МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу
«Исследование операций»
ЗАДАНИЕ №1

ОТЧЕТ

о выполненном задании

студентов 311 - 312 учебных групп факультета ВМК МГУ
Волкова А., Камкия Н., Кожух П., Мирзоева Е., Сюаньлина В.
(фамилия имя студентов)

гор.Москва
2020

Оглавление

1 Постановка задачи **3**

2 Теоретическая часть **4**

2.1 Общее описание 4

2.2 Формальное определение 4

2.3 Алгоритм решения матричной игры 5

3 Реализация и визуализация **6**

4 Литература **7**

Глава 1

Постановка задачи

Задание 1 состоит в численном решении антагонистической матричной игры. Врамках данного задания необходимо:

1. написать код, решающий матричную игру путем сведения ее к паре двойственных задач линейного программирования,
2. проиллюстрировать работу данного кода путем визуализации спектров оптимальных стратегий,
3. написать автоматические тесты для решения.

Цель задания 1 заключается в том, чтобы познакомиться с языком программирования $Python$, библиотекой $SciPy$, интерактивной средой разработки $Jupyter$ и с системой тестирования $Nose$.

Формально, задача заключается в следующем:

1. (25 баллов) Необходимо написать функцию $nash_equilibrium()$, которая принимает матрицу выигрыша и возвращает значение игры и оптимальные стратегии первого и второго игроков;
2. (25 баллов) Проиллюстрировать работу кода путем решения нескольких игр и визуализации спектров оптимальных стратегий игроков в $Jupyter$. В частности, нужно привести игры, в которых:
3. (a) спектр оптимальной стратегии состоит из одной точки (т.е. существует равновесие Нэша в чистых стратегиях);
   (b) . спектр оптимальной стратегии неполон (т.е. некоторые чистые стратегии не используются);
   (c) спектр оптимальной стратегии полон.
4. (25 баллов) Оформить ваше решение в виде пакета $[fn : packaging]$;
5. (25 баллов) Написать unit-тесты для функции $nash_equilibrium$ $[fn : testing]$.

Глава 2

Теоретическая часть

2.1 Общее описание

Игра – это математическая модель реальной конфликтной ситуации. Конфликтная ситуация двух игроков называется парной игрой. Парную игру с нулевой суммой удобно исследовать, если она описана в виде матрицы. Такая игра называется матричной; матрица, составленная из чисел $a_{i,j}$, задаёт игру.

$$\begin{matrix} 0 & 1 & -2 \\ 4 & 7 & 0 \\ 5 & 2 & 3 \end{matrix}$$

Математической моделью антагонистической игры является матричная игра с матрицей A, в которой ходы (стратегии) игрока A расположены по строкам, а ходы (стратегии) игрока В расположены по столбцам. В самой матрице записаны выигрыши игрока A при соответствующих ходах игроков A и B (отрицательный выигрыш – это проигрыш).

2.2 Формальное определение

В терминах линейного программирования

Пусть матричная игра задана множеством стратегий первого игрока M, множеством стратегий второго игрока N и матрицей платежей $A[M,N]$. Рассмотрим две задачи линейного программирования:

Задача 1

Найти максимум $1^T[N]y[N]$

При ограничениях

$A[M,N]y[N] \le 1[M]$

$y[N] \ge 0[N]$

Задача 2 (двойственная)

Найти минимум $1^T[M]x[M]$

При ограничениях

$A^T[N,M]x[M] \ge 1[N]$

$x[M] \ge 0[M]$

Известно, что следующие утверждения эквивалентны

1. Матричная игра имеет положительную цену игры;
2. Задачи 1 и 2 разрешимы, при этом, если v — цена игры, $x[M]$ и $y[N]$ — оптимальные решения, то $1/v = 1^T[N]y[N] =$ и $1^T[N]y[N]$, $1^T[M]x[M]$ будут оптимальными смешанными стратегиями игроков.

