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

BACHELOR THESIS

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

2020 Stanislav Kubis

Czech Technical University in Prague
Faculty of Electrical Engineering

BACHELOR THESIS

Study program : Open Informatics – Computer and Information Science

Topic: Games with Piecewise Affine Utility Functions

Author: Stanislav Kubis
Year: Third
Academic year: 2019/2020
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: **14.08.2020**

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

Date of assignment receipt   Student’s signature

Sincere thank you belongs to the leader 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

Keywords

Game Theory, Piecewise Affine Games, Zero-Sum Games, Nash Equilibrium, Two-Player Games

Abstract

The goal of the paper is to experimentally verify whether infinite two-player zero-sum games with payoff functions, present in a form of piecewise affine functions have finite equilibria. Piecewise-affine functions are defined over a continuous domain, in our case our triangulated square describing continuous subdivision of itself, where there is an affine function for each triangle. In our experiment, we will focus on computing an equilibrium based on linear programming. The experiment is based on the generation of random triangulations with piecewise affine functions arising from them with an approximation of such functions over a grid and calculating the equilibrium of the respective finite game. To demonstrate it we create a square $[0,1] \times [0,1]$ where we randomly insert points along axis decided by the user and triangulate them. Initial points have random values as interpolated function, we will call heights. After this base state, we make an iteration consisting of finding new points by intersecting initial points with lines from triangulation. New points have heights computed with interpolated functions of triangles from the triangulation. Next, we create a grid from all existing points. The zero-sum game will use the grid point's heights as entries. The crucial conjecture is that after some number of iterations a finite equilibrium from a zero-sum game will arise.

Klíčová slova

Teorie her, Po částech afinní hry, Hry s nulovým součtem, Nashovo equilibrium, Hry o dvou hráčích

Anotace

Cílem této práce je experimentálně ověřit jestli nekonečná hry o dvou hráčích s nulovým součtem s výplatní funkcí, která je přítomna ve formě po částech afinní funkce má konečné equilibrium. Po částech afinní funkce jsou definované přes kontinuální doménu, v našem případě triangulovaný čtverec popisující kontinuální subdivizi samo sebe, kde je affinní funkce pro každý trojúhelník. V našem experimentu se zaměříme na počítání equilibria založeném na lineárním programování. Experiment je založen na generování náhodných triangulací s počástech affinními funkcemi z nich vytvořených a aproximaci těchto funkcí přes grid pro výpočet equilibria té konečné hry. Pro demonstraci uděláme čtverec $[0,1] \times [0,1]$ kde máme náhodně dané body podél dimenzí určených uživatelem a triangulujeme je. Původní body mají náhodné hodnoty jako interpolované funkce, kterým budeme říkat výšky. Po tomto základním stavu vytvoříme iteraci obsahující nalezení nových bodů z průsečíků původních bodu a úseček z triangulace. Nové body mají výšky spočtené z interpolovaných funkcí trojúhelníku triangulace. Dále vytvoříme grid ze všech existujících bodů. Hra s nulovým součtem použije výšky z gridu jako své vstupy. Kritická myšlenka je, že po několika iteracích konečné equilibrium vznikne ze hry s nulovým součtem.

Table of Contents

1 Introduction

It this thesis, we firstly go into some properties normal form games as a building stone for other, more specific, types of games in game theory. Next, we look into zero-sum games as a main topic in our project and talk about some other uses in computer science. When solving zero-sum games, we introduce Nash equilibrium in them, and describe it to realize, what we are searching for. Then, we describe continuous games as to gain more detailed information about what are dealing with and some other specific types we can meet when solving a similar task. Lastly in our theory, we talk about possible ways in approaching this setup. After all this theoretical knowledge, we are able to comment on our code and its specific steps along with examples along the way. Then we summarize all of the above in our conclusion.

2 Properties of Payoff Mixed Strategies Matrix

At first, we want to look at our games in general, where we build a basic theoretical structure for latter usage.

2.1 Normal form game

A finite game of n-players is in normal-form when we can form it as a tuple (N, A, u), where $N = \{1, \dots n\}$ is a set of players, $A = A_1 \times \dots \times A_n$ are all actions (strategies) with A_i being a finite set of those available to player i and $a = (a_1 \dots a_n) \in A$ is called strategy profile. Lastly, u where $u_i : A \to R$ is for each profile $a \in A$ utility (payoff), which describes a utility of player i [1, p.56].

2.2 Mixed strategies

A type of strategy, which may not seem obvious, is called a mixed strategy. For each player it consists of randomizing over a set of available options according to some probability distribution. This can be formally 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, where $p_i \in S_i$ is 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 [1, p.60].

2.3 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. Furthermore, its focus is to understand how such preferences change when a player deals with uncertainty about alternatives it will receive [1, p.47].

2.4 Utility functions

When we talk about utility functions, as will be done much later in the text, we will be trying to make a specific assumption that our player has desires about how to behave, which are consistent with utility theory mentioned previously [1, p.56].

2.5 Expected utility

As we start, we calculate the probability for each strategy in our set of strategies, from which we measure the average payoff among them all 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 [1, p.60]

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

3 Zero-sum Games

3.1 Description

When we talk about zero-sum games, we have a matrix game with matrix M. Values in M are called utilities.

