Final Report – COEN 244 – RPS Project

Description of project:

The objective of this project is to create a learning AI for the game "Rock, Paper, Scissors" that can learn what to play depending on what the opponent plays.

The three playstyles that are used by AI are: **Random**, **Against Human Player**, and **Statistical**

Random:
The Random type of play consists of using the available functions from C++ to make the program randomly choose between Rock, Paper or Scissors.

The Random playstyle is the most risk free play style as it more than often results in the player scoring about even.

Against Human Player:
The Against Human Player playstyle consists of exploiting the playstyle of an average human player.

While doing research, I found a video called "Winning at Rock Paper Scissors – Numberphile" which talked about how to win at rock paper scissors.

According to the video from the YouTube channel "Numberphile", an average human player will tend to play in a certain predictable manner:

- If they win, the human player will tend to play the same hand again
- If they lose, the human player will tend to change their hand to the hand that would beat the hand that his opponent just played
- If they draw, the human will play randomly

In this case, the counter to this provided by the AI would this consists of:

- If you won last game: Play the hand that your opponent played last game.
- If you lost last game: Play the hand that beats what your opponent just played.
- If you draw last game: Play randomly.

The Against Human Player playstyle should work decently against an average human player, as the name suggests, however, the randomness of the draw does not guarantee a win for every turn, but it should guarantee a majority of wins for the player that uses the strategy against a human player.

Statistical:
The Statistical playstyle consists of trying to find patters of play from a player and trying to predict the play of such a player in order to beat them.

How the system works is that it will take every move made during the game at that point and try to set percentage chances of what the player will probably do after a certain result is obtained.

These three results consist of when, the last game was a **win**, the last game was a **loss**, or when the last game was a **draw**.

At the start of the game, all the odds for what the opponent will do, meaning that he will play the thing that beats the thing that he just played, the thing that loses to the thing he just played or that he will play the same thing again are set at 33% for the three categories.

At the end of each round, the program will tally the results and increase the odd by a confidence factor (β which is usually 5%) using a probability equation of the form:

New probability = (1 - β) * (Old probability) + β

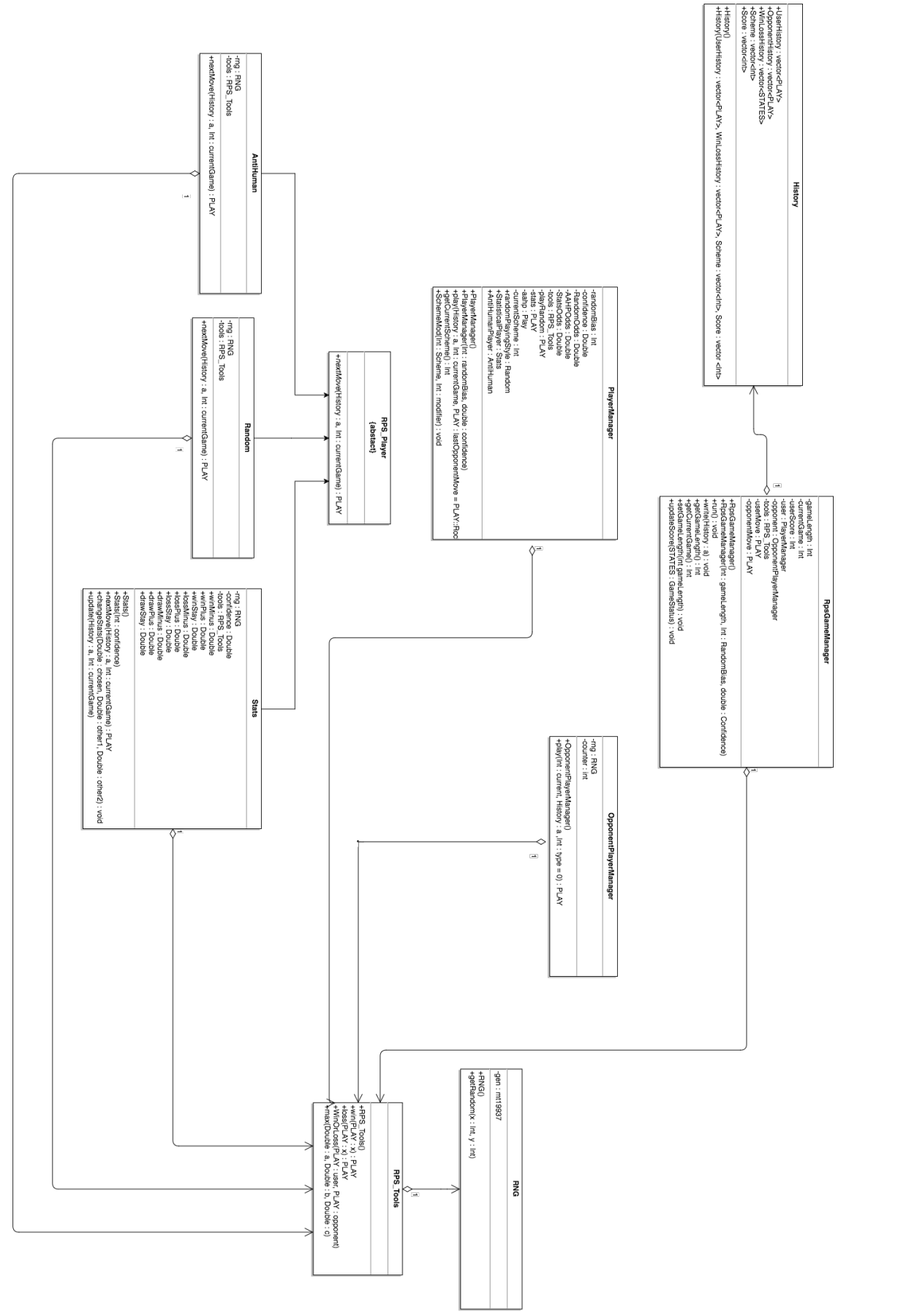
And by then equilibrating the odds by reducing the two other odds by 1/2 of what you just increased the first probability.

I also added that when a probability reaches a number under 0%, it is reset to zero and the difference is subtracted equally from the two other probabilities.

As well as that when a probability reaches a percentage higher that 80%, the probability will be reduced to 80%, and the difference will be added equally to the two other probabilities.

White-Box description:

UML:


History
+UserHistory : vector<PLAY>
+OpponentHistory : vector<PLAY>
+WinLossHistory : vector<STATES>
+Scheme : vector<int>
+Score : vector<int>
+History()
+History(UserHistory : vector<PLAY>, WinLossHistory : vector<PLAY>, Scheme : vector<int>, Score : vector <int>
RpsGameManager
-gameLength : Int
-roundsPlayed : Int
-currentGame : Int
-user : PlayerManager
-opponent : OpponentPlayerManager
-tools : RPS_Tools
-userMove : PLAY
-opponentMove : PLAY
+RpsGameManager()
+RpsGameManager(Int : gameLength, Int : RandombBias, double :Confidence)
+run() : void
+writeHistory : a) : void
+getGameLength() : Int
+getCurrentGame() : Int
+setGameLength(Int : gameLength) : void
+updateScore(STATES :GameStatus) : void
PlayerManager
-randombBias : Int
-confidence : Double
-RandomOdds : Double
-AntiHOdds : Double
-StatsOdds : Double
-tools : RPS_Tools
-playRandom : PLAY
-stats : PLAY
-aiHp : Play
-currentScheme : Int
+randomPlayingStyle : Random
+StatisticalPlayer : Stats
+AntiHumanPlayer : AntiHuman
+PlayerManager()
+PlayerManager(Int : randomBias, double : confidence)
+play(History : a, Int : currentGame, PLAY : lastOpponentMove = PLAY.Roc
+getCurrentScheme() : Int
+SchemeMod(Int : Scheme, Int : modifier) : void
OpponentPlayerManager
-rng : RNG
-counter : Int
+OpponentPlayerManager()
+play(Int : current, History : a ,Int : type = 0) : PLAY
RPS_Player
(abstract)
+nextMove(History : a, Int : currentGame) : PLAY
AntiHuman
-rng : RNG
-tools : RPS_Tools
+nextMove(History : a, Int : currentGame) : PLAY
Random
-rng : RNG
-tools : RPS_Tools
+nextMove(History : a, Int : currentGame) : PLAY
Stats
-rng : RNG
-confidence : Double
-tools : RPS_Tools
+winMinus : Double
+winPlus : Double
+winStay : Double
+lossMinus : Double
+lossPlus : Double
+lossStay : Double
+drawMinus : Double
+drawPlus : Double
+drawStay : Double
+Stats()
+Stats(Int : confidence)
+nextMove(History : a, Int : currentGame) : PLAY
+changeStats(Double : chosen, Double : other1, Double : other2) : void
+update(History : a, Int : currentGame)
RPS_Tools
+RPS_Tools()
+win(PLAY : x) : PLAY
+loss(PLAY : x) : PLAY
+WinOrLoss(PLAY : user, PLAY : opponent)
+max(Double : a, Double : b, Double : c)
RNG
-gen : mt19937
+RNG()
+getRandom(x : Int, y : Int)

