Alice row 540 starting 1.909999966621399 ) = 1.0708056e-12
x_559 ( Alice row 626 starting 2.4719998836517334 ) = 0.10195388
Alice sum: 0.9999999960010708
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_386 ( Bob row 450 starting 1.3329999446868896 ) = 0.022528585
x_670 ( Bob row 763 starting 3.3440001010894775 ) = 0.0063482268
x_671 ( Bob row 764 starting 3.365000009536743 ) = -9.4496929e-10
x_824 ( Bob row 932 starting 4.333000183105469 ) = 0.0070705587
Bob sum: 0.9999999995550306
New round:
x_1700 ( Alice row 789 starting 1.9140000343322754 ) = 0.72204605
x_513 ( Alice row 1167 starting 3.424999952316284 ) = 0.041983107
x_564 ( Alice row 1223 starting 3.6679999828338623 ) = 0.13401696
x_894 ( Alice row 1586 starting 4.813000202178955 ) = 0.10195388
Alice sum: 0.999999997
x_00 ( Bob row 0 starting 0.0 ) = 0.96405263
x_1217 ( Bob row 252 starting 0.3790000081062317 ) = 0.0070705586
x_1218 ( Bob row 253 starting 0.382999986410141 ) = -6.4113955e-11
x_530 ( Bob row 1144 starting 3.447999954223633 ) = 0.022528583
x_966 ( Bob row 1623 starting 4.959000110626221 ) = 0.0063482276
Bob sum: 0.9999999991358861
```

6 Conclusion

Our goal was to experimentally verify whether infinite two-player zero-sum games with piecewise affine payoff functions have finite equilibria. After introducing the problem, we started explaining the backgrounds of zero-sum games as we began to understand how to approach it. The experiment is based on the generation of random triangulations with their affine functions. It is needed to approximate such functions over a finite grid and determine the respective finite game's equilibrium. The approximation of Nash equilibria for the game is made using linear programming. Before the start of the experiment, it was known that polynomial games hold the property of finite equilibria. Since zero-sum games belong to the same family and have similar properties as polynomial ones, we strengthened the belief equilibria finiteness would hold for them too.

The experiment is set on a square $[0,1] \times [0,1]$ holding all strategy spaces for our players. To represent the square, a two-dimensional array was created in Python. Its size is determined by the user's input, specifically by `kernel_sizer`, setting each box's size to space out the $[0,1]$ interval evenly. All randomization and most other mathematical instruments are done using the NumPy library, which is widely used in similar cases. The algorithm uses `points_number` to randomly assign points to a place with their generated value of interpolated functions or heights. Then the algorithm continues by creating a triangulation from these randomly generated points inside the square. Triangles created in the process represent affine functions in the game with values in the form of heights. To triangulate such space, it was needed to choose a library for creating triangulations. Our research led us to use the Delaunay function from the scipy library. With triangles now in place, we had to count equations of affine functions to represent them to find heights for new points generated in each iteration. The mentioned iteration starts with a search for intersections, which is done by checking every point present to see if it crosses any line created by the triangulation mentioned above. A new point is added to the pool of points if it has not been found previously. Once all possible intersections have been checked, we create a grid from all now available points using their unique x, y coordinates' Cartesian product. For all points in the grid, there is a specific height value. To assign it to them, each point checks in what triangle it lies and uses this triangle's function equation plugs its coordinates to find the height. After it is done for all of them, we put grid height values into a matrix to represent strategies in the game and solve it for our two players using linear programming. This all happens in one iteration. The user-specified the number of them. We expect it to converge with a result for our game while adding the rest of the iterations. Here each new iteration adds a countable number of points (later used as strategies), which enabled us to use theory based on finite games even when our game ultimately approaches infinity.

For the games tested, we monitored the player's probability distributions stabilizing with increasing strategy sets. This was needed to say that if we observed a finite equilibrium of a zero-sum game, then we could use the initial grid to represent a game, as it is the case for polynomial games, and save computational power. Looking back at the experiment, we learned that Python might not be the greatest programming language for this calculation type. Furthermore, this is due to many libraries' encounter for linear programming, which could not calculate needed probability distributions, showed inconsistent or wrong results before using PuLP for the algorithm. For these reasons, Matlab will be preferred in future research of similar problems since we needed to investigate which library had the best implementation to show the required results.

7 Resources

1. PARRILO P.: *Polynomial games and sum of squares optimization. In Decision and Control*. 2006. 45th IEEE Conference on, pages 2855–2860. [ref. 12-09-2020].
2. HANAPPI H.: *The Neumann-Morgenstern Project – Game Theory as a Formal Language for the Social Sciences*. 2013. Game Theory Relaunched, Hardy Hanappi, IntechOpen, DOI: 10.5772/56106. [ref. 12-09-2020]. Available from: https://www.intechopen.com/books/game-theory-relaunched/the-neumann-morgenstern-project-game-theory-as-a-formal-language-for-the-social-sciences
3. NISAN N.; ROUGHGARDEN T.; TARDOS E.; and VAZIRANI V. V.: A*lgorithmic game theory*. 2007. Cambridge University Press Cambridge. [ref. 12-09-2020].
4. SHOHAM Y.; and LEYTON-BROWN K.: *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. 2008. Cambridge University Press. [ref. 12-09-2020].
5. LEMKE C.; and HOWSON J.: *Equilibrium points of bimatrix games. 1964. Journal of the Society for Industrial and Applied Mathematics*, pages 413–423. [ref. 12-09-2020].
6. GLICKSBERG I. L.: *A further generalization of the Kakutani fixed point theorem, with application to Nash equilibrium points*. 1952. Proceedings of the American Mathematical Society, pages 170–174. [ref. 12-09-2020].
