OCR GCE A

COMPUTER SCIENCE PROJECT

H446-03

Name : Alex Dyer

Candidate Number: 8572

Tuxford Academy: 28148

Title of Project: KnightOwl Chess Bot

H446-03 – Project CONTENTS

Table of Contents

[A. Analysis 4](#_Toc189659950)

[Project Identification 4](#_Toc189659951)

[Stakeholders 4](#_Toc189659952)

[Why is this Suited to a Computational Solution? 4](#_Toc189659953)

[Stakeholder Statistics 6](#_Toc189659954)

[My Microsoft Forms Questions 6](#_Toc189659955)

[Final Stakeholder Responses 7](#_Toc189659956)

[Research 10](#_Toc189659957)

[Existing Similar Solutions 10](#_Toc189659958)

[Features of My Proposed Solution 14](#_Toc189659959)

[Requirements 14](#_Toc189659960)

[Software and Hardware Requirements 14](#_Toc189659961)

[Stakeholder Requirements 15](#_Toc189659962)

[Success Criteria 16](#_Toc189659963)

[B. Design 17](#_Toc189659964)

[Systems diagram 17](#_Toc189659965)

[Intro Screen 17](#_Toc189659966)

[Chess Games 18](#_Toc189659967)

[Analysis/Review Screen 18](#_Toc189659968)

[Problem Decomposition – Two Player Chess Game 18](#_Toc189659969)

[Algorithms and Pseudocode 20](#_Toc189659970)

[All Algorithms for Main.py 20](#_Toc189659971)

[All Algorithms for Moves.py 21](#_Toc189659972)

[All Algorithms for Graphics.py 29](#_Toc189659973)

[Usability Features 34](#_Toc189659974)

[Navigation 34](#_Toc189659975)

[Consistency 34](#_Toc189659976)

[User Feedback 34](#_Toc189659977)

[Visual Clarity 34](#_Toc189659978)

[Error Prevention 34](#_Toc189659979)

[Variables and Classes 37](#_Toc189659980)

[Iterative Development Test Data 38](#_Toc189659981)

[C: Developing the coded solution 44](#_Toc189659982)

[Development Process 44](#_Toc189659983)

[D. Evaluation 51](#_Toc189659984)

[Project Appendixes 52](#_Toc189659985)

[References 52](#_Toc189659986)

# A. Analysis

## Project Identification

There have been many different games made over the millennia. Some are older than others, some are newer, and some can be played on a screen with a controller or a keyboard. But one game that has withstood the test of time is Chess. Chess is a strategy game where each player controls sixteen pieces – a king, a queen, two rooks, two knights, two bishops, and eight pawns. They must manoeuvre these pieces around an 8x8 chequered board to checkmate their opponent’s King. This game has been played and studied obsessively for over 1,500 years.

More recently, AI has advanced so that many bots have been programmed to play chess to a higher level than any human. Working off the Elo scale, the greatest player of all time, Magnus Carlsen, has an approximate Elo of 2,900. This is the highest human chess Elo in history. However, chess computers such as Stockfish, Torch, and AlphaZero have been made to beat Magnus Carlsen and the rest of the world’s approximately 1,800 GMs. But one thing that is a lot rarer is a chess bot aimed at being beatable by people of around 1000-1300 Elo. People always go on about these absurdly good chess players and chess bots. How about one that plays similarly to a human at an intermediate to advanced level of chess? This is the absence that I would like to remedy with my project.

I will use Python and one of Python’s graphics libraries Pygame to create KnightOwl. The features of a computer that would be required to use this would be the use of a keyboard and mouse as Python does not work with touchscreens.

### Stakeholders

The clients and general demographics of this bot are chess players from complete beginners to about 1,600 Elo. Because over 70% of chess players fall within this range, the stakeholders will be a representative sample, ranging from people who have just begun playing chess to people who are very comfortable with the game and its various strategies.

Stakeholders for the chess bot represent casual players who play chess occasionally to take a break from work/school to experienced people who want a new challenge or an easy game.

The game, for the most part, will be playable with just a mouse, but there will be a function that takes chess notation as an input too. It will come with an analysis feature that breaks a game down into single moves and explains the pros and cons of each move.

### Why is this Suited to a Computational Solution?

This project lends itself to a computational solution in many ways. The solution will be a chess computer that isn’t so hard to beat that it demoralises new players and angers gifted players, but one that anyone can play against and have a decent chance at winning against.

#### Problem Recognition

The main problem with this is creating a chess bot that plays like a human, as most chess bots have a completely different play style from any human because they’re programmed to make a good move or a worse move randomly. The chances of a good move compared to a bad move differ based on the not’s supposed Elo. My problem is that to make this work, I have to program a specific style into the computer and get it to stick to it. Most human chess players play with their unique style and preferences, and that’s why no two chess games are the same. The challenge is to create a bot that works in the same way.

#### Decomposition

This issue can be broken down into x smaller steps:

1. Work out the graphics for the program (e.g. the board, pieces, movement animation, etc)
2. Code the game itself
3. Build a basic bot
4. Improve the bot
5. Work out the finer details, such as a playstyle and aggressive/defensive tactical/positional preferences

When these steps are complete, the program doesn’t lag, and the bot can beat me (13-1400 Elo) about half the time, it’s done. With machine learning and reinforcement learning, I should be able to get the bot to play against itself a sufficient amount of times so that it can train itself to get to a high enough level, putting its findings into an SQL database. I can hard-code any chess theory needed into the database, to make it seem more like a human.

