

Search, Actuate, and Navigate Lab course 2016-2017

Faculty of Science,
University of Amsterdam

June 5, 2017

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 | Component interfaces | 3 |
| 3 | Hardware setup | 4 |
| 3.1 | The chess board | 4 |
| 3.2 | Robot arm configurations | 6 |
| 3.3 | Radial motion of the UMI-RTX robot | 6 |
| 4 | Tasks | 7 |
| 4.1 | Task 1: SimpleChess | 7 |
| 4.2 | Implementing <code>legal_moves</code> | 9 |
| 4.3 | Implementing <code>evaluate_board</code> | 9 |
| 4.4 | Implementing <code>minimax</code> | 9 |
| 4.5 | Task 2: A chess playing robot arm | 11 |
| 4.6 | Task 3: Go, where no one has gone before | 13 |
| 5 | General lab course instructions | 13 |
| 5.1 | General instructions | 13 |
| 5.2 | Writing software | 14 |
| 5.3 | Testing your software on the robot | 14 |
| 5.4 | How to send your code | 15 |
| 5.5 | Grading | 15 |

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,

Nick, Thomas, Boas, Pieter, Simon, Tim and Douwe

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 virtually and 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 computer to help it along), you shall succeed, and furthermore be led, in time to come, towards the creation of an intentional system¹. 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 van Dam)*

1.2 The lab course

In this lab course you will, solve chess **endgames**, 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 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 that is partly because some parts are really old and some are brand new (just finished). 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. In the last two weeks, you will encounter more of this situations. Report them to the assistants (who perhaps already know the feature/bug) and try to

¹Daniel Dennett, *Brainstorms*, MIT Press 1978.

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.

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 assistants for the lab course are **Nick de Wolf**, **Boas Kluiwing**, **Thomas Groot**, **Pieter Kronemeijer**, **Simon Paul**, **Tim Smit** and **Douwe van der Wal**. For questions, remarks or whatever outside the practical hours, the lab assistants can be reached by email at:

N.J.G.deWolf@uva.nl
R.B.Kluiwing@uva.nl
thomas--g@hotmail.com
p.kronemeijer@xs4all.nl
S.Pauw@uva.nl
groeten.aan.tim@hotmail.com
douwev.dwal@live.nl

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

https://staff.fnwi.uva.nl/a.visser/education/ZSB/lab_course.html

Check this site frequently, 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 Python software.

2 The programming environment

2.1 Component interfaces

For the most part communication between the components inside the main program takes place by writing and reading files, which will be briefly described below. The choice for this interface is partly historical, (it allows us to interface with older software written in other programming languages (C, Java, Pascal, Prolog) and other operating systems (Windows 3.1, Sun Unix)) but also very practical. While you are testing your software it is often useful to inspect the contents of these files and maybe even correct them to allow further testing (while your partner solves the bug that led to this incorrect input).

2.1.1 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 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`.

2.1.2 Chess move

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

2.1.3 Joint path representation

The `joints.txt` file is quite similar to the `positions.txt` file. Your 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 1, 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!