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

The common example uses two players. We will call them A, as in Alice, and B as in Bob. One of which uses row and other columns as their respective set of strategies. There is a finite number of them I and J. The chosen one is called a play. Players choose what they play at the same time. Moreover, one picks $i \in I$, the other $j \in J$. Value m_{ij} is called a gain for player A, which is at the ame time a loss of player B. Specifically,

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

Hence the name zero-sum games. A rational player $i \in N$ chooses a strategy that maximizes u_i gain [1, p.56].

3.2 The Idea of Solving

Players will always choose the best strategies among the worst possible variants. One picks a strategy, which maximizes his payoff taking into consideration the fact that others act in the same way. Let us consider the following matrix.

$$M = \begin{bmatrix} 8 & 1 & -3 \\ 6 & 4 & 5 \\ 0 & 2 & 12 \end{bmatrix}$$

The row minima are $-3, 4, 0$ and the columns maxima are $8, 4, 12$. We can observe that row two and columns of the same index have the same value being 2. So $(2, 2)$ is a saddle point with a payoff $(4, -4)$[1].

3.3 Uses

There are two big areas, where there is a possible usage of zero-sum games in computer science field of study. They use Nash Equilibria in their models. Firstly cloud computing, we can set players as a client, who wants to purchase the service, and cloud provider, who owns the cloud. Client decides on whether to buy the service or not based of off transparency provided by the provider or not based on Nash Equilibria. Secondly cyber security, where there is a unique approach to the scenario where hacker is one player and system administrators, who defend a system, are the other. When comparing these two scenarios cloud computing does not possess the needed quantity of providers and clients for making the models more scalable for application in practice. On the other hand, in cyber security there exists an uncertainty which makes it impossible to be presicely quantified in these models [2].

[1]https://bcourses.berkeley.edu/courses/1454200/files/69611157/download?verifier=aGpeJ5AgTbKsnB3Sa8cX16sKDKSG1pK0fbCFKVHH pages 8-12

4 Nash Equilibrium of a PA Games is Located at Vertices

4.1 Nash equilibrium

A Nash equilibrium is a set of strategies, one for each player i, where none has the incentive to deviate, since it would result in a loss of ones payoff. It can be written formally as $s = (s_1 \dots s_n)$, where for every player s is s_i best response to s_{-i}. With best response we are pointing out that if one player was to choose another response he would get a lower payoff compared to the one received with best response [1, p.62].

4.2 ε-Nash

A type of Nash equilibria is called ε-Nash. It shows that for an area surrounding an existing equilibria a result will be the same. Formally, for any fix ε larger than zero a strategy profile $s = (s_1 \dots s_n)$ is an ε-Nash equilibrium if, for all players i and for all strategies $s'_i \neq s_i$
, $u_i(s_i, s_{-i}) \geq u_i(s_i, s_{-i}) - \varepsilon$ [1, p.85].

4.3 Location at vertices

An equilibrium strategy of a player is a vertex of his best response polyhedron, which can also be describes as a convex combination of these vertices. Therefore, in two-player game it can be found by cycling through combinations of these best response vertices. Now, we know that equilibria will be found on these vertices, which are represented in our program as points. Lexicographic reverse search will lead us back to the original points[2]. We ask ourselves, what brought us to this decision and do it, until we are back at the beginning [3].

4.4 Linear program

Our problem, zero-sum game, is the easiest to solve using Nash equilibrium. It can be expressed as a linear program (LP), which 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 $U^*{}_i$ be the expected utility for player i in equilibrium, also known as the value of the game. 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 0. The min-max theorem (in Section 3.4.1 and Theorem 3.4.4. in masfoundation book) tells us that our expected utility remains constant in all equilibria and that it is the same as the value player 1 achieves under a min-max strategy by player 2. Using this, we can construct the linear program as follows.

[2] https://faculty.coe.drexel.edu/jwalsh/JayantLRS.pdf

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

With this, we are now going to make a dual program, by transforming this minimization [1, p.89-90].

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

5 Continuous games

We consider strategic games in which players may have infinitely many pure strategies. A pure strategy is a strategy, where a player selects a single action to play. Specifically, we are counting the real valued interval $[0, 1]$ as a strategy space.

5.1 Why is it hard?

Let us look at some properties that make continuous games hard to solve. First of all, is the fact global minimization/maximization of a polynomial is hard. These optimization problems are typically non-convex and highly nonlinear. Complexity is usually non-deterministic polynomial-time hardness, even for special cases such as maximizing a quadratic form in binary variables. Finding a solution is difficult for its complexity. It is a combination of heuristics and insight into the special structure of the game. Lastly, we have different types of games, where some have a special equilibrium. Convex-concave games, where the first player minimizes and other maximizes, have a goal of finding a saddle-point. Games of timing, where players start at time zero and the probability of their success increases with time with known probabilities at given times. We also have games with bell-shaped utility functions or invariants under symmetries[3].

5.2 Selected families of continuous games have special equilibria

5.2.1 Convex/concave games

Convex-concave games are built on a similar principle. They operate on one player minimizing and other maximizing what is here called payments. It is an 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 get $f(u, v)$ Lastly, a solution of 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 want to deviate, since it would only worsen their 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 convex function of u, and for each u, $f(u, v)$ to be concave function of v. When f is differentiable our saddle-point will be characterized by a gradient of (u^*, v^*) equal to 0, which can formally be noted as $\nabla(u^*, v^*) = 0$ [4].

5.2.2 Games of timing

When talking about games of timing we set a stage in a game which happens in a $[0, 1]$ windows, where each point in it is a specific moment in time. As an example, we imagine a game, where two player each have a basketball hoop and a ball. It slowly approaches them and therefore increases the probability of one player making a hoop. Players do not know, when the other one shot, but if both make it, the one who did it earlier gets a point. It is not a game of zero-sum since one's point do not take away from the other [5].

