

BACHELOR THESIS
COMPUTER SCIENCE



RADBOUD UNIVERSITY

Writing with Dobot Magician

Author:
Dennis Kleverwal
s4598164

First supervisor/assessor:
dr. P.W.M. Koopman
pieter@cs.ru.nl

Second assessor:
dr P.M. Achten
P.Achten@cs.ru.nl

April 4, 2019

Abstract

In this thesis a solution is given to write on the curved surface of a cylindrical object in personal handwriting with the Dobot Magician. The Dobot Magician is a robotic arm developed for practical education by the company Dobot. The Dobot Magician can not turn its arm aside, so a non trivial solution was needed. The points to control the Dobot Magician are created by converting png images of characters to bmp files and trace those with autotrace to get svg files. Those svg files are used to obtain coordinates, which are send to the Dobot Magician by API calls provided by Dobot. Writing in personal handwriting is tried, but this is not fully accomplished. So, characters in computer generated font are used for the result. Writing on a curved surface is solved by turning the object when a threshold is reached with the suction cup of the Dobot Magician in a separate standard.

Contents

1	Introduction	3
2	Preliminaries	5
2.1	Dobot Magician	5
2.1.1	Specifications of Dobot Magician	5
2.1.2	Ways to control Dobot Magician	6
3	Programming language	8
4	Controlling the Dobot Magician with script	10
5	Representation of characters	13
5.1	Input file format	13
5.2	File format for coordinates	14
5.3	Converting png to svg	14
5.3.1	Potrace	15
5.3.2	Autotrace	15
5.3.3	Centerline-trace extension for Inkscape	15
5.3.4	Tool choice	16
5.4	From svg file to usable points	16
6	Writing words	18
6.1	Writing an character in a with computer generated font . . .	18
6.2	Writing a word	19
6.2.1	Spacing between characters	19
6.2.2	Putting characters on one line	19
6.2.3	Iterating over the characters	20
6.2.4	Scaling when out of bound	20
6.3	Writing words in personal handwriting	20
7	Writing on a curved surface	22
7.1	Turn the curved object	22
7.1.1	Implementation 1: write and turn at the same time .	23

7.1.2	Implementation 2: Turn last point before threshold to starting point	24
7.2	Writing a whole word on a curved surface	26
8	Future Work	28
9	Related Work	29
10	Conclusions	30
A	Appendix	32
A.1	dobot.py	32
A.2	DobotControl.py	36

Chapter 1

Introduction

The Dobot Magician is a robotic arm of the company Dobot, developed for practical training education purposes. Through the variety of extensions, the small size and the relatively low cost, is the Dobot Magician also usable for a lot of other projects. The Dobot Magician is relatively new, what makes that there is not that much known about the performance of the Dobot Magician beside the specifications Dobot wrote on his own website.

One of the things Dobot Magician can do is writing on flat objects. In this thesis we research whether the Dobot Magician is able to write on a curved surface of a cylinder shaped object and whether this can be done in personal handwriting. So, the research question is:

How can the Dobot Magician write on a curved surface in someones personal handwriting?

Writing on a curved object with the Dobot Magician is not trivial, because an 2D images has to be written in 3D on a cylindrical object and the arm of the Dobot Magician can not turn aside to stay (almost) perpendicular to the surface. Instead of turning the pen aside, the cylindrical object will be turned. Doing it this way, the Dobot Magician can write always from top and thus (almost) perpendicular.

Writing in a personal handwriting is also not trivial, because the ink of the pen and the lines that are not smooth can cause multiply problems.

To get a answer on the research question, we need at least answers on the following questions:

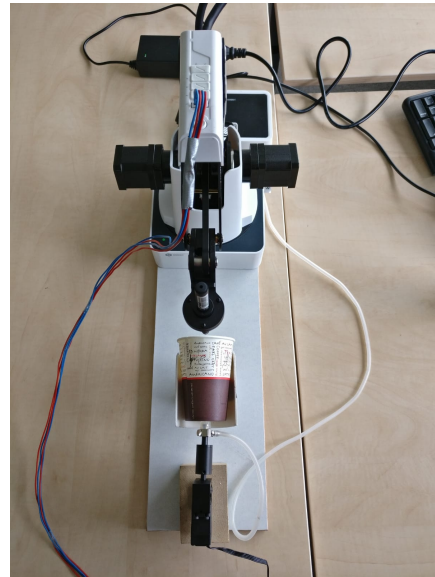
- Which programming language is most usable for programming the Dobot Magician?
- How can the Dobot Magician write in someones handwriting?
- How can the Dobot Magician write on a curved surface?

Through the process of researching the setup as shown in figure 1.1 is build in different steps.

The other part of this thesis looks like this: Chapter 2 tells something more about the Dobot Magician. Chapter 3 gives the supported programming languages and the program language choice for this thesis. The mode to control the Dobot Magician in this programming language is chosen and the most used functions in this mode are explained in Chapter 4. Chapter 5 explains which file formats are used to represent the characters and which tool is used to convert. It also covers the parsing of the svg files to get usable points of them. How to write a word in personal handwriting with those points is described in Chapter 6. However writing in personal handwriting is not accomplished, because lack of time, the stages before are. Chapter 7 is about the implementation of the algorithm to write on the curved surface of a cylindrical object. Chapter 8 is the future work chapter where is told what could be improved and what still needs to be finished. Chapter 9 shows where this thesis is comparable to. The last chapter is Chapter 10, where the conclusion of this thesis is given.



(a) Setup from the side.



(b) Setup from above.

Figure 1.1: Setup of the Dobot Magician.

Chapter 2

Preliminaries

2.1 Dobot Magician

To understand this thesis better, it is important to know the specifications and the ways of controlling the Dobot Magician. This is stated out in this subsection of the preliminaries.

The Dobot Magician (figure 2.1) is a small robotic arm developed for practical training education, but can also be used for other kind of projects. Dobot Magician can use its different end tools to do different tasks, such as printing, laser engrave, writing and pick up something with the gripper or suction cup. Those tools are delivered with the Dobot Magician, but the Dobot Magician could also be extended with a micro controller or other kinds of hardware.



Figure 2.1: The Dobot Magician with the pen end tool.

2.1.1 Specifications of Dobot Magician

To work with the Dobot Magician, it is important to know what the Dobot Magician is able to do physically and what not.

The size of the Dobot Magician is 158mm x 158mm, but when the arm stretches out the radius becomes 320mm.

To achieve this reach among other things, the Dobot Magician has in total four joints driven by servo motors. The first three joints are shown in figure 2.1. Those joints causes that the Dobot Magician can move in three different directions: x-, y-, and z-direction. The angle of the joints are limited and are as follow:

- J1: -90° to 90°
- J2: 0° to 85°
- J3: -10° to 95°

The fourth joint exists only if the gripper or suction cup is used. The angle reach of those two end tools is -90° to 90° .

The servo motors for the joints have a maximum speed. The servo motors in the Dobot Magician have a maximum speed of $320^{\circ}/s$ and the servo motor in the gripper and suction cup have a maximum speed of $480^{\circ}/s$ at a workload of 250 g. The maximum workload of the Dobot Magician is 500 g.

