Chess Playing Robot using Lego Mindstorms

Raghav Gupta
Nikhil Agarwal
Mentor: Dr. Amitabha Mukerjee

Abstract

The main objective of this project was to create a fully autonomous chess playing robot capable of playing chess with any opponent on any chessboard without requiring any special chessboard for sensors to detect moves of humans. This report describes our approach for building such a robot using the Lego Mindstorms Robotic Invention System kit and a Logitech webcam to capture images which would be used to detect the moves of the opponent. Our approach is divided mainly into three parts. First, detecting the move of the human using image processing. Second, to use a chess engine to generate moves to counter the move of the human. Third, to execute the move generated by the engine using a bot created using the Lego kit.

Previous work

Most of the work done previously on Autonomous chess playing robots makes use of sensory boards to detect moves using magnetic pieces and sensors for example in [1]. This is done with the help of magnetic pieces and sensors kept beneath the board to detect the movement of the pieces. But we prefer to use image processing to detect moves since in that case any simple chess board can be used to play chess and we can play chess in any environment-anytime and anywhere without any special requirements. The paper described in [2] uses a vision algorithm to detect moves and a robotic arm to execute the moves. But using robotic arms to pick up pieces and place them is very inaccurate and unreliable since the gripper cannot be kept exactly vertical at each moment of time which causes difficulty in picking up pieces. Also the design is unstable because of the fact that the gripper along with the arm is very heavy thus destabilizing the system grounded through a single base.

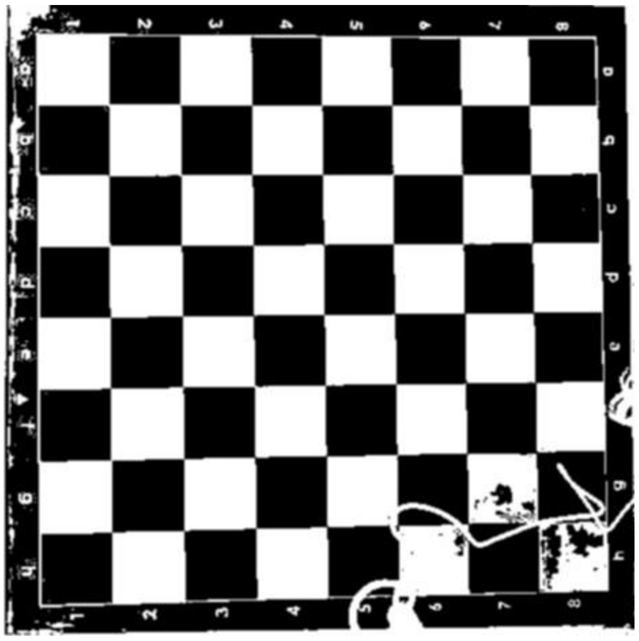
Our Work

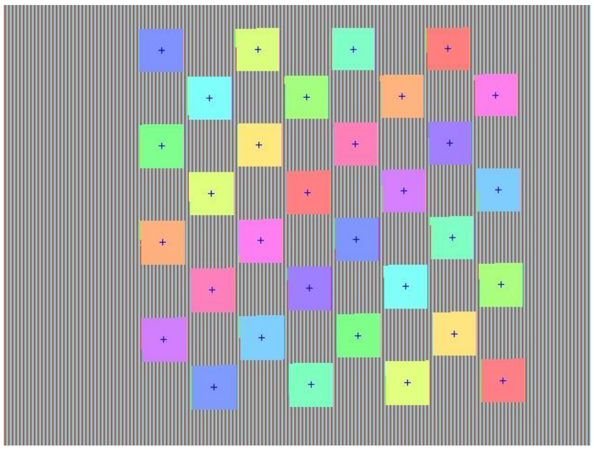
Our work is basically divided into 3 stages:

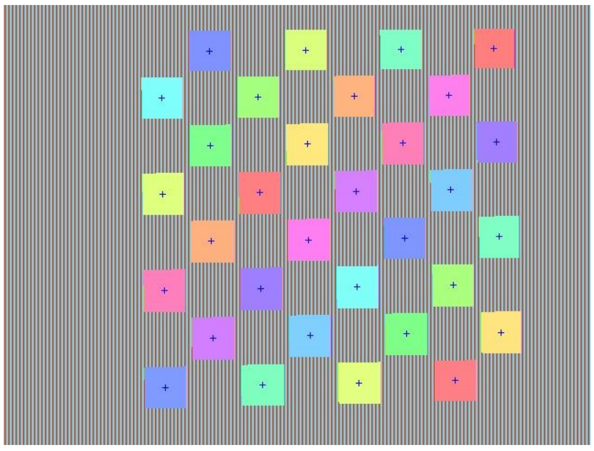
1. Detecting the move using Image Processing
2. Generating the move using chess engine
3. Executing the move