#### Divide and Conquer

Although these steps seem challenging, they are perfectly feasible. Solving each of these independently and combining them into a complete, modular program uses the divide-and-conquer way of problem-solving.

## Stakeholder Statistics

### My Microsoft Forms Questions

I will summarise the main questions that I have asked my stakeholders about their general chess-playing, and whether they’d be interested in my proposed project.

#### Current Chess Info for each Stakeholder

Questions 1-3 summarise how often the stakeholder plays chess and how good they are at it

1. Do they play chess?
2. Current chess Elo
3. How often do they play chess?

#### Views on Current Chess Bots

Questions 4-6 are on the stakeholder’s typical opponents and their views on the existing bots

1. How often do they play against bots?
2. Would they consider playing a new chess bot?

#### Would they Play my Chess Bot?

Questions 7-9 are views regarding my chess bot

1. Would they be interested in seeing a chess bot that plays like a human would up to an Elo of 1600?
2. What name would be suggested for the chess bot?

### Final Stakeholder Responses

A white background with black text

Description automatically generated

A graph with different colored bars

Description automatically generated

A white background with black text

Description automatically generated

The first three questions show what quantity of stakeholders play chess and their strengths/regularity with chess. The statistics show that most of the stakeholders play chess and, of the people who do play chess, the majority are between 400 and 1100 Elo. This would mean that a 1600 Elo bot would be a difficult, but reachable challenge for them. Also, a lot of them either play rarely or weekly, meaning that, hopefully, my bot would get the people who play rarely to play more often and therefore increase the popularity of chess.

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

According to the statistics from the survey, most people usually or only play other people, however, they would like to see and, for the most part, would definitely or possibly consider playing a bot at around 1600 Elo. As shown above, all stakeholders asked would like to see a bot that plays like a human to an Elo of 1600. Therefore, I think that the chess bot will be popular among chess players of all skill levels.

A screenshot of a phone

Description automatically generated

I will decide on a name nearer the time

## Research

### Existing Similar Solutions

#### chess.com

##### Overview

A screenshot of a game

Description automatically generated

Chess.com contains over fifty chess bots of various skill levels, ranging from absolute beginner levels to being better than any Grandmaster. This provides a wide range of fun bots who each go for their own various opening selection, tactics, fortes, and catchphrases. Complete with an easy-to-use user interface, social/friends database to play chess with friends, online capabilities, puzzles, and lessons, this is a user-friendly and fun way to play chess against anyone, regardless of whether or not you know them.

The user interface is immensely simple and easy to follow as the menu is always displayed and there are large buttons on-screen labelled with anything that can be done on the site. The home screen contains the menu, a suggestion of a player on your friends list to play against, puzzles and a game review suggestion, as shown below:

A screenshot of a game

Description automatically generated

A screenshot of a game

Description automatically generated

The ‘Game Review’ function allows a player to get the chess engine to analyse one game a day on the free version, or unlimited depending on what subscription plan they have. It contains game accuracy, the number of different types of moves made, the best moves for each position, why they were the best moves and an Elo prediction.

A screenshot of a game

Description automatically generated

A screenshot of a game

Description automatically generated

Higher ratings mean better gameplay, as do higher accuracies. The Game Review feature is very easy to understand and very informative of good and bad moves and sums them up into two numerical values (Elo and accuracy)

##### What can I apply To My Solution?

In my solution, I would like to build an easy-to-follow user interface, similar to chess.com’s, as well as an analysis/game review function to show players where they can improve. The main difference between the bots on chess.com and the bot I plan to code is that the bots on chess.com play completely differently from humans, in so far as they are designed to have a randomised chance to play a move, changed based on the supposed rating of each bot. Mine is supposed to play more similarly to a human and play more human moves that, although they may not be the best moves of an engine, are more likely to feel natural to a beginner/intermediate chess player.

#### Lichess.org

A screenshot of a computer

Description automatically generated

This is another chess site that is one of chess.com’s main competitors. Although it isn’t as good as chess.com, it has a massive opening database, showing the most common and the best human moves for hundreds of assorted chess openings. It also has a FEN analysis function, meaning you can get the engine to analyse any given position. Similar to chess.com, it also has a game analysis function, however, this one isn’t as good as it only shows the bad moves and doesn’t rate the good moves made by the player. As shown in the chess.com diagram above, chess.com shows all moves ranging through brilliant, great, best…mistake, miss, blunder. Lichess only focuses on inaccuracies, mistakes, and blunders. It is a very good, albeit immensely critical chess engine.

Another feature that it has in common with chess.com is the fact that the player can also play the computer. However, the computers here aren’t as immersive as the ones on chess.com, as they don’t possess the same skill range, catchphrases, or range of bots.

##### What Can I Apply to My Solution?

The only feature that I would probably use for my chess bot would be the opening database because it would help me differentiate ‘Book’ moves and theory from any middlegame/endgame moves. It would also vastly help see what rating the user plays like, based on how common the opening is and whether they do something classic for many people of a low Elo, which is to attempt to invent and utilise their openings, instead of learning any theory.

### Features of My Proposed Solution

#### My Initial Solution Proposition

My solution will be a chess bot that plays like a 1,600 Elo human because, although both lichess and chess.com both contain their own bots, neither of them contains bots that play like a human. In this case, I think that a combination of the lichess opening database, put into my own SQL database, and a similar range to the chess.com bots, in that, I could attempt to do different difficulties, ie: 400, 700, 1000, 1300, 1600 Elos. These bots will have a visual move-log and game review/self-analysis functions similar to the ones offered by chess.com, however, unlike chess.com, there is unlimited access to these features for free. There will also be a function to save played games so that they can be looked at again later down the line for improvements and future analysis.

