Оглавление

[Техническое задание 2](#_Toc55227239)

[Глава 1 Метод Брауна-Робинсон 3](#_Toc55227240)

[1.1 Теоретическая часть 3](#_Toc55227241)

[1.2 Программный код 4](#_Toc55227242)

[1.3 Результаты работы программы 6](#_Toc55227243)

[1.4 Теоретические вопросы 7](#_Toc55227244)

[Глава 2 Накопленный и средний выигрыш в длинной серии антагостичесих игр 8](#_Toc55227245)

[2.1 Теоретическая часть 8](#_Toc55227246)

[2.2 Программный код 9](#_Toc55227247)

[2.3 Результаты работы программы 11](#_Toc55227248)

[Глава 3 Классическая задача о разорении игрока 12](#_Toc55227249)

[Теоретическая часть 12](#_Toc55227250)

[3.2 Программный код 13](#_Toc55227251)

[3.3 Результаты работы программы 15](#_Toc55227252)

[3.4 Теоретические вопросы 16](#_Toc55227253)

[Глава 4 Последовательный анализ 17](#_Toc55227254)

[4.1 Теоретическая часть 17](#_Toc55227255)

[4.2 Программный код 18](#_Toc55227256)

[4.3 Результаты работы программы 20](#_Toc55227257)

[Глава 5 Графская иллюстрация множества достижимых итогов в неантагонистической биматричной игре 21](#_Toc55227258)

[5.1 Теоретическая часть 21](#_Toc55227259)

[5.2 Программный код 22](#_Toc55227260)

[5.3 Результаты работы программы 23](#_Toc55227261)

[Глава 6 Вычисление вектора Шепли 24](#_Toc55227262)

[6.1 Теоретическая часть 24](#_Toc55227263)

[6.2 Программный код 25](#_Toc55227264)

[6.3 Результаты работы программы 26](#_Toc55227265)

[Заключение 27](#_Toc55227266)

# Техническое задание

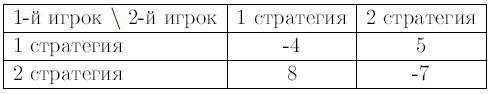
В ходе данной курсовой работы будет необходимо реализовать 6 практических заданий:

* Метод Брауна-Робинсон
* Накопленный и средний выигрыш в длинной серии антагонистичесих игр
* Классическая задача о разорении игрока
* Последовательный анализ
* Графская иллюстрация множества достижимых итогов в неантагонистической биматричной игре
* Вычисление вектора Шепли

# Глава 1 Метод Брауна-Робинсон

## 1.1 Теоретическая часть

Антагонистическая игра, или игра с нулевой суммой, это игра, где выигрыш одного игрока всегда равен проигрышу другого. Условия игры можно свести в таблицу, называемую матрицей платежей.



В таблице приведены выигрыши первого игрока при различных вариантах выбора игроков, называемых стратегиями. Например, если первый игрок применит вторую стратегию, а второй применит первую стратегию, то первый игрок получит выигрыш в 8 единиц. Ясно, что обоим игрокам следует использовать смешанные стратегии применять первую или вторую из своих возможных стратегий поочередно случайным образом с некоторыми вероятностями.

Можно доказать, что каждая антагонистическая игра имеет решение - пару оптимальных стратегий, отступать от которых невыгодно ни одному игроку, если другой продолжает ее придерживаться.

Метод Брауна - Робинсон заключается в следующем: при первой игре стратегии выбираются произвольно, а в каждой из последующих игр каждый из игроков выбирает стратегию, обеспечивающую максимальный выигрыш против того набора стратегий, который противник использовал в предыдущих партиях. Можно доказать, что вероятности выбора стратегий сходятся к оптимальным, а средний выигрыш - к цене игры.