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

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

Firsly, we want to introduce piecewise-affine game. A strategic zero-sum gqme is piecewise affine, if strategy sets are $[0,1]$ and $u\colon [0,1]^2 \rightarrow R$ is a piecewise affine function. What are piecewise functions? They are function which consist of more that one function, either continuous or non-continuous. When these piecewise functions are affine, what this means is that the collections is created from separate affine functions. When we have a piecewise game, we need to think about a way that would represent our pieces correctly. Each piece can then be represented by its function since it is as a regular single three-dimensional function would be. If we have a way to represent each piece with some precision, we will keep it throughout the whole game. What was an impossible task is now uses splitting the problem into small parts and solve each one and put them back together to solve the big problem? We will describe two options for solving such a game [6, p.79-80].

6.1 Domains of linearity are polyhedral

Convex polyhedron is an intersection of finitely many halfspaces. In our experiments, we have a square of size $[0,1] \times [0,1]$ where we have random points with random heights placed. Therefore, polyhedra could be used to connect our vertices. The advantage of this shape is the fact it binds together three or more points. Therefore, if it makes sense it might be able to save some computation, while it does not always need to use just three points like the second option we will describe [6, p.156-160].

6.2 Triangulations

When we want to describe a triangulation for a set of points P. We subdivide these points into maximum possible non-intersecting lines, where vertices of these lines are points from P. Maximum describes the fact that for every other possible connection of points from P an intersection in formed with other already existing lines [6, p.59-80].

7 Code Explanation

7.1 Summary

The algorithm has two main parts. The first part creates a plane with triangulated points. The second part uses these points to find intersections they have with lines from triangulation and points creates a grid, which is used in solving our two-player Zero-Sum Game. There are pictures from important steps made with temporary prints in the code. The program takes following input of four parameters:

`points_number` – Describes how many points you want to have triangulated. Note that 4 are used for corners. Therefore, if chosen a lower number, the program sets the number to 4.

`heights_number` – This number changes the height of our points. Note that height must be greater than 0, otherwise changed to 0 by default.

`kernel_sizer` – We have a Matrix 1×1. Furthermore, there can be points anywhere in the interval of <0,1>. `Kernel_sizer` makes boxes in between the numbers 0 and 1, therefore this number needs to be modus of 1, otherwise changed to the default value of 0.2.

`steps` – Lastly, we use steps to indicate how many iterations we want to calculate. Since we want our program to get some results is it best to choose a value of 2 or greater. The default value is set at 2.

For the first script, 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 box border

Second script 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 box border with steps

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

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

Along with our printable outputs we show a triangulated space and intersection points for each iteration and probability distribution over mixed strategies of players. As an example, we follow a random generation of 5 points, with `kernel_sizer` set to 0.2 and `height_number` set to 50 to get rid of fractals or at least mitigate them.

7.2 Triangulation Part

7.2.1 Square creation

The function to complete the first part is called `pa_games_watch`. Firstly, to avoid working with fractals we use `kernel_sizer` (now called `kernel_size`) to calculate how many boxes there are between 0 and 1. For example with 0.2 we have 5 boxes, so we create 0,1,2,3,4,5 coordinates.

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

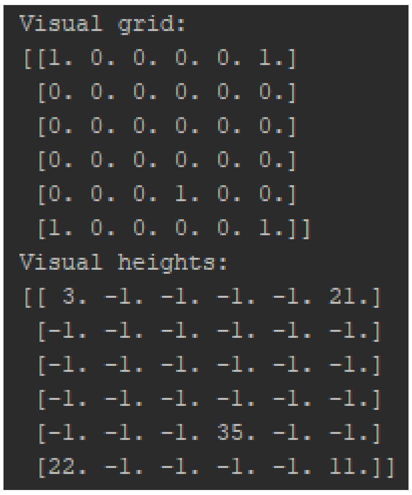
Output: Square [0,1] × [0,1] with size 6 × 6 (corresponding to `1 / kernel_size + 1`)

7.2.2 Value assigning

The next step is to use random generation to pick points in the square matrix, we already have 4 values set, one for each corner. There is an addition of visual matrix for points (indicated by number 1) to illustrate better, which points we will be working with. The final coordinates are taken from the grid of points. After, we create heights for these points and make a visualization (indicated with their height).

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)



```
Visual grid:
[[1. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 1.]]
Visual heights:
[[ 3. -1. -1. -1. -1. 21.]
 [-1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. 35. -1. -1.]
 [22. -1. -1. -1. -1. 11.]]
```

7.2.3 Triangulation

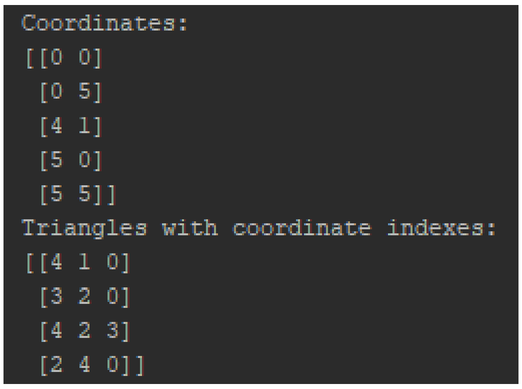
Now when we have all points and height for the triangulation we use a `scipy.spatial.Delaunay` to create triangles from our points. This library uses Qhull algorithms. The convex hull of a set of points P in n dimensions is R^n. A set $Q \in R^n$ is convex if for all q_1, $q_2 \in Q$ the line q_1q_2 is fully within Q. The Convex hull of set of points P can be described as the smallest wrapping of such points[4]. When we ask how many triangles, we have in triangulation we look at our wrapped points, shaped like a polygon, and mark the number of points in consists of as n corners. The rest of the points which

