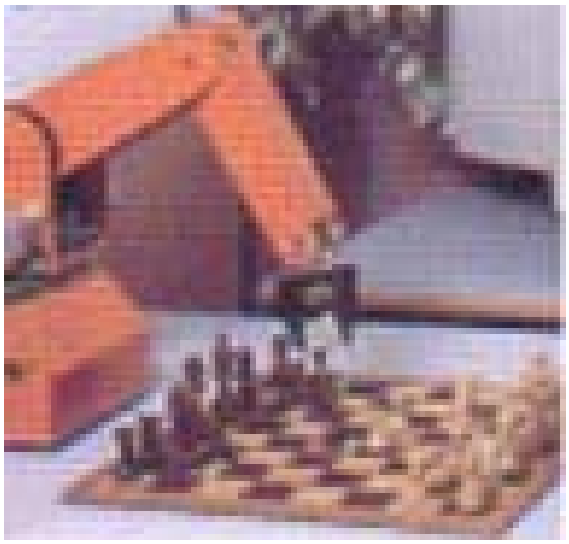
Autonomous Chess-playing Robot

Timothée COUR
Rémy LAURANSON
Matthieu VACHETTE



ECOLE POLYTECHNIQUE, **july 2002**

Table of Contents

INTRODUCTION

The project we are presenting in this paper was realised during the senior year of undergraduate studies at the Ecole Polytechnique. We chose to create an autonomous chess-playing robot, involving vision, robotics, and a chess algorithm. The robot could play autonomously a whole game against a human opponent.

This paper is divided into four parts: the robot, the vision algorithm, the chess engine and the main program. The following points are addressed:

- How to make the robot play a move, given the physical constraints (the actuators and the sensors have a limited accuracy, some locations are out of reach of the robot arm, and the robot pliers are quite thick) ? How to convert a chess move into a sequence of instructions using Cartesian coordinates, and then into control voltage for each motor ?

- How to see the chessboard, given the technical limitations (the camera has a limited field of vision, the images are of medium quality, and the chess pieces are seen from above). The vision algorithm must be robust enough to compute the position of each piece under various conditions of luminosity and board positioning. However, computations must be fast enough to play a real-time game.

- How to compute the « best » move from a given position ? The most important constraint here is time, therefore a good algorithm must be capable of travelling through the tree of variations in an efficient way.

Finally, we had to connect the three parts (the robot, the vision and the chess engine) into a single device. We chose to create the main program in *Java*, because this programming language is more adapted to create complex programs. We had to deal with the interface between *Java* and *Turbo C*, so as to communicate with the actuators and the sensors of the robot, and with the camera.

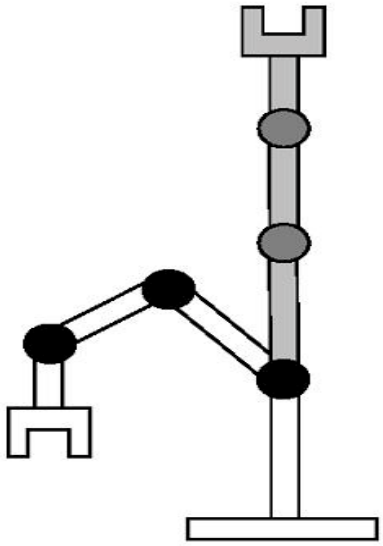
PART I: THE ROBOT

Description of the robot

1 The robot parts

(1) The robot motion

The robot has 4 degrees of freedom (a vertical axis to allow rotations in the horizontal plane, and 3 axes for the arm itself). The axes have roller bearings, so as to restrict friction during movement. The robot pliers can have only 2 states : either opened or closed.



(2) The actuators

The robot is controlled by 5 motors: one on each axis and one for the pliers. They use direct current, and the interface allows us to apply a voltage between –10 and +9.99 Volts.

(3) The sensors

The state of the motors is given by four potentiometers, which convert the angular positions into readable voltages. However, they are not very accurate, and may lead to an error of several millimetres on the position of the pliers.

The potentiometers are really the major defect of the robot. Indeed, playing chess requires an accuracy of about the diameter of a pawn, therefore we have to use big pieces and a big chessboard in order to play accurately. As we will see, this forces us to use a reduced chessboard (8x6 squares instead of 8x8 squares). The use of precise sensors (with an error of about 1/10 000th of degree) would have been a better solution.

2 Interfacing with the computer

The interface between the computer and the robot is made by an ISA card controlled by a driver, and connected to the computer bus. Basic I/O functions are available to read the sensor inputs and write the actuator outputs: ADC (Analog to Digital Converter), DAC (Digital to Analog Converter), plus a function controlling the opening and closing of the pliers.

The robot control

1 Choosing the right coordinate system

We chose to work in a Cartesian coordinate system to locate the robot pliers, so as to simplify high-level tasks (in Java). Indeed, chess pieces are located using a 2-D coordinate system (example: the white king is on "e1" at the beginning of the game), and it is easier to compute the sequence of instructions for the robot using xyz coordinates rather than angle coordinates.

2 Planning the motion

The robot command uses the DAC (Digital to Analog Converter) function to send a signal to the designated motor. We have therefore created a function that converts a desired position of the pliers into a list of angles for each axis, and then into voltage. It is written below in pseudo-code.

```
position_to_voltage(x,y,z) {
   // robot characteristics
   a = 11.7; // length of robot segment n°1
   b = 15.4; // length of robot segment n°2
   c = 13.5; // length of robot segment n°3

   // intermediate computations
   r = √( x² + y² + (z-c)² );
   u = ( a² - b² + r²) / (2 * r);

   // angle conversions for each axis
   g = atan((z-c)/√( x² + y² ));
   θ₀ = atan (-x/y);
   θ₁ =acos ( √( a² – u² ) /a ) -g;
   θ₂ =acos ( -√( a² – u² ) /b ) -t1 -g;

   // voltage conversions for each motor
   control_voltage [0] = 0.02 + 3.3 * θ₀ ;
   control_voltage [1] = -2.4 + 3.3*(π/2 - θ₁) ;
   control_voltage [2] = -10 + 3.3*(π-θ₂) ;
   control_voltage [3] = θ₁ + θ₂ - π) / 0.3;
}
```

The voltage that was just computed on each track must be thresholded and normalised before it is effectively sent to the 4 motors. The inverse of these functions have to be computed as well, for the feedback in the servo-loop control.

3 Communicating the instructions between Java and Turbo C

The robot is commanded from the main (high-level) Java program, and has to communicate with the low-level Turbo C program. Because of the hardware configuration, we couldn't use the *Java Native Interface* usually used to communicate between a C program and a Java program. Indeed, our Java program couldn't interact with DOS. Therefore, we relied on another solution (much more time-expansive), which used simple text files. Such a file contains the sequence of instructions required to play a chess move. Each time a new move is required, Java writes the file and then Turbo C reads it. The instructions written in the file use the grammar described in the following example:

c_2.0_17.0_15.0/p_0/c_2.0_17.0_14.0/c_2.0_17.0_13.0/c_2.0_17.0_12.0 ;

Instructions are separated by the a "/" character. The end of file is mentioned by the " ; " character.

