# Search, Actuate, and Navigate Lab course 2011-2012

Faculty of Science,
University of Amsterdam

May 29, 2012

This manual describes the lab course of the first year AI course "Zoeken, Sturen en Bewegen" a.k.a. "Search, Actuate, and Navigate".

Before you ask any questions, **read through the WHOLE manual!**

## Contents

**1 Introduction** **2**
1.1 AI through robotic chess . . . 2
1.2 The lab course . . . 2
1.3 Other resources of information . . . 3

**2 The programming environment** **3**
2.1 Playchess . . . 3
2.2 The robot simulator . . . 4
2.3 The chess program 'GNUChess' . . . 4
2.4 Component interfaces . . . 4

**3 Hardware setup** **6**
3.1 The chess board . . . 6
3.2 Robot arm configurations . . . 6
3.3 Radial motion of the UMI-RTX robot . . . 6

**4 Tasks** **8**
4.1 Task 1: Endgames . . . 8
4.2 Task 2: Path planning module . . . 9
4.3 Task 3: Inverse kinematics module . . . 11
4.4 Task 4: Go, where no one has gone before . . . 12

**5 General lab course instructions** **12**
5.1 General instructions . . . 12
5.2 Writing reports . . . 12
5.3 Writing software . . . 12
5.4 Testing your software on the robot . . . 13
5.5 How to send your code . . . 13
5.6 Grading . . . 14
5.7 Getting started . . . 14

# 1 Introduction

This is the manual of the 1st year Search, Actuate, and Navigate lab course (formerly known as the Robotics course). It gives a general overview of the problem to be solved and its subdivision to assignments.
This manual contains a lot of information which might tend to blur your vision of the problem at hand. This is why you have to:

1. Read the manual very carefully before starting the lab course.
2. Attend the lab course sessions every time where your lab assistants will try to guide you and provide help.

We hope you will enjoy the lab course,

Julian Kooij

## 1.1 AI through robotic chess

Pioneers of artificial intelligence and robotics were more than enthusiastic and had envisioned, through rosy spectacles, a world populated with decision-making, moving machinery. With this high expectation, research concentrated on high levels: theories, models, ethics, and how this impinges on other subjects. This did sometimes appear to be esoteric; it eventually came to light that there were indeed more down-to-earth problems yet to be solved - mechanical structures, movement execution, optimal path-planning with obstacle avoidance etc. It is with these latter issues that this exercise is concerned, consciously keeping in mind throughout however, that it is these high levels that take precedence in any 'outlet'.
Chess will be the game at hand and it will stand for the essential enigma to be solved. You are to program a *robot arm* enabling it to physically carry out the solution of an arbitrary game of chess. It is a mild but challenging and entertaining problem, and with the fine guide of your intense, learned brain (and a workstation to help it along), you shall succeed, and furthermore be led, in time to come, towards the creation of an intentional system[1]. With the strong foundation in robotics that you will acquire, we hope that you, as stars of tomorrow, will build a solid aesthetics of method, to serve to the betterment of society. Yes! my dear friends, Science shall march boldly forward. *(thus spake Joris)*

## 1.2 The lab course

In this lab course you will, solve chess **endgames**, **plan paths** for the chess pieces, and **calculate** the required **poses** of the robot, and so solve the problem of a robot playing a game of chess. All other (non-trivial) tasks have already been solved and the solutions will be provided by your lab assistants.
Your lab assistants will provide you with software for separately testing your solutions to all problems. We do, however, expect that the solutions to the three subtasks will be submitted as a single, working, integrated program (using our main module). This means that you should test that your solutions to the subtasks can work together.
While you are trying to solve the tasks you will for sure stumble upon some awkward features (bugs) in the provided software and documentation. Do not blame the lab assistants for this, well, just a tiny bit. We know the software can be improved in many ways and the interfaces are not that great. But actually this perfectly mirrors the problems you will be confronted with in the "real world", when dealing with practical problems that require a lot of different parts of software including drivers for old hardware. Report them to the assistants (who perhaps already know the

[1] Daniel Dennett, Brainstorms, MIT Press 1978.

feature/bug) and try to work around them. Solving these bugs is nice, but costs a lot of time which is one thing you will lack the coming weeks.
This manual gives you a precise description of the tasks you have to solve and provides you with the background needed to do this. First we will explain how the software and hardware works. Read it carefully before starting to work on the assignments. Section 4 gives the assignments you'll be working on. And finally some general lab instructions are given in Section 5, ending with the "Getting started" (Section 5.7).