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

is inside our polygon will be m. We can find a triangulation of a pivot corner, p, with drawing a line to n-3 corners with no edge to p. This together with polygon edges gives us
$n + (n - 3) = 2n - 3$ edges, and $n - 2$ triangles, since converting points to edges is done by dividing the number of edges by
We have n corners out 2 and subtracting 1. 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[5]. Even though this is a special way of creating triangles the number of them and edges remains constant for all other forms of triangulation. The plane we perform this in is not without special cases, for this kind of triangulation to stay omnipresent there needs to be no four points along a circle circumference, which even for our basic points does not apply. To clarify, imagine having a square $ABCD$, when you want to triangulate it you can either connect AC or BD. Since we added the required corner points of our plane, triangulation will always be possible since all our points will never be on a straight line. After the algorithms run, we can get tri.simplices for triangles marked by indices of the points in an array with coordinates.

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

Output: List of coordinates and list with array describing indices from the list of coordinates



```
Coordinates:
[[0 0]
 [0 5]
 [4 1]
 [5 0]
 [5 5]]
Triangles with coordinate indexes:
[[4 1 0]
 [3 2 0]
 [4 2 3]
 [2 4 0]]
```

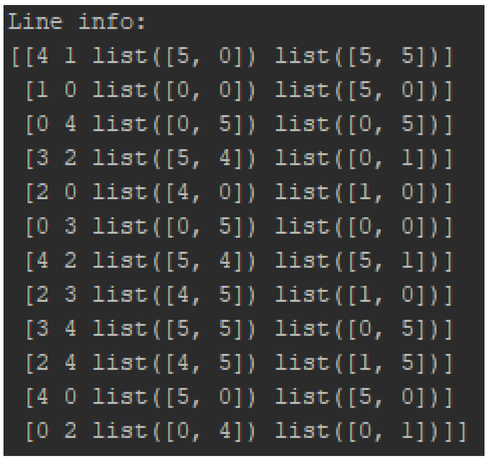
[5] https://www.uio.no/studier/emner/matnat/ifi/INF4130/h18/slides/forelesning-11---triangulering-og-convex-hull.pdf

7.3 Part of Creating and Solving Zero-Sum Game

7.3.1 Line information

For work clarification, we call a function `line_info`, which iterates through all lines and for each saves the information into an array. It consists of first point's index, second point's index, the array of size two with their x coordinates, and an array of the same size with y coordinates. We work with this array when we search for intersections.

Input: List of coordinates and list with 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

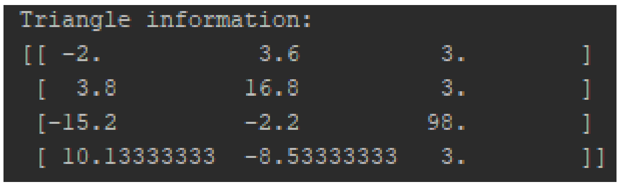


```
Line info:
[[4 1 list([5, 0]) list([5, 5])]
 [1 0 list([0, 0]) list([5, 0])]
 [0 4 list([0, 5]) list([0, 5])]
 [3 2 list([5, 4]) list([0, 1])]
 [2 0 list([4, 0]) list([1, 0])]
 [0 3 list([0, 5]) list([0, 0])]
 [4 2 list([5, 4]) list([5, 1])]
 [2 3 list([4, 5]) list([1, 0])]
 [3 4 list([5, 5]) list([0, 5])]
 [2 4 list([4, 5]) list([1, 5])]
 [4 0 list([5, 0]) list([5, 0])]
 [0 2 list([0, 4]) list([0, 1])]]
```

7.3.2 Height

The next function is called `find_A_b`. This is used to describe each triangle in our triangulated plane. We have a mathematical equation:

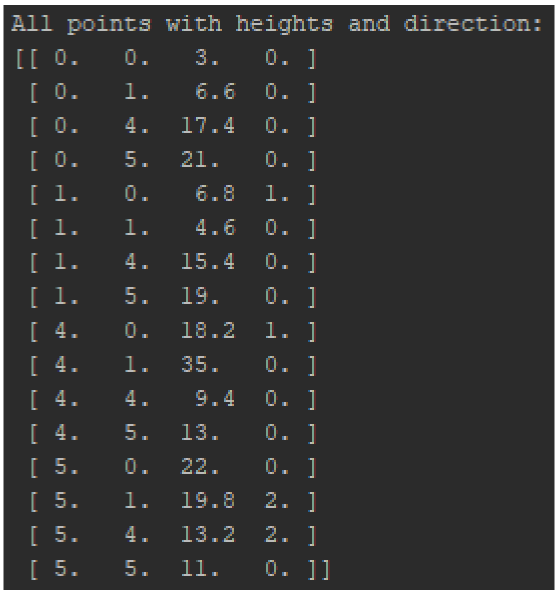
$$Ax + b = y$$



```
Triangle information:
[[ -2.           3.6          3.        ]
 [  3.8         16.8          3.        ]
 [-15.2         -2.2         98.        ]
 [ 10.13333333  -8.53333333   3.        ]]
```

When we know A and b for a triangle, if we give it point's coordinates, we can calculate its height very precisely. X in this case consists of coordinates and y is height returned. To explain it further A is a matrix of size 1×2, we multiply it by our coordinates and add b to receive the final height. This is used when we receive new coordinates from iteration in function `find_height`.
Before we can call for `find_height` it is necessary to look for new points to assign height to.