Detecting the move

To initiate the chessboard, we first capture an image of the chess board without the pieces. The first step of our image processing algorithm is to calculate the centroids of the squares of the chessboard to use further in determining the position of the pieces. We do this by first converting the image of the chessboard into binary (for the green and white squares separately). This image is first eroded so that individual squares may be observed separately without mixing to remove since otherwise the squares appear to be connected at the corners. To detect the squares in the image we have used the library "cvBlobs" to detect blobs and label them. Individual Blobs are then labelled and their centroids are stored in an array sorted from a1, a2 … h8 positions.





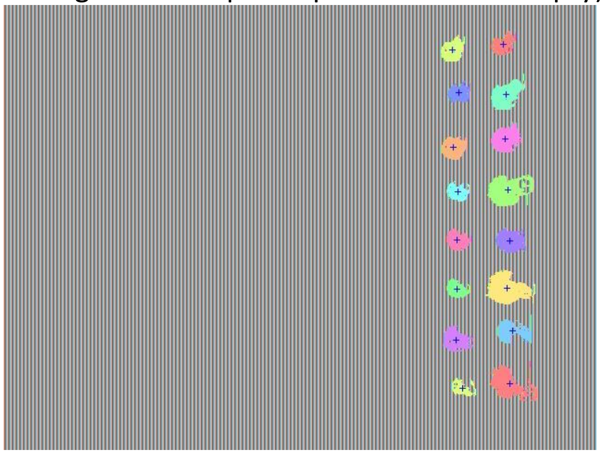


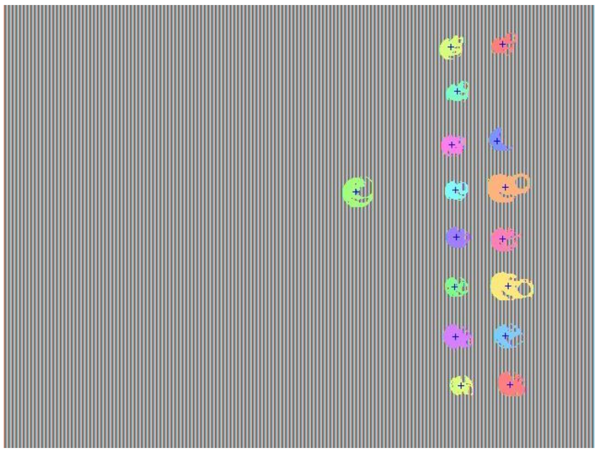
Blobs of White squares with centroids

Blobs of Black Squares

Now, after placing the pieces the game is started and the user is asked to make his move.
After the move of the opponent, another image is captured and converted to binaries of the

white colour of the pieces and the black colour for the black pieces. Again using the blob method the centroid of the pieces in the current configuration are calculated and then compared with the centroid of the squares to determine which square the lie in. This is also done before the move and both configurations are stored. The move is then simply determined by subtracting the two configurations (which are stored as arrays of length 64 having 1 for occupied square and 0 for empty).


Before Move


After Move

This move calculated above is then written into a file which our chess engine reads to calculate a move to counter it. This move is then updated in the array we maintain to keep track of the pieces and then again then the opponent is allowed to make his next move.

The Chess Engine

We are using GNU chess algorithm (version 6.0.3) to calculate the robot moves. GNU chess algorithm consists of three modules which are loosely related to each other:

1. Front end
2. Adapter
3. Engine

The main program consists of frontend and starts two threads one for adapter and other for engine. There are two links:

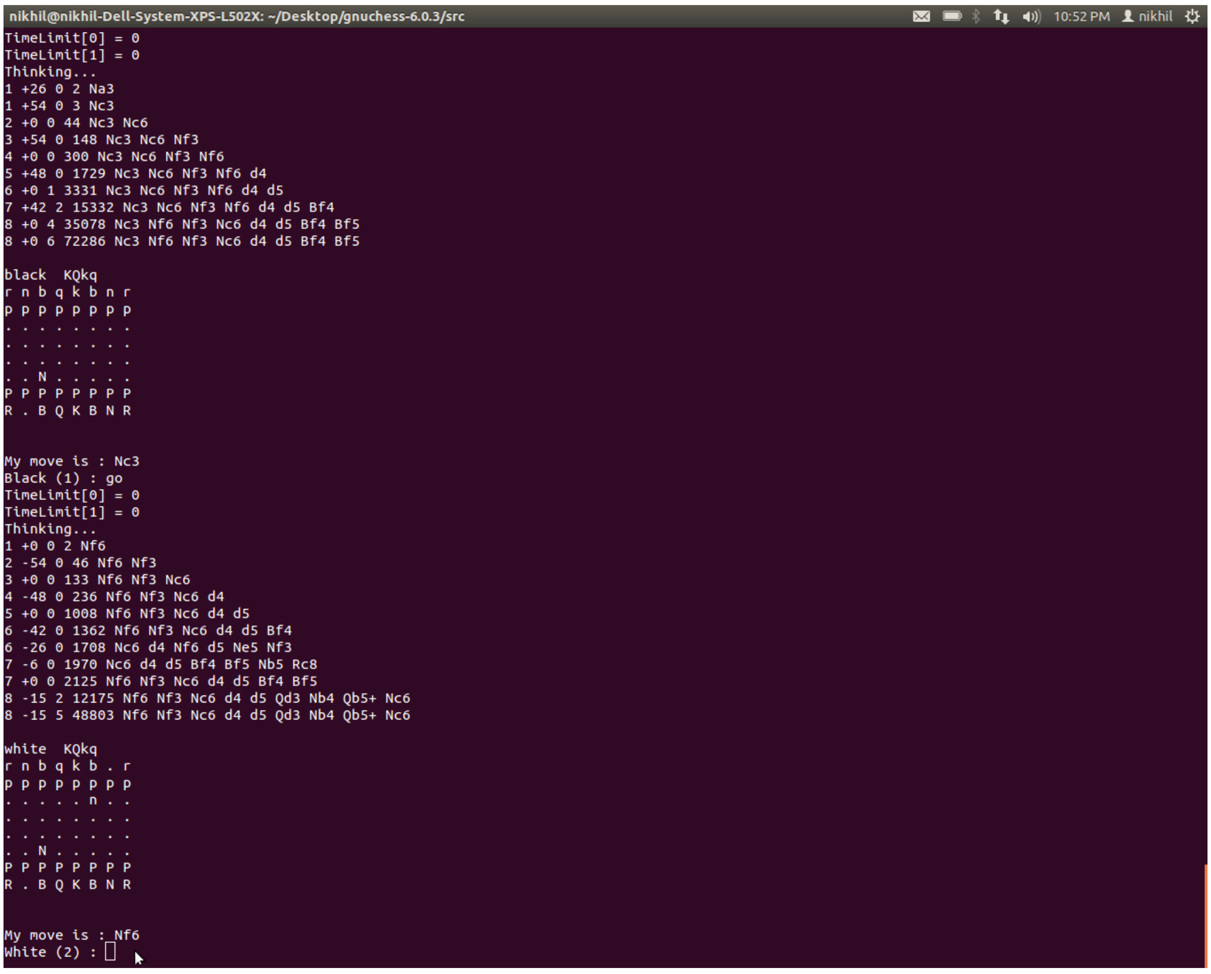
1. frontend <--> adapter
2. adapter <--> engine

The links are based on pipes so no additional synchronisation is required. WE made changes in the frontend of the algorithm. Instead of taking input from a user from terminal we made it to take from the file where user moves are stored.

When user is done taking his move and it has been detected by the robot that user moved from x square to y square it writes it in a file from where the GNU chess algorithm reads it and calculates new move of robot. Algorithm waits until the user has done his move. Now once algorithm has calculated his move it writes it in a file from where the robot reads and executes the move. It take the user input in form of buffer. As soon as user move is available it provides his counter move. If the user move is invalid than it will raise an error and ask user to make valid make and halts until provided with a valid move.

We can also set the thinking depth of the algorithm. Generally the depth we used our robot to play at is Depth 8. We can change the depth of the thinking even between the games. Also we have two modes for playing: easy and hard. In easy mode algorithm doesn't think when user is making his move but in hard mode it make his strategy predicting the user move.

We need to set the mode of play at the starting of the game. Image below shows the algorithm thinking and move taken. Depth set was Depth 8.