### 1.3 Other resources of information

Of course not all info you need is in this manual. Therefore one or two assistants will be present in the lab during the practical hours (look at the main course page for the schedule), helping you with the tasks, trying to help with programming, and operating the real robot. Your assistant for the lab course is **Julian Kooij**. For questions, remarks or whatever outside the practical hours, Julian can be reached by email at:

`J.F.P.Kooij@uva.nl`

The lab assistants can be contacted at:

`Veschoor@uva.nl`
`koster.elise@gmail.com`

**IMPORTANT:** When you send an email, please **start the subject field** with the three letter word `ZSB`, such that your email can be handled appropriately by the email filters.
The webpage for the practical course can be found at

`http://student.science.uva.nl/~rtxipc/` .

Check this site frequently, especially the *news* section, as it will be updated with valuable info during the course. On the webpage you will also find a link to the documentation of the Java software.

## 2 The programming environment

### 2.1 Playchess

We will be using a simple program called `playchess` which does the following (also see figure 1):

1. Call an existing chess program (GnuChess) or your endgame solver to play a white move.
2. Call the Java path planning module with white move.
3. Execute the Java inverse kinematics module to convert the Cartesian positions into values of the joint angles of the robot arm.
4. Send the joint angles to a simulator (`umirtxsimulator`) for inspection, and send them to the real robot.
5. Ask the user for a black move.

The derived programs `endgamerook` and `endgamepawn` call the Prolog endgame solver for white moves while the user (you, or gnuchess if you choose to) provides the black moves. This way you can test the endgame solver. Basic path planning and inverse kinematics modules are already provided for testing. However, during the practical course you will develop your own modules which are hopefully as good as or better than the ones provided. In the `playchess`, `endgamerook` and `endgamepawn` programs you can decide which modules to use, the ones provided or the ones you developed, by changing the settings.
During this lab exercise, `endgamepawn` will be the way to test the king and pawn versus king endgame solver. When configured to use your Prolog code, it will execute the following command, which you can also test in a terminal yourself:

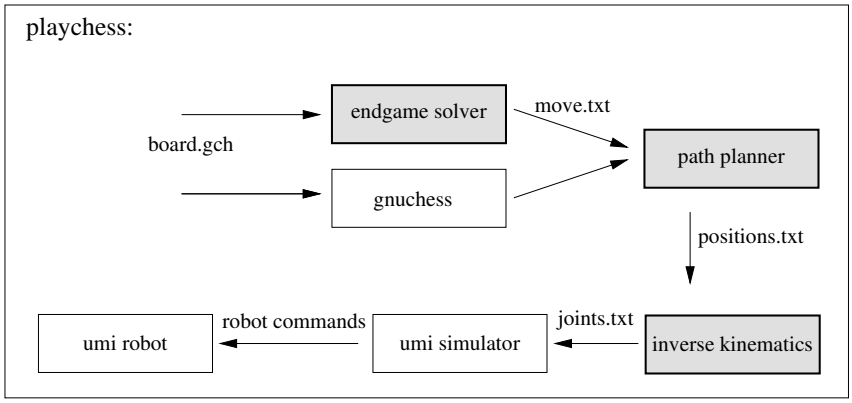



Figure 1: Information flow of the different components controlled by the `playchess` program. The gray-filled components are only partly provided and should be completed during the course. `board.gch` and `*.txt` denote actual files which can be inspected and manipulated.

```
pl -s chess.pl -g true -t startPawn
```

The program `endgamerook` works similarly for the king and rook versus king endgame and is given as an example implementation.

## 2.2 The robot simulator

In order to check that the robot will not demolish its surroundings, your commands will be sent to a simulator. Thus to determine whether a particular solution to the problem is correct it is first visualized. The simulator can be started using the program `umirtxsimulator`. By running the program in the same directory as the `playchess` program, the simulator will perform the black move.
When your lab assistants have confidence in your approach, they can put the real robot arm into work using your software, via the 'move robot'-button of the simulator. This button is the interface to a server that controls the robot. One of the activities of the server is to calculate the 'encoder counts' based on the specified joint-angles. These encoder counts are the number of steps the motors of the robot arm have to make to arrive at the given joint angles. In general controllers are interesting things, but for now of no importance to you. However, it is important that the controller is there. It forms the interface between the joint angles and the flow of currents to the actuators; in this case electrically powered rotary motors.

## 2.3 The chess program 'GNUChess'

This is a chess playing program which decides what move the robot should play given the current board setting. In tasks 2 and 3 your modules have to make sure that the move suggested by GnuChess is the one actually played by the robot.

## 2.4 Component interfaces

For the most part communication between the components and `playchess` takes place by writing and reading files, which will be briefly described below. The interface is not the best possible, but it works and it allows us to use older software for "stub" procedures. While you are testing your software it is sometimes useful to look at the contents of these files and maybe even change them.