## 1.2 Программный код

|  |
| --- |
| *"""*  *Программа, моделирующая последовательность*  *антагонистических игр с заданной матрицей платежей*  *Автор: Афанасьев И.Е.*  *Дата написания: 20.09.2020*  *"""*  iterations = 10000 *# Количество ходов*  start\_money = 10000 *# Начальное количество денег у каждого игрока*  matrix1 = [ [-4, 5], [8, -7] ] *# Матрица платежей*  *# Инвертируем матрицу (Платежи для второго игрока)*  q = len(matrix1)  matrix2 = []  **for** i **in** range(q):  matrix2.append([])  **for** i **in** matrix1:  **for** pos, j **in** enumerate(i):  matrix2[pos % q].append(- j)  *# Класс игрок с матрицей платежей, текущим количеством*  *# денег, коэффициентами для принятия решения на основе*  *# стратегии противника и последним ходом*  **class** **gamer**:  **def** \_\_init\_\_(self, matrice, money):  self.m = matrice  self.money = money  self.koef = [0] \* len(matrice[0])  self.last = 0  **def** strikeback(self, enemy\_strat):  self.koef[enemy\_strat] += 1  self.money += self.m[self.last][enemy\_strat]  **return** self.money  *# Вычисление оптимального хода на основе*  *# предыдущих ходов противника*  **def** math(self):  M = []  **for** i **in** self.m:  process = 0  **for** j **in** range(len(self.koef)):  process += self.koef[j] \* i[j]  M.append(process)  self.last = M.index(max(M))  **return** self.last  *# Импортируем графическую библиотеку*  **import** **matplotlib.pyplot** **as** **plt**  *# Создаём два объекта класса "игрок"*  player1 = gamer(matrix1, start\_money)  player2 = gamer(matrix2, start\_money)  *# Инициализируем фигуру, на которой будем рисовать*  fig = plt.figure()  plt.ion()  color = ['red', 'green', 'blue', 'brown']  y = [] *# Ось y*  x = [] *# Ось x*  **for** i **in** range(len(color)):  y.append([])  x.append([])  *# Собственно, игра*  **for** i **in** range(iterations):  str1 = player1.math()  str2 = player2.math()    y[str1].append(player1.strikeback(str2))  x[str1].append(i)    y[str2 + 2].append(player2.strikeback(str1))  x[str2 + 2].append(i)  **for** i **in** range(len(x)):  plt.plot(x[i], y[i], c = color[i])  plt.show() *# Отображение графика*  print('Длительность тестовой серии игры: ', iterations)  print(  'Стратегии первого игрока: **%s\n**Деньги первого игрока: **%d\n**' %  (player2.koef, player1.money)  )  print(  'Стратегии второго игрока: **%s\n**Деньги второго игрока: **%d**' %  (player1.koef, player2.money)  ) |

Код 1. «Программа, моделирующая последовательность

антагонистических игр с заданной матрицей платежей»

## 1.3 Результаты работы программы

Длительность тестовой серии игры: 10000

Стратегии первого игрока: [6300, 3700]

Деньги первого игрока: 14763

Стратегии второго игрока: [5013, 4987]

Деньги второго игрока: 5237

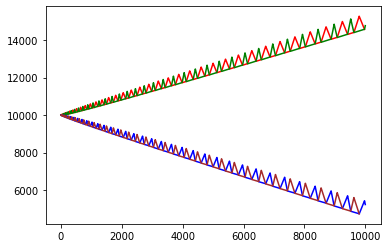


График 1. «Изменение количества денег первого и второго игрока»

## 1.4 Теоретические вопросы

1. Каковы вычисленные теоретические стратегии игроков и цена игры?

matrix = [ [-4, 5], [8, -7] ]

S(x, y) = −4xy + 5x(1 − y) + 8(1 − x)y − 7(1 − x)(1 − y) =

= −4xy - 5xy + 5x + 8y – 8xy – 7 + 7x + 7y -7xy = -24xy + 12x + 15y - 7

S(x, y) = -24xy + 12x + 15y – 7

Sx = -24y + 12 = 0

y = 1/2

Sy = -24x + 15 = 0

x = 15/24

S(15/24, 1/2) = -15/2 + 15/2 + 15/2 – 7 = ½

1. Велика ли разница между теоретическими и найденными программой значениями?

Разница не велика, при увеличении числа ходов она стремится к теоретическим показателям

1. Кто из игроков выигрывает чаще?

Второй игрок

1. Верно ли, что тот, кто выигрывает чаще, выигрывает больше?

Нет

# Глава 2 Накопленный и средний выигрыш в длинной серии антагонистичесих игр

## 2.1 Теоретическая часть

Пусть двое играют в некоторую антагонистическую игру. Проводится некоторое достаточно большое число партий. Накопленный выигрыш — это суммарный выигрыш во всех партиях. Средний выигрыш — это накопленный выигрыш, разделенный на число сыгранных партий. Средний выигрыш стремится к цене игры, когда число партий велико. Однако при малом числе партий тенденция может быть не так заметна, поскольку разброс может быть большим. Для исследования вопроса о скорости сходимости среднего выигрыша к цене игры следует использовать метод статистических испытаний. Разыгрывается достаточно длинная серия игр, в каждой из которых оба игрока применяют свои оптимальные смешанные стратегии. (Эти стратегии могут быть найдены как аналитически, так и методом Брауна - Робинсон, описанном в предыдущей лабораторной работе). В каждой партии стратегия выбирается случайным образом в соответствии с оптимальными вероятностями смешанной стратегии.