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



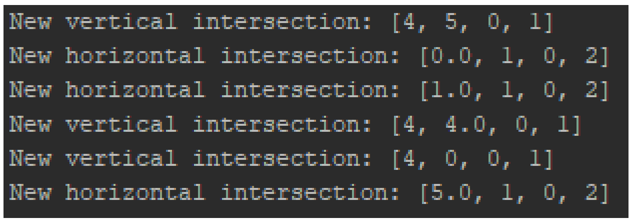
```
All points with heights and direction:
[[ 0.   0.   3.   0. ]
 [ 0.   1.   6.6  0. ]
 [ 0.   4.  17.4  0. ]
 [ 0.   5.  21.   0. ]
 [ 1.   0.   6.8  1. ]
 [ 1.   1.   4.6  0. ]
 [ 1.   4.  15.4  0. ]
 [ 1.   5.  19.   0. ]
 [ 4.   0.  18.2  1. ]
 [ 4.   1.  35.   0. ]
 [ 4.   4.   9.4  0. ]
 [ 4.   5.  13.   0. ]
 [ 5.   0.  22.   0. ]
 [ 5.   1.  19.8  2. ]
 [ 5.   4.  13.2  2. ]
 [ 5.   5.  11.   0. ]]
```

7.3.3 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 randomly generated points, while `next_rounds` calls it for new points found in the first round and then reruns it step minus one time. As an improvement to the time computation, we added direction from which was the point found. Since when the points were found by a horizontal intersection from its origin, there is no need to search for horizontal intersections again. `Quadratic_direction` checks each line segment for possible intersections, there can be multiple with one line segment, afterward, it looks if such point doesn't already exist and if that is the case it gets added to the pool of points. 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 a new point's coordinates, being x or y, as one of the points from the line segment and the other from the points that were 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 in the equation.

Input: All points in set and in 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: [4, 5, 0, 1]
New horizontal intersection: [0.0, 1, 0, 2]
New horizontal intersection: [1.0, 1, 0, 2]
New vertical intersection: [4, 4.0, 0, 1]
New vertical intersection: [4, 0, 0, 1]
New horizontal intersection: [5.0, 1, 0, 2]
```

7.3.4 Zero-sum game

When all new points are received we again use numpy to find all unique X and Y coordinates to create a grid from, list of coordinates in a gridded space, using itertools.product(X, Y). `Grid_maker` does two things. Firstly, it assigns points to their triangles for them to be assigned a height using the aforementioned `find_height`. For triangles, we calculate $c1, c2, c3$ from out points, and then if all are either non-negative or non-positive we have a point inside that triangle.

