Determining Optimal Strategies in Matrix Games

Hemal Ramesh Patel

Bachelor of Science in
Computer Information Systems with Honours
The University of Bath
May 2007
Determining Optimal Strategies in Matrix Games

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed: Hemal Patel
Determining Optimal Strategies in Matrix Games

Submitted by: Hemal Ramesh Patel

COPYRIGHT
Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration
This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: Hemal Patel
Determining Optimal Strategies in Matrix Games

Abstract

The origin of game theory can be traced back to the 18th century, but the development of the topic began with the works of John Von Neuman, Oskar Morgenstern and John Forbes Nash Jr.

This dissertation looks at the history of game theory and applications in today’s world. Depth into solving Zero-Sum Two Person Matrix Games are investigated and implemented.
Contents

CONTENTS ......................................................................... IV.
LIST OF FIGURES ............................................................. VII.
LIST OF TABLES ............................................................... IX.
ACKNOWLEDGEMENTS .................................................... X.
INTRODUCTION .................................................................. 11.
LITERATURE SURVEY ..................................................... 13.

2.1 INTRODUCTION .......................................................... 13.
2.2 GAME THEORY .......................................................... 13.
  2.2.1 What is Game Theory ............................................... 13.
  2.2.2 Brief History of Game Theory ..................................... 14.
  2.2.3 Application .......................................................... 14.
2.3 GAMES AND REPRESENTATIONS OF GAMES ................. 14.
  2.3.1 Definition of Games ................................................ 14.
  2.3.2 Normal and Extensive Form of Games ....................... 14.
  2.3.3 Zero-Sum Games .................................................. 15.
2.4 COMPUTATION OF OPTIMAL STRATEGIES .................. 16.
  2.4.1 Saddle Points ...................................................... 16.
  2.4.2 Domination ........................................................ 16.
  2.4.3 Solving 2*2 Matrix Games with no Saddle Points ....... 18.
  2.4.4 Solving M*N Matrix Games .................................... 20.
  2.4.5 Linear Programming .............................................. 21.
   2.4.5.1 What is a Basic Linear Programming Situation? .... 21.
   2.4.5.2 Simplex Method .............................................. 24.
  2.4.5.2 How to Solve a Linear Program using Simplex Method .... 26.
2.4 SUMMARY ................................................................... 29.

REQUIREMENTS .............................................................. 30.

3.1 INTRODUCTION .......................................................... 30.
3.2 REQUIREMENTS ANALYSIS ......................................... 30.
3.3 REQUIREMENTS SPECIFICATION ................................. 30.
  3.3.1 2*2 Matrix Games ................................................. 31.
  3.3.2 2*N and M*2 Matrix Games ..................................... 32.
  3.3.3 Graphical Method ................................................ 32.
  3.3.4 Simplex Method .................................................. 32.
  3.3.5 Non-Functional Requirements ................................. 33.
  3.3.6 Hardware Requirements ......................................... 33.
3.4 REQUIREMENTS VALIDATION ...................................... 33.
Determining Optimal Strategies in Matrix Games

3.5 REQUIREMENTS VERIFICATION ................................................................. 34.

DESIGN ........................................................................................................... 35.

4.1 INTRODUCTION ...................................................................................... 35.

4.2 DATA TYPES AND STRUCTURE ......................................................... 35.

4.2.1 Data Types ...................................................................................... 36.

4.2.2 Data Structures ............................................................................. 36.

4.3 PROGRAM DESIGN SPECIFICATION ............................................... 36.

4.3.1 Structure of System ....................................................................... 36.

4.3.2 Class 2*2 Matrix Games ............................................................... 37.

4.3.2.1 Saddle Point ............................................................................ 37.

4.3.2.2 Two by Two Matrix Games .................................................... 37.

4.3.2.3 Specification ........................................................................... 38.

4.3.2.4 Pseudo-code ........................................................................... 40.

4.3.3 Class 2*N Matrix Games .............................................................. 45.

4.3.3.1 Two by N Matrix Games ....................................................... 45.

4.3.3.2 Special Cases of Two by N Matrix Games ............................. 47.

4.3.3.3 Specification ........................................................................... 48.

4.3.3.4 Pseudo-code ........................................................................... 49.

IMPLEMENTATION ..................................................................................... 53.

5.1 INTRODUCTION ................................................................................. 53.

5.2 PROGRAMMING LANGUAGES ......................................................... 53.

5.3 IMPLEMENTATION ............................................................................ 54.

5.3.1 Two by Two Matrix Games .......................................................... 54.

5.3.1.1 Discussion of Algorithm ..................................................... 54.

5.3.1.2 Discussion of Programming .................................................. 55.

5.3.2 Two by N Matrix Games ............................................................... 55.

5.3.2.1 Discussion of Algorithm ..................................................... 55.

5.3.2.2 Discussion of Programming .................................................. 58.

TESTING ..................................................................................................... 62.

6.1 INTRODUCTION ................................................................................. 62.

6.2 SOFTWARE TESTING ................................................................. 62.

6.3 TEST SCRIPTS .................................................................................. 63.

6.3.1 Two by Two Matrix Class .......................................................... 64.

6.3.2 Two by N Matrix Class ............................................................... 64.

CONCLUSIONS ............................................................................................ 65.

7.1 INTRODUCTION .................................................................................. 65.

7.2 ANALYSIS OF TESTING ................................................................. 65.

7.2.1 Testing of 2*2 Matrix Class ....................................................... 66.

7.2.2 Testing of 2*N Matrix Class ....................................................... 66.

7.3 PROJECT Evaluation ................................................................. 67.

7.4 POSSIBLE EXTENSIONS ................................................................. 67.

7.5 PERSONAL EVALUATION ................................................................. 68.

BIBLIOGRAPHY ......................................................................................... 69.
Determining Optimal Strategies in Matrix Games

APPENDIX A – GLOSSARY
APPENDIX B – LINEAR ALGEBRA
APPENDIX C - SOURCE CODE
APPENDIX D – TEST CASES
APPENDIX E – TEST DATA
APPENDIX F – RAW RESULTS OUTPUT
APPENDIX G- ADDITIONAL TESTING
List of Figures

FIGURE 2.3.2A GAME SHOWN IN EXTENSIVE FORM ..................................................15.
FIGURE 2.3.2 A MATRIX .......................................................................................15.
FIGURE 2.3.3A ZERO-SUM MATRIX GAME ..........................................................15.
FIGURE 2.4.1A MATRIX WITH NO SADDLE POINTS ...........................................16.
FIGURE 2.4.1B MATRIX WITH A SADDLE POINT ..............................................16.
FIGURE 2.4.2A EXAMPLE MATRIX FROM OWEN (1982) .......................................17.
FIGURE 2.4.2B REMAINING MATRIX ....................................................................17.
FIGURE 2.4.2C REMAINING MATRIX ....................................................................18.
FIGURE 2.4.2D FINAL MATRIX ..............................................................................18.
FIGURE 2.4.2E THREE BY FOUR MATRIX REDUCED TO A TWO BY TWO MATRIX ...18.
FIGURE 2.4.3A EXAMPLE 2*2 MATRIX FROM OWEN (1982) ...............................19.
FIGURE 2.4.4B GRAPHICAL REPRESENTATION OF THE EXAMPLE MATRIX ........21.
FIGURE 2.4.4C GRAPHICAL REPRESENTATION OF THE EXAMPLE MATRIX ........21.
FIGURE 2.4.5.1A GRAPHICAL SOLUTION TO THE LINEAR PROGRAM ................23.
FIGURE 4.3.2.1 EXAMPLES OF GAMES WITH AND WITHOUT SADDLE POINTS ....37.
FIGURE 4.3.2.2A TWO BY TWO MATRIX GAME ..............................................37.
FIGURE 4.3.3.1A TWO BY N MATRIX GAME ....................................................45.
FIGURE 4.3.3.1B EXAMPLE FROM OWEN (1982) ................................................46.
FIGURE 4.3.3.1C GRAPHICAL REPRESENTATION OF 2*N MATRIX .....................46.
FIGURE 4.3.3.2A SPECIAL CASE 2 OF 2*N MATRIX ...........................................47.
FIGURE 4.3.3.2B SPECIAL CASE 3 OF 2*N MATRIX ...........................................47.
FIGURE 5.3.2.1A SOLVING CASE 2 OF 2*N MATRIX GAMES ..............................56.
FIGURE 5.3.2.1B SOLVING CASE 3 OF 2*N MATRIX GAMES ..............................57.
FIGURE B.1A DETERMINANT ...............................................................................73.
FIGURE D.2A ADJOINT OF MATRIX ....................................................................73.
FIGURE B.3A MULTIPLICATION OF VECTORS AND MATRIX ...............................74.
FIGURE B.4A SCALAR MULTIPLICATION OF MATRIX .........................................74.
FIGURE B.5A DOT PRODUCT OF TWO VECTORS ..............................................74.
FIGURE E.2.1.1A 2 *3 MATRIX CASE 1 ..............................................................113.
FIGURE E.2.1.1B 2 *3 MATRIX CASE 2 ..............................................................113.
FIGURE E.2.1.1C 2 *3 MATRIX CASE 3 ..............................................................114.
FIGURE E.2.1.2A 2 *4 MATRIX CASE 1 ..............................................................114.
FIGURE E.2.1.2C 2 *4 MATRIX CASE 3 ..............................................................115.
FIGURE E.2.1.3A 2 *5 MATRIX CASE 1 ..............................................................116.
FIGURE E.2.1.3C 2 *5 MATRIX CASE 3 ..............................................................117.
FIGURE G.1A 2 *5 MATRIX CASE 1 .................................................................169.
Determining Optimal Strategies in Matrix Games

FIGURE G.1B 2 *5 MATRIX CASE 1 .............................................................. 169.
Determining Optimal Strategies in Matrix Games

List of Tables

TABLE 2.3.2A GAME SHOWN IN NORMAL FORM 14.
TABLE 2.4.5.2A TABLE SHOWING THE ENTRIES AND THE CONSTRAINT 24.
TABLE 2.4.5.2B TABLE 2.4.5.2A PLUS THE OBJECTIVE FUNCTION 24.
TABLE 2.4.5.2C DUAL TABLE 25.
TABLE 2.4.5.2D SIMPLIFIED TABLE 25.
TABLE 2.4.5.3A INITIAL TABLE 27.
TABLE 2.4.5.3B TABLE AFTER FIRST SUBSTITUTION 28.
TABLE 2.4.5.3C FINAL TABLE AFTER SECOND SUBSTITUTION 29.
Acknowledgements

I would like to acknowledge my project supervisor, Professor Nicolai Vorobjov for his valuable and constant assistance throughout the lifecycle of this dissertation. I would also like to give thanks to Richard Walklate and David Murrant for their assistance. Finally I would like to thank my family, especially my mother, for her constant support and encouragement throughout my educational life.
Chapter 1

Introduction

Games are defined as a competitive activity involving skill, chance, or endurance on the part of two or more persons who play according to a set of rules, usually for their own amusement or for that of spectators.

Games could be in the form of physical activity such as sports or mental activity such as parlour games. The common characteristic of all games is that players will have a strategy that they intend to follow through. The strategies that the players will adopt will depend on the skill of the player or random chance/luck.

Take three parlour games; chess, poker and roulette. These types of games do vary with the application of chance and skill, therefore altering the strategies that the players adopt.

In the game of chess, chance or luck does not exist as the moves are skilfully thought through and planned ahead to devise a checkmate. Poker combines chance and skill, as the cards that a player will play with, will depend on the “hand” of the player and the other players’ cards. The game of roulette is a game of chance where skill plays no part.

Common with all games is the amount of knowledge available to the players. The knowledge that the players have can assist in determining the strategies that they can adopt. In the game of chess, knowledge is perfect as the moves that the players have made are known throughout the game. The strategies adopted by the players can complement the knowledge that they have.

Poker unlike chess has imperfect knowledge because the player will not know the deck of the opponents and therefore the strategy will be based on the cards that the player holds.

The main objective of all games is to win and in particular to maximise the win. Winning could be in the form of pride, trophies or money. In order to win at any game the players of the game will have to find and play their best strategies.
Determining Optimal Strategies in Matrix Games

The strategy in games is studied in a field of mathematics and economics and is known as **Game Theory**. Game theory will be the chosen domain for this dissertation.

The aims of the project are to research into the topic of Game Theory and design and implement an application to determine strategies and payoffs for players, for a given game. The project dissertation will be split into several sections.

The first section will contain the literature review, which will provide the foundation of the project and research into the existing solutions of the problem. Following the literature review will be the requirements section, which will formally specify the functionality of the application. Leading the requirements will be the design section. This section will discuss the algorithms needed to solve the problem. The implementation section will discuss the implementation of the algorithms mentioned in the design section. The next section will be the testing, which will detail the testing of the application. The project dissertation will conclude with the conclusion. The appendix will contain the source code and test data for the application.

Attached along with the dissertation will be a CD containing all of the code.
Chapter 2

Literature Survey

2.1. Introduction

The purpose of this literature review is to investigate the topic of game theory. The topic needs to be fully understood, from what the topic briefly is, how it has evolved, who uses game theory and its applications.

Dawson (2000) states that the literature review process, serves the following purposes:

- It justifies the project such that it shows that the project is worth doing and the area being investigated, in this case Game Theory, is recognised and meaningful.
- It sets the project within context by analysing past and current research in the chosen area. Through this, the identification of how the project fits within and contributes to area is determined.
- It allows other researchers to use the work done on the project, for their own research purposes and continue where the project left off.

The literature review will follow the structure of an overview to what game theory is from a brief history, the applications of game theory and representation of games, to the computation of optimal strategies in particular types of games. Concluding the literature review will be a summary.

2.2. Game Theory

2.2.1 What is Game Theory?

Game Theory is how decision makers interact with each other in a competitive sense and the process they take in making a rational/strategic choice (Osborne 2004) and a choice that is the best response to the other player’s strategy (McCain 2004).
2.2.2 Brief History of Game Theory

Early parts of game theory can be traced back to the 18th century, however game theory went through major developments in the early part of the 20th century, which lead to a book being written by one of the “godfathers” of game theory, mathematician John Von Neumann. His book was called, Theory of Games and Economic Behaviour (Osborne 2004).

In 2001 the game theory was glamorised by the red carpets of Hollywood, as the film, A Beautiful Mind that starred Russell Crowe as the economist John Forbes Nash (McCain 2004), who in the 1950’s:

“developed a key concept (Nash Equilibrium) and initiated the game-theoretic study of bargaining” – Osborne (2004, p2).

2.2.3 Applications

Game theory is often described as a branch of applied mathematics and economics. However game theory has got an interest in vast amounts of academic subjects. This ranges from Biology to Sociology and also including Computer Science (Wikipedia 2006 and McCain 2004).

Computer scientists are interested in game theory, as game theory has got important links with logic and several logical theories have a basis in game semantics (Wikipedia 2006).

Game theory also has a wide range of applications from business, auctions, military, political and more famously gambling (McCain 2004 and Osborne 2004).

2.3. Games and Representation of Games

2.3.1 Definition of Games

According to sources, Owen (1982) and Osborne (2004) the definition of a game consists of; a set of players, the strategies/choices that they have available to them and a payoff table/preference. The payoff for the strategies could either be in the form of money or pride etc...

2.3.2 Normal and Extensive Form of Games

Table 2.3.2a Game shown in Normal Form

<table>
<thead>
<tr>
<th></th>
<th>Confess</th>
<th>Don't</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joggs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Confess</td>
<td>10 yr, 10 yr</td>
<td>0, 20 yr</td>
</tr>
<tr>
<td>Don't</td>
<td>20 yr, 0</td>
<td>1 yr, 1 yr</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

Game theory has two common ways of representing games; either in table format (Normal Form (Table 2.3.2a)) or in tree format (Extensive Form (Figure 2.3.2a)).

Figure 2.3.2a and Table 2.3.2a are representations of the game Prisoner's Dilemma (McCain 2004). From those representations, it can be deduced that the strategies of games can be written in a matrix. This is also supported by Owen (1982).

The individual elements within the matrix represent the payoff for undertaking a certain strategy. Also for choosing a particular strategy, there is a probability that the strategy will and will not work. This is taken into account in the payoff. During a game, players may sometimes try to reduce the payoff amount that they have to pay. This is called the minimax theorem. This can be also seen as maximising the minimum gain (Owen 1982).

2.3.3 Zero-Sum Games

Matrix games can come in various forms, from symmetric and asymmetric to zero and non-zero-sum games (Osborne 2004).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2, −2</td>
<td>−1, 1</td>
</tr>
<tr>
<td>B</td>
<td>−1, 1</td>
<td>3, −3</td>
</tr>
</tbody>
</table>

Figure 2.3.3a Zero-Sum Matrix Game
Determining Optimal Strategies in Matrix Games

Owen (1982) and McCain (2004) both state that zero-sum matrix games are games such that the payoff inside the matrix, adds up to zero. Figure 2.3.3a shows this, as within each cell, the payoff adds up to zero.

2.4. Computation of Optimal Strategies

Owen (1982) states that due to the minimax theorem, in every two person zero-sum matrix game, optimal strategies will exist.

2.4.1 Saddle Points

The simplest method of computing optimal strategies in a two person zero-sum game, is determining whether a saddle point exists. A saddle point is element, $a_{ij}$ in the matrix, where it is both the maximum entry in its column and the minimum entry in its row.

$$
\begin{pmatrix}
5 & 5 \\
6 & 5
\end{pmatrix}
$$

Figure 2.4.1a Matrix with No Saddle Points

Looking at Figure 2.4.1a, it can be seen that within the matrix there are no saddle points.

$$
\begin{pmatrix}
2 & 5 \\
1 & 0
\end{pmatrix}
$$

Figure 2.4.1b Matrix with Saddle Point

Looking at Figure 2.4.1b, it can be seen that the element $a_{11}$, is the saddle point as according to the definition stated by Owen (1982), the element $a_{11}$, is the maximum value in its’ column and the minimum value its’ row. From this it can be seen that this entry is the optimal strategy for the two players and that the probabilities of the players choosing this option will be one (i.e. $x_i = 1$ and $y_j = 1$) and the rest of the entries in the matrix will have a probability of zero, as the players will not chose those options.

Saddle points work well with 2*2 matrices, however it can also work with matrices of sizes greater than 2*2.

2.4.2 Domination

Owen (1982) discusses the usage of a term called domination, when dealing with matrices of sizes greater than 2*2.

"Domination. In a matrix, $A$, we say the $i^{th}$ row dominates the $k^{th}$ row if $a_{ij} \geq a_{kj}$ for all $j$.\)
Determining Optimal Strategies in Matrix Games

And
\[ a_{ij} \geq a_{kj} \quad \text{for at least one } j \]
Similarly, we say the \( j^{\text{th}} \) row dominates the \( i^{\text{th}} \) row if
\[ a_{ij} \geq a_{il} \quad \text{for all } i \]
And
\[ a_{ij} \geq a_{il} \quad \text{for at least one } i \]

A pure strategy is said to be a dominate strategy when that strategy offers a bigger payoff, regardless of which strategy the other player has chosen (Owen 1982 and McCain 2004). When deciding upon strategies, players will choose the strategies that give them the best payoff, therefore deciding not to choose the other player’s dominate strategy.

The idea behind domination is to reduce the size of the matrix as following example shows (Owen 1982, pp.22-23);

There are two players; Player I and Player II where;
Player I is trying to minimise the payout that he would have to pay to Player II, i.e. make the loss as minimum as possible.
Player II is trying to maximise the payoff paid by Player I, i.e. make the payout as high as possible.

Figure 2.4.2a is the matrix of the game that the players above are playing;

\[
\begin{pmatrix}
2 & 0 & 1 & 4 \\
1 & 2 & 5 & 3 \\
4 & 1 & 3 & 2 \\
\end{pmatrix}
\]

Figure 2.4.2a Example Matrix from Owen (1982)

The game begins with Player I. Since Player I’s strategy is to minimise the loss, Player I would remove the columns.
Looking at the matrix, column two is dominate over column four as the payoffs in column four are greater than the payoffs in column two. This results in Player I discarding column four and making the probability that column four is chosen, zero.

\[
\begin{pmatrix}
2 & 0 & 1 \\
1 & 2 & 5 \\
4 & 1 & 3 \\
\end{pmatrix}
\]

Figure 2.4.2b Remaining Matrix

Player II now makes his move from the remaining values in the matrix. Player II’s strategy is to maximise the payoff received from Player I, so Player II is trying to remove the rows that would result in a low payoff received.
Looking at the remaining values in the matrix, row three is dominate over row one, as the payoff gained from row three is greater than the payoff received.
Determining Optimal Strategies in Matrix Games

from row one. With this deduction, Player II removes the dominated row and like with the removed column, the probability of that row chosen, is zero.

\[
\begin{pmatrix}
2 & 0 & 1 \\
1 & 2 & 5 \\
4 & 1 & 3 \\
\end{pmatrix}
\]

Figure 2.4.2c Remaining Matrix

Back to Player I. With the leftover matrix, column three is dominated by column one resulting in Player I, discarding the corresponding column.

\[
\begin{pmatrix}
2 & 0 & 1 \\
1 & 2 & 5 \\
4 & 1 & 3 \\
\end{pmatrix}
\]

Figure 2.4.2d Final Matrix

With the remaining elements this has resulted in three by four matrix to be reduced to a two by two matrix (which offers the final payouts, chosen by the players), as Figure 2.4.2e shows;

\[
\begin{pmatrix}
2 & 0 & 1 & 4 \\
1 & 2 & 5 & 3 \\
4 & 1 & 3 & 2 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 \\
4 & 1 \\
\end{pmatrix}
\]

Figure 2.4.2c Three by Four Matrix Reduced to a Two by Two Matrix

The current matrix is in the best optimal state for Player I, because regardless of which column Player II decides to chose, Player I can chose the row, in which the payout would be at a minimum.

For instance, if Player II decides to choose column one, Player I can respond in choosing row one. This would mean that element (1, 1) in Figure 2.4.2e is chosen and the payoff would be 1, which is the minimum payoff.

The second option open to Player II, is to choose the second column. With that selection, Player I again would choose the row that gives the lowest payout. The corresponding row chosen is row two. This would mean that the element (2, 2) is chosen and the payoff again would be 1, which again is the minimum payoff.

2.4.3 Solving 2*2 Matrix Games with no Saddle Points

With a given two by two matrix game, optimal strategies and a payoff can be determined as stated earlier by finding the saddle point. What happens if no saddle point exists? How are the optimal strategies and payoffs of the players determined?
Determining Optimal Strategies in Matrix Games

The answers to those questions are simple, as Owen (1982) states a theorem which determines optimal unique strategies and payoff for 2*2 matrices that do not contain any saddle points.

The theorem states that the optimal strategies for Player I represented by \( x \), Player II represented by \( y \) and the payoff, \( v \), will be determined from the following formulas:

\[
x = \frac{JA^*/JA^*J^T}{|A|/JA^*J^T} \\
y = \frac{A*J^T / JA^*J^T}{|A|/JA^*J^T} \\
v = \frac{|A|/JA^*J^T}{|A|/JA^*J^T}
\]

(2.4.3.1) \hspace{1.5cm} (2.4.3.2) \hspace{1.5cm} (2.4.3.3)

Where \( A \) is the matrix, \( |A| \) is the determinant of \( A \), \( A^* \) is the adjoint of \( A \) and \( J \) is the vector \((1, 1)\) and \( J^T \) is the transpose of \( J \).

E.g. Take the following example (Owen 1982, p.26)

\[
\begin{pmatrix}
1 & 0 \\
-1 & 2
\end{pmatrix}
\]

Figure 2.4.3a Example 2*2 Matrix from Owen (1982)

It can be seen that there are no saddle points. To determine the solutions, the formulas (2.4.3.1), (2.4.3.2) and (2.4.3.3) must be applied.

\[
A = \begin{pmatrix}
1 & 0 \\
-1 & 2
\end{pmatrix} \hspace{1cm} |A| = 2 \\
A^* = \begin{pmatrix}
2 & 0 \\
1 & 1
\end{pmatrix} \hspace{1cm} A^*J^T = (2, 2) \\
JA^* = (3, 1) \hspace{1cm} JA^*J^T = 4
\]

From the values above, the strategies \( x \) and \( y \) for Players I and II can be determined and the payoff, \( v \), can also be determined.

\[
\begin{align*}
x &= \frac{JA^*/JA^*J^T}{|A|/JA^*J^T} \\
x &= (3, 1)/4 = (\frac{3}{4}, \frac{1}{4}) \\
y &= \frac{A*J^T / JA^*J^T}{|A|/JA^*J^T} \\
y &= (2, 2)/4 = (\frac{1}{2}, \frac{1}{2}) \\
v &= \frac{|A|/JA^*J^T}{|A|/JA^*J^T} \\
v &= 2/4 = \frac{1}{2}
\end{align*}
\]

The payoff is \( \frac{1}{2} \) and the optimal strategy for Player I is \((\frac{3}{4}, \frac{1}{4})\) and \((\frac{1}{2}, \frac{1}{2})\) for Player II. It can be checked that the strategies \( x \) and \( y \) are correct because strategies are the probabilities of each players’ move. Probabilities all have the common characteristics, which are as follows;

- They range from zero to one
- The sum of all probabilities must equal one
Determining Optimal Strategies in Matrix Games

This is backed up by Owen (1982) on page 13. Looking at the strategies of the two players, it can be clearly seen that the probabilities hold true, as the probabilities are not non-negative and the sums of the probabilities all add up to one.

As Dawson (2000) stated the literature review will have to be continued throughout the lifetime of the project, as new information and improved understanding of the topic will have to be documented.

The formulas (2.4.3.1), (2.4.3.2) and (2.4.3.3) are used to determine optimal strategies and payoffs in 2*2 matrix games. However there is another set of formulas that can compute optimal strategies and payoffs in a 2*2 zero-sum matrix games (Mathematical Physics – University of Ireland 2007).

The idea behind the new formulas is that no saddle points must exist and the first and last elements in the matrix must be greater than second and third, or the second and third elements must be greater than the first and fourth elements, within the matrix. Then the optimal strategies can be found with the probabilities x and y and the payoff being v given by;

\[ x = \frac{d - c}{a + d - b - c} \]  \hspace{1cm} (2.4.3.4)

\[ y = \frac{d - b}{a + d - b - c} \]  \hspace{1cm} (2.4.3.5)

\[ v = \frac{ad - bc}{a + d - b - c} \]  \hspace{1cm} (2.4.3.6)

Similar to formulas (2.4.3.1), (2.4.3.2) and (2.4.3.3), the probabilities will be in the form of \((x, 1-x)\) for Player I and \((y, 1-y)\) for Player II.

2.4.4 Solving M*N Matrix Games

2*2 matrix games are not the only size of matrices that exist in two person zero-sum games. Other matrices that exist within the subset of two person zero-sum matrix games are; 2*N and M*2 (or M*N). In these types of games the player has only two strategies to choose from.

Similar to 2*2 matrix games, players are trying to maximise their payoff. The solutions can be summed up in a linear graph, where the elements in the matrix are matched up, and used to plot a graph.

E.G. take for instance Figure 2.4.4a;

\[
\begin{pmatrix}
2 & 3 & 1 & 5 \\
4 & 1 & 6 & 0
\end{pmatrix}
\]

Figure 2.4.4a Example of a 2*N Matrix from Owen (1982)
Determining Optimal Strategies in Matrix Games

A graph can be drawn by plotting lines from \((0, a_{11})\) to \((1, a_{21})\), as the diagram below, shows.

![Figure 2.4.4b Graphical Representation of the Example Matrix](image)

The thick black line shows maximum values of all of the lines and represents the payoff. The maximum value is the point which is at the highest on that line and is denoted as \(x\). The optimal strategy at that the highest point is in the form \((x, 1-x)\), which according to the diagram above is \((\frac{2}{7}, \frac{5}{7})\) and the corresponding payoff is \(\frac{17}{7}\).

Should the player wish to minimise the payoff, i.e. minimise the loss gained, the graph would look like this;

![Figure 2.4.4c Graphical Representation of the Example Matrix](image)

The thick black line represents the loss. The minimum value is the lowest point on the line, which is denoted as \(z\). The optimal strategy at that the lowest point is in the form \((y, 1-y)\) and the payout is \(v\).

For a \(M*2\) matrix, a similar process used for \(2*N\) matrix, can be applied.

### 2.4.5 Linear Programming

A common solution to determine the best optimal strategy, in game theory, is to use **Linear Programming**. The word *programming* in this context should not be taken literally as it means planning (Bunday 1984).

According to Bunday (1984) the first ideas behind linear programming were developed during the Second World War. This also backs up McCain (2004), as he stated that applications of game theory were used by the military.