Pseudo-Code:

RpsGameManager:

Class RpsGameManager{
 	
Private:
Int gameLength;
\\ Length of game (# of rounds)
 	
Int currentGame;
\\ Current round
 	
Int userScore;
\\ current score of AI
 	
PlayerManager user;
\\ Instance of PlayerManager for the AI
 	
OpponentPlayerManager opponent;
\\ Instance of OpponentPlayerManager for the player or opponent
 	
RPS_Tools tools;
\\ Instance of tools from RPS_Tools for calculation help
 	
PLAY userMove;
\\ Stores the current move of the user
 	
PLAY opponentMove;
\\ Stores the current move of the opponent or player
 	
Public:
 	
RpsGameManager();
\\ Default constructor
 	
RpsGameManager(int gameLength, int RandomBias, double Confidence);
\\ Non-Default constructor with parameters to set the length of the game and to set the RandomBias and Confidence of the AI through the user object.
	Void run();
	\\ Initialises the game with the information given like the length of the game.
	
	Void write(History a);
	\\ Function that writes down the results obtained during the whole game into .csv files.
	
	Int getGameLength();
	\\ returns the length of the game
	
	Int getCurrentGame();
	\\ return the number of the current round
	
	void setGameLength(int gameLength);
	\\ function to modify the length of the current game
	
	void updateScore(STATES GameStatus);
	\\ Updates the userScore variable based on the GameStatus state.
	
	}

PlayerManager:

Class PlayerManager{
 	
Private:
Int randomBias;
\\ The number of turns the player manager will run Random before letting the program play on its own
 	
double confidence;
\\ The confidence factor used in the probability equation
 	
double RandomOdds;
\\ The current odds of using the random playstyle during the next game
 	
double AAHPOdds;
\\ The current odds of using the anti-human playstyle during the next game.
 	
double StatsOdds;
\\ The current odds of using the statistical playstyle during the next game.
 	
PLAY playRandom;
\\ stores the play that random would play
 	
PLAY stats;
\\ stores the play that statistical would play
 	
PLAY aahp;
\\ stored the play that anti-human would play
 	
RPS_Tools tools;
\\ Instance of tools from RPS_Tools for calculation help
 	
int currentScheme;
\\ integer value that stores which playstyle is currently being used.
	Public:
	
	Random randomPlayingStyle;
	\\ stores an instance of Random to use during the game
	
	Stats StatisticalPlayer;
	\\ stores an instance of Stats to use during the game
	
	AntiHuman AntiHumanPlayer;
	\\ stores an instance of AntiHuman to use during the game.
	
	PlayerManager() \\ Default Constructor
	
	PlayerManager(int randomBias, double confidence);
	\\ Non-Default PlayerManager constructor to initialise other objects inside the class
	
	PLAY play (History const &a, int currentGame, PLAY lastOpponentMove = PLAY::Rock)
	\\ returns a play based on the given History object, the currentGame and the last move of the player or opponent. If no argument is given, the last move of the opponent will be defaulted as Rock.
	
	Int getCurrentScheme();
	\\ return the scheme being currently used
	
	void SchemeMod(int Scheme, int modifier);
	\\ will modify the odds of the Schemes based on which Scheme number is given in the arguments and based on the modifier, which refers whether the Scheme would have won or lost (1 or -1)
	
	}

OpponentPlayerManager:

```
Class OpponentPlayerManager{
Private:
Int counter; \\ counter is required for one of the playstyles

RNG rng; \\ rng object for getting a random value

RPS_Tools tools; \\ Instance of tools from RPS_Tools for calculation help

public:
OpponentPlayerManager();// Default-Constructor

PLAY play (int currentGame, History a, int type = 0);
\\ will return a play based on what the type is:
\\ 0: Random
\\ 1: RRRR…
\\ 2: RPSRPS…
\\ 3: Average Human (Requires History object and currentGame)
\\ 4 or other: actual on computer input

}
```

RPS_Player:

```
Class RPS_Player{
Public:
Virtual PLAY nextMove(History const &a, int currentGame) = 0
\\ Pure virtual function that returns a move based on a history object and the currentGame
}
```

Stats:

Class Stats : public RPS_Player{ Private: RNG rng; \\ rng object for getting a random value Double confidence; \\ The confidence factor used in the probability equation RPS_Tools tools; \\ Instance of tools from RPS_Tools for calculation help Public Double winMinus; \\ Odds that after a win, opponent plays +1 Double winPlus; \\ Odds that after a win, opponent plays -1 Double winStay; \\ Odds that after a win, opponent plays the same move Double lossMinus; \\ Odds that after a loss, opponent plays +1 Double lossPlus; \\ Odds that after a loss, opponent plays -1 Double lossStay; \\ Odds that after a loss, opponent plays the same move Double drawMinus; \\ Odds that after a draw, opponent plays +1 Double drawPlus; \\ Odds that after a draw, opponent plays -1 Double drawStay; \\ Odds that after a draw, opponent plays the same move	Stats() \\ Default Constructor Stats(double confidence); \\ Non-Default contructor with confidence PLAY nextMove(History const &a, int currentGame); \\ returns a move according to the History object, the currentGame, and the odds accumulated across the game. Void changeStats(double &chosen, double &other1, double &other2); \\ Changes the stats of a playstyle Void update(History const&a, int currentGame); \\ Updates the stats of the playstyles based on the information given by the History and currentGame objects.

Random:

```
Class Random : public RPS_Player {

Private:
RNG rng;
\\ rng object for getting a random value

RPS_Tools tools;
\\ Instance of tools from RPS_Tools for calculation help


Public:

PLAY nextMove(History const &a, int currentGame);
\\ Function inherited from RPS_Player :
\\ returns move randomly (Does not use History or currentGame)
}
```

Anti-Human:

```
Class AntiHuman : public RPS_Player{

Private:
RNG rng;
\\ rng object for getting a random value

RPS_Tools tools;
\\ Instance of tools from RPS_Tools for calculation help

Public:
PLAY nextMove(History const &a, int currentGame);
\\ Function inherited from RPS_Player :
\\ returns move based on the information from History and currentGame based on the Anti-
\\Human strategy.

}
```

RPS_Tools:

```
enum PLAY {Rock,Paper,Scissors};
\\ Enumeration for the moves used throughout the program

enum STATES {Win,Loss,Draw};
\\ Enumeration from the states used throughout the program

Class RPS_Tools{
Public:
RPS_Tools();
\\ Default constructor

PLAY win(PLAY x);
\\ Returns the move that wins against the given move (+1)

PLAY loss(PLAY x);
\\ Returns the move that loses against the given move (-1)

STATES WinOrLoss(PLAY user, PLAY opponent);
\\ Returns the states of a given round (Win, Loss, Draw)

Double max(double a, double b, double c);
\\ Returns the max from three double values
}
```

History:

```
Class History{
Public:

Vector<PLAY> UserHistory;
\\ Vector containing the history of moves made by the AI during the game

Vector<PLAY> OpponentHistory;
\\ Vector containing the history of moves made by the opponent or player during the game

Vector<STATES> WinLossHistory;
\\ Vector containing the history of wins, losses and draw during the game.

Vector<int> Scheme;
\\ Vector containing the schemes used during the game

Vector<int> Score;
\\ Vector containing the current score during every round of the game.

History();\\ Default Constructor

History(Vector<PLAY> UserHistory, Vector<PLAY> OpponentHistory,Vector<STATES>
WinLossHistory, Vector<int> Scheme, Vector<int> score);
\\ Non-Default Constructor with UserHistory, OpponentHistory, WinLossHistory, Scheme and
\\ Score arguments to initialise them through the constructor

}
```

RNG:

(THIS CLASS WAS GENEROUSLY GIVEN TO US BY THE TA IN ORDER TO SIMPLIFY OUR LIVES)

```
Class RNG{

Public:
RNG();// Default constructor to seed the mt19937 gen object

Int getRandom(int x, int y);
\\ return a randomly generated number between x and y inclusively

Private:
mt19937 gen;
\\ mt19937 object used to obtain a random object
}
```

Main:

```
Int main(){

Say begin to indicate beginning;

Say end to indicate End;

Initialise RPSGameManager with specific arguments: Game length, how long to make AI stay on random playstyle, confidence factor.;

Run RPSGameManager ;

Skip line;

System(“pause”);

Return 0;

}
```

Output Files & Format:

For the outputting of data, since I already store all of the data from the game

For the format, I use a .CSV format that eases the transition from txt file to excel file, as the CSV type is already an Excel file.