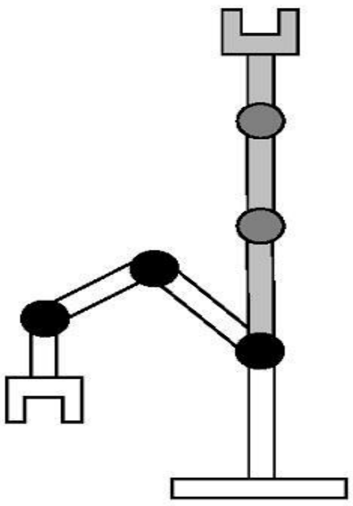


```
nikhil@nikhil-Dell-System-XPS-L502X: ~/Desktop/gnuchess-6.0.3/src
TimeLimit[0] = 0
TimeLimit[1] = 0
Thinking...
1 +26 0 2 Na3
1 +54 0 3 Nc3
2 +0 0 44 Nc3 Nc6
3 +54 0 148 Nc3 Nc6 Nf3
4 +0 0 300 Nc3 Nc6 Nf3 Nf6
5 +48 0 1729 Nc3 Nc6 Nf3 Nf6 d4
6 +0 1 3331 Nc3 Nc6 Nf3 Nf6 d4 d5
7 +42 2 15332 Nc3 Nc6 Nf3 Nf6 d4 d5 Bf4
8 +0 4 35078 Nc3 Nf6 Nf3 Nc6 d4 d5 Bf4 Bf5
8 +0 6 72286 Nc3 Nf6 Nf3 Nc6 d4 d5 Bf4 Bf5

black  KQkq
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . N . . . . .
P P P P P P P P
R . B Q K B N R


My move is : Nc3
Black (1) : go
TimeLimit[0] = 0
TimeLimit[1] = 0
Thinking...
1 +0 0 2 Nf6
2 -54 0 46 Nf6 Nf3
3 +0 0 133 Nf6 Nf3 Nc6
4 -48 0 236 Nf6 Nf3 Nc6 d4
5 +0 0 1008 Nf6 Nf3 Nc6 d4 d5
6 -42 0 1362 Nf6 Nf3 Nc6 d4 d5 Bf4
6 -26 0 1708 Nc6 d4 Nf6 d5 Ne5 Nf3
7 -6 0 1970 Nc6 d4 d5 Bf4 Bf5 Nb5 Rc8
7 +0 0 2125 Nf6 Nf3 Nc6 d4 d5 Bf4 Bf5
8 -15 2 12175 Nf6 Nf3 Nc6 d4 d5 Qd3 Nb4 Qb5+ Nc6
8 -15 5 48803 Nf6 Nf3 Nc6 d4 d5 Qd3 Nb4 Qb5+ Nc6

white  KQkq
r n b q k b . r
p p p p p p p p
. . . . . n . .
. . . . . . . .
. . . . . . . .
. . N . . . . .
P P P P P P P P
R . B Q K B N R


My move is : Nf6
White (2) :
```