#### 2.4.5.1 What is a Basic Linear Programming Situation?

A basic linear programming problem or situation is either maximising or minimising a function due to a set of constraints. A common example is a firm trying to maximise profits, by producing two goods with resource constraints.
Determining Optimal Strategies in Matrix Games

Take the following example (Bunday 1984, pp.1-3);

A firm produces two types of goods, good A and good B.
The goods are limited to the amount of resources and machine time available.
Each unit of A needs 3m² of board and each unit of B needs 4m² of board. Firms can only get 1700m² of board, each week from suppliers.
To produce both goods, the same machinery must be used. The total machine time available is 160mins. A single unit of good A takes 12mins of machine time. A single unit of good B takes 30mins of machine time.
The revenue made from selling a single unit of good A is $2 and $4 for good B.

To determine linear equations replace good A with the term \(x_1\) and good B with the term \(x_2\).
From the above the company must decide how much of the goods they must produce, each week in order to maximise profits. The profit can be summed up in the following equation;
- \(P = 2x_1 + 4x_2\). This equation is known as the **objective function**.

The goal of the firm is to maximise the objective function.
In order to maximise the objective function, the company could just increase \(x_1\) and \(x_2\). This option is not feasible as there are certain constraints, mentioned earlier that the company must acknowledge.
Since the production of both goods each week can’t be negative, the following constraint is determined;
- \(x_1 \geq 0, x_2 \geq 0\).

The constraints of the amount of board and machine time that the firm can work with can be summed up in the following linear inequalities.
Board:
- \(3x_1 + 4x_2 \leq 1700\)

Machine Time:
- \(\frac{1}{5}x_1 + \frac{1}{2}x_2 \leq 160 \Rightarrow 2x_1 + 5x_2 \leq 1600\)

Using the inequalities mentioned the following graph can be produced, in which will solve the problem.
Determining Optimal Strategies in Matrix Games

Looking at the Figure 2.4.5.1a, the arrows satisfy the inequalities mentioned and the shaded region OABC represents the points \((x_1, x_2)\), in which can satisfy the constraints and the objective function. This area is called the feasible region and the points within the region and the boundary are called feasible solutions.

The goal of the firm is to maximise profits. In order to do so, the firm can look at the graph, take the objective function and find parallel lines of the function, going right of the objective function. Since the firm is maximising, the last feasible solution, to intersect the parallel lines is the optimal solution.

From looking at the graph, the optimal solution is at point B where \(x_1 = 300\) and \(x_2 = 200\). To check that the values are correct substitute the values back into the constraints;

\[
3x_1 + 4x_2 \leq 1700 \\
3(300) + 4(200) \leq 1700 \\
900 + 800 \leq 1700 \\
1700 \leq 1700
\]

\[
2x_1 + 5x_2 \leq 1600 \\
2(300) + 5(200) \leq 1600 \\
600 + 1000 \leq 1600 \\
1600 \leq 1600
\]

The solutions are optimal as they satisfy the inequalities. To determine the maximum profit, substitute the values into the objective function;

\[
P = 2x_1 + 4x_2 \\
P = 2(300) + 4(200) \\
P = 600 + 800 \\
P = 1400
\]

The maximum profit is $1400.
Determining Optimal Strategies in Matrix Games

The above example is also supported by the work of Lau (1984), as he stated that the definition of linear programming was a “class of problems” with the following characteristics:
1. All decision variables are non-negative
2. The objective function is a linear equation of the decision variables and requires be maximising or minimising.
3. There are constraints that are made up from the decision variables and are in the form or inequalities or equations.

2.4.5.2 Simplex Method Algorithm

The simplex method is an algebraic method used to determine optimal solutions for players’ mixed strategies and the payoff they can expect. This is supported by works of Owen (1982) and Lau (1984).

Simplex method begins with a set of inequalities, as shown below;

\[
\begin{align*}

\text{(2.4.5.2.1)} & \\

a_{11}x_1 + a_{21}x_2 + \ldots + a_{m1}x_m & \leq b_1, \\

a_{12}x_1 + a_{22}x_2 + \ldots + a_{m2}x_m & \leq b_2, \\

\vdots & \vdots & \ddots & \vdots \\

a_{1n}x_1 + a_{2n}x_2 + \ldots + a_{mn}x_m & \leq b_n,
\end{align*}
\]

The above is then put together to form a linear program, by including nonnegative constraints. The above can then be written to form the following;

\[
\begin{align*}

\text{(2.4.5.2.2)} & \\

a_{11}x_1 + a_{21}x_2 + \ldots + a_{m1}x_m & \leq b_1 = -u_1, \\

a_{12}x_1 + a_{22}x_2 + \ldots + a_{m2}x_m & \leq b_2 = -u_2, \\

\vdots & \vdots & \ddots & \vdots \\

a_{1n}x_1 + a_{2n}x_2 + \ldots + a_{mn}x_m & \leq b_n = -u_n,
\end{align*}
\]

The variables \(u_j\), where \(j\) can take the values from 1 to \(n\), are called slack variables, which are prohibited by the simplex method to take only positive values. This reduces the constraints of the system also to positive, i.e. \(x_i \geq 0\) and \(u_j \geq 0\). The purpose of slack variables is to turn a system of equalities (as shown in (2.4.5.2.1)) into a system of equations, as Table 2.4.5.2a shows.

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>\ldots</th>
<th>(x_m)</th>
<th>1</th>
<th>= (-u_1)</th>
<th>= (-u_2)</th>
<th>\ldots</th>
<th>= (-u_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_{11})</td>
<td>(a_{21})</td>
<td>\ldots</td>
<td>(a_{m1})</td>
<td>(-b_1)</td>
<td>(-u_1)</td>
<td>(-u_2)</td>
<td>\ldots</td>
<td>(-u_n)</td>
</tr>
<tr>
<td>(a_{12})</td>
<td>(a_{22})</td>
<td>\ldots</td>
<td>(a_{m2})</td>
<td>(-b_2)</td>
<td>(-u_1)</td>
<td>(-u_2)</td>
<td>\ldots</td>
<td>(-u_n)</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>(a_{1n})</td>
<td>(a_{2n})</td>
<td>\ldots</td>
<td>(a_{mn})</td>
<td>(-b_n)</td>
<td>(-u_1)</td>
<td>(-u_2)</td>
<td>\ldots</td>
<td>(-u_n)</td>
</tr>
</tbody>
</table>

From Table 2.4.5.2a a linear equation can be determined by taking the element within the table and multiplying it with the term outside the corresponding
Determining Optimal Strategies in Matrix Games

column and equalling the total sum to the term outside the corresponding row on the right.

E.G. Take the first row;
The equation that can be formed is as follows;
\[ a_{11}x_1 + a_{21}x_2 + \ldots + a_{m1}x_m = b_1 \]
Etc...

With Table 2.4.5.2a, the objective function can now be introduced, as shown in Table 2.4.5.2b;

**Table 2.4.5.2b Table 2.4.5.2a plus the Objective Function**

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( \ldots )</th>
<th>( x_m )</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{11} )</td>
<td>( a_{21} )</td>
<td>( \ldots )</td>
<td>( a_{m1} )</td>
<td>(-b_1)</td>
</tr>
<tr>
<td>( a_{12} )</td>
<td>( a_{22} )</td>
<td>( \ldots )</td>
<td>( a_{m2} )</td>
<td>(-b_2)</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( a_{1n} )</td>
<td>( a_{2n} )</td>
<td>( \ldots )</td>
<td>( a_{mn} )</td>
<td>(-b_n)</td>
</tr>
<tr>
<td>( c_1 )</td>
<td>( c_2 )</td>
<td>( \ldots )</td>
<td>( c_m )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

The objective function is represented by the term \( w \).

Looking at Table 2.4.5.2b, a linear program can be produced. Taking into consideration that \( x \) corresponds to Player I, the table can also give out a linear program for Player II, dubbed as \( y \). This can be seen as a dual program, in which the table can give out linear equations for Player II.

**Table 2.4.5.2c Dual Table**

\[
\begin{array}{cccc|c}
 x_1 & x_2 & \ldots & x_m & 1 \\
 y_1 & a_{11} & a_{21} & \ldots & a_{m1} & -b_1 \\
y_2 & a_{12} & a_{22} & \ldots & a_{m2} & -b_2 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
y_n & a_{1n} & a_{2n} & \ldots & a_{mn} & -b_n \\
-1 & c_1 & c_2 & \ldots & c_m & 0 \\
\end{array}
\]

Looking at Table 2.4.5.2c, there are \( n+1 \) equations and there are \( n+m+1 \) unknowns \((x_1, \ldots, x_m, u_1, \ldots, u_n \text{ and } w)\).

Currently the equations above all show \( u \) and \( w \) in terms of \( x \). To solve the equations, \( x \) must be shown in terms of \( u \), i.e. \( n+1 \) equations must be shown in terms of \( m+1 \).

Looking at the table above, the table can be made simpler to solve by replacing certain variables, due to the dual factor. The \( n+1 \) equations can be replaced with variables \( q_1, q_2, \ldots, q_n \) and \( w \), in terms with the \( m+1 \) unknowns,
Determining Optimal Strategies in Matrix Games

which can be replaced with the variables, \( f_1, f_2, \ldots, f_m \). Thus forming Table 2.4.5.2d, making the table easier to work with.

**Table 2.4.5.2d Simplified Table**

<table>
<thead>
<tr>
<th>( f_1 )</th>
<th>( f_2 )</th>
<th>( \ldots )</th>
<th>( f_m )</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a'_{11} )</td>
<td>( a'_{21} )</td>
<td>( \ldots )</td>
<td>( a'_{m1} )</td>
<td>( -b'_1 )</td>
</tr>
<tr>
<td>( a'_{12} )</td>
<td>( a'_{22} )</td>
<td>( \ldots )</td>
<td>( a'_{m2} )</td>
<td>( -b'_2 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( a'_{1n} )</td>
<td>( a'_{2n} )</td>
<td>( \ldots )</td>
<td>( a'_{mn} )</td>
<td>( -b'_n )</td>
</tr>
<tr>
<td>( c'_1 )</td>
<td>( c'_2 )</td>
<td>( \ldots )</td>
<td>( c'_m )</td>
<td>( \delta )</td>
</tr>
</tbody>
</table>

In the Table 2.4.5.2d every entry in the bottom row expect the last element and every element in the last column, again expect the last element, \( \delta \), must be negative. Once this occurs the solution should be present.

**2.4.5.3 How to Solve a Linear Program using Simplex Method**

In-order to solve the program, terms must be altered so that certain terms can be expressed in different terms. Lau (1984) and Owen (1982) both state that the variables outside the table on the far right hand column (the \( q_n \) variables) are called **basic variables** and the variables outside the table, on the top most row, (the \( f_m \) variables) are called **non-basic variables**.

To solve the linear program, the sets of variables will need to get changed one by one, (express the n+1 basic variables, 1 of the non-basic variables and the payoff, \( w \), in terms of the remaining m).

To do this Owen (1982) and Lau (1984) both state that **pivots** must be used. Pivots are functions that allow basic variables to be changed one by one.

When deciding upon the pivots to be used, there are two options that can be applied. Option one is choosing any \( c_\alpha \) from the bottom row excluding the last entry in the row, where \( c_\alpha \) is a positive entry. The pivot will be in the column of the entry chosen. To decide which pivot is to be chosen from the column, the \( -b_\alpha \) value will be divided by their corresponding entries in the \( c_\alpha \) column. The value will be negative and the value closest to zero will be the pivot. If there are any ties, then the element which gives the maximum value will be chosen as the pivot.

The second option is choosing the lowest positive \( -b_k \) entry, expect the last entry. The entries in the corresponding row of the chosen \( -b_k \) entry, could be the pivot. To determine the pivot, the \( -b_k \) value will be divided by their corresponding entries in the \( k^{th} \) row. The entry in the \( k^{th} \) row that gives the closest value to zero, when the \( -b_k \) value is divided by it, will be the pivot. Again if there is a tie, the element which gives the maximum value will be chosen.

Once the pivot has been chosen the linear program can be solved. As stated before, the sets of variables have to be altered.

I.E. take the following example;
\[ y = mx + c \]
Determining Optimal Strategies in Matrix Games

This is the equation of a linear line. Here y is in terms of x. To express x in terms of y, the equation would have to be modified to the following;

\[ x = (y - c)/m \]

With those equations, all of the values of x and y can be found out.
To fully show the solution of linear programs, using simplex method, take the following example (Owen 1982, p.53).

The objective function is to maximise the following;

\[ W. \ 5x_1 + 2x_2 + x_3 \]

Against the following constraints;

1. \( x_1 + 3x_2 - x_3 \leq 6 \)
2. \( x_2 + x_3 \leq 4 \)
3. \( 3x_1 + x_2 \leq 7 \)
4. \( x_1, \ x_2, \ x_3 \leq 0 \)

From the above the Table 2.4.5.3a can be obtained;

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>-1</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-4</td>
</tr>
<tr>
<td>*3</td>
<td>1</td>
<td>0</td>
<td>-7</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

From the explanation of choosing pivots, it can be seen that the entry \( a_{31} = 3 \), is a suitable pivot. This is indicated with the asterisk.

The column that the pivot is in is \( x_1 \) therefore \( x_1 \) must be expressed in terms of \(-u_3\). To do this the equation for row three is needed (this can be deduced by multiplying the entries in the table with the non-basic variables and equalling the sum to the basic variables, as explained earlier).

\[ 3x_1 + x_2 - 7 = -u_3 \]
\[ u_3 + x_2 - 7 = -3x_3 \]

\[ (\div3) \ 1/3u_3 + 1/3x_2 - 7/3 = -x_1 \]

Using the value of \(-x_1\), the other entries in the table can be deduced from substituting the value of \( x_1 \) into the other equations.

1. \[ x_1 + 3x_2 - x_3 - 6 = -u_1 \]
\[ - (1/3u_3 + 1/3x_2 - 7/3) + 3x_2 - x_3 - 6 = -u_1 \]
Determining Optimal Strategies in Matrix Games

\[- \frac{1}{3}x_3 - \frac{1}{3}x_2 + \frac{7}{3} + 3x_2 - x_3 - 6 = -u_1\]
\[- \frac{1}{3}u_3 + \frac{6}{3}x_2 - x_3 - \frac{11}{3} = -u_1\]

2. \[x_2 + x_3 - 4 = -u_2\]

W. \[5x_1 + 2x_2 + x_3 - 0 = w\]

\[-5\left(\frac{1}{3}u_3 + \frac{1}{3}x_2 - \frac{7}{3}\right) + 2x_1 + x_3 - 0 = w\]
\[-\frac{5}{3}u_3 - \frac{5}{3}x_2 + \frac{35}{3} + 2x_2 + x_3 - 0 = w\]
\[-\frac{5}{3}u_3 + \frac{1}{3}x_2 + x_3 + \frac{35}{3} = w\]

With the new entries, they can be substituted back into the table, with the changes that \(x_1\) is now in the basic variables grouping and \(u_3\) is in the non-basic variables grouping.

**Table 2.4.5.3b Table after First Substitution**

<table>
<thead>
<tr>
<th>(u_3)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(\frac{1}{3})</td>
<td>(\frac{8}{3})</td>
<td>-1</td>
<td>(-\frac{11}{3})</td>
</tr>
<tr>
<td>0</td>
<td>(\frac{1}{3})</td>
<td>-1</td>
<td>(-\frac{1}{3})</td>
</tr>
<tr>
<td>(\frac{1}{3})</td>
<td>(\frac{1}{3})</td>
<td>0</td>
<td>(-\frac{7}{3})</td>
</tr>
<tr>
<td>(-\frac{5}{3})</td>
<td>(\frac{1}{3})</td>
<td>1</td>
<td>(\frac{35}{3})</td>
</tr>
</tbody>
</table>

The process is repeated again because there are still some positive values left in the bottom row. This time the pivot is entry \(a_{23} = 1\), as shown by the asterisk. Now \(x_3\), must be expressed in terms of \(u_2\).

2. \[x_2 + x_3 - 4 = -u_2\]
\[u_2 + x_2 - 4 = -x_3\]

Now the value of \(x_3\) can be substituted into the other equations.

1. \[-\frac{1}{3}u_3 + \frac{6}{3}x_2 - x_3 - \frac{11}{3} = -u_1\]
\[-\frac{1}{3}u_3 + \frac{6}{3}x_2 + u_2 + x_2 - 4 - \frac{11}{3} = -u_1\]
\[-\frac{1}{3}u_3 + \frac{11}{3}x_2 + u_2 - \frac{23}{3} = -u_1\]

W. \[-\frac{5}{3}u_3 + \frac{1}{3}x_2 - (u_2 + x_2 - 4) + \frac{35}{3} = w\]
\[-\frac{5}{3}u_3 + \frac{1}{3}x_2 - u_2 - x_2 + 4 + \frac{35}{3} = w\]
\[-\frac{5}{3}u_3 - \frac{2}{3}x_2 - u_2 + \frac{47}{3} = w\]
Determining Optimal Strategies in Matrix Games

Again the new entries can be placed into a table, with the changes that \( x_3 \) is now in the basic variables grouping and \( u_2 \) is in the non-basic variables grouping.

Table 2.4.5.3c Final Table after Second Substitution

<table>
<thead>
<tr>
<th>( u_3 )</th>
<th>( x_2 )</th>
<th>( u_2 )</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\frac{1}{3})</td>
<td>(\frac{11}{3})</td>
<td>(-1)</td>
<td>(-\frac{11}{3})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(-4)</td>
</tr>
<tr>
<td>(\frac{1}{3})</td>
<td>(\frac{1}{3})</td>
<td>0</td>
<td>(-\frac{7}{3})</td>
</tr>
</tbody>
</table>

Looking at Table 2.4.5.3c, there are no positive entries in the bottom row and in the bottom column (excluding the entry in the position of \( a_{4,4} \)). Therefore the linear program has been solved. The solution is the values of the \( x_i \) variables in the basic variables section. If there aren’t any \( x_i \) variables in the basic variables section, that corresponding \( x_i \) variable is equal to zero.

From looking at Table 2.4.5.3c, it can be seen that the solution is in terms of \((x_1, x_2, x_3), \left(\frac{7}{3}, 0, 4\right)\) and the payoff, \( w \) is \(\frac{47}{3}\).

2.5. Summary

At the start of this literature review, the topic of game theory was introduced, with the origins of game theory, applications through to the types of games that exist.

Then the literature review went on to discuss how optimal strategies and payoffs were determined. The solutions that the discussed were of a graphical and algebraic nature and examples were shown. Following on, linear programming was discussed stating what linear programming was, by providing an example. A method of linear programming, called simplex method was discussed in depth as it is a method for working out optimal solutions for player’s strategies. An example using simplex method was also shown, and went through step by step discussing, how players’ optimal strategies and payoffs were determined.

A comparison can be made between the solutions available in determining optimal strategies and payoffs, in two person zero-sum matrix games. The solutions of games with saddle point are simple to solve, as are two by two matrix games with no saddle points.

However \( 2^N \) and \( M^2 \) (and effectively \( M^N \)) matrix games are not so simple to solve as with the graphical method, there is no algorithm presently available.

Although there is no algorithm available to compute optimal strategies and payoffs in \( N^M \) matrix games, linear programming is an avenue that can be explored, as the simplex method can be applied to determine optimal strategies and payoffs in \( N^M \) matrix games.
Chapter 3

Requirements

3.1. Introduction

The work undertaken before this document was, the Project Proposal and the Literature Review. The purpose of the project proposal was to undergo some initial research into the domain of Game Theory and determine an avenue worth exploring.

The purpose of the literature review was to lay foundations of the project (Dawson 2000), to justify the reason why the project was undertaken and to investigate and further examine the domain in greater depths.

The reason for the requirements document is to take the knowledge gained from undertaking the project proposal and especially the literature review and determine a set of functional statements that the program should be able to satisfy.

3.2. Requirements Analysis

The main aim for the program is to compute optimal strategies in matrix games and to obtain the payoff. The methods that the program can use to determine the optimal strategies and payoffs were described in the literature review. Therefore the literature review and the research done for the literature review will be the source of knowledge, which will be used to determine the functionality of the program.

3.3. Requirements Specification

From knowledge gained from the literature review and the understanding the domain and understanding what the end product should achieve, it is now
Determining Optimal Strategies in Matrix Games

possible to determine functional and non-functional requirements for the program.

This section will state what the requirements of the program are, in the sections that they are relevant in. The requirements will be split up into six sections;

- 2*2 Matrix Games
- 2*N and M*2 Matrix Games
- Graphical Method
- Simplex Method
- Non-Functional Requirements
- Hardware Requirements

3.3.1 2*2 Matrix Games

3.3.1.1. Matrix generated by the program must be of correct size.
3.3.1.2. Saddle points must be identifiable by the program.
3.3.1.3. The application must provide the means, for elements to be randomly generated inside the matrix.
3.3.1.4. The application must provide the means, for elements to be entered inside the matrix, by users.
3.3.1.5. Elements in the matrix table must be valid i.e. not characters, floats and string.
3.3.1.6. Negative values are acceptable for the elements within a matrix.
3.3.1.7. With valid data the program must be able to work out from a 2*2 matrix;
   3.3.1.7.1. The determinant of the matrix
   3.3.1.7.2. Adjoint of the matrix
   3.3.1.7.3. J vector and the transpose of J
3.3.1.8. Using the determinant, J vector, transpose of J vector and original matrix values, the program must be able to work out, the probabilities for the players involved in the game and the payoff of the game.
3.3.1.9. The probabilities for both players and the payoff, must be determined via the following formulas;
   3.3.1.9.1. Player I = JA*/JA*J
   3.3.1.9.2. Player II = A*Jᵀ / JA*Jᵀ
   3.3.1.9.3. Pay Off = |A|/ JA*Jᵀ

Where
   J = Vector (1, 1)
   A* = Adjoint of matrix
   Jᵀ = Transpose of J
   |A| = Determinant of matrix

3.3.1.10. The program must indicate whether the probabilities are valid or invalid by performing the following;
Determining Optimal Strategies in Matrix Games

3.3.1.10.1 Checking that the sum of the probabilities are equal to 1
3.3.1.10.2 The probabilities are positive and range between 0 and 1

3.3.2 2*N and M*2 Matrix Games

3.3.2.1 Program must allow users to enter in the length or width of their choice or randomly generate the length or width of the matrices.
3.3.2.2 That value entered in by the user or by the program must be valid. Any invalid data such as floats, characters and strings must be rejected by the program and the program must ask for valid data.
3.3.2.3 Removing of dominant strategies must be possible on 2*N and M*2 matrices.
3.3.2.4 Once the matrices are in 2*2 size, the program must be able to use the formulas in requirement 3.3.1.9, to determine the optimal strategies and payoff for the players.

3.3.3 Graphical Method

3.3.3.1 The matrix must be represented in a graph.
3.3.3.2 The application must be able to determine optimal strategies and payoffs for maximising gain and minimising loss, for both of the players.

3.3.4 Simplex Method

3.3.4.1. The program must provide the means of allowing the user to enter in the number of constraints or randomly generate the number of constraints.
3.3.4.2. The program must either randomly generate the pay off function or allow the user to enter in a pay off function.
3.3.4.3. Values randomly generated by the program or entered in by the user must be valid. String, characters, floats must not be accepted.
3.3.4.4. The payoff and constraints must be in the form of inequalities.
3.3.4.5. From the inequalities, the program must be able to determine a matrix of suitable size.
3.3.4.6. In matrix form the program must be able to determine the starting point, in order to perform the simplex method.
3.3.4.7. The starting point will be determined by the program selecting pivot points.
3.3.4.8. The program must select pivot points.
3.3.4.9. The program must be able to perform the simplex method.
3.3.4.10. After completing the simplex method, the program should read off the probabilities and the pay off function.
3.3.4.11. The program must be able to determine whether the probabilities are valid or invalid.
Determining Optimal Strategies in Matrix Games

3.3.5 Non-Functional Requirements

3.3.5.1. The program must be easy to use for any user.
3.3.5.2. The speed of the program must be suitable.
3.3.5.3. The program must be thoroughly tested to limit the number of errors/bugs possible.
3.3.5.4. The program must not crash while performing any of its main functionalities.
3.3.5.5. The program should be portable on any platform.
3.3.5.6. The project must be completed and handed in by 3rd May 2007, possibly a few weeks before.
3.3.5.7. Good programming practices (documenting and well structured flow of code) must be used in development of the program.
3.3.5.8. The program must be well written so that in the future the program can be added to and is easily maintainable.

3.3.6 Hardware Requirements

No specific piece of computing hardware will be required. The only thing that will be required is a standard computer with a reasonable amount of processing power, hard drive space and RAM and a programming application to write the program. This can be easily available either at the university library, in the university lab rooms or at my term time address. As deadlines start to get nearer, it will be difficult to gain access to the university library and lab computers, as other students will be needing access as well. If this is the case then there will be no concern for worry as the laptop located at my term time address, should be appropriate.

External devices that will be required will be a printer, USB key stick and a mouse.

3.4. Requirements Validation

According to Sommerville (2001, p137), “requirements validation is concerned with showing the requirements actually define the system that the customer wants”. This is extremely important as errors in the requirements stages could be costly mainly in terms of time, in the latter stages of the dissertation.

To validate the requirements the validation technique that will be used is requirements reviews (Sommerville 2001), in which the project supervisor will systematically analyse the requirements to see if they are;

- Validity - To see if all of the functionality of the program has been thought of.
- Consistent - There are no requirements that contradict each other.
- Realism - With the technology available, the requirements should be checked to ensure that they can be implemented.
- Verifiable - To provide evidence that the requirement can checked to see if it meets the need of the user.
3.5. Requirements Verification

Requirements verification is extremely important as it checks to see if the program meets the requirements. In order to prove that the program meets requirements, this document will be used when developing the testing section, in which the test cases will test program, to see if it does meet the requirements.
Chapter 4

Design

4.1. Introduction

The work undertaken before this document was, the Requirements Section. The purpose of the requirements section was to formally underline the objectives that the end product (the system), should meet. The requirements were then checked over by a stakeholder (Prof. N. Vorobjov), to determine whether the requirements were adequate to pursue.

The reason for this design document is to detail how the proposed system, will meet the requirements determined during the analysis stage of the development (Laudon, Laudon 2006).

From the discussions held between the stakeholder and the developer, the requirements that will be designed and implemented are requirements from, Section 3.3.1 - 2*2 Matrix Games, Section 3.3.2 - 2*N and M*2 Matrix Games and Section 3.3.3 - Graphical Method.

Not all of the requirements will be met, as not all of them will be implemented due to the time frame available. With the Graphical Method requirements, no graphical representation of the game will be produced due to the time frame and the complexity of the requirement.

4.2. Data Types and Structure

The data types and the structure that will mainly be used throughout the development is a two-dimensional array, since a matrix is a two-dimensional array. Another data type that will be used is a one-dimension array. The data types that will be used will be primitive types such as int, double, Boolean and object types such as String.
Determining Optimal Strategies in Matrix Games

4.2.1 Data Types

The data type String will be used to provide feedback and allow the user of the program, to input data into the program that can be manipulated by the program. The data type int (Integer) will be used to store elements inside the matrix. The data type double will be used to convert the integer elements inside the matrix to double, when determining the probabilities of the players and the payoff. Probabilities are expressed as fractions so the data type double will be useful in providing the precision required. The data type Boolean will be used for the termination of loops and exiting if statements.

4.2.2 Data Structures

As stated earlier the data types that will be used are a one and two dimensional array. The two dimension array will be used to create and store elements of a matrix, as pointed out earlier a matrix is basically a two by two array. The one dimensional array will be used to store multiple data. Since Java can not return two types of data using the array, the program will be able to return multiple data, which will be used in determining the probabilities of the players and the payoff.

4.3. Program Design Specification

4.3.1 Structure of System

The game theory application that must be implemented in such a way that it will accept two forms of, two person zero-sum matrix games. For each game, the application must know how to behave.

For a 2*2 zero-sum matrix game, the program must check if a saddle point exists to see if the matrix can be solved. Saddle points are later discussed in Section 4.3.2.1. If there are no saddle points in the matrix, then the program must use formulas to determine optimal strategies. This is discussed in Section 4.3.2.2.

For a 2*N zero-sum matrix game, the program will compute all the intersections that are made from the columns of the matrix. The program will then determine the payoff function and find the highest intersecting vertex on that payoff function and determine the optimal strategies and payoff for the players. This is discussed in Section 4.3.3.

From reading the above section it can be seen that two classes must be implemented, one for solving 2*2 matrix games and one for solving 2*N matrix games.
Determining Optimal Strategies in Matrix Games

4.3.2 Class 2*2 Matrix Games

4.3.2.1 Saddle Point

Mentioned previously in Section 2.4.1, a saddle point is an element \( a_{ij} \) in the matrix, where it is both the maximum entry in its column and the minimum entry in its row. The saddle point corresponds to the optimal strategies for Players I and II and the element that the saddle point corresponds to, is the payoff.

\[
\begin{pmatrix}
1 & 0 \\
-1 & 2
\end{pmatrix}
\quad
\begin{pmatrix}
5 & 6 \\
1 & 6
\end{pmatrix}
\quad
\begin{pmatrix}
-2 & -4 \\
3 & 0
\end{pmatrix}
\quad
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]

(a) (b) (c) (d)

Figure 4.3.2.1 Examples of games with and without saddle points

Figure 4.3.2.1(a) does not contain any saddle points. The minimum value in the first row is 0, where the maximum value in that column is 2. The minimum value in the second row is -1, where the maximum value in that column is 1. Figure 4.3.2.1(d) is another game that does not have any saddle points, as that matrix is the identity matrix.