The instruction "c_2.0_17.0_15.0" has the following elements :

- The type of command ("c" = movement command, "p" = pliers command).
- If the instruction starts with a "p", "0" means "open the pliers", whereas "1" means "close the pliers".
- If the instruction starts with a "c", the numbers refer to the targeted xyz coordinates.
- The different parts of an instruction are separated by the " _" character.

We also made an elementary syntax analyser in Turbo C so as to read the file that Java created.

4 General command of the robot in Turbo C

The Turbo C program analyses the file written by Java, initialises the command parameters, and then executes the sequence of instructions. First, the opening/closing of the pliers is executed, if required by the first instruction. Then, for each one of the following instructions, voltage is constantly applied to the four other motors in order to minimise the distance between the desired angle and the measured angle of each robot actuator. A thread creates a time counter which allows to compute new values of the voltage at each tick of the clock (every 10 milliseconds). When all the robot's angular positions are near their desired value, the next instruction is read, and so on.

When the last instruction is executed, the robot is abandoned and switched off. That is why it must be left in a position that is not dangerous for the chess pieces (it must not fall on the chess board).

We have kept the Turbo C program quite simple, leaving all the high-level tasks to the Java main program. Thanks to this, Java and Turbo C only have to communicate once per move.

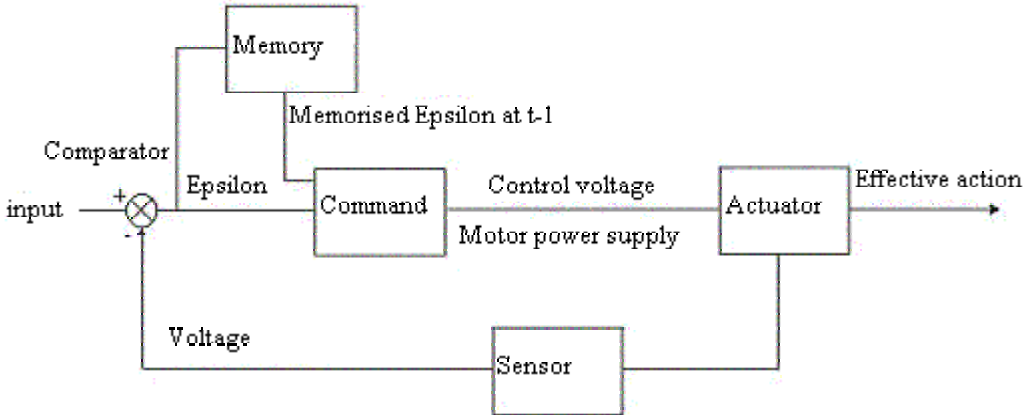
5 Servo-loop control

A feedback is used in order to increase the speed, accuracy and smoothness of the movements. At any time between two instructions, the robot tries to keep the same position even if some forces are applied to it. For this purpose, input and output of the actuators must

be constantly compared. The disparity estimates obtained by the potentiometers are used in a feedback architecture described below. At each time step (every 10 milliseconds), error signals are computed and provide an input for motor control. The time gaps are short enough for the command to be smooth.

Our first choice was to use a linear control, with a simple gain. This configuration gave acceptable results, but the system was unstable (high-frequency vibrations) for high gain values, and otherwise was too slow. Therefore, we used an additional derivative term, and headed for a proportional-derivative control. This enables much faster movements without any loss of precision.

The servo-loop control is represented on the figure below.



The input is the voltage measured by the potentiometers. The epsilon term is equal to the difference between the measured and the desired voltage. The control voltage is given by the equation below:

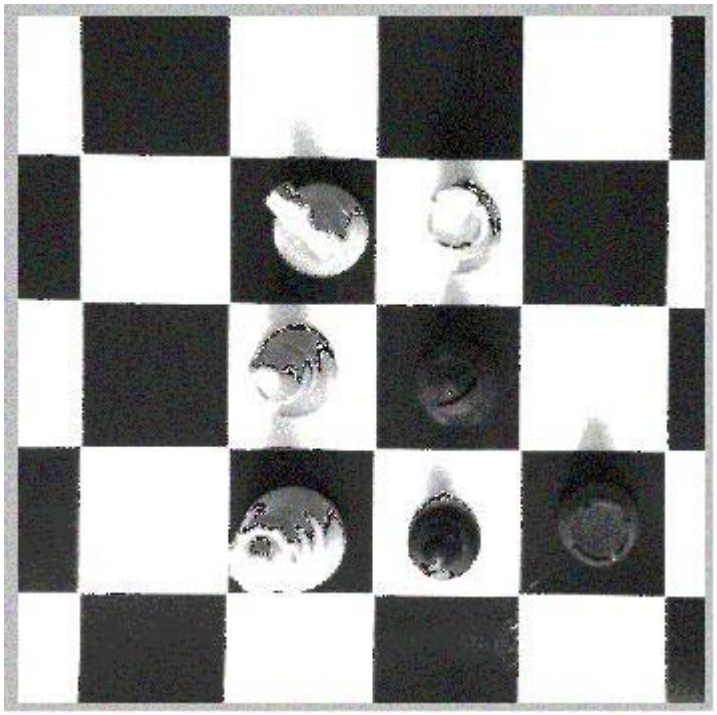
$$control\_voltage = Gain * (epsilon(t) + Td\ x\ (epsilon(t)-epsilon(t-1)),$$

Where Gain and Td are parameters that affect the speed and stability of the system.

PART II: THE VISION ALGORITHM

Presentation of the problem

A fixed camera is focused on the chessboard. It takes black and white pictures of medium quality (512x512 pixels, with 256 grey levels) and stores them in data files. The camera is commanded by Turbo C (under DOS), therefore Java must once again communicate with Turbo C in order to take pictures.



example of an image taken by the camera

The purpose of the vision algorithm is to process an image provided by the camera, so as to recognise a chess position (in terms of chess coordinates for each piece on the board). More precisely, the system must be able to perform the following tasks:

- Recognise the chess squares and give their Cartesian coordinates in a robot-centred frame of reference.
- Detect the presence of a piece on a given square
- Recognise its type (Queen, Pawn, etc.) and tell if it is black or white

Recognising the pieces

Seen from above (as is the case here), a chess piece is difficult to recognise, given the medium quality of the camera. The worst case is when a black piece stands on a black square: in this case it is very difficult to make the difference between a bishop and a pawn. As the pieces differ only by their diameter (seen from above), a direct method of recognition would entail a very accurate image processing algorithm.