#### Potential Limitations

The main limitation of my solution is that, although it incorporates a lot of functions based around the bot, it won’t include any online functionality, puzzles, or lessons, as the main point of this solution is to code the bot(s).

Another is that, because I’m using Pygame instead of a more advanced GUI, the UI is likely to be fairly straightforward. It will include some buttons for which bot you want to play against, and a list of recent saved games, with analysis if analysis is requested for the game in question.

## Requirements

### Software and Hardware Requirements

#### Hardware Requirements

**A computer capable of running an up-to-date version of Pygame** –The chess bot will need to be run on a computer capable of running Python 3.10, MySQL and Pygame. A laptop with a processor capable of running at 30+ fps is recommended, but not required

**A mouse and keyboard** – Pygame isn’t compatible with touchscreens, so mobile devices/iPads won’t be compatible with the software

**6GB RAM** – 4GB is the minimum RAM requirement to run Pygame, and 2GB is my current estimate of how much RAM will be required to access the MySQL database. Anything below this could potentially cause the chess bot to crash

**3GB HDD/SSD space** – Pygame will need 3GB of HDD or SSD space to run, MySQL needs another 2GB and the image files need about 100KB, so at least 7-8GB will be needed to record

#### Software Requirements

**Windows, Mac, or Linux OS** – These are the Operating Systems that support Python and SQL

**480p** **Resolution** – This will be the minimum monitor resolution required to play the chess game as it needs to be a resolution where the user does not need to strain their eyes to make out what pieces are what, so they don’t damage their eyes

**Python Interpreter** – The code will be written in Python, with MySQL and Pygame libraries

**MySQL for Python** – The opening database will require an SQL database, and therefore a Python plugin for SQL

**Pygame for Python** – The GUI will require a Python GUI library to run the chess bot graphics

### Stakeholder Requirements

#### Design

|  |  |
| --- | --- |
| Requirement | Explanation |
| Basic user interface with the ability to change the difficulties of the bots | So the user can adjust and access the features of the program easily |
| Easy-to-understand, informative analysis programs for any game played | Allows the user to rate their performance on each game based on the engine’s analysis of their game, and their analysis |
| An instructions page | Shows the user any key binds associated with the program, how they are used and how to navigate the program if they don’t understand it or aren’t good with technology in general |

#### Functionality

|  |  |
| --- | --- |
| Requirement | Explanation |
| Use of mouse to control moves | The mouse’s position is recorded by Pygame and is used to control which piece was clicked and where it was moved to |
| Use of keyboard to undo moves and input chess notation | The backspace key is going to be the keybind for undoing moves, and the main keyboard will be used to input chess notation as another way to control piece movement |
| Instructions on how to edit the theme of the board and the style of the pieces | If the user dislikes the look of the chess board and pieces, there will be instructions on how to edit the themes within the game |

## Success Criteria

|  |  |
| --- | --- |
| Criterion | How to show it’s been met |
| The main window showing a working chess engine | Screenshot of working graphics/video of a working engine |
| The AI makes non-randomised moves | Run a game through Stockfish to show a decent amount of logic |
| Simple, navigable design | Screenshot of the GUI, with a menu, chess engine and instructions |
| A working Exit button | The Exit button (namely the big X in the top right) is clicked and shows that it closes the window |
| Instructions | A screenshot of a page that’s accessible through the main menu, walking the user through the keybinds and any other features that they need to know and that points them to a site where they can learn to play chess if they don’t already know |
| The chess AI plays at the desired level (1,600 Elo) | The AI can beat me over 50% of the time that it plays me, as shown through the PGN files of the games |

# B. Design

## Problem Decomposition

### Decomposition Overview

There are many smaller steps that my chess engine can be broken down into. These are as follows:

* Board Representation and Generation
* Turn Tracking
* Piece Movement
* Move Generation (excluding special moves)
* Castling
* Pawn Promotion
* En Passant
* Move Validation and Illegal Move Prevention
* Detecting Check, Checkmate and Stalemate
* Event Handling
* Board Colour and move log

### Detailed Decomposition

#### Board Representation and Generation

This is solved specifically in my graphics.py file and is a problem that can be solved on its own, but it’s vital that it be solved first before anything else so that testing is easier. These functions (namely generating the board and pieces) will be called upon for each iteration of the main loop and therefore, to a degree, these are closely linked to the main.py file. Although there are existing functions that could do this for me, I felt that it would suit my code better if I coded these algorithms myself, to ensure that they work exactly how I want and need them to, especially if some of the code required is closed-source. Although the main functions of this are to generate the board and the pieces, it also highlights squares that pieces can move to and the square that gives the user the ability to select a piece to promote a pawn to (I will go over these in more detail later in the decomposition).

#### Turn Tracking

Although I would consider this a problem in its own right, and a vital component to the overall solution, this can be implemented easily through the use of a Boolean flag, which is true if its white’s turn to move and false if its black’s turn, or vice versa. This would be used within the main.py and moves.py files, as it would be essential to the main workings of the program, as well as finding all the legal moves for one side or the other in a position so that it knows which side it should be finding moves for. This is used as I coded the board array so each square containing a piece or a pawn also holds its colour (i.e. ‘bQ’ would mean Black Queen – each square has two characters, with ‘--’ meaning empty)

## Systems diagram

There will be many elements to my project. The main one is the chess engine that will be originally made for two players on the same device. The following design is for that and for the prospective main menu, and other elements will be shown later depending on my timescales.

