MAS248 - Robotteknologi

Project Report

Emanuel Matt, Raphael Rieder, Robin Jesse Schwarz

supervised by

Ilya Tyapin

November 23, 2023

Mandatory Group Declaration

Each student is solely responsible for familiarizing themselves with the legal aids, guidelines for their use, and rules regarding source usage. The declaration aims to raise awareness among students of their responsibilities and the consequences of cheating. Lack of declaration does not exempt students from their responsibilities.

<table>
<tr><td>1.</td><td>We hereby declare that our submission is our own work and that we have not used other sources or received any help other than what is mentioned in the submission.</td><td>Yes</td></tr>
<tr><td>2.</td><td>We further declare that this submission:
Has not been used for any other examination at another department/university/-college domestically or abroad.
Does not reference others' work without it being indicated.
Does not reference our own previous work without it being indicated.
Has all references included in the bibliography.
Is not a copy, duplicate, or transcription of others' work or submission.</td><td>Yes</td></tr>
<tr><td>3.</td><td>We are aware that violations of the above are considered to be cheating and can result in cancellation of the examination and exclusion from universities and colleges in Norway, according to the Universities and Colleges Act, sections 4-7 and 4-8 and the Examination Regulation, sections 31.</td><td>Yes</td></tr>
<tr><td>4.</td><td>We are aware that all submitted assignments may be subjected to plagiarism checks.</td><td>Yes</td></tr>
<tr><td>5.</td><td>We are aware that the University of Agder will handle all cases where there is suspicion of cheating according to the university's guidelines for handling cheating cases.</td><td>Yes</td></tr>
<tr><td>6.</td><td>We have familiarized ourselves with the rules and guidelines for using sources and references on the library's website.</td><td>Yes</td></tr>
<tr><td>7.</td><td>We have in the majority agreed that the effort within the group is notably different and therefore wish to be evaluated individually. Ordinarily, all participants in the project are evaluated collectively.</td><td>No</td></tr>
</table>

Publishing Agreement

Authorization for Electronic Publication of Work The author(s) hold the copyright to the work. This means, among other things, the exclusive right to make the work available to the public (Copyright Act. §2).

Theses that are exempt from public access or confidential will not be published.

We hereby grant the University of Agder a royalty-free right to make the work available for electronic publication:	Yes
Is the work confidential?	No
Is the work exempt from public access?	No

Abstract

This report documents the process of programming a UR5 robot to copy and transfer a reference image to a plastic sheet using a pen attached as end effector to the robot. The robot was programmed using Ubuntu 20.04.5 with ROS Noetic (Robot Operating System) installed. To visualise the robots movement, the ROS applications Gazebo and RViz were used. The reference image was processed and converted to a path using a python script which then was executed on the robot. The image reproduced by the robot fulfilled the expectations and therefor validates the function of the program. The base program is solid. Further improvements to speed and accuracy are possible.

Keywords: ROS Noetic, UR5, Path planning, Image reproduction, Image processing

Contents

1 Introduction **1**
1.1 Project Tasks 1

2 Methods **3**
2.1 Description for MoveIt 3
2.2 Robot Description 4
2.3 Image Processing 5
2.4 Define Workspace 6
2.5 Plan Robot movement based on Path 7

3 Results **8**

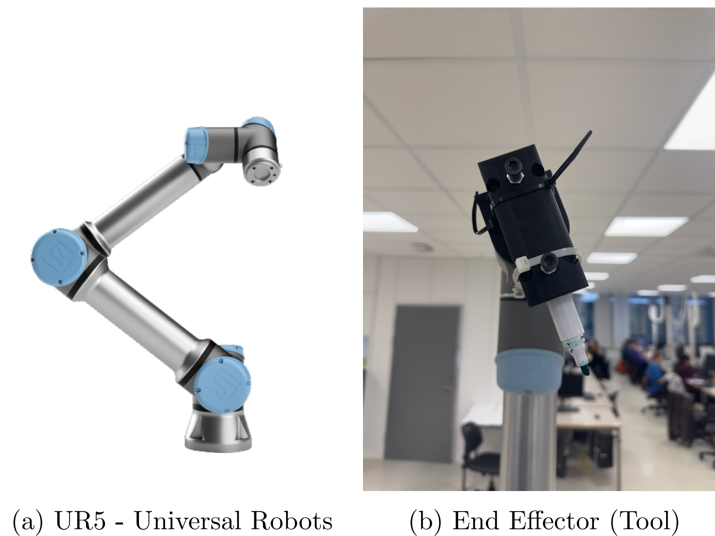
4 Discussion **9**
4.1 Virtual Machine 9
4.2 Windows WSL Subsystem 9
4.3 Connection to Robot 10