Executing the Move

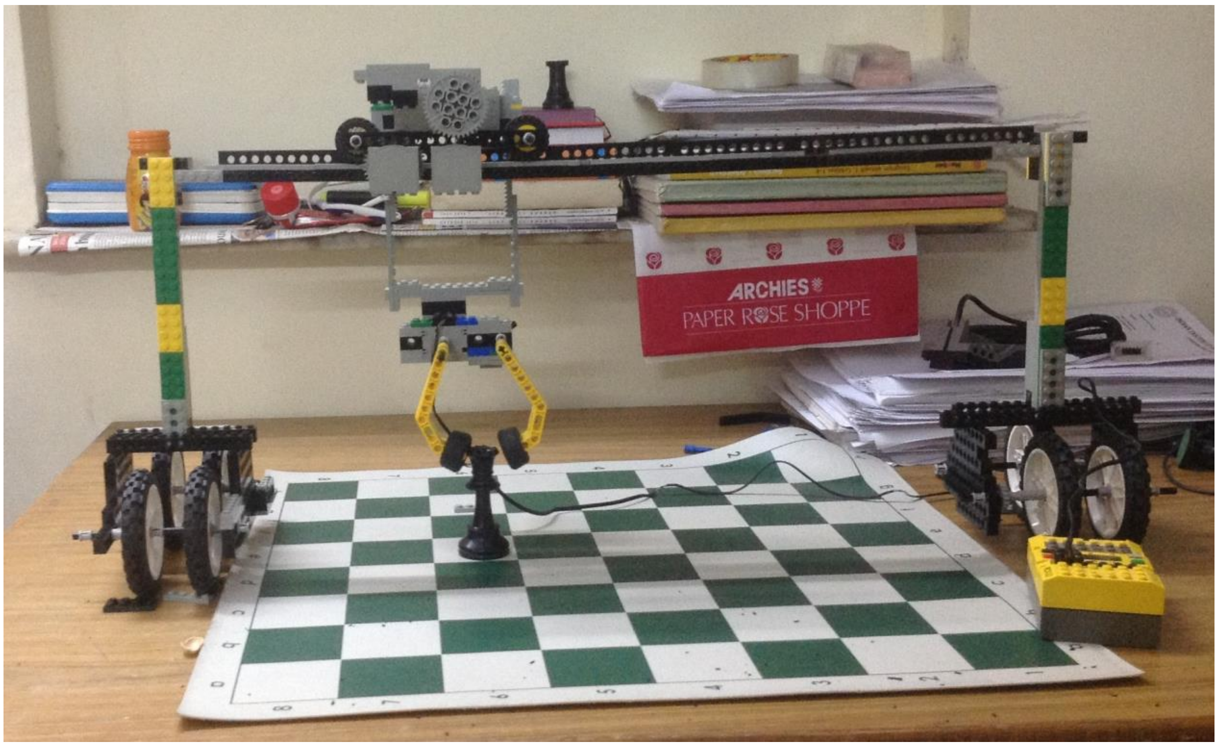
1. Design of the Bot

Initially we had started with the idea of making a robotic arm with the structure shown in the image below, to pick up pieces and put them on the required position. The arm was supposed to be mounted on a base that could move along a set of rails along the edge of the chessboard. The other movements to be controlled were at the three hinges to move the arm and a gripper to pick up the pieces.

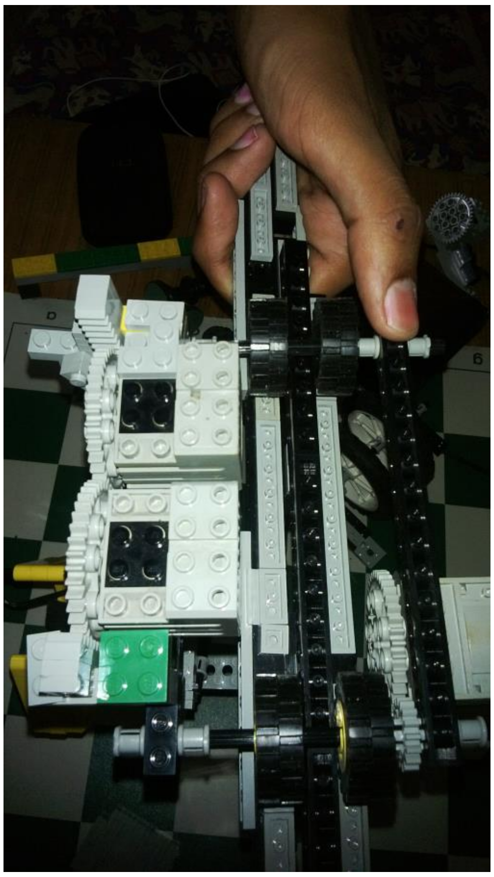


But, the above design was unstable and unreliable since it was difficult to bring the gripper at the exact position and vertically above the piece to pick it up. Also the design was unstable and prone to falling forward due to the heavy weight of the arms and the gripper.

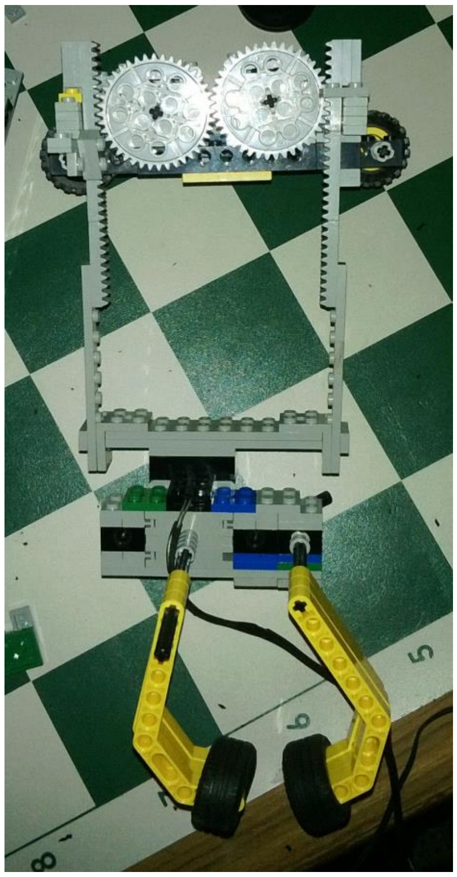
To counter the above drawbacks we came up with another design that consisted of a bridge like structure with the supports lying on two bases, capable of moving laterally along the edges of the chessboard. This constituted the forward/backward motion of the bot. The gripper was suspended from this bridge using gears and motors were attached to control the up and down motion of this gripper to pick up a piece and lower it.