### 2.4.1 Chess board representation

The chess board is represented as an ASCII file called `board.gch`. This file is constantly updated while a game is being played and is read by the Prolog endgames implementation as well as the Java pathplan code to get the most recent boardsetting. The 8 by 8 matrix describing the position of the pieces is the only information used by our software components. The other values in the file such as the time control and the score are neglected. Lower caps are the white pieces.

```
Black computer White Human 1
Castled White false Black false
TimeControl 0 Operator Time 0
White Clock 0 Moves 0
Black Clock 0 Moves 0

8 R.BQKBNR 0 0 0 0 0 0 0 0
7 PPPPPPPP 0 0 0 0 0 0 0 0
6 ..N..... 0 0 0 0 0 0 0 0
5 ........ 0 0 0 0 0 0 0 0
4 .....p.. 0 0 0 0 0 0 0 0
3 ........ 0 0 0 0 0 0 0 0
2 ppppp.pp 0 0 0 0 0 0 0 0
1 rnbqkbnr 0 0 0 0 0 0 0 0
  abcdefgh

move  score depth  nodes  time flags capture color
```

You may edit this file by hand to change the position of the pieces or replace it by another file with a different chess setup while a game is being played. To start with a different game setting than the default, you can add a file with the name `renewboard.gch`. This file will be loaded by playchess when a reset board command is send by hitting "r". In `/opt/stud/robotics/data` a few board settings are given on which your implementation can be tested.

### 2.4.2 Chess move

The file `move.txt` is written by the Prolog endgame solver to communicate the chosen chess move to `playchess`.

### 2.4.3 Forcing tree

The file `forcingTree.pl` is also written by the Prolog endgame solver (see section 4.1 of this manual). It is used to store the forcing tree, for the next move (the endgame solver is called for each chess move separately). You could also change it while playing a game for debugging purposes.

### 2.4.4 Cartesian path representation

The Java path plan algorithm produces a list of Cartesian positions of the robot gripper and puts these in the file `positions.txt`. For example:

```
124.279671 313.029663 37.500000 4.188790 30.000000
124.279671 313.029663 37.500000 4.188790 0.000000
124.279671 313.029663 218.000000 4.188790 0.000000
13.937696 427.147583 218.000000 4.188790 0.000000
13.937696 427.147583 58.000000 4.188790 0.000000
```

On each line a different gripper position is described using the $x$-, $y$-, $z$-coordinate of the tip of the gripper, the roll of the gripper and the angle between the two gripper elements respectively.

#### 2.4.5 Joint path representation

The `joints.txt` file is quite similar to the `positions.txt` file. The Java inverse kinematics algorithm writes in `joints.txt` a path as a list of configurations of the joints of the robot. For example:

```
244.5 -29.710975377910298 100.87246341721223 -20.72525633069582 -90.0 0.0 30.0
244.5 -29.710975377910298 100.87246341721223 -20.72525633069582 -90.0 0.0 0.0
425.0 -29.710975377910298 100.87246341721223 -20.72525633069582 -90.0 0.0 0.0
425.0 -34.28883336382281 69.66192932102668 -0.5421312966905262 -90.0 0.0 0.0
```

On each line a certain robot arm configuration is written. See the Java documentation for a precise description of these values.

# 3 Hardware setup

For the path planning task it is necessary to know the position of the chess board related to the position of the robot, and for the inverse kinematics task you have to be aware of the possible configurations of the robot arm.

## 3.1 The chess board

The drawing of the board placement, see figure 2, gives the representations of its position and some arbitrary position of the garbage places. The $x, y, z$ position of the board is determined by the outer corner of field h8, with the edge included.
**IMPORTANT:** Note that the coordinate system is a *left*-handed system. This means, that the path planning module must express Cartesian coordinates in this very same left handed module!

## 3.2 Robot arm configurations

This section is related to the subtask of inverse-kinematics. We discuss here some details of the motion of the UMI-RTX robot. This robot is designed to work in a cylindrical workspace. Vertical movements and rotations around the base are easily performed by controlling a single joint. Radial movements of the wrist from and to the base involve movements of several joints, but this is also made simple by a trick, which is discussed in the next section.
To illustrate the difference between the performance level of the human arm and this robot arm the following can be said: a 'natural' arm of a human has about 42 degrees of freedom. The most subtle artificial arm, for general human use, has less than 10 degrees of freedom. The drawing, figure 3, specifies only 6 degrees of freedom which are the minimal demands for an artificial fore-arm and wrist.
The UMI-RTX robot has precisely those minimal 6 degrees of freedom, enough for finding a solution for this robot arm in the domain of chess playing.