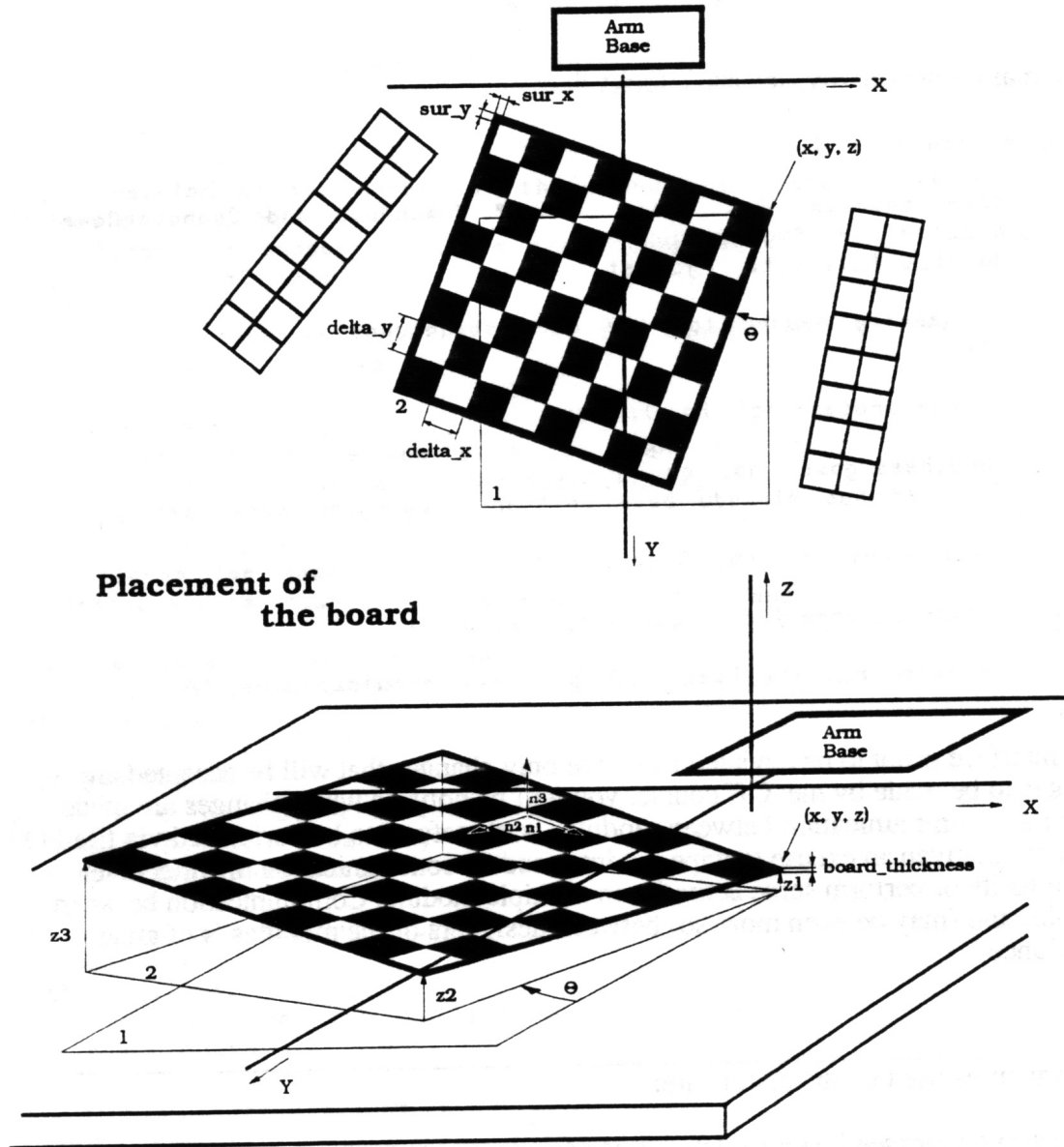


Figure 1: The chess board.

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 2, specifies only 6 degrees of freedom which are the minimal demands for an artificial fore-arm and wrist.

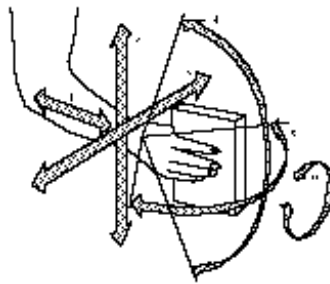
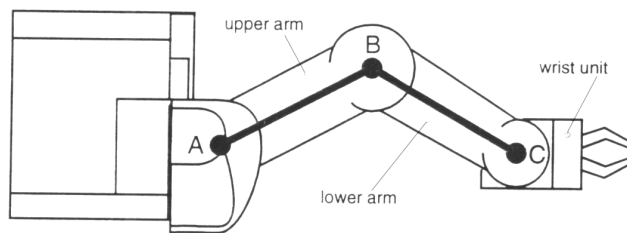


Figure 2: Degrees of freedom of a human arm.

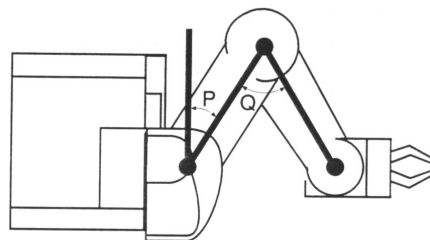
The UMI-RTX robot has precisely those minimal 6 degrees of freedom, enough for finding a solution for this robot arm in the domain of chess playing.