2.1.2 Ways to control Dobot Magician

Dobot provides also DobotStudio which could be downloaded from the website of Dobot[5]. DobotStudio is a GUI were the user could use the Dobot Magician in 8 different ways, which are listed and explained very short below:

- Teaching and Playback - In this part of the GUI, the user could create a program for the Dobot Magician by moving the arm to the right places by hand. Every time the unlock button to free up the motors in the inflection points is released, the program will save the current state (Teaching). After this teaching part, the user can play the program and the Dobot will go to every point saved before (Playback).
- Write and Draw - The user could draw and write with this part of DobotStudio, as the name reveals already. This could be done by dragging or inserting images or text into the semicircle. This semicircle is the reach of the Dobot Magician. After clicking "start", the Dobot Magician will draw all the things dragged on the screen.
- Blockly - The user could code their own program for the Dobot Magician here, with the language Blockly. This is a language with literally blocks of code which could be dragged in or below each other.

- Script - A Python program could be written here to control the Dobot Magician.
- LeapMotion - The Dobot Magician could be controlled by motion in this part of Dobot Magician. Hold your hand above the motion sensor and move it after. This will cause that the head of the arm follows the movements of the hand. By closing the hand, the gripper will close or the suction cup will suc. Opening the hand does the reverse.
- Mouse - The user could use the mouse in this part of the GUI to control the Dobot Magician. After typing "v" the head of the arm will follow the mouse.
- LaserEngraving - This part of the GUI is as the name tells to engrave with the laser. It works the same as Write and Draw, but you could not write words or sentences in it.
- 3DPrinter - Here the firmware of the 3D printing head will be loaded. With help of this firmware the objects could be printed.

Beside DobotStudio, the Dobot could also be controlled by programs in various programming languages via an API/communication protocol. Over 20 programming languages are supported, which is covered in chapter 3. The Dobot Magician has also 13 extensible interfaces to connect the end tools and hardware written about before. This makes that the Dobot Magician together with the high amount of programming languages could be used easily for a diversity of projects.

Chapter 3

Programming language

In this chapter we state out the different programming languages usable and motivate the choice of the programming language used in this project.

Dobot Magician supports, as said before in section 2.1, over 20 programming languages according to their website. However, only 15 names of languages (and software stacks) can be derived of the demo projects on their website[5]. Those are:

- | | | |
|-----------|---------------------|-----------|
| • Scratch | • Visual studio C++ | • STM32 |
| • Python | • C# | • Arduino |
| • Java | • Visual Basic | • IOS |
| • C++ | • Qt5.6 | • Android |
| • LabVIEW | • ROS | • Matlab |

Demo projects are downloadable for most of those programming languages via the website of Dobot. The benefit of the demo projects is the module they provide. The functions in the module are created to control the Dobot in that specific programming language. All languages could use JOG(does only a step in one direction per API call), CP(target point or increment can be given in this mode), ARC(to move the arm from point to point in an arc) and PTP(Point To Point) mode to communicate the movements from the computer to the Dobot Magician, so this makes no difference. It is obvious to choose one of those languages which has already the code for the API calls, because it less work programming and there is enough to choose from.

The best language to choose of the languages named before is the language which the programmer likes the most, because all the programming languages can use the API calls. In this case Java or Python are preferred. Python is already used as script language in DobotStudio as said before in

section 2.1.2. For this reason this seems to be the best language of the two to work with. Also there are slightly more general Python libraries available than general Java libraries, which could make the programming easier in the way of less time consuming. The reason for programming outside DobotStudio is because other IDEs are more preferred.

Chapter 4

Controlling the Dobot Magician with script

In this chapter the mode choice is made and explained. After this, the most important and the most used commands in this thesis of this mode are explained to get an idea of how they work and how they are supposed to be used.

The PTP mode will be used in this thesis. The PTP mode has different movement types, namely MOVJ, MOVL and JUMP. Those three movement types have also a subdivision, namely moving on the basis of a new point, joint angles or increments of points. MOVJ goes from point A to point B and does not care about the path in between. MOVL goes from point A to point B in a straight line. JUMP goes also from point A to point B in a straight line, but the end of the arm goes first up, does then the line and after the line the arm goes down. The JUMP movement type is very handy, because there will be lines that are not connected, for instance the character i. To go from the dot to the line, the pen has to go up first before it moves. Otherwise the Dobot Magician draws a line between them. The other modes does not have this movement types.

CP mode is on a second place, because CP mode could also move between two points, but there is no jump movement type and otherwise more API calls are needed to achieve this. JOG and ARC are not an option. The Dobot Magician has to make a lot of diagonal movements, which results in a lot of API calls with JOG. ARC is not an option, because characters consists not only out of circles and we do not want to use multiple modes. Otherwise the program to write words will get more complicated.

The dobot files in the demo projects consists a variety of functions and enumerations. The enumerations are mostly used for choosing a certain state of the Dobot Magician. The functions in the module are in general for connect and disconnect the Dobot Magician and change the state with getters and setters. Not all those functions are needed for this thesis, because

we chose the PTP mode and there are also for instance functions for setting up a WiFi connection and for the other modes.

Some functions will be used more than others. The ones that are most important in this thesis are pointed out shortly[1]:

- `ConnectDobot/DisconnectDobot` - To use Dobot Magician, a connection is needed between the Dobot Magician and the computer. The baud rate that needs to be set to connect is 115200 bps for USB. For disconnection nothing is needed.
- `SetQueuedCmdStartExec/SetQueuedCmdStopExec` - Those functions are needed to start and stop the queue with instructions for the Dobot Magician. They are not needed if the choice is made to do not queue them, but execute them immediately. The queue is needed for movements and setting parameters to execute them after each other. Otherwise the Dobot Magician will execute those command when the command before is not ready yet.
- `GetPose` - This function returns the coordinates of the different axis. It is important to know where the arm is at the start of the program. If only two of the three axis of the coordinate need to change, the other one needs to kept the same. This is only possible if this axis is known on forehand.
- `SetPTPCmd` - This function takes a mode and x, y and z coordinates, a head parameter(which is an angle) and a queue parameter. To move the Dobot Magician to a certain point with the pen, the x, y and z coordinates of this point needs to be filled in as parameter. x, y and z could also be angles dependend on which mode is used. The mode is the movement type which are explained above. The head parameter has only effect if the gripper or suction cup is connected, because those tools can only turn their head. The pen has no motor to turn around, so it can not. The queue parameter is to state if a command needs to be executed immediately or after all commands are called and queue it till then.
- `SetEndEffectorSuctionCup` - This function takes especially the parameters `enableCtrl` and `suc`, which are two booleans to put the air pump on or off and to tell the pump that it has to suc or blow.
- `dSleep` - This function could be used to create more time between different commands, like wait till the queue is empty before disconnecting.