This obstacle can be bypassed with the following argument: one can deduce the position of each piece after the n^{th} move by simply comparing the position of pieces after the $(n-1)^{th}$ move and the position of undetermined objects after the n^{th} move (hereafter, an *object* will designate a chess piece whose type hasn't been recognised yet, whereas a *piece* refers to a chess piece whose type is known). The only relevant information is the colour of these objects, so as to deal with the case where a piece can take two possible opponent pieces. The diagrams below illustrate this situation:

after the $(n-1)^{th}$ move

after the n^{th} move:
pawn on d4 takes pawn on e5

After the n^{th} move, we only know the position and colour of each object, but not their type. If the colour of the object on e5 were unknown, it would be impossible to know if the white pawn took the black pawn on c5 or on e5.

Remark: in fact, to be perfectly rigorous, a rare situation may arise in which this method is unable to determine the position. This is the case of under-promotion, when a pawn advances to the last rank and promotes to something else than a Queen (a Knight, a Bishop, or a Rook).

A problem of classification

The whole vision task consists in two steps:

- segmentation of the image into chess squares. After this, the issue of piece recognition all over the board is uncoupled into 8x8 (or 8x6) independent atomic problems
- detection of an object on a given square, and recognition of the type of the piece. According to the last section, the only required task is to detect whether an object lies in a given square, and to tell its colour.

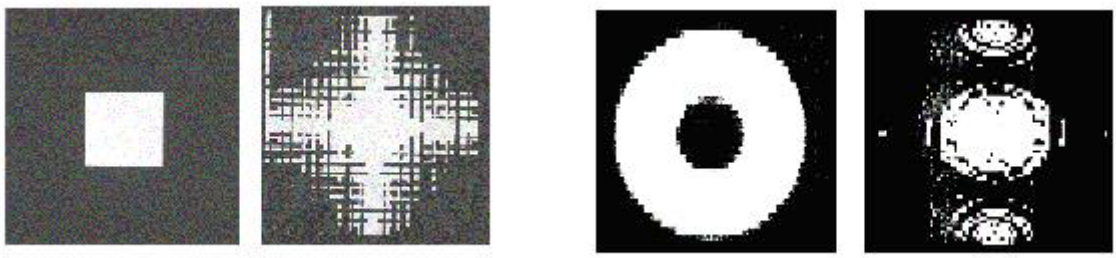
These are typically problems involving classification, and can be addressed by general data processing algorithms. The aim is to classify high-dimensional data (for example, the image of a square and a possible object lying on it) into a discrete (and small) set of categories (for example, “empty black square”, “white piece on a white square”, etc.).

An efficient classification depends on the choice of pertinent data descriptors. Initially, the only available data is a pixel matrix of grey level values.

1 Choice of a method of classification

- Spatial frequency analysis

This method is quite fit to our problem, and could also have given good results, as is shown by the four images below. On the left (right) we can see a square (circle) and its 2D Fourier transform. The left might represent a chessboard square, while the right might represent a piece. The 2D Fourier transforms are very specific in those cases, and thus this method provides good data descriptors.



With the Fast-Fourier Transform algorithm, this method would have been possible (given the time constraints), but it seemed to us that we could use more basic descriptors.

- Neural network

We had initially planned to use a neural network for our recognition task. However, we usually use these in situations where it is difficult to build efficient “handmade” descriptors, which is not the case here.

- Principal components analysis

This technique is often employed in classification tasks. It requires an important database of examples to tune the different parameters. In our case, we are working with a training set of only two images: an empty chessboard, and a chessboard with pieces showing all types of configuration (black piece on white square, etc.).

- Filtering and template matching

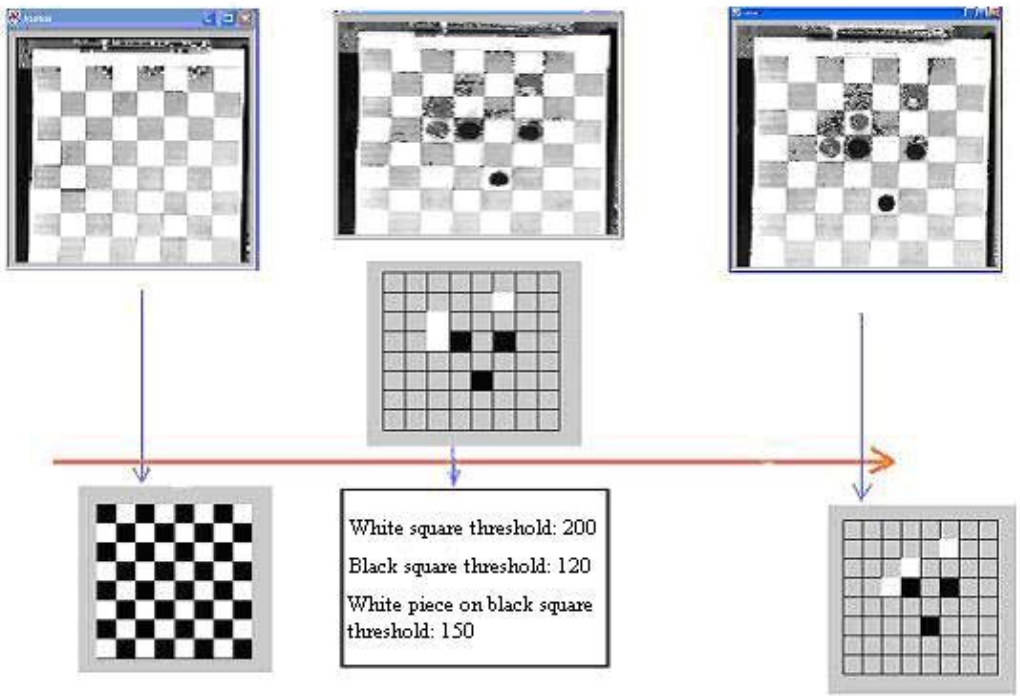
The method we used is simpler than the methods mentioned above. It tries to tackle the problem of classification in a more straight-forward manner, using all the information that is available. In addition, it is time-efficient as it uses only local processing.

General description of the vision algorithm

Once the image is segmented into individual chess squares, objects are detected by applying thresholds to adequate descriptors. This section deals with the following issues:

- Segmenting the image of a chessboard into chess squares. The length of the chessboard is unknown and the borders may not be perfectly parallel to the borders of the image. They may even be invisible in the image.
- Choosing the right filters and templates, and initialising the thresholds
- dealing with various light conditions and with damaged images (the images taken by the camera sometimes had dark spots that made the recognition process difficult)

The vision algorithm is summarised below:



1st step (before the game): chessboard segmentation

2nd step (before the game): initialisation of the different thresholds

3rd step (during the game): building a matrix of type {empty, white, black}

Before playing a game, a picture is taken of the empty chessboard and used to recognise the black and white squares without being disturbed by the pieces. Then, the user is asked to set the position he wants to start with on the board, and on the computer as well (via a window interface). A second picture is taken to initialise certain thresholds. After this the game can begin, and no entering of moves on the computer is required.