3.3 Radial motion of the UMI-RTX robot

The upper arm and lower arm have the same length: AB equals BC :

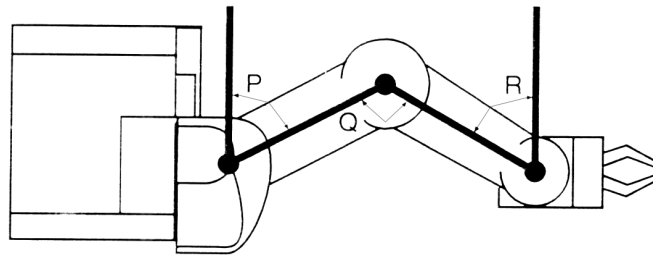


This means that you can move the wrist in a straight line outwards from the column - a radial line between the shoulder and wrist spindles - by rotating the two parts of the arm, making sure that angle P is always half angle Q :



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).

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: SimpleChess

Introduction

In this assignment, you will program a chess engine for a simplified version of chess. You will be using the minimax algorithm to implement the chess engine. Minimax allows the computer to evaluate all possible moves up to a certain depth and then choose the one that maximizes the win, given that an ideal opponent plays against you.

The framework of with the assignment contains the basic functionality to play chess. It is now your task to implement the minimax function (plus alpha-beta optimization) and all helper functions necessary to make the chess engine work correctly.

SimpleChess

Chess itself contains many different and special rules, which can take some time to implement. In order to make this assignment more containable and less tedious to implement, we will be looking at a simplified kind of chess game called SimpleChess. In SimpleChess, the only chess pieces used are the king, rook and pawn. While the king can still reach any adjacent square (horizontal, vertical and diagonal) and the rook can still reach all horizontal and vertical squares, the pawn always only moves one place forward.

You may use the following simplifications:

- The only pieces on the board are the king, pawn and rook.
- A pawn cannot promote when it reaches the other side.
- A pawn can only move one piece forward at a time (even the first turn).
- There is no checkmate; the game is over when one of the kings is hit.
- You do not need to worry about stalemate until later in the assignment

The framework

Helper functions

The static classes `Material` and `Side` are just used to provide enums for board pieces and the player turn, so you can refer to a rook with `Material.Rook` instead of a string/char and refer to the white player with `Side.White`. A chess piece is then represented as an `Piece` object with two properties: `side` and `material`.

The helper functions `to_coordinate` and `to_notation` when converting a position in x,y-notation (e.g. (2,5)) to a board coordinate in chess notation (e.g. "c3") and vice versa. Note: (0,0) corresponds to "a8".

The ChessBoard class

A single board state is specified by the `ChessBoard` class. The `ChessBoard` class contains two variables. The `turn` variable specifies whether the white or black player is on turn. The `board_matrix` variable contains an 8 by 8 2d-array array with at each cell either `None` if no chess piece is present at that position or the `Piece` object specifying the chess piece that is present at that position. You can use getter function `get_boardpiece` and setter function `set_boardpiece` to retrieve and set a chess piece at a specified position.

Apart from representing a board state, the `ChessBoard` class is also responsible for reading a board configuration from input (`load_from_input`) and printing itself (`__str__`). It also contains the `make_move` function, which, given a certain move, returns a new board configuration with that move executed.

Lastly, it contains the `legal_moves` function, which should return all the possible moves for the current board state. You will need to implement this function yourself.

Other classes

The `ChessComputer` class provides functions to calculate the best computer move using minimax or alphabeta. You will need to implement both functions. The function `evaluate_board` should give a score to each board configuration as to how favourable this configuration is for the white or black player.

Lastly, the `ChessGame` class contains functionality for playing the game in the command line and reads input from the user. This class will automatically load the configuration in `board.chb` in the same directory. If you would like to open another board configuration, you can specify this as a command line argument (e.g. `python chessgame.py board.configurations/capture_king1.chb`). You will not need to make changes to this class.

The assignment

In order to implement the minimax algorithm, we first need a few helper functions. More specifically, we need a function that gives us the moves between states and a scoring function for states.

4.2 Implementing `legal_moves`

The first function we need to implement when writing the chess engine is a function that returns all the legal moves of the rooks, pawns and king that are possible given a certain board configuration. To this end, you will need to implement `legal_moves` in the `ChessBoard` class. Make sure you test this functionality before continuing with the next part, since this function will be used in later parts of the assignment.

4.3 Implementing `evaluate_board`

Secondly, we need to assign a score to every board position. We will do this in the `evaluate_board` function in the `ChessEngine` class. A simple way to give a score to a board configuration is to give a score for each pawn, rook and king white has on the board and perform the same negatively for the material that black has on the board. This way, a board configuration in which a rook of the black opponent is captured will have a higher score. Make sure you form the score in such a way that a rook has a higher score when captured than multiple pawns and capturing the king the highest (since then the game is won). Besides counting material, you would also like to prefer strategies for capturing material that take fewer turns than long strategies (e.g. if you can capture a king in 2 turns, you should not take 4 turns to do so). You can use the `depth_left` variable to take this into account when designing your scoring function.