To give an example of using those PTP commands, example code is given listing 4.1 which draws a square of 10mm x 10mm. Set up and break down the connection is included. Setting all the starting params is let out.

```

1  #set up the connection
2  api = dType.load()
3  state = dType.ConnectDobot(api, "", 115200)[0]
4
5  if state == dType.DobotConnect.DobotConnect_NoError:
6      #get the current position
7      x, y, z, rHead = dType.GetPose(api)
8
9      #code to draw a square of 10mm x 10mm
10     dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, x +
11         10, y, z, rHead, 1)
12     dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, x +
13         10, y + 10, z, rHead, 1)
14     dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, x, y
15         + 10, z, rHead, 1)
16     lastindex = dType.SetPTPCmd(api, dType.PTPMode.
17         PTPMOVLXYZMode, x, y, z, rHead, 1)
18     dType.SetQueuedCmdStartExec(api, lastindex)
19     while lastIndex[0] > dType.GetQueuedCmdCurrentIndex(api)
20         [0]
21         dType.dSleep(100)
22
23     #break down the connection
24     dType.setQueuedCmdStopExec(api)
25     dType.disconnectdobot(api)

```

Listing 4.1: Draw a square with PTP mode in python

Chapter 5

Representation of characters

In this chapter we will decide about the representation of the characters. The algorithm that we are creating for writing needs some input to know how it has to write the different characters. There exists a lot of different file formats with different properties, but which one suits the best. Two different file formats are chosen to make use of the different properties of the two different file formats.

The Dobot Magician needs coordinates or angles to move, so we could choose immediately a file format that is based on points (vectors). However, one of the goals is writing words in handwritten font, so the characters have to be photographed or scanned in. Pictures and scans are never vector based, but they are bitmap based. A conversion is needed between the bitmap based file format and the vector based file format to use them both.

5.1 Input file format

The input file format of the algorithm that will draw the characters is png. Png is chosen as input for a few reasons. First of all, when the Dobot Magician writes in personal handwriting, pictures of characters are used. So, a file format based on pictures should be the input of the algorithm that controls the Dobot Magician. Png is one of this file formats together with jpg. Second, using a widely used format ensures that the program could be extended easily for other purposes by other people.

Jpg was also an option, but after comparing both, png was a better option[8]. Jpg pictures are of less quality, because they are lossy compressed. Also a disadvantageous property is that the lossy compression could not be reversed. This makes that the jpg files can not be converted to bmp, which has no compression. Why a conversion to bmp is needed, is explained in Section 5.3.

A disadvantage of png over jpg is the higher amount of memory they use. Even though the memory size of png is bigger as jpg, the difference

is relatively small. The files are local on a computer, so the difference is negligible.

5.2 File format for coordinates

On the other hand, a file format is needed that consists of points, because the Dobot Magician could only move with coordinates or arm angles given. DobotStudio has already the ability to draw images and characters with the Dobot Magician. Those images or characters have to be of svg, plt or dxf format, because those formats are based on vectors which are almost points. Svg stands for scalable vector graphics and is the open standard for vectors. Plt is a vector based plot file developed by Autodesk. Dxf is a drawing exchange format for 3D and also developed by Autodesk. Dxf is mostly used by CAD programs.

Bmp is not based on vectors, but is also supported. DobotStudio converts the bmp images to svg format to use them. Png and jpg are also allowed, but then DobotStudio converts the file also to svg format first. In addition to the vector file formats mentioned before, there are a lot more vector based file formats, such as fig, sk, ai, wmf, cgm, ps, eps, emf and skl. Most of those file formats are specific developed for certain programs.

However DobotStudio could convert images to vectors, it could not be used for the conversion. The reason for this is that there is no way to convert the images in the write and draw part and program around it, because program is in another part of DobotStudio. DobotStudio has also no command line instructions to use its conversion ability.

So, to use the input file format, a conversion is needed from png to one of the vector based file formats mentioned before by another tool.

5.3 Converting png to svg

After some search on the web, for tools or python packages to convert images from png to a vector based file format, it was clear to convert to svg. Most of the file formats were especially for specific programs. Svg is the most general file format, because it is the open standard. Also there were a lot of suggestions for converting png to svg. Three tools are found that can convert png files or bitmap files (which is the raw version of a png file) to svg files. They are usable in python or in command line and thus also in python. This was a requirement.

Only one of the three tools could take a png file directly as input. The other two tools can only use bitmaps(bmp) files. This means that they need a step in between. Converting to bmp before will obvious be this step. This is not difficult, because png files are basically compressed bmp files. There are already python packages that can convert png files to bmp files. The

one that is used for this thesis is Pillow[7]. After converting the png file to bmp, the other two tools can also be used. An example of a png/bmp image to compare with the outcome of the different tools is shown in 5.1a. This image is shown for both formats, because they look exactly like each other.

5.3.1 Potrace

Potrace is a tracing tool for creating vectors out of bitmaps[9]. A disadvantage of this tool is that it traces the outlines of an image. An image of a single lined character will be converted to a double lined character. This is not how it should be, because the images has to be the same as the original character.

Even though it does not the right thing that is needed, it creates really good vectors which looks almost exactly like the original character. An example of a potrace result is in figure 5.1b. This image is bigger here, because it is stretched out since it has no background. In the file it is the same size as the png.

5.3.2 Autotrace

Autotrace[2] traces the images quite well and gives a reasonable output svg. It is quite hard to find the right settings for the thresholds and other parameters. When the right settings are found for a character, it could be that they mess up another character.

There is no python import for autotrace, so there is a sub process needed to use autotrace with bash commands in the background. An example of a svg traced by autotrace is shown in figure 5.1c.

Autotrace creates also some noise in colors other than the black color of the character, so the parameter -color-count is set to 2, which shows only the two most used colors. Actually it should be set to 1, but this will filter out the color that occurs second most, which is the black colored character. The color that occurs the most is a random white line in the background. This white path is also visible in figure 5.1c. The white path can be filtered out very easily in python by checking the hexadecimal color value when iterating over the paths, so it does not matter.

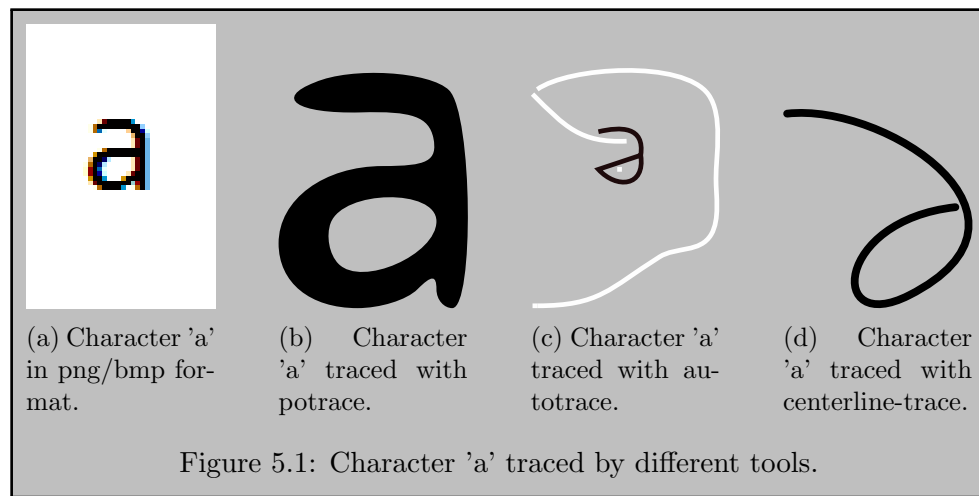
5.3.3 Centerline-trace extension for Inkscape

The centerline-trace[4] extension for Inkscape is also a tracing tool for creating vectors out of bitmaps. This program makes use of autotrace and traces the center of the characters, but the outcomes of the centerline-trace extension does not even look like the original character. An example of a svg traced by the centerline extension of Inkscape is shown in figure 5.1d. This image is bigger here, because it is stretched out since it has no background. In the file it is the same size as the png. The svg image looks like a

drawing of a child. This is strange, because the examples that were given in the README file looks exactly like the input png images. It is also strange that autotrace creates images which look like more on the original one in comparison to the centerline-trace that makes use of autotrace.