1 Detecting the chess squares

As we worked with various types of chessboard to adjust its length and colour, we wanted the vision algorithm to adapt to any kind of chessboard. More precisely, the initial chessboard segmentation must be capable of recognising the squares under the following situations:

(1) various square sizes
(2) various chessboard orientations
(3) various light conditions
(4) some squares may be partially occluded, especially if the image is too small
(5) some parasite spots may appear on the image

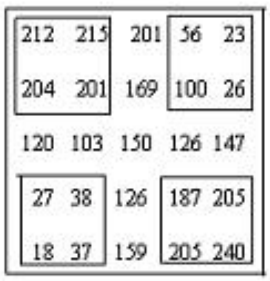
The segmentation must provide a matrix of 8x6 squares with the following fields:

- position (coordinates of the upper left corner of a square, in a robot-centred frame of reference)
- colour of the square (black or white)
- size of the square

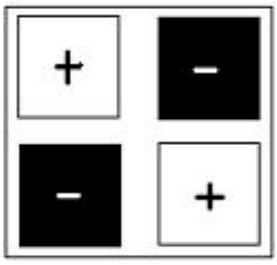
This information is sufficient to locate any piece and give its chess coordinates later on.

Our first idea was to compute the gradient of the image in the horizontal and vertical directions, and extract lines of high contrast. Alas, this method was unable to deal with the situations (2) and (4) above.

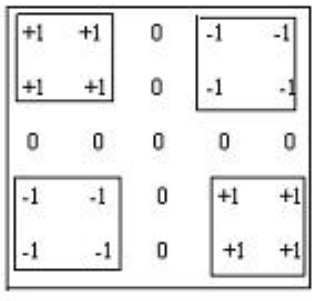
We therefore used a method based on template-matching. The templates are supposed to detect the corners of the chess squares, as shown below:



Typical values of grey levels around a corner



A corner in between four squares



The 5x5 coefficients used in the one of the templates

The coefficients of the template in the two median lines are equal to zero because grey level values are seldom significant at the frontier between two squares. High values of the filtered image denote the proximity of the upper-left corner of white square (or the upper-left corner of a black square, with a similar template). As we only want one corner per square, we take the local maximum each time there is more than one point of high response to the filter. This

simple filtering method gave very good results under the situations (1), (2) and (4) mentioned above.

The situation (5) is tackled with a simple smoothing filter, and the situation (3) is dealt with by using a threshold, which sets low grey values to 0 and high values to the maximum.

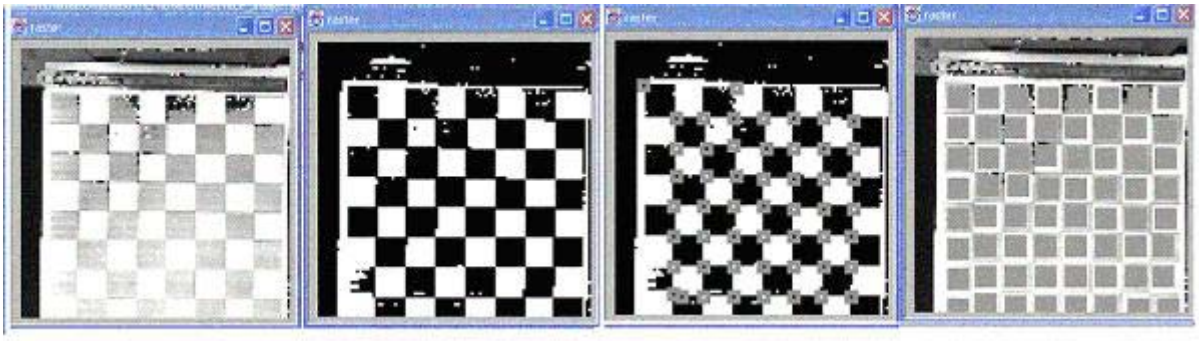
2 Construction of the chessboard

Before the construction of a matrix of chess squares, we must deal with the fact that :

(1) we only have a list of corners, not squares, and they are not ordered in a matrix
(2) corners on the border of the chess board are usually not detected (the template uses four squares to detect a corner)
(3) some other corners may not have been detected, due to light conditions
(4) false corners may appear, due to spots

The program searches for maximum alignments of corners in the horizontal and vertical directions (allowing for a slight error and a slight angle shift). Corners that do not fit in those alignments are eliminated. The average distance between two consecutive horizontal lines is computed, and gives the size of the squares. The lines found separate the image in as much regions as squares on the chessboard, which allows the construction of the matrix of squares that was required.

The four images below show the processing steps required for the segmentation:



1. initial image | *2. smoothed and thresholded image* | *3. filtering to detect corners* | *4. chessboard segmentation*

As we can see on the third image, the situations (2), (3) and (4) have occurred. On the fourth image, however, every square is detected, and no false square is detected.

The complete segmentation algorithm gave very good results. Let us now look at the piece recognition algorithm.

Analysis of the content of a square

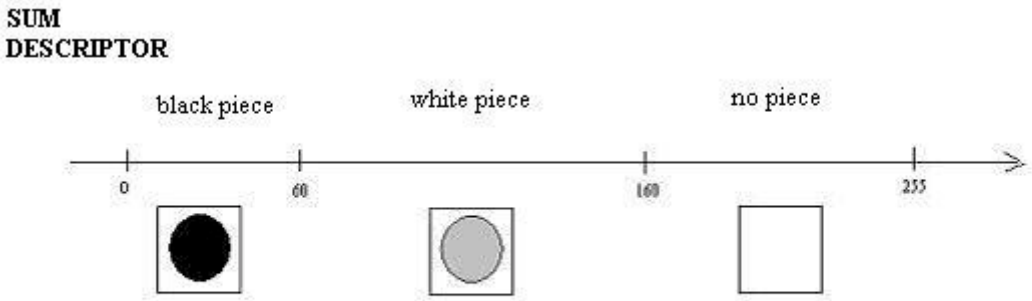
We assume here that we have already segmented the image into a matrix of squares. We are now analysing one of these squares.
This step must answer to the following questions:

(1) Is there a piece on the square ?
(2) If yes, what colour is it (as stated before, only the colour is needed, not the type) ?