## 3.3 Radial motion of the UMI-RTX robot

The upper arm and lower arm have the same length: AB equals BC:
This means that you can move the wrist in a straight line outwards from the column - a radial line between the shoulder and wrist spindles - by rotating the two parts of the arm, making sure that angle $P$ is always half angle $Q$:
The gear ratio from the shoulder motor to the upper arm is twice that of the elbow motor to the lower arm. To move the wrist in a radial line, both motors are driven at the same speed but in opposite directions. The benefit of this arrangement is that the robot controller can keep track of the position of the wrist in cylindrical coordinates very easily, by simple calculations based on the motor encoder counts. (Encoder counts are the units in which the movement of an rtx motor is measured).

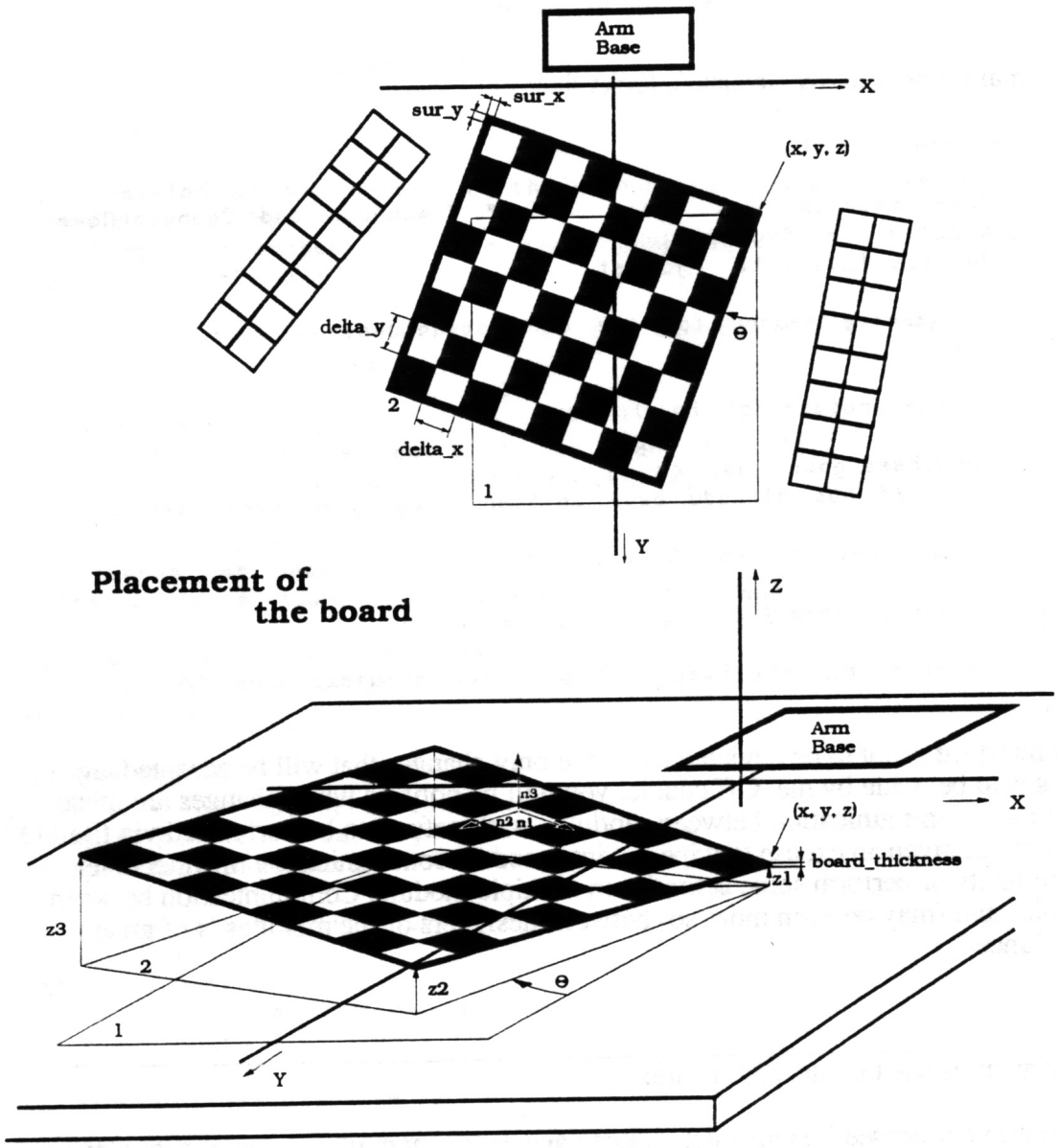



Figure 2: The chess board.