5.3.4 Tool choice

In advance the centerline-trace extension for inkscape thought to be the best, but the output was not as expected. Potrace returns really good svg files, but the characters are double lined. This is not what is needed, so potrace is not a option. In the end autotrace gives the best result that are the most equal to the original characters. For this reason autotrace is used as tool to convert the bmp file into a svg file.



5.4 From svg file to usable points

The svg file, which is written in XML contains paths with points, but the points in between are also needed. The points in between lay not always on a straight line between the two points, but can also lay on a circle or on another shape. The kind of shape between the two points is indicated in the svg file by a character. Figure 5.2 shows a svg file with its corresponding image. If a line was drawn between the four points in the file in figure 5.2a, then the image was never shaped like in figure 5.2b. They will have spikes in that case.

To obtain enough points of the path to represent the image well, the function `svg2paths2` of the `svgpathtools`[10] package is used. This function reads all the paths and creates sub paths from the svg file, which are called segments. A segment is exactly one shape. The function saves them in a 2D array as a list of paths containing lists of segments.

To get the points of the segment, the function `point()` is used on the segments with a decimal number as parameter. This number should be greater or equal to 0 and smaller or equal to 1 to get the right points. The segments have a polynomial representation, which is used together with the parameter to get the point. So, for example filling in 0.5 returns the point exactly at the middle of the segment.

The output of the point function is still not directly usable, because they are returned as a complex number, namely $(x+yj)$. This is an easy way to store an x- and y-coordinate without using a tuple. After separating the complex number into two parts, a real part which is the x-coordinate and a imaginary part which is the y-coordinate, there are finally usable coordinates.

```
<?xml version="1.0" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      viewBox="0 0 40 40"
      height="40"
      width="40">
  <path d="M10 10
          C8 30 25 5 20 20"
        style="stroke:#000000; fill:none;" />
</svg>
```

(a) Svg file.



(b) Corresponding image of svg file.

Figure 5.2: Svg with its corresponding image.

Chapter 6

Writing words

Now it is the task to write the words in personal handwriting in 2D. This is build up in three steps to make it easier. The first step is writing a single character in a with computer generated font. If a single character is succeeded, a whole word is the next step. The last step is writing the word in personal handwriting, but after a few weeks this seems to take too much time. The font is chosen for the steps before writing in personal handwriting, because the color of the characters is equally distributed and the characters are well shaped. In this way, it becomes easier to accomplish the main structure of the algorithm before focusing on the personal handwriting.

6.1 Writing an character in a with computer generated font

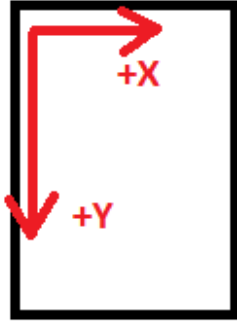
The goal now is writing a character with the points obtained from the svg file, in the way explained in section 5.4.

However, those are relatively to the upper left corner of that file, which is point (0,0). So, if the points are fed to the `setPTPCmd` function from Chapter 4 to move the arm of the Dobot Magician, then the Dobot Magician will not start writing at the point where the pen is put down to write. The first point of the first segment is somewhere else then the pen is.

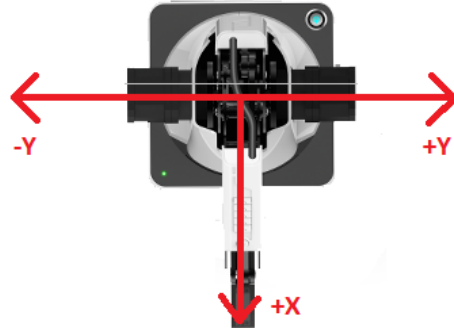
To solve this, the points should be relative to the pen instead of relative to point (0,0). This is done by subtracting the first point of the character from all the points by subtract the x-coordinate from the x-coordinate and the y-coordinate from the y-coordinate. Adding the start position of the pen to the relative positions to the pen will give the right positions.

When using the coordinates of those position in the `setPTPCmd` function, it turns out that the x-coordinates should be put on the spot of the y-coordinate and the y-coordinate on the spot of the x-coordinate. This is caused by the fact that y is the vertical axis and x the horizontal in the svg file (figure 6.1a), while for the Dobot Magician it is the other way around

(figure 6.1b).



(a) X and Y direction of svg file.



(b) X and Y direction of Dobot Magician.

Figure 6.1: X and Y directions.

6.2 Writing a word

Writing words in 2D is in basic the same as writing a character in 2D. The difference is adding space between the characters, get the character at the same height, iterating over the characters and add some checks to scale the word when it does not fit on the cylindrical object or is outside the reach of the Dobot Magician. The changes follow in the subsections.

6.2.1 Spacing between characters

Adding space between characters is not as easy as moving the pen a constant distance aside. The pen does not always end at the most right point of a character and does also not begin always at the most left point of a character. This will cause different distances between characters if only a constant is used. To solve this, the y-coordinate of where the pen ended in the left character is subtracted from the highest y-coordinate in this character. Then the space between the characters is added. The last step is adding the result of the lowest y-coordinate of character on the right minus the y-coordinate of the begin point of the character on the right.

6.2.2 Putting characters on one line

The begin points of the characters are not on the same x-coordinate as the x-coordinates of the the end point of the characters before. If only the y-coordinate changes to write the next character, then the characters will not be on one line.

To correct this, a correction value is calculated for every character and added to its x-coordinates. This is done by taking the highest x-coordinate for every character and subtract the highest x-coordinate of the first character. The highest x-coordinate is used to get the bottoms of the characters on one line instead of the lowest, even though it is against intuition. The reason is that the coordinate system of the Dobot Magician is based on its own point of view and not the users point of view.

6.2.3 Iterating over the characters

For every character in the word there is a check for existence of the corresponding svg file to save resources. If it does not exists, then it will be created with autotrace, which is the chosen tool in section 5.3. The second step is getting the coordinates needed for the corrections, explained in the previous two subsections. Third step is writing the actual character as described in section 6.1. The last step adding the correction for the y direction.

The code of writing a character is slightly changed too. Every first point is done in JUMP mode instead of MOVL mode, because the pen is still on the endpoint of the character before and has to go to the begin point of the next character. Those point may not be connected, so the Dobot Magician has to make a jump to the begin point of the next character.