Figure 4.3.2.1(b) does contain a saddle point. The element \( a_{11} \) is the saddle point, as it is the minimum value in the row and the maximum value in the column. Similarly in figure 4.3.2.1(c), element \( a_{22} \) is the saddle point, as the element is the maximum value in the column and the minimum value in its row. If a saddle point is found, the mixed strategy is 1 for that element and 0 for everything else.

If a saddle point does not exist, formulas are used to determine the players’ optimal strategies and payoffs.

4.3.2.2 Two by Two Matrix Games

Two by two matrix games are in the form of;

\[
\begin{pmatrix}
\bar{a}_{11} & \bar{a}_{12} \\
\bar{a}_{21} & \bar{a}_{22}
\end{pmatrix}
\]

Figure 4.3.2.2a Two by Two Matrix Game

(Owen 1982). If saddle points do not exist within the game, then optimal strategies for the two players and the payoff must be determined. The optimal strategies are in the form of \((x, 1-x)\) for player one where \(0 \leq x \leq 1\) and \((y, 1-y)\) for player two where \(0 \leq y \leq 1\). The reason for this as explained in the Literature Review, is that the optimal strategies are probabilities and the common characteristics of probabilities are that the sum of all probabilities must be equal to one and all probabilities must be positive.
Determining Optimal Strategies in Matrix Games

As mentioned in section 4.3.2.1, if no saddle points exist in a 2*2 matrix game, then formulas are used to determine unique optimal strategies for both players and the corresponding payoff is calculated.

The following is taken from Owen (1982). There is a 2*2 matrix game denoted matrix A, which does not have any saddle points. The optimal strategies and the payoff value will be determined via the following formulas;

\[
\begin{align*}
    x &= JA^*/JA^*J^T \quad (4.3.2.2.1) \\
    y &= A^*J^T / JA^*J^T \quad (4.3.2.2.2) \\
    v &= |A| / JA^*J^T \quad (4.3.2.2.3)
\end{align*}
\]

Where \(|A|\) is the **determinant** of A, \(A^*\) is the **adjoint** of A and J is the vector \((1,1)\) and \(J^T\) is the **transpose** of vector J. \(X\) corresponds to Player I, \(Y\) corresponds to Player II and \(V\) is the payoff.

From looking at the formulas it can clearly be seen that the value for \(JA^*J^T\) must never equal zero as it would result in dividing by zero. For the optimal strategies, \(x\) and \(y\), they have to be subjected for validity checks.

4.3.2.3 Specification

The purpose of the 2*2 matrix program is to take a two dimensional array, a matrix and determine the strategies for the two players and the payoffs for those players. To do this the following methods will be implemented;

- **RandomMatrix** – The purpose of this method is to create a matrix, in which the elements inside the matrix are randomly generated.
- **userMatrix** – The purpose of this method is to create a matrix, in which the elements inside the matrix are entered in by the user. This method must be aware that the user could enter invalid data and must take the necessary precautions.
- **saddlePoint** – The purpose of this method is to take a given matrix and check if a saddle point exists. If a saddle point exists then that element is the optimal strategy and the players are notified what the strategies are and the payoff.
- **printMatrix** – The purpose of this method is to print off the matrices that are generated.
- **determinant\(^1\)** – The purpose of this method is to calculate the determinant value, which is used to calculate the players’ payoff.

\(^1\) See Appendix B - Linear Algebra for explanation of determinant
Determining Optimal Strategies in Matrix Games

- **adjoint**$^2$ – The purpose of this method is to take a given matrix and return the adjoint of the matrix, which in turn will be used to calculate the players’ strategies and the payoff.

- **vectorTimesAdjoint**$^3$ – The purpose of this method is to take the vector (1, 1) and multiply it against the adjoint matrix. The returning results in be the form (x,y). That result will be stored in a one dimension array, which will be used to compute the players’ optimal strategies and the payoffs.

- **vectorTimesTranspose**$^4$ - The purpose of this method is to produce the transpose of vector (1, 1). The returning results will be in the form (x,y). That result will be stored in a one dimension array, which will be used to compute the players’ optimal strategies and payoffs.

- **adjointTimesVectorTimesTranspose**$^5$ - The purpose of this method is to take the adjoint matrix and multiply it against the product of the vector of the given matrix against the transpose vector, which will be deduced from the previous function. The returning results will be in the form (x,y). That result will be stored in a one dimension array, which will be used to compute the players’ optimal strategies and payoffs.

- **vectorTimesAdjointTimesVectorTimesTranspose**$^6$ - The purpose of this method is to take the results from the previous functions and calculate the denominator, which will be in the form of x and will be used to determine the players’ optimal strategies and payoffs.

- **Prob_Payoff** - The purpose of this method is to figure out the optimal strategies for the players and the corresponding payoffs, using the results computed from the earlier functions.

- **check** - The purpose of this method is to see if the two optimal strategies, which were calculated in the Prob_Payoff function, are valid.

---

$^2$ See Appendix B - Linear Algebra for explanation of adjoint

$^3$ See Appendix B - Linear Algebra for explanation of multiplication of elements and matrices

$^4$ See Appendix B - Linear Algebra for explanation of multiplication of elements and matrices

$^5$ See Appendix B - Linear Algebra for explanation of multiplication of vectors and matrices

$^6$ See Appendix B - Linear Algebra for explanation of multiplication of two vectors
Determining Optimal Strategies in Matrix Games

To create the matrix it will have to be initialised in the constructor which will also initialise the global Boolean value and also the mechanism to allow for random number generation.

After the methods have been written, the main method must be written. It is this method that will be executed and so must execute the previous functions.

4.3.2.4 Pseudo-code

This section will show the pseudo-code for the specification mentioned in Section 4.3.2.3.

Java import statements
Beginning of class
{
    Variables random number and integer 2D array
    Six integer variables and a Boolean value

    Constructor of class()
    {
        Initialisation of variables
    }

    public int [][] RandomMatrix()
    {
        Four global integer variables assigned random numbers from 0 - 100
        Four cells within the matrix assigned to the four global integer variables
        Printing of the matrix
        Returning of the matrix;
    }

    public int [][] userMatrix(Four integer variables)
    {
        Four cells within the matrix are assigned to the four formal parameters passed by the user
        Printing of the matrix
        Returning of the matrix;
    }

    public void printMatrix(Integer 2D array)
    {
        For loop to allow the printing of rows
            For loop to allow the printing of columns
                Printing off the cells within the matrix, leaving a tab in between the cells
            End of column loop
        Printing a new line
        End of row loop
    }
Determining Optimal Strategies in Matrix Games

```java
}  

public void check(double x, y)  
{
    if (x and y < 0 and (x+y) <> 1.0)
        Print probabilities (x,y) are not valid.
    else
        Print probabilities (x,y) are valid.
}

public int determinant(Integer 2D Array)
{
    Three local integer variables  
    First variable = the product of the first and fourth elements in the matrix.  
    Second variable = the product of the second and third elements in the matrix.  
    Third variable = first variable minus the second variable  
    Print off the third variable  
    Returning the third variable
}

public void saddlePoint(Integer 2D Array)
{
    Four integer variables  
    Local Boolean variable assigned false value  
    Four cells within the matrix assigned to the four local integer variables

    if (local Boolean is false and first element in matrix is greater than third element and less than second element)
        Print off first element indicating that it is the saddle point.  
        Local and global Boolean parameters are set to true

    if (local Boolean is false and third element in matrix is greater than first element and less than fourth element)
        Print off third element indicating that it is the saddle point.  
        Local and global Boolean parameters are set to true

    if (local Boolean is false and second element in matrix is greater than fourth element and less than first element)
        Print off second element indicating that it is the saddle point.  
        Local and global Boolean parameters are set to true

    if (local Boolean is false and fourth element in matrix is greater than second element and less than third element)
        Print off fourth element indicating that it is the saddle point.  
        Local and global Boolean parameters are set to true
```
Determining Optimal Strategies in Matrix Games

```java
public int[] vectorTimesAdjoint(Integer 2D Array) {
    Three integer variables
    Local 2D initialised Array
    Integer 1D Array of size 2 with both elements equalling 1
    First variable equals the first element in the 1D Array
    Local 2D Array is assigned the return array of the adjoint function, with
    the formal 2D Array as its parameter
    Second variable = sum of first variable multiplied by the first element in
    the local 2D Array and the first variable multiplied by the second element in
    the local 2D Array.
    Third variable = sum of first variable multiplied by the second element in
    the local 2D Array and the first variable multiplied by the fourth element
    in the local 2D Array.
    Printing off the second and third variables
    Creating a one dimensional array and storing the second and third
    variables in that array
    Returning the one dimensional array
}
```
Determining Optimal Strategies in Matrix Games

```java
public int[] adjointTimesVectorTimesTranspose(Integer 2D Array, Two Integer arrays)
{
    Two integer variables
    Integer two dimensional Array
    Initialisation of Two dimensional Array
    Local 2D Array is assigned the return array of the adjoint function, with
    the formal 2D Array as its parameter
    First integer variable = the sum of the first element in the first 1D array
    passed multiplied by the first element in the local 2D Array and the first
    element in the first 1D array passed multiplied by the second element in
    the local 2D Array.
    Second integer variable = the sum of the second element in the second
    1D array passed multiplied by the third element in the local 2D Array and
    the second element in the first 1D array passed multiplied by the fourth
    element in the local 2D Array.
    Printing off the first and second integer variables
    Creating a one dimensional array and storing the first and second
    variables in that array
    Returning the one dimensional array
}

public int[] vectorTimesAdjointTimesVectorTimesTranspose(Four Integer 1D Arrays)
{
    Five integer variables
    Assigning the first four local variables to specific positions in the arrays
    that were passed
    Fifth variable = sum of products between the first, third and second,
    fourth local integer variables
    Printing off the fifth variable
    Creating a one dimensional array and storing fifth variable in that array
    Returning the one dimensional array
}

public void Prob_Payoff(Five Integer 1D Arrays, integer number)
{
    Eleven double variables
    Seventh variable = double casting of the formal integer parameter passed
    Sixth variable = double casting of the first element of the fifth formal
    array passed
    if (sixth variable equals 0){
        Print no probabilities can be deduced
    } else {
        First variable = double casting of the first element of the first
        formal array passed
    }
}
```
Determining Optimal Strategies in Matrix Games

Second variable = double casting of the second element of the second formal array passed
Third variable = double casting of the first element of the third formal array passed
Fourth variable = double casting of the second element of the fourth formal array passed
Eighth variable = first variable divided by the sixth variable
Ninth variable = second variable divided by the sixth variable
Print probability for player one is (eighth variable, ninth variable)
Checking that the probability for player one is valid
Tenth variable = third variable divided by sixth variable
Eleventh variable = fourth variable divided by sixth variable
Print probability for player two is (tenth variable, eleventh variable)
Checking that the probability for player two is valid
Fifth variable = seventh variable divided by sixth variable
Print pay off is, (fifth variable)

public static void main(String[] args)
{
    Creation of instance of the class, Two By Two Matrix
    Creation of instance of scanner import
    Integer two dimensional array
    Four integer one dimensional arrays
    Six integer variables
    String variable
    Boolean variable
    Initialisation of Boolean and arrays variables
    Print enter one to create a random matrix or two to enter in the elements into a matrix

    While looping continuing until the Boolean value is true
    {
        Error handling
        {
            Integer variable is assigned the value read in from the users’ input
            If (integer variable is invalid)
            {
                Print number invalid, valid numbers are either 1 or 2
            }
            Switch statement according to the integer variable
            First case:
            While loop terminated
            Random generation of 2D array, which is assigned to the 2D Array variable

}
Determining Optimal Strategies in Matrix Games

End of first case

Second case:
Print enter in four valid elements
While loop terminated
Four elements entered in by the user are assigned to four
integer variables and are entered into a matrix
End of second case

} Catch statement
{
  Print invalid data types entered
}

If global Boolean value is false, run the saddle point function, passing as
its parameter the 2D Array
If global Boolean value is still false
  Integer variable is assigned the worked out determinant value
  Adjoint of 2D Array is determined
  Running the functions to determine the probabilities and payoff for
  the players

} End of class

4.3.3 Class 2*N Matrix Games

4.3.3.1 Two by N Matrix Games

Two by N zero-sum matrix games are in the form of;

\[
\begin{pmatrix}
  a_{11} & a_{21} & a_{31} & \ldots & a_{1n} \\
  a_{21} & a_{22} & a_{23} & \ldots & a_{2n}
\end{pmatrix}
\]

Figure 4.3.3.1a Two by N Matrix Game

Solving these types of matrix games is relatively simple. The problem for player
one is that they wish to maximise;

\[
v(x) = \min \{a_{ij}x_1 + a_{3j}x_2\} \quad (4.3.3.1.1)
\]

From the characteristics of probabilities, \(x_1 = 1 - x_2\). Therefore

\[
v(x) = \min \{(a_{2j} - a_{1j})x_2 + a_{1j}\} \quad (4.3.3.1.2)
\]

It can be seen that \(v(x)\) is the minimum of the \(n\) linear functions (within the
matrix) of the single variable \(x_1\). It can be seen that these linear functions,
Determining Optimal Strategies in Matrix Games

(calculated with the linear equation: \((a_{2j} - a_{1j}) x + a_{1j}\)) can be plotted on a graph and if plotted the linear functions will pass through the points \((0, a_{1j})\) and \((1, a_{2j})\), as the example from Owen (1982) shows;

Take the following matrix,

\[
\begin{pmatrix}
2 & 3 & 1 & 5 \\
4 & 1 & 6 & 0
\end{pmatrix}
\]

Figure 4.3.3.1b Example from Owen (1982)

The above matrix and be represented by the graph as shown below;

Figure 4.3.3.1c Graphical Representation of 2*N Matrix shown in Figure 4.3.3.1b

The red line represents the payoff function of \(v(x)\).

The optimal strategy for Player I is \((5/7, 2/7)\) and for Player II the optimal strategy is \((2/7, 3/7)\) with the payoff, \(v\) being \(17/7\).

This can be determined by taking the second and third columns and disregarding the unwanted columns. The reason why the second and third columns were chosen was because the highest vertex of the payoff function, is determined by those columns and also unlike the other vertices, vertex X does not have any other vertices below it that do not lay on different lines. With the remaining columns a two by two matrix is left. Using the formulas below optimal strategies and the payoff can be determined.

\[
\text{Player I} = \frac{d - c}{a + d - b - c} \quad (4.3.3.1.4)
\]

\[
\text{Player II} = \frac{d - b}{a + d - b - c} \quad (4.3.3.1.5)
\]

\[
\text{Payoff} = \frac{ad - bc}{a + d - b - c} \quad (4.3.3.1.6)
\]
Determining Optimal Strategies in Matrix Games

The resulting probabilities for the players will be the $z_1$ value of optimal strategies ($z_1$, $z_2$). $z_2$ will be the one minus the probability of $z_1$. Therefore the optimal strategies for the players will be ($z_1$, 1-$z_1$).

The reason why formulas (4.3.2.2.1), (4.3.2.2.2) and (4.3.2.2.3) were not used was because these formulas were another method of working out optimal strategies and payoffs for a 2*2 matrix game. The discovery of formulas (4.3.3.1.4), (4.3.3.1.5) and (4.3.3.1.6) was after the first draft of the literature review, the requirements and Section 4.3.2.2. This discovery which is mentioned towards the end of Section 2.4.3 is an example of the literature review evolving overtime as the project is moving forward (Dawson 2000).

The reason why formulas (4.3.2.2.1), (4.3.2.2.2) and (4.3.2.2.3) were still used for the Two by Two Matrix Class, was because the algorithms in Section 4.3.2.4 were already implemented and tested, when the formulas (4.3.3.1.4), (4.3.3.1.5) and (4.3.3.1.6) were discovered.

4.3.3.2 Special Cases of 2 by N Matrix Games

Figure 4.3.3.1c shows a typical example of how 2*N matrix games could look like graphically. There are also two other ways that 2*N matrix games could be represented graphically. These are shown in Figures 4.3.3.2a and 4.3.3.2b.
Figure 4.3.3.2a, is the second special case and shows only one peak vertex point in which all of the lines intersect at that point. The solution for this is similar to the solution for Figure 4.3.3.1c as two columns will be placed in a 2*2 matrices and the formulas (4.3.3.1.4), (4.3.3.1.5) and (4.3.3.1.6) will be used to determine the optimal strategies for the players and the payoff for those players. With this case, the players’ strategies and payoff should be the same, regardless of which matrix is chosen.

Figure 4.3.3.2b, is the third special case and is different to the previous figures, as there are two peak vertices, X and Y. The solutions for this case of 2*N matrix game will be, between X and Y with all payoffs being V. This means that probabilities between vertices X and Y will be optimal strategies for the players and all of those optimal strategies will have the same payoff.

4.3.3.3 Specification

The purpose of the 2*N matrix program is to create a matrix of n columns specified by the user. Then the program will work out all of the intersections of the lines and determine the players’ strategies and payoffs. To do this the following methods will be implemented;

- **userMatrix** – The purpose of this method is to create a matrix, in which the elements inside the matrix are entered in by the user. This method must be aware that the user could enter invalid data and must take the necessary precautions.

- **printMatrix** – The purpose of this method is to print off the matrices that are generated.

- **check** - The purpose of this method is to see if the two optimal strategies that were calculated were valid.

The above functions perform the same functionality as their corresponding functions in Section 4.3.2.3. However the above functions have to deal with n columns when the functions in Section 4.3.2.3 only have to deal with 2 columns. The implementation of generating random matrices and finding saddle points will not be implemented for the 2*N matrix program because the implementation of those functions is trivial as they were implemented in the 2*2 matrix game. Also the implementation of those functions does not add any uniqueness to the project.

- **printEquations** – The purpose of this function is to calculate the equations of the lines, from the columns given inside the matrix. The equations of the lines are deduced from the following linear equation; \( y = (a_{2j} - a_{1j}) x + a_{1j} \).
Determining Optimal Strategies in Matrix Games

- **payoff** – The purpose of this function is to implement the formula (4.3.3.1.6) and determine the payoff of a matrix.
- **Player1** – The purpose of this function is to implement the formula (4.3.3.1.5) and determine the optimal strategy for Player I.
- **Player2** – The purpose of this function is to implement the formula (4.3.3.1.4) and determine the optimal strategy for Player II.
- **smallMatrix** – The purpose of this function is to take a given 2*N matrix and produce all of the combinations of 2*2 matrices, which will be used to figure out the intersections of all the lines.

To create the 2*N matrix, it will have to be initialised in the constructor, which will determine the number of columns.

After the methods have been written the **main** method then must be written. It is this method that will be executed and so must execute the previous functions.

### 4.3.3.4 Pseudo-code

This section will show the pseudo-code for the specification mentioned in Section 4.3.3.3.

Java import statements

Beginning of class

```
{ Variables integer 2D array, two integers and a scanner variable

Constructor of class()
{
    Initialisation of variables and initialisation of matrix, with the
    number of columns determined by the user
}

public void printMatrix(Integer 2D array)
{
    For loop to allow the printing of rows
        For loop to allow the printing of columns
            Printing off the cells within the matrix, leaving a tab in
            between the cells
        End of column loop
    Printing a new line
    End of row loop

public int[][] userMatrix()
{
```
Determining Optimal Strategies in Matrix Games

Integer variable
Print statement telling the user to enter in valid elements
For loop to allow entering data in the number of rows
  For loop to allow entering data in the number of columns
    Try statement
      Assigning the position in the matrix, to the value
      that the user entered
    Catch statement
      Print statement, stating to the user that invalid data
      was entered
      Exiting the application
    Printing off the matrix
    Returning the matrix

public int[] printEquations()
{
  Three integer variables
  One dimensional integer array
  For loop against the number of columns
    First integer variable set to the element in the first row of the
    column
    Second integer variable set to the element in the second row of
    the column
    Third variable is the difference of the first two variables
    Print off the third variable and first variable to be the equation of
    the line.
    Store the values in an array
    Return the array
}

public Boolean check( Two double parameters)
{
  Double variable equalling to the sum of the two double actual parameters
  passed into the function
  If statement checking that both actual parameters are positive
    Returning false, if either parameter is negative
  Else checking if the variable is equal to 1
    Retuning false if variable is not equal to 1, true if variable is equal
    to 1
}

public double pay_off( Integer 2D Array)
{
  Six integer variables
  First four variables are assigned a position in the 2D Array
  Fifth variable is the sum of the difference between the product of the
  first, fourth and second, third elements
Determining Optimal Strategies in Matrix Games

Sixth variable is the sum of the first two variables, minus the second two variables
Double variable is the casting value of the fifth variable divided by the sixth variable
Returning the double variable

public double Player1(Integer 2D Array)
{
    Six integer variables and one double variable
    First four integer variables assigned a position in the 2D Array
    Fifth variable is the value of the difference between the fourth and third elements
    Sixth variable is the sum of the first two variables, minus the second two variables
    Double variable is the casting value of the fifth variable divided by the sixth variable
    Returning the double variable
}

public double Player2(Integer 2D Array)
{
    Six integer variables and one double variable
    First four integer variables assigned a position in the 2D Array
    Fifth variable is the value of the difference between the fourth and second elements
    Sixth variable is the sum of the first two variables, minus the second two variables
    Double variable is the casting value of the fifth variable divided by the sixth variable
    Returning the double variable
}

public double[] smallMatrix(Integer 2D Array)
{
    Integer variable 2D Array of 2 by 2 size
    One dimensional double array
    For loop for the length of column size minus one
        For loop for the variable equalling the variable from the first loop plus one, less than the number of columns
            Positions local 2 by 2 array are assigned positions from the passed 2D Array
            Printing off the 2 by 2 array
            Pass the 2*2 array into the functions that calculate the probabilities and payoff and assign those values into the array
        Return the array
    Return the array
}
public static void main(String[] args)
{
    Create an instance of the class;
    Integer 2 by N matrix variable
    Three one dimensional double arrays
    Boolean variable set to false;

    Call the function that allows the user to enter in elements into a matrix
    Assign an array to hold the probabilities for player one
    Assign an array to hold the probabilities for player two
    Assign an array to hold the probabilities for the payoffs

    For loop checking the array holding the payoffs
    If statement checking comparing the first element in the
    array against the other elements
        Boolean value set to true if elements are the same
        Else Boolean value set to false if elements are not the same

    For loop against the total number of columns
    If statement checking the Boolean value and probabilities
    of the two players
        If the values are true then printing off the
        probabilities and payoff

    While loop against the length of the array that is storing the payoff and
    the Boolean variable set to false
        For loop against the size of the array that is storing the payoff
            Variable set to a position in the payoff array
            If statement checking to see if any two payoffs match
                If payoffs match, Boolean value set to true else
                value remains false
            If statement checking the Boolean value and probabilities
            of the two players
                If the values are true then printing off the
                probabilities and payoff

    For loop finding out what intersections lay on the line
        If intersection lay on the line storing the intersection in array
        Taking the intersection that generates the lowest payoff

    For loop removing any duplicate values from the array
    For loop finding out from the remaining intersections lay on the two lines
    closest to the horizontal axis
        If intersection lay on the line storing the intersection in array
    Comparing the two intersections, choosing the intersection with the
    highest payoff
    Printing off the optimal strategies and payoff

}
Chapter 5

Implementation

5.1. Introduction

The work undertaken before this document was, the Implementation of the Two by Two Matrix Class and the Two by N Matrix Class. The purpose of the implementation stage of the project was to produce an application that satisfied the requirements and design specifications.

The purpose for this document is to discuss and justify the decisions made and also discuss the algorithms of the application.

5.2. Programming Languages

When determining what languages would be best suited for the development of the program, originally Maple (a general purpose, commercial mathematics software package) was the chosen application to write the program. The reason behind this choice was the fact that previous knowledge and usage of the application, the conclusion was made that Maple was a good application in dealing with matrices and manipulation of matrices.

When conveying the initial idea to other students, the advice that was given was that MATLAB (a high-level language application) would be a better application than using Maple, because MATLAB allows easy matrix manipulation, plotting of functions and data.

When discussing this with my stakeholder, the advice that was given to me was not to use MATLAB because most of the functions that would be needed were already pre-written as libraries in MATLAB. The use of MATLAB would prove to be a poor project. With that in mind, the only languages that were considered were JAVA and C. The only reason why those languages were considered was because of the experience associated with those languages and the fact that
Determining Optimal Strategies in Matrix Games

using a new language such as C++ would require the learning of such a language, which would eat into the time of developing the program.

The stakeholder stated that C or Java would suffice to implement the program. The decision was made that Java would be the intended language that the program would be implemented in because the personal experience of using Java was greater than the experience gained with using C. Also certain concepts were difficult to grasp when using C, such as the use of pointers.

5.3. Implementation

5.3.1 Two by Two Matrix Games

5.3.1.1 Discussion of Algorithm

The algorithm behind solving 2*2 matrix games was simple to devise. From initial research stated in Section 2.4.1, one method to solve 2*2 matrix games is to determine if a saddle point exists. A saddle point is an element in the matrix, which is the highest value in the column and lowest value in the row, that it is in.

The algorithm behind this was four if statements checking each individual element against the criteria for the saddle point. To solve 2*2 matrix games, where no saddle points exist, the formulas in Section 2.4.3 must be applied.

The algorithm behind the formulas was as more complex than the saddle point algorithm. It was decided upon that the best way to implement the formulas was to create functions for each of the numerators and denominators for the formulas. Where the numerators and denominators were determined by more that one value, it was decided upon to have a function that created that value, so that the algorithm was kept as simple as possible.

The functions for determining the formulas consisted of creating six functions:

- Determinant – This function calculated the numerator for formula (4.3.2.2.3), which is the formula to determine the payoff for the game.
- Vector Times Adjoint – This function calculated the numerator for formula (4.3.2.2.1), which is the formula to determine Player I’s optimal strategy.
- Vector Times Transpose – This function calculated part of the denominator for the three formulas.
- Adjoint Times Vector Times Transpose – This function takes the values from the Vector Times Transpose and multiplies them against the adjoint matrix. The resulting values are then used in calculating the numerator for formula (4.3.2.2.2), which is the formula to determine Player II’s optimal strategy.
- Vector Times Adjoint Times Vector Times Transpose – This function calculated the denominator for all of the formulas and was calculated by taking the results from the Vector Times Adjoint and Vector Times Transpose and finding the sum of products.
- Prob Payoff – This was the function that used the values determined from the above functions, to determine optimal strategies and payoffs for players in a 2*2 zero-sum matrix game. Within that function the validity checks on the probabilities were carried out.
Determining Optimal Strategies in Matrix Games

5.3.1.2 Discussion of Programming

Looking at the pseudo-code in Section 4.3.2.4 for the Two by Two Matrix Class, the only difference is that there is an extra function called wrapText, which as the name suggests, wraps text so that the displayed text is neat. The reason why this function was not mentioned in the Section 4.3.2.4 was because after running the completed code, the application did not display the text in a neat and concise manner. Therefore it was decided upon to create a function that wrapped text, so that the text displayed to the user was neat and concise.

This function was located on the website;

http://progcookbook.blogspot.com/2006/02/text-wrapping-function-for-java.html

The reason why the function was copied from the website, rather than being implemented was simple. The function was already available on the net, so as the expression goes, ‘why reinvent the wheel’. Also the time that would have been spent on implementing and testing the function, would mean that less time would be spent on the rest of the dissertation.

5.3.2 Two by N Matrix Games

5.3.2.1 Discussion of Algorithm

The algorithm for solving 2*N matrix games was more complex than the algorithm for solving 2*2 matrix games, as this algorithm would have to deal with unknown columns. The algorithm that was implemented also had to deal with the three special cases that were mentioned in Section 4.3.3.2.

When looking at the three cases, the decision was made to use modularisation for determining an algorithm for each individual case. The reason behind this decision was that modularisation allows a big problem to be broken down to smaller manageable problems (Barnes and Kolling 2003). Determining one algorithm for the three cases would have been a big problem!

Now that the special cases were sorted out, the solution was still not complete as the intersections were needed to solve the matrix game. An intersection is when two lines cross over, the point that they cross is known as the intersection. In order to obtain the intersections from the 2*N matrix, all 2*2 matrices had to be deduced. This is because two lines were needed for one intersection.

The function for determining all of the 2*2 matrices from the given 2*N matrix was;

- Small Matrix – This function took the 2*N matrix entered by the user and calculated the number of 2*2 matrices and found the intersections of the matrices that it produced.
Determining Optimal Strategies in Matrix Games

The intersections were calculated from the 2*2 matrices, via the payoffs (v) and the optimal strategy for Player I, at those intersections.