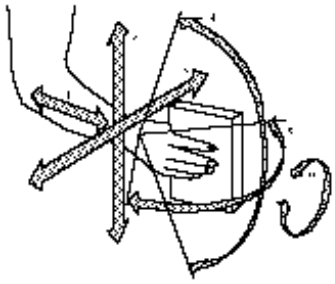
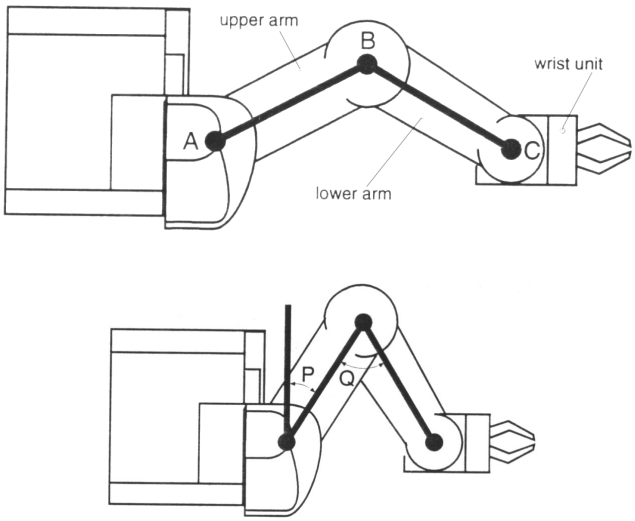
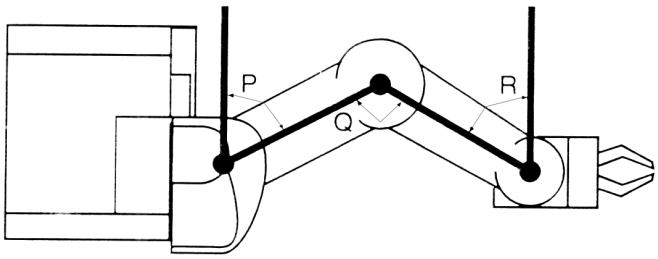


Figure 3: Degrees of freedom of a human arm.




In addition, the gripper is automatically pointing along the radial line:




This is achieved without needing to drive the yaw motor (angle R): the yaw is not only coupled to its own motor, but also coupled to the elbow motor! When the lower arm moves through an angle Q, the wrist automatically moves through R, which is Q/2, because of the 2:1 gear ratio from the combined pulley which rotates on the elbow spindle and the wrist pulley. This behavior is by the way not correctly produced by the simulator.
Because of this automatic compensation, it is possible with certain yaw orientations for the wrist to hit its end-stops when driving the arm radially in and out.

# 4 Tasks

## 4.1 Task 1: Endgames

In the first week you are going to solve a chess endgame of "king and pawn versus king", in similar fashion as the rook (in Dutch: toren) endgame discussed in the Bratko book (chapter 22). The main task is that the white pawn reaches the other side of the board, without being attacked by the black king. Use the white king to defend the pawn if necessary. In chess, this reaching the other side would allow the pawn to promote to a different type of piece (such as a queen), thereby deciding the game in a win for white. Your program should be able to promote the pawn for any board setup where it is possible. The provided `renewboard.gch` is one possible setup for you to test, but try other board setups yourself too.
If you feel really creative, and have time left, you are free to try and implement other endgames for a higher grade. Of course, be sure to write a clear section in your report about this extra work too, as your lab assistant should be able to understand exactly what you did from your report alone.

First read the instructions on how to setup your environment properly, see section 5.7. Then copy the files required for this part of the lab course in your newly created directory by typing:

```
cd ~/zsb
cp -r /opt/stud/robotics/software4students/pl/* . # don't forget the dot
```

In your `zsb` directory you should now find a number of Prolog files, that are an implementation of Bratko's Advice Language 0 and the predicate library plus advice rules for the "king and rook versus king" endgame. Before one starts to program, it is strongly recommended to read chapter 22 of Bratko thoroughly.
The Prolog code consists of various files. The file `AL0.pl` contains the general Advice Language 0 implementation as is provided by Bratko (some minor adjustments were made to make it compatible with SWI-Prolog). The file `KRPL.pl` contains some general help functions for the chess domain. The file `KRPLrook.pl` contains functions that deal with rooks and the file `KRAProok.pl` contains the advice rules. Note that the advice rules use functions defined in the `KRPLrook.pl` file.
To try the rook endgame, run `endgamerook` in your `zsb` directory. This will start the `playchess` program which connects all parts of the lab course (see section 2.1). As this is the first time you start the program you have to initialize the chessboard setting it starts from by hitting "r". By pressing "p" you can now check the performance of our rook endgame implementation. The program provides the white moves, you can try to escape by entering the black moves. You can try other starting settings by editing the chess board file or copying one of the prebuild settings to `renewboard.gch`, see section 2.4.1.
Now that you know how the rook endgame works, it is time to setup the pawn endgame that you are going to implement. The code for this endgame should be put in the files `KRPLpawn.pl` and `KRAPpawn.pl`. You might also want to make some changes to the `KRPL.pl` file. To try the pawn endgame, run `endgamepawn` in your `zsb` directory, and press "r" to reset the board with a pawn. The following steps may guide you through the implementation process:

- Implement how a pawn moves. For inspiration refer to KRPLrook.pl.
- Test if your pawn moves correctly. To do this, create an if -rule that is always true and use it to select a 'dummy' advice. Think carefully about the better goal and holding goal of this dummy advice!
- Bring along with you a real chess board and try to think of a strategy to checkmate the black king, no matter what the black king does.
- Transform your strategy into better goals and holding goals. Using these goals, create rules that select advices in turn.

Eventually, your program should be able to win the game from any (reasonable) board setup. The default board is one possible setup for you to test, but try other board setups yourself too.

## 4.2 Task 2: Path planning module

This section contains a detailed description of the second task you have to perform, which is to find a path in Cartesian coordinates. The main input of this module is the chess move generated by GnuChess (e.g. a2a3). The output of this module is a list of $x, y, z$ positions that the robot arm has to follow in order to perform the chess move generated by GnuChess. This $x, y, z$ list is subsequently provided as input to the inverse kinematics module. More specifically, the involved piece has to be transported on a height of 'two fingers' above the board, if that is possible.
This task takes you through several steps which will lead you to a modular solution to the task. See these steps as functional steps, not as strict temporal orders. Decisions taken at the beginning can have strong influence on the final performance, so start pondering the whole task before getting

into the details. You are then advised to write some routines to print the input and output of your module, as a good insight in the data-flow is essential during the execution phase.
The necessary Java files should be copied in your lab course directory by typing:

```
cd ~/zsb
cp -r /opt/stud/robotics/software4students/pp/* . # don't forget the dot
```

### 4.2.1 Elementary transformation

As an introduction you'll write some functions essential for planning a path for the robot arm. The assignment consists of three parts: access some data of the board, write a class to convert positions ("e3") to locations (column 4, row 2) and write a function, that converts locations to Cartesian coordinates $(x, y, z)$.
What board parameters would you use to determine a transformation from board coordinates (e.g. "a2") to Cartesian $(x, y, z)$ coordinates? Are they all given by the ChessBoard class interfaces? (see the Java documentation and see the board figure in section 3.1). Create such a transformation. Would you use homogeneous transformation matrices or smart vector manipulation? Discuss your findings with your assistant and implement your ideas.
You will first edit the file `BoardTrans.java` which contains some code that will be used by `PP.java`. Before doing anything more try reading the comments contained in `BoardTrans.java` while also having a look at the documentation of the data structures. The file contains various gaps indicated by question marks (?????). You have to first carefully fill in the gaps and then compile your code by typing:

```
javac -1.6 BoardTrans.java
```

If this part has been done successfully the compiler should not have printed any errors and your directory should contain a file called `BoardTrans.class`. To test your code type:

```
java BoardTrans
```

or to test out different board positions, such as "f2":

```
java BoardTrans f2
```

### 4.2.2 High path

From now on you will complete the unfinished code in the `PP.java` file. You will begin by implementing a procedure

`highPath(String from, String to, ChessBoard b, Vector p),`

that transports the involved piece on a 'safe height' directly to its destination. You will have to steer the gripper of the robotarm along ten positions:

1. safe height over the piece and its gripper is open.
2. low height over the piece and its gripper is open.
3. half pieceheight and its gripper is open.
4. half pieceheight and its gripper is closed.
5. safe height with gripper closed.
6. safe height over the new position and the gripper is closed.
7. low height plus half pieceheight and the gripper is closed.
8. half low height plus half pieceheight and the gripper is closed.
9. half pieceheight and the gripper is open.
10. safe height and the gripper is open.

As you can see position 1 to 5 are over the old position and position 6 to 10 are over the new position of the piece. Safe height and low height are defined in your java program. The second and seventh position are necessary to compensate for the overshoot of the robot arm.
This procedure will be valuable if later it is found that a height of 'two fingers' is an impossible constraint. Test your program by running `playchess` and change the settings to use your path planning modules. You can see the results of your functions in the `umirtxsimulator` (make sure to start it in the same directory as you program).

#### 4.2.3 Removing pieces from the board

Improve your `highPath()` function by writing a

`moveToGarbage(String to, ChessBoard b, Vector g)`

function that removes checked pieces. This function should grab a piece of the board and drop it somewhere outside the board. You are free to implement this as involved as you want. You could for example choose to just drop the pieces at a fixed spot outside the board. You could also plan a nice garbage setup (e.g. as depicted in figure 2). Think about what will happen if the board is put in a different location.