7. DRESHER S. K. W.; and SHAPLEY L. S.: *Polynomial games. In H. W. Kuhn and A. W. Tucker, editors, Contributions to the Theory of Games, volume I of Annals of Mathematics Studies*. 1950. Princeton University Press. pages 161–180. [ref. 12-09-2020].
8. STEIN D.N.; OZDAGLAR A.; and PARRILO A.P.: *Separable and Low-Rank Continuous Games* [online]. [ref. 11-09-2020]. Accessed from: http://www.mit.edu/~nstein/documents/SeparableGamesCDC2006Extended.pdf
9. SHOHAM, Y. and LEYTON-BROWN, K.: *MULTIAGENT SYSTEMS Algorithmic, Game-Theoretic, and Logical Foundations* [online]. 2009. Cambridge: Cambridge University Press. [ref. 25-07-2020]. Accessed from: http://www.masfoundations.org/mas.pdf
10. LEVIN J.:*Choice under Uncertainty* [online]. 2006. [ref. 11-09-2020]. Accessed from: http://web.stanford.edu/~jdlevin/Econ%20202/Uncertainty.pdf
11. BOARD S.:Choice under Uncertainty [online]. 2009. [ref. 11-09-2020]. Accessed from: http://www.econ.ucla.edu/sboard/teaching/econ11_09/econ11_09_lecture2.pdf
12. *Linear Programming Notes IX: Two-Person Zero-Sum Game Theory* [online]. [ref. 11-09-2020]. Accessed from: https://econweb.ucsd.edu/~jsobel/172aw02/notes9.pdf
13. *Zero-Sum (and Constant Sum) Games* [online]. [ref. 11-09-2020]. Accessed from: https://www3.nd.edu/~apilking/math10170/information/Lectures/14%20Zero%20Sum%20Games.pdf
14. KAKKAD, V.; SHAH, H.; PATEL R. and DOSHI N.: *A Comparative study of applications of Game Theory in Cyber Security and Cloud Computing* [online]. 2019. Pandit Deendayal Petroleum University, Gandhinagar, India. [ref. 26-07-2020]. Accessed from: https://www.sciencedirect.com/science/article/pii/S1877050919310130, pages 681-684
15. MELL P.; and GRANCE T.: *The NIST Definition of Cloud Computing* [online]. 2011. National Institute of Standards and Technology. [ref. 29-11-2020] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

16. YUAN W.; YONGJUN W.; JING L.; ZHIJIAN H.; and PEIDAI X.: *A Survey of Game Theoretic Methods for Cyber Security*. 2016. IEEE First International Conference on Data Science in Cyberspace. [ref. 12-09-2020].
17. BURCH N.: *Time and Space: Why Imperfect Information Games are Hard*. 2017. Ph.D. Dissertation, University of Alberta. [ref. 12-09-2020].
18. KÜNSEMOELLER J.; and HOLGER K.: *A game-theoretical approach to the benefits of cloud computing*. 2012. Economics of Grids, Clouds, Systems, and Services. Springer Berlin Heidelberg. pages 148-160. [ref. 12-09-2020].
19. AMADI CH.; EZE U.; and IKERIONWU CH.: *Game Theory Basics and Its Application in Cyber Security*. 2017. Advances in Wireless Communications and Networks. pages 45-49. [ref. 12-09-2020].
20. SHIVA S.; SANKARDAS R.; and DIPANKAR D.: *Game theory for cyber security*. 2010. ACM Comput. Surv. [ref. 12-09-2020].
21. HU J.; and P. WELLMAN M,: *Nash Q-Learning for General-Sum Stochastic Games*. 2003. Journal of Machine Learning Research, pages 1039-1069. [ref. 12-09-2020].
22. BING S.; HUANG; YALONG; WANG; JINWEN; and SHENGWU X.: *A Game-Theoretic Analysis of Pricing Strategies for Competing Cloud Platforms*. 2016. pages 653-660.
23. OZDAGLAR A.: *Continuous and Discontinuous Games* [online]. [ref. 11-09-2020]. Accessed from: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-254-game-theory-with-engineering-applications-spring-2010/lecture-notes/MIT6_254S10_lec06b.pdf
24. KOLLER D.; MEGIDDO N.; and STENGEL von B.: *Fast algorithms for finding randomized strategies in game trees*. 1994. In Proc. 26thSTOC, pages 750-759. [ref. 12-09-2020].
25. MILTERSEN P.; and SØRENSEN T.: *Fast algorithms for finding proper strategies in game trees*. 2008. In Proc. 19th SODA, pages 874-883. [ref. 12-09-2020].
26. WIMPEE J.: *Finding Nash equilibria in two-player, zero sum games. 2008. Computer Science Graduate and UndergraduateStudent Scholarship. 3*. [online]. [ref. 11-09-2020]. Accessed from: https://cedar.wwu.edu/cgi/viewcontent.cgi?article=1002&context=computerscience_stupubs
27. DEVADOSS L.S.; and O'ROURKE J.: *Discrete and Computational Geometry*. *Princeton University Press Princeton and Oxford*. [ref. 11-09-2020].
28. NISAN N.; ROUGHGARDEN T.; TARDOS E.; and VAZIRANI V.V.: *Algorithmic game theory*. 2007. Cambridge University Press Cambridge. [ref. 29-11-2020]
29. NASH j.: *Non-cooperative games*. 1951. Ann. Math, pages 286-295. [ref. 29-11-2020]
30. RADONS M.: *A note on surjectivity of piecewise affine mappings*. 2018. [ref. 29-11-2020]. Accessed from: https://arxiv.org/pdf/1707.08786.pdf