```
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)
```
[6]

Secondly, it computes a game for Alice and Bob and finds their preferred strategies. Dual programming was supposed to be the easiest part, but it turned out that not all python libraries suite this problem well. Firstly, we set up a `linprog()` library, where you only need to insert your values into a function and you receive an output. After getting various nonsensical outputs and internet searching we found this is not as robust as in its Matlab programming language counterpart. We moved on to `nashpy()`, which was meant to be more stable with our examples. It held better, but we could not test if there is something wrong or it is simply not enough. Lastly, we came across `pulp()`. Pulp is the most stable out of the three, it is harder to set up but easier to illustrate. When we wanted to put our dual program into a pulp problem, we need to do each part separately.

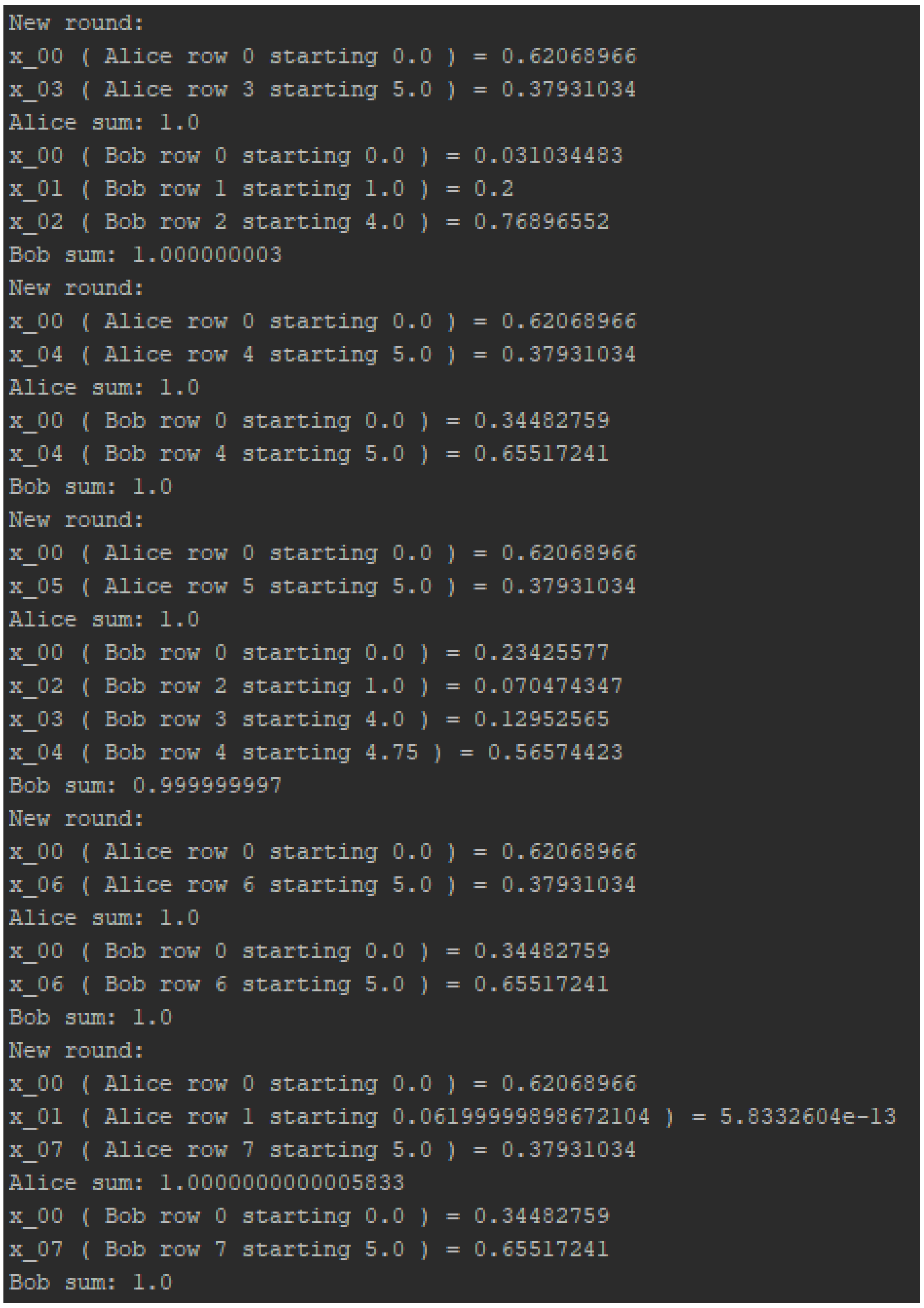
```
Alice = np.rot90(A, -1) # We use a different orientation for dual program
x_names = [format(x, '02d') for x in range(np.shape(Alice)[1] + 1)]
x = pulp.LpVariable.dicts("x", x_names, cat="Continuous")
for i in x.keys():
    if i == np.shape(Alice)[1]:
        continue # x0 has no bounds
    else: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

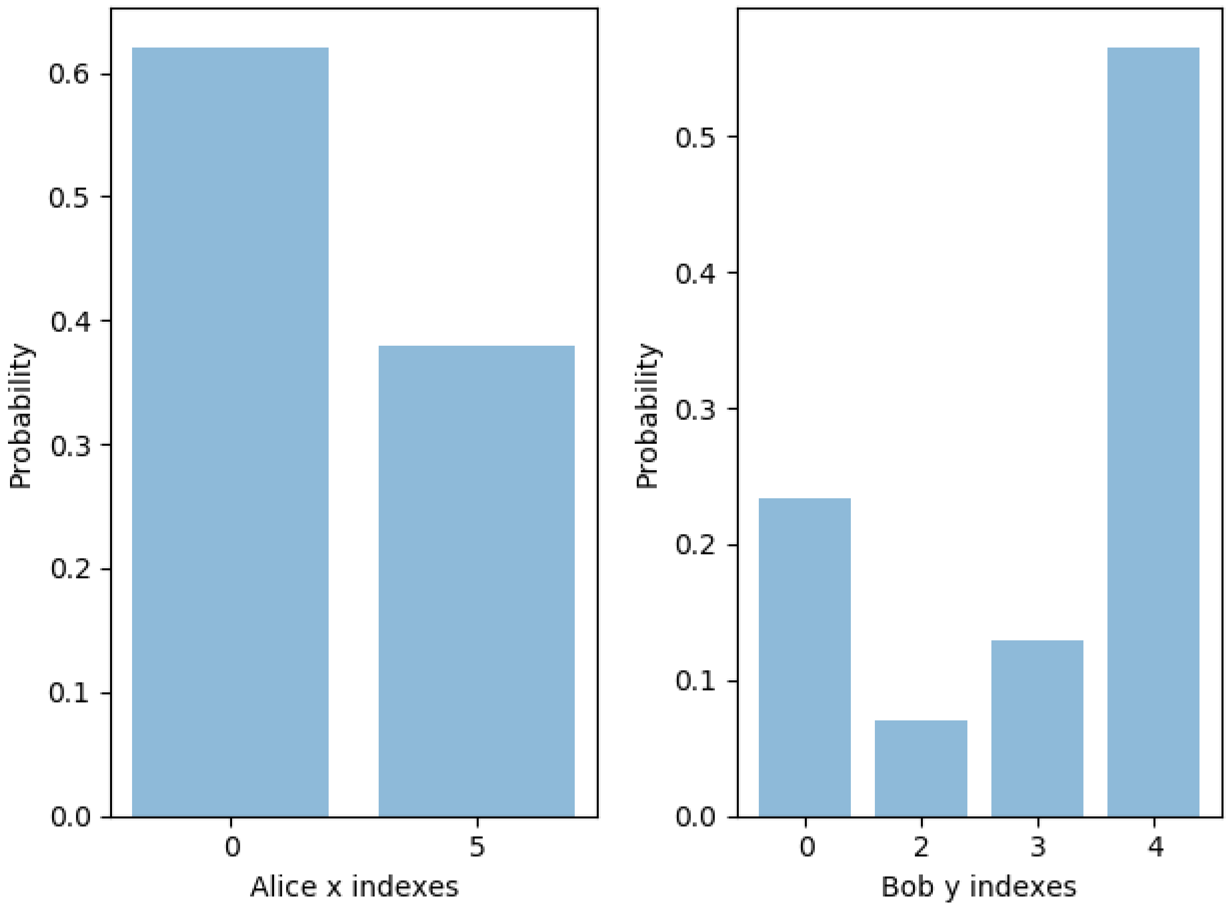
[6] https://www.w3resource.com/python-exercises/basic/python-basic-1-exercise-40.php



```
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.62068966
x_03 ( Alice row 3 starting 5.0 ) = 0.37931034
Alice sum: 1.0
x_00 ( Bob row 0 starting 0.0 ) = 0.031034483
x_01 ( Bob row 1 starting 1.0 ) = 0.2
x_02 ( Bob row 2 starting 4.0 ) = 0.76896552
Bob sum: 1.000000003
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.62068966
x_04 ( Alice row 4 starting 5.0 ) = 0.37931034
Alice sum: 1.0
x_00 ( Bob row 0 starting 0.0 ) = 0.34482759
x_04 ( Bob row 4 starting 5.0 ) = 0.65517241
Bob sum: 1.0
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.62068966
x_05 ( Alice row 5 starting 5.0 ) = 0.37931034
Alice sum: 1.0
x_00 ( Bob row 0 starting 0.0 ) = 0.23425577
x_02 ( Bob row 2 starting 1.0 ) = 0.070474347
x_03 ( Bob row 3 starting 4.0 ) = 0.12952565
x_04 ( Bob row 4 starting 4.75 ) = 0.56574423
Bob sum: 0.999999997
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.62068966
x_06 ( Alice row 6 starting 5.0 ) = 0.37931034
Alice sum: 1.0
x_00 ( Bob row 0 starting 0.0 ) = 0.34482759
x_06 ( Bob row 6 starting 5.0 ) = 0.65517241
Bob sum: 1.0
New round:
x_00 ( Alice row 0 starting 0.0 ) = 0.62068966
x_01 ( Alice row 1 starting 0.06199999898672104 ) = 5.8332604e-13
x_07 ( Alice row 7 starting 5.0 ) = 0.37931034
Alice sum: 1.0000000000005833
x_00 ( Bob row 0 starting 0.0 ) = 0.34482759
x_07 ( Bob row 7 starting 5.0 ) = 0.65517241
Bob sum: 1.0
```