You are free to implement this scoring function any way you like, but for full credits you do need to make sure that the minimax algorithm will perform optimally on all the board configurations that are included with the assignment.

4.4 Implementing `minimax`

If you finished the scoring function and the legal moves function, you have essentially all the basics you need to implement the minimax algorithm. We are going to implement a depth-bounded minimax algorithm. This means that we will only look forward a pre-defined amount of moves. After this depth is reached, the current board position is evaluated using the scoring function, even though it might not yet be an endposition of the game. Use `legal_moves` to enumerate all the moves possible, `make_move` to execute this move and `evaluate_board` to give a score to a board position after we have reached the maximum depth.

Once you have the minimax algorithm working, you can implement the alpha-beta optimization in function `alpha_beta`.

The different parts of the assignment are weighed as follows:

- (3 pt) Implement `legal_moves` in the `ChessBoard` class
- (2 pt) Implement `evaluate_board` in the `ChessEngine` class
 - For full credits, the chess engine should handle all of the board configurations attached in the assignment well.
- (2 pt) Implement `minimax` in the `ChessEngine` class
- (1 pt) Implement `alphabeta` in the `ChessEngine` class
- (2 pt) Implement two of the following enhancements:
 - Make the chess engine able to prevent stalemate situations
 - Include at minimum two other chess pieces, such as the queen and the knight

4.5 Task 2: A chess playing robot arm

Introduction

In this assignment you will be implementing inverse kinematics in a VPython simulation in order to make a robot arm play chess. There are three parts to this assignment:

- Cartesian Coordinate conversion
- Inverse kinematics
- High path planning

The high path planning is a series of instructions the robot arm will need to follow in order to move a piece. The inverse kinematics part will tell the robot exactly what angles the individual arm joints need to have so it will be at a desired location. And the coordinate conversion means that you translate a board position, e.g. "a1", to a real world coordinate. You can determine yourself in what order you want to complete these modules as they work independently. However, we suggest starting with high path planning and coordinate conversion. These are the easier components, so it will allow you to familiarize yourself with python again.

VPython simulator

The simulator uses "Classic" VPython 6 in combination with Python 2.7, and the version we use works by default on Windows and not on Linux (yes, we managed to find a Python package that does not work on Linux by default...)

Installation

Windows

For more download instructions, you can refer to http://www.vpython.org/contents/download_windows.html.

Check if you have Python 2 installed on your computer, at the following location:

```
C:\Python27
```

If this is the case you can download either the 32-bit or the 64-bit VPython depending on what python version you have installed. To check what version you have installed use the command:

```
C:\Python27\python.exe
```

And you should see a line of text showing you your version details, including the bits. If you have Anaconda installed for Python 2, you can use this command instead:

```
conda install -c https://conda.binstar.org/mwcraig vpython
```

After you install the correct VPython version, it should work right away.

Linux

For more download instructions, you can refer to http://www.vpython.org/contents/download_linux.html. Linux requires slightly more work and requires you to use Wine (basically a bit of windows inside Linux). The installation instructions of Wine, can be found here: <https://wiki.winehq.org/Ubuntu>. Depending on the version you installed, pick the correct files from the download_linux link, and place them in the correct folders (once again check the bit versions that you use) and follow the *Installing VPython 6 under Wine* instructions on the web page.

What is provided?

In order for you to find the correct parameters for the robot, without having to go measure it yourself, and probably end up making mistakes, we provide you with the Denavit-Hartenberg convention for robot joint parameters of the simplified UMI in Table 1.

| Joint _{<i>i</i>} | Θ_i | α_i | a_i | d_i | Joint range |
|---------------------------|------------|------------|-------|--------|----------------|
| riser/zed | 90.0 | 0.0 | 67.5 | 1082.0 | 0.0 - 925.0 |
| shoulder | 0.0 | 0.0 | 253.5 | 95.0 | -90.0 - 90.0 |
| elbow | 0.0 | 0.0 | 253.5 | 80.0 | 180.0 - 110.0 |
| wrist | 0.0 | -90.0 | 0.0 | 90.0 | -110.0 - 110.0 |
| gripper | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 - 50.0 |

Table 1: The Denavit Hartenberg notation of the simplified UMI robot arm. The d_i of the wrist includes the d_i of the gripper. Distances are given in mm (**the simulator uses meters**)