#### 4.2.4 Low path

This is the real path planning part. Instead of moving on a 'safe height', try to move on a height of 'two fingers', in between the pieces on the board. Create a path planning algorithm that searches a path between any two places on the board, while some board positions are occupied by obstacles (other pieces). Is such a path always possible? Do you plan such a path in board or Cartesian coordinates? Discuss your algorithm with your assistants before you start to add the function

`lowPath(String from, String to, ChessBoard b, Vector p)`

to PP.java. This function should plan a path for moving pieces on a low height (add a "private static double LOWPATH_HEIGHT=20;") while avoiding all other pieces on the board. Diagonal moves are not allowed. Beware that the robot has an overshoot which could cause the robot to press pieces "through" the board. Look at the given high path positions for a way to circumvent this.

### 4.3 Task 3: Inverse kinematics module

The main task here is to generate an arm configuration according to the coordinates generated by the path planner module. A list of Cartesian coordinates is to be transformed to a list of joint values. Copy the inverse kinematics files to your local directory:

```
cd ~/zsb
cp -r /opt/stud/robotics/software4students/ik/* . # don't forget the dot
```

#### 4.3.1 The simulator

We will first work with the `umirtxsimulator`, which contains a model of the actual robot. To get a feeling for the joints play around with the controls. The simulator will change from a right- to a left-configuration and back. Try using the joint-control-buttons of the simulator to see if it is possible to continue on the path without changing configuration.
The inverse kinematics problem can be decomposed in several sub-problems. Can the position be achieved by controlling only a small number of joints? Which joints?

#### 4.3.2 Do the math

Solve the inverse kinematics problem for the robot arm. Would you use a homogeneous transform or smart vector manipulation? How many solutions are possible for a certain Cartesian coordinate? Discuss your results with the lab assistants.

#### 4.3.3 Implementation phase

Design an algorithm that chooses between alternative solutions in a proper manner. This can be quite involved, taking into consideration that the chessboard can be moved around and the robot arm has to be maneuvered according to the planned low path. Discuss your ideas with your assistants.
Implement your algorithm in `IK.java`, which already contains some code you can use.
Finally, verify your program on the simulator. How accurate is the robot? Are there systematic errors?

### 4.4 Task 4: Go, where no one has gone before

*Section intentionally left blank, content to be provided by you. Look at the main webpage for more info.*

## 5 General lab course instructions

### 5.1 General instructions

During the whole lab course you will be working in *pairs*. By making you work in groups of two we aim to give you an idea of how it is to work in a team. If there are any unsolvable organizational difficulties, troubles, hitches, impasses et cetera, do not act too late and inform your lab assistants at an early stage before everything gets out of hand. We would like to prevent students from getting lost. So definitely do not hesitate to inform us that there is some problem.
For each task you will hand in your software and a small report per pair. And at the end of third week you will demonstrate all your code on the robot to one of the lab assistants.

### 5.2 Writing reports

The reports you hand in should be about 2 to 3 pages, written in English or Dutch and contain (at least) the following topics:

- explain the problem at hand
- discuss possible solutions you considered
- why did you choose the solution you implemented?
- how did you test your program?
- conclude with any suggestions for improving your algorithm or code

Email your reports in a printable format, that is: PDF or Postscript (so no tex-code, plain text-files or Word-files). Instructions on writing reports can also be found on the lab course website.

### 5.3 Writing software

Implementation of your ideas into code is only half of the job to be done for this lab course. Inside your source-files we want the following information:

a header:

- the filename
- a short description what's in the file
- the names, registration-numbers and login-names of both partners
- the id of your group.

- the current date

comments:

- on a tactical level: to indicate where dirty tricks are performed
- on a strategic level: to explain what was your intention for a piece of code

self-explaining function- and variable-names:

- the most important issue is to be consistent
- don't mix Nederlands and English

structure:

- make use of empty space to make your program readable
- order your functions in a logical way

error reports:

- describe how the software was tested
- make sure that both programmer and user can understand what went wrong

Instructions on writing software, and commenting examples, can also be found on the lab course website.

## 5.4 Testing your software on the robot

If you tested your code on the simulator and you are (very) confident that it works correctly and does not, for example, press a chess piece through the chess-board, then you may want to test it on the real robot. However students do not have the right to control the robot. Only when a lab assistant runs the simulator the "move-robot" button is available. So in order to test your implementation you should ask one of the present lab assistants if he has time to control the robot. If so, send all your code to the lab assistant in the following manner.
Compose an email with the subject reading `ZSB Your Name` and attach your code, see Section 5.5 how to append your code. Specify your names and student-ids in the mail.