2.3 Алгоритм решения матричной игры

Описание алгоритма

1. На основании анализа платёжной матрицы следует определить, существуют ли в ней доминируемые стратегии, и исключить их.
2. Найти верхнюю и нижнюю цены игры и определить, имеет ли данная игра седловую точку (нижняя цена игры должна быть равна верхней цене игры).
3. Если седловая точка существует, то оптимальными стратегиями игроков, являющимися решением игры, будут их чистые стратегии, соответствующие седловой точке. Цена игры равна верхней и нижней цены игры, которые равны между собой.
4. Если игра не имеет седловой точки, то решение игры следует искать в смешанных стратегиях. Для определения оптимальных смешанных стратегий в играх $m \times n$ следует использовать симплекс-метод, предварительно переформулировав игровую задачу в задачу линейного программирования.

Имплементация решения матричной игры с помощью симплекс-метода

Подготовка

```
Подключение необходимых подпроектов
- lib.simplex_method
- utils.matrix_stuff

А также библиотеки matplotlib для построения графиков. Для
удобства прописана директива для
препроцессора IPython Notebook о требовании встраивать результат
работы библиотеки matplotlib
```

In [1]:
```
%matplotlib inline

import sys

import lib.simplex_method
import utils.matrix_stuff
import matplotlib.pyplot as plotter
```

Обработка ввода

```
Данный блок необходим для поддержки возможности считывать
матрицу из консоли
```

In [2]:
```
def source_input():
    '''
    Prints start information, read source data.
    Returns matrix, count of rows and columns
    '''
    print("**********************************************************")
    print("Input source data")
    try:
        N = int(input("Count of strategies of 1 player: "))
        M = int(input("Count of strategies of 2 player: "))
        print("Enter the game matrix:")
        matrix = []

        i = 0
        while i < N:
            row = input().split()
            assert len(row) == M
            matrix.append([float(j) for j in row])
            i += 1
    except AssertionError:
        print("Wrong count of column")
        sys.exit()
    except ValueError:
        print("Wrong type of value")
        sys.exit()
    return N, M, matrix
```

Подготовка матрицы

В данной функции матрица обрабатывается следующим образом:

1) Удаляются доминирующие стратегии.
2) Матрица приводится к матрице, содержащей только положительные элементы

Данная функция возвращает значение минимального элемента матрицы.

In [3]:
```
def prepare_matrix(matrix, indexes1, indexes2, n, m, enable_rows_and_columns_
    '''
    matrix = list[list[float]]; copy of source_matrix
    indexes1 = list[int]; usable indexes of rows
    indexes2 = list[int]; usable indexes of columns
    n = int: count of strategies of first player
    m = int: count of strategies of second player
    enable_rows_and_columns_elimination = bool:
        enables dominated or equal rows and columns elimination
    ---------------------------------------------
    Delete dominated strategies, transform to matrix with only positive eleme
    Returns the value of minimal element in matrix
    '''

    if (enable_rows_and_columns_elimination):
        utils.matrix_stuff.delete_dominated_n_equal(matrix, indexes1, indexes

    n = len(matrix)
    m = len(matrix[0])

    min_el = matrix[0][0]
    is_any_negative = False

    for i in range(n):
      for j in range(m):
          if matrix[i][j] < 0:
              is_any_negative = True
          min_el = min(min_el, matrix[i][j])

    if is_any_negative:
        for i in range(n):
            for j in range(m):
                matrix[i][j] -= min_el

    return min(min_el, 0)
```

Построение графиков

Функция занимающаяся построением спектра для каждого игрока.

In [4]:
```
def plot (game_plot, player_no):
    fig = plotter.figure(figsize=(10, 10))
    ax1 = fig.add_subplot(211)
    ax1.stem(range(1, len(game_plot) + 1), game_plot, use_line_collection=Tru
    fig.suptitle("Vector of optimal strategies for player{}".format(player_nc
```

Приведение матрицы к задаче линейного программирования и решение.

В данной функции преобразованная матрица приводится к задаче линейного программирования, которую мы решаем с помощью симплекс-метода.

Данная функция возвращает симплекс-таблицу.