6.2.4 Scaling when out of bound

The servo motor inside the suction cup end tool can only turn 180 degrees as written before in 2.1.1. This means that a very long word can not be written down on the cylindrical object, because the suction cup can not turn further. To solve this, the characters need a scale down to fit. However, there was no time left to implement this. The text can still be scaled by a multiplication with a scaling factor, but only manually with a command line parameter.

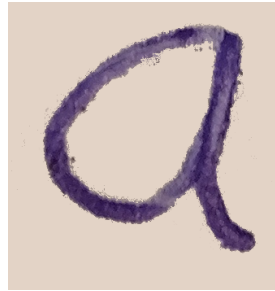
The idea was to calculate the total width of the word in current state and check if this fits on half of the object. If not, the points are multiplied with a scaling factor, calculated through dividing the half of the circumference by the total length of the word.

6.3 Writing words in personal handwriting

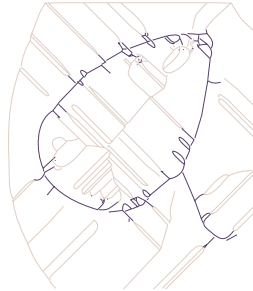
Writing words in personal handwriting instead of a with computer generated font starts with switching the images of the with computer generated font with those of the handwritten ones. Very quick turned out that the conversion of the image went not as it supposed to be. The lines of the pen are not equally filled with ink, shown in figure 6.2a. This causes outliers(in the blue line), shown in figure 6.2b. The grey colored lines created by the

color differences in the paper are not a problem and can simply be filtered out by checking the color of the paths.

Solving those outliers to write in personal handwriting will take too much time, so it is not accomplished. Even it seems to be a small step to go from a word in a computer generated font to a word in personal handwriting, it costs a lot of time.



(a) Character 'a' in png file format.



(b) Character 'a' in svg file format.

Figure 6.2: Handwritten character.

Chapter 7

Writing on a curved surface

To write on a curved surface of an object, the pen need to go up and down if the pen is above to the surface of the object. The Dobot Magician is able to do this, but it can not see how far it has to go up or down.

Also a problem is the pen that can not be turned on its side to stay perpendicular to the surface. So, the Dobot Magician need to write in another way as a person does (turn their wrist to keep the right angle between the pen and the surface). A way to write on a curved surface is to turn the object in the y-direction (horizontal) and let the arm of the Dobot Magician only move in the x-direction (vertical). In this way, the pen is always on the highest point of the curve and the height will not differ. This makes that the arm does not have to move in the z-direction (up or down).

7.1 Turn the curved object

To turn the cylindrical object, the suction cup of the Dobot Magician is used. The suction cup is supposed to be used at the end of the arm, but is place in a self made standard in front of the Dobot Magician (figure 7.1). The suction cup of the Dobot Magician has a very short cable, which is handy if it is used at the end of the arm. But for the purpose of writing on a curved surface the suction cup is not attached to the end of the arm. This will make that a longer cable is needed.

The complete cable with connectors attached is not to buy, so we had to make it our self. The output port on the Dobot Magician for the suction cup is a male jst xh-2.54 connector and the cable attached to the suction cup has a female connector. So, a cable from female to male is needed. The female connectors could only be connected to cable, so two female connectors, one female to male connector and some cable are needed to create the right extending cable.

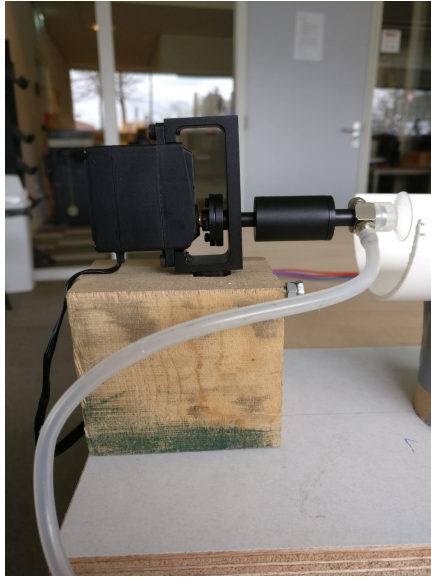


Figure 7.1: Standard for suction cup.

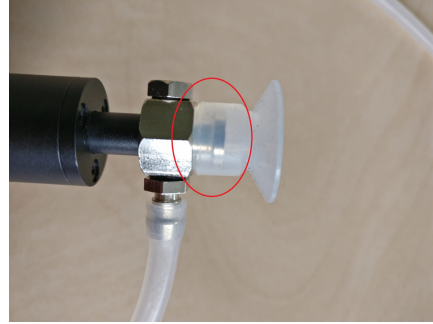


Figure 7.2: Suction cup attached to servo motor.

7.1.1 Implementation 1: write and turn at the same time

The first implementation is moving the arm only in the x-direction and the cylindrical object with the suction cup in the y-direction. Implementing was not difficult, because the angles to turn the suction cup could be calculated easily from the distance in y-direction. This is done by taking the inverse sinus of the distance divided by the radius. The result is in radians, so it is converted to degrees. Those degrees are added to the current angle to get the new angle. An example of this calculation is in listing 7.1. The radius is a default value of 30 millimeters, because we used a object with this radius, but could be set to another value via a command line parameter.

```
1 moveRad = math.asin((YNew - YOld) / r)
2 moveDegree = math.degrees(moveRad)
```

Listing 7.1: Example of calculating the angle out of the y-coordinate

Although very soon after implementing this, some difficulties came up, which were very hard to solve. If the arm writes on the object, it pushes the object only down and does only draw scratchy lines, because it can not give enough pressure on the object. The object is pushed down, because the suction cup is made of rubber which is very flexible. A way to tackle this problem is creating a standard for the object to give counter pressure.

After the standard was made and used (figure 7.3), the object was not pushed down anymore. A side effect of the fact that the arm can give pressure on the object, is that the object does not turn around while it is writing. This is caused by the rubber suction cup that is not stuck on the

metal part that rotates, but is only put over it, visible in figure 7.2. This gives the rubber suction cup the opportunity to turn apart of the metal. The resistance of the object between the pen and the standard is higher as the resistance of the suction cup on the part that turns around, so the object does not turn. This problem is hard to tackle without using another tool to turn the object or using another implementation.

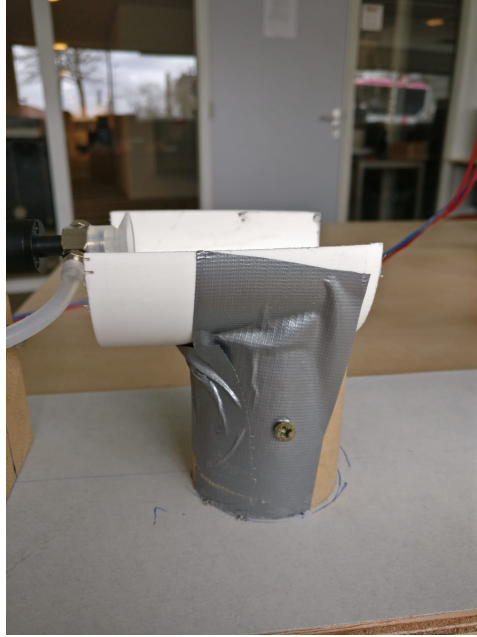


Figure 7.3: Standard for object.