For the formatting inside the text file, you put your data, then, a comma, then your data again, and then a comma and keep going in this format

Ex:

In text file: "Data1,Data2,Data3,Data4,…,DataN"

In Excel file:

Data1	Data2	Data3	…	DataN

This was the format used for the following files:

OpponentPlays.csv
UserPlays.csv
Score.csv
SchemesChosen.csv

For the other files, which are the indicators for the Schemes, I used the following scheme:

Ex:

In text file: ",,1,1,1,,"

In Excel file:

		1	1	1		

This was the format used from the following files:

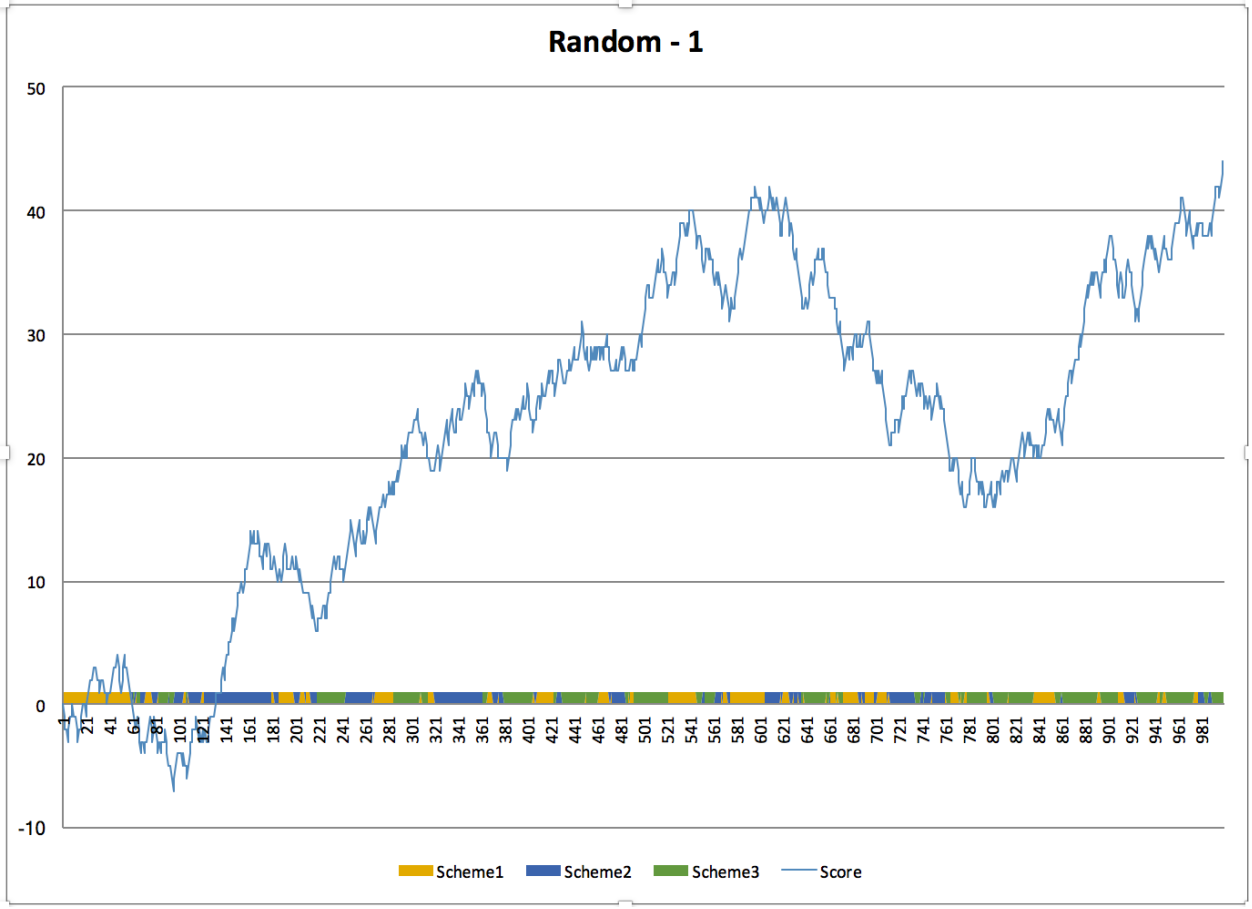
Scheme1.csv
Scheme2.csv
Scheme3.csv

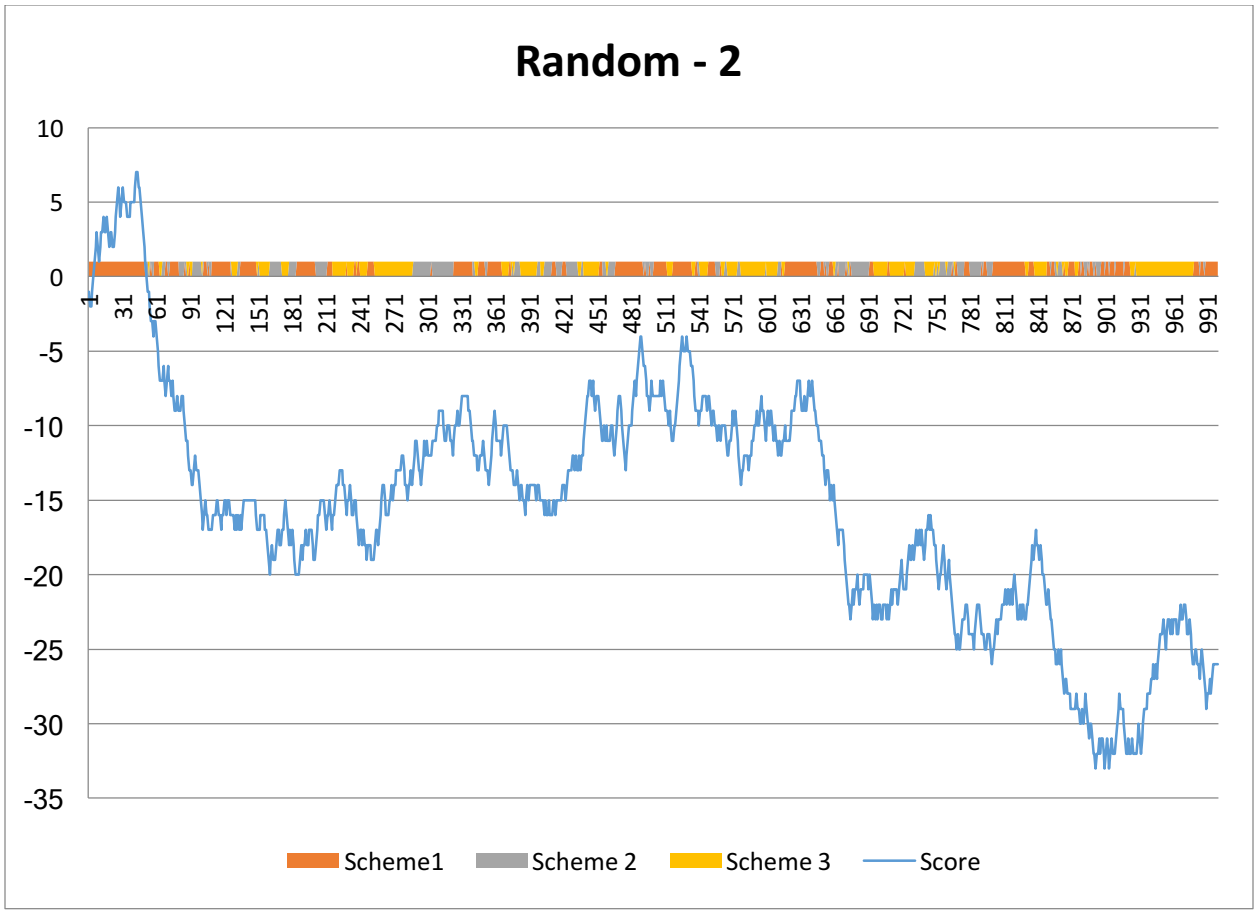
Output Testing & Tests:

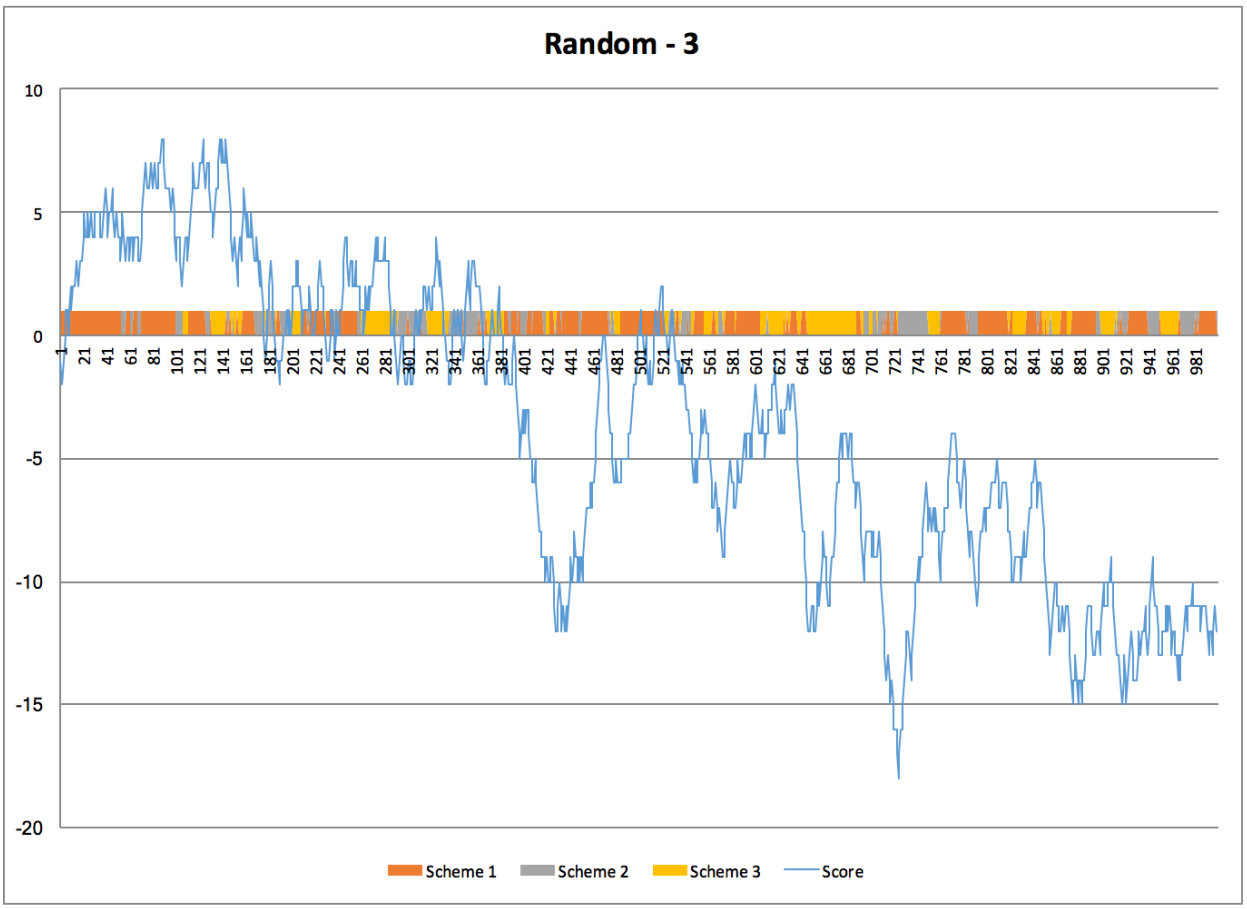
For the testing, I tested my program against a player with four different playing styles:
1. Playing randomly
2. Only playing Rock
3. Only playing Rock, then Paper, then Scissors
4. Playing like an average human

I will show three different test for each of these testing environments and explain how the algorithms are used during the game.

1) Playing randomly




Random - 2
10
5
0
-5
-10
-15
-20
-25
-30
-35
1
31
61
91
121
151
181
211
241
271
301
331
361
391
421
451
481
511
541
571
601
631
661
691
721
751
781
811
841
871
901
931
961
991
Scheme1
Scheme 2
Scheme 3
Score



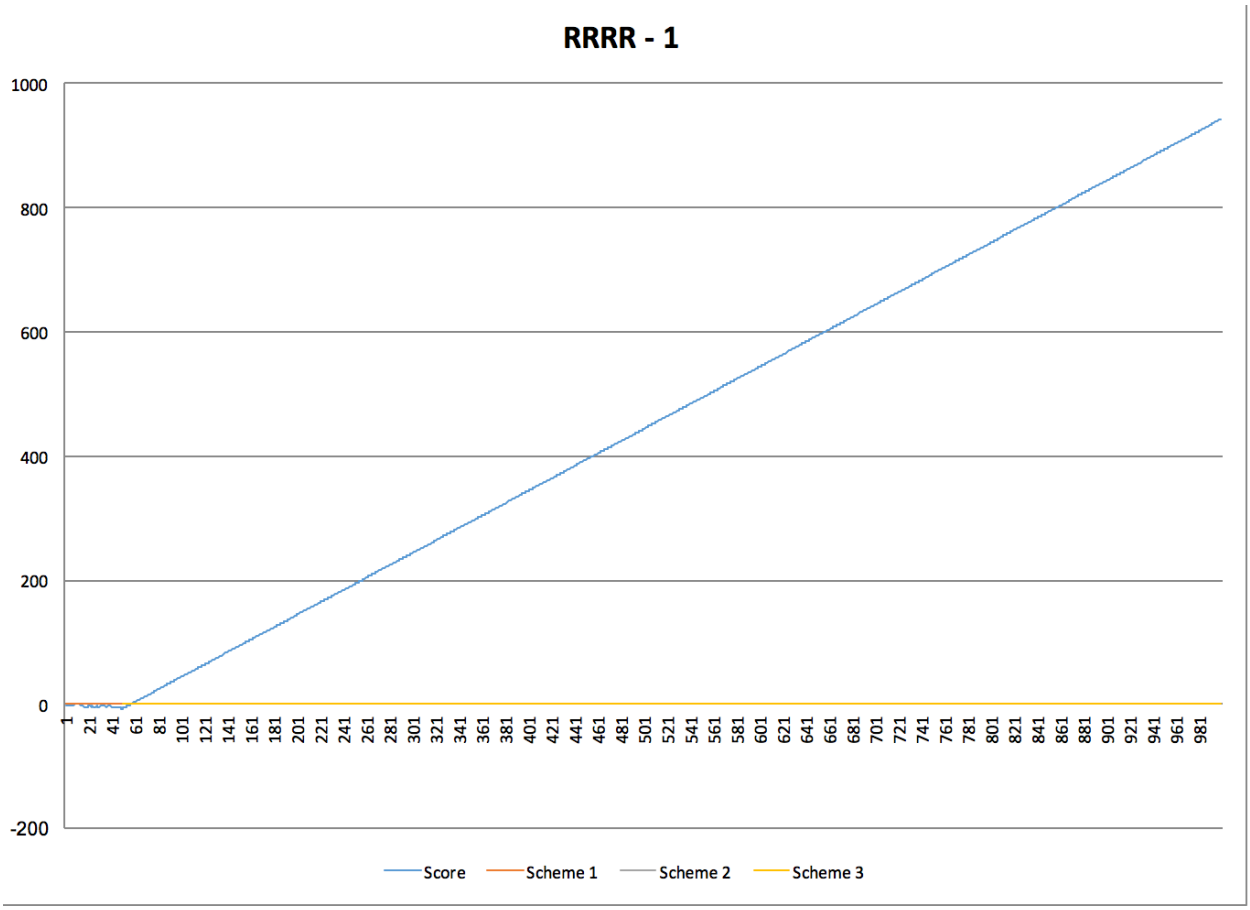
As we can see, the results range from about -27 to 45, which averages to around 0.
This is the result that was expected, as having two players playing Rock, Paper, Scissors randomly should end in a draw, since the chances of winning are about equal for each player.