In [5]:
```
def get_simplex(matrix, n, m, pl_type = 0):
    '''
    matrix = list[list[float]]: modified matrix of game
    n = int: count of rows
    m = int: count of columns
    pl_type = int: 0 - first player, 1 - second player
    Cast modified matrix of game to linear programming task and run simplex-m
    Returns result simplex-table
    '''
    lp_matrix = []

    if pl_type == 0:
        for i in range(m):
            row = []
            for j in range(n):
                row.append(matrix[j][i])
            lp_matrix.append(row)
    else:
        for i in range(n):
            row = []
            for j in range(m):
                row.append(matrix[i][j])
            lp_matrix.append(row)

    return lib.simplex_method.run_simplex(lp_matrix, pl_type)
```

Интерфейс для тестирования

Данная функция предоставляет интерфейс для тестирования, получая на вход матрицу игры и выводя решение.

In [6]:
```
def nash_equilibrium(source_n, source_m, source_matrix):
    '''
    main function
    '''
    matrix = source_matrix.copy()
    n, m = source_n, source_m

    indexes1 = list(range(n))
    indexes2 = list(range(m))

    min_el = prepare_matrix(matrix, indexes1, indexes2, n, m, 1)

    n = len(matrix)
    m = len(matrix[0])

    simplex_matrix1 = get_simplex(matrix, n, m, 0)
    simplex_matrix2 = get_simplex(matrix, n, m, 1)

    target_f = 0
    target_plan1 = [0] * (len(simplex_matrix1[0]) - 2)
    target_plan2 = [0] * (len(simplex_matrix2[0]) - 2)
    for i in range(1, len(simplex_matrix1)):
        curr_value = simplex_matrix1[i][len(simplex_matrix1[i]) - 1]
        target_plan1[simplex_matrix1[i][0] - 1] = curr_value
        target_f += curr_value * simplex_matrix1[0][simplex_matrix1[i][0]]
```

```python
            curr_value = simplex_matrix2[i][len(simplex_matrix2[i]) - 1]
            target_plan2[simplex_matrix2[i][0] - 1] = curr_value

    V = 1 / target_f
    print("Target F = {}".format(target_f))
    print("Then V = {}".format(V))

    if min_el != 0:
        V = 1 / target_f - min_el
        print("Source matrix was modified by adding min_el = {}".format(min_e
        print("V = V' - min_el = {}".format(V))

    first_player_tactics = []
    it = 0

    for i in range(source_n):
        if it < len(indexes1) and i == indexes1[it]:
            first_player_tactics.append(target_plan1[it] * V)
            it += 1
        else:
            first_player_tactics.append(0)

    second_player_tactics = []
    it = 0

    for i in range(source_m):
        if it < len(indexes2) and i == indexes2[it]:
            second_player_tactics.append(target_plan2[it] * V)
            it += 1
        else:
            second_player_tactics.append(0)

    print("First player tactics: {}".format(first_player_tactics))
    print("Second player tactics: {}".format(second_player_tactics))

    plot(first_player_tactics, 1)
    plot(second_player_tactics, 2)
```

Тестирование

```
Для тестирования рассмотрим несколько принципально различных
матричных игр. Будем различать игры по
выходному результату.

1) Спектр оптимальной стратегии состоит из одной точки (т.е
существует равновесие Нэша в чистых
стратегиях).

Пример игры:

1  4 1
2  3 4
0 -2 7

2) Спектр оптимальной стратегии неполон (т.е. некоторые чистые
стратегии не используются)

    a) Из-за наличия доминируемых чистых стратегий

    Пример игры:
```