## 2.2 Программный код

|  |
| --- |
| *"""*  *Программа, рассчитывающая вероятность гипотезы методом последовательного*  *анализа*  *Автор: Афанасьев И.Е.*  *Дата написания: 20.09.2020*  *"""*  **import** **numpy** **as** **np**  p = 0.75 *# вероятность единицы*  p1, p2, alpha = 0.45, 0.85, 0.05 *# гипотезы и порог*  iterations = 10000 *# Количество итераций*  *# функция, возвращающая единицу с вероятностью p и нуль с вероятностью*  *# 1 - p*  **def** random(p):  **return** np.random.binomial(n = 1, p = p)  *# Вычисление апостериорной вероятности PH1A по формуле Байеса*  **def** Bayes1(PAH1, PH1, PAH2, PH2):  **return** PAH1 \* PH1 / (PAH1 \* PH1 + PAH2 \* PH2)    *# Вычисление апостериорной вероятности PH2A по формуле Байеса*  **def** Bayes2(PAH1, PH1, PAH2, PH2):  **return** PAH2 \* PH2 / (PAH1 \* PH1 + PAH2 \* PH2)  *# Начальные значения*  PH1 = 0.5  PH2 = 0.5  PH1A = 0.5  PH2A = 0.5  zeros = 0 *# Количество нулей*  masPH1A = [] *# Массив вероятностей для графика*  masPH2A = [] *# Массив вероятностей для графика*  ones = 0 *# Количество единиц*  *# До тех пор пока не превысили порог или количество итераций не превысило*  *# максимум*  **while** PH1A < 1 - alpha **and** PH2A < 1 - alpha **and** zeros + ones < iterations:  number = random(p) *# очередное число последовательности*  *# Если это единица*  **if** number == 1:  PAH1 = p1  PAH2 = p2  ones += 1  *# Если нуль*  **else**:  PAH1 = 1 - p1  PAH2 = 1 - p2  zeros += 1  *# Вычисляем апостериорные вероятности и добавляем их в массив*  PH1A = Bayes1(PAH1, PH1, PAH2, PH2)  PH2A = Bayes2(PAH1, PH1, PAH2, PH2)  masPH1A += [PH1A]  masPH2A += [PH2A]  *# Теперь априорные вероятности для следующего вычисления - апостериорные*  *# вероятности предыдущего*  PH1 = PH1A  PH2 = PH2A    print("Длительность серии: ", ones + zeros)  print("Количество 1: ", ones)  print("Количество 0: ", zeros)  print("PH1A: ", PH1A)  print("PH2A: ", PH2A)  **if** PH1A > 1 - alpha:  print("Принимаем первую гипотезу")  **elif** PH2A > 1 - alpha:  print("Принимаем вторую гипотезу")  *# В случае, если был первышен максимум итераций*  **elif** PH1A > PH2A:  print("Принимаем первую гипотезу")  **else**:  print("Принимаем вторую гипотезу")    *# Строим графики*  **import** **matplotlib.pyplot** **as** **plt**  fig = plt.figure()  plt.subplot(2, 1, 1)  plt.plot([i **for** i **in** range(ones + zeros)], masPH1A)  plt.subplot(2, 1, 2)  plt.plot([i **for** i **in** range(ones + zeros)], masPH2A)  plt.show() |

Код 2. «Программа, рассчитывающая вероятность гипотезы методом последовательного анализа»

## 2.3 Результаты работы программы

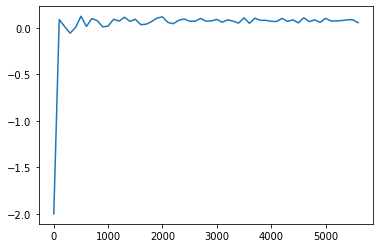


График 2. «Зависимость среднего выигрыша от числа сыгранных партий»

Аналогично предыдущей лабораторной работе:

S(x, y) = −2xy + 3x(1 − y) + 3(1 − x)y − 4(1 − x)(1 − y) = = −12xy + 7x + 7y − 4.

x = 7/12

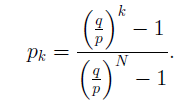
y = 7/12

S(7/12, 7/12) = **1/12** – соответствует практическому результату

# Глава 3 Классическая задача о разорении игрока

## Теоретическая часть

Классическая задача о разорении игрока рассматривалась и была решена в XIX веке. Постановка задачи такова: пусть двое игроков играют в некоторую игру, причем первый выигрывает каждую партию независимо от других с вероятностью p, проигрывает с вероятностью q = 1 − p, а ничьих не бывает. В каждой партии разыгрывается ставка в 1 рубль. Первоначальный капитал первого игрока составляет k рублей, а второго - N − k рублей. Партии продолжаются до полного разорения одного из игроков, то есть до того момента, когда ему будет нечего поставить на кон. Можно представить условие задачи как случайное блуждание некоторой фигурки на отрезке прямой: если капитал первого игрока на текущий момент равен x, то фигурка находится в точке в координатой x. После следующей партии она смещается либо в x + 1, либо вx−1, в зависимости от того, выиграл или проиграл первый игрок эту партию. Если фигурка попадает в точку с координатой 0 или N, случайное блуждание заканчивается - происходит разорение одного из игроков. Задача имеет аналитическое решение. Если, как было указано, первоначальный капитал первого игрока составляет k рублей, а второго - N − k рублей, то вероятность выигрыша первого игрока, то есть разорения второго, равна