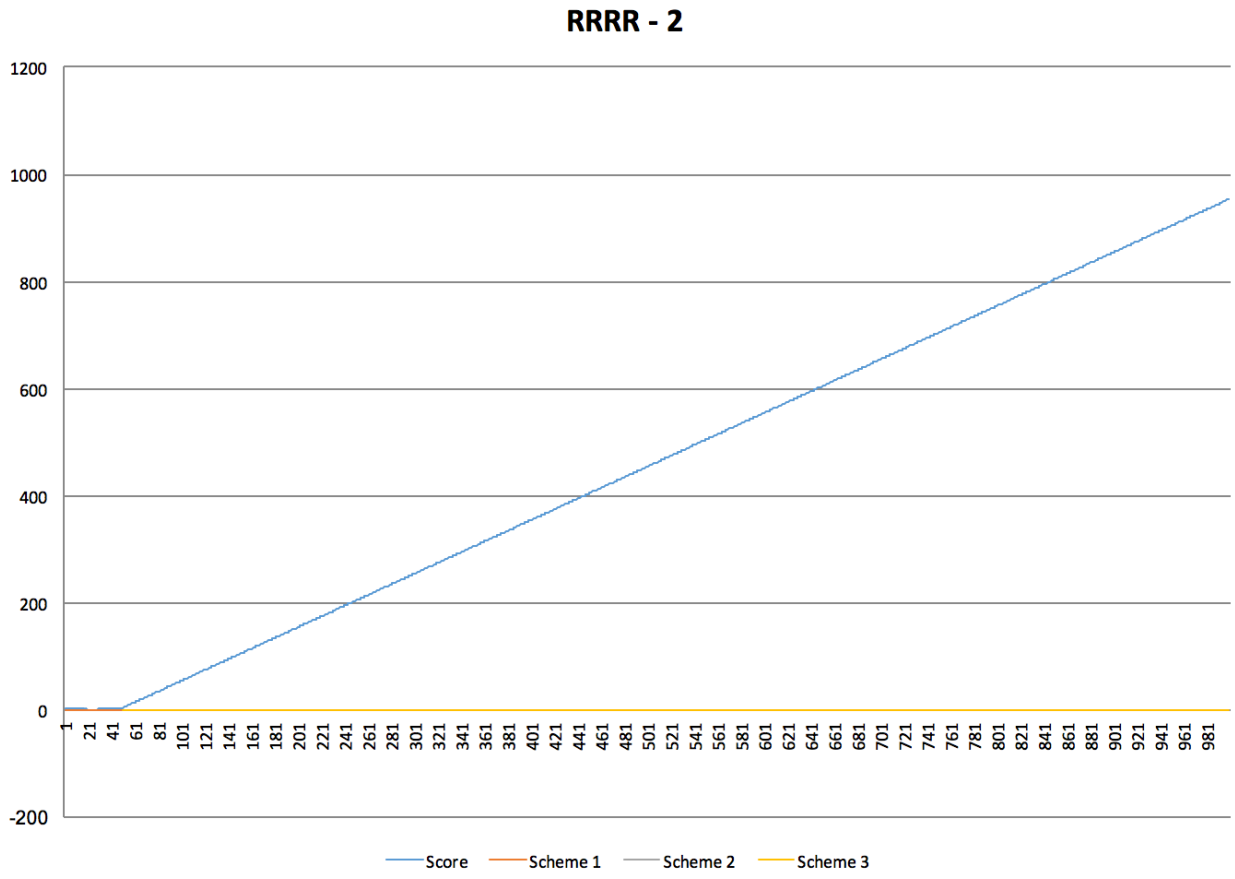
Here, the algorithms are used here are, from most used to least used:

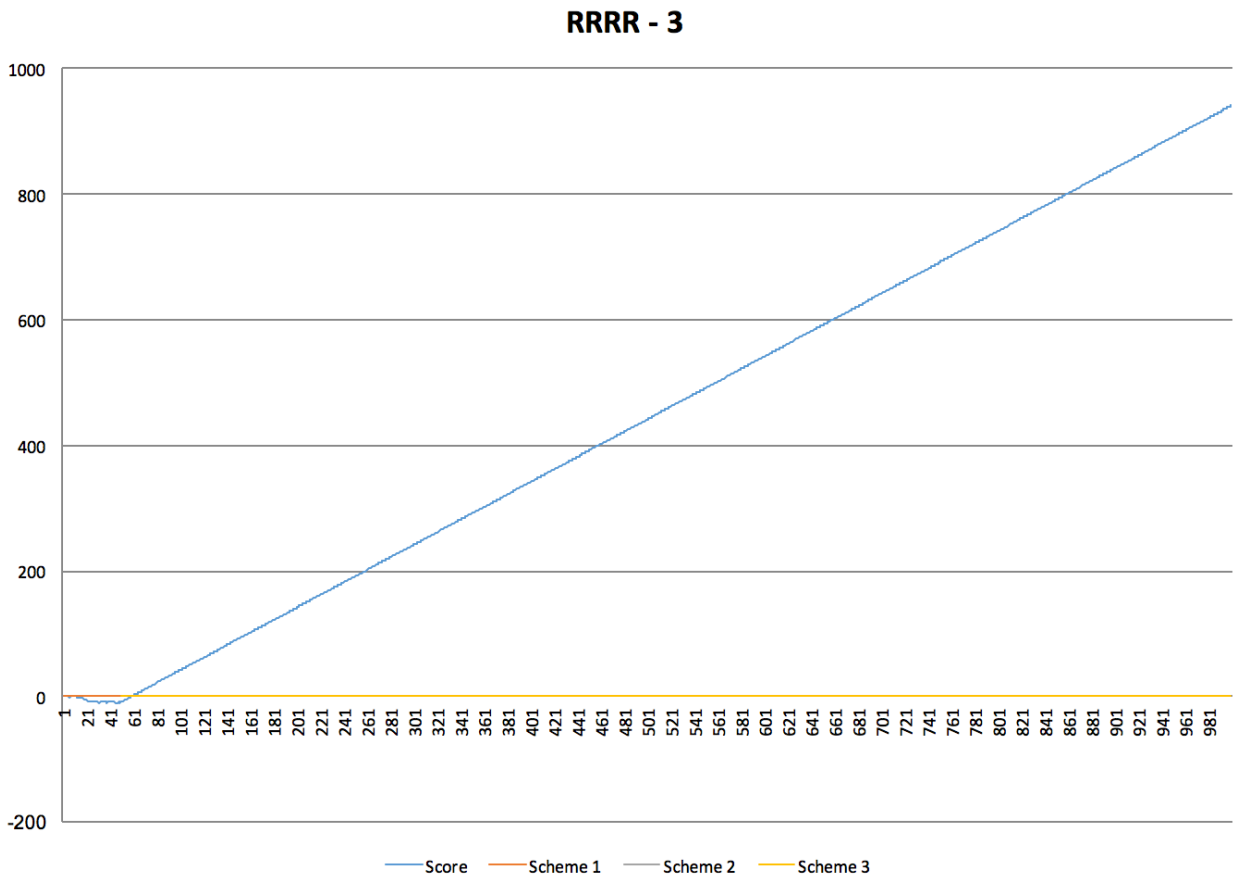
- Scheme 1: Random
- Scheme 3: Statistical
- Scheme 2: Anti-Human

These specifications make sense as the AI would gain the most from playing randomly, being about even, would lose only a little by playing statistically, and would lose quite a lot if playing as if it was a human player.