### Intro Screen

The Main Menu screen will be fairly simple, with options to play the bot, play two-player, enter settings or access the openings database. The option to play KnightOwl will take the user straight to a game and will randomly assign a side (white or black) to each player.

Settings, as shown in the diagram above, will allow the user to alter the designs and colour schemes of the board and pieces so that they can use their preferred look for the board. These preferences will be saved to a text file, so they don’t need to be configured every time the game is loaded. It will also allow the user to adjust the difficulty level (Elo) of KnightOwl so they can play at a level that challenges them.

Lastly, the openings database will contain a board and FEN input bar, so that the most common opening moves can be found in a position (if no games have been played in a position, it will say so). This information will be pulled from lichess.org’s API whenever a new position is given to the system. The API will show what win/draw/loss rates, the average ratings of the players who use that specific move and the most common moves afterwards.

### Chess Games

The chess games, although the basic GUI makes them look pretty straightforward, are in fact, extremely complex. These include a series of interlinked algorithms which narrow down the moves available by considering checks, pins and of course the general movement rules of the game, algorithms to search for pins and checks and a constantly changing 2-dimensional board array.

After seeing the difficulty ranges of the bots available on chess.com, I’ve decided to make KnightOwl a bot playable by anyone and everyone by attempting to include difficulty levels up to (hopefully) 3,000 Elo, however I don’t know how manageable that will be. The pseudocode for the game as a two-player game and with the computer will be shown below separately.

### Analysis/Review Screen

These screens will be available after any game played against a bot or 2-player on the same device. This will give a rundown of a player’s performance in a game by showing an evaluation bar of who was winning and by how much after each move, as well as a system used by chess.com to class moves based on how good they were (Best, Excellent, Good, Inaccuracy, Mistake, Blunder) with Brilliant and Great moves being shown as well as Misses and Book (Opening Theory) moves. Essentially, I want to make this similar to the one offered by chess.com without having to pay an extortionate amount of money to get more than one a day. It will also give an estimate of a player’s rating based on the single game, as well as a rating based on which bots they beat and how consistently.

## Problem Decomposition – Two Player Chess Game

There are various sub-problems that my chess bot needs to be broken down into. Each of these will require its own separate algorithm (flow charts and pseudocode later). I’ll start with the chess game itself, separately to the bot (minus the analysis and review screen for now too).

The problems that I will be breaking down are as follows:

1. Adding the board and pieces to the screen – this will be done using the blit operation in Pygame to add 32 darker squares to the screen over a white background, then I can download and once again use blit to add

these to my screen. This is fairly self-explanatory: if I don’t do this, then the user will have no usable interface with which to interact and play chess on

1. Moving the Pieces Randomly – I’ll use a two-dimensional array called ‘board’ to store the current positions of all the pieces on the board currently. When a piece is moved, the array can be altered appropriately to portray the new layout. Without this ability, I can’t start to code and test move generation for the different pieces. This is immensely important as the users need to be able to see the current board state to make it easier to visualise follow-up moves and to calculate
2. Move Generation – If the pieces could move anywhere randomly, the King would be taken on the first move and it would be game over. This makes the game playable (mostly – en passant, pawn promotion and castling aren’t added yet) and uses a Boolean flag to show whose turn it is. This will be done using Object Oriented Programming in a separate file to create all the different algorithms and functions needed to find all the possible moves and narrow them down to the valid moves – the flow charts later lay out how I plan to cover all the necessary functions for this.
3. Check, checkmate and stalemate – These are some of the most important rules of chess. It will determine whether the King is being attacked and whether any pieces are pinned. This will be done through more functions within the moves.py file and a long algorithm to remove any ‘illegal’ moves, including linked algorithms to search for pins and checks.
4. Pawn Promotion – This is one of three more complex rules in chess. This will check if a pawn is on the ‘back rank’ and if it is will promote it to a Queen, Rook, Knight or Bishop, depending on the user’s choice.

I can do this using some graphics to create a square on the board containing a rook, queen, knight and bishop where the user clicks one of them and the pawn promotes to one or other of them

1. En Passant – The second more complex rule in chess. If one pawn moves two squares forward and is horizontally adjacent to an enemy pawn, that enemy pawn can take it as if it had only moved one square. I can make a subroutine that is called from the pawn move function that checks if a pawn moved two squares on the previous go and if so, add any moves for pawns that could play en passant
2. Castling – The third more complex rule in chess. If there are no pieces between the King and Rook and neither has moved, the king can move two spaces to the side and the rook stands on the adjacent square, closer to the centre. This will need to be a separate function in the moves class which detects if the king and/or rooks have moved

## Algorithms and Pseudocode

### All Algorithms for Main.py

Pseudocode can be found in the Pseudocode Folder within write-up

#### Main Loop

[main.py flow chart view link](https://miro.com/welcomeonboard/YlFpaEFWZ1lrY3oxM3dtc2cwTEVRMHJ1VnJ1c1BnYmJmd2JMREdCUWhRSUY5N2tkd25EOFVtSFB0RlF0bUdHd3wzNDU4NzY0NTk3NzA1ODM5NDE2fDI=?share_link_id=241004072459)