## 3.2 Программный код

|  |
| --- |
| *"""*  *Программа, производящая вычисление по формуле*  *вероятности общей победы и строящая графики зависимости*  *в задаче о разорении игрока*  *Автор: Афанасьев И.Е.*  *Дата написания: 20.09.2020*  *"""*  *# Функция расчёта вероятности общей победы*  **def** fun(p, k, N):  q = 1 - p  **return** ((q/p)\*\*k - 1) / ((q/p)\*\*N - 1)  *# импорт графической библиотеки*  **import** **matplotlib.pyplot** **as** **plt**  *# Задаём пул различных параметров задачи*  *# P - массив вероятность выигрыша первого игрока*  *# K - массив количества денег первого игрока (n - k - деньги второго игрока)*  *# N - массив общего количества денег игроков*  P = [i/100 **for** i **in** range(55, 100, 5)]  K = [i **for** i **in** range(1, 10, 1)]  N = [i **for** i **in** range(5, 15, 1)]  *# для первого задания*  *# Фиксируем n и k, p - меняем*  funcP = []  **for** p **in** P:  funcP += [fun(p, K[len(K) // 2], N[len(N) // 2])]    *# Фиксируем p и n, k - меняем*  funcK = []  **for** k **in** K:  funcK += [fun(P[len(P) // 2], k, N[len(N) // 2])]    *# Фиксируем p и k, n - меняем*  funcN = []  **for** n **in** N:  funcN += [fun(P[len(P) // 2], K[len(K) // 2], n)]  *# Строим графики*  fig = plt.figure()  plt.subplot(4, 1, 1)  plt.plot(P, funcP)  plt.subplot(4, 1, 2)  plt.plot(K, funcK)  plt.subplot(4, 1, 3)  plt.plot(N, funcN)    *# для второго задания*  *# k и n меняются в соотношении 1 к 10*  K = [i **for** i **in** range(1,10,1)]  N = [i **for** i **in** range(10,100,10)]  p = 0.55  *# Вычисляем по формуле*  funcNK10 = []  **for** i **in** range(len(K)):  funcNK10 += [fun(p, K[i], N[i])]  *# Строим графики*  plt.subplot(4, 1, 4)  plt.plot(K, funcNK10)  *# Выводим на экран*  plt.show()  *# 3 задание - при p -> 1*  # График случайного блуждания - зависимость  # капитала игрока от количества сыгранных партий  iters = 100000  start\_money = 1000 # Начальное количество денег у каждого игрока  player1\_money = start\_money  p1 = 0.51  y = [] # Ось y  x = [] # Ось x  import numpy as np  i = 0  while player1\_money > 0 and player1\_money < start\_money \* 2 and iters > 0:  player1\_money += ((np.random.binomial(n = 1, p = p1)) \* 2 - 1)  y.append(player1\_money)  x.append(i)  i += 1  iters -= 1      fig = plt.figure(figsize=(30,30))  plt.subplot(5, 1, 5)  plt.plot(x, y)    # Выводим на экран  plt.show() |

Код 3. «Программа, производящая вычисление по формуле вероятности общей победы и строящая графики зависимости в задаче о разорении игрока»

## 3.3 Результаты работы программы

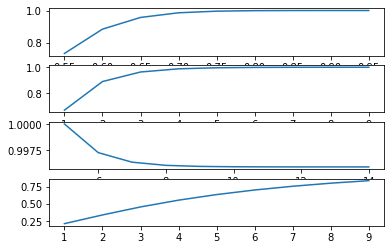
Графики:

Зависимость вероятности общей победы от p – вероятности победы первого игрока

Зависимость вероятности общей победы от k – числа денег первого игрока в начале

Зависимость вероятности общей победы от n – общего числа денег

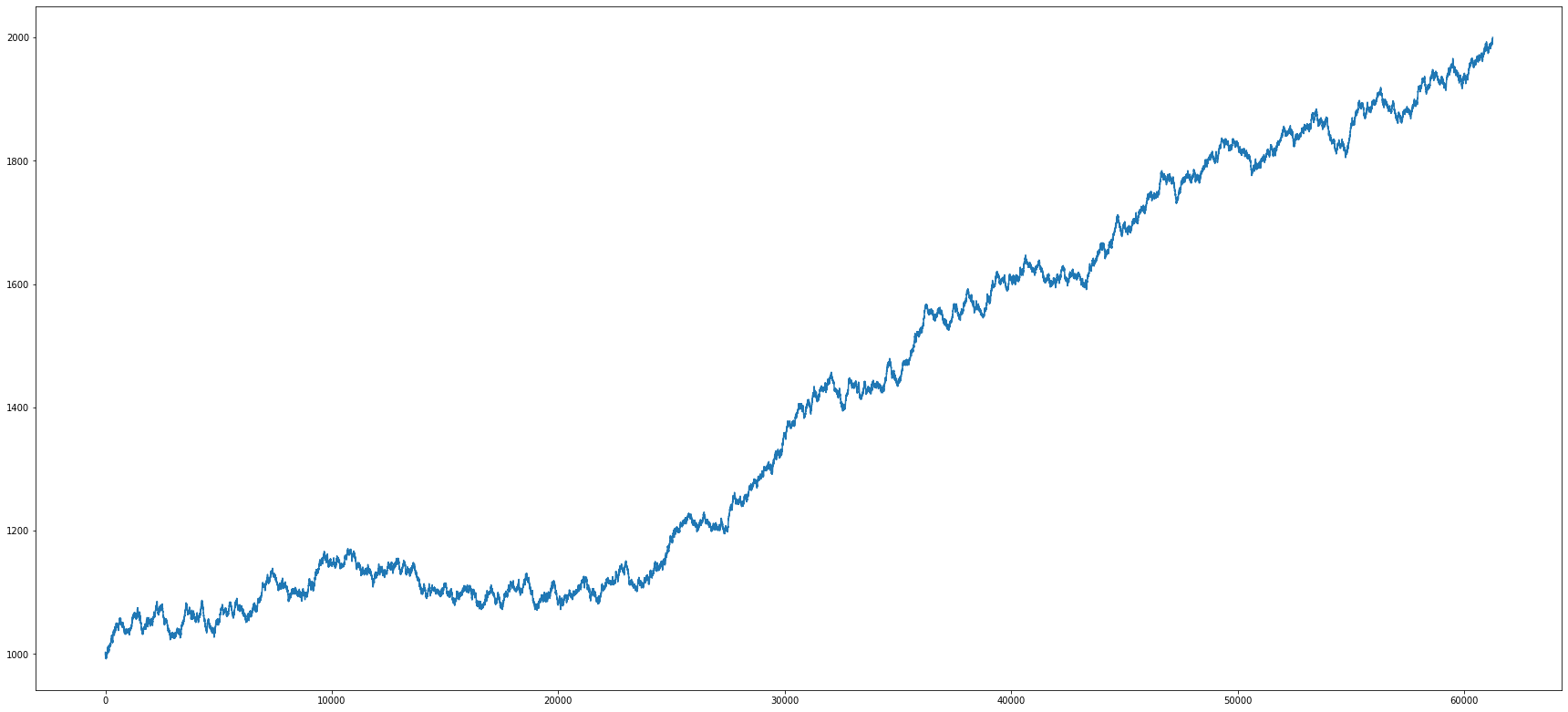
Зависимость общей победы от k при соотношении k / n = 0.1



Чтобы первый игрок почти наверняка выигрывал: вероятность p -> 1

Умение играть важнее

График случайного блуждания (p = 0.51 – вероятность победы первого игрока, стартовый капитал - 1000):



## 3.4 Теоретические вопросы

1) При p -> 0 – pk -> 0

При p -> 1 – pk -> 1

Эти пределы зависят от N и k только при условии N = k (У второго игрока денег нет изначально, тогда при любом p – первый игрок уже выиграл)

Или при условии k = 0 (Аналогично, второй игрок уже выиграл)

При N = 0 – решения нет, поскольку не имеет смысла, игра вообще не состоится (капитала нет изначально)

2) Pk = k! / (N \* (N – 1) \* (N – 2) \* … \*(N – k + 1))

3) Более богатого.

# Глава 4 Последовательный анализ

## 4.1 Теоретическая часть

Допустим, что на вход поступает последовательность нулей и единиц, причем вероятность появления единицы равна p. Значение этой вероятности p нам не известно, известно только, что существуют две возможности: p = p1 и p = p2. Как сделать выбор между этими двумя возможностями? Можно, проведя достаточно много измерений, то есть получив достаточно длинную входную последовательность нулей и единиц, рассчитать, какова вероятность такой серии при условии выполнения каждой из гипотез и после этого остановиться на той гипотезе, для которой эта вероятность выше. Этот подход имеет существенный недостаток: может оказаться, что выбранная длина серии слишком мала, обе гипотезы вполне вероятны и отличить их сложно. С другой стороны, может оказаться, что серия слишком велика. Если получение данных - дорогостоящая процедура, то имеет смысл сделать входную серию максимально короткой. Эти трудности преодолены в предложенном А. Вальдом методе последовательного анализа. Идея последовательного анализа заключается в следующем. Каждое новое наблюдение служит основанием для пересчета вероятностей гипотез. В какой-то момент вероятность одной из гипотез достигает некоторого критически большого значения, а вероятность второй - критически малого. В этот момент наблюдения заканчиваются и принимается первая гипотеза.