7.1.2 Implementation 2: Turn last point before threshold to starting point

Another implementation is to set a threshold on the amount of millimeters between the top of the cylindrical object and the place where the pen is at that moment. This threshold is calculated according to the Pythagoras theorem, despite the surface is curved, it is seen as a straight line. The calculations for the turns does also not take the curve into account, because the Dobot writes small parts of the lines, so the distance lost is negligible.

The threshold is calculated as follow: The square root of the radius to the power two minus the radius minus the maximal difference in height to the power two. So, calculating the threshold in formula form is as follow:

$$\text{threshold} = \sqrt{r^2 - (r - d)^2} \quad \text{where } r \text{ is the radius and } d \text{ the maximal difference in height.}$$

An example in python of this calculation is given in listing 7.2.

```
1 threshold = math.sqrt(math.pow(r, 2) - math.pow(r - d, 2))
```

Listing 7.2: Example of calculating the threshold

We used 1mm for d in the project, because a lower number gives very short lines. This gives more risk on lines that do not run into each other. A higher number causes that the Dobot Magician can write further in the y-direction in once. Since the object is curved and the pen stays on the same height, the pen will go off the object at a certain point. The difference of 1mm is easy to absorb for the spring in the pen end tool. If the difference in height is higher, then the spring of the pen has to be pushed in further to write till the threshold. This gives more pressure from the pen on the higher points on the object. This can cause that the object does not turn, which happens also in 7.1.1.

If the Dobot has to write on a part further away from the top than the threshold, then the pen has to go up first. After that, the Dobot Magician has to turn the cylindrical object back until the point where the pen ended is on top. Now can the pen go back to the point where it wrote as last, but still above it. When the pen is above the point, the pen can go down and write further. In this way the Dobot puts its pen only on the object if it does not turn around. The benefit of this approach is that the object has no pressure on it when the object is turning around. In this way, the object is free to move.

The angles for turning the suction cup are calculated in the same way as in listing 7.1, but with different values for YNew. This is explained later in this section.

This angle is only used if the next point is further away as the point of the distance that is already turned plus or minus the threshold distance. This shown in listing 7.3. MiddleAt is the distance that is already turned. MaxMovementWidth is the threshold that is calculated. Dy is the coordinate of the point in the y-direction relative to the starting point of the pen instead of relatively to the upper left corner of the svg file.

```
1 (middleAt + maxMovementWidth < dy) or (middleAt -  
maxMovementWidth > dy)
```

Listing 7.3: Check if point is passing threshold

The angle to turn is calculated different for different cases. One case for when the next point is further than the threshold away of where the pen is at that moment. Then the next point must be a new segment not directly connected to the line where the Dobot Magician was working on, because the characters are not that big and all segments are divided into 10 points. In this case the new point is set on top to draw the new segment not connected to the segment before.

Another case is that the next point is closer than the threshold away of where the pen is at that moment. In this case the old point is set on top and the Dobot Magician can continue with the segment it was working on.

The counterpart of this implementation is that the character does not consist out of longer lines, but some smaller lines. This can cause that the lines do not exactly run into each other.

7.2 Writing a whole word on a curved surface

To write a word instead of only one character on the cylindrical object the code for writing a word on a flat surface as told in section 6.2 is used combined with the code to write a character on a cylindrical object from section 7.1.2. This results in the code in listing 7.4. This code is also in Appendix A.1. The line numbers in the following part are line numbers from the listing.

First the coordinates of the pen are collected. Next, some variables are initialized. Then the threshold is calculated in line 5 and the path to the python file is saved in line 6. This happens outside the for loop, because they do not change in between the characters.

After this kind of initialization, the iterations over the characters starts. If the svg file of the character is not found, then it will be created in line 11. The next step is getting the extreme coordinates of the character in line 12. Those are needed to calculate the correction in x-direction and the angle to turn to write the character. Calculating the angle happens in line 17-19 and the correction in x-direction in line 21. To calculate the correction in x-direction for every character, the maximal x-coordinate of the first character is needed and is saved in between in line 13-15. One of the last steps is writing the character in line 22. The function to write the character returns the y-coordinate of the last point it wrote and the angle the object has turned. The last y-coordinate is needed to calculate the angle to move between the characters before writing the next one. The angle that the object is turned is needed by the next character to know how far the object is turned already.

The last step is moving the pen above the starting point of the word and starting the queue with commands. This may only happen when there is a word inserted.

```
1 def writeWord(api, word):
2     x, y, z, rHead = dType.GetPose(api)
3     degreesTurned, lastYCoordinate, maxXCharFirst,
        maxYCharBefore = 0, 0, 0, 0
4     firstChar = True
5     maxMovementWidth = math.sqrt(math.pow(r, 2) - math.pow(r -
        dropOfPencil, 2))
```

```

6 path = os.getcwd()
7
8 for character in word:
9     svgfile = path + '/' + character + '_ascii.svg'
10    if not os.path.isfile(svgfile):
11        convertToSVG(path, character, svgfile)
12    maxXChar, maxYChar, minYChar = getExtremes(svgfile)
13    if firstChar:
14        maxXCharFirst = maxXChar
15        firstChar = False
16
17    moveDistanceBetweenChar = maxYCharBefore -
18        lastYCoordinate + spaceBetweenChars - minYChar
19    moveRadBetweenChar = math.asin(moveDistanceBetweenChar /
20        r)
21    moveDegreeBetweenChar = math.degrees(moveRadBetweenChar)
22
23    correctionX = maxXCharFirst - maxXChar
24    lastYCoordinate, degreesTurned = writeChar(api, svgfile,
25        maxMovementWidth, degreesTurned, lastYCoordinate, x,
26        y, z, rHead, correctionX)
27    degreesTurned += moveDegreeBetweenChar
28
29    maxYCharBefore = maxYChar
30    dType.dSleep(500)
31 if word != "":
32     lastindex = dType.SetPTPCmd(api, dType.PTPMode.
33         PTPJUMPXYZMode, x, y, z + 10, rHead, 1)
34     DobotControl.startexec(api, lastindex[0])

```

Listing 7.4: The code to write a word on a curved surface

Chapter 8

Future Work

Even the algorithm to write words on curved surfaces works, there are still some improvements to make. There are also some implementations missing, wherefore the reason is told before.

1. The tool to trace the bmp images into svg file format could be improved. Now the characters are not that good that they look exactly like the original characters in the png files. Maybe the parameters could be changed or another tool could be tried out.
2. A thing that needs to be implemented is the scaling of the words when the word is too big and goes out of reach of the suction cup. In section 6.2.4 is talked about this, but this is not implemented.
3. The outliers talked about in 6.3 could be solved to write words in personal handwriting. There was no time left to solve this in this thesis. Try to filter out little dots around the character might help or make the density of the ink of the character everywhere equal.
4. The code could be optimized. Now the algorithm checks every character before to retrieve the outer coordinates to calculate the correction and then it calculates the points again to write them. It could be an option to save the coordinates in an array or something the first time and use them later to write.