2) Only playing Rock




RRRR - 2
1200
1000
800
600
400
200
0
-200
1
21
41
61
81
101
121
141
161
181
201
221
241
261
281
301
321
341
361
381
401
421
441
461
481
501
521
541
561
581
601
621
641
661
681
701
721
741
761
781
801
821
841
861
881
901
921
941
961
981
Score
Scheme 1
Scheme 2
Scheme 3

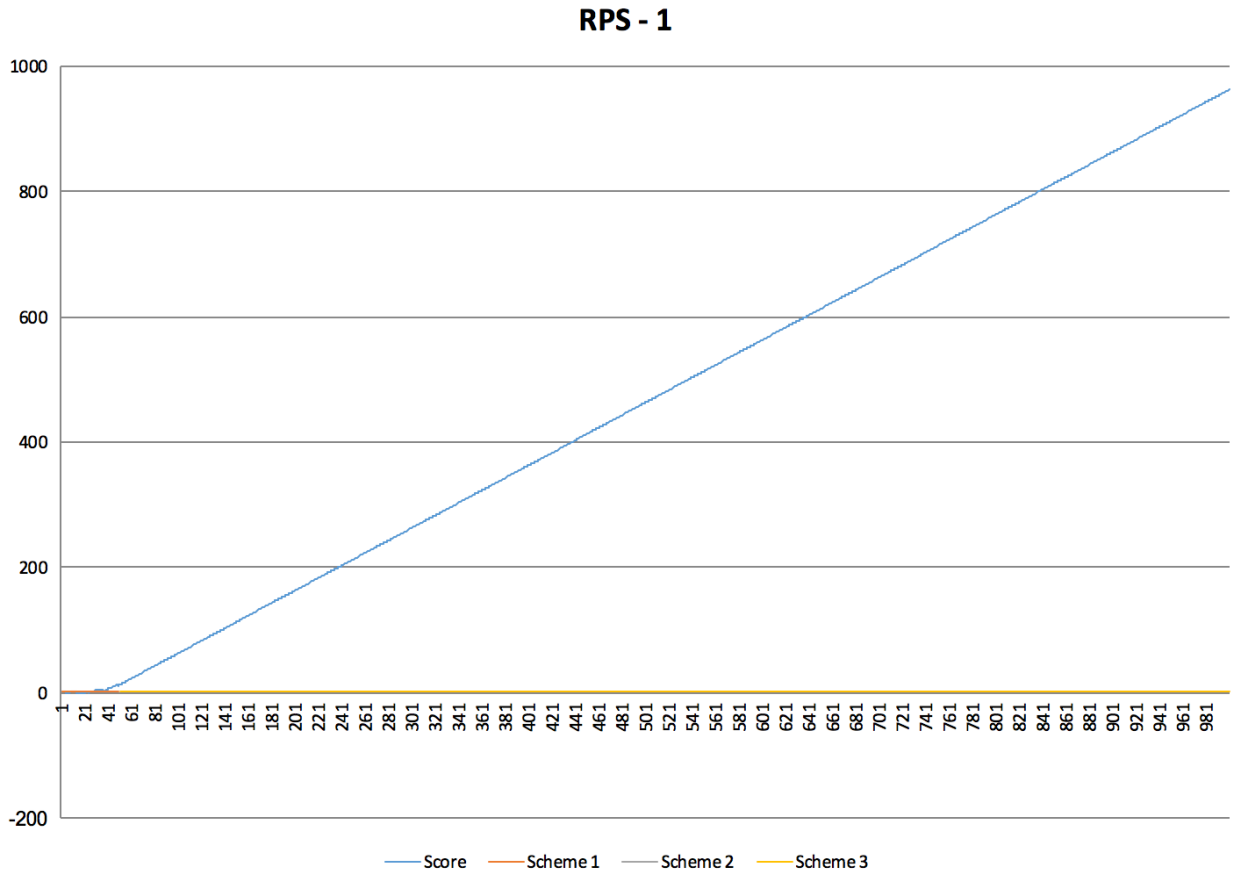


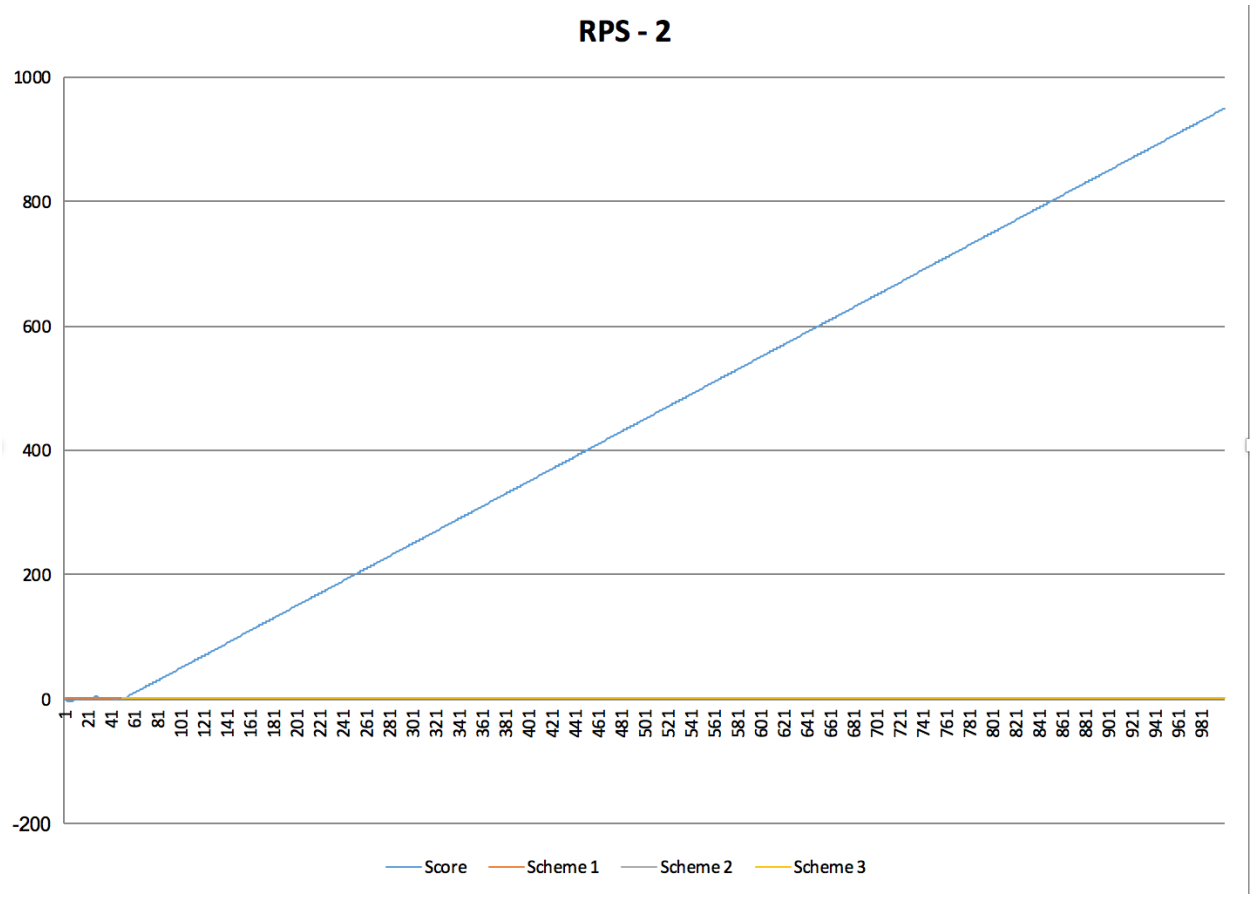
As we can see here, when the opponent only plays Rock, the AI dominates the game by scoring from 940 to 960, depending on the first 50 turns, which are biased into being random.