As an example we also create probability graphs. Here is one example from the third round.

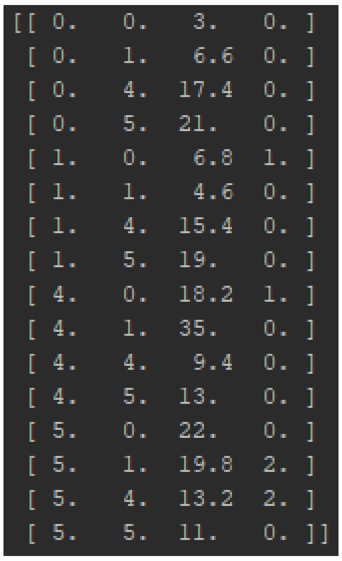
Input: Grid of points with all information for this iteration
Output: Mixed strategies for Alice and Bob in this iteration and visual output showing these values



7.3.5 Point Storing

We have a dictionary for each round, where we store all the points from that round. As the last thing, we use our grid 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.

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



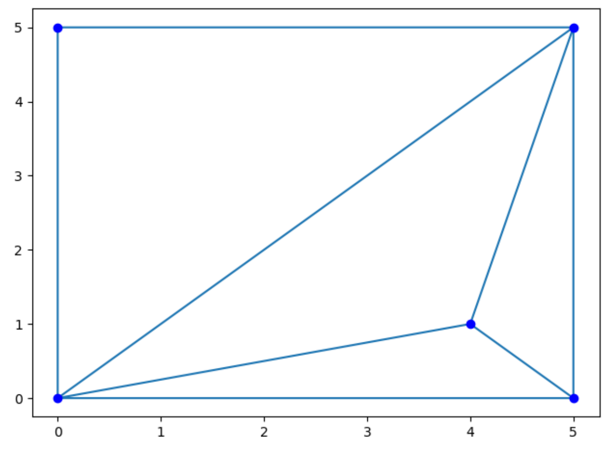
```
[[ 0.    0.    3.    0. ]
 [ 0.    1.    6.6   0. ]
 [ 0.    4.   17.4   0. ]
 [ 0.    5.   21.    0. ]
 [ 1.    0.    6.8   1. ]
 [ 1.    1.    4.6   0. ]
 [ 1.    4.   15.4   0. ]
 [ 1.    5.   19.    0. ]
 [ 4.    0.   18.2   1. ]
 [ 4.    1.   35.    0. ]
 [ 4.    4.    9.4   0. ]
 [ 4.    5.   13.    0. ]
 [ 5.    0.   22.    0. ]
 [ 5.    1.   19.8   2. ]
 [ 5.    4.   13.2   2. ]
 [ 5.    5.   11.    0. ]]
```