Chapter 9

Related Work

It is not the first time a machine draws on a curved surface. There are some other machines that are somehow equal, but still different.

The eggbot[6] is such a machine for instance. This machine draws on eggs, which is also a curved surface. Something that is different is that it can write and turn the egg at the same time. Notable is that Inkscape is used for the images.

Another machine is the Tag on That[11]. This machine is also able to put ink on curved surfaces, but does this with a stamp. The stamp is flexible and makes that the machine could put words on curved surfaces. However this machine can write on curved surfaces, it is totally different as the approach with the Dobot Magician. The Dobot Magician has a pen instead of a stamp.

There are also machines that can write in personal handwriting, like the Axidraw[3]. This machine has two motors, which both move in one direction separately. This is totally different as the Dobot Magician. The Axidraw can write perfect in a persons handwriting, but it writes only on a flat surface.

Chapter 10

Conclusions

The language that is most usable to program the Dobot Magician is dependent of the programmer, since all API commands are implemented for a variety of programming languages. Most likely is to use python, because this is the language used by DobotStudio to script.

The Dobot Magician seems to be able to write in personal handwriting. A way of doing this is converting pictures of characters written in personal handwriting in png format first to bmp file format and then to files in svg format. Those svg files are usable to obtain points to control the Dobot Magician.

However, this is not enough to get a character that looks exact like the original written character. To accomplish this, additional actions are needed on the png or svg file.

The Dobot Magician is also able to write on a curved surface. A way to do this is, is writing the characters in pieces till a maximum threshold. When the threshold is reached, the pen goes up and the point where the pen wrote at last is turned to the top with help of the suction cup. After the point is on top, the pen goes to the point on the object where it wrote as last and continues writing. When the character is finished, then the angle is calculated to turn the object to start with the next character. For every character is also a correction in the x-direction calculated to the character on one line.

Writing on a curved object could use still some improvements. One of this is scaling large words. Otherwise some characters are out of range of the suction cup, since the suction cup can only turn 180 degrees.

When writing the characters in personal handwriting is finished, then is combining things we said before a way to write with the Dobot Magician on a curved surface in a personal handwriting.

Bibliography

- [1] Api calls. download.dobot.cc/development-protocol/dobot-magician/pdf/en/Dobot-Magician-API-Description-V1.2.2.pdf.
- [2] Autotrace. <http://autotrace.sourceforge.net/>.
- [3] Axidraw. <https://interestingengineering.com/personal-writing-machine-will-draw-whatever-you-want>.
- [4] Centerline-trace. <https://github.com/fablabnbg/inkscape-centerline-trace>.
- [5] Dobotstudio. <https://www.dobot.cc/downloadcenter.html>.
- [6] Eggbot. <https://egg-bot.com/>.
- [7] Pillow. <https://pypi.org/project/Pillow/>.
- [8] png vs jpg. <http://fixthephoto.com/tech-tips/difference-between-jpeg-and-png.html>.
- [9] Pypotrace. <https://pypi.org/project/pypotrace/>.
- [10] Svgpathtools. <https://pypi.org/project/svgpathtools/>.
- [11] Tag on that. <https://newatlas.com/tag-on-that-printer/27694/>.

Appendix A

Appendix

A.1 dobot.py

```
1 import DobotControl
2 import DobotDllType as dType
3 from PIL import Image
4 import math
5 from svgpathtools import svg2paths2
6 import subprocess
7 import sys
8 import os
9 import getopt
10
11 num_points = 10 # number of point of beziercurve
12 scale = 1.0 # scalingsfactor of character
13 r = 30.0 # milimeters
14 dropOfPencil = 1.0 # milimeters
15 spaceBetweenChars = 6.0 # milimeters
16
17
18 def getX(point):
19     return point.real
20
21
22 def getY(point):
23     return point.imag
24
25
26 def putPenOnTop(api, dy, middleAt, r, degreesTurned, dyBefore, x
27 , y, z, rHead, dxBefore, maxMovementWidth, correctionX, dx):
28     if (middleAt + maxMovementWidth < dy) or (middleAt -
29         maxMovementWidth > dy):
30         middleAtBefore = middleAt
31         if y + dyBefore + maxMovementWidth < y + dy or y +
32             dyBefore - maxMovementWidth > y + dy:
33             moveRad = math.asin((dy - middleAt) / r)
34             middleAt = dy
35     else:
```

```

33         moveRad = math.asin((dyBefore - middleAt) / r)
34         middleAt = dyBefore
35
36         moveDegree = math.degrees(moveRad)
37
38         oldX = x + dxBefore + correctionX
39         newX = x + dx + correctionX
40         oldY = y + dyBefore - middleAtBefore
41         oldR = rHead + degreesTurned
42         newR = rHead + degreesTurned + moveDegree
43
44         dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, oldX,
45                        oldY, z + 10, oldR, 1)
46         dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, oldX,
47                        oldY, z + 10, newR, 1)
48         dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, newX,
49                        y, z + 10, newR, 1)
50         dType.SetPTPCmd(api, dType.PTPMode.PTPMOVLXYZMode, newX,
51                        y, z, newR, 1)
52
53         degreesTurned += moveDegree
54     return middleAt, degreesTurned
55
56 def convertToSVG(path, character, svgfile):
57     infile = path + '/' + character + '_ascii.png'
58     outfile = path + '/' + character + '_ascii.bmp'
59     Image.open(infile).save(outfile)
60     bashCommand = "autotrace " + outfile + " -centerline -color-
61                   count 2 -corner-threshold 90 -corner-surround 1 -input-
62                   format bmp -output-file " + svgfile
63     process = subprocess.Popen(bashCommand.split(), stdout=
64                                subprocess.PIPE)
65     process.communicate()
66
67 def writeChar(api, svgfile, maxMovementWidth, degreesTurned,
68             dyBefore, x, y, z, rHead, correctionX):
69     middleAt, ybegin, xbegin, dxBefore = 0, 0, 0, 0
70     begin = True
71
72     paths, attributes, svg_attributes = svg2paths2(svgfile)
73     for index, path in enumerate(paths):
74         if "stroke:#0" in attributes[index]['style'] or "stroke
75           :#1" in attributes[index]['style'] or "stroke:#2" in
76           attributes[index]['style']:
77             if begin:
78                 xbegin = float(getY(path[0].point(0)))
79                 ybegin = float(getX(path[0].point(0)))
80                 begin = False
81             for indexseg, segment in enumerate(path):
82                 for i in range(num_points + 1):
83                     dx = (getY(segment.point(float(1) /
84                                                  num_points * i)) - xbegin) * scale