5 Conclusion **11**

References **12**

1 Introduction

The goal of this project was to draw an image on a plastic sheet using a UR5 robot from Universal Robots with a pen as its end effector (see figure 1)[1]. The robot's path should be planned and executed based on the reference image chosen by the group (see figure 2). Several sets of slides and predefined files were available to help with this, which were modified and implemented in order to be able to paint the picture.



(a) UR5 - Universal Robots (b) End Effector (Tool)

Figure 1: Robot Setup

1.1 Project Tasks

- Preparing Linux system

  The first step is to set up the virtual machine with Ubuntu 20.04. In addition, some packages, such as the guest-addon for the virtual machine, are helpful to be installed at the beginning.

- Installing ROS

  The next step is to install ROS on the Linux virtual machine. The Robot Operating System is the base program which all the simulations run on.

- Installing MoveIt!

  MoveIt! has to be installed for the motion planning and kinematic simulations.

- MAS248_Robotics project setup

  The next step us to install the drivers for the target robot and set up the Catkin workspace. The Universal_Robots_ROS_Driver contains all the necessary files to control the UR5 robot.

- Changing robot description

  An important step is to adapt the description to the needs of the project. The tooltip and base has to be added, and the workspace has to be defined to ensure security.

- Plan a path/trajectories

  The aim of the project is to draw a picture with a pen held by the robot. Therefore trajectories have to be planned, which is done by a Python script that has to be costumised.

- Visualise/run robot in rviz

  To ensure that the robot moves as it should, the trajectories need to be simulated. This ensures correct movement and also shows whether the robot description is correct.

- Run on the robot

  To check the trajectory, boundary box and home position, the script must be run on the robot itself. To check the movement afterwards, a video is recorded.

- Develop scripts

  Additional scripts for motion or other special functions can be developed if required.

- Report writing

  The final step of the project is to write a report so that the steps taken can be followed up at any time.

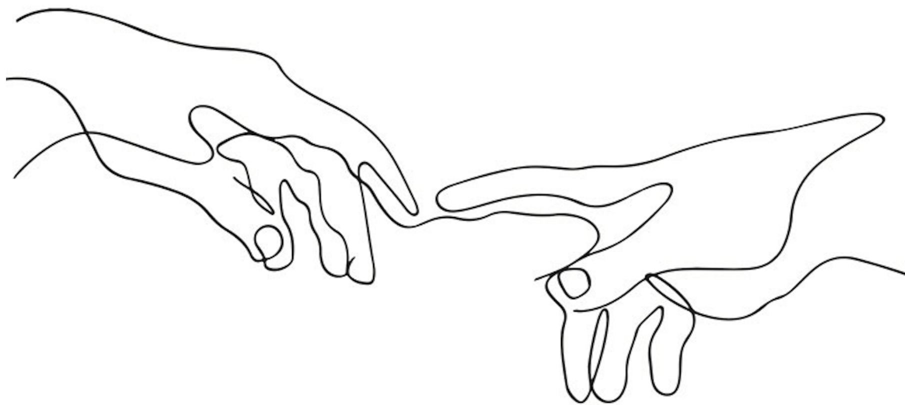


Figure 2: Reference Image

2 Methods

The following subsections discuss the changes made to the original scripts downloaded from Canvas [2]. Most changes were made to the python script responsible for the movement of the robot (move_group_interface_final.py).