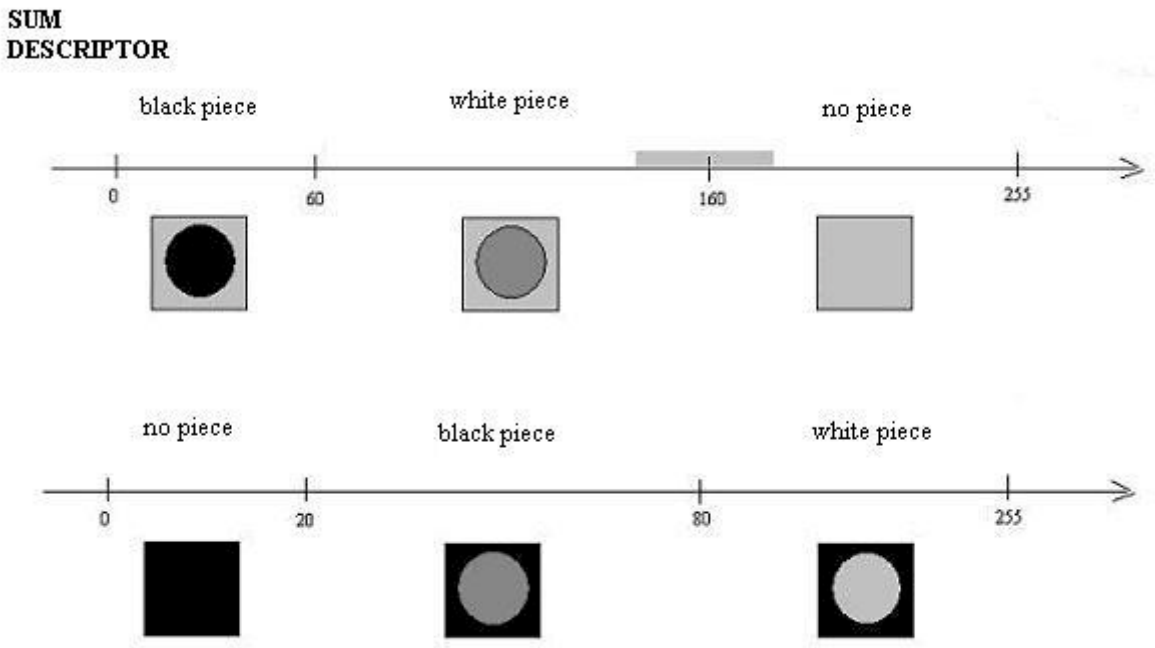
Both questions are answered at the same time by using thresholds on two square descriptors.

1 The "sum" descriptor

The average grey level value is computed in the square. The following figure shows the results for a white square with a black piece, a white piece, and no piece at all (typical threshold values are written).



The problem is more complex for a black square, where two situations may occur:



We chose to work with chessboards with rather light dark squares (printed chessboards), so as to be in the first situation. This descriptor gave acceptable results, but some mistakes occurred when the light conditions changed (it worked in the morning but not at noon, for example) or when the light intensity varied over the board). To deal with this situation, we use an additional descriptor : the difference descriptor.

2 The "difference" descriptor

This descriptor measures the difference between two consecutive "sum" descriptors on a given square.

Let "a" be the average value of the "sum" descriptor for a black square with a *black piece*.
Let "b" be the average value of the "sum" descriptor for a black square with a *white piece*.
Let "c" be the average value of the "sum" descriptor for a black square with *no piece*.

Under the assumption that white pieces are darker than black squares, we he have:

$$(1) \qquad a < b < c$$

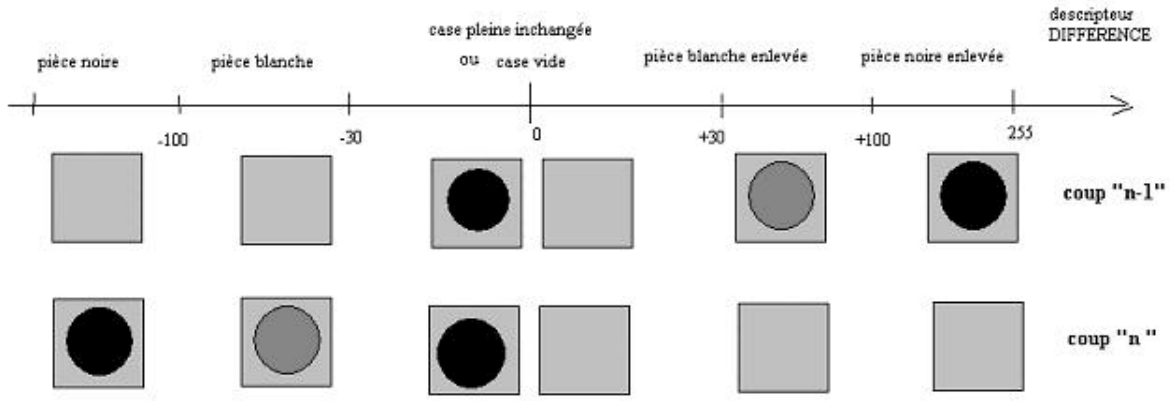
The following table gives the values of the difference descriptor:

(n-1)th move \ n^{th} move	a	b	c
a	0	b-a	c-a
b	a-b	0	c-b
c	a-c	b-c	0

Equation (1) gives:

$$(2) \qquad a\text{-}c < a\text{-}b < 0 < b\text{-}a < c\text{-}a$$
$$(3) \qquad a\text{-}c < b\text{-}c < 0 < c\text{-}b < c\text{-}a$$

For example, the situation (3) is illustrated in the figure below:



The only question is to know whether $a\text{-}b < b\text{-}c$ or $a\text{-}b > b\text{-}c$.
In fact, we don't even need this information, because the position after the $(n\text{-}1)^{th}$ move is supposed to be known, so there is no ambiguity.

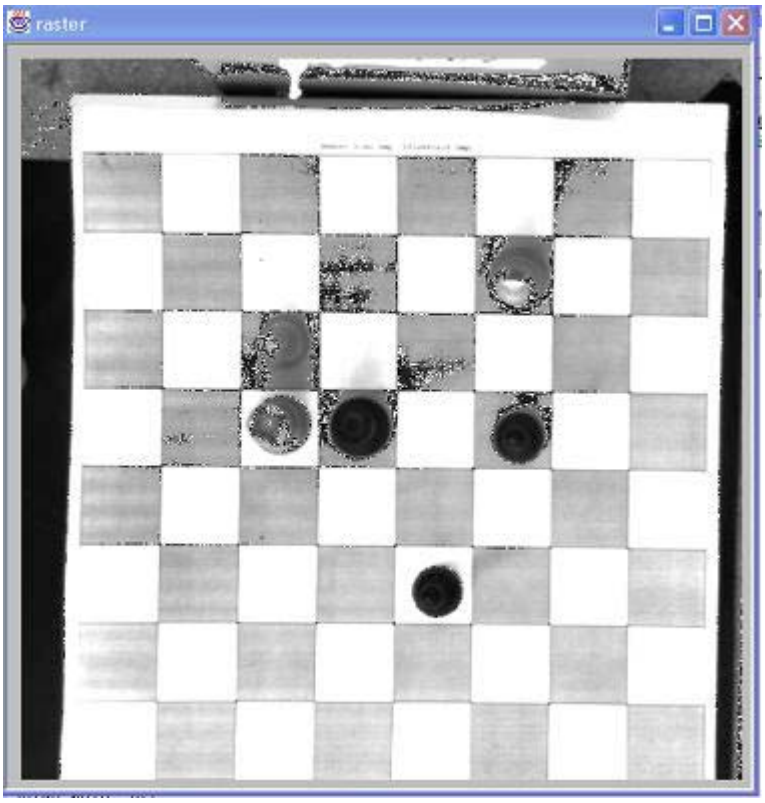
The “difference” descriptor gives a good estimation of the position whatever the light conditions. When there is still an ambiguity, we can use the fact that only two squares may have a non zero value for the difference descriptor (there is also a constraint on the colour of the piece that moved).