## 5.5 How to send your code

The code should be send as one *tar* file attached to an email to the assistant. To create a tar file do the following.

- Open a terminal.
- Go to the directory with your code (`cd ~/zsb/`).
- make a tarball of it by executing `tar -cvf YourName_Code.tar .` # don't forget the dot
- Attach the tarball to your mail.

If you do not understand how to work with a terminal or you do not want to learn it (sigh...), you can also make a zip file of your code with your favorite gui zip-program and attach the zip file.

## 5.6 Grading

Each task as well as the final demonstration will be graded. For grading the tasks we will not only look at the approach that was chosen to solve the problems but also the reasoning behind these choices. Explain in your report why you implemented algorithm $B$ and not algorithm $A$, and show that you understand the pros and cons of the different methods. Also the readability of the report and the code will be taken into account. Your lab course assistants will try their best to finish evaluating your reports the week after they were handed in. So the feedback can be used to improve the next report.
The demonstration will really be a crash-test for your system. In a few minutes you will have to show that your software works on a couple of tasks. Be sure to have a stable final version, because there will be no time for in-between hacking.

## 5.7 Getting started

The first time you start working you should setup your environment. This section assumes that you are using the `bash` shell, which is the default on student accounts nowadays.
To setup the environment, you need to enable certain software packages such that you can run the required programs. Software packages are managed by the `softpkg` program (e.g. you can use `softpkg -l` to list available packages). Setting up the software packages for this course is done in two steps. First, you need to define where some additional packages for this lab course are located in the system. This will be done by extending the `PACKAGEPATH` environment variable on your account. Second, you need to tell `softpkg` *which* packages you want enabled on your account.

Assuming you are using the bash shell, you will need to update your `~/.bashrc` (i.e. "bash resource file"). If you do not know which bash configuration files are set on your account, list them with:

```
ls -la ~/.bash*
```

The list will probably include `.bash_history`, and may or may not include `.bashrc` and other configuration files.

**Step 1a** If you already have a `~/.bashrc` file, edit it and add directly after any existing "`export PACKAGEPATH=...`" line the following:

```
export PACKAGEPATH=/opt/stud/robotics/pkg:$PACKAGEPATH
```

Now softpkg can find the additional packages for the robotics course.

**Step 1b** If you do not have a `~/.bashrc` file yet, we will provide you with an already correctly configured version. Copy it using the following command:

```
cp /opt/stud/robotics/software4students/bashrc ~/.bashrc
```

Note that the target file should be called `.bashrc`, so starting with a '.'! Additionally you need to copy the following file too if it does not exist in your home directory yet:

```
cp /opt/stud/robotics/software4students/bash_profile ~/.bash_profile
```

**Step 2** Next, tell softpkg which software packages should be enabled on your account. Edit your `~/.pkgrc` (i.e. "package resource file") file and add these two lines at the end:

```
robotics
jdk
swiprolog-5.4.7
```

and remove or comment out any lines with just `swiprolog`.
Close any open terminal window, and open a new one. This will start a new bash session, with the correct packages enabled. Now you're ready to start the simulator of the robot arm and your java `CLASSPATH` should be set.

**Step 3** Test if your account is configured correctly by running

```
umirtxsimulator
```

Typing just `umi` and pressing the TAB key should automagically complete the command.
You should now create a directory for all your work in the lab course. Type in the shell for example:

```
cd
mkdir zsb
cd zsb
```

### Tip for KDE

In this lab course, you will mostly be working in the terminal (running bash). However, most people login into a graphical desktop environment to run and manage terminal, editor and browser windows. For new student accounts this environment is KDE, which has as default behavior that the window focus follows the mouse, i.e. you always type in the window currently under the mouse pointer. If you prefer that the window focus is set by clicking on a window, you can change this behavior by following these instructions:
From the 'start' button at the button left of the screen, open 'Control Center'. In the control center, select 'Desktop', then 'Window Behavior'. You now see an option for the focus policy, which states *Focus Follows Mouse*. Change this to *Click to Focus* and click 'Apply'.

## Acknowledgments

Toto van Inge and George den Boer undertook the painstakingly effort of setting up the entire practical course and also wrote the first version of this manual. Their successors were Arnoud Visser and Joris van Dam. In turn they were succeeded by Gerben Venekamp. And after that Daan van Schaijk and Nikos Massios adjusted some exercises to make them suitable for 1st year students. And finally Paul Ruinard made some adjustments for the course 2000-2001.
The manual was originally written by Arnoud Visser and Joris van Dam and subsequently modified by Nikos Massios, Paul Ruinard, Matthijs Spaan, Olaf Booij and Julian Kooij.