2.1 Description for MoveIt

In this file, the following part were modified in order to setup the robot properly.

```
12 <group name="manipulator">
13     <chair base_link="world" tip_link="tooltip" />
14 </group>
15 <group name="endeffector">
16     <link name="tooltip" />
17 </group>
```

```
36 <end_effector name="moveit_ee" parent_link="tooltip" group="endeffector" />
```

```
51 <disable_colissions link1="tablemount" link2="base_link_inertia" reason="
       Adjecent" />
52 <disable_colissions link1="wrist_3_link" link2="tool" reason="Adjecent" />
```

Listing 1: ur5.srdf

2.2 Robot Description

This code was added to ur5_macro.xacro to attach base and end effector to the robot.

```
<link name="world"/>
<link name="tooltip"/>

<link name="toolstart">
<origin xyz="0 0 0" rpy="0 0 0"/>
    <visual>
        <geometry>
            <mesh filename="package://ur_description/meshes/ur5/collision/finaltool.STL" scale="0.001 0.001 0.001"/>
        </geometry>
    </visual>
    <colission>
        <geometry>
            <mesh filename="package://ur_description/meshes/ur5/visual/finaltool.dae" scale="0.001 0.001 0.001"/>
        </geometry>
    </colission>
</link>

<link name="tablemount">
<origin xyz="0 0 0" rpy="0 0 0"/>
    <visual>
        <geometry>
            <mesh filename="package://ur_description/meshes/ur5/collision/tablemount.STL" scale="0.001 0.001 0.001"/>
        </geometry>
    </visual>
    <colission>
        <geometry>
            <mesh filename="package://ur_description/meshes/ur5/visual/tablemount.dae" scale="0.001 0.001 0.001"/>
        </geometry>
    </colission>
</link>

<joint name="world_joint" type="fixed">
    <parent link="world"/>
    <child link="tablemount"/>
    <origin xyz="0 0 0.303" rpy="0 0 0"/>
</joint>

<joint name="tablemount_to_base_link_joint" type="fixed">
    <parent link="tablemount"/>
    <child link="${prefix}base_link"/>
    <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
</joint>

<joint name="wrist3_to_toolstart_joint" type="fixed">
    <parent link="${prefix}wrist_3_link"/>
    <child link="toolstart"/>
    <origin xyz="0 0 0" rpy="0 0 3.5"/>
</joint>

<joint name="toolstart_to_tooltip_joint" type="fixed">
    <parent link="toolstart"/>
    <child link="tooltip"/>
    <origin xyz="0.000 -0.118 0.107" rpy="-1.570796327 0 0"/>
</joint>
```

Listing 2: ur5_macro.xacro

2.3 Image Processing

This function was used to read the reference image and from that, calculate the path for the robot. The function uses algorithms from the OpenCV-Library. First it blurs the images and applies a black and white threshold. Then it applies a canny-edge-detection algorithm (`cv2.canny()`)[3] to find edges in the image from which the `cv2.findContours()`[4] function can find contours in the image. The contours then get converted in a path for execution on the robot (see figure 3b).

```
109 def process_image():
110     img = cv2.imread('src/python_scripts/ReferenceImage.jpg')
111 
112     img = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
113 
114     blur = cv2.GaussianBlur(img,(5,5),cv2.BORDER_DEFAULT)
115 
116     imgray = cv2.cvtColor(blur,cv2.COLOR_BGR2GRAY)
117 
118     thresh_image = cv2.adaptiveThreshold(imgray,255,cv2.
    ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,11,5)
119 
120 
121     # Find Canny edges
122     edged = cv2.Canny(thresh_image, 30, 200)
123 
124     contours, hierarchy = cv2.findContours(edged,cv2.RETR_TREE,cv2.
    CHAIN_APPROX_SIMPLE)
125 
126     # drawing contours over blank image
127     contours_image = np.zeros(img.shape, dtype=np.uint8);
128     cv2.drawContours(contours_image, contours, -1, (0,255,0), 3)
129 
130     print("Number of Contours found = " + str(len(contours)))
131 
132     x_coord = []
133     y_coord = []
134     z_coord = []
135     for cont in contours:
136         pixel_pose = cont.reshape(-1,2)
137         x_coord.extend(pixel_pose[:,1])
138         y_coord.extend(pixel_pose[:,0])
139 
140         z_coord_zero = [0.0 for i in pixel_pose]
141         z_coord.extend(z_coord_zero)
142 
143 
144 
145         x_coord.extend([pixel_pose[-1,1]])
146         y_coord.extend([pixel_pose[-1,0]])
147         z_coord.extend([0.1])
148 
149     #scale the image to reasonable size
150     print("Image size = ",img.shape)
151     x_coord = (-min(x_coord) + x_coord)/(max(x_coord)-min(x_coord))*0.210 #+
    0.20;
152     y_coord = (-min(y_coord) + y_coord)/(max(y_coord)-min(y_coord))*0.290 #+
    0.20;
153 
154     return x_coord, y_coord, z_coord
```