3 Fixing the thresholds

Thresholds are fixed thanks to the first image of the chessboard, before the beginning of the game. A Window Interface asks the player to enter the position on the board. “Sum” and “difference” descriptors are computed for each situation (black piece on white square, etc.) and thresholds are fixed by taking the median value of the descriptors. For example, using the notations of the above section, the threshold used to choose between a white piece and a black piece for the “sum” descriptor is (a+b)/2.

4 Difficulties

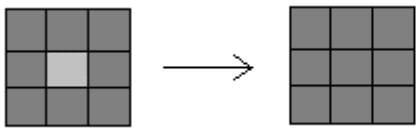
Two recurring artefacts were encountered: a gradient of brightness along the image, with bright dark regions, and parasite spots.



gradient of brightness and contrasted spots

Sometimes, an ambiguity arises after thresholding the descriptors. Each time a move is made, we use a simple algorithm of *winner-takes-all* to determine which two squares have the best answer for the “difference” descriptor. Indeed, only two squares are selected at each move (the square from which a piece left, and the arrival square). Thresholds are adapted if a descriptor value is close to a threshold.

For an unknown reason, contrasted spots appear near the chess pieces, especially in black squares. This is probably due to defects in the camera. To deal with this problem, isolated values of grey-levels are set to the value of neighbouring pixels, as illustrated below:



Results and improvements

The program that we developed is based on simple filtering and thresholding methods. However, it provides sufficient results for the recognition task. We didn't have enough time to test the results and produce statistical estimations of the error rate, but under favourable circumstances (paper chessboard illuminated by lamplight), the vision algorithm enables the robot to play a game without mistaking the human moves. When an occasional error does slip through, it comes from the parasite spots we mentioned.

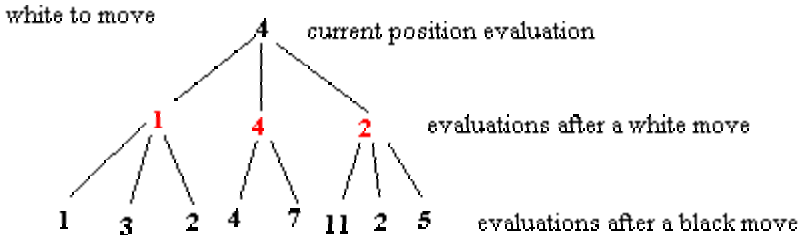
We could also use high-level information provided by the main program (by listing all the legal moves in a position and searching for one of those). If there is still an ambiguity, we could search for the most logical move, using the chess engine described in the next section.

PART III: THE CHESS ENGINE

General overview of chess engine algorithms

The chess softwares available on the market use algorithms based on brute force, which consists in travelling through the tree of variations until a certain depth is reached. Short-cuts are often employed to reduce dramatically the computation time, and good algorithms usually avoid the exploration of useless variations.

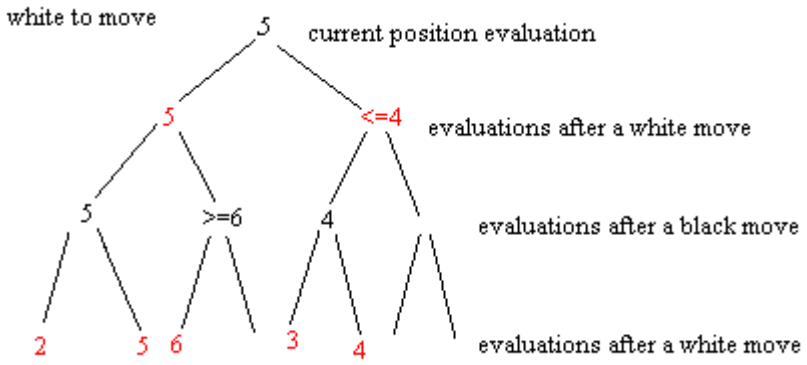
The basic principle is the *minimax* algorithm, which can be found in a lot of games for two players. In a given position, all the legal moves are listed, and an evaluation is calculated for each one of them. This evaluation is computed recursively by listing the sons of a node until a fixed depth is reached. At this point, a static evaluation is computed for the leaves, taking in account various strategic heuristics. It is positive if white has the advantage, and negative otherwise. The evaluation at a given node is set to the maximum (respectively minimum) of the evaluations of its sons if the node refers to a white (respectively black) move. Therefore, the leaves of the tree of variations are computed first, before climbing back to the root.
The figure below illustrates this algorithm, with a maximum depth of 2 (counted in half-moves, so it explores one move for white and one move for black).



The drawback with this algorithm is its exponential complexity:

Average complexity of the *minimax* algorithm : $O(c^p)$
Where: c *: average number of moves in a position*
p *: depth of the tree of variations*

The *alpha-bêta* pruning provides a significant improvement. The idea is to explore only useful variations, by fixing on each node a maximum or minimum evaluation. Those are computed by using the evaluations that have already been computed. The figure below illustrates this algorithm, with a maximum depth of 3:



The complexity of this algorithm is "only" the square root of the complexity of *minimax*, which means that with the same amount of computations, the maximum depth can be doubled:

Best-case complexity of the *alpha-beta* pruning algorithm : $O(\sqrt{(c^p)})$.

Other common improvements include:

(1) transposition tables, to avoid exploring twice the same position
(2) iterative tree exploration to allow breadth-first search; the most promising moves are analysed first
(3) use of a database of chess openings and chess endings

Let us now describe our own algorithm.

Description of our chess engine algorithm

1 Implementing the chess rules

A chessboard is considered as a matrix of squares (the size of the matrix is variable, because different types of chessboards were tested for the robot). Each square has local references targeting its four neighbours, so as to make only local accesses.

Special care has been given to optimise the speed of move listing. With the use of object oriented programming, each piece has a method to calculate its legal moves in a given position. This is quite simple for a Knight, for which 8 fixed squares must be tested. For a long-range piece such as a Rook, the chessboard has to be scanned in certain directions until a piece (black or white) is encountered.

Of course, all the other rules of chess have been implemented to check if a move is legal:

(1) check if the King is exposed to capture after a move is made
(2) check if castling is legal (and therefore memorising the fact that a Rook has moved, etc.)
(3) enable the promotion of a Pawn that has advanced to the last rank (four possible moves, depending on the piece of promotion: Queen, Rook, Bishop, or Knight)

Implementing the rules requires additional fields such as:

(1) whose turn is it
(2) is the computer black or white
(3) memorising if castling is still legal (both Queen-side and King-side)
(4) memorising if the last move enables a capture *en passant*, etc.

We may now look at how the "best move" is computed in a position.