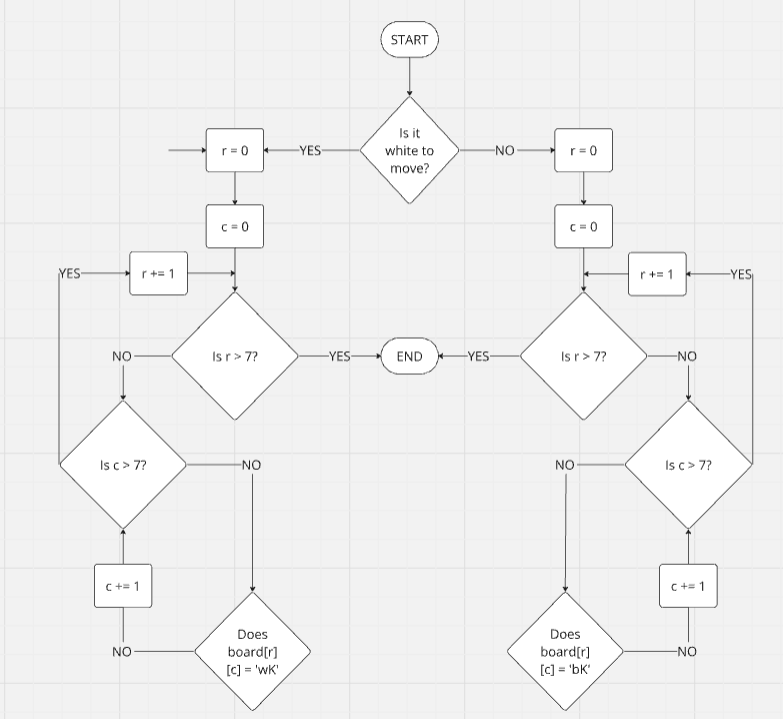
A diagram of a flowchart

Description automatically generated

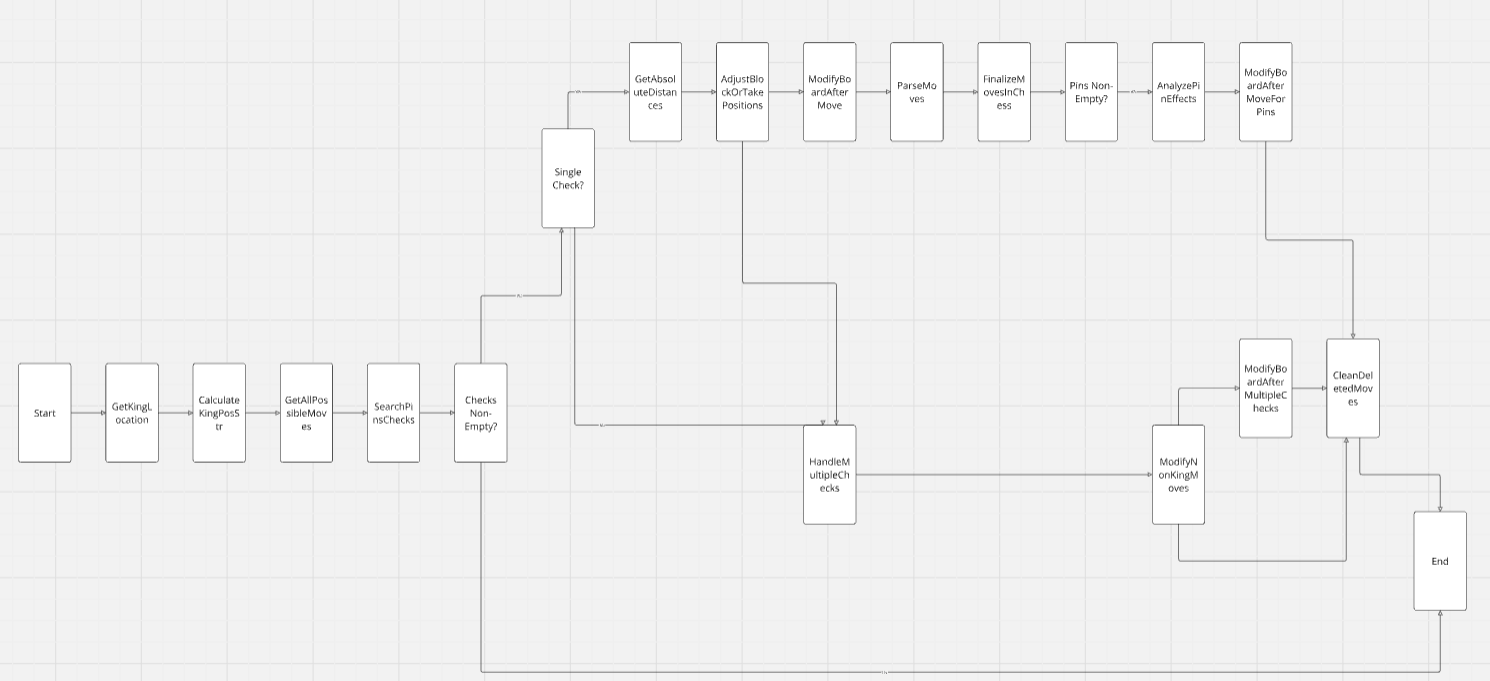
### All Algorithms for Moves.py

Pseudocode can be found in the Pseudocode Folder within write-up

#### Finding the King’s Location



#### Getting All Valid Moves



#### Getting All Possible Moves (Excluding Checks and Pins)

A diagram of a flowchart

Description automatically generated

#### Checking If Castling Is Possible

A diagram of a black king

Description automatically generated

#### All Algorithms for PIece Movement

<https://miro.com/app/board/uXjVKi6uJQE=/>

#### Algorithm for Searching for Pins and Checks

A diagram of a check

Description automatically generated

#### Algorithm for Knight CHecks

A diagram of a algorithm

Description automatically generated

#### Algorithm for Pawn Checks

A screenshot of a computer screen

Description automatically generated

#### Checking for Rook/Queen Checks

Queen checks in this flow chart are horizontal or vertical only

<https://miro.com/app/board/uXjVKidCQyE=/>

A diagram of a flowchart

Description automatically generated

#### Algorithm for Bishop/Queen Checks

Queen checks in this algorithm are diagonal only

A white square with black text

Description automatically generated

### All Algorithms for Graphics.py­­­­­­­­

Pseudocode can be found in the Pseudocode Folder within write-up

#### Drawing the 8x8 Board

A diagram of a flowchart

Description automatically generated

#### Move HIghlighting

A diagram of a flowchart

Description automatically generated

#### Creating the Pawn Promotion Options Square

A diagram of a flowchart

Description automatically generated

#### Drawing the Pieces

A diagram of a flowchart

Description automatically generated

Every part of my solution, as can be seen in the flowcharts above, performs its own unique task and is used alongside several other functions to create the chess engine.

main.py: This is used to hold all the functions and procedures together and make them run in the correct order. It also checks if a selected move is valid and if it is, it adjusts the board array to show the new position.

moves.py: This is a huge file that holds all the functionality needed to find all the legal moves in a position for either side including the removal of any moves that would endanger the King and returns all the moves in an array that can be understood by main.py. I will go over the functions within this later.

graphics.py: This file contains the methods to draw the pieces and the board to the screen. It also highlights squares that each piece can move to when they are selected and creates the square when a pawn promotes

moves.py in detail:

I will need a lot of functions to determine which moves are legal and which put the King in danger. I will go through each necessary function with an overview of what it is used for and why

getKingLocation: This will iterate through each square on the board searching for the King of the player whose move is next

getValidMoves: This will be the main function in this class. It is the one that is called from main.py and will call all the others in the file in the necessary order. In addition, it takes all the feasible moves in the position (by calling getAllPossibleMoves) and any checks and pins (by calling searchForChecksAndPins) and removes any ‘Illegal’ moves (moves that would endanger the player’s own King

getAllPossibleMoves: This iterates through every square on the board. For every piece on the current player’s side, it will run a function corresponding to what it is and stores the potential moves in an array to be altered by getValidMoves

castling: This checks whether or not the current player is able to castle. It does this by checking there are no pieces between the King and Rook, that neither has moved, no hostile piece attacks a square the King would have to move over and the King isn’t in check

get[PIECE]Moves: There are six of these functions. Each one finds the moves for one type of piece and adds them to the possible move array. Pawn moves and King moves have the additional features that they check for En Passant and removing squares attacked by the opponent respectively.