Listing 3: move_group_interface_final.py: def process_image()

2.4 Define Workspace

This class was added to define the workspace and add collision objects to RViz in order to confine the movement of the robot to the table its standing on.

```
163 class CollisionSceneExample(object):
164     def __init__(self):
165         self._scene = PlanningSceneInterface()
166 
167         # clear the scene
168         self._scene.remove_world_object()
169 
170         self.robot = RobotCommander()
171 
172         rospy.sleep(2)
173 
174     def add_floor(self):
175         box1_pose = [(0.9-0.165), -(0.4-0.165), -0.026, 0, 0, 0, 1]
176         box1_dimensions = [1.8, 0.8, 0.05]
177 
178         self.add_box_object("floor", box1_dimensions, box1_pose)
179 
180         self.add_box_object("floor", box1_dimensions, box1_pose)
181 
182     def add_four_walls(self):
183         box1_pose = [-0.165, -(0.4-0.165), 1, 0, 0, 0, 1]
184         box1_dimensions = [0.002, 0.8, 2]
185 
186         box2_pose = [(0.9-0.165), 0.165, 1, 0, 0, 0, 1]
187         box2_dimensions = [1.8, 0.002, 2]
188 
189         box3_pose = [1.8-0.165, -(0.4-0.165), 1, 0, 0, 0, 1]
190         box3_dimensions = [0.002, 0.8, 2]
191 
192         box4_pose = [(0.9-0.165),-(0.8-0.165), 1, 0, 0, 0, 1]
193         box4_dimensions = [1.8, 0.002, 2]
194 
195         self.add_box_object("wall1", box1_dimensions, box1_pose)
196         self.add_box_object("wall2", box2_dimensions, box2_pose)
197         self.add_box_object("wall3", box3_dimensions, box3_pose)
198         self.add_box_object("wall4", box4_dimensions, box4_pose)
199 
200     def add_all(self):
201       self.add_floor()
202       self.add_four_walls()
203 
204     def add_box_object(self, name, dimensions, pose):
205         p = geometry_msgs.msg.PoseStamped()
206         p.header.frame_id = self.robot.get_planning_frame()
207         p.header.stamp = rospy.Time.now()
208         p.pose.position.x = pose[0]
209         p.pose.position.y = pose[1]
210         p.pose.position.z = pose[2]
211         p.pose.orientation.x = pose[3]
212         p.pose.orientation.y = pose[4]
213         p.pose.orientation.z = pose[5]
214         p.pose.orientation.w = pose[6]
215 
216         self._scene.add_box(name, p, (dimensions[0], dimensions[1], dimensions
    [2]))
```

Listing 4: move_group_interface_final.py: class CollisionSceneExample()