```

```

76         dy = (getX(segment.point(float(1) /
77             num_points * i)) - ybegin) * scale
78         middleAt, degreesTurned = putPenOnTop(api,
79             dy, middleAt, r, degreesTurned, dyBefore,
80             x, y, z, rHead, dxBefore,
81             maxMovementWidth, correctionX, dx)
82         if indexseg == 0 and i == 0:
83             mode = dType.PTPMode.PTPJUMPXYZMode
84         else:
85             mode = dType.PTPMode.PTPMOVLXYZMode
86         dType.SetPTPCmd(api, mode, x + dx +
87             correctionX, y + dy - middleAt, z, rHead
88             + degreesTurned, 1)
89         dxBefore = dx
90         dyBefore = dy
91     dType.dSleep(500)
92     return dyBefore, degreesTurned
93
94 def getExtremes(svgfile):
95     ybegin, xbegin, maxXChar, maxYChar = 0
96     minYChar = sys.float_info.max
97     begin = True
98
99     paths, attributes, svg_attributes = svg2paths2(svgfile)
100     for index, path in enumerate(paths):
101         if "stroke:#0" in attributes[index]['style'] or "stroke
102             :#1" in attributes[index]['style'] or "stroke:#2" in
103             attributes[index]['style']:
104             if begin:
105                 xbegin = getY(path[0].point(0))
106                 ybegin = getX(path[0].point(0))
107                 begin = False
108             for indexseg, segment in enumerate(path):
109                 for i in range(num_points + 1):
110                     dx = (getY(segment.point(float(1) /
111                         num_points * i)) - xbegin) * scale
112                     dy = (getX(segment.point(float(1) /
113                         num_points * i)) - ybegin) * scale
114                     if dx > maxXChar:
115                         maxXChar = dx
116                     if dy > maxYChar:
117                         maxYChar = dy
118                     if dy < minYChar:
119                         minYChar = dy
120     return maxXChar, maxYChar, minYChar
121
122 def terminalCommunication(argv):
123     word = ''
124     try:
125         opts, args = getopt.getopt(argv, "hw:r:s:", ["help", "
126             word=", "radius=", "scale="])

```

```

119 except getopt.GetoptError:
120     sys.exit(2)
121 for opt, arg in opts:
122     if opt in ("-h", "help"):
123         print('dobot.py -w <word> -r <radius>')
124         sys.exit()
125     elif opt in ("-w", "--word"):
126         word = arg
127     elif opt in ("-r", "--radius"):
128         global r
129         r = float(arg)
130     elif opt in ("-s", "--scale"):
131         global scale
132         scale = float(arg)
133 return word
134
135
136 def writeWord(api, word):
137     x, y, z, rHead = dType.GetPose(api)
138     degreesTurned, lastYCoordinate, maxXCharFirst,
139     maxYCharBefore = 0, 0, 0, 0
140     firstChar = True
141     maxMovementWidth = math.sqrt(math.pow(r, 2) - math.pow(r -
142     dropOfPencil, 2))
143     path = os.getcwd()
144
145     for character in word:
146         svgfile = path + '/' + character + '_ascii.svg'
147         if not os.path.isfile(svgfile):
148             convertToSVG(path, character, svgfile)
149         maxXChar, maxYChar, minYChar = getExtremes(svgfile)
150         if firstChar:
151             maxXCharFirst = maxXChar
152             firstChar = False
153
154         moveDistanceBetweenChar = maxYCharBefore -
155         lastYCoordinate + spaceBetweenChars - minYChar
156         moveRadBetweenChar = math.asin(moveDistanceBetweenChar /
157         r)
158         moveDegreeBetweenChar = math.degrees(moveRadBetweenChar)
159
160         correctionX = maxXCharFirst - maxXChar
161         lastYCoordinate, degreesTurned = writeChar(api, svgfile,
162         maxMovementWidth, degreesTurned, lastYCoordinate, x,
163         y, z, rHead, correctionX)
164         degreesTurned += moveDegreeBetweenChar
165
166         maxYCharBefore = maxYChar
167         dType.dSleep(500)
168 if word != "":
169     lastindex = dType.SetPTPCmd(api, dType.PTPMode.
170     PTPJUMPXYZMode, x, y, z + 10, rHead, 1)
171     DobotControl.startexec(api, lastindex[0])

```

```

166
167 def main(argv):
168     word = terminalCommunication(argv)
169
170     state, api = DobotControl.connectdobot()
171     if state == dType.DobotConnect.DobotConnect_NoError:
172         DobotControl.initializing(api)
173         command = 'nothing'
174
175         while command != 'stop':
176             command = input('Put the pencil at the right
177                             position after the bleep and type "start" to
178                             write. Type "stop" to disconnect. \n')
179             if command == 'start':
180                 writeWord(api, word)
181             elif command == 'suc':
182                 dType.SetEndEffectorSuctionCup(api, 1, 1, 1)
183                 DobotControl.startexec(api, 1)
184             elif command == 'stopsuc':
185                 dType.SetEndEffectorSuctionCup(api, 0, 1, 1)
186                 DobotControl.startexec(api, 1)
187
188             DobotControl.stopexec(api)
189             DobotControl.disconnectdobot(api)
190         else:
191             DobotControl.disconnectdobot(api)
192
193 if __name__ == "__main__":
194     main(sys.argv[1:])

```

A.2 DobotControl.py

```

1 import DobotDllType as dType
2
3 CONSTSTR = {
4     dType.DobotConnect.DobotConnect_NoError: "
5         DobotConnect_NoError",
6     dType.DobotConnect.DobotConnect_NotFound: "
7         DobotConnect_NotFound",
8     dType.DobotConnect.DobotConnect_Occupied: "
9         DobotConnect_Occupied"}
10
11 def connectdobot():
12     #Load Dll
13     api = dType.load()
14
15     #Connect Dobot
16     state = dType.ConnectDobot(api, "", 115200)[0]
17     print("Connect status:", CONSTSTR[state])
18     return state, api

```

```

18
19 def initializing(api):
20     #Clean Command Queued
21     dType.SetQueuedCmdClear(api)
22
23     #Async Motion Params Setting
24     dType.SetHOMEParams(api, 250, 0, 100, -90, isQueued=1)
25     dType.SetPTPJointParams(api, 200, 200, 200, 200, 200, 200,
26                             200, 200, isQueued=1)
27     dType.SetPTPCommonParams(api, 100, 100, isQueued=1)
28     dType.SetPTPCoordinateParams(api, 200, 200, 200, 200)
29     dType.SetPTPJumpParams(api, 10, 200)
30
31     settings = dType.GetPTPCoordinateParams(api)
32     dType.SetPTPCoordinateParams(api, settings[0], settings[2],
33                                 settings[0], settings[2])
34
35     #Async Home
36     dType.SetHOMECmd(api, temp=0, isQueued=1)
37
38 def startexec(api, lastIndex):
39     #Start to Execute Command Queued
40     dType.SetQueuedCmdStartExec(api)
41
42     #Wait for Executing Last Command
43     while lastIndex > dType.GetQueuedCmdCurrentIndex(api)[0]:
44         dType.dSleep(100)
45
46
47 def stopexec(api):
48     #Stop to Execute Command Queued
49     dType.SetQueuedCmdStopExec(api)
50
51
52 def disconnectdobot(api):
53     #Disconnect Dobot
54     print("disconnected")
55     dType.DisconnectDobot(api)

```