searchForPinsAndChecks: This runs 4 functions to find checks from any piece except the King. Pawns and Knights have their own. Rook/Queen does horizontal and vertical. Bishop/Queen does diagonals.

find[PIECE]Checks: These search for checks from their specific type of piece. If any are found, they are returned in an array.

## Usability Features

### Navigation

I will go for a basic system navigation system to make it as easy as possible for the users to navigate. To achieve this, I will make a menu screen (see below) with ‘Play’ and ‘How to Play’ buttons to take them to the game and to instructions on using the system respectively. How to play also includes an interactive, clickable link to a website that teaches the user how to play chess.

### Consistency

All data used in the game is kept consistent. Being an engine, no personal data is stored, so I don’t need to worry about the Data Protection Act (1998) or GDPR (2018). The only data required is the data needed to get the current game to run properly. This includes the current position on the board, the current colour of the dark squares on the board (customisable by clicking the left control buton) and the piece that has currently been selected.

### User Feedback

I will take user feedback into great consideration when I create the engine. I will ensure that, for every prototype, I get feedback so I know what needs to be improved and/or added.

### Visual Clarity

Everything in the program will be very obvious and easy to understand. When a piece is clicked, legal moves for that piece will be highlighted and the pieces will be easy to distinguish from one another.

### Error Prevention

I can prevent errors by ensuring that any checks of arrays remains within the index limits of the array. For example, if I’m looking for checks on the board, I could make sure that the board co-ordinate (7, 7) is never exceeded.

#### Main Menu

A green sign with black text and a horse head

Description automatically generated

#### Instructions Screen

A green and black text on a green background

Description automatically generated

## Variables and Classes

Here is a list of the key variables and classes and what they are used for:

#### Variables and Data Structures:

width – Holds the width of the screen

height – Holds the height of the screen

row/colSize – A constant to hold how many rows or columns there are (stays as 8 for the whole program)

squareSize – width // rowSize – Holds the length/width of each square

Board – 2D array holding the current position on the board. I used a 2D array here because it allows me to use the first dimension for the columns and the second for rows

moves – Holds all the valid moves for that turn in the same format as in squares to make it easy to see if they match

squares – Holds the coordinates of the squares clicked. If the entire array doesn’t match up to a value in moves, it gets emptied

clicks – Works closely with the squares variable. This holds the number of clicks the player has done. If it reaches two, that’s how the computer knows to check the squares variable

w/bKingLocation – Holds the location of each King on the board to help search for checks and pins

w/bKingHasMoved – Boolean flags that determine whether each King has moved and therefore whether castling is a possibility

inCheck – Boolean flag that shows whether or not a player is in check

checks – Array that holds the current check (or two checks if it is a double check)

pins – Array that holds any current pins

validMoves – A list of every legal move in the position

font – Holds the font for any text required

r/c – Shorthand for any row or column variables

moveLog – This holds all the previous moves (in coordinate notation – [1, 3, 2, 1]

RFMoveLog – RankFileMoveLog holds the previous moves in Rank/File Notation (i.e. Nxc7+). (Post-development note: This would only have been used if I had done an AI. Because of time constraints, I had to stick to a two-player chess engine)

#### Classes:

Main: This holds all the code to run the software, ensure that a move selected is valid and to ensure that the rest of the code within other classes run in the correct order and at the correct times. The reasoning behind this was so that I had a file in which a structure for the sequence of the code was both visible and obvious.