2.5 Plan Robot movement based on Path

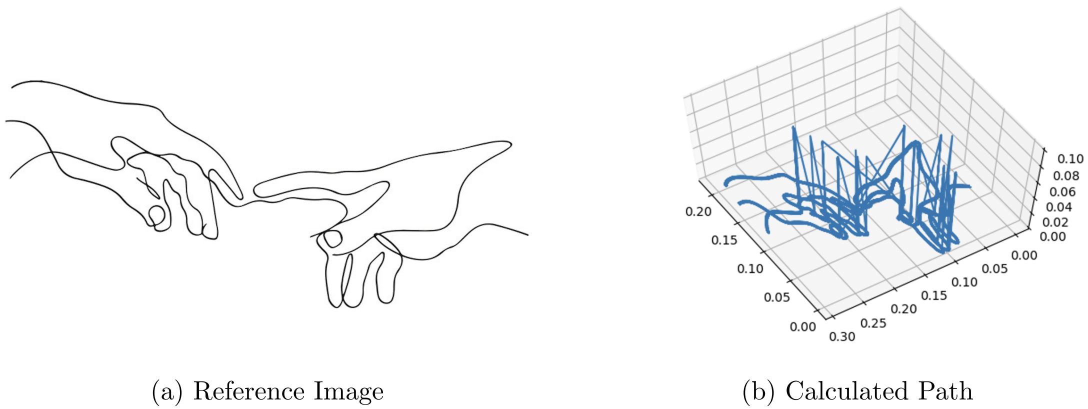
This function converts the path planned in section 2.3 into a trajectory for the robot. In row 181-182, the offset +0.3m in x and −0.55m in y define the displacement of the image's origin to the world origin.

```
163 def plan_cartesian_path(self, scale=1):
164     # Copy class variables to local variables to make the web tutorials more
        clear.
165     # In practice, you should use the class variables directly unless you have a
         good
166     # reason not to.
167     move_group = self.move_group
168     x_coord, y_coord, z_coord = process_image()
169 
170     waypoints = []
171 
172     wpose = move_group.get_current_pose().pose
173 
174     wpose.orientation.x = 0.0
175     wpose.orientation.y = 0.0
176     wpose.orientation.z = 0.0
177     wpose.orientation.w = 1.0
178 
179     for i in range(len(x_coord)):
180 
181         wpose.position.x = x_coord[i] + 0.3
182         wpose.position.y = y_coord[i] - 0.55
183         wpose.position.z = z_coord[i]
184         waypoints.append(copy.deepcopy(wpose))
185 
186 
187     poses = geometry_msgs.msg.PoseArray()
188     poses.header.frame_id = self.move_group.get_planning_frame()
189     poses.poses = [point for point in waypoints]
190     self.poseArray_publisher.publish(poses)
191 
192 
193     # We want the Cartesian path to be interpolated at a resolution of 1 cm
194     # which is why we will specify 0.01 as the eef_step in Cartesian
195     # translation.  We will disable the jump threshold by setting it to 0.0,
196     # ignoring the check for infeasible jumps in joint space, which is
        sufficient
197     (plan, fraction) = move_group.compute_cartesian_path(
198         waypoints, 0.01, 0.0  # waypoints to follow  # eef_step
199     )  # jump_threshold
200 
201     return plan, fraction
```

Listing 5: move_group_interface_final.py: def plan_cartesian_path()

3 Results

As can be seen in figure 3, the reference image was converted into the path for the robot. Because the lines of the reference image were to thick, the function described in section 2.3 interpreted the individual lines as full contours leading to the robot drawing each line at least two times. Additionally, the format of the reference image was to wide, which lead to the image being slightly squeezed together in order to fit on the plastic sheet with format A4 (see figure 4). Due to a small offset in orientation of the plastic sheet, the right part of the image was cut of by 4cm in Y-direction. Apart from this small offset, the program executed flawlessly on the physical robot.



(a) Reference Image

(b) Calculated Path

Figure 3: From Reference to calculated Path

Figure 4: Painted Image

4 Discussion

During the setup and programming phase we encountered a number of problems. Most of these were related to the virtual machine or subsystem used for Linux. There were also a few programming issues that needed to be resolved.

4.1 Virtual Machine

The virtual machine performed very poorly in simulating the robot on a machine running Windows 11. With the default settings in the virtual machine setup, the Gazebo simulation ran at less than one frame per second. This resulted in a programming stop until the problem was resolved.