2 Evaluation of a position

Let us first of all define a few words:

Tree (of variations): the moves that are analysed form a tree. One node refers to a position, and the sons of this node refers to all the positions that may arise after one legal move is played from the node
Depth (of a move): depth of a node, counted in number of half-moves
Dynamic evaluation (of a position): the process of evaluating a node by using the sons of the node.
Static evaluation: the process of evaluating a node (or a leave) by counting the value of pieces on the board, and by applying various strategic heuristics

The rules have been completely separated from the evaluation process (strategy, tree search). This means that only legal moves are analysed in the tree, and in particular, a King is never captured during the analysis.

Of course, the tree is only implicit in the evaluation process, and is never explicitly built as a whole. Once a move is analysed (once a sub-tree is explored), the move is erased from the list

of possible moves. It is very important to progressively erase the tree while exploring it, so as not to exceed memory capacities. Thus, the only limitation is time of computation, not memory storage.

The basic algorithm uses a *minimax* with *alpha-beta pruning* (described earlier), over which we developed and tested a lot of ideas.

(1) Reducing the initial interval for alpha-beta pruning (a variation on the aspiration search algorithm)

The evaluation of a sub-tree (at depth d) stops when a sufficient evaluation is found at depth d+1. If white is to play, the stopping condition is:

$$evaluation(son) > Max\ (1,\ 1 + evaluation)$$

Where : *evaluation* refers to the static evaluation of the root of the sub-tree (at depth d)
evaluation(son) refers to the dynamic evaluation of the current son (at depth d+1)

This formula states that we continue the exploration of a sub-tree unless we obtain an significant advantage (+1 for white) *and* the move sufficiently improves the static evaluation of the position (*1 + evaluation).*

This pruning is done everywhere except in the main line, so as to find the best possible variation among "good moves".

(2) Static sorting of the list of moves

When the list of legal moves is computed, moves are sorted with a hash-table according to their *utility* (this notion is explained in the next paragraph, and measures weather a move is useful). At a given node, we only consider the *first few* sons in the sorted list of moves, instead of all the moves.

In fact, we explore the first $n = f\ (depth,\ position\_complexity)$ moves,
where: *depth* is the depth of the current node
position_complexity is a number that reflects the complexity of the position at the current node (measured by the number of legal moves and the number of pieces that can be captured)
f is an integer-valued function that decreases with *depth* and increases with *position_complexity*

Typical values for f are shown below (under the assumption of an average value for the paraleter *position_complexity*):

depth	*f (depth)*
0-1	10
2-3	8
4-5	6
6-12 *	3
>13	0

* *Only forced moves and forcing moves are examined (see further below for details)*

It gives a tree of size of theoretical size: $10x10x8x8x6x6x3^7 \sim 5x10^8$, with an alpha-beta best-case complexity of about $\sqrt{(5x10^8)} \sim 2x10^4$, which is quite reasonable (in fact the program actually analyses about $2x10^4$ nodes before playing a move, as can be seen by printing the tree of variations that have been explored).

3 Utility of a move

The utility of a move determines in which order the list of moves will be sorted (and eventually analysed). This term refers to several strategic and tactical concepts.

Tactical considerations include:

(1) does the move capture a piece ? If yes, of what value ?
(2) Is the arrival square attacked ? If yes, is it also protected ?
(3) Was the moving piece attacked on its departure square ? If yes, was it also protected ?

These questions are efficiently answered to by listing on each square the pieces that attack and protect it (sorted by the piece value). For example, if a Knight is protected by a Pawn, it is defended against a Queen attack, but not against a Pawn attack. Each time we move in the tree, this information is updated.

Strategic considerations include:

(1) Does the move prevent the player (or its opponent) from castling from now on ?
(2) How does the move alter the player's (or its opponent's) Pawn structure ?

When the position is complex and a lot of pieces are attacked, a very short dynamic evaluation may also be processed in order to sort the candidate moves. To avoid too much branching, the tree is bushy near the root but narrows near the leaves. At high depths, only sequences of *forced moves* (escaping from check or avoiding losing material) and *forcing moves* (checking the King or capturing a piece) are examined.

4 Static evaluation of a position

This is used to compute the evaluation on a leaf. It mainly concerns strategic aspects of the position, as opposed to tactics of dynamic evaluation. A score is estimated by weighting the following terms:

(1) material (sum of the values of white pieces minus sum of the values of black pieces)
(2) influence of pieces on the board
(3) control of the centre
(4) security of the King

These are quickly evaluated by listing the pieces attacking and protecting a given square, as we did before. For example, point (4) can be estimated by simply counting the pieces that attack and protect the King and its immediate neighbourhood.

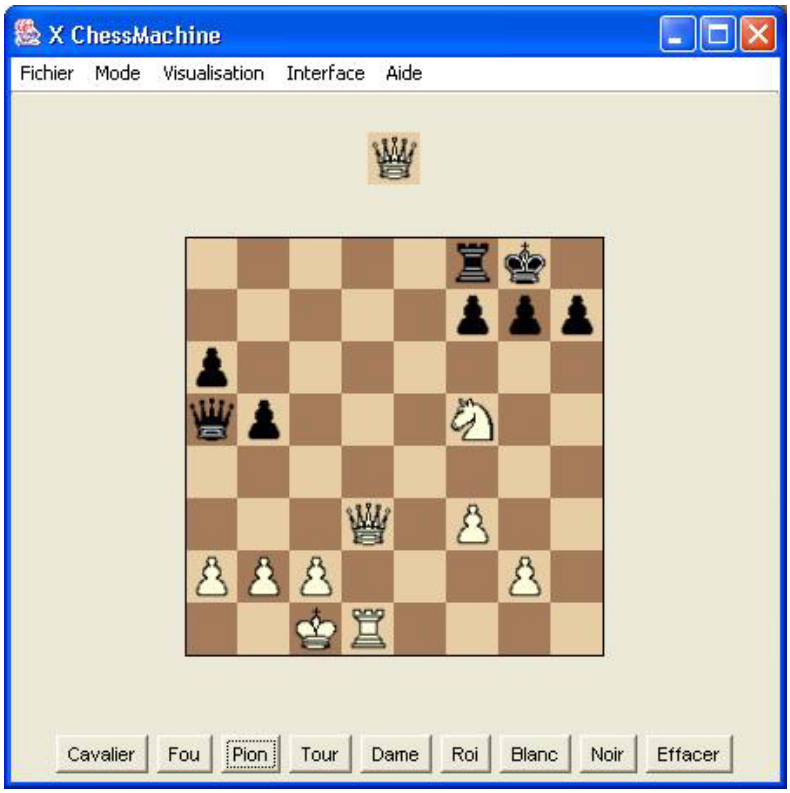
Results

The chess program has the approximate level of an intermediate chess player. This evaluation was made after completing 20 test games against players of different strength. The program plays at a speed of about 1 move in 10 seconds on a Pentium III. One move typically requires the evaluation of 50,000 positions, at a rate of 5,000 nodes per second.

In fact, the strength of the program highly depends on the type of position.