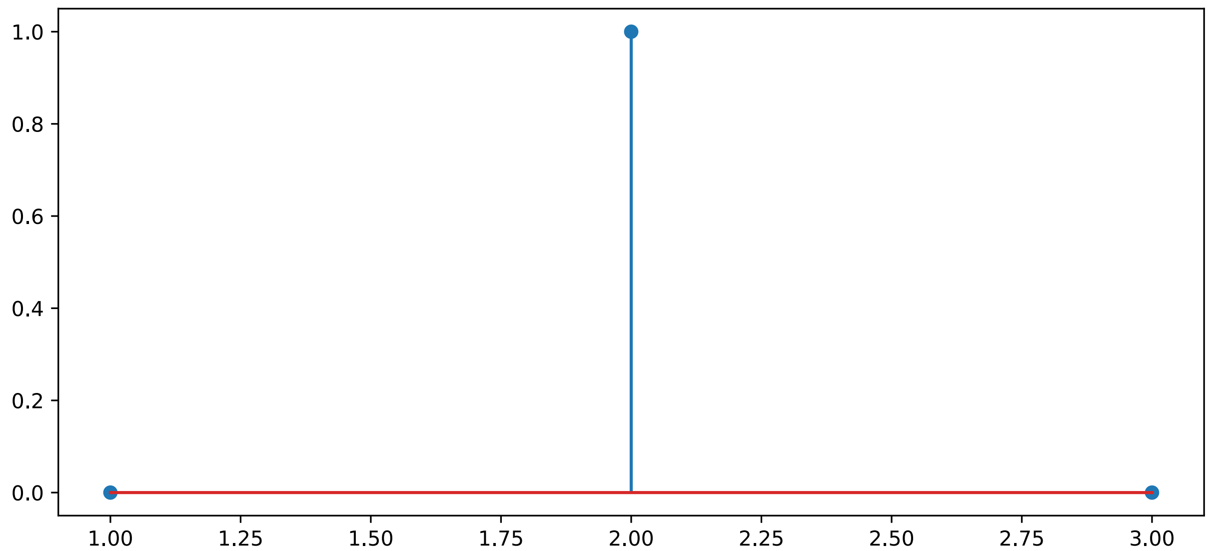
```
2 4 5
0 5 1
4 5 1

b) Из-за наличия доминируемых смешанных стратегий

Пример игры:

1 4 6
7 2 0
5 3 2

c) Спектр оптимальной стратегии полон:

Пример игры:

4 2 2
2 5 0
0 2 5
```

Спектр оптимальной стратегии состоит из одной точки (т.е существует равновесие Нэша в чистых стратегиях)

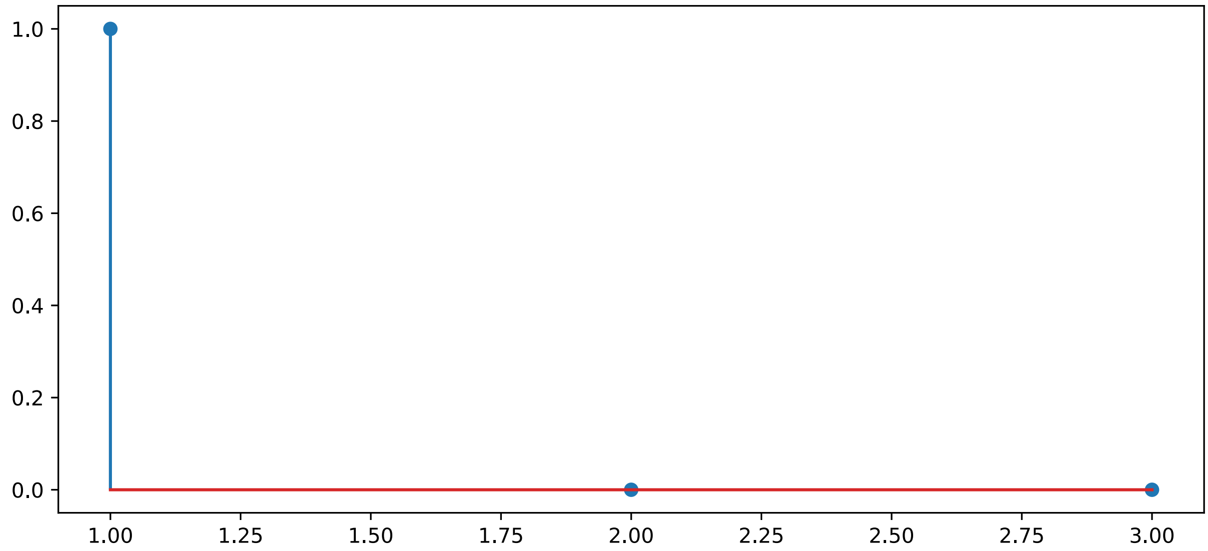
In [7]:
```
source_n = 3
source_m = 3
source_matrix = [[1, 4, 1],
                 [2, 3, 4],
                 [0, -2, 7]]
nash_equilibrium(source_n, source_m, source_matrix)
```

```
Target F = 0.5
Then V = 2.0
First player tactics: [0, 1.0, 0]
Second player tactics: [1.0, 0, 0]
```