The functions that calculated the intersections from the 2*2 matrices were:

- Payoff – This function calculated the payoff value at the intersection, using the formula (4.3.3.1.4).
- Player I – This function calculated Player I’s optimal strategy at the intersection, using the formula (4.3.3.1.5).
- Player II – This function calculated Player II’s optimal strategy at the intersection, using the formula (4.3.3.1.6).

The actual intersection, which is in the form of (x, y), was formed from taking the payoff value and the difference between, 1 and Player I’s optimal strategy; i.e. ((1 – Player I), v).

Once all of the intersections had been calculated they were stored into an array, to be used later with the algorithms that dealt with the special cases of the 2*N matrix games.

The algorithm that dealt with case 2 (Figure 4.3.3.2a), in which all lines intersected at one point, thus resulting in one intersection, was simple to determine and implement.

Since all lines intersected at the same point, the payoff value (y in the intersection coordinates), for every intersection, had to be the same. This meant that the algorithm would check to see that every payoff value was the same. If the values were the same the algorithm would state this and give the corresponding optimal strategies for the players, as the diagram below shows.

![Array holding payoffs](x x x x x x)

![Array holding Player I’s probabilities](1 0 1 1 2 1 3 1 4 1 5)

![Array holding Player II’s probabilities](2 0 2 1 2 2 2 3 2 4 2 5)

Figure 5.3.2.1a Solving Case 2 of 2*N Matrix Games

Figure 5.3.2.1a shows three arrays. One to store payoffs, one to store Player I’s probabilities and one to store Player II’s probabilities. The arrays are all of size 6, with array positions 0 – 5.

Notice that all the payoffs are the same. Therefore the algorithm would choose the first payoff and from the array position of that payoff, chose the corresponding optimal strategies for the players, as the highlighted boxes in Figure 5.3.2.1a shows.
Determining Optimal Strategies in Matrix Games

The algorithm that dealt with case 3 (Figure 4.3.3.2b), in which there were two vertices that resulted in two different optimal strategies that generated the same payoffs, was simple to determine and implement.

Since only two of the intersections resulted in the same payoffs, the algorithm would search the array holding the payoffs and look for payoffs that were identical. When the payoffs were determined, the algorithm would state this and give the corresponding optimal strategies for the players, as the diagram below shows.

Figure 5.3.2.1b Solving Case 3 of 2*N Matrix Games

Figure 5.3.2.1b shows three arrays, similar to Figure 5.3.2.1a. Notice that there are two payoffs that are the same. Therefore the algorithm would choose those payoffs and from the array positions of the payoffs, choose their corresponding probabilities for two players, as the highlighted boxes in Figure 5.3.2.1b shows.

The final algorithm, that dealt with case 1 (Figure 4.3.3.1c), in which there was a high peak that resulted in one unique payoff and unique optimal strategies, was more complex to implement when compared to the previous discussed cases.

Looking at that unique point, it was seen that it was the lowest point on the line. Therefore the algorithm had to find for each line the lowest point. This was done by checking that every intersection lay on the line.

```java
double value = y - ((m*x)+c);
if (Math.abs(value) <= 0.00001)
{
    temp[o] = y;
    o++;
}
```

The above is a fragment of code that was implemented. The above determines whether the intersection (x, y) lay on the line. The code checked that if the intersection lay on the line that the variable value should equal 0. If the value was marginally equalled to zero the payoff would get stored in an array. The reason why the algorithm, checked to see if the value was marginally equalled to zero and not zero itself, was because when calculating the probabilities the
Determining Optimal Strategies in Matrix Games

computer would round off, which would lead to a precision error. The only way to counteract the precision was to have the value checked against 0.00001.

The intersections that did not lie on the line were discarded. With the remaining valid intersections, the intersection that gave the lowest payoff was kept and the rest of the intersections were discarded. This continued until the lowest points for each line were determined.

With the remaining intersections, a check was made so that no duplicate intersections were present in the array.

```java
int noDupsSize = store.length;
for (int i = 0; i < store.length - 1; i++) {
    if (store[i] == store[i+1]) {
        noDupsSize--;
    }
}
double[] noDups = new double[noDupsSize];
noDups[0] = store[0];
int noDupsCounter = 1;
for (int i = 0; i < store.length - 1; i++) {
    if (store[i] != store[i + 1]) {
        noDups[noDupsCounter++] = store[i + 1];
    }
}
```

The above is a fragment of code that was implemented. The array store was the name of the array in which all of the valid payoffs were stored. The array noDups was the name of the array in which all of the non-duplicated, valid payoffs were stored.

The first for loop determined the size of the noDups array, by checking to see if elements within the store array were identical. If the elements inside that array were identical the value of noDupSize (which was the length of the store array), was decremented. The second for loop was the loop that stored the non-duplicated payoffs, into the noDups array.

The elements that were left in the array went through a final check. From looking at the Figure 4.3.3.1c, two points were required. To obtain those two points the elements in the array had to be checked against the lines that were closest to the horizontal axis. The elements that were not on the line were discarded. With the remaining elements, the element that was the highest payoff was chosen, along with the optimal strategies of the players.

### 5.3.2.2 Discussion of Programming

Comparing the pseudo-code in Section 4.3.3.4 and the actual implementation of the Two by N Matrix Game, there were many differences. The differences were that nine extra functions that were not mentioned in Section 4.3.3.4.
Determining Optimal Strategies in Matrix Games

Out of the nine functions three functions were unique and the remaining six were duplicate methods. The three unique functions are as follows;
- Bubble Sort
- Lowest Lines
- Highest Peak

The function Bubble Sort was used within the algorithm that dealt with case 1 of 2*N matrix games. The purpose of this function was to take all of the intersections that were on a line and order them from lowest to highest, so that the lowest payoff value could be chosen. The idea originally to chose the lowest payoff, was to have an *if statement* and using a temporary variable that was assigned to the first element, check each value in the array. The reason why this method was not implemented was because this method of searching is called a linear search (Horstmann 2002 & Wikipedia 2007). This method is not an ideal solution for searching for the lowest element because if the lowest element was at the end of the array and the array size was huge, it could take a long time for the lowest element to be discovered. This would result in requirement 3.3.5.2, not being met.

The solution was instead of searching for the lowest element in an unsorted array, the array would be sorted using a sorting algorithm and then the first element would be chosen from the array.

The sorts that were available for implementation were;
- Selection Sort – The algorithm for this sort, “sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.” (Horstmann 2002 p704)
- Quick Sort – The algorithm for this sort chooses a pivot element within the array. The algorithm then looks at each element to the left and right of the pivot. If the element is greater or less than the pivot, it swaps the positions of the elements.
- Bubble Sort – The algorithm for this sort, works by repeatedly checking the array comparing two elements at a time and swapping them if they are in the wrong order.

The reason why quick sort was not implemented because the worst case scenario for the quick sort was that it would make $O(n^2)$ comparisons, again failing requirement 3.3.5.2.

Selection sort and bubble sort both make $O(n^2)$ comparisons and selection sort is more efficient than bubble sort. The reason why selection sort was not implemented was because that sort will depend upon the number of elements in the array. If the number of elements doubled, it would result in the number of comparisons increasing fourfold. This again would fail requirement 3.3.5.2.

That would also mean that with bubble sort the number of comparisons would increase fourfold, as bubble sort also makes $O(n^2)$ comparisons. However that is the worst case scenario. The best case scenario for bubble sort is that the array is already sorted, which would result in the array only being searched

---

7 Big O Notation
Determining Optimal Strategies in Matrix Games

once, thus the array having $\Theta(n)$ comparisons. Another reason behind implementing the bubble sort over the selection sort was that bubble sort was the easiest sort to implement.

The purpose of the function Highest Peak was to find, from the all the lines the lowest intersections that lay on those lines. The purpose of the function Lowest Lines was to determine from the 2*N matrix, entered by the user, the equation of the two lines closest to the horizontal axis. This function was used with the Highest Peak function within the algorithm that dealt with case 1 of 2*N matrix games.

The number of intersections that Highest Peak produced was one less the number of columns of the 2*N matrix. From looking at Figure 4.3.3.1c, only two intersections laid on the payoff function. To obtain those intersections, the intersections that were produced from the Highest Peak function were passed into the function Lowest Lines. From that function, it produced the two intersections that were on the payoff function. Then it was a simple case of the algorithm choosing the greater payoff and the strategies for the players.

The reason why those functions were not mentioned in the pseudo-code in the Design Specification was because those methods were going to be implemented with the algorithm that dealt case 1. The reason why this option was not executed was that further modularisation was used, to keep the coding of the 2*N Matrix Class from getting over complicated.

Using modularisation is a good method for keeping code from getting over complicated but implementing duplicate methods adds inconsistencies and can make the code complicated.

The six duplicate methods were:

- Gradient
- Intersection
- Number of Matrices
- Small Matrix Player 1
- Small Matrix Player 2
- Small Matrix Payoff

The functions Gradient, Intersection were duplicated from the function, Print Equations. The purpose of the functions Gradient and Intersection was to determine the gradient and intersection of the lines, that the 2*N matrix formed.

The functions Number of Matrices, Small Matrix Player 1, Small Matrix Player 2 and Small Matrix Payoff were duplicated from the function Small Matrix. The purpose of the function Number of Matrices was to determine the number 2*2 matrices that were produced from the 2*N matrix. The functions Small Matrix Player 1, Small Matrix Player 2 and Small Matrix Payoff produced the probabilities for both players and the payoff, for each 2*2 matrix. Those values were later used to form the coordinates of the intersecting lines.

Even though duplicated functions acts against requirements 3.3.5.7 and 3.3.5.8, the reason why duplicated functions were implemented was because of simplicity and reducing the complexity of programming. Without the duplicate functions, the functions Print Equations and Small Matrix would have to return
Determining Optimal Strategies in Matrix Games

multiple arrays and perform multiple operations, which would have complicated the programming of the 2*N Matrix Class, further along.
Chapter 6

Testing

6.1. Introduction

The work undertaken before this document was the implementation of the Two by Two and Two by N Matrix Classes. The importance of that work was to translate the specifications prepared during the requirement and design stages into program code (Laudon, Laudon 2006), which was Java.

The purpose for this testing document is to see whether the requirements that were identified at the beginning of the dissertation, have been satisfied.

6.2. Software Testing

Software testing is a technique used for system checking and analysis. It is used within verification and validation (Sommerville 2001).

Verification and validation (V & V) as Sommerville (2001) states, is the name given to the checking and analysis process that ensure that software conforms to its specification and meets the needs of the stakeholders who will be using the software.

Verification and validation is a process that must occur throughout the lifecycle, from requirement reviews to design reviews and code inspections, which have been apparent throughout this project, as the Requirements and Design sections were checked over by stakeholders, in which an agreement was made for which requirements were to be implemented, through to vigorous code inspections.

Even though verification and validation are apart of a vital process they are in fact separate entities as expressed by Sommerville (2001). Validation involves checking to see whether the system will meet the expectations of the stakeholders and verification involves checking that the software meets the requirements.
Determining Optimal Strategies in Matrix Games

The goal of V & V process is to raise some confidence in the application and the application’s abilities. The confidence of a system does not necessarily mean completely free of errors. What the term confidence means in this context is that the system must be good enough for its intended use. The intended use and purpose for the application is to determine optimal strategies in matrix games.

As stated software testing is apart of V & V. Software testing involves running an executable version of the application and testing the application with test data or test scripts/cases and examining the behaviour and outputs of the application, to check that it is functioning as required.

Sommerville (2001) states that there are two distinct types of testing;

- Defect testing
- Statistical testing.

Defect testing is about finding inconsistencies between the application and specification. Defect testing is concerned with finding program faults or defects. Statistical testing is concerned with testing the performance and reliability of the application, under test conditions. It is also concerned with the time taken for the application to be executed.

With the purposes of testing the game theory application only defect testing will be employed. The reason why statistical testing will not be executed is because the application is not greatly concerned, with the time taken for the application to be executed, to a certain extent.

Appendix D will contain the test scripts that were used to test the application. Those test scripts, will detail the steps that were required to create the test and note the expected and actual behaviour of the system.

The types of testing that will be performed are; unit testing and system testing combined with black-box testing. The reason why black-box testing is considered is so that the application can be tested to see if the requirements that were devised in Section 3.3 were met or not.

The reason why unit testing is considered is so that the each part of the application can be thoroughly tested and also be subjected to stress testing to see what the behaviour of the program is when invalid data is entered. To test the application using unit testing the main method will have to be slightly modified so that the remaining code applies to the test that is carried out.

Finally system testing will test the application to see if all of the functions are all working as specified and that the application as a whole performs the job that it was designed to do. To test the application using system testing, all the code in the main method will be executed.

6.3. Test Scripts

The test scripts will document the descriptive nature of the test, the pre-conditions necessary before the tests are performed. Next, will include, the steps required for the test and the expected results, followed by the actual results, resulting in a pass or fail, with the additional space for the tester to make additional comments. Appendix E will hold the test data used for the testing and Appendix F will hold the screen shots of the testing.
Determining Optimal Strategies in Matrix Games

6.3.1 Two by Two Matrix Class

Looking at Sections; 3.3.1, 4.3.2 and 5.3.1 for the Two by Two Matrix Class, the following unit tests were devised;

- Creation a random matrix and printing the random matrix
- Creation of user specified matrix and printing the user specified matrix
- Checking for saddle points
- Calculating the determinant
- Performing the adjoint operation on a matrix and printing off the adjoint matrix
- Calculating the numerator for the equation (4.3.2.2.1)
- Calculating the denominator for the equations (4.3.2.2.1), (4.3.2.2.2) and (4.3.2.2.3)
- Calculating the numerator for the equation (4.3.2.2.2)
- Calculating the numerator for the equation (4.3.2.2.3)
- Calculating the probability for players I and II
- Performing probability validity check
- Calculating the payoff

The unit test cases and system test case for the Two by Two Matrix Class can be found in Section D1.

6.3.2 Two by N Matrix Class

Looking at Sections; 3.3.2, 4.3.3 and 5.3.2 for the Two by N Matrix Class, the following unit tests were devised;

- Creation of the 2*N matrix
- Entering data into the matrix and printing of the matrix
- Working out the linear equations of the lines
- Working out the payoffs of all the lines
- Working out the probabilities of all the lines
- Performing probability validity check
- Determining the case of the graphical nature of the matrix

The unit test cases and system test case for the Two by N Matrix Class can be found in Section D2.
Chapter 7

Conclusions

7.1. Introduction

The work undertaken before this document was the testing of the Two by Two and Two by N Matrix Classes. The importance of testing was to see if the application performs to it expectations and meet the requirements specified in Sections 3.3.1, 3.3.2 and 3.3.3 and design specifications specified in, Sections 4.3.2 and 4.3.3.

The purpose for the conclusion section is to discuss the results of the testing and present a concluding remark about the dissertation.

7.2. Analysis of Testing

The test scripts in Appendix D hold all of the results for the testing. Within those test scripts is an indication of whether the test passed/failed and comments, that were added by the tester. Appendix F shows the screen shots from the testing. The test data for $2^2$ and $2^N$ matrices are in Appendix E. The test data contains worked out solutions that were used to compare the output that the application produced.

Also present in Appendix E, is the graphical representation of the matrices that were used for the testing.
Determining Optimal Strategies in Matrix Games

7.2.1 Testing of 2*2 Matrix Class

Looking at the overall results from the testing of 2*2 Matrix Class, it can be seen that the program does compute optimal strategies and payoffs for matrices that it is given. The program also does cope with invalid scenarios such as;
- The user entering invalid integers when the application asks the user to enter in 1 to generate a random matrix or enter in 2, to allow the user to enter elements themselves into the matrix.
- The user entering non-integer values when the program requires integer values to be entered.

The only criticism with the error handling that was noted by the tester was that when the user enters in an non-integer value, the program terminates, with a message saying that invalid data types was entered. This is not an ideal solution as, software applications used all over the world have methods to stop users from entering the wrong data type;
  - Disabling the keys on the keyboard, that will prompt the user to enter in invalid data. I.E. To only allow integers to be entered for the matrix, the only the digital keys on the keyboard and the Enter key, would remain enabled, while the rest of the keys would be disabled.
  - When the user enters in the wrong data type, the application give the user a message and the program loops until the user enters in the correct data type.

7.2.2 Testing of 2*N Matrix Class

Looking at the overall results from the testing of 2*N Matrix Class, it can be seen that the program does have some bugs present and needs refining.

The algorithm for case 2 (Figure 4.3.3.2a), worked perfectly fine, as the program correctly worked out the players’ optimal strategies and payoffs, for the matrices specified.

Looking at case 1 (Figure 4.3.3.1c) there is a flaw with the algorithm. The 2*5 Matrix (Figure E.2.1.3a) caused the program to crash. However when entering a different 2*5 matrix that generated case 1 (Figure G.1a), the program does not crash and calculates correctly the optimal strategy and payoff! Also for the same case, matrices of 2*3 and 2*4 do not crash.

Comparing all of the matrices, the difference was that the matrices that worked did not have 3 lines going into any intersection, while the matrix that caused the program to crash, did have three lines going into one intersection. This can also be seen with another matrix (Figure G.1b) that has got 3 lines going into one intersection.

Looking at case 3 (4.3.3.2b), the algorithm would need refining as currently the algorithm uses true and false values. This was found be a problem, as if the last intersection did not match the common payoff, then the Boolean variable would be set to false, even though there were two intersections that generated the same payoff. This can be seen with the Figure E.2.1.3c.

Another refinement needed with the algorithm would be to deal with matrices that had three intersections that generated the same payoff as shown in Figure E.2.1.2c. That matrix has got three intersections that generate the
Determining Optimal Strategies in Matrix Games

same payoff. The program picks the intersection that is not on the payoff function as an optimal solution.

Looking at the error handling of the application, it received the same criticism as the 2*2 Matrix Class received.

7.3. Project Evaluation

From the Introduction and the Literature Review, the aim of the dissertation was to develop an application that produced the optimal strategies and payoff for players, in matrix games. The literature review went into detail and identified the scope of the problem.

The Requirements and Design Section produced the functionality that the program must abide to. The Implementation discussed the coding of the application and explained further the algorithms that were implemented. It also provided justification to certain programming practices and choice of algorithms.

The Testing actually tested the application to see if the application was functional. From Section 7.2, even though the 2*2 Matrix Game had a criticism against it, it did fulfil its requirements and function accordingly.

Even though from Section 7.2, it can deduced that for the 2*N Matrix Class not all of the requirements were met and the program did not function accordingly, the confidence generated from the testing was high. On a scale of 0-100, the level of confidence is 77, as 7 out of 9 matrices worked during the system test of the 2*N Matrix Class.

From the requirements that were implemented, not all of the non-functional requirements were implemented as I felt that more time should be spent, making sure that the functional requirements that were to be implemented, were actually implemented. I had also had to balance between the implementation of functional requirements and non-functional requirements, as mentioned in the Section 5.3, there were a few issues that lead to duplication of functions, which resulted in some non-functional requirements not being met.

On the whole I strongly feel that it was necessary to not meet the majority of the non-functional requirements and justified, as if I did manage to meet all of the non-functional requirements, I feel that I would not have met the aims and fulfilled the objectives of the dissertation, which was to produced an application to determine optimal strategies and payoffs in matrix games.

7.4. Possible Extensions

From reading Section 7.3 and looking at the testing results, the obvious extension is to determine a solution for the crash for the 2*N Matrix Game and refine the algorithm that dealt with special case 3. For both the 2*N and 2*2 Matrix Game programs, the suggestions mentioned in Section 7.2.1 with dealing with invalid data types, is another valid extension.

A possible refinement of the algorithm that dealt with special case 3 would be to take from the payoffs array, all of the payoffs that were the same. Then take the intersections associated with those payoffs and check to see if they lay on the two lines, closest to the horizontal axis. This should solve the
Determining Optimal Strategies in Matrix Games

issues that are present with the current algorithm. Due to the timing constraints I have not been able to implement the above solution.

The solution for the crash would be to investigate where the crash took place and understand why the crash occurred and try to determine a solution. Also further investigation into three lines intersecting into a single vertex would be explored. Again due to the timing constraints I have not been able to pursue this avenue.

Another apparent extension would be to implement the requirements that were not implemented such as, producing a graphical representation of the 2*N matrices, implementing a program that dealt with M*2 matrices and also implement the simplex method algorithm. Another extension could be implementation of a GUI, as the GUI could be used with the other extensions to allow the user to choose the type of matrices that they wish to deal with.

The above were extensions to the application. There could also be extensions made to the dissertation as well. Currently the dissertation deals with matrix games that are of type Zero-Sum. Extensions could be made to the dissertation by researching into the other types of matrix games, such as Non-Zero-Sum, Symmetric, Asymmetric and more.

7.5. Personal Evaluation

At the very start of the dissertation, I was terrified with the prospect of doing a dissertation. This was because the dissertation is worth 3 units and if failing the dissertation, would result in failing my degree. What lead to the terrifying notion of doing a dissertation was the programming aspect, as my programming skills were up to the standards as to my peers and I came to the conclusion that the ability to program would determine the final mark.

However due to attending the dissertation lectures, I found out the dissertation mark was made up from the writing of the actual dissertation report and not the programming. I also found out that the purpose of a dissertation was to bring forward new concepts and ideas and to use the skills that I learnt in the previous years at university.

The reason why I implemented the 2*2 Matrix Game, even though I was ill advised by my supervisor, was to battle the fears of my programming abilities, as I thought that it was a good starting point to begin the programming of the application.

Looking at 2*N Matrix Game, although the program does have some flaws with it, I am still pleased with the outcome, as the algorithms behind the program were challenging and did require me to spend copious amounts of time thinking logically.

Now at the end of the dissertation, I am satisfied with my performance as I was able to show and develop the skills that I developed, whilst studying at the University of Bath. I was particularly pleased with my programming as this was the first time at university that I had to write a large application and I feel that my programming skills have been increased! Even thought the end product did a have few criticisms and flaws, the aims and objectives of the dissertation were met, as I did produce an application that did manage to produce optimal strategies and payoffs for players, in zero-sum matrix games.
Determining Optimal Strategies in Matrix Games

Bibliography

Determining Optimal Strategies in Matrix Games

Appendix A

Glossary

A
Adjoint – The inverse of a matrix.

B
Basic Variables – Variables on the outer far right hand side of a table.

D
Determinant – The difference of the products of the diagonals inside a 2*2 matrix.

Dominant Strategy – when players certain strategy gives a higher payoff than the second strategy, regardless of which strategy the other player decides to use.

Domination – The removing of dominant strategies.

E
Extensive Form – A game, along with the strategies of the players and payoffs, are represented in a tree diagram.

F
Feasible Region – The region that satisfies the inequalities of a linear program.

Feasible Solutions – All the points in the feasible region.

G
Game Theory – A study and the field of strategic rational choices made by people.

L
Linear Programming – A function for maximising or minimising a linear equation subject to some constraints.
Determining Optimal Strategies in Matrix Games

**M**
*Matrix* – A rectangular table, full of numbers.

*Minimax Theorem* – A strategy that gives the smallest maximum. i.e. smallest payoff.

**N**
*Nash Equilibrium* – When a player chooses a strategy that is the best response to the strategies chosen by other players.

*Non-Basic Variables* - Variables on the outer upper side of a table.

*Normal Form* - A game, along with the strategies of the players and payoffs, are represented in table format.

**O**
*Objective Function* – Function subject to be maximised or minimised in linear programming.

**P**
*Pivots* – An entry in a matrix that is used to switch the other entries inside that matrix.

*Pure Strategy* – List of strategies and payoffs shown in a normal form game.

**S**
*Saddle Point* – An entry in a matrix, in which it is the maximum entry in its column and minimum entry in its row.

*Simplex Method* – An algorithm used for solving linear programming problems.

*Slack Variables* – A variable which is added to a constraint in linear programming, to turn the inequality into an equation.

**T**
*Transpose* – A matrix that is produced by taking the original matrix and turning the rows into columns and vice versa.

**Z**
*Zero-Sum Game* – A game in which the sum of the payoffs, are always equal to zero. This means that whatever one player wins, the other player losses.
Appendix B

Linear Algebra

B.1. Determinants
Associated with every square matrix is a number, known as the determinant. The determinant of a matrix is a tool used in many branches of mathematics, science and engineering (Williams 2001 and Simmons 2006).

The determinant of a matrix is determined by finding the difference of the products of the two diagonals of a matrix. Commonly, the determinant is denoted |<Name of matrix>|.

Therefore the determinant of a 2 * 2 Matrix, A is denoted |A| and is calculated as shown below;

\[ |A| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} \]

Figure B.1a Determinant

B.2. Adjoint of a Matrix
The adjoint of a matrix A, is denoted either as adj (A) or A* and is the transpose of the cofactors of matrix A. The cofactor of the matrix is determined via the removal of a row and column and then working out the determinant. The cofactor is then placed within the matrix and can be either positive or negative depending on the position it is placed in (Williams 2001 and Simmons 2006).