Moves: This class has a large number of different functions and procedures to allow for a sequential process of finding all the possible moves in a given position and narrowing them down to ensure that checks and pins are taken into consideration. This file was kept separate from the main body of the code so that the code was more easily readable and therefore more maintainable. The only reference to it inside the Main class was to call the ‘getValidMoves’ function, which links to every other in the file.

Screens: This contains the code that creates the Main Menu screen and How To Play which appear on startup of the program. This was implemented within my final changes because it came to my attention after some initial client feedback that some people were struggling to understand how to use the software on their own. As a result of this, I added the Menu and How to Play, including an interactive link to a webpage with a very good tutorial on how to play chess. In addition, I added a small README file in the hopes that some users would read that as well, but in the understanding that that was unlikely, hence the need for the How to Play screen.

Graphics: This has 4 helper methods to, once again, make the code more readable and therefore more maintainable. These methods are to draw the board and pieces, to highlight squares that a selected piece can move to and to create a square so the user can select a piece to promote a pawn to.

Engine: I was planning to have more in this class, but it just holds the initial board position and the move logs. It also contains a get method for the board, so that, if needed, I could have multiple instances of the board that don’t affect each other without having to create a separate object.

## Iterative Development Test Data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No. | What is being tested | Expected Result | Actual Result | How I will test this | Actions Needed |
| 1 | Board Rendering | The board shows on the screen | The board showed correctly on the screen | I will run the program and see if the board appears | Running the program |
| 2 | Piece Rendering | The pieces show on the screen in their correct positions | The pieces showed on the screen in their correct positions | Run the program to check whether the pieces are on the board and in the correct places | Running the program |
| 3 | Colour Customisation | When I press the ‘Ctrl’ button on the left of the keyboard, the dark squares should change colour | The colour changed correctly when I pressed ‘Ctrl’ | Press the ‘Ctrl’ button to see if the colour of the dark squares changes to the correct colour | Press the ‘Ctrl’ button |
| 4 | Movement (Limitless) | I can select a piece and a destination with the mouse and it should move the selected piece there, removing any piece that was already there if needed | I selected a piece and a destination and it moved there without any trouble | Move each piece around to random squares. If they move properly, this is working | Move pieces to random squares |
| 5 | Pawn Movement | I should be able to select a pawn and it should be able to move to a square that complies with the rules of chess | I was able to select a pawn and move it to a square that complied with the rules of chess  Try to move each piece, in turn, to squares they can move to, and to squares they can’t move to | Run the program and select a pawn. Try to move it somewhere it can go, then somewhere it can’t | Selection of a pawn and a square it can move to, then one it can’t move to |
| 6 | Added Piece Movement (ignoring check and pins) | All the pieces need to move in the correct way and to the correct squares | All the pieces moved to the right places (pins and check were ignored) |  | Selection of each piece and squares they can and can’t move to respectively. Testing both valid and erroneous data |
| 7 | Finding the King’s Location | The program should return the location (coordinates) of the King, respective of whose move it is | The program correctly returned the coordinates | Print the returned coordinates from the function. If the coordinates match the Kings’ locations ((0, 4) for black and (7, 4) for white), it works | Add a print statement to return the coordinates, run the program to see if the starting coordinates are correct, then move the Kings to see if the new coordinates are also correct |
| 8 | Pawn Checks | If the King is in check from a pawn, remove all moves that leave the King in check and leave any that don’t | Checks from a Pawn were correctly recognised | See if the game correctly recognises a pawn check | Alter the board array so the King is easy to put in check from a pawn, then run the game and see if it is recognised. Test non-check pawn moves to see if they are flagged too |
| 9 | Knight Checks | If the King is in check from a Knight, remove all moves that leave the King in check and leave any that don’t | Checks from a Knight were correctly recognised | See if the game correctly recognises a knight check | Alter the board array so the King is easy to put in check from a Knight, then run the game and see if it is recognised. Test non-check knight moves to see if they are flagged too |
| 10 | Rook and Queen Checks | If the King is in check from a Rook or a Queen, remove all moves that leave the King in check and leave any that don’t | Checks from a Rook or Queen were correctly recognised | See if the game correctly recognises a rook check or a horizontal/vertical check from a queen | Alter the board array so the King is easy to put in check from a rook or queen, then run the game and see if it is recognised. Test non-check moves to see if they are flagged too |
| 11 | Bishop and Queen Checks | If the King is in check from a Bishop or a Queen, remove all moves that leave the King in check and leave any that don’t | Checks from a Bishop or Queen were correctly recognised | See if the game correctly recognises a bishop check or a diagonal check from a queen | Alter the board array so the King is easy to put in check from a bishop or queen, then run the game and see if it is recognised. Test non-check moves to see if they are flagged too |
| 12 | Making Escaping Check Mandatory | Now that the program understands when the King is in check, it needs to be able to narrow down the moves to the ones which allow it to escape check | All illegal moves were removed for each move and all legal moves were kept | Put the King in check from a variety of pieces and check the moves array to see if the only available moves are ones that get the King out of check | I will add a print statement to output the array for moves, and I can check it to ensure that there are no illegal moves in it |
| 13 | Adding Checkmate and Stalemate | The game will recognise either a side winning or a draw by stalemate | Both checkmate and stalemate were recognised and playable | Play through game to get a variety of checkmates and stalemates. If they are all correctly recognised, it works | Play through several games to get to checkmate and to get to stalemate. The relevant message will appear if it is recognised |
| 14 | RankFile Notation Conversion | The latest move in the movelog variable will be converted to chess notation and stored in a separate array | The conversion was done successfully. Castling and Promotion will be added when they are coded in | I’ll check the move log and the chess notation move log and see if they match up | Print both move logs and ensure that they match |
| 15 | Move Highlighting | When a piece is clicked, its possible moves will be shown on screen | Upon being clicked, the potential moves were highlighted with a red outline | When a piece is selected, the squares it can move to should be highlighted | Run the program and select each piece in turn. Then make some moves and make sure that the correct squares are still highlighted |
| 16 | Stopping the King Moving into Check by Taking a Piece | The piece movement functions had to be edited so that removing illegal moves for the King included being unable to take a piece that is defended | The King can no longer take defended enemy pieces | I added this step during the development, when I realised that, because I was reusing the same algorithms, the King’s moves weren’t being amended to prevent it taking a defended enemy piece. I altered them so that, if the Kings moves are being checked, taking a piece and moving into check is removed | Put the King in check from a defended piece that is one square away from the King. I can then select the King to see if taking the piece is highlighted as an option. I can then do the same for an undefended piece |
| 17 | Castling | The King should be able to castle following the general rules of the move | Castling was added successfully and the moves were correctly added to the move log | Get to a position in which castling is possible and try to castle. Also, get to positions in which castling would be possible, but one condition isn’t met to test that it all works | Play the game so that it gets to any necessary position to ensure that castling is only permitted where it should be |
| 18 | Pawn Promotion | Add the functionality so that a pawn at the end of the board can become a piece of the player’s choosing | The pawn promotion algorithm worked and allowed the pawn to change into any given piece (within the limits needed) | Get to a position where a pawn can promote and try changing it, in turn, to each piece (Bishop, Knight, Rook, Queen) | Alter the board array so that a pawn can promote next move. Then I can try promoting it to each piece |
| 19 | En Passant | Allow for the En Passant rule when a pawn moves two squares | En Passant got added to the moves array and is playable | Ensure that en passant is only playable when an enemy pawn has just moved two squares and is horizontally adjacent to a friendly pawn, but is only playable for that pawn | Play through so that en passant is playable, then test if it works |
| 20 | Menu Screen | I added a menu screen so that a ‘How to Play’ screen can be added later | The menu screen worked, the play button took the player to the game and the How to Play button will be coded in due course | Check that the Menu Screen is the first thing to appear upon running the program, and that the buttons work | Run the program to check it runs as expected. Check each button in turn to make sure that they do what is needed |
| 21 | How to Play Screen | The ‘How to Play’ button was coded to take the user to an instructions screen | The button and screen worked correctly and included an interactive link to a site where the user can learn chess | Make sure that the Menu and How-to-Play screens can go back and forth smoothly, and that the link on ‘How to Play’ works correctly | Run the program and press the ‘How to Play’ button, then ESC to go back to the Main Menu. After that, I can go back to the Instructions screen and click the link to check that that works too |