Vector of optimal strategies for player1



Vector of optimal strategies for player2



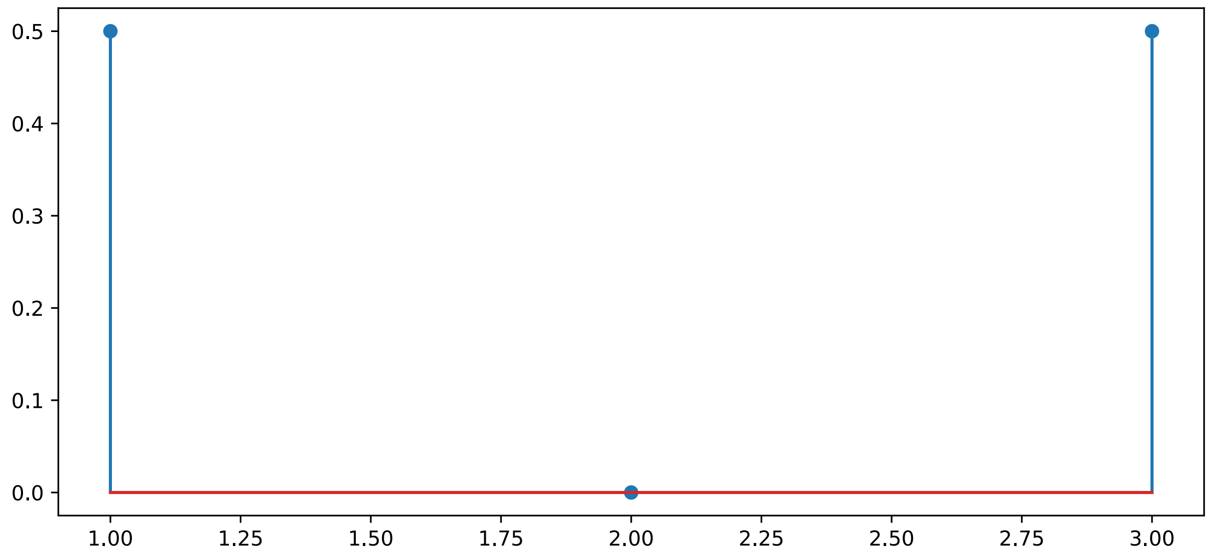
Спектр оптимальной стратегии неполон (т.е. некоторые чистые стратегии не используются) из-за наличия доминируемых чистых стратегий

In [8]:

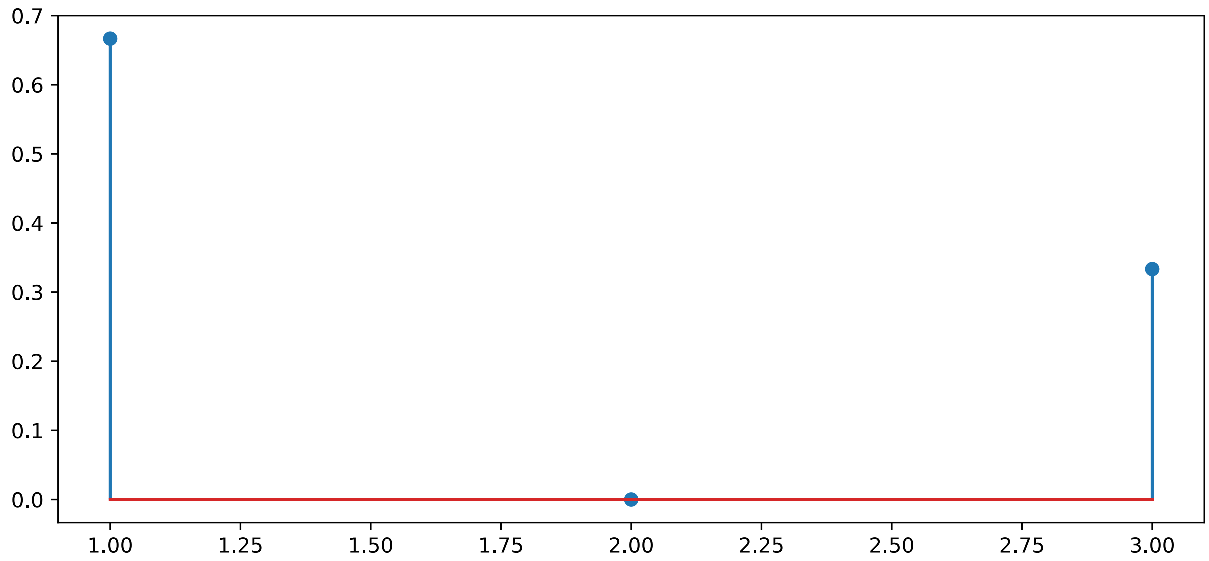
```
source_n = 3
source_m = 3
source_matrix = [[2, 4, 5],
                 [0, 5, 1],
                 [4, 5, 1]]
nash_equilibrium(source_n, source_m, source_matrix)
```

```
Target F = 0.33333333333333337
Then V = 2.9999999999999996
First player tactics: [0.4999999999999999, 0, 0.5]
Second player tactics: [0.6666666666666665, 0, 0.33333333333333326]
```

Vector of optimal strategies for player1



Vector of optimal strategies for player2



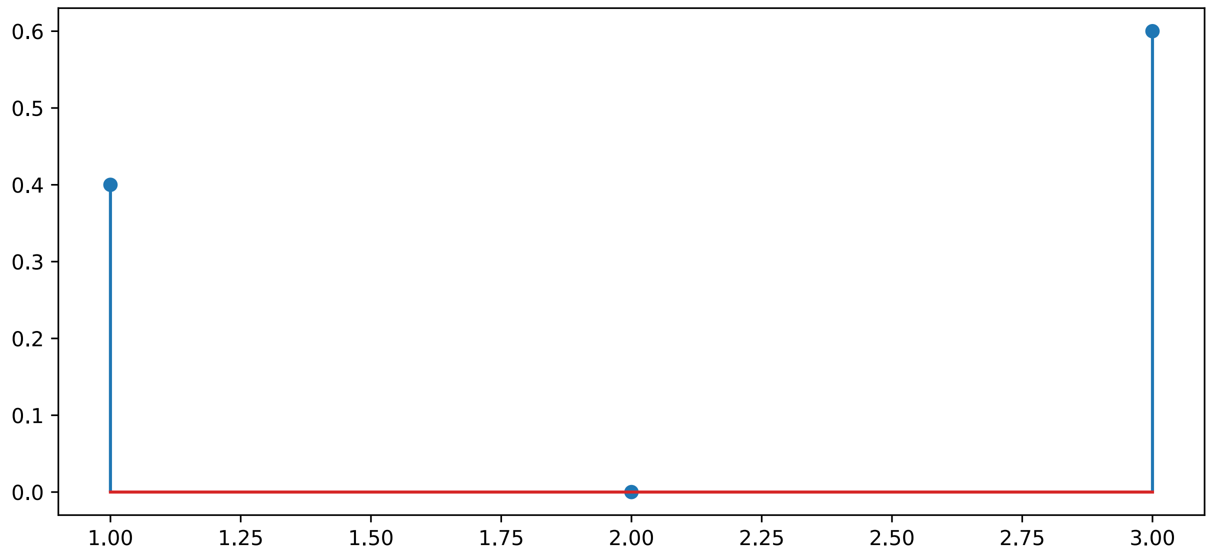
Спектр оптимальной стратегии неполон (т.е. некоторые чистые стратегии не используются) из-за наличия доминируемых смешанных стратегий

In [9]:

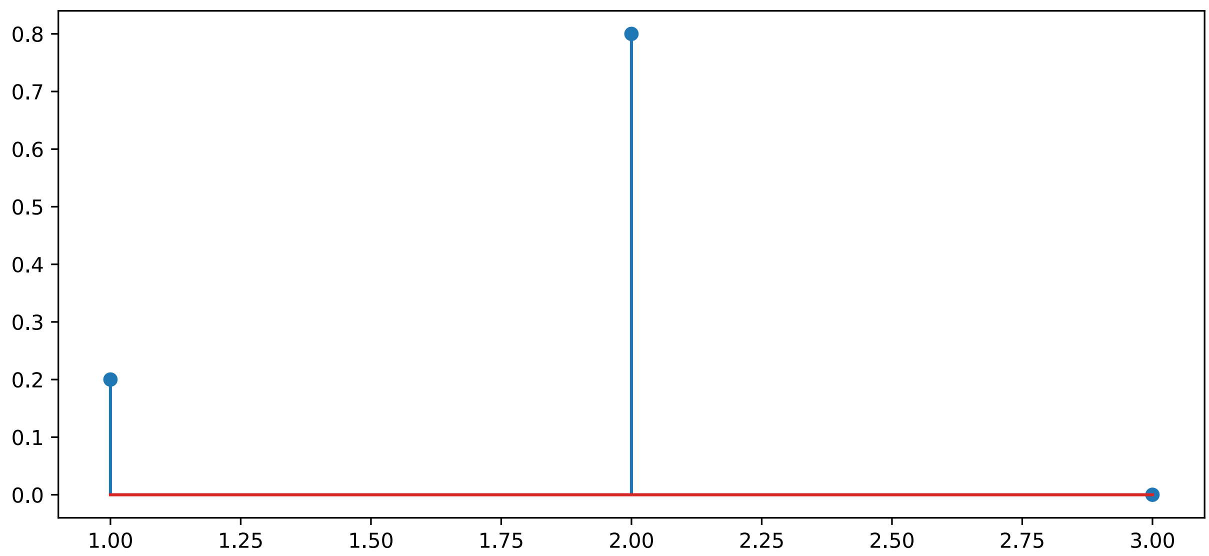
```
source_n = 3
source_m = 3
source_matrix = [[1, 4, 6],
                 [7, 2, 0],
                 [5, 3, 2]]
nash_equilibrium(source_n, source_m, source_matrix)
```

```
Target F = 0.29411764705882335
Then V = 3.4000000000000004
First player tactics: [0.4000000000000001, 0.0, 0.5999999999999999]
Second player tactics: [0.19999999999999993, 0.8, 0.0]
```