The calculation of the adjoint of a matrix is shown below;

\[ A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{Cofactor (A)} = \begin{pmatrix} a_{22} & -a_{21} \\ -a_{12} & +a_{11} \end{pmatrix} \]

\[ \text{Adj (A) or } A^* = (\text{Cofactor (A)})^T = \begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & +a_{11} \end{pmatrix} \]

Figure B.2a Adjoint of Matrix
Determining Optimal Strategies in Matrix Games

B.3. Multiplication of Vectors and Matrices

The multiplication of vectors and matrices is determined via the sum of products of the corresponding rows of a matrix with the columns of the vector. The figure below shows the multiplication of vectors and matrices.

\[
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad V = [x, y] \\
AV = [xa_{11} + xa_{21}, ya_{12} + ya_{22}]
\]

Figure B.3a Multiplication of Vectors and Matrix

B.4. Multiplication of Elements and Matrices

The multiplication of elements and matrices is referred to as a **Scalar Multiple** (Williams 2001). Scalar multiple of a matrix is determined by multiplying every single element inside the matrix with the scalar value (Williams 2001). The resulting scalar matrix will be the same size as the original matrix. The figure below shows scalar multiplication of a matrix;

\[
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{Scalar Value} = X \\
XA = \begin{pmatrix} xa_{11} & xa_{12} \\ xa_{21} & xa_{22} \end{pmatrix}
\]

Figure B.4a Scalar Multiplication of Matrix

B.5. Multiplication of Two Vectors

The multiplication of two vectors is referred to as the **Dot Product** (Williams 2001). The dot product is calculated via the sum of products of the corresponding elements within the two vectors (Williams 2001 and Simmons 2006), as shown below;

\[
U = (x_1, y_1) \quad V = (x_2, y_2) \\
\text{Dot product of } U \text{ and } V \text{ denoted as } U.V = x_1x_2 + y_1y_2
\]

Figure B.5a Dot Product of Two Vectors
Appendix C

Source Code

This Appendix contains the source for the 2*2 Matrix and 2*N Matrix Classes.
Determining Optimal Strategies in Matrix Games

```java
// This class is called the Two By Two Matrix. The function of this class
// is to generate a 2x2 matrix, in which the elements within the matrix, will
// either be randomly generated by the program or entered in by the user.
// Then the program will compute the optimal strategies for both players
// and the payoff function.

// Package name: TwoByTwoMatrix

class TwoByTwoMatrix {
    private int a, b, c, d;

    // Constructor which does not take any parameters.
    // The constructor assumes the size of the matrix
    // and sets the boolean value and initializes the
    // random generator.

    public TwoByTwoMatrix() {
        a = b = c = d = new Random();
        saddle = false;
    }

    // Method called RandomMatrix of integer array type.
    // This method does not take any parameters and returns
    // a two dimensional array. The purpose of this method
    // is to create 4 random numbers and assign them
    // into the two by two matrix and then print off the matrix.
    // using a created method called printMatrix().

    public int[][] RandomMatrix() {
        a = b = c = d = new Random();
        b = a.nextInt(100); // Random generation of numbers
        c = b.nextInt(100); // from 0 - 100
        d = c.nextInt(100); // from 0 - 100
        matrix = new int[2][2];
        matrix[0][0] = a; // Assigning the randomly generated
        matrix[0][1] = b; // numbers to their positions, within the
        matrix[1][0] = c; // matrix[1][1] = d;
        System.out.println("The matrix is as follows:");
        printMatrix();
        return matrix;
    }

    // Method called UserMatrix of integer array type.
    // This method takes four integer parameters and returns
    // a two dimensional array. The purpose of this method
    // is to allow the user to input 4 numbers into the

    public int[][] UserMatrix(int a, int b, int c, int d) {
        matrix[0][0] = a; // Assigning the parameters entered in
        matrix[0][1] = b; // by the user to their positions within
        matrix[1][0] = c; // the matrix.
        matrix[1][1] = d;
        System.out.println("The matrix is as follows:");
        printMatrix();
        return matrix;
    }

    // Method called saddlePoint which takes a two dimensional
    // array as a single parameter and does not return anything.
    // The purpose for this method is to determine given a matrix,
    // whether or not a saddle point exists. If a saddle point exists, the
    // function
    // prints off the saddle point element otherwise it states
    // that there is no saddle point in the given matrix.

    public void saddlePoint(int[] matrix) {
        saddle = false;
        a = matrix[0][0];
        b = matrix[1][1];
        saddle = true;
        System.out.println("The optimal strategy is for player 1 to choose
        the element " + matrix[0][0] + ";
        System.out.println("The optimal strategy is for player 2 to choose
        the element " + matrix[1][1] + ");
    }

    // Method called printMatrix() which takes a two dimensional
    // array as a parameter and prints all of its elements.
    // The purpose for this method is to display the
    // values of the array in a readable format.

    public void printMatrix() {
        System.out.println("The matrix is as follows:");
        System.out.println("Row 1:");
        System.out.println("Row 2: x x x x");
        System.out.println("Row 3: x x x x");
        System.out.println("Row 4: x x x x");
    }
}
```
Determining Optimal Strategies in Matrix Games

in

row i and for player j to choose column i.1
The payoff for player i is *: b * the payoff for
Player =

2 * i = x + by;

nondpoint = true;
saddle = true;

if (nondpoint == false || (b <= 0) & (d <= 0))
System.out.println("The element " + matrix[i][j] + " is a saddle point.");

System.out.println("The optimal strategy is for player i to choose
Player =

" + i + ")

nondpoint = true;
saddle = true;

if (nondpoint == false) // not nondpoint
System.out.println("There are no saddle points in this payoff
table."");
saddle = false;

/*
Method called printMatrix: This method takes a two dimensional
integer array as a single parameter and does not return anything.
The purpose of this method is to print off the matrix table.

public void printMatrix(int[][] matrix)
{
System.out.println("***** Beginning of Matrix *****");
for (int row = 0; row < row; row++)
for (int column = 0; column < col; column++)
System.out.print(matrix[row][column] + " ");
System.out.println();

System.out.println("***** End of Matrix *****");
System.out.println();

*/

/*
Method called determinant: This method takes in an integer
the two dimensional array as a single parameter and returns
on integer. The purpose of this method is to work out the
Determinant for a given matrix.

public int determinant(int[][] matrix)
{
int determinant, x, y;
//Performing the necessary calculations in working out the determinant
x = (matrix[0][0] * matrix[1][1]) - (matrix[0][1] * matrix[1][0]);
y = (matrix[0][0] * matrix[1][1]) - (matrix[0][1] * matrix[1][0]);

//Determinant is: x = y / working out the determinant.
System.out.println("The determinant is " + determinant + ");
return determinant;
*/

/*
Method called adjoint: This method takes in an integer
2 by 2 array as a single parameter and returns
an integer two dimensional array. The purpose of this method
is to work out the adjoint of a given matrix and assign it
to a newly created matrix.

public int[][] adjoint(int[][] matrix)
{
int[][] newMatrix = new int[row][col];
int a, b, c, d;
a = matrix[0][0];
b = matrix[0][1];
c = matrix[1][0];
d = matrix[1][1];
newMatrix[0][0] = d;
newMatrix[0][1] = -b;
newMatrix[1][0] = -c;
newMatrix[1][1] = a;
return newMatrix;
*/

/*
Method called matrixPrint: This method does not return anything.
but takes in a two dimensional integer array as a single parameter.
The purpose for this method is to print off the adjoint
matrix, using the printMatrix method.

public void matrixPrint(int[][] matrix)
{
int[][] newMatrix = new int[row][col];

int & adjoint = new int[][col]; // creation and initialization of
adjoint = adjoint(matrix) //performing the adjoint
System.out.println("Printing off the Adjoint Matrix: ");
printMatrix(adjoint) // printing off the adjoint matrix.
*/

/*
Method called vectorFromMatrix: Adjoint of integer array type.
This method takes in an integer two dimensional array
as a single parameter and returns an integer array.
The purpose of this method is to determine values
to be used to perform the calculations to work out the optimal
strategy for the players and the payoff.

public int[] vectorFromMatrix(int[][] matrix)
{
int[], a, y;
int[][] newMatrix;
newMatrix = new int[row][col];
int[] newMatrix = new int[2];
matrix[0] = 1;
matrix[1] = 1;
y = matrix[0][0];
newMatrix[0] = a;
newMatrix[1] = b;
for (int row = 0; row < row; row++)
System.out.print(matrix[row][col] + ");

int result[] = new int[2];
result[0] = x / y; // storing the results in an array
result[1] = y;
return result // returning the array.
Determine optimal strategies in Matrix Games.

```java
public int[] vectorTimesTranspose(int[] matrix)
{
    int x, y;
    int[] result = new int[2];
    matrix[0] = x;
    matrix[1] = y;
    return result;
}
```

```java
public double adjointTimesVectorTimesTranspose(int[] matrix, int[] q, int[] w)
{
    // q = x value from vector times transpose function --> array position
    // w = y value from vector times transpose function --> array position
    int x, y;
    int[] newMatrix;
    newMatrix = new int[2][1];
    // performing the adjoint and assigning it to new matrix created
    // assigning new A*AJ
    x = (q[0] * newMatrix[0][0]) + (q[1] * newMatrix[1][0]);
    y = (w[0] * newMatrix[0][0]) + (w[1] * newMatrix[1][0]);
    System.out.println("The value of A*AJ = \(x, y\) = \(" + x + ", " + y + ");
    return result[0] = new int[3];
    result[0] = x; // storing the results in the array
    result[1] = y;
    return result[2]; // returning the array
}
```

```java
public void probPayoff(int[] players, int[] playersInt) {...
    int vec_ad_x, vec_ad_y,
    int vec_trans_x,
    int vec_trans_y;
    vec_ad_x = (x * vec_trans_y) + (y * vec_trans_y);
    vec_ad_y = (y * vec_trans_x) + (y * vec_trans_y);
    int result[0] = new int[1];
    System.out.println(\"The value of JA*JT = \" + theResult[0] + \");
    return result[2]; // returning the array
}
```

```java
public boolean check(double x, double y)
{
    // if statements checking whether the probabilities are valid
    if (x > 0 && y > 0) {
        System.out.println(\"Probabilities \(x, y\) are valid.\"");
        return true;
    }
    if (x < 0 || y < 0) {
        System.out.println(\"Probabilities \(x, y\) are invalid.\"");
        return false;
    }
    if (x + y < 1) {
        System.out.println(\"Probabilities \(x, y\) are invalid.\"");
        return false;
    }
    return true;
}
```

```java
public void probPayoff2(int[] players, int[] playersInt) {...
    int vec_ad_x, vec_ad_y,
    int vec_trans_x,
    int vec_trans_y;
    vec_ad_x = (x * vec_trans_y) + (y * vec_trans_y);
    vec_ad_y = (y * vec_trans_x) + (y * vec_trans_y);
    int result[0] = new int[1];
    System.out.println(\"The value of JA*JT = \" + theResult[0] + \");
    return result[2]; // returning the array
}
```
Determining Optimal Strategies in Matrix Games

n = false
System.out.println("Enter your name. ");
name = in.nextLine(); //reading in the user input

String text = "Welcome " + name + " to the program that works out
optimal strategies in a 2 x 2 two-person zero sum matrix game.
\nString [] lines = game.wrapText(text, 60);
for (int i = 0; i < lines.length; ++i)
System.out.println(lines[i]);

if (n == false)
System.out.println("Enter 1 to generate a random matrix or 2 to generate a
custom matrix which you will have to enter. ");

//while loop, looping until the user enters the input 1 or 2
while (n == false)
try//error handling, just in case the user enters invalid data.

if (number > 2 || number < 1) //if the user does not enter 1 or 2 they are notified

System.out.println("You must enter 1 or 2.");
//Number 1 to generate a randomly generated matrix
//Number 2 to generate a custom matrix which you can enter.

//in here: number = Integer.parseInt(n);

lines = game.wrapText(text, 60);
for (int i = 0; i < lines.length; ++i)
System.out.println(lines[i]);

switch (number) {

if (number == 1)
System.out.println("I was entered. ");
//providing feedback
System.out.println("The matrix will be randomly generated. ");

//terminating the loop
break;
}

if (number == 2)
System.out.println("I was entered. ");
System.out.println("Enter the four inputs. Invalid values
are as follows:
Strings, Floats, Doubles and Characters ");

//terminating the loop
break;

if (number > 2 || number < 1)
//error handling still monitoring
System.out.println("Enter 1 or 2. ");
break;
}

System.out.println("Using random matrix generator.
//matrix is generated with

match(Exception e) //error handling message
System.out.println("You were supposed to enter in an
integer, but a string was entered. ");
System.out.println("This will stop an infinite loop running

//checking whether saddle point exists using global parameters, saddle
// if saddle point exists, global parameter is set to true, within
// the saddlePoint() function, thus terminating the if statement.

if (game.saddle == false)

if (game.saddlePoint(gTheoy);
//If the global parameter is still false i.e. no saddle point exists,
//the probabilities of the players and payoff are determined.
if (game.saddle == false)

//if the user entered the input 1 or 2
while (n == false)
try//error handling, just in case the user enters invalid data.

if (number > 2 || number < 1) //if the user does not enter 1 or 2 they are notified

System.out.println("You must enter 1 or 2.");
//Number 1 to generate a randomly generated matrix
//Number 2 to generate a custom matrix which you can enter.

//in here: number = Integer.parseInt(n);

lines = game.wrapText(text, 60);
for (int i = 0; i < lines.length; ++i)
System.out.println(lines[i]);

switch (number) {

if (number == 1)
System.out.println("I was entered. ");
//providing feedback
System.out.println("The matrix will be randomly generated. ");

//terminating the loop
break;
}

if (number == 2)
System.out.println("I was entered. ");
System.out.println("Enter the four inputs. Invalid values
are as follows:
Strings, Floats, Doubles and Characters ");

//terminating the loop
break;

if (number > 2 || number < 1)
//error handling still monitoring
System.out.println("Enter 1 or 2. ");
break;
}

System.out.println("Using random matrix generator.
//matrix is generated with

match(Exception e) //error handling message
System.out.println("You were supposed to enter in an
integer, but a string was entered. ");
System.out.println("This will stop an infinite loop running

//checking whether saddle point exists using global parameters, saddle
// if saddle point exists, global parameter is set to true, within
// the saddlePoint() function, thus terminating the if statement.
Determining optimal strategies in matrix games

```java
def game(matrix):
    # ... code for finding optimal strategies...
```

```java
if game.middlePoint == False:
    print(game.middlePoint) # If the global parameter is still false i.e. no saddle point exists
```

```java
while (number != False):
    try:
        while True:
            # Code for input handling and error checking ...

            # Code for generating and inputting matrix ...

            # Code for error handling and user feedback ...
```

```java
if game.middlePoint == False:
    print(game.middlePoint)
```

public double gradient(int [][]eMatrix, int size)
{
    int v, q, h;
    double [] equations = new double[size]; //array that will depend on the number of columns
    int apos = 0;
    for(int c = 0; c < size; c++)
        { p = eMatrix[0][c];
          q = eMatrix[1][c];
          h = q*p;
          equations[apos] = h; //storing the gradient values in the array
          apos++;
        }
    return equations;
}

public double intersection(int [][]eMatrix, int size)
{
    int p;
    double [] equations = new double[size]; //array that will depend on the number of columns
    int apos = 0;
    for(int c = 0; c < size; c++)
        { p = eMatrix[0][c];
          //storing the intersection value (i.e. value from ay = bx + c)
          equations[apos] = p;
          apos++;
        }
    return equations;
}
Determining Optimal Strategies in Matrix Games

```java
public void printEquations(int[] equation, int n, int n2):
    for(int i = 0; i < n; i++)
        System.out.println("Equation "+equation[i]+" = "+");
```

Method called gradient, which takes an integer and an integer as its parameter. This function returns a double array. The purpose of this method is to compute the gradient of the equations of the lines that are formed from the matrix.

```java
public double[] gradient(int size)
```

Method calls intersection, which takes an integer and a double as its parameter. The purpose of this method is to compute the intersection of the equations of the lines that are formed from the matrix.

```java
public double intersection(int size)
```

Method called pay_off, which takes an integer and a double as its parameters. The purpose of this method is to compute the payoff of the matrix.

```java
public double pay_off(int[][] matrix)
```

Method called intersection, which takes an integer and a double as its parameters. The purpose of this method is to compute the intersection of the equations of the lines that are formed from the matrix.

```java
public double intersection(int size)
```
Determining Optimal Strategies in Matrix Games

```java
public void printEquations(int [][]matrix, int size) {
    // Need to print off the equations in the form
    // p - (a1 * x1) + (a2 * x2) + ... + (a11 * x11)
    // from the second column and add a to the first column
    int p, q, r;
    for(int c = 0; c < size; c++)
    {
        int number = ++c;
        p = matrix[0][c];
        q = matrix[1][c];
        h = p + q;
        System.out.println("Equation ", number, ": i = \"h \times x\" + ", " + p);
    }
}
```

Method called gradient, which takes an integer.
- 2D array and an integer as its parameter. This function returns a double array. The purpose of this method is to compute the gradient of the equations of the lines that are formed from the matrix.

```java
public double[] gradient(int [][]matrix, int size) {
    int p, q, r;
    double [] equations = new double[size]; // Arrays will depend on the number of columns
    int apex = 0;
    for(int c = 0; c < size; c++)
    {
        p = matrix[0][c];
        q = matrix[1][c];
        h = p + q;
        equations[apex] = h; // Storing the gradient values in the array
        apex++;
    }
    return equations;
}
```

Method called intersection, which takes an integer.
- 2D array and an integer as its parameter. This function returns a double array. The purpose of this method is to compute the intersection of the equations of the lines that are formed from the matrix.

```java
public double[] intersection(int [][]matrix, int size) {
    int p, q, r;
    double [] equations = new double[size]; // Array size will depend on the number of columns
    int apex = 0;
    for(int c = 0; c < size; c++)
    {
        p = matrix[0][c];
        q = matrix[1][c];
        h = p + q;
        equations[apex] = h; // Storing the intersection values in the array
        apex++;
    }
    return equations;
}
```

Method called payoff, which takes an integer.
- 3D array as a single parameter. This function returns a double value. The purpose of this method is to compute the payoff off the matrix.

```java
public double payoff(int [][]matrix) {
    int a, b, c, d;
    a = matrix[0][0];
    b = matrix[0][1];
    c = matrix[1][0];
    d = matrix[1][1];
    int p = (a * d) - (b * c); // Performing calculations
    int post = a + b + c + d;
    double payoff = (double)p / (double)post; // Determining payoff
    return payoff;
}
```

Method called Player1, which takes an integer.
- 3D array as a single parameter. This function returns a double value. The purpose of this method is to compute the probability for player 1.

```java
public double Player1(int [][]m) {
    int stop, shot;
    double x;
    int a, b, c, d;
    a = m[0][0];
    b = m[0][1];
    c = m[1][0];
    d = m[1][1];
    stop = a + c - d - b;
    shot = a + b + c + d;
    x = (double)stop / (double)shot;
    return x;
}
```

Method called Player2, which takes an integer.
- 3D array as a single parameter. This function returns a double value. The purpose of this method is to compute the probability for player 2.

```java
public double Player2(int [][]m) {
    int ystop, yshot;
    double y;
    int a, b, c, d;
    a = m[0][0];
    b = m[0][1];
    c = m[1][0];
    d = m[1][1];
    ystop = a + b;
    yshot = a + b + c; // Determine y
    y = (double)ystop / (double)yshot;
    return y;
}
```

Method called smallestMatrix, which takes an integer.
- 3D array as a single parameter. This function does not return any value. The purpose of this method is to compute and print all the 2*N matrices that can be formed from the 2*N matrix, entered in by the user.
Determining Optimal Strategies in Matrix Games

```java
// need to figure out i-player
for(int n = 0; n < player.length; n++)
    prob[n] = (i - player[n]);

// three spaces for these points
for(int lines = 0; lines < intersection.length; lines++)
    if (\n        if (Math.abs(value) <= 0.00001)
            // check that the intersections lay on the line.
            
int i = 0;
    for(int counter = 0; counter < prob.length; counter++)
        
// if (Math.abs(value) <= 0.00001) // checking the value is close to 0.
    
// performing the sorts
    if (sort[i] - sort[i-1] > 0) // comparing the 2 values
        temp = sort[i-1];
    sort[i-1] = sort[i];
    sort[i] = temp;

    return sort;
```
Appendix D

Test Cases

This appendix contains the test scripts for the unit testing and system testing of the 2*2 Matrix Class and 2*N Matrix Class. The test cases give an indication of whether the test passed or failed and allowed the tester to add further comments.
Determining Optimal Strategies in Matrix Games

D.1 Two by Two Matrix Game Test Scripts

All of the following test scripts were performed using the following conditions:

<table>
<thead>
<tr>
<th>Laptop Type</th>
<th>Sony VAIO PCG-FR285M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Version</td>
<td>Two By Two Matrix</td>
</tr>
<tr>
<td>Processor</td>
<td>Pentium (R) 4 CPU 2.40 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>488 MB of RAM</td>
</tr>
<tr>
<td>IDE</td>
<td>Eclipse SDK Version 3.2.2</td>
</tr>
</tbody>
</table>

Unit Test Case 1: Creation a random matrix and printing the random matrix

BRIEF DESCRIPTION:
This test case will test the following requirements;
- 3.3.1.1 – Matrix generated by the program must be of correct size
- 3.3.1.3 – The application must provide the means, for elements to be randomly generated inside the matrix
- 3.3.1.5 – Elements in the matrix table must be valid i.e. not characters, floats and string

PREPARATION:
Run the Two by Two Matrix code on Eclipse. When program starts enter in your name.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The program should welcome the user, to the application. The program will ask the user to either chose 1 to generate a random matrix or 2 to generate a matrix, in which users can enter in data themselves.</td>
<td>1. The user should be given a choice of 1 to generate a random matrix or 2 to generate a matrix, in which users can enter in data themselves. 2. The program should state that the number</td>
<td>PASS</td>
<td>Possible extension to the program is that when illegal parameters are entered, such as strings etc... that</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

2. Enter in the number 3.
3. Enter in number -1.
4. Enter in a character.
5. Re-do preparation and enter in a string.
6. Re-do preparation and enter in a float.
7. Re-do preparation and enter in 1.

<table>
<thead>
<tr>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>the program should not terminate but should loop, until valid parameters are entered.</td>
<td></td>
</tr>
</tbody>
</table>

Three is not a valid choice and will ask the user to enter in valid data, such as 0 or 1.
3. See expected result number 2.
4. Program should terminate indicating to the user that an illegal data type was entered.
5. See expected result number 4.
6. See expected result number 4.
7. This is valid data. Program should generate a two dimensional matrix with the column and row size of 2. The elements inside the matrix should be randomly generated. The application should print off the matrix.

Unit Test Case 2: Creation of user specified matrix and printing the user specified matrix

**BRIEF DESCRIPTION:**
This test case will test the following requirements;
- 3.3.1.1 – Matrix generated by the program must be of correct size
- 3.3.1.4 – The application must provide the means, for elements to be entered inside the matrix, by users
- 3.3.1.5 – Elements in the matrix table must be valid i.e. not characters, floats and string

**PREPARATION:**
Run the Two by Two Matrix code on Eclipse. When program starts enter in your name.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The program should welcome the user, to the application. The program will ask the user to either chose 1 to generate a random matrix or 2</td>
<td>1. The user should be given a choice of 1 to generate a random matrix or 2 to generate a matrix, in which users can enter in data</td>
<td>PASS</td>
<td>Possible extension to the program is that when illegal parameters are entered.</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

to generate a matrix, in which users can enter in data themselves.
2. Enter in the number 3.
3. Enter in number -1.
4. Enter in a character.
5. Re-do preparation and enter in a string.
6. Re-do preparation and enter in a float.
7. Re-do preparation and enter in 2.
8. Enter in a character.
9. Re-do step 7 and enter in a string.
10. Re-do step 7 and enter in a float.
11. Re-do preparation and enter in four integer elements, with one of the numbers being 0 and one number being negative.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2. The program should state that the number three is not a valid choice and will ask the user to enter in valid data, such as 0 or 1.</td>
<td></td>
</tr>
<tr>
<td>3. See expected result number 2.</td>
<td></td>
</tr>
<tr>
<td>4. Program should terminate indicating to the user that an illegal data type was entered.</td>
<td></td>
</tr>
<tr>
<td>5. See expected result number 4.</td>
<td></td>
</tr>
<tr>
<td>6. See expected result number 4.</td>
<td></td>
</tr>
<tr>
<td>7. This is valid data. The program should indicate to the user the option they have chosen and inform the user to enter in 4 elements. The program should inform the user what elements are valid and invalid.</td>
<td></td>
</tr>
<tr>
<td>8. See expected result number 4.</td>
<td></td>
</tr>
<tr>
<td>9. See expected result number 4.</td>
<td></td>
</tr>
<tr>
<td>10. See expected result number 4.</td>
<td></td>
</tr>
<tr>
<td>11. The program should expect the users’ input and produce a two by two matrix with the elements inside the matrix, being the values that the users entered in. The matrix should be printed off.</td>
<td></td>
</tr>
</tbody>
</table>

Unit Test Case 3: Checking for saddle points

**BRIEF DESCRIPTION:**
This test case will test the following requirements;
- 3.3.1.2 – Saddle points must be identifiable by the program

**PREPARATION:**
Determining Optimal Strategies in Matrix Games

Generate the following user specified matrices \( \{\{1, 0\}, \{-1, 2\}\} \) and \( \{\{-2, -2\}, \{3, 0\}\} \) and two randomly generated matrices and assign them to different two dimensional matrices.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the function saddle point, passing the first user specified matrix.</td>
<td>1. The program should state to the user that the matrix does not have any saddle points.</td>
<td>PASS</td>
<td></td>
</tr>
<tr>
<td>2. Run the function saddle point, passing the second user specified matrix.</td>
<td>2. The program should state to the user that the element 0, in matrix position 2, 2 is the saddle point.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Run the function saddle point, passing the first randomly generated matrix.</td>
<td>3. Depending on the matrix generated, the program should either state that there is a saddle point and state the element. Or the program should state that no saddle point exists within the matrix.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Run the function saddle point, passing the second randomly generated matrix.</td>
<td>4. See expected result 3.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unit Test Case 4: Calculating the determinant

**BRIEF DESCRIPTION:**
This test case will test the following requirements;
- 3.3.1.7.1 – The determinant of the matrix must be calculated.
- 3.3.1.9.3 – Calculate the numerator for the following formula \(|A| /JA^T\).

**PREPARATION:**
Re-use the matrices without saddle points used in unit test 3.
The random matrix without saddle points that were used in unit test 3 was; \( \{\{93, 4\}, \{7, 55\}\} \)
The user specified matrix without saddle points that were used in unit test 3 was; \( \{\{1, 0\}, \{-1, 2\}\} \)
## Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
</table>
| 1. Run the function determinant passing the random matrix specified in the preparation.  
  2. Run the function determinant this time passing the user specified matrix specified in the preparation. | 1. The program should calculate the determinant\(^8\).  
  2. The program should calculate the determinant\(^9\). | PASS          |          |

### Unit Test Case 5: Performing the adjoint operation on a matrix and printing off the adjoint matrix

**BRIEF DESCRIPTION:**

This test case will test the following requirements:

- 3.3.1.7.2 – The adjoint of the matrix must be calculated.

**PREPARATION:**

Re-use matrices that were used in unit test 4.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
</table>
| 1. Run the function adjoint passing the random matrix specified in the preparation.  
  2. Run the function adjoint passing the user specified matrix specified in the preparation. | 1. The program should produce the adjoint of the random matrix\(^10\).  
  2. The program should produce the adjoint of the user specified matrix\(^11\). | PASS          |          |

\(^8\) See Appendix E – Test Data, for manual results.  
\(^9\) See Appendix E – Test Data, for manual results.
Determining Optimal Strategies in Matrix Games

Unit Test Case 6: Calculating the numerator for the formula (4.3.2.2.1)

**BRIEF DESCRIPTION:**
This test case will test the following requirements:
- 3.3.1.9.1 – Calculate probability for player one by calculating the numerator for the following formula \( \frac{JA^*}{JA^*J^T} \).

**PREPARATION:**
Re-use matrices that were used in unit test 5.

**ACTION:**

<table>
<thead>
<tr>
<th></th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Run the function vector times adjoint passing the random matrix specified in the preparation.</td>
<td>1. The program should calculate using the random matrix, the JA* which is the numerator used in the formula (4.3.2.2.1) (^{12}).</td>
<td>PASS</td>
</tr>
<tr>
<td>2.</td>
<td>Run the function vector times adjoint passing the user specified matrix specified in the preparation.</td>
<td>2. The program should calculate using the user specified matrix, the JA* which is the numerator used in the formula (4.3.2.2.1) (^{13}).</td>
<td></td>
</tr>
</tbody>
</table>

Unit Test Case 7: Calculating the denominator for the formulas (4.3.2.2.1), (4.3.2.2.2) and (4.3.2.2.3)

**BRIEF DESCRIPTION:**
This test case will test the following requirements:
- 3.3.1.9 – Calculate the denominator for the following formula \( \frac{JA^*}{JA^*J^T} \).
- 3.3.1.9 – Calculate the denominator for the following formula \( \frac{A^*J}{JA^*J^T} \).

\(^{10}\) See Appendix E – Test Data, for manual results.
\(^{11}\) See Appendix E – Test Data, for manual results.
\(^{12}\) See Appendix E – Test Data, for manual results.
\(^{13}\) See Appendix E – Test Data, for manual results.
Determining Optimal Strategies in Matrix Games

- 3.3.1.9 – Calculate the denominator for the following formula $|A|/JA^*J^T$.

**PREPARATION:**

Re-use matrices that were used in unit test 6.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the function vector times adjoint times vector times transpose, passing the random matrix as specified in the preparation. 2. Run the function vector times adjoint times vector times transpose, passing the user specified matrix as specified in the preparation.</td>
<td>1. The program should calculate using the random matrix, the $JA^*J^T$ which is the denominator used in formulas; (4.3.2.2.1), (4.3.2.2.2) and (4.3.2.2.3)(^{14}). 2. The program should calculate using the user specified matrix, the $JA^*J^T$ which is the denominator used in the formulas (4.3.2.2.1), (4.3.2.2.2) and (4.3.2.2.3)(^{15}).</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>

Unit Test Case 8: Calculating the numerator for the formula (4.3.2.2.2)

**BRIEF DESCRIPTION:**

This test case will test the following requirements;

- 3.3.1.9.2 – Calculate the numerator for the following formula $A*J^T / JA^*J^T$.

**PREPARATION:**

Re-use matrices that were used in unit test 7.

---

\(^{14}\) See Appendix E – Test Data, for manual results.  
\(^{15}\) See Appendix E – Test Data, for manual results.
## Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the function adjoint times vector times transpose, passing the random matrix as specified in the preparation. 2. Run the function adjoint times vector times transpose, passing the user specified matrix as specified in the preparation.</td>
<td>1. The program should calculate using the random matrix, the $A^*J^T$ which is the numerator used in the formula $\frac{\text{formula}}{\text{formula}}$(^{16}). 2. The program should calculate using the user specified matrix, the $A^*J^T$ which is the numerator used in the formula $\frac{\text{formula}}{\text{formula}}$(^{17}).</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>

### Unit Test Case 9: Calculating the numerator for the formula (4.3.2.2.3)

### BRIEF DESCRIPTION:
This test case will test the following requirements:
- 3.3.1.9.3 – Calculate the numerator for the following formula $|A| / JA^*$.  

### PREPARATION:
Re-use matrices that were used in unit test 8.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. See unit test 4.</td>
<td>1. See unit test 4.</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>

---

\(^{16}\) See Appendix E – Test Data, for manual results.  
\(^{17}\) See Appendix E – Test Data, for manual results.
Determining Optimal Strategies in Matrix Games
Unit Test Case 10: Calculating the probability for players I and II

**BRIEF DESCRIPTION:**
This test case will test the following requirements;
- 3.3.1.8 – Using the determinant, J vector, transpose of J vector and original matrix values, the program must be able to work out, the probabilities for the players involved in the game and the payoff of the game

**PREPARATION:**
Use 2*2 matrices specified in Appendix <appendix name> testing and test data handout

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
</table>
| 1. Run the function prob payoff, using the results gathered from the random matrix.  
2. Run the function prob payoff, using the results gathered from the user specified matrix. | 1. The program should calculate the probability for both players\(^{18}\).  
2. The program should calculate the probability for both players\(^{19}\). | PASS | |

**ACTION:**
1. Run the function prob payoff, using the results gathered from the random matrix.
2. Run the function prob payoff, using the results gathered from the user specified matrix.

**EXPECTED RESULT:**
1. The program should calculate the probability for both players\(^{18}\).
2. The program should calculate the probability for both players\(^{19}\).

**ACTUAL RESULT (PASS/FAIL):**
PASS

**UNIT TEST CASE 11: Performing probability validity check**

**BRIEF DESCRIPTION:**
This test case will test the following requirements;
- 3.3.1.10 – The program must indicate whether the probabilities are valid or invalid.