## 4.2 Программный код

|  |
| --- |
| *"""*  *Программа, рассчитывающая вероятность гипотезы методом последовательного*  *анализа*  *Автор: Афанасьев И.Е.*  *Дата написания: 20.09.2020*  *"""*  **import** **numpy** **as** **np**  p = 0.75 *# вероятность единицы*  p1, p2, alpha = 0.45, 0.85, 0.05 *# гипотезы и порог*  iterations = 10000 *# Количество итераций*  *# функция, возвращающая единицу с вероятностью p и нуль с вероятностью*  *# 1 - p*  **def** random(p):  **return** np.random.binomial(n = 1, p = p)  *# Вычисление апостериорной вероятности PH1A по формуле Байеса*  **def** Bayes1(PAH1, PH1, PAH2, PH2):  **return** PAH1 \* PH1 / (PAH1 \* PH1 + PAH2 \* PH2)    *# Вычисление апостериорной вероятности PH2A по формуле Байеса*  **def** Bayes2(PAH1, PH1, PAH2, PH2):  **return** PAH2 \* PH2 / (PAH1 \* PH1 + PAH2 \* PH2)  *# Начальные значения*  PH1 = 0.5  PH2 = 0.5  PH1A = 0.5  PH2A = 0.5  zeros = 0 *# Количество нулей*  masPH1A = [] *# Массив вероятностей для графика*  masPH2A = [] *# Массив вероятностей для графика*  ones = 0 *# Количество единиц*  *# До тех пор пока не превысили порог или количество итераций не превысило*  *# максимум*  **while** PH1A < 1 - alpha **and** PH2A < 1 - alpha **and** zeros + ones < iterations:  number = random(p) *# очередное число последовательности*  *# Если это единица*  **if** number == 1:  PAH1 = p1  PAH2 = p2  ones += 1  *# Если нуль*  **else**:  PAH1 = 1 - p1  PAH2 = 1 - p2  zeros += 1  *# Вычисляем апостериорные вероятности и добавляем их в массив*  PH1A = Bayes1(PAH1, PH1, PAH2, PH2)  PH2A = Bayes2(PAH1, PH1, PAH2, PH2)  masPH1A += [PH1A]  masPH2A += [PH2A]  *# Теперь априорные вероятности для следующего вычисления - апостериорные*  *# вероятности предыдущего*  PH1 = PH1A  PH2 = PH2A    print("Длительность серии: ", ones + zeros)  print("Количество 1: ", ones)  print("Количество 0: ", zeros)  print("PH1A: ", PH1A)  print("PH2A: ", PH2A)  **if** PH1A > 1 - alpha:  print("Принимаем первую гипотезу")  **elif** PH2A > 1 - alpha:  print("Принимаем вторую гипотезу")  *# В случае, если был первышен максимум итераций*  **elif** PH1A > PH2A:  print("Принимаем первую гипотезу")  **else**:  print("Принимаем вторую гипотезу")    *# Строим графики*  **import** **matplotlib.pyplot** **as** **plt**  fig = plt.figure()  plt.subplot(2, 1, 1)  plt.plot([i **for** i **in** range(ones + zeros)], masPH1A)  plt.subplot(2, 1, 2)  plt.plot([i **for** i **in** range(ones + zeros)], masPH2A)  plt.show() |

Код 4. «Программа, рассчитывающая вероятность гипотезы методом последовательного анализа»

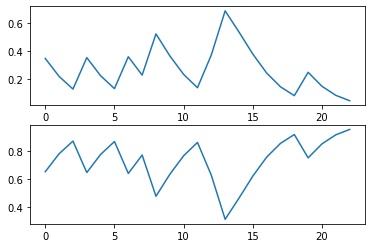
## 4.3 Результаты работы программы

Пусть:

p = 0.75 - вероятность единицы

p1, p2 = 0.45, 0.85 - гипотезы

График зависимости апостериорной вероятности (первой и второй) от числа поступивших нулей и единиц



Длительность серии: 23

Количество 1: 17

Количество 0: 6

PH1A: 0.04670331034630615

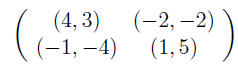
PH2A: 0.9532966896536939

Принимаем вторую гипотезу

# Глава 5 Графская иллюстрация множества достижимых итогов в неантагонистической биматричной игре

## 5.1 Теоретическая часть

Неантагонистические игры — это игры, в которых интересы сторон не обязательно противоположны. Такие игры описываются матрицами, например, такого вида:



В каждой клетке матрицы стоит пара чисел: выигрыши первого и второго игроков соответственно. Например, если первый игрок применит вторую стратегию, а второй применит первую стратегию, то первый игрок проиграет 1 единицу, а второй проиграет 4 единицы. Игры такого типа называют биматричными. Как видно, интересы игроков могут частично совпадать. В данной лабораторной работе рассматривается случай некооперативного поведения игроков: игроки полностью лишены возможности как-либо договариваться и согласовывать свои ходы. В таких случаях, как и в антагонистических играх, бывает уместно применять смешанные стратегии. Пусть дана матрица биматричной игры 2×2. Пусть каждый из игроков применяет смешанные стратегии: первый выбирает свою первую стратегию с вероятностью x, а вторую \_ с вероятностью 1 − x. Аналогично, второй игрок выбирает свои стратегии с вероятностями y и 1 − y. Тогда могут быть подсчитаны средние выигрыши обоих игроков: S1(x, y) для первого игрока и S2(x, y) для второго. При заданных вероятностях x и y выигрыш может быть изображен точкой (S1(x, y), S2(x, y)) в плоскости с координатами S1, S2. При изменении вероятностей выбора стратегий x и y эта точка также будет меняться. Для всех возможных наборов (x, y) точки (S1(x, y), S2(x, y))будут заполнять некоторую область в плоскости с координатами S1, S2.Таким образом, пара функций (S1(x, y) и S2(x, y)) задает отображение единичного квадрата [0, 1] × [0, 1] в плоскости x, y на некоторую область в плоскости S1, S2. Эта область и есть множество достижимых итогов.