The first solution was to change the virtual machine settings. With all eight cores and twelve gigabytes of memory dedicated to the virtual machine, the simulation ran a little more smoothly. After these changes, the Gazebo simulation ran at around 4-6 frames per second. This was still too slow to effectively inspect the robot's movements as it carried out its task.

4.2 Windows WSL Subsystem

As it was not possible to get reasonable performance in Gazebo in a virtual machine, we switched to the Windows subsystem WSL for Linux.

Installation was fairly quick, and performance was much better than on the virtual machine. Installing the entire baseframe for the robot simulation took a quarter of the time it did on the virtual machine. The Gazebo simulation also had a much higher frame rate than the virtual machine simulation.

Once the performance problem was solved, the next major problem arose, because we were using WSL2, an improved version of the Windows subsystem WSL, it was not up to date with modern hardware. The computer we were using to run the simulations had a built-in RTX 3050Ti running DirectX12. WSL2, as it is now, does not work with this version of DirectX. This resulted in no or degraded graphics in the Gazebo simulation. Two solutions were given:

Running the simulation on the CPU made the simulation much more unstable and even after upgrading the system the corrupted graphics returned.

The second solution was to use a different backend for the graphics rendering of the WSL2 subsystem. Mesa3D is a software package that allows DirectX12 to run on the subsystem. After a long and complicated installation, the simulation worked in RViz and Gazebo. RViz also showed the Robot with base and Tooltip. Nontherless, the Gazebo simulation was not able to render the imported 3D files.

4.3 Connection to Robot

As we could not use the Windows subsystem WSL on our powerful computer, we switched to an older, weaker model. With this older laptop we were able to run the simulation without any graphics problems. However, this older laptop also had Windows 11 installed, which led to another problem, the connection between WSL and the robot. A plug-in for the WSL had to be installed in order to establish a connection. Once installed, however, there were problems with the IP address and we were unable to establish a connection in the short time available.

The solution is simple. We use a laptop running Windows 10, which is possible because we already know how to connect to the robot. This allowed us to make the necessary IP adjustments and start the robot, which led to the final result.

5 Conclusion

Installing the virtual machine and the Windows subsystem WSL took the most time, accounting for 60% of the total workload. There were many problems that took a lot of time to resolve, which led to the virtual machine being reinstalled several times.

In the end, to program and simulate the project we used a Macbook with macOS Sonoma V14.0 with a Parallels-VM running Ubuntu 20.04.5. This worked well and we were able to run the simulation with up to 30 frames per second. Nonetheless, we were not able to connect the Macbook to the physical robot and switch to an older laptop with Windows 10 installed to run the final program on the physical robot.

The painted image differs from the simulation as the zero point of the image was not measured precisely during the test setup. The robot therefore paints outside the drawing area. Otherwise the process works flawlessly.

The pen holder can still be optimised so that there is less play between pen and holder. A more flat base (in our case the table) can also minimise the pressure differences during the drawing process, which would further improve the quality of the drawing.

Apart from these minor issues, the robot performed as intended and the goal of the project was fulfilled.

List of Figures

1 Robot Setup 1
2 Reference Image 2
3 From Reference to calculated Path 8
4 Painted Image 8

Listings

1 ur5.srdf 3
2 ur5_macro.xacro 4
3 move_group_interface_final.py: def process_image() 5
4 move_group_interface_final.py: class CollisionSceneExample() 6
5 move_group_interface_final.py: def plan_cartesian_path() 7

References

[1] "UR5 technical specifications." (2016), [Online]. Available: https://www.universal-robots.com/media/50588/ur5_en.pdf.

[2] "Canvas - lecture 08: Projectfiles." (2023), [Online]. Available: https://uia.instructure.com/courses/13950/pages/lecture-08-project?module_item_id=532564 (visited on 11/21/2023).

[3] "OpenCV: Canny edge detection." (), [Online]. Available: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html (visited on 11/21/2023).

[4] "OpenCV: Contours." (), [Online]. Available: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html (visited on 11/21/2023).