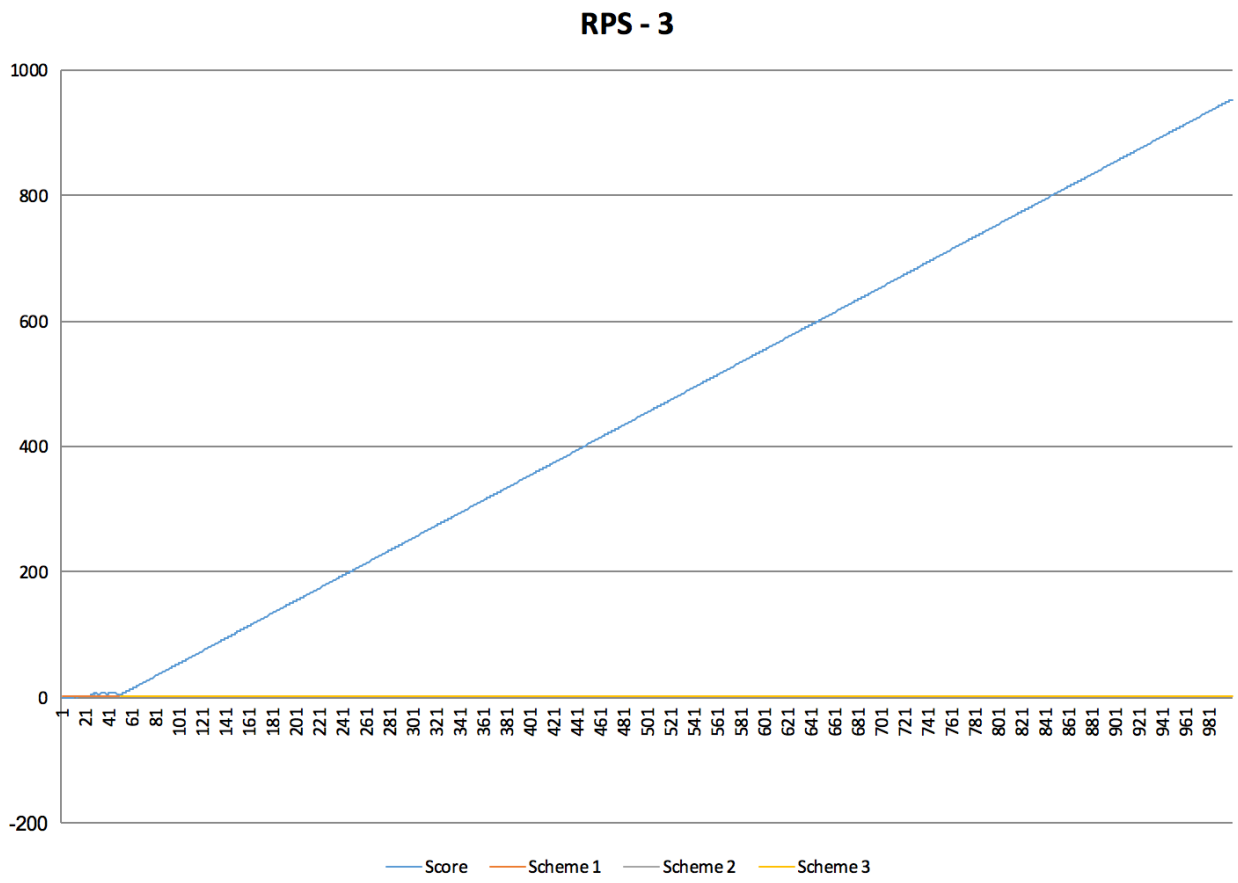
This is also the expected result, as only playing Rock is a very predictable and simple strategy.

Here, the program first starts using the random playstyle, as it is programmed to do so for the first 50 turns, and then uses the statistical playstyle, as the player only plays R,R,R,R… which a very easy pattern to spot.

3) Only playing Rock, then Paper, then Scissors




RPS - 2
1000
800
600
400
200
0
-200
1
21
41
61
81
101
121
141
161
181
201
221
241
261
281
301
321
341
361
381
401
421
441
461
481
501
521
541
561
581
601
621
641
661
681
701
721
741
761
781
801
821
841
861
881
901
921
941
961
981
Score
Scheme 1
Scheme 2
Scheme 3

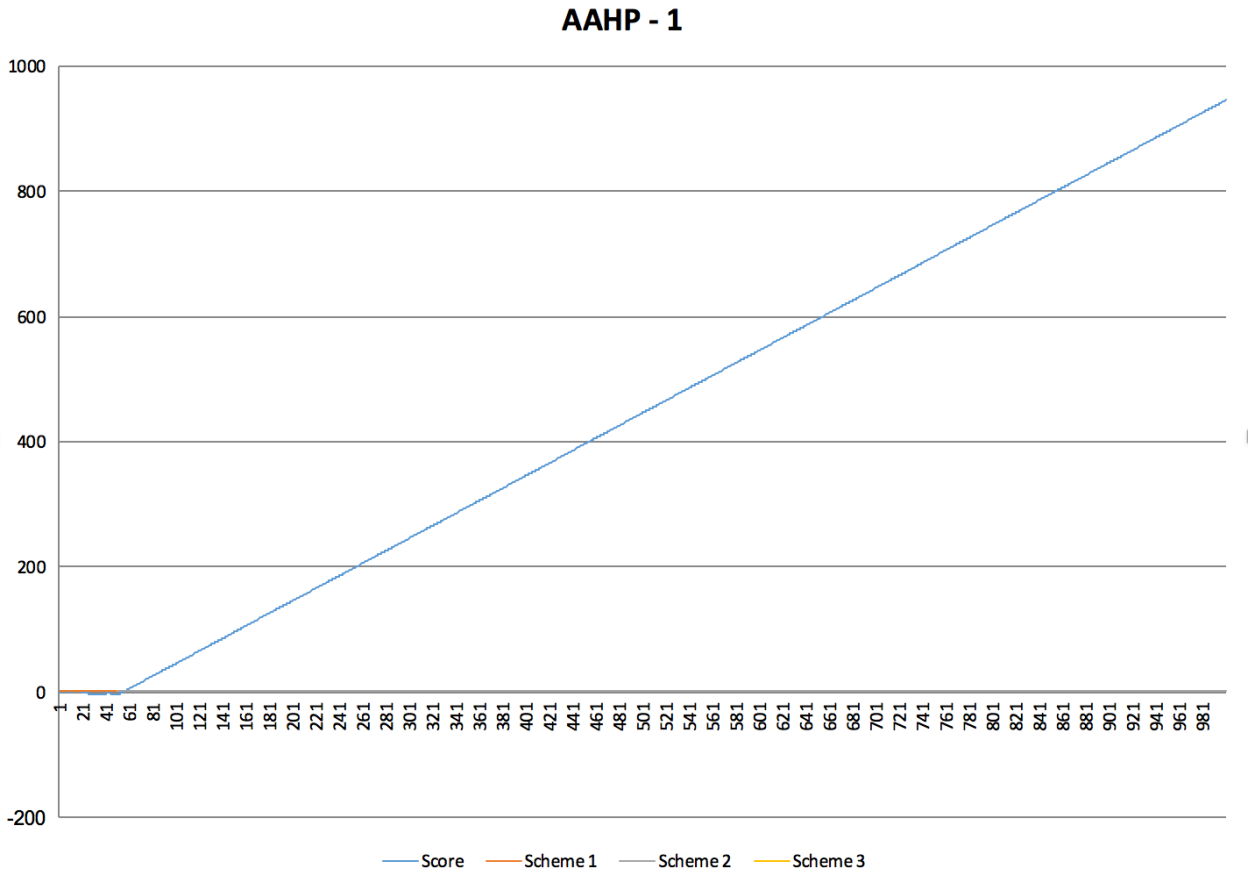


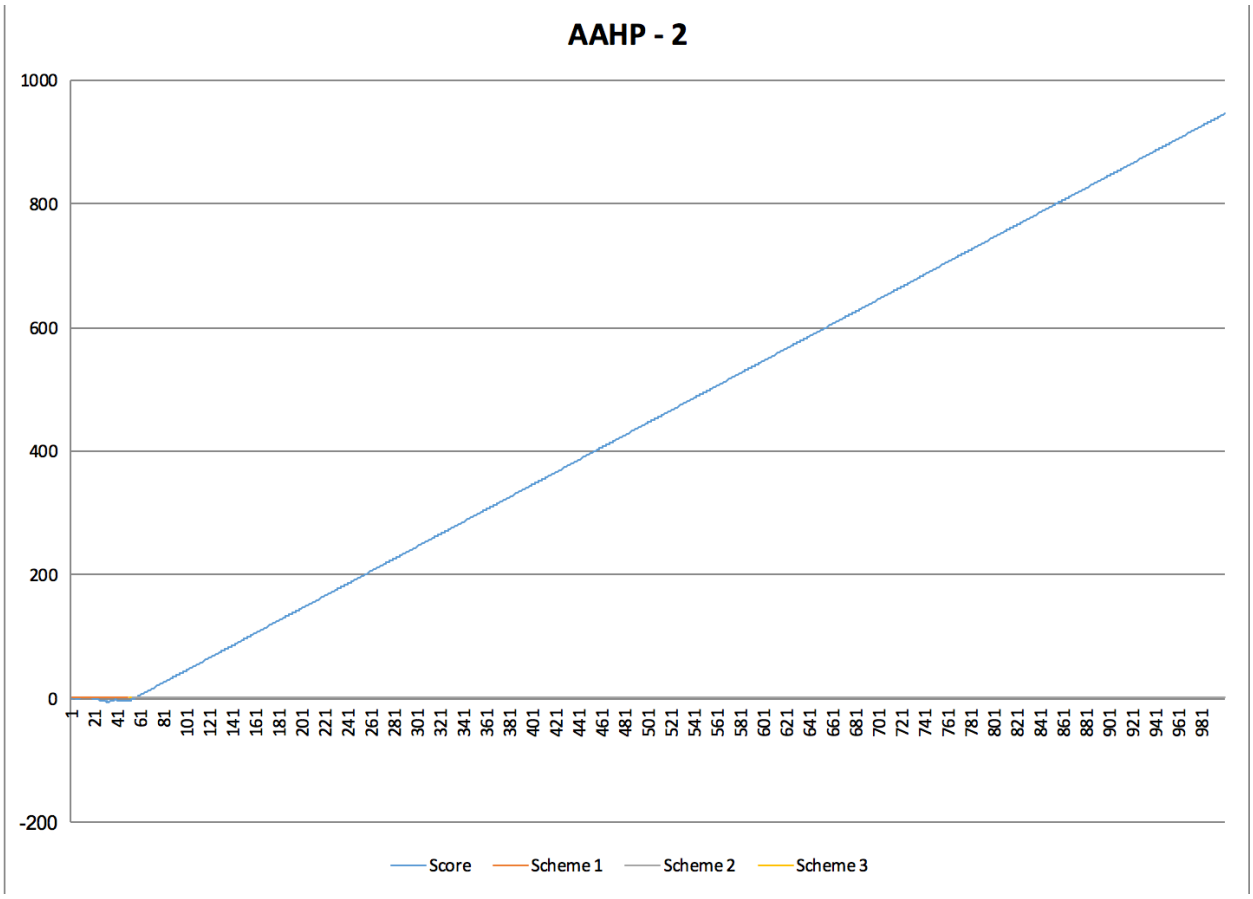
As we can see here, when the opponent only plays Rock, then Paper, the Scissors, the AI dominates the game by scoring from 940 to 960, depending on the first 50 turns, which are biased into being random.