**PREPARATION:**
Re-use the probabilities gathered from unit test 10.

\(^{18}\) See Appendix E – Test Data, for manual results.
\(^{19}\) See Appendix E – Test Data, for manual results.
Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Pass the probabilities from the random matrix into the function check.</td>
<td>1. The program should determine that the probabilities are valid(^{20}).</td>
<td>PASS</td>
<td></td>
</tr>
<tr>
<td>2. Pass the probabilities from the user specified matrix into the function check.</td>
<td>2. The program should determine that the probabilities are valid(^{21}).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Pass the values (1, 3) into the function check.</td>
<td>3. The program should state that the probabilities are invalid.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Pass the values (1/4, 3/4) into the function check.</td>
<td>4. The program should state that the probabilities are valid.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Pass the values (-1, 2) into the function check.</td>
<td>5. The program should state that the probabilities are invalid.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Pass values (-2/5, 3/5) into the function check.</td>
<td>6. The program should state that the probabilities are invalid.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unit Test Case 12: Calculating the payoff

**BRIEF DESCRIPTION:**
This test case will test the following requirements;
- 3.3.1.8 – Using the determinant, adjoint, transpose and original matrix values, the program must be able to work out the payoff for the players involved in the game.

**PREPARATION:**
Re-use values that were used in unit test 10.

\(^{20}\) See Appendix E – Test Data, for manual results.
\(^{21}\) See Appendix E – Test Data, for manual results.
## Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
</table>
| 1. Run the function prob payoff, using the results gathered from the random matrix.  
2. Run the function prob payoff, using the results gathered from the user specified matrix. | 1. The program should calculate the payoff of the game\(^{22}\).  
2. The program should calculate the payoff of the game\(^{23}\). | PASS |  |

### System Test Case 1: System Test

**BRIEF DESCRIPTION:**
This test case will test all of the requirements specified in Section 3.3.1 and the specification mentioned in Section 4.3.2.

**PREPARATION:**
Have the original main method of the application executed.
Use user specified matrices that were used in the unit tests.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
</table>
| 1. Enter your name.  
2. Enter in 1.  
3. Run the application again.  
4. Enter your name. | 1. Program should greet the user with a welcome message.  
2. Random matrix should be generated and optimal strategies and payoff should be | PASS |  |

\(^{22}\) See Appendix E – Test Data, for manual results.  
\(^{23}\) See Appendix E – Test Data, for manual results.
## Determining Optimal Strategies in Matrix Games

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>Program should ask for the users’ name.</td>
</tr>
<tr>
<td>4.</td>
<td>Program should greet the user with a welcome message and ask the user whether they wish to randomly generate a matrix or enter in the elements into a blank 2*2 matrix, themselves.</td>
</tr>
<tr>
<td>5.</td>
<td>Program should ask the user to enter in four integer values.</td>
</tr>
<tr>
<td>6.</td>
<td>The program should state that the matrix does have a saddle point.</td>
</tr>
<tr>
<td>7.</td>
<td>See expected results 3 – 5.</td>
</tr>
<tr>
<td>8.</td>
<td>The program should state that the matrix does not have a saddle point and therefore workout the probabilities and payoff of the players, using the formulas and then checking to see if the probabilities are valid or not.</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

D.2 Two by N Matrix Game Test Scripts

All of the following test scripts were performed using the following conditions;

<table>
<thead>
<tr>
<th>Laptop Type</th>
<th>Sony VAIO PCG-FR285M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Version</td>
<td>Two By N Matrix</td>
</tr>
<tr>
<td>Processor</td>
<td>Pentium (R) 4 CPU 2.40 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>488 MB of RAM</td>
</tr>
<tr>
<td>IDE</td>
<td>Eclipse SDK Version 3.2.2</td>
</tr>
</tbody>
</table>

Unit Test Case 1: Creation of the 2*N matrix

**BRIEF DESCRIPTION:**
This test case will test the following requirement;

- 3.3.2.1 – Program must allow users to enter in the length or width of their choice or randomly generate the length or width of the matrices.

**PREPARATION:**
Run the Two by N Matrix code on Eclipse.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Enter in string.</td>
<td>1. Program should inform the user that invalid data was entered and then terminate.</td>
<td>PASS</td>
<td>Possible extension to the program is that when illegal parameters are entered, such as strings etc... that the program</td>
</tr>
<tr>
<td>2. Rerun the code again.</td>
<td>2. Program should ask the user to enter in the number of columns they wish the matrix to have.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Enter in a float.</td>
<td>3. See expected result 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Rerun the code again.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Enter in a negative number.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Enter in the number 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

7. Enter in the number 1.
8. Enter a number greater than 1.

4. See expected result 2.
5. The program should tell the user a negative value is not valid.
6. The program should tell the user that 0 is not valid.
7. The program should tell the user that 1 is not valid.
8. Program should expect the number that the user enters and state to the user that they have to input elements into the matrix.

should not terminate but should loop, until valid parameters are entered.

Unit test Case 2: Entering data into the matrix and printing off the matrix

BRIEF DESCRIPTION:
This test case will test the following requirement;
  3.3.2.2 – That value entered in by the user or by the program must be valid. Any invalid data such as floats, characters and strings must be rejected by the program and the program must ask for valid data.

PREPARATION:
Run test 1.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. When the program asks for elements to be entered, enter in a float number.</td>
<td>1. Program should inform the user that invalid data was entered and then terminate.</td>
<td>PASS</td>
<td></td>
</tr>
<tr>
<td>2. Re-do the preparation task.</td>
<td>2. The program should ask the user to input (the number of columns times two) elements into the matrix.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Re-do action 1, this time entering a string.</td>
<td>3. See expected result 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Re-do action 2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Re-do action 1, this time entering a negative</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

1. integer.
2. Enter a positive integer.
3. Enter the integer 0.
4. Enter more integer elements until the matrix is full.
5. See expected result 2.
6. Program should expect the negative integer and inform the user that there are (number of columns - 1) elements left to enter into the matrix.
7. Program should expect the positive integer and inform the user of the number of elements left to enter into the matrix.
8. Program should expect the integer 0 and inform the user of the number of elements left to enter into the matrix.
9. User should be able to enter the elements into the matrix. Once the matrix is full, the program should print off the matrix.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Test Case 3: Working out the linear equations of the lines</td>
<td></td>
</tr>
<tr>
<td>BRIEF DESCRIPTION:</td>
<td>This test case will verify that the program, produces linear equations from 2*N matrices. The equations will be used to determine optimal strategies and payoffs for the players.</td>
</tr>
<tr>
<td>PREPARATION:</td>
<td>Create a 2*4 matrix and input the following elements into the matrix; {{2, 3, 1, 5},{4, 1, 6, 0}}</td>
</tr>
</tbody>
</table>
### Determining Optimal Strategies in Matrix Games

| 1. Run the function print Equations, passing the matrix created in the preparation. | 1. The following lines should be created, in the following order\(^\text{24}\);  
  - \( Y = 2x + 2 \)  
  - \( Y = -2x + 3 \)  
  - \( Y = 5x + 1 \)  
  - \( Y = -5x + 5 \) | PASS |

#### Unit Test Case 4: Working out the payoffs of all the lines

**BRIEF DESCRIPTION:**
This test case will verify that the program can compute the payoffs for where all of the lines intersect and also obtain \(2*2\) matrices from \(2*N\) matrices. This test case will test the variation of requirements;
- 3.3.2.4 - Once the matrices are in \(2*2\) size, the program must be able to used the formulas in requirement 3.3.1.9, to determine the optimal strategies and payoff for the players, using formulas (4.3.3.1.4), (4.3.3.1.5) and (4.3.3.1.6).
- 3.3.3.2 - The application must be able to determine optimal strategies and payoffs for maximising gain and minimising loss, for both of the players.

**PREPARATION:**
Create numerous amounts of \(2*N\) matrices\(^\text{25}\).

<table>
<thead>
<tr>
<th>ACTION: Pass the (2*N) matrices into the function small</th>
<th>EXPECTED RESULT: For each (2*N) matrix entered in by the user, it</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Pass the (2*N) matrices into the function small</td>
<td>1. For each (2*N) matrix entered in by the user, it</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>

\(^{24}\) See Appendix E – Test Data, for manual results.  
\(^{25}\) See Appendix E – Test Data, for manual results.
Determining Optimal Strategies in Matrix Games

| matrix and into the function small matrix payoff. | will be displayed as a combination of 2*2 matrixes and for each 2*2 matrix, the payoff should be computed\(^\text{26}\). |

Unit Test Case 5: Working out the probabilities of all the lines

**BRIEF DESCRIPTION:**
This test case will verify that the program can compute the probabilities for where all of the lines intersect. This test case will test the variation of requirements;
- 3.3.2.4 - Once the matrices are in 2*2 size, the program must be able to used the formulas in requirement 3.3.1.9, to determine the optimal strategies and payoff for the players, using formulas (4.3.3.1.4), (4.3.3.1.5) and (4.3.3.1.6).
- 3.3.3.2 - The application must be able to determine optimal strategies and payoffs for maximising gain and minimising loss, for both of the players.

**PREPARATION:**
Re-use matrices from unit test 4.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Pass the 2*N matrix into the function small matrix player 1 and small matrix player 2.</td>
<td>1. For each 2<em>N matrix entered in by the user, it will be displayed as a combination of 2</em>2 matrices and for each 2*2 matrix, the probabilities for the intersecting lines should be computed(^\text{27}).</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>

\(^{26}\) See Appendix E – Test Data, for manual results.
\(^{27}\) See Appendix E – Test Data, for manual results.
Unit Test Case 6: Performing probability validity check

BRIEF DESCRIPTION:
This test case will check to see that the following requirement;
- 3.3.1.10 should be maintained for 2*N matrices.

PREPARATION:
Reuse the probabilities determined in unit test 5. Alter the code for the function check, by adding print statements, indicating whether the if statement is true or false.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Pass the values (1.0, 0.0) into the check function. 2. Pass the values (-1.0, 2.0) into the check function. 3. Pass the values (-0.25, 1.25) into the check function. 4. Pass the values (0.2, 0.8) into the check function. 5. Pass the values (0.3, 0.65) into the check function.</td>
<td>1. The program should state that the probabilities are valid. 2. The program should state that the probabilities are invalid. 3. The program should state that the probabilities are invalid. 4. The program should state that the probabilities are valid. 5. The program should state that the probabilities are invalid.</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>

Unit Test Case 7: Determining the case of the graphical nature of the matrix

BRIEF DESCRIPTION:
This test case will verify that the program can determine the cases of matrices entered in by the user. This test case will test the variation of requirements;
### Determining Optimal Strategies in Matrix Games

- 3.3.3.1 - The matrix must be able to be represented in a graph.

#### PREPARATION:

Use the matrices specified in Appendix E.

<table>
<thead>
<tr>
<th>ACTION:</th>
<th>EXPECTED RESULT:</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the application, with each of the matrices.</td>
<td>1. For each matrix, the graphical nature should be identified.</td>
<td>PASS</td>
<td></td>
</tr>
</tbody>
</table>
## BRIEF DESCRIPTION:

This test case will test the majority of the requirements specified in Section 3.3.2 and 3.3.3 and all of the specification mentioned in Section 4.3.3.

## PREPARATION:

Have Appendix E to hand.

<table>
<thead>
<tr>
<th>ACTION</th>
<th>EXPECTED RESULT</th>
<th>ACTUAL RESULT (PASS/FAIL)</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the main function passing the matrices from Section E.2 of Appendix E – Test Data.</td>
<td>1. The optimal strategies and payoff that the application generates, should match the optimal strategies and payoff worked out in Section E.2 of Appendix E – Test Data.</td>
<td>FAIL</td>
<td>There is a crash with case 1 for the 2*5 matrix – this is due to 3 lines intersecting at on point. Algorithm for case 2 has got a flaw in it.</td>
</tr>
</tbody>
</table>
Appendix E

Test Data

This Appendix contains the test data for the 2*2 and 2*N Matrix Classes, that was used with Appendix D, during the testing of the applications. The appendix is structured starting with the 2*2 matrices and then 2*N matrices.

The 2*N section is further structured by showing the diagrams of each 2*N matrix, used in the testing and then manually working out the solution for each matrix.
Determining Optimal Strategies in Matrix Games

**E.1. 2*2 Test Data**

Saddle points

A saddle point is an element \( a_{ij} \) in the matrix, where it is both the maximum entry in its column and the minimum entry in its row.

Formula

\[
x = \frac{JA^*/JA^*J^T}{y = A^*J^T / JA^*J^T} = \frac{v = |A|/ JA^*J^T}{|A| = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11} \ a_{22} - a_{12} \ a_{21}}
\]

\[
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{Cofactor (A)} = \begin{pmatrix} +a_{22} & -a_{21} \\ -a_{12} & +a_{11} \end{pmatrix}
\]

\[
\text{Adj (A) or } A^* = (\text{Cofactor (A)})^T = \begin{pmatrix} +a_{22} & -a_{12} \\ -a_{21} & +a_{11} \end{pmatrix}
\]

\[
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{Scalar Value} = X
\]

\[
XA = \begin{pmatrix} xa_{11} & xa_{12} \\ xa_{21} & xa_{22} \end{pmatrix}
\]

\[
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad V = [x, y]
\]

\[
AV = [xa_{11} + xa_{21}, ya_{12} + ya_{22}]
\]

\[
U = (x_1, y_1) \quad V = (x_2, y_2) \quad \text{Dot product of } U \text{ and } V \text{ denoted as } U.V = x_1x_2 + y_1y_2
\]
Determining Optimal Strategies in Matrix Games

E.1.1 User Generated Matrices

Matrix 1 = \[
\begin{pmatrix}
1 & 0 \\
-1 & 2 \\
\end{pmatrix}
\]

No Saddle point

Need to use the formula to work out unique optimal strategies for Players I and II and their corresponding payoff.

\[|A| = (1)2 - 0(-1) = 2 - 0 = 2\]

\[J = (1, 1)\]

\[J^T = \begin{pmatrix}
1 \\
1
\end{pmatrix}\]

\[A^* = \begin{pmatrix}
+2 & -0 \\
-1 & +1
\end{pmatrix} = \begin{pmatrix}
2 & 0 \\
1 & 1
\end{pmatrix}\]

\[JA^* = (1, 1)\begin{pmatrix}
2 & 0 \\
1 & 1
\end{pmatrix} = (1(2) + 1(1), 1(3) + 1(1)) = (2 + 1, 0 + 1) = (3, 1)\]

\[A^*J^T = \begin{pmatrix}
2 & 0 \\
1 & 1
\end{pmatrix}\begin{pmatrix}
1 \\
1
\end{pmatrix} = \begin{pmatrix}
2(1) + 0(1) \\
1(1) + 1(1)
\end{pmatrix} = \begin{pmatrix}
2+0 \\
1+1
\end{pmatrix} = \begin{pmatrix}
2 \\
2
\end{pmatrix}\]

\[JA^*J^T = JA^*(J^T) = (3, 1)\begin{pmatrix}
1 \\
1
\end{pmatrix} = 3(1) + 1(1) = 4\]

Player I = \((3/4, 1/4)\)
Player II = \((1/2, 1/2)\)
Payoff, \(v = 1/2\)

Probabilities for Player I and Player II are both valid.

Matrix 2 = \[
\begin{pmatrix}
-2 & -2 \\
3 & 0
\end{pmatrix}
\]

110
Determining Optimal Strategies in Matrix Games

Saddle point exists. Element $a_{22}$ is a saddle point. Optimal strategy for $a_{22} = 1$ rest of elements, probability is 0. 0 is the payoff.

E.1.2 Randomly Generated Matrices

Matrix 1 = \[
\begin{pmatrix}
93 & 4 \\
7 & 55
\end{pmatrix}
\]

$|A| = (93)55 - 4(7) = 5115 - 28 = 5087$

$J = (1, 1)$

$J^T = \begin{pmatrix}
1 \\
1
\end{pmatrix}$

$A^* = \begin{pmatrix}
55 & -4 \\
-7 & 93
\end{pmatrix}$

$JA^* = (1, 1) \begin{pmatrix}
55 & -4 \\
-7 & 93
\end{pmatrix} = (1(55) + 1(-7), 1(-4) + 1(93)) = (48, 89)$

$A^*J^T = \begin{pmatrix}
55 & -4 \\
-7 & 93
\end{pmatrix} \begin{pmatrix}
1 \\
1
\end{pmatrix} = \frac{1(55) + 1(-4)}{1(-7) + 1(93)} = \frac{51}{85}$

$JA^*J^T = JA^*(J^T)$

$= (48, 89) \begin{pmatrix}
1 \\
1
\end{pmatrix} = 48(\frac{1}{1}) + 89(\frac{1}{1}) = 137$

Player I = (48/137, 89/137) = (0.3503, 0.6496)
Player II = (51/137, 86/137) = (0.3722, 0.6277)
Payoff, $v = 5087/137 = 37.12128$

Probabilities for Player I and Player II are both valid.

Matrix 2 = \[
\begin{pmatrix}
89 & 28 \\
30 & 13
\end{pmatrix}
\]
Determining Optimal Strategies in Matrix Games

Saddle point exists. Element $a_{12}$ is a saddle point. Optimal strategy for $a_{12} = 1$ rest of elements, probability is $0.28$ is the payoff.

**E.2. 2*N Test Data**

Equation of lines

\[ Y = (a_{2j} - a_{1j}) x + a_{1j} \]

Probabilities of players and payoffs

\[ x = \frac{d - c}{a + d - b - c} \]
\[ y = \frac{d - b}{a + d - b - c} \]
\[ v = \frac{ad - bc}{a + d - b - c} \]

Optimal strategies and payoffs in Matrix Games

Player 1 = (x, 1-x)
Player 2 = (y, 1-y)
Payoff = v

Values used to work out whether the intersections determined from the matrices lye on a particular line;

(((1-x), v)
E.2.1 Graphical Representations of Matrices

**E.2.1.1 2*3 Matrices**

\[
\begin{pmatrix}
2 & 0 & 4 \\
0 & 5 & -1 \\
\end{pmatrix}
\]

**Figure (a): 2*3 Matrix – Case 1 – Unique Vertex**

\[
\begin{pmatrix}
-1 & 3 & 1 \\
5 & 1 & 3 \\
\end{pmatrix}
\]

**Figure (b): 2*3 Matrix – Case 2 – Single Vertex**
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
1 & 3 & 5 \\
6 & 3 & 2 \\
\end{pmatrix}
\]

**E.2.1.2 2*4 Matrices**

\[
\begin{pmatrix}
2 & 3 & 1 & 5 \\
4 & 1 & 6 & 0 \\
\end{pmatrix}
\]

![Figure (a): 2*4 Matrix – Case 1 – Unique Vertex](image)

![Figure (c): 2*3 Matrix – Case 3 – Two Vertices](image)
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
0 & 2 & 4 & 6 \\
6 & 4 & 2 & 0 \\
\end{pmatrix}
\]

Figure (b): 2×4 Matrix – Case 2 – Single Vertex

\[
\begin{pmatrix}
1 & 2 & 4 & 5 \\
4 & 2 & 2 & 0 \\
\end{pmatrix}
\]

Figure (c): 2×4 Matrix – Case 3 – Two Vertices
Determining Optimal Strategies in Matrix Games

E.2.1.3  2*5 Matrices

\[
\begin{pmatrix}
1 & 3 & 5 & 2 & 4 \\
7 & 4 & 1 & 5 & 7
\end{pmatrix}
\]

Figure (a): 2*5 Matrix – Case 1 – Unique Vertex

\[
\begin{pmatrix}
0 & 1 & 2 & 4 & 7 \\
7 & 6 & 5 & 3 & 0
\end{pmatrix}
\]

Figure (b): 2*5 Matrix – Case 2 – Single Vertex
Determining Optimal Strategies in Matrix Games

$$\begin{pmatrix}
6 & 3 & 2 & 4 & 5 \\
4 & 3 & 5 & 6 & 1
\end{pmatrix}$$

Figure (c): 2*5 Matrix – Case 3 – Two Vertices
Determining Optimal Strategies in Matrix Games

E.2.2 2*3 Matrices Test Data

E.2.2.1 Case 1: One high vertex

\[
\begin{pmatrix}
2 & 0 & 4 \\
0 & 5 & -1
\end{pmatrix}
\]

Equation of lines:

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Equation of Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( y = -2x + 2 )</td>
</tr>
<tr>
<td>1</td>
<td>( y = 5x )</td>
</tr>
<tr>
<td>2</td>
<td>( y = -5x + 4 )</td>
</tr>
</tbody>
</table>

2*3 Matrices \(\rightarrow\) 2*2 Matrices

Number of matrices = 3

\[
\begin{pmatrix}
2 & 0 \\
0 & 5
\end{pmatrix}
\]

\[
X = \frac{(5-0)}{(2+5-0-0)} = \frac{5}{7} = 0.71428
\]
\[
Y = \frac{(5-0)}{(2+5-0-0)} = \frac{5}{7} = 0.71428
\]
\[
V = \frac{((2*5) - (0*0))}{(2+5-0-0)} = \frac{10}{7} = 1.42857
\]

Player 1 = (5/7, 2/7)  \( \pm 2/7 = \pm 0.28571 \)

Player 2 = (5/7, 2/7)

\( V = 10/7 \)

\[
\begin{pmatrix}
2 & 4 \\
0 & -1
\end{pmatrix}
\]

\[
X = \frac{(-1-0)}{(2+(-1)-4-0)} = \frac{-1}{-3} = 0.333333
\]
\[
Y = \frac{(-1-4)}{(2+(-1)-4-0)} = \frac{-5}{-3} = 1.66666
\]
\[
V = \frac{((2*(-1)) - (4*0))}{(2+(-1)-4-0)} = \frac{-2}{-3} = 0.66666
\]

Player 1 = (1/3, 2/3)  \( \pm 2/3 = \pm 0.66666 \)

Player 2 = (5/3, -2/3)

\( V = 2/3 \)
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
0 & 4 \\
5 & -1
\end{pmatrix}
\]

- \( X = (-1-5) / (0+ (-1)-4-5) = -6/-10 = 0.6 \\
- \( Y = (-1-4) / (0+(-1)-4-5) = -5/-10 = 0.5 \\
- \( V = ((0(-1)) - (5(4))) / (0+(-1)-4-5) = -20/-10 = 2 \\

Player 1 = (0.6, 0.4) \\
Player 2 = (0.5, 0.5) \\

\( V = 2 \)

Points of intersection (taking the 1-prob for Player 1)

- Intersection 1: \((2/7, 10/7)\)
- Intersection 2: \((2/3, 2/3)\)
- Intersection 3: \((0.4, 2)\)

Line 1: \( Y = -2x + 2 \)

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Calculation</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 1</td>
<td>(10/7 = -2(2/7) + 2 = 10/7)</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>Intersection 2</td>
<td>(2/3 = -2(2/3) + 2 = 2/3)</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>Intersection 3</td>
<td>(2 = -2(0.4) + 2 = 1.2)</td>
<td>point not on line</td>
<td></td>
</tr>
<tr>
<td>Lowest intersection</td>
<td></td>
<td></td>
<td><strong>intersection 2</strong></td>
</tr>
</tbody>
</table>

Line 2: \( Y = 5x \)

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Calculation</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 1</td>
<td>(10/7 = 5(2/7) = 10/7)</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>Intersection 2</td>
<td>(2/3 = 5(2/3) = 10/3)</td>
<td>point not on line</td>
<td></td>
</tr>
<tr>
<td>Intersection 3</td>
<td>(2 = 5(0.4) = 2)</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>Lowest intersection</td>
<td></td>
<td></td>
<td><strong>intersection 1</strong></td>
</tr>
</tbody>
</table>

Line 3: \( Y = -5x + 4 \)

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Calculation</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 1</td>
<td>(10/7 = -5(2/7) + 4 = 18/7)</td>
<td>point not on line</td>
<td></td>
</tr>
<tr>
<td>Intersection 2</td>
<td>(2/3 = -5(2/3) + 4 = 2/3)</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>Intersection 3</td>
<td>(2 = -5(0.4) + 4 = 2)</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>Lowest intersection</td>
<td></td>
<td></td>
<td><strong>intersection 2</strong></td>
</tr>
</tbody>
</table>

Low intersections left after removing repeated low intersections

- Intersection 1: \((2/7, 10/7)\)
- Intersection 2: \((2/3, 2/3)\)

Equation of the 2 lowest lines, to the horizontal axis;
Determining Optimal Strategies in Matrix Games

Line 2: \( Y = 5x \)

Line 3: \( Y = -5x + 4 \)

Re-doing points of intersections

Line 2: \( Y = 5x \)

| Intersection 1 | \( \frac{10}{7} = 5(\frac{2}{7}) = \frac{10}{7} \) \( \rightarrow \) point on line |
| Intersection 2 | \( \frac{2}{3} = 5(\frac{2}{3}) = \frac{10}{3} \) \( \rightarrow \) point not on line |

Line 3: \( Y = -5x + 4 \)

| Intersection 1 | \( \frac{10}{7} = -5(\frac{2}{7}) + 4 = \frac{18}{7} \) \( \rightarrow \) point not on line |
| Intersection 2 | \( \frac{2}{3} = -5(\frac{2}{3}) + 4 = \frac{2}{3} \) \( \rightarrow \) point on line |

Two remaining intersections

Intersection 1: \( (\frac{2}{7}, \frac{10}{7}) \)
Intersection 2: \( (\frac{2}{3}, \frac{2}{3}) \)

Choosing the intersection that result in the highest payoff;

Intersection 1: \( (\frac{2}{7}, \frac{10}{7}) \)

Optimal strategies and payoffs;

Player 1 = \( (\frac{5}{7}, \frac{2}{7}) \) \( \frac{5}{7} = 0.71428 \)
Player 2 = \( (\frac{5}{7}, \frac{2}{7}) \) \( \frac{2}{7} = 0.28571 \)
\( V = \frac{10}{7} \) \( \frac{10}{7} = 1.42857 \)
Determining Optimal Strategies in Matrix Games

**E.2.2.2**  
*Case 2: All intersections result in one unique vertex*

\[
\begin{pmatrix}
-1 & 3 & 1 \\
5 & 1 & 3
\end{pmatrix}
\]

2*3 Matrices → 2*2 Matrices

\[
\begin{pmatrix}
-1 & 3 \\
5 & 1
\end{pmatrix}
\]

\[
X = (1-5) / (-1+1-3-5) = -4/-8 = 0.5
\]
\[
Y = (1-3) / (-1+1-3-5) = -2/-8 = 0.25
\]
\[
V = ((-1(1)) - (3(5))) / (-1+1-3-5) = -16/-8 = 2
\]
Player 1 = (0.5, 0.5)
Player 2 = (0.25, 0.75)
V = 2

\[
\begin{pmatrix}
-1 & 1 \\
5 & 3
\end{pmatrix}
\]

\[
X = (3-5) / (-1+3-5-1) = -2/-4 = 0.5
\]
\[
Y = (3-1) / (-1+3-5-1) = 2/-4 = -0.5
\]
\[
V = ((-1(3)) - (1(5))) / (-1+3-5-1) = -8/-4 = 2
\]
Player 1 = (1/2, 1/2)
Player 2 = (-1/2, 3/2)
V = 2

\[
\begin{pmatrix}
3 & 1 \\
1 & 3
\end{pmatrix}
\]

\[
X = (3-1) / (3+3-1-1) = 2/4 = 0.5
\]
\[
Y = (3-1) / (3+3-1-1) = 2/4 = 0.5
\]
\[
V = ((3(3)) - (1(1))) / (3+3-1-1) = 8/4 = 2
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/2, 1/2)
V = 2

All intersecting points result in the same payoff.