## 5.2 Программный код

|  |
| --- |
| *"""*  *Программа, иллюстрирующая множество всех возможных исходов в неантагонистической*  *биматричной игре для случая некооперативного поведения игроков*  *Автор: Афанасьев И.Е.*  *Дата написания: 20.09.2020*  *"""*  *# Матрица*  *#m = [[2, 1], [-1, -1], [-1, -1], [1, 2]]*  m = [[4, 3], [-2, -2], [-1, -4], [1, 5]]  *# Импортируем библиотеку для поиска экстремумов*  **from** **scipy.optimize** **import** minimize  *# формула выигрыша первого игрока*  **def** S1(x):  **return** m[0][0]\*x[0] \* x[1] \  + m[1][0]\*(1 - x[0]) \* x[1] \  + m[2][0]\*x[0] \* (1 - x[1]) \  + m[3][0]\*(1 - x[0]) \* (1 - x[1])  *# формула выигрыша второго игрока*  **def** S2(x):  **return** x[0] \* m[0][1]\*x[1] \  + (1 - x[0]) \* m[1][1]\*x[1] \  + x[0] \* m[2][1]\*(1 - x[1]) \  + (1 - x[0]) \* m[3][1]\*(1 - x[1])  *# Ищем экстремальную точку для первого игрока*  res = minimize(S1, [1, 1])  print("Экстремум для первого игрока: ", res.x)  print("Значение выигрыша: ", S1(res.x))  *# Ищем экстремальную точку для второго игрока*  res = minimize(S2, [1, 1])  print("Экстремум для второго игрока: ", res.x)  print("Значение выигрыша: ", S2(res.x))  *# Библиотека для матричных вычислений, нам она нужна для создания*  *# массива от 0 до 1*  **import** **numpy** **as** **np**  xval = np.linspace(0, 1, 51)  yval = np.linspace(0, 1, 51)  *# Библиотека для графиков*  **import** **matplotlib.pyplot** **as** **plt**  x, y = np.meshgrid(xval, yval)  z1 = S1([x, y])  z2 = S2([x, y])  plt.scatter(z1, z2)  plt.show() |

Код 5. «Программа, иллюстрирующая множество всех возможных исходов в неантагонистической биматричной игре для случая некооперативного поведения игроков»

## 5.3 Результаты работы программы

Пусть:

m = [[1, 2], [-1, -1], [-1, -1], [2, 1]]

Результат:

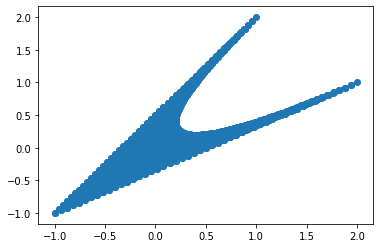
Экстремум для первого игрока: [0.6 0.6]

Значение выигрыша: 0.20000000000000007

Экстремум для второго игрока: [0.4 0.4]

Значение выигрыша: 0.2

Область достижимых исходов:



Пусть:

m = [[4, 3], [-2, -2], [-1, -4], [1, 5]]

Область достижимых исходов:



Пусть:

m = [[2, 3], [-2, -1], [-1, -2], [3, 2]]

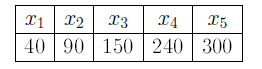
Область достижимых исходов:



# Глава 6 Вычисление вектора Шепли

## 6.1 Теоретическая часть

Вектор Шепли — это один из возможных подходов к решению проблемы справедливого дележа. Мы рассмотрим его на примере модели «Малое Гадюкино». Село Малое Гадюкино примыкает к автомобильной дороге. Все n домов в этом селе стоят на одной-единственной улице и находятся на расстояниях x1, x2, . . . , xn метров от автодороги. Жители села решили заасфальтировать эту улицу. Каждый метр асфальта стоит 1. Как жителям распределить взносы? Мы можем с самого начала считать, что все расстояния упорядочены, то есть x1 < x2 < . . . < xn. Каждый из Гадюкинцев хотел бы заплатить поменьше. Вообще, каждый из Гадюкинцев хотел бы участвовать в финансировании дороги только до своего дома, потому что остальной частью улицы он не пользуется. Тем не менее общие интересы требуют асфальтирования всей улицы. Концепция вектора Шепли предлагает такой рецепт. Допустим, что в число плательщиков все жители деревни добавляются в некотором определенном порядке. Если дорога до вновь добавившегося жителя не оплачена, он оплачивает строительство участка от уже построенного ранее до его дома. Если дорога до его дома уже оплачена, то он не платит ничего. Всех n жителей можно расположить в некотором порядке n! способами. Набор усредненных по всем n! перестановкам платежей для всех жителей деревни и есть вектор Шепли. Рассмотрим пример. Пусть n = 5, а расстояния до дороги приведены в следующей таблице



Рассмотрим перестановку (42153). Первым в число плательщиков добавляется четвертый житель, и он оплачивает 240 метров дороги. Следующими добавляются второй и первый жители, и они не доплачивают ничего, потому что дорога до них уже про финансирована. Следующим добавляется пятый житель, и он оплачивает оставшиеся до него 60 метров дороги. Третий житель также ничего не платит.