(1) Openings: there is no library of chess openings included in the program. However, (and quite surprisingly), the moves that are played are quite logical at this level of the game. A very simple heuristic (measuring the influence of pieces on the board and especially in the center) makes the computer develop his pieces smoothly. It often happens that a few (5 or 6) theoretical moves (coming from books of chess openings) are played by chance by the machine.
(2) Middle game: tactical play is good and strategic play is acceptable. In particular, winning combinations and checkmates are quickly found.
(3) Endings: the level is very weak at this point. The program has no long-term strategy and plays rather insipid moves.

Here is an example of a checkmate combination that the engine found (the position was set with the "position setup" option provided in the window interface).



The computer (with the white pieces) found the mate in 3 :
1. Ne7 check, Kh8 ; 2. Qxh7 check, Kxh7 ; 3. Rh1 checkmate

Problems encountered

The greatest difficulty in the development of the program concerned the pruning of the tree of variations. A bad pruning invariably causes the program to play absurd moves (among the list of legal moves, of course). Printing the analysis and examining the tree enabled us to find the origin of most of the bugs and fix them.

Fundamental problems still remain though, such as the weakness of the program in the endings, absurd moves that are played from time to time, strange ordering of moves, etc.

Possible improvements

Besides implementing existing move selection algorithms (iterative tree exploration, hash tables for transpositions, loading a library of openings or endings), we have imagined a learning algorithm and started to work on its implementation. The idea is to ameliorate the static evaluation function by weighing correctly the different strategic/tactical parameters. The error to minimise is the difference between the static evaluation and the dynamic evaluation at a given node. This process is applied to each node of the tree, during the move selection analysis.

A Neural Network architecture is especially adapted to this learning task. The input layer is composed of the most relevant parameters describing a position (parameters measuring the influence of pieces over the board and in the center, the safety of the King, the material equilibrium, the Pawn structure or even the piece configuration itself). The output layer computes the static evaluation as a linear combination of the input layer parameters. The coefficients of this linear combination (called the weights) are adapted during the learning process with, for example, a backpropagation algorithm. The dynamic evaluation is provided as the desired output of the network, so as to minimise the error mentioned above.

PART IV: THE MAIN PROGRAM

Program flow

After the initialisation process for the chessboard recognition (described in the Vision part), the game can start. The camera takes a picture of the board every 10 seconds, recognises the position and waits until it has changed. The move that was played is deduced from the new position (special care is needed to recognise castling and *en passant* capture, for which the content of more than 2 squares is changing), and sent to the chess engine. The latter computes the reply and sends it to the main program. The main program computes the sequence of instructions required by the robot to play the move, writes it in a file and calls the Turbo C program. This program analyses the instructions file step by step, and initiates the motor servo-loop control. Low-level instructions are computed by the Turbo C program and sent to the robot, with adequate voltage and digital to analog conversions. The robot plays the move and a new cycle may begin (the computer waits until the human player makes a move, etc.).

Planning the sequence of instructions to play a move

The pseudo-code below illustrates how a move is planned for the robot to play it on the board. Instructions are first written in a text file (*robot_instructions_file*), before being read by the Turbo C program for further processing.

```
play_move (departure_square, arrival_square, piece_capture) {
    if piece_capture then take_away_piece (arrival_square)
    move_piece(departure_square, arrival_square);
    go_to_rest_position;
    execute_robot_instructions_file;
}

move_piece(departure_square,
arrival_square) {
    lift_piece (departure_square);
    put_down_piece(arrival_square);
}

lift_piece (square) {
    go_to_chess_square(square);
    open_pliers;
    go_down;
    close_pliers;
    go_up;
}

take_away_piece (square) {
    lift_piece(square);
    throw_piece_to_garbage;
}

put_down_piece (square) {
    go_to_chess_square (square);
    go_down;
    open_pliers;
    go_up;
    close_pliers;
}
```

The format of the *robot_instructions_file* has been explained earlier, in the Robot part. Given below is an example of such a file, with all the instructions required to play a certain chess move.

```
c_2.0_17.0_15.0/p_0/c_2.0_17.0_14.0/c_2.0_17.0_13.0/c_2.0_17.0_12.0/c_2.0_1
7.0_11.0/c_2.0_17.0_10.0/c_2.0_17.0_9.0/c_2.0_17.0_8.0/c_2.0_17.0_7.0/c_2.0
_17.0_6.0/c_2.0_17.0_5.0/c_2.0_17.0_4.0/p_1/c_2.0_17.0_4.0/c_2.0_17.0_5.0/c
_2.0_17.0_6.0/c_2.0_17.0_7.0/c_2.0_17.0_8.0/c_2.0_17.0_9.0/c_2.0_17.0_10.0/
c_2.0_17.0_11.0/c_2.0_17.0_12.0/c_2.0_17.0_13.0/c_2.0_17.0_14.0/c_2.0_21.0_
15.0/c_2.0_21.0_14.0/c_2.0_21.0_13.0/c_2.0_21.0_12.0/c_2.0_21.0_11.0/c_2.0_
21.0_10.0/c_2.0_21.0_9.0/c_2.0_21.0_8.0/c_2.0_21.0_7.0/c_2.0_21.0_6.0/c_2.0
_21.0_5.0/c_2.0_21.0_4.0/p_0/c_2.0_21.0_4.0/c_2.0_21.0_5.0/c_2.0_21.0_6.0/c
_2.0_21.0_7.0/c_2.0_21.0_8.0/c_2.0_21.0_9.0/c_2.0_21.0_10.0/c_2.0_21.0_11.0
/c_2.0_21.0_12.0/c_2.0_21.0_13.0/c_2.0_21.0_14.0/p_1/c_15.0_0.0_15.0;
```

Chessboard limitations

The pliers are somewhat imprecise (error of several millimetres) and cumbersome, so we have to work with big chess pieces. On the other hand, the robot arm has a limited scope of action (the travel range is 22 cm), therefore we must use a small chessboard. These constraints left us no choice but to use a 8x6 chessboard instead of the standard 8x8 one.

CONCLUSION

We have had the pleasure to see through the project in a four months period of time. After fixing some bugs in the vision algorithm and linking the sub functions, a few games have been played by the machine against a human opponent. The results were successful and showed that the robot, the chess engine and the vision part all worked. However, a lot of improvements could still be made in all those parts. In particular, the use of thinner pliers and more accurate sensors would enable the use of smaller pieces, so as to play on a normal chessboard (instead of the reduced 8x6 chessboard). Also, connecting the hardware and the Turbo C program to the main program in Java was no easy task, and we had to rely on tricks (like writing instructions in a text file) that were not especially elegant.

This project was instructive in many ways. First of all, it helped us in managing a project and sharing the load of work. Secondly, it showed that simple but well adapted algorithms are often more efficient than more general and complex ones. Lastly, it gave us the opportunity to work at the interface between three related disciplines: Artificial Intelligence, Vision and Robotics, which lead to very interesting issues when studied together.