Optimal strategies and payoffs;
Player 1 = (0.5, 0.5)
Player 2 = (0.25, 0.75)
V = 2
Determining Optimal Strategies in Matrix Games

**E.2.2.3  Case 3: Two probabilities resulting in the same payoff**

\[
\begin{pmatrix}
1 & 3 & 5 \\
6 & 3 & 2
\end{pmatrix}
\]

2*3 Matrices → 2*2 Matrices

\[
\begin{pmatrix}
1 & 5 \\
6 & 2
\end{pmatrix}
\]

\[
X = (2-6) / (1+2-5-6) = 4/-8 = 0.5 \\
Y = (2-5) / (1+2-5-6) = 3/-8 = 0.375 \\
V = ((1(2)) - (5(6))) / (1+2-5-6) = -28/-8 = 3.5 \\
Player 1 = (0.5, 0.5) \\
Player 2 = (0.375, 0.625) \\
V = 3.5
\]

\[
\begin{pmatrix}
1 & 3 \\
6 & 3
\end{pmatrix}
\]

\[
X = (3-6) / (1+3-3-6) = 3/-5 = 0.6 \\
Y = (3-3) / (1+3-3-6) = 0/-5 = 0 \\
V = ((1(3)) - (3(6))) / (1+3-3-6) = -15/-5 = 3 \\
Player 1 = (0.6, 0.4) \\
Player 2 = (0, 1) \\
V = 3
\]

\[
\begin{pmatrix}
3 & 5 \\
3 & 2
\end{pmatrix}
\]

\[
X = (2-3) / (3+2-5-3) = 1/-3 = 1/3 \\
Y = (2-5) / (3+2-5-3) = -3/-3 = 1 \\
V = ((3(2)) - (5(3))) / (3+2-5-3) = -9/-3 = 3 \\
Player 1 = (1/3, 2/3) \\
Player 2 = (1, 0) \\
V = 3
\]

Two intersecting points generate the same payoffs. Optimal strategies and payoffs between;

Player 1 = (1/3, 2/3) and (0.6, 0.4)

Player 2 = (1, 0) and (0, 1)

V = 3
Determining Optimal Strategies in Matrix Games

E.2.3 2x4 Matrices Test Data

E.2.3.1 Case 1 – one high vertex

\[
\begin{pmatrix}
2 & 3 & 1 & 5 \\
4 & 1 & 6 & 0
\end{pmatrix}
\]

Equation of lines:

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Equation of Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( Y = 2x + 2 )</td>
</tr>
<tr>
<td>1</td>
<td>( Y = -2x + 3 )</td>
</tr>
<tr>
<td>2</td>
<td>( Y = 5x + 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( Y = -5x + 5 )</td>
</tr>
</tbody>
</table>

2x4 Matrices \(\rightarrow\) 2x2 Matrices

Number of matrices = 6

\[
\begin{pmatrix}
2 & 3 \\
4 & 1
\end{pmatrix}
\]

\[
X = (1-4) / (2+1-3-4) = -3/-4 = 3/4 \\
Y = (1-3) / (2+1-3-4) = -2/-4 = 1/2 \\
V = ((2(1)) - (3(4))) / (2+1-3-4) = -10/-4 = 2.5
\]

Player 1 = (3/4, 1/4) \\
Player 2 = (1/2, 1/2) \\
V = 2.5

\[
\begin{pmatrix}
2 & 1 \\
4 & 6
\end{pmatrix}
\]

\[
X = (6-4) / (2+6-1-4) = 2/3 \\
Y = (6-1) / (2+6-1-4) = 5/3 \\
V = ((2(6)) - (1(4))) / (2+6-1-4) = 8/3
\]

Player 1 = (2/3, 1/3) \\
Player 2 = (5/3, -2/3) \\
V = 8/3

\[
\begin{pmatrix}
2 & 5 \\
4 & 0
\end{pmatrix}
\]

\[
X = (0-4) / (2+0-5-4) = -4/-7 = 4/7 \\
Y = (0-5) / (2+0-5-4) = -5/-7 = 5/7 \\
V = ((2(0)) - (5(4))) / (2+0-5-3) = -20/-7 = 20/7
\]

Player 1 = (4/7, 3/7) \\
Player 2 = (5/7, 2/7) \\
V = 20/7
Determining Optimal Strategies in Matrix Games

\[
\begin{bmatrix}
3 & 1 \\
1 & 6
\end{bmatrix}
\]

\[
X = (6-1) / (3+6-1-1) = 5/7 \\
Y = (6-1) / (3+6-1-1) = 5/7 \\
V = ((3(6)) - (1(1))) / (3+6-1-1) = 17/7 = 2.42857 \\
\text{Player 1} = (5/7, 2/7) \\
\text{Player 2} = (5/7, 2/7) \\
V = 17/7 \\
\]

\[
\begin{bmatrix}
3 & 5 \\
1 & 0
\end{bmatrix}
\]

\[
X = (0-1) / (3+0-1-5) = -1/-3 = 1/3 \\
Y = (0-5) / (3+0-1-5) = -5/-3 = 5/3 \\
V = ((3(0)) - (5(1))) / (3+0-1-5) = -5/-3 = 5/3 \\
\text{Player 1} = (1/3, 2/3) \\
\text{Player 2} = (5/3, -2/3) \\
V = 5/3 = 1.66666 \\
\]

\[
\begin{bmatrix}
1 & 5 \\
6 & 0
\end{bmatrix}
\]

\[
X = (0-6) / (1+0-5-6) = -6/-10 = 6/10 = 0.6 \\
Y = (0-5) / (1+0-5-6) = -5/-10 = 1/2 \\
V = ((1(0)) - (5(6))) / (1+0-5-6) = -30/-10 = 3 \\
\text{Player 1} = (6/10, 4/10) \\
\text{Player 2} = (1/2, 1/2) \\
V = 3 \\
\]

Points of intersection (taking the 1-prob for Player 1)
Intersection 1: (1/4, 2.5)
Intersection 2: (1/3, 8/3)
Intersection 3: (3/7, 20/7)
Intersection 4: (2/7, 17/7)
Intersection 5: (2/3, 5/3)
Intersection 6: (4/10, 3)

Line 1: \(Y = Y = 2x + 2\)

<table>
<thead>
<tr>
<th>Intersection 1</th>
<th>2.5 = 2(1/4) + 2 = 2.5 (\Rightarrow) point on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>8/3 = 2(1/3) + 2 = 8/3 (\Rightarrow) point on line</td>
</tr>
<tr>
<td>Intersection 3</td>
<td>20/7 = 2(3/7) + 2 = 20/7 (\Rightarrow) point on line</td>
</tr>
<tr>
<td>Intersection 4</td>
<td>17/7 = 2(2/7) + 2 = 18/7 (\Rightarrow) point not on line</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>Intersection 5</th>
<th>5/3 = 2(2/7) + 2 = 10/3  →  point not on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 6</td>
<td>3 = 2(4/10) + 2 = 28/10  →  point not on line</td>
</tr>
<tr>
<td>Lowest intersection = intersection 1</td>
<td></td>
</tr>
</tbody>
</table>

Line 2: \( Y = -2x + 3 \)

<table>
<thead>
<tr>
<th>Intersection 1</th>
<th>2.5 = -2(1/4) + 3 = 2.5  →  point on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>8/3 = -2(1/3) + 3 = 7/3  →  point not on line</td>
</tr>
<tr>
<td>Intersection 3</td>
<td>20/7 = -2(3/7) + 3 = 15/7  →  point not on line</td>
</tr>
<tr>
<td>Intersection 4</td>
<td>17/7 = -2(2/7) + 3 = 17/7  →  point on line</td>
</tr>
<tr>
<td>Intersection 5</td>
<td>5/3 = -2(2/3) + 3 = 5/3  →  point on line</td>
</tr>
<tr>
<td>Intersection 6</td>
<td>3 = -2(4/10) + 3 = 22/10  →  point not on line</td>
</tr>
<tr>
<td>Lowest intersection = intersection 5</td>
<td></td>
</tr>
</tbody>
</table>

Line 3: \( Y = 5x + 1 \)

<table>
<thead>
<tr>
<th>Intersection 1</th>
<th>2.5 = 5(1/4) + 1 = 2.25  →  point not on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>8/3 = 5(1/3) + 1 = 8/3  →  point on line</td>
</tr>
<tr>
<td>Intersection 3</td>
<td>20/7 = 5(3/7) + 1 = 22/7  →  point not on line</td>
</tr>
<tr>
<td>Intersection 4</td>
<td>17/7 = 5(2/7) + 1 = 17/7  →  point on line</td>
</tr>
<tr>
<td>Intersection 5</td>
<td>5/3 = 5(2/3) + 1 = 13/3  →  point not on line</td>
</tr>
<tr>
<td>Intersection 6</td>
<td>3 = 5(4/10) + 1 = 3  →  point on line</td>
</tr>
<tr>
<td>Lowest intersection = intersection 4</td>
<td></td>
</tr>
</tbody>
</table>

Line 4: \( Y = -5x + 5 \)

<table>
<thead>
<tr>
<th>Intersection 1</th>
<th>2.5 = -5(1/4) + 5 = 3.75  →  point not on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>8/3 = -5(1/3) + 5 = 10/3  →  point not on line</td>
</tr>
<tr>
<td>Intersection 3</td>
<td>20/7 = -5(3/7) + 5 = 20/7  →  point on line</td>
</tr>
<tr>
<td>Intersection 4</td>
<td>17/7 = -5(2/7) + 5 = 25/7  →  point not on line</td>
</tr>
<tr>
<td>Intersection 5</td>
<td>5/3 = -5(2/3) + 5 = 5/3  →  point on line</td>
</tr>
<tr>
<td>Intersection 6</td>
<td>3 = -5(4/10) + 5 = 3  →  point on line</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

**Lowest intersection = intersection 5**

Removing repeated low intersections

Intersection 1:  \( (1/4, 2.5) \)
Intersection 4:  \( (2/7, 17/7) \)
Intersection 5:  \( (2/3, 5/3) \)

Equation of the 2 lowest lines, to the horizontal axis

Line 3: \( Y = 5x + 1 \)
Line 4: \( Y = -5x + 5 \)

Re-doing points of intersections

Line 3: \( Y = 5x + 1 \)

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Equation</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5 = 5(1/4) + 1 = 2.25</td>
<td>point not on line</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17/7 = 5(2/7) + 1 = 17/7</td>
<td>point on line</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5/3 = 5(2/3) + 1 = 13/3</td>
<td>point not on line</td>
<td></td>
</tr>
</tbody>
</table>

Line 4: \( Y = -5x + 5 \)

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Equation</th>
<th>Result</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5 = -5(1/4) + 5 = 3.75</td>
<td>point not on line</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17/7 = -5(2/7) +5 = 25/7</td>
<td>point not on line</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5/3 = -5(2/3) +5 = 5/3</td>
<td>point on line</td>
<td></td>
</tr>
</tbody>
</table>

Two remaining intersections

Intersection 4:  \( (2/7, 17/7) \)
Intersection 5:  \( (2/3, 5/3) \)

Choosing the intersection that result in the highest payoff;

Intersection 4:  \( (2/7, 17/7) \)

Optimal strategies and payoffs;

Player 1 = \((5/7, 2/7)\)  \( 5/7 = 0.71428 \)
Player 2 = \((5/7, 2/7)\)  \( 2/7 = 0.28571 \)
\( V = 17/7 \)  \( 17/7 = 2.42857 \)
Determining Optimal Strategies in Matrix Games

**E.2.3.2 Case 2 – All intersections result in one unique vertex**

\[
\begin{pmatrix}
0 & 2 & 4 & 6 \\
6 & 4 & 2 & 0
\end{pmatrix}
\]

2*4 Matrices $\rightarrow$ 2*2 Matrices
Number of matrices = 6

\[
\begin{pmatrix}
0 & 2 \\
6 & 4
\end{pmatrix}
\]
\[
X = (4-6) / (0+4-2-6) = -2/-4 = 1/2 \\
Y = (4-2) / (0+4-2-6) = 2/-4 = -1/2 \\
V = ((0(4)) - (2(6))) / (0+4-2-6) = -12/-4 = 3
\]
Player 1 = (1/2, 1/2)
Player 2 = (-1/2, 3/2)
V = 3

\[
\begin{pmatrix}
0 & 4 \\
6 & 2
\end{pmatrix}
\]
\[
X = (2-6) / (0+2-4-6) = -4/-8 = 1/2 \\
Y = (2-4) / (0+2-4-6) = -2/-8 = 1/4 \\
V = ((0(2)) - (4(6))) / (0+2-4-6) = -24/-8 = 3
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/4, 3/4)
V = 3

\[
\begin{pmatrix}
0 & 6 \\
6 & 0
\end{pmatrix}
\]
\[
X = (0-6) / (0+0-6-6) = -6/-12 = 1/2 \\
Y = (0-6) / (0+0-6-6) = -6/-12 = 1/2 \\
V = ((0(0)) - (6(6))) / (0+0-6-6) = -36/-12 = 3
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/2, 1/2)
V = 3
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix} 2 & 4 \\ 4 & 2 \end{pmatrix}
\]

\(X = \frac{2-4}{2-4} = \frac{-2}{-4} = \frac{1}{2}\)

\(Y = \frac{2-4}{2-4} = \frac{-2}{-4} = \frac{1}{2}\)

\(V = \frac{(2(2)) - (4(4))}{2+2-4-4} = -12/-4 = 3\)

Player 1 = (1/2, 1/2)
Player 2 = (1/2, 1/2)

\(V = 3\)

\[
\begin{pmatrix} 2 & 6 \\ 4 & 0 \end{pmatrix}
\]

\(X = \frac{0-4}{2+6-4} = \frac{-4}{-8} = \frac{1}{2}\)

\(Y = \frac{0-6}{2+0-4} = \frac{-6}{-8} = \frac{3}{4}\)

\(V = \frac{(2(0)) - (6(4))}{2+0-6} = -24/-8 = 3\)

Player 1 = (1/2, 1/2)
Player 2 = (3/4, 1/4)

\(V = 3\)

\[
\begin{pmatrix} 4 & 6 \\ 2 & 0 \end{pmatrix}
\]

\(X = \frac{0-2}{4+0-6} = \frac{-2}{-4} = \frac{1}{2}\)

\(Y = \frac{0-6}{4+0-6} = \frac{-6}{-4} = \frac{3}{2}\)

\(V = \frac{(4(0)) - (6(2))}{4+0-6} = -12/-4 = 3\)

Player 1 = (1/2, 1/2)
Player 2 = (3/2, -1/2)

\(V = 3\)

All intersecting points result in the same payoff.

Optimal strategies and payoffs;
Player 1 = (1/2, 1/2)
Player 2 = (1/4, 3/4)

\(V = 3\)
Determining Optimal Strategies in Matrix Games

E.2.3.3  Case 3 – Two probabilities resulting in the same payoff

\[
\begin{pmatrix}
1 & 2 & 4 & 5 \\
4 & 2 & 2 & 0
\end{pmatrix}
\]

2*4 Matrices → 2*2 Matrices  
Number of matrices = 6

\[
\begin{pmatrix}
1 & 2 \\
4 & 2
\end{pmatrix}
\]

X = (2-4) / (1+2-2-4) = -2/-3 = 2/3  
Y = (2-2) / (1+2-2-4) = 0/-3 = 0  
V = ((1(2)) - (2(4))) / (1+2-2-4) = -6/-3 = 2  
Player 1 = (2/3, 1/3)  
Player 2 = (0, 1)  
V = 2

\[
\begin{pmatrix}
1 & 4 \\
4 & 2
\end{pmatrix}
\]

X = (2-4) / (1+2-4-4) = -2/-5 = 2/5  
Y = (2-4) / (1+2-4-4) = -2/-5 = 2/5  
V = ((1(2)) - (4(4))) / (1+2-4-4) = -14/-5 = 14/5  
Player 1 = (2/5, 3/5)  
Player 2 = (2/5, 3/5)  
V = 14/5 = 2.8

\[
\begin{pmatrix}
1 & 5 \\
4 & 0
\end{pmatrix}
\]

X = (0-4) / (1+0-5-4) = -4/-8 = 1/2  
Y = (0-5) / (1+0-5-4) = -5/-8 = 5/8  
V = ((1(0)) - (5(4))) / (1+0-5-4) = -20/-8 = 5/2  
Player 1 = (1/2, 1/2)  
Player 2 = (5/8, 3/8)  
\[5/8 = 0.625, \ 3/8 = 0.375\]  
V = 5/2 = 2.5
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
  2 & 4 \\
  2 & 2 \\
\end{pmatrix}
\]

\[
X = \frac{(2-2)}{(2+2-2-4)} = \frac{0}{-2} = 0
\]

\[
Y = \frac{(2-4)}{(2+2-2-4)} = \frac{-2}{-2} = 1
\]

\[
V = \frac{(2(2)) - (4(2)))}{(2+2-2-4)} = \frac{-4}{-2} = 2
\]

Player 1 = (0, 1)
Player 2 = (1, 0)
V = 2

\[
\begin{pmatrix}
  2 & 5 \\
  2 & 0 \\
\end{pmatrix}
\]

\[
X = \frac{(0-2)}{(2+0-5-2)} = \frac{-2}{-5} = 2/5
\]

\[
Y = \frac{(0-5)}{(2+0-5-2)} = \frac{-5}{-5} = 1
\]

\[
V = \frac{(2(0)) - (5(2)))}{(2+0-5-2)} = \frac{-10}{-5} = 2
\]

Player 1 = (2/5, 3/5)
Player 2 = (1, 0)
V = 2

\[
\begin{pmatrix}
  4 & 5 \\
  2 & 0 \\
\end{pmatrix}
\]

\[
X = \frac{(0-2)}{(4+0-5-2)} = \frac{-2}{-3} = 2/3
\]

\[
Y = \frac{(0-5)}{(4+0-5-2)} = \frac{-5}{-3} = 5/3
\]

\[
V = \frac{(4(0)) - (5(2)))}{(4+0-5-2)} = \frac{-10}{-3} = 10/3
\]

Player 1 = (2/3, 1/3)
Player 2 = (5/3, -2/3)
V = 10/3

Two intersecting points generate the same payoffs.
Optimal strategies and payoffs between;
Player 1 = (2/3, 1/3) and (2/5, 3/5)
Player 2 = (0, 1) and (1, 0)
V = 2
Determining Optimal Strategies in Matrix Games

E.2.4 2*5 Matrices Test Data

E.2.4.1 Case 1 – one high vertex

\[
\begin{pmatrix}
1 & 3 & 5 & 2 & 4 \\
7 & 4 & 1 & 5 & 7
\end{pmatrix}
\]

Equation of lines:

<table>
<thead>
<tr>
<th>Column Number</th>
<th>Equation of Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( Y = 6x + 1 )</td>
</tr>
<tr>
<td>1</td>
<td>( Y = x + 3 )</td>
</tr>
<tr>
<td>2</td>
<td>( Y = -4x + 5 )</td>
</tr>
<tr>
<td>3</td>
<td>( Y = 3x + 2 )</td>
</tr>
<tr>
<td>4</td>
<td>( Y = 3x + 4 )</td>
</tr>
</tbody>
</table>

2*5 Matrices \(\rightarrow\) 2*2 Matrices

Number of matrices = 10

\[
\begin{pmatrix}
1 & 3 \\
7 & 4
\end{pmatrix}
\]

\[
X = (4-7) / (1+4-3-7) = -3/-5 = 3/5 \\
Y = (4-3) / (1+4-3-7) = 1/-5 = -1/5 \\
V = ((1(4)) - (3(7))) / (1+4-3-7) = -17/-5 = 17/5 \\
Player 1 = (3/5, 2/5) \\
Player 2 = (-1/5, 6/5) \quad 6/5 = 1.2 \\
V = 17/5 = 3.4
\]

\[
\begin{pmatrix}
1 & 5 \\
7 & 1
\end{pmatrix}
\]

\[
X = (1-7) / (1+1-5-7) = -6/-10 = 6/10 \\
Y = (1-5) / (1+1-5-7) = -4/-10 = 4/10 \\
V = ((1(1)) - (5(7))) / (1+1-5-7) = -34/-10 = 34/10 = 3.4 \\
Player 1 = (6/10, 4/10) \\
Player 2 = (4/10, 6/10) \\
V = 3.4
\]

\[
\begin{pmatrix}
1 & 2 \\
7 & 5
\end{pmatrix}
\]

\[
X = (5-7) / (1+5-2-7) = -2/-3 = 2/3 \\
Y = (5-2) / (1+5-2-7) = 3/-3 = -1 \\
V = ((1(5)) - (2(7))) / (1+5-2-7) = -9/-3 = 3 \\
Player 1 = (2/3, 1/3) \\
Player 2 = (-1, 2) \\
V = 3
\]
Determining Optimal Strategies in Matrix Games

\[
\begin{bmatrix}
3 & 5 \\
4 & 1 \\
\end{bmatrix}
\]

\[
X = (1 - 4) / (3 + 1 - 5 - 4) = -3 / -5 = 3 / 5 \\
Y = (1 - 5) / (3 + 1 - 5 - 4) = -4 / -5 = 4 / 5 \\
V = ((3(1)) - (4(5))) / (3 + 1 - 5 - 4) = -17 / -5 = 17 / 5 = 3.4 \\
\text{Player 1} = (3 / 5, 2 / 5) \\
\text{Player 2} = (4 / 5, 1 / 5) \\
V = 17 / 5
\]

\[
\begin{bmatrix}
3 & 2 \\
4 & 5 \\
\end{bmatrix}
\]

\[
X = (5 - 4) / (3 + 5 - 2 - 4) = 1 / 2 \\
Y = (5 - 2) / (3 + 5 - 2 - 4) = 3 / 2 \\
V = ((3(5)) - (2(4))) / (3 + 5 - 2 - 4) = 7 / 2 = 3.5 \\
\text{Player 1} = (1 / 2, 1 / 2) \\
\text{Player 2} = (3 / 2, -1 / 2) \\
V = 3.5
\]

\[
\begin{bmatrix}
3 & 4 \\
4 & 7 \\
\end{bmatrix}
\]

\[
X = (7 - 4) / (3 + 7 - 4 - 4) = 3 / 2 \\
Y = (7 - 4) / (3 + 7 - 4 - 4) = 3 / 2 \\
V = ((3(7)) - (4(4))) / (3 + 7 - 4 - 4) = 5 / 2 = 2.5 \\
\text{Player 1} = (3 / 2, -1 / 2) \\
\text{Player 2} = (3 / 2, -1 / 2) \\
V = 2.5
\]

\[
\begin{bmatrix}
5 & 2 \\
1 & 5 \\
\end{bmatrix}
\]

\[
X = (5 - 1) / (5 + 5 - 2 - 1) = 4 / 7 = 0.57142 \\
Y = (5 - 2) / (5 + 5 - 2 - 1) = 3 / 7 = 0.42857 \\
V = ((5(5)) - (2(1))) / (5 + 5 - 2 - 1) = 23 / 7 = 3.28571 \\
\text{Player 1} = (4 / 7, 3 / 7) \\
\text{Player 2} = (3 / 7, 4 / 7) \\
V = 23 / 7
\]
Determining Optimal Strategies in Matrix Games

\[
\begin{bmatrix}
2 & 4 \\
5 & 7 \\
\end{bmatrix}
\]

\[
X = (7-5) / (2+7-4-5) = 2/0 = \text{infinity}
\]
\[
Y = (7-4) / (2+7-4-5) = 3/0 = \text{infinity}
\]
\[
V = ((2(7)) - (4(5))) / (2+7-4-5) = -3/0 = \text{infinity}
\]

Player 1 = infinity

Player 2 = infinity

V = infinity

\[
\begin{bmatrix}
1 & 4 \\
7 & 7 \\
\end{bmatrix}
\]

\[
X = (7-7) / (1+7-4-7) = 0/-3 = 0
\]
\[
Y = (7-4) / (1+7-4-7) = 3/-3 = -1
\]
\[
V = ((1(7)) - (4(7))) / (1+7-4-7) = -21/-3 = 7
\]

Player 1 = (0, 1)

Player 2 = (-1, 2)

V = 7

\[
\begin{bmatrix}
5 & 4 \\
1 & 7 \\
\end{bmatrix}
\]

\[
X = (7-1) / (5+7-4-1) = 6/7 = 0.85714
\]
\[
Y = (7-4) / (5+7-4-1) = 3/7 = 0.42857
\]
\[
V = ((5(7)) - (4(1))) / (5+7-4-1) = 31/7 = 4.42857
\]

Player 1 = (6/7, 1/7)      1/7 = 0.14285

Player 2 = (3/7, 4/7)      3/7 = 0.42857

V = 31/7

Points of intersection (taking the 1-prob for Player 1)

Intersection 1:  (2/5, 3.4)
Intersection 2:  (4/10, 3.4)
Intersection 3:  (1/3, 3)
Intersection 4:  (1, 7)
Intersection 5:  (2/5, 17/5)
Intersection 6:  (1/2, 3.5)
Intersection 7:  (-1/2, 2.5)
Intersection 8:  (3/7, 23.7)
Intersection 9:  (Null) → due to infinite probabilities
Intersection 10:  (1/7, 31/7)

Line 1:  \(Y = 6x + 1\)

| Intersection 1 | 3.4 = 6(2/5) + 1 = 3.4 → point on line |
## Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Equation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>$3.4 = \frac{6}{4} \times \frac{10}{1} + 1 = 3.4 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 3</td>
<td>$3 = \frac{6}{1} \times \frac{1}{3} + 1 = 3 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 4</td>
<td>$7 = \frac{6}{1} \times 1 + 1 = 7 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 5</td>
<td>$\frac{17}{5} = \frac{6}{2} \times \frac{5}{1} + 1 = \frac{17}{5} \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 6</td>
<td>$3.5 = \frac{6}{1} \times \frac{1}{2} + 1 = 4 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 7</td>
<td>$2.5 = \frac{6}{1} \times \frac{-1}{2} + 1 = -2 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 8</td>
<td>$23/7 = \frac{6}{3} \times \frac{7}{1} + 1 = 25/7 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 9</td>
<td>Null</td>
<td></td>
</tr>
<tr>
<td>Intersection 10</td>
<td>$31/7 = \frac{6}{1} \times \frac{1}{7} + 1 = 13/7 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Lowest intersection</td>
<td>$= \text{intersection 5}$</td>
<td></td>
</tr>
</tbody>
</table>

### Line 2: $Y = x + 3$

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Equation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 1</td>
<td>$3.4 = \frac{2}{5} \times \frac{5}{1} + 1 = 3.4 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 2</td>
<td>$3.4 = \frac{4}{10} \times \frac{10}{1} + 1 = 3.4 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 3</td>
<td>$3 = \frac{1}{3} \times \frac{3}{1} + 1 = 10/3 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 4</td>
<td>$7 = \frac{1}{1} \times \frac{1}{3} + 1 = 4 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 5</td>
<td>$\frac{17}{5} = \frac{2}{5} \times \frac{5}{1} + 1 = \frac{17}{5} \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 6</td>
<td>$3.5 = \frac{1}{2} \times \frac{2}{1} + 1 = 3.5 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 7</td>
<td>$2.5 = \frac{-1}{2} \times \frac{2}{1} + 1 = 2.5 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 8</td>
<td>$23/7 = \frac{3}{7} \times \frac{7}{1} + 1 = 24/7 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 9</td>
<td>Null</td>
<td></td>
</tr>
<tr>
<td>Intersection 10</td>
<td>$31/7 = \frac{1}{1} \times \frac{1}{7} + 1 = 22/7 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Lowest intersection</td>
<td>$= \text{intersection 7}$</td>
<td></td>
</tr>
</tbody>
</table>

### Line 3: $Y = -4x + 5$

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Equation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 1</td>
<td>$3.4 = -\frac{4}{2} \times \frac{5}{1} + 5 = 3.4 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 2</td>
<td>$3.4 = -\frac{4}{4} \times \frac{10}{1} + 5 = 3.4 \rightarrow \text{point on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 3</td>
<td>$3 = -\frac{4}{1} \times \frac{1}{3} + 5 = 11/3 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 4</td>
<td>$7 = -\frac{4}{1} \times 1 + 5 = 2 \rightarrow \text{point not on line}$</td>
<td></td>
</tr>
<tr>
<td>Intersection 5</td>
<td>$\frac{17}{5} = -\frac{4}{2} \times \frac{5}{1} + 5 = \frac{17}{5} \rightarrow \text{point on line}$</td>
<td></td>
</tr>
</tbody>
</table>
## Determining Optimal Strategies in Matrix Games

### Line 4: Y = 3x + 2

<table>
<thead>
<tr>
<th>Intersection 1</th>
<th>3.4 = 3(2/5) + 2 = 3.2 → point on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>3.4 = 3(4/10) + 2 = 3.2 → point on line</td>
</tr>
<tr>
<td>Intersection 3</td>
<td>3 = 3(1/3) + 2 = 3 → point on line</td>
</tr>
<tr>
<td>Intersection 4</td>
<td>7 = 3(1) + 2 = 5 → point not on line</td>
</tr>
<tr>
<td>Intersection 5</td>
<td>17/5 = 3(2/5) + 2 = 16/5 → point not on line</td>
</tr>
<tr>
<td>Intersection 6</td>
<td>3.5 = 3(1/2) + 2 = 3.5 → point on line</td>
</tr>
<tr>
<td>Intersection 7</td>
<td>2.5 = 3(-1/2) + 2 = 0.5 → point not on line</td>
</tr>
<tr>
<td>Intersection 8</td>
<td>23/7 = 3(3/7) + 2 = 23/7 → point on line</td>
</tr>
<tr>
<td>Intersection 9</td>
<td>Null</td>
</tr>
<tr>
<td>Intersection 10</td>
<td>31/7 = 3(1/7) + 2 = 17/7 → point not on line</td>
</tr>
<tr>
<td>Lowest intersection = intersection 3</td>
<td></td>
</tr>
</tbody>
</table>