Рассмотрим теперь перестановку (21534). В этом случае второй оплачивает 90 метров, пятый оплачивает 210 метров, а остальные не платят ничего. Для других перестановок платежи, разумеется, будут другие. Из 5 объектов можно составить 5! = 120 перестановок. Усредненные по всем этим перестановкам платежи и есть вектор Шепли.

## 6.2 Программный код

|  |
| --- |
| *"""*  *Программа, вычисляющая вектор Шепли*  *Автор: Афанасьев И.Е.*  *Дата написания: 20.09.2020*  *"""*  *# импортируем перестановки*  **from** **itertools** **import** permutations  *# функция, вычисляющая факториал числа*  **def** fact(i):  **if** i <= 1:  **return** 1  **return** i \* fact(i - 1)  *# Задаём расстояния до домов ("Малое Гадюкино")*  X = [40, 90, 150, 240, 300, 500]  *# Порядковые номера домов*  X\_i = [i **for** i **in** range(len(X))]  *# Факториал от числа домов*  factX = fact(len(X))  *# Перестановки всех домов (И дублирование для номеров домов)*  perm = permutations(X)  perm\_i = list(permutations(X\_i))  *# Характеристическая функция*  hf = []  **for** i **in** perm:  hf1 = []  **for** j **in** range(len(i)):  **if** j == 0:  hf1 += [i[j]]  **else**:  *# Отражает вклад каждого жителя в постройку дороги*  hf1 += [i[j] - max(i[:j]) **if** i[j] - max(i[:j]) > 0 **else** 0]  hf += [hf1]  *# Инициализируем вектор Шепли*  shapley = [0]\*len(X)  *# Пробегаемся по всем перестановкам*  **for** i **in** range(factX):  **for** j **in** range(len(X)):  **for** k **in** range(len(shapley)):  **if** perm\_i[i][j] == k:  shapley[k] += hf[i][j]  *# Усредняем*  **for** i **in** range(len(shapley)):  shapley[i] /= factX  print(shapley) |

## 6.3 Результаты работы программы

При X = [40, 90, 150, 240, 300]

Шепли:

[8.0, 20.5, 40.5, 85.5, 145.5]

При X = [40, 90, 150, 240, 300, 500]

Шепли:

[6.666666666666667, 16.666666666666668, 31.666666666666668, 61.666666666666664, 91.66666666666667, 291.6666666666667]

# Заключение

В ходе данного курсового проекта была реализована задача, поставленная техническим заданием.

Были изучены теоретические основы к каждому заданию и выполнена как программная работа (написание исходного кода согласно поставленной задачи), а также выполнены аналитические расчеты в случае необходимости и проведено сравнение с полученными практическими результатами.

Данный курсовой проект представляет практическую значимость, так как показывает не только аналитическую значимость «Теории игр», но и возможную практическую реализацию средствами языка программирования.

# Список литературы

1. Вентцель Е. С. Исследование операций. М.: Наука, 1980-208с.
2. Данилов В.И. Лекции по теории игр. М.: ЭШ, 2002: [Электронный ресурс]

http://www.nes.ru/RUssIan/researh/abstrats/2002/Danilov-r.htm

(Дата обращения: 20.09.2020)

1. Демешев Б. Кооперативная теория игр: [Электронный ресурс]

http://demeshev.\_les.wordpress.om/2010/06/oop\_gt1.pdf

(Дата обращения: 20.09.2020)

1. Докинз . Эгоистичный ген. М.: Мир, 1993.-318 с.
2. Коковин С. . Лекции по теории игр и политологии.

http://math.ns.ru/matheon/Kokovin/m1tigran.pdf

(Дата обращения: 20.09.2020)

1. Колобашкина Л. В., Алюшин М. В. Информационные технологии

принятия решений в условиях конфликта. Часть 1. Основы теории

игр. М.: МИФИ, 2010-64 с.

1. Пиндайк . С., Рубинфельд Д. Л. Микроэкономика. М.: Дело, 2001-771с.
2. Феллер В. Введение в теорию вероятностей и ее приложения. М.:

Мир, 1984-528с.

1. Авинаш Диксит, Барри Нейлбфф. Теория игр. Искусство стратегического мышления в бизнесе и жизни. М.: Манн, Иванов и Фербер, 2015-464с.