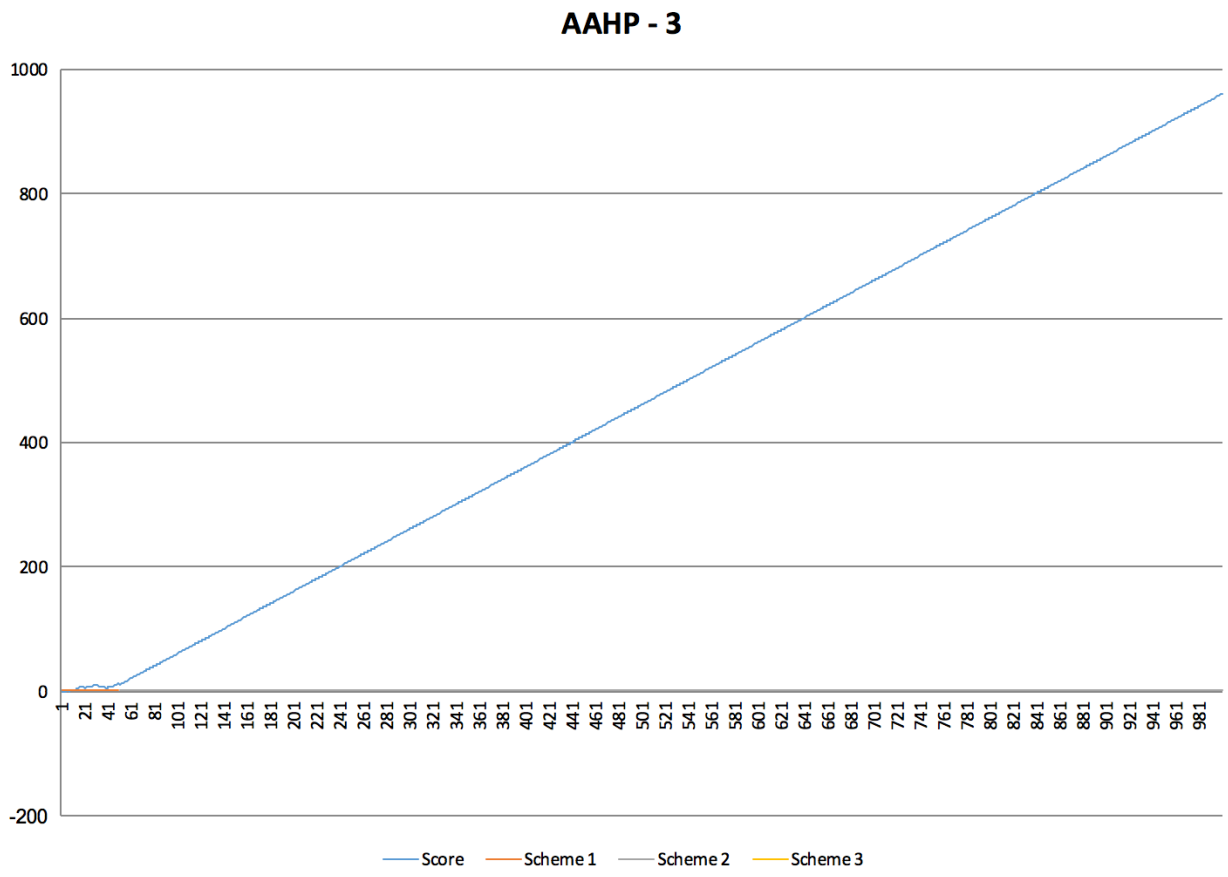
This is also the expected result, as only playing Rock, then Paper, then Scissors is a predictable and simple strategy.

Here, the program first starts using the random playstyle, as it is programmed to do so for the first 50 turns, and then uses the statistical playstyle, as the opponent only plays R,P,S,R… which a very easy pattern to spot for the AI.

4) Playing like an average human




AAHP - 2
1000
800
600
400
200
0
-200
1
21
41
61
81
101
121
141
161
181
201
221
241
261
281
301
321
341
361
381
401
421
441
461
481
501
521
541
561
581
601
621
641
661
681
701
721
741
761
781
801
821
841
861
881
901
921
941
961
981
Score
Scheme 1
Scheme 2
Scheme 3



As we can see here, when the opponent plays using the average human strategy, the AI dominates the game by scoring from 940 to 960, depending on the first 50 turns, which are biased into being random.

This is also the expected result, as scheme 2, which is the anti-average human strategy is the perfect counter to this strategy.

Here, the program first starts using the random playstyle, as it is programmed to do so for the first 50 turns, and then uses the anti-human playstyle for the rest of the game, as the opponent follows the human strategy outlined in the beginning of this document.

Help for testing purposes:

In order to test the program, the main changes are to be done inside the main file and inside the RPSGameManager file.

In the Main file, the RPSGameManager constructor controls the game length, the random bias of the player manager and the Confidence of the player manager.

For the Game Manager, the OpponentPlayerManager constructor works as such:

OpponentPlayerManager (int currentGame, History a, int type)

These are the corresponding types available for the opponentPlayerManager:

- If Type = 0 –> Opponent will be random
- If Type = 1 -> Opponent will only play Rock
- If Type = 2 –> Opponent will only play Rock, then Paper, then Scissors
- If Type = 3 –> Opponent will play average human playstyle
- If Type is anything else –> Opponent will be controllable through Cin to input moves.

Cin input testing:

For the Cin input testing, the interface is extremely simplistic.

It will show :
Enter your value : (0 : Rock, 1: Paper, 2:Scissors):

You then enter the wanted value and the move will be processed.

If you enter any other number than 0, 1, or 2, it will default at 2.

I also came into the problem of entering characters and strings. When this happens, its automatically runs the program as if the user entered 2 every turn left and then stops when the program ends.

PS for testing:

For testing different opponents, do not forget to change both values of type for the OpponentPlayerManager as not doing so will make the opponent no do what is requested.

```
void RpsGameManager::run() {
    History a;

    for (int i = 0; i < gameLength; i++) { //Loop to accomplish the number of games given

        if (i == 0) {
            userMove = user.play(a,i);              //User requires Last opponent move, however, none exist at
                turn 0, then -> default
            opponentMove = opponent.play(i,a,1);    //Opponent move : (i,a,0) -> Random
                                                    //(i,a,1) -> RRRR...
                                                    //(i,a,2) -> RPSRPS...
                                                    //(i,a,3) -> Average Human
                                                    //(i,a,4) -> Actual play
                                                    //Modify X {(i,a,X)} to change the opponent type
                                                    //DONT FORGET TO CHANGE THIS IF YOU WANT TO CHANGE
                                                        OPPONENT TYPE
        }
        else {
            userMove = user.play(a,i,a.OpponentHistory[i-1]); //Uses opponent's last move
            opponentMove = opponent.play(i,a,1);    //Same as before
                                                    //DONT FORGET TO CHANGE THIS IF YOU WANT TO CHANGE OPPONENT
                                                        TYPE
        }
```