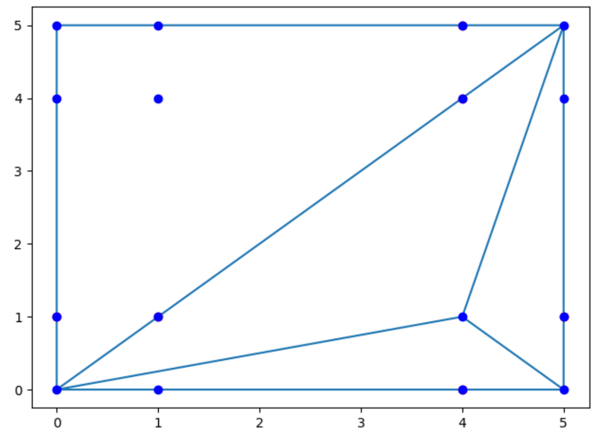
7.3.6 Visual Output

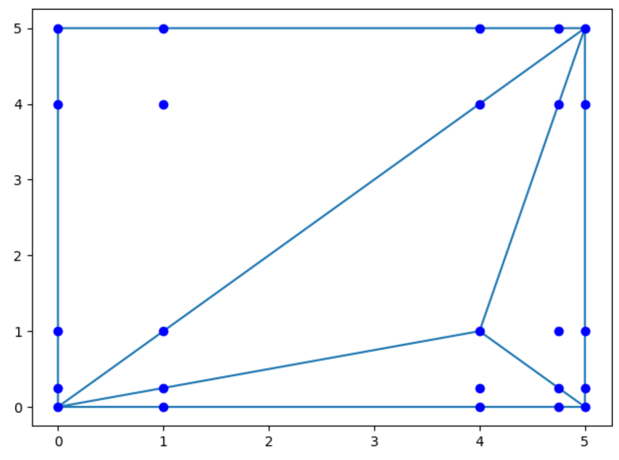
Now all the steps are finished. The first thing on the list to be drawn is a probability distribution over the possible strategies for each player side by side. The second thing is to show the viewer points active in each round with a triangulated original picture in the background. Below are the first two rounds.

Input: Triangulated space with dictionary with points in each iteration

Output: Triangulated space with grid points for each iteration







8 Conclusion

Our goal was to experimentally verify whether infinite two-player zero-sum games with payoff functions, present in the form of piecewise affine functions have finite equilibria. While making an experiment, we needed to find the best way to illustrate our zero-sum two-player game with piecewise utility functions was with a square $[0,1] \times [0,1]$. We soon realized we can use arrays and indices in them as initial points. There were many options for deciding the points, we went with pseudo-randomization of python's `NumPy` library since it would be the one most beneficial for the whole project. Next was to have triangles representing our piecewise functions. To do this, we used the `Delaunay` function from the `scipy` library for triangulation. As we focused on solving the game using linear programming, we needed to have a matrix. Entries for the matrix were interpolated functions, we called them heights, of points. As a matrix needs to be rectangular and we only had sparsely allocated points, we decided that each iteration of our game would consist of finding intersections of points present currently and lines from triangles created in triangulation. Then we created a grid using the Cartesian product of unique x, y coordinates of all points now. We used interpolated functions of the previous point along with their coordinates to describe each triangle in triangulation to give new points their height for the matrix to use in the game. Results of which we wanted to see stabilize into a finite equilibrium while iterating over and over. We learned that Python is not the greatest mathematical programming language and will rather choose Matlab for similar problems since we needed to investigate which library was implemented the best way to show the most stable results. Therefore, one iteration consists of finding intersection, creating a grid, and computing a two-player zero-sum game with linear programming. After what we learned from a theoretical background, we knew that if we observed a finite equilibrium of a zero-sum game, we could use it for the initial point as well as it is the case for polynomial games.

9 References

[1] SHOHAM, Y. and LEYTON-BROWN, K.: *MULTIAGENT SYSTEMS Algorithmic, Game-Theoretic, and Logical Foundations* [online]. 2009. Cambridge: Cambridge University Press, 2009 [ref. 2020-07-25]. Accessed from: http://www.masfoundations.org/mas.pdf

[2] 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. 2020-07-26]. Accessed from: https://www.sciencedirect.com/science/article/pii/S1877050919310130, page 681-684

[3] AVIS, D.; D. ROSENBERG, G.;·SAVANI, R.;·VON STENGEL, B.; *Enumeration of Nash equilibria for two-player games* [online]. 2008. Springer-Verlag, 2009 [ref. 2020-07-26]. Accessed from http://maths.lse.ac.uk/Personal/stengel/ETissue/ARSvS.pdf, pages 7-22

[4] GHOSH, A. and BOYD, S.: *Minimax and Convex-Concave Games* [online]. 2003. Stanford University, 2003 [ref. 2020-07-25]. Accessed from: https://web.stanford.edu/class/ee392o/cvxccv.pdf, page 9

[5] GARNAEV, A.: *Games of Timing. In: Search Games and Other Applications of Game Theory. Lecture Notes in Economics and Mathematical Systems* [online], vol 485. Springer, Berlin, Heidelberg, 2000 [ref. 2020-07-25]. Accessed from: https://link.springer.com/chapter/10.1007/978-3-642-57304-0_5, pages: 81-82

[6] L. DEVADOSS, S. and O'ROURKE, J.: *Discrete and Computational GEOMETRY*. 2011. Princeton University Press, [ref. 2020-07-25]