Vector of optimal strategies for player1



Vector of optimal strategies for player2

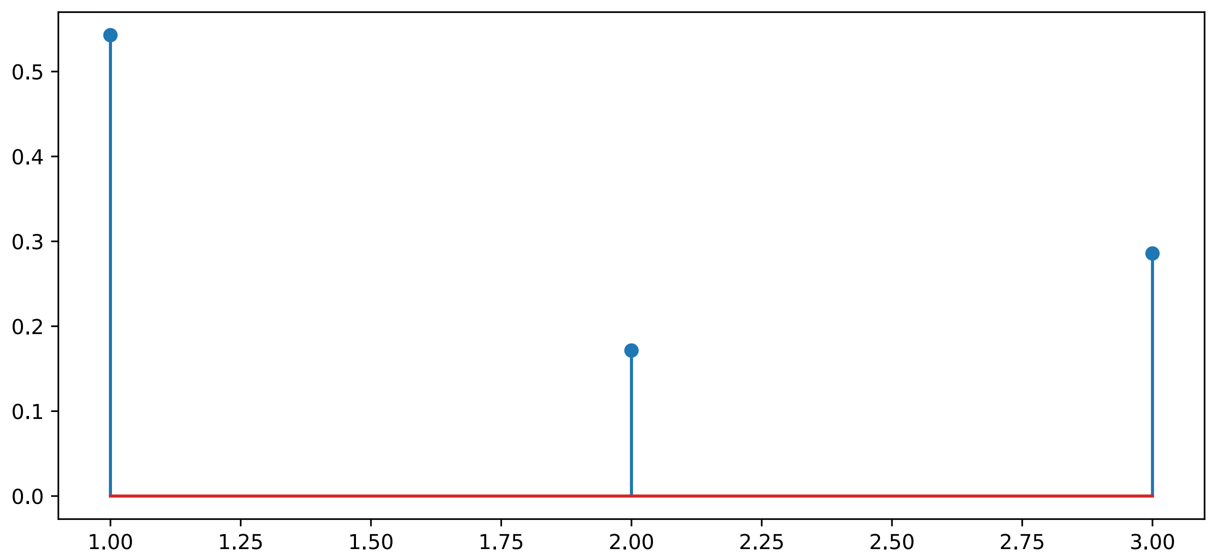


Спектр оптимальной стратегии полон

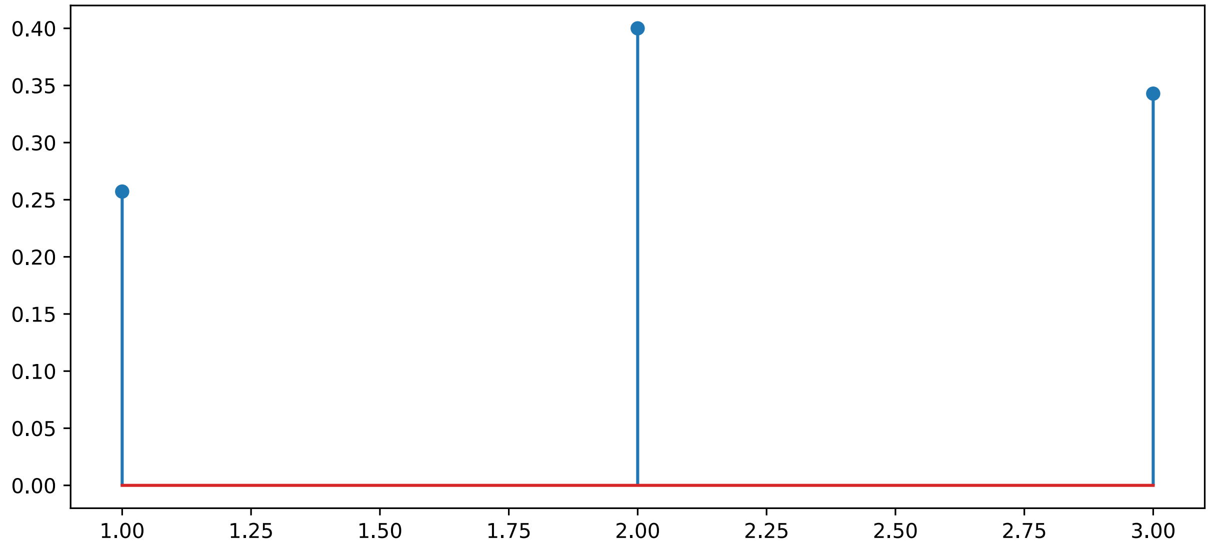
In [10]:
```
source_n = 3
source_m = 3
source_matrix = [[4, 2, 2],
                [2, 5, 0],
                [0, 2, 5]]
nash_equilibrium(source_n, source_m, source_matrix)
```

```
Target F = 0.3977272727272727
Then V = 2.5142857142857142
First player tactics: [0.5428571428571428, 0.1714285714285714, 0.2857142857142857]
Second player tactics: [0.2571428571428572, 0.39999999999999997, 0.3428571428571428]
```

Vector of optimal strategies for player1



Vector of optimal strategies for player2



In [ ]:

Глава 4

Литература

1. $https://www.coursera.org/learn/matematicheskaya-teoria-igr/home/welcome$ - общая теория игр;
2. $https://math.semestr.ru/games/linear-programming.php$ - сведение матричной игры к задаче линейного программирования;
3. $https://programforyou.ru/calculators/simplex-method$ - описание алгоритма симплекс-метода, а также калькулятор, служивший эталоном для проверки решений;
4. $https://programforyou.ru/calculators/simplex-method$ - описание алгоритма симплекс-метода, а также калькулятор, служивший эталоном для проверки решений.