### Line 5: Y = 3x + 4

<table>
<thead>
<tr>
<th>Intersection 1</th>
<th>3.4 = 3(2/5) + 4 = 5.2 → point not on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection 2</td>
<td>3.4 = 3(4/10) + 4 = 5.2 → point not on line</td>
</tr>
<tr>
<td>Intersection 3</td>
<td>3 = 3(1/3) + 4 = 5 → point not on line</td>
</tr>
<tr>
<td>Intersection 4</td>
<td>7 = 3(1) + 4 = 7 → point on line</td>
</tr>
<tr>
<td>Intersection 5</td>
<td>17/5 = 3(2/5) + 4 = 26/5 → point not on line</td>
</tr>
<tr>
<td>Intersection 6</td>
<td>3.5 = 3(1/2) + 4 = 5.5 → point not on line</td>
</tr>
<tr>
<td>Intersection 7</td>
<td>2.5 = 3(-1/2) + 4 = 2.5 → point on line</td>
</tr>
<tr>
<td>Intersection 8</td>
<td>23/7 = 3(3/7) + 4 = 37/7 → point not on line</td>
</tr>
<tr>
<td>Intersection 9</td>
<td>Null</td>
</tr>
</tbody>
</table>

Lowest intersection = intersection 3
Determining Optimal Strategies in Matrix Games

<table>
<thead>
<tr>
<th>Intersection 10</th>
<th>$31/7 = 3(1/7) + 4 = 31/7 \Rightarrow \text{point on line}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest intersection = <strong>intersection 7</strong></td>
<td></td>
</tr>
</tbody>
</table>

Removing repeated low intersections

Intersection 3: (1/3, 3)
Intersection 5: (2/5, 17/5)
Intersection 7: (-1/2, 2.5)
Intersection 8: (3/7, 23/7)

Equation of the 2 lowest lines, to the horizontal axis

Line 1: $Y = 6x + 1$
Line 3: $Y = -4x + 5$

Re-doing points of intersections

<table>
<thead>
<tr>
<th>Line 1: $Y = 6x + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intersection 3</strong></td>
</tr>
<tr>
<td><strong>Intersection 5</strong></td>
</tr>
<tr>
<td>Intersection 7</td>
</tr>
<tr>
<td>Intersection 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 3: $Y = -4x + 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intersection 3</strong></td>
</tr>
<tr>
<td><strong>Intersection 5</strong></td>
</tr>
<tr>
<td>Intersection 7</td>
</tr>
<tr>
<td><strong>Intersection 8</strong></td>
</tr>
</tbody>
</table>

Choosing the intersection that result in the highest payoff;

Intersection 5: (2/5, 17/5)

Optimal strategies and payoffs;

Player 1 = (3/5, 2/5)
Player 2 = (4/5, 1/5)

$V = 17/5 = 3.4$
Determining Optimal Strategies in Matrix Games

**E.2.4.2 Case 2 – All intersections result in one unique vertex**

\[
\begin{pmatrix}
0 & 1 & 2 & 4 & 7 \\
7 & 6 & 5 & 3 & 0
\end{pmatrix}
\]

2*5 Matrices → 2*2 Matrices
Number of matrices = 10

\[
\begin{pmatrix}
0 & 1 \\
7 & 6
\end{pmatrix}
\]

- X = (6-7) / (0+6-7-1) = -1/-2 = 1/2
- Y = (6-1) / (0+6-7-1) = -5/-2 = 5/2 = 2.5
- V = ((0(6)) - (1(7))) / (0+6-7-1) = -7/-2 = 7/2 = 3.5
- Player 1 = (1/2, 1/2)
- Player 2 = (5/2, -3/2)
- V = 3.5

\[
\begin{pmatrix}
0 & 2 \\
7 & 5
\end{pmatrix}
\]

- X = (5-7) / (0+5-2-7) = -2/-4 = 1/2
- Y = (5-2) / (0+5-2-7) = 3/-4 = -3/4
- V = ((0(5)) - (2(7))) / (0+5-2-7) = -14/-4 = 7/2 = 3.5
- Player 1 = (1/2, 1/2)
- Player 2 = (-3/4, 7/4) 7/4 = 1.75
- V = 3.5

\[
\begin{pmatrix}
0 & 4 \\
7 & 3
\end{pmatrix}
\]

- X = (3-7) / (0+3-4-7) = -4/-8 = 1/2
- Y = (3-4) / (0+3-4-7) = -1/-8 = 1/8
- V = ((0(3)) - (4(7))) / (0+3-4-7) = -28/-8 = 3.5
- Player 1 = (1/2, 1/2)
- Player 2 = (1/8, 7/8) 1/8 = 0.125 7/8 = 0.875
- V = 3.5
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
0 & 7 \\
7 & 0 \\
\end{pmatrix}
\]

\[
X = (0-7) / (0+0-7-7) = -7/-14 = 1/2
\]
\[
Y = (0-7) / (0+0-7-7) = -7/-14 = 1/2
\]
\[
V = ((0(0)) - (7(7))) / (0+0-7-7) = -49/-14 = 7/2 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/2, 1/2)
V = 3.5

\[
\begin{pmatrix}
1 & 2 \\
6 & 5 \\
\end{pmatrix}
\]

\[
X = (5-6) / (5+1-2-6) = -1/-2 = 1/2
\]
\[
Y = (5-2) / (5+1-2-6) = 3/-2 = -3/2
\]
\[
V = ((1(5)) - (2(6))) / (5+1-2-6) = -7/-2 = 7/2 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (-3/2, 5/2)
V = 3.5

\[
\begin{pmatrix}
1 & 4 \\
6 & 3 \\
\end{pmatrix}
\]

\[
X = (3-6) / (1+3-4-6) = -3/-6 = 1/2
\]
\[
Y = (3-4) / (1+3-4-6) = -1/-6 = 1/6 = 0.16666
\]
\[
V = ((1(3)) - (4(6))) / (1+3-4-6) = -21/-6 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/6, 5/6) 5/6 = 0.83333
V = 3.5

\[
\begin{pmatrix}
1 & 7 \\
6 & 0 \\
\end{pmatrix}
\]

\[
X = (0-6) / (1+0-7-6) = -6/-12 = 1/2
\]
\[
Y = (0-7) / (1+0-7-6) = 7/-12 = 7/12 = 0.58333
\]
\[
V = ((1(0)) - (7(6))) / (1+0-7-6) = -42/-12 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (7/12, 5/12) 5/12 = 0.41666
V = 3.5
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
2 & 4 \\
5 & 3
\end{pmatrix}
\]

\[
X = (3-5) / (2+3-4-5) = -2/-4 = 1/2
\]
\[
Y = (3-4) / (2+3-4-5) = -1/-4 = 1/4
\]
\[
V = ((2(3)) - (4(5))) / (2+3-4-5) = -14/-4 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/4, 3/4)
V = 3.5

\[
\begin{pmatrix}
4 & 7 \\
3 & 0
\end{pmatrix}
\]

\[
X = (0-3) / (4+0-7-3) = -3/-6 = 1/2
\]
\[
Y = (0-7) / (4+0-7-3) = -7/-6 = 7/6 = 1.16666
\]
\[
V = ((4(0)) - (3(7))) / (4+0-7-3) = -21/-6 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (7/6, -1/6)
V = 3.5

\[
\begin{pmatrix}
2 & 7 \\
5 & 0
\end{pmatrix}
\]

\[
X = (0-5) / (2+0-7-5) = -5/-10 = 1/2
\]
\[
Y = (0-7) / (2+0-7-5) = -7/-10 = 7/10
\]
\[
V = ((2(0)) - (7(5))) / (2+0-7-5) = -35/-10 = 3.5
\]
Player 1 = (1/2, 1/2)
Player 2 = (7/10, 3/10)
V = 3.5

All intersecting points result in the same payoff.

Optimal strategies and payoffs;
Player 1 = (0.5, 0.5)
Player 2 = (1/8, 7/8)
\[
V = 3.5
\]
Determining Optimal Strategies in Matrix Games

E.2.4.3  Case 3 – Two probabilities resulting in the same payoff

\[
\begin{pmatrix}
6 & 3 & 2 & 4 & 5 \\
4 & 3 & 5 & 6 & 1
\end{pmatrix}
\]

2*5 Matrices → 2*2 Matrices
Number of matrices = 10

\[
\begin{pmatrix}
6 & 3 \\
4 & 3
\end{pmatrix}
\]

\[
X = (3-4) \div (6+3-3-4) = -1/2
\]
\[
Y = (3-3) \div (6+3-3-4) = 0/2 = 0
\]
\[
V = ((6(3)) - (3(4))) \div (6+3-3-4) = 6/2 = 3
\]
Player 1 = (-1/2, 3/2)
Player 2 = (0, 1)
\[
V = 3
\]

\[
\begin{pmatrix}
6 & 2 \\
4 & 5
\end{pmatrix}
\]

\[
X = (5-4) \div (6+5-2-4) = 1/5
\]
\[
Y = (5-2) \div (6+5-2-4) = 3/5
\]
\[
V = ((6(5)) - (2(4))) \div (6+5-2-4) = 22/5 = 4.4
\]
Player 1 = (1/5, 4/5)
Player 2 = (3/5, 2/5)
\[
V = 4.4
\]

\[
\begin{pmatrix}
3 & 2 \\
3 & 5
\end{pmatrix}
\]

\[
X = (5-3) \div (3+5-2-3) = 2/3
\]
\[
Y = (5-2) \div (3+5-2-3) = 3/3 = 1
\]
\[
V = ((3(5)) - (2(3))) \div (3+5-2-3) = 9/3 = 3
\]
Player 1 = (2/3, 1/3)
Player 2 = (1, 0)
\[
V = 3
\]

\[
\begin{pmatrix}
3 & 4 \\
3 & 6
\end{pmatrix}
\]

\[
X = (6-3) \div (3+6-4-3) = 3/2
\]
\[
Y = (6-4) \div (3+6-4-3) = 2/2 = 1
\]
\[
V = ((3(6)) - (4(3))) \div (3+6-4-3) = 6/2 = 3
\]
Player 1 = (3/2, -1/2)
Player 2 = (1, 0)
\[
V = 3
\]
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
4 & 5 \\
6 & 1
\end{pmatrix}
\]
\[
X = (1-6) / (4+1-5-6) = -5/-6 = 5/6 = 0.83333
\]
\[
Y = (1-5) / (4+1-5-6) = -4/-6 = 4/6 = 0.66666
\]
\[
V = ((4(1)) - (5(6))) / (4+1-5-6) = -26/-6 = 4.33333 = 13/3
\]
Player 1 = (5/6, 1/6)  
1/6 = 0.16666
Player 2 = (4/6, 2/6)  
2/6 = 0.33333
V = 13/3

\[
\begin{pmatrix}
3 & 5 \\
3 & 1
\end{pmatrix}
\]
\[
X = (1-3) / (3+1-5-3) = -2/-4 = 1/2
\]
\[
Y = (1-5) / (3+1-5-3) = -4/-4 = 1
\]
\[
V = ((3(1)) - (5(3))) / (3+1-5-3) = -12/-4 = 3
\]
Player 1 = (1/2, 1/2)  
Player 2 = (1, 0)  
V = 3

\[
\begin{pmatrix}
2 & 4 \\
5 & 6
\end{pmatrix}
\]
\[
X = (1-4) / (6+1-5-4) = -3/-2 = 3/2
\]
\[
Y = (1-5) / (6+1-5-4) = -4/-2 = 2
\]
\[
V = ((6(1)) - (5(4))) / (6+1-5-4) = -14/-2 = 7
\]
Player 1 = (3/2, -1/2)  
Player 2 = (2, -3)  
V = 7

\[
\begin{pmatrix}
6 & 5 \\
4 & 1
\end{pmatrix}
\]
\[
X = (6-5) / (2+6-4-5) = 1/-1 = -1
\]
\[
Y = (6-4) / (2+6-4-5) = 2/-1 = -2
\]
\[
V = ((2(6)) - (4(5))) / (2+6-4-5) = -8/-1 = -8
\]
Player 1 = (-1, 2)  
Player 2 = (-2, 3)  
V = -8

141
Determining Optimal Strategies in Matrix Games

\[
\begin{pmatrix}
6 & 4 \\
4 & 6
\end{pmatrix}
\]

\[
X = \frac{6-4}{6+6-4-4} = \frac{2}{4} = \frac{1}{2}
\]
\[
Y = \frac{6-4}{6+6-4-4} = \frac{2}{4} = \frac{1}{2}
\]
\[
V = \frac{(6(6)) - (4(4))}{6+6-4-4} = \frac{20}{4} = 5
\]
Player 1 = (1/2, 1/2)
Player 2 = (1/2, 1/2)
V = 5

\[
\begin{pmatrix}
2 & 5 \\
5 & 1
\end{pmatrix}
\]

\[
X = \frac{1-5}{3+2-5-3} = \frac{-4}{-3} = \frac{4}{3}
\]
\[
Y = \frac{1-5}{3+2-5-3} = \frac{-4}{-3} = \frac{4}{3}
\]
\[
V = \frac{(2(1)) - (5(5))}{3+1-5-3} = \frac{-23}{-3} = 7.66666 = \frac{23}{3}
\]
Player 1 = (4/3, -1/3)
Player 2 = (4/3, -1/3)
V = 23/7

Two intersecting points generate the same payoffs.

Optimal strategies and payoffs between;
Player 1 = (2/3, 1/3) and (1/2, 1/2)
Player 2 = (1, 0) and (1, 0)
V = 3
Appendix F

Raw Results Output

This appendix contains all of the screen shots from the testing of the application.
Determining Optimal Strategies in Matrix Games

F.1. 2*2 Matrix Game Testing

Unit Test 1

Enter your name:
Hemal Ramesh Patel
Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

You were supposed to enter either: Number 1 to create a randomly generated matrix or Number 2 to create a matrix, which you can enter in the elements within the table! The number you entered in was:
1
You were supposed to enter either: Number 1 to create a randomly generated matrix or Number 2 to create a matrix, which you can enter in the elements within the table! The number you entered in was:
1
You were supposed to Enter in an integer.
Error Type: java.util.InputMismatchException

You were supposed to Enter in an integer.
Error Type: java.util.InputMismatchException
Determining Optimal Strategies in Matrix Games

Enter your name:
Hemal Ramesh Patel
Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

5.504
You were supposed to Enter an integer.
Error Type: java.util.InputMismatchException

***** Beginning of Matrix *****
7   6
54   63
***** End of Matrix *****
Determining Optimal Strategies in Matrix Games

Unit Test 2

Enter your name: Hemal Ramesh Patel
Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

You were supposed to enter either: Number 1 to create a randomly generated matrix or Number 2 to create a matrix, which you can enter in the elements within the table! The number you entered in was: 2

You were supposed to enter either: Number 1 to create a randomly generated matrix or Number 2 to create a matrix, which you can enter in the elements within the table! The number you entered in was: 1

You were supposed to Enter in an integer.
Error Type: java.util.InputMismatchException
Determining Optimal Strategies in Matrix Games

Welcome Hemal Ramchh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramchh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

2 was entered.

Enter in four Integers. Invalid values are as follows: Strings, Floats, Doubles and Characters.

Hemal

You were supposed to Enter in an integer.

Error Type: java.util.InputMismatchException

Hemal Ramchh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

4.332

You were supposed to Enter in an integer.

Error Type: java.util.InputMismatchException
Determining Optimal Strategies in Matrix Games

Enter your name: Hemal Ramesh Patel
Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel entered in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

2 was entered.
Enter in your integers. Invalid values are as follows; Strings, Floats, Doubles and Characters
1 0 -1 2

• The matrix is as follows;

***** Beginning of Matrix *****
1 0
-1 2
***** End of Matrix *****
Unit Test 3

<table>
<thead>
<tr>
<th>Java - Two_By_Two_Matrix.java - Eclipse SDK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter your name:</td>
</tr>
<tr>
<td>Hemal Ramesh Patel</td>
</tr>
<tr>
<td>Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.</td>
</tr>
<tr>
<td>Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.</td>
</tr>
<tr>
<td>2 was entered.</td>
</tr>
<tr>
<td>Enter in four integers. Invalid values are as follows; Strings, Floats, Doubles and Characters</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>The matrix is as follows:</td>
</tr>
<tr>
<td>******* Beginning of Matrix *******</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>******* End of Matrix *******</td>
</tr>
<tr>
<td>There are no saddle points in this payoff table.</td>
</tr>
</tbody>
</table>
Determining Optimal Strategies in Matrix Games

Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

2 was entered.
Enter in four Integers. Invalid values are as follows:
Strings, Floats, Doubles and Characters
-2
-2
0

The matrix is as follows;

***** Beginning of Matrix *****
-2  -2
  3  0

***** End of Matrix *****

The element 0 is a saddle point
The optimal strategy is for player 1 to chose row 2 and for player 2 to chose column 2.
The payoff for Player 1 is 0 the payoff for Player 2 is -0

Welcome Hemal to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

1 was entered.
The matrix will be randomly generated.

The matrix is as follows;

***** Beginning of Matrix *****
93  4
  7  55

***** End of Matrix *****

There are no saddle points in this payoff table.
Determining Optimal Strategies in Matrix Games

Welcome Hemal to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal, enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

1 was entered.
The matrix will be randomly generated.
• The matrix is as follows;

***** Beginning of Matrix *****
40   61
58  88
***** End of Matrix ******

The element 40 is a saddle point
The optimal strategy is for player 1 to chose row 1 and for player 2 to chose column 1.
The payoff for Player 1 is 40 the payoff for Player 2 is -40
Determining Optimal Strategies in Matrix Games

Unit Test 4

```java
2
2 was entered.
Enter in four Integers. Invalid values are as follows;
Strings, Floats, Doubles and Characters
1
0
-1
2

* The matrix is as follows;

***** Beginning of Matrix *****
1    0
-1    2
***** End of Matrix *****

There are no saddle points in this payoff table.

The determinant is 2
```

```java
93
4
7
55

* The matrix is as follows;

***** Beginning of Matrix *****
93    4
7    55
***** End of Matrix *****

There are no saddle points in this payoff table.

The determinant is 5067
```
Determining Optimal Strategies in Matrix Games

Unit Test 5

```
<terminated> Two_By_Two_Matrix [Java Application] C:\Program Files\Java\jre6\bin\java.exe

* The matrix is as follows:

***** Beginning of Matrix *****
93   4
7    55
****** End of Matrix ******
* Printing off the Adjoint Matrix:

***** Beginning of Matrix *****
55   -4
-7    93
****** End of Matrix ******
```

Unit Test 6

```
<terminated> Two_By_Two_Matrix [Java Application] C:\Program Files\Java\jre6\bin\java.exe

* The matrix is as follows:

***** Beginning of Matrix *****
3    0
-1    2
****** End of Matrix ******
* Printing off the Adjoint Matrix:

***** Beginning of Matrix *****
2    0
1    1
****** End of Matrix ******
```

Unit Test 7

```
<terminated> Two_By_Two_Matrix [Java Application] C:\Program Files\Java\jre6\bin\java.exe

* The matrix is as follows:

***** Beginning of Matrix *****
93   4
7    55
****** End of Matrix ******

The value of JA* = (49,89)

* The matrix is as follows:

***** Beginning of Matrix *****
1    0
-1    2
****** End of Matrix ******

The value of JA* = (3,1)
```
Determining Optimal Strategies in Matrix Games

Unit Test 8

\[ \begin{pmatrix} 93 & 4 \\ 55 & \end{pmatrix} \]

***** Beginning of Matrix *****
93 4
55
***** End of Matrix *****
The value of JA^TJT = 137

The matrix is as follows:

Unit Test 9 - see results for Unit Test 4
Determining Optimal Strategies in Matrix Games

Unit Test 10

```
// Java - Two_By_Two_Matrix.java - Eclipse SDK

// Program File

** *** Beginning of Matrix ******
93  4
7  55
****** End of Matrix   ******

The optimal strategy for Player One is
[0.3503649350364965, 0.6496350364963503]
The optimal strategy for Player Two is
[0.37226273726273727, 0.6277372627737273]

** *** Beginning of Matrix ******
1   0
-1  2
****** End of Matrix   ******

The optimal strategy for Player One is
[0.75, 0.25]
The optimal strategy for Player Two is
[0.5, 0.5]
```
Determining Optimal Strategies in Matrix Games

Unit Test 11

```
<table>
<thead>
<tr>
<th>90</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

***** Beginning of Matrix *****

Probabilities (0.3005695803365985, 0.6994304196401415) are valid.
Probabilities (0.372267747262777, 0.627732267737233) are valid.

***** Beginning of Matrix *****

1 0
-1 2

***** End of Matrix *****

Probabilities (0.75, 0.25) are valid.
Probabilities (0.5, 0.5) are valid.
Probabilities (1.0, 1.0) are invalid.
Probabilities (0.25, 0.75) are valid.
Probabilities (-1.0, 2.0) are invalid.
Probabilities (-0.4, 0.6) are invalid.
```

Unit Test 12

```
<table>
<thead>
<tr>
<th>93</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

***** Beginning of Matrix *****

The payoff is 37.1313866131187

***** End of Matrix *****

The payoff is 0.5
```

```
Determining Optimal Strategies in Matrix Games

System Test

Enter your name:
Hemal Ramgoh Patel
Welcome Hemal Ramgoh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramgoh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

1
1 was entered.
The matrix will be randomly generated.
• The matrix is as follows:

```
   48  22
  01  30
```

The element 30 is a saddle point
The optimal strategy is for player 1 to choose row 2 and for player 2 to choose column 2.
The payoff for Player 1 is 30 the payoff for Player 2 is -30
Determining Optimal Strategies in Matrix Games

```
Enter your name:
Hemal Ramesh Patel

Welcome Hemal Ramesh Patel to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

2

2 was entered.
Enter in four Integers. Invalid values are as follows:
Strings, Floats, Doubles and Characters
-2
-2
3
0

• The matrix is as follows:

***** Beginning of Matrix *****
-2   -2
0   0
***** End of Matrix *****

The element 0 is a saddle point
The optimal strategy is for player 1 to choose row 2 and for player 2 to choose column 2.
The payoff for Player 1 is 0 the payoff for Player 2 is -0
```
Determining Optimal Strategies in Matrix Games

Hemal Ramesh Patel welcome to the program that works out optimal strategies in a Two Person Zero Sum Matrix game.

Hemal Ramesh Patel enter in 1 to generate a random matrix or 2 to generate a blank matrix which you will have to enter the numbers yourself.

2 was entered.

Enter in four integers. Invalid values are as follows;

1, 0, -1, 2

The matrix is as follows;

***** Beginning of Matrix *****

1  0
-1  2

***** End of Matrix *****

There are no saddle points in this payoff table.

The determinant is 2

Printing off the Adjoint Matrix:

***** Beginning of Matrix *****

2  0
1  1

***** End of Matrix *****

The value of JA*T = (3,1)

The value of the vector JT = (1,1)

The value of ATJ = (2,2)

The value of JA*T = 9

The optimal strategy for Player One is

[0.75, 0.25]

Probabilities [0.75, 0.25] are valid.

The optimal strategy for Player Two is

[0.5, 0.5]

Probabilities [0.5, 0.5] are valid.

The payoff is 0.5
Determining Optimal Strategies in Matrix Games

F.2. 2*N Matrix Game Testing

Unit Test 1

```
Java - Two_By_N_Matrix.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help

Javadoc    Console    

Two_By_N_Matrix [Java Application] C:\Program Files\Java\jdk1.5.0_03\bin\java.exe (2)

How many columns do you wish to have.
-1
Please enter in a positive value greater than one!
The value that you entered in was -1

0
Please enter in a positive value greater than one!
The value that you entered in was 0

1
Please enter in a positive value greater than one!
The value that you entered in was 1

Enter in 6 elements. Invalid values are as follows:
Strings, Floats, Doubles and Characters
```

```
Java - Two_By_N_Matrix.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help

Javadoc    Console    

Two_By_N_Matrix [Java Application] C:\Program Files\Java\jdk1.5.0_03\bin\java.exe (2)

How many columns do you wish to have.
3.145
You were supposed to enter in an integer.
Error Type: java.util.InputMismatchException
```

```
Java - Two_By_N_Matrix.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help

Javadoc    Console    

Two_By_N_Matrix [Java Application] C:\Program Files\Java\jdk1.5.0_03\bin\java.exe (2)

How many columns do you wish to have.
-1
Please enter in a positive value greater than one!
The value that you entered in was -1

0
Please enter in a positive value greater than one!
The value that you entered in was 0

1
Please enter in a positive value greater than one!
The value that you entered in was 1

Enter in 6 elements. Invalid values are as follows:
Strings, Floats, Doubles and Characters
```
Determining Optimal Strategies in Matrix Games

Unit Test 2

Error Type: `java.util.InputMismatchException`
Determining Optimal Strategies in Matrix Games

Unit Test 3

***** Beginning of Matrix *****
-1  5  0  8
4  3  2  12
***** End of Matrix *****

The linear equations for the 2 by 4 matrix is as follows:

Equation 1:  \( y = 2x + 2 \)
Equation 2:  \( y = -2x + 3 \)
Equation 3:  \( y = 3x + 1 \)
Equation 4:  \( y = -5x + 5 \)
Determining Optimal Strategies in Matrix Games

Unit Test 4

Programming output:

```
***** Beginning of Matrix *****
2    0    4
0    5   -1
***** End of Matrix *****

Number of 2x2 Matrices = 3
Printing off the 2x2 matrices that exist within the 2*N matrix.

***** Beginning of Matrix *****
2    0
0    5
***** End of Matrix *****

***** Beginning of Matrix *****
2    4
0   -1
***** End of Matrix *****

***** Beginning of Matrix *****
0    4
5   -1
***** End of Matrix *****

Number of 2x2 Matrices = 3
Payoff is 1.4285714285714286
Payoff is 0.5666666666666667
Payoff is 2.0
```
Determining Optimal Strategies in Matrix Games

******** Beginning of Matrix ******
0   2   4   6
6   4   2   0
******** End of Matrix ******

Number of 2x2 Matrices = 6
Printing off the 2x2 matrices that exist within the 2xN matrix.

***** Beginning of Matrix *****
0   2
6   4
***** End of Matrix *****

***** Beginning of Matrix *****
0   4
6   2
***** End of Matrix *****

***** Beginning of Matrix *****
0   6
6   0
***** End of Matrix *****

***** Beginning of Matrix *****
2   4
4   2
***** End of Matrix *****

***** Beginning of Matrix *****
2   6
4   0
***** End of Matrix *****

***** Beginning of Matrix *****
4   6
2   0
***** End of Matrix *****

Number of 2x2 Matrices = 6
Payoff is 3.0
Payoff is 3.0
Payoff is 3.0
Payoff is 3.0
Payoff is 3.0
Payoff is 3.0
Determining Optimal Strategies in Matrix Games

Unit Test 5

```
***** Beginning of Matrix *****
2   0   4
0   5   -1
***** End of Matrix *****

Printing off the 2x2 matrices that exist within the 2^N matrix.

Number of 2x2 Matrices = 3

***** Beginning of Matrix *****
2   0
0   5
***** End of Matrix *****

***** Beginning of Matrix *****
2   4
0   -1
***** End of Matrix *****

***** Beginning of Matrix *****
0   4
5   -1
***** End of Matrix *****

Player 1
| 0.7142857142857143 , 0.2857142857142857 |
| 0.3333333333333333 , 0.66666666666667 |
| 0.6 , 0.4 |

Player 2
| 0.7142857142857143 , 0.2857142857142857 |
| 1.66666666666667 , -0.3333333333333333 |
| 0.5 , 0.5 |
```
Determining Optimal Strategies in Matrix Games

```
          0  2  4  6
         6  4  2  0

Printing off the 2x2 matrices that exist within the 2xN matrix.

Number of 2x2 Matrices = 6

          0  2
         6  4

          0  4
         6  2

          0  6
         6  0

          2  4
         4  2

          2  6
         4  0

          4  6
         2  0

Player 1

| 0.5, 0.5 |
| 0.5, 0.5 |
| 0.5, 0.5 |
| 0.5, 0.5 |
| 0.5, 0.5 |
| 0.5, 0.5 |

Player 2

| -0.5, 1.5 |
| 0.25, 0.75 |
| 0.5, 0.5 |
| 0.5, 0.5 |
| 0.75, 0.25 |
| 1.5, -0.5 |
```
Determining Optimal Strategies in Matrix Games

Unit Test 6

Unit Test 7

Case 1: This matrix has got a unique high vertex.

Case 2: All columns in matrix intersect at the same single point.

Case 3: This matrix has got two probabilities that generate the same payoff.
Additional Testing

This appendix contains the extra 2*5 matrices that were used for additional testing.
Determining Optimal Strategies in Matrix Games

G.1 Additional 2*5 Matrices

\[
\begin{pmatrix}
0 & 1 & 3 & 4 & 5 \\
6 & 4 & 4 & 7 & 2
\end{pmatrix}
\]

Figure (a): 2*5 Matrix – Case 1 – Unique Vertex

\[
\begin{pmatrix}
1 & 1 & 2 & 3 & 4 \\
2 & 5 & 3 & 1 & 1
\end{pmatrix}
\]

Figure (b): 2*5 Matrix – Case 1 – Unique Vertex