The body of the gripper moves on the bridge on rails so that the motioned is constrained and it doesn’t fall off. The gripper consists of two fingers one of which is fixed to the body and the other is moved through a motor to grab the pieces.



Gripper moving on tracks



Model of Gripper

This entire setup (the gripper, the rails and the bridge) are attached to two bases which consist of 4 wheels and move along the edge of the board to accomplish the forward motion of the bot.

2. Coding the RCX Brick

The Lego Mindstorms Robotics Invention System contains a RCX brick which controls the motors in the system and can be programmed from a PC using a USB IR tower provided by Lego. The software that Lego provides to programme the brick is a graphical interface which does not allow full control over the system. To gain full control over the system we searched for tools to programme the brick according to our own needs using C/C++ or other similar languages. We were able to come up with two solutions.

One of the solutions is to install an alternative firmware on the brick so that we upload programs of our choice onto it. It is called brickOS and can be installed easily onto the brick using Cygwin on windows and a package to transfer files to the brick. Instructions on how to install can be found in [3] and [4]. This also requires installing a gcc-H8 cross compiler since the Lego brick consists of a Hitachi H8 compiler and ".c" files cannot be used to control it. The cross compiler compiles the files into ".lx" format which can then easily be transferred onto the brick (again instructions can be found in [3]). This enables us to fully control the brick according to our code and operate it as required. But the only drawback of this solution is that it does not enable real time control over the bot meaning that to execute each move, we would have to program the bot again each time and hit run on the brick to execute the move.

The other solution to the problem that we found was using a software called bricx Comand Centre (bricxCC). This program enables us to achieve real time control over the brick and also allows us to send messages to the brick using the IR tower while also sending messages to another RCX brick from the original brick to enable more control. The language we used for this was NQC (not quite C) information for which can be found in [5] and tutorials for programming in NQC can be found in [6].

Results

Due to some problems, we could not use the 2nd method described above and had to create different codes for different moves and hardcode them according to the move predicted by the chess engine and transfer them to the RCX. In the end we were able to make the bot execute certain moves which can be seen below.

Example Moves.

1. Move from c6 to d5 and back - Combined motion
2. Move From c5 to c6 to show Horizontal motion
3. Move From c4 to d4 to show Forward Motion

Major Challenges

The biggest challenges that we faced when we took up this project was to be able to find the drivers and software for the kit for windows 7. The drivers were only available for windows 98 and NT. We also find some tutorials to use the kit on windows XP. But we had to do a lot of research to find drivers compatible with windows 7. What we found was that the drivers for the USB IR tower are not compatible with any of the 64 bit versions of windows. The serial USB tower on the other hand that Lego provides can be used to work with the 64 bit version of windows (using a serial to USB converter to use in our laptops). So we installed windows 7 32 bit on our machine to work with the machine. We found the drivers online to work with this version of windows and were able to make it work just fine! The other problems were finding documentation to program the brick using C. After a lot of research we were also able to find the tutorials listed above to program the brick according to our wish.

Another major challenge was designing the bot such that to give it maximum stability. After experimenting a lot with the arm structure we finally concluded that it was unstable and unreliable. So we switched to the bridge structure described above.

Also the image processing part is affected by lighting conditions and has some probability of error, so we had to use various error checking techniques in our code to avoid incorrect inference from images and store only the correct data for centroids etc. from the images, since a lot of times noise from the background may also come into the picture and cause bugs in the program.

References

[1] Autonomous Chess Playing Bot by Timothee Cour, Remy Lauronson and Matthieu Vachette

[2] Lego Chess Bot by Stewart Gracie, Jonathan Mathey, David Rankin, Konstantious Topoglidis

[3] http://brickos.sourceforge.net/docs/INSTALL-cygwin.html

[4] USING BRICKOS WITH LEGO MINDSTORMS RCX BRICK AND ESTABLISHING INFRARED COMMUNICATION by Tero Mäkelä

[5] http://bricxcc.sourceforge.net/nqc/

[6] RCXCC Manual by Mark Overmars