The simulator that will be provided to you consists of the following files:

- **umi_chessboard.py** - Contains the representation of the chessboard and the pieces. Also contains functions that have to do with the translation and rotation of the board.
- **umi_common** - Contains some functions that are used across all the files, also included the functions for file interactions.
- **umi_parameters** - This file contain all the parameters of the robot arm, but it is your task to retrieve this information, and add it correctly.
- **umi_simulation** - Contains the simulator for the robot arm, and all the functions that have to do with the visualizations. This file is also the main function to call, if you want to run the program. The sliders work per default, as long as you have the correct parameters.
- **umi_student_functions** - Empty shells of functions for you to fill, such that all functionality works correctly.

High path - 2 points

In order to play chess, the robot arm needs to be able to move a chess piece from one location to another. To achieve this the robot arm will need to follow a series of instructions:

1. Open gripper
2. Move to safe height over piece
3. Move to low height over piece
4. Move to piece height
5. Close gripper
6. Move to safe height over piece
7. Move to safe height over new position
8. Move to low height over new position
9. Move to piece height
10. Open gripper
11. Move to safe height

Where safe height is an arbitrary height above a board location where it is impossible to for the robot arm to interact with the pieces. Low height is right above the piece and piece height is the height where the robot arm is able to actually grab the piece. These instructions can be implemented in the **high_path(chessboard, from_location, to_location)** function

In addition, very similar to the above function, you also have a **move_to_garbage** function. With the difference that it now drops the pieces on a garbage pile outside of the board. You can choose different ways to solve this problem e.g. place the in a nice row next to the board, or just toss them in your imaginary box. These instructions can be implemented in the **move_to_garbage(chessboard, from_location, to_location)** function

Cartesian Coordinate Conversion - 4 points

Because the UMI robot arm is blind, it is difficult for him to determine what to do if you tell it to move a piece from "a1" to "a3". After all, it can not see the chessboard. Hence, you will write a function to help him out, by writing the function that returns a (x,y,z) tuple, when given a position in the form <letter><digit> e.g. "a1". For this you will use the rotation and translation of the board, as well as the position of the piece. The rotation point of the chessboard is next to the position "h8". These instructions can be implemented in the **board_position_to_cartesian(chessboard, position)** function.

Inverse kinematics - 4 points

Solve the inverse kinematics problem for the robot arm: "Given a point on the board and a height, what are the joint angles needed for the robot arm to reach this point?". Keep in mind that there is no right answer for this. Inverse kinematics generally does not have one solution as there are multiple ways to reach the same position. This means that you can implement movement that you think is best as long as it works within the simulator. Inverse kinematics can be implemented in the **apply_inverse_kinematics(x, y, z, gripper)**

4.6 Task 3: Go, where no one has gone before

This section is intentionally left blank, content to be provided by you. We expect a research proposal with your plans in the third week.

The only requirement is that you come up with an idea of a thing 'never done before' in this course.

An overview of things that were already tried:

- <https://staff.fnwi.uva.nl/a.visser/education/ZSB/2016/Experiment/>
- ...
- <https://staff.fnwi.uva.nl/a.visser/education/ZSB/2004/Experiment2004/>

5 General lab course instructions

5.1 General instructions

During the first two weeks you will be working in *pairs*. For the second two weeks you will be working in a *quartet*. By making you work in groups 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 etcetera, 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 the first two tasks you will hand in your software (including documentation) per pair.

For the last two weeks we will expect a more extensive research proposal and technical report.

Details how to write a research proposal and technical report can be found at the *Practicum Academische Vaardigheden*-website:

- <http://practicumav.nl/onderzoeken/onderzoeksvoorstel.html>
- <http://practicumav.nl/onderzoeken/rapport.html>

Upload your reports in a printable format, that is: PDF (so no tex-code, plain text-files or Word-files).

5.2 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

5.3 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 there is only one robot and limited time available. When an assistant is confident in your implementation, he could send you to the robotlab to test it on the actual UMI-RTX the robot. In that case, 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 5.4 how to append your code. Specify your names and student-ids in the mail.

5.4 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.

5.5 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.

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. The manual was subsequently modified by Gerben Venekamp, Daan van Schaijk, Nikos Massios, Paul Ruinard, Matthijs Spaan, Olaf Booij, Julian Kooij, Tim van Rossum, Elise Koster, Michael Cabot, Robrecht Jurriaans, Toto van Inge, Nick de Wolf and Boas Kluiving.