# C: Developing the coded solution

## Development Process

A green rectangular object with a white border

Description automatically generated

This was my first step. I created the Main class to hold the main sequencing for the program and to create the order for the events to happen in. I decided to take an Object-Oriented approach to the program so that there is a definite structure throughout, I can access my various modules quickly, and I can access variables in one file from another easily.

A green and white square with dots

Description automatically generated

The next step was to start creating some of the main variables throughout the program. I called this file ‘Vars’, though it would later be changed to ‘chess\_engine’. I made the self.board

variable as a 2D list so that it can be altered after each move, and so that it has a row/column structure to it to make it easy to access each square on the board. self.whiteToMove determines whose move it is. If it is true, it is white’s turn to move, otherwise it is black’s.

main.py

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

vars.py



I made the new vars file, after moving the previous contents to chess\_engine.py. Although you can’t make a constant in Python, I moved the general constants to a separate file where I can manually change them.

In the main.py file, I made the screen and a dictionary containing the image files (PNGs) for each piece. I then filled the screen in white and made a draw\_squares method to draw the board. This would later be moved to graphics.py and the colour of the dark squares would be cycled through by pressing the left control key.

main.py

A screenshot of a computer program

Description automatically generated

chess\_engine.py (formerly vars.py)

A screenshot of a computer program

Description automatically generated

I had a bug with the board array (as seen above) where I made it a 1 dimensional list by accident, so I changed that to 2 dimensional here.

I also coded the algorithm to draw the pieces on the screen (they are not currently moveable)

main.py

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

I added piece movement by assigning the mouse position tuple to a variable and using the coordinates to work out what, if anything, was clicked. Later, I’ll extend this to check if the desired move is a legal move (involving castling, en passant and pawn promotion)

moves.py

A computer screen shot of a program code

Description automatically generated

A screenshot of a computer program

Description automatically generated

I created a number of dictionaries to convert row-column notation to rank-file notation, as well as creating an algorithm to collect all the possible moves (discounting checks and pins).

# D. Evaluation

<See H446-03 Project Advice Booklet for help and guidance of what must go here.>

# Project Appendixes

Insert as many project appendixes as you need for your project.

These might include, but are not limited to:

* Complete Code Listing (ESSENTIAL)
* Interview Transcripts
* Meeting notes
* Observation notes or questionnaires

# References