Prototyping Project - Bin Picking

1 Overview - Introduction

The goal of this project is to visualize a bin picking task (pick and place) with a 6-DoF robotic arm using ROS2 Humble. The efforts are based upon a real-life prototype/application developed as a collaboration between Liebherr and Fanuc which was presented at the BAUMA 2022. Using a camera system, the robot is able to identify and locate specific objects inside a bin, pick the item and place it inside a designated target location. This technology has the potential to increase efficiency and productivity in the construction industry by automating tasks which are currently performed manually, e.g.: placing bricks from a conveyor belt onto a Euro pallet to help with packaging in the factory.
The project was adjusted to the requirements of the Prototyping Project and involved the development and implementation of the necessary software components, including the model of the robotic arm as well as object detection and robot movement algorithms.

Keywords: bin picking, pick and place task, robot, ROS2 Humble, OpenCV, movement planning

GitLab project link (private): https://git.rwth-aachen.de/prototypingproject20222023/groupg

Project Parts:
- Object Identification
- Environment - Robot model and staffage objects
- Movement planning - kinematics and path planning
- Data transfer and Communication

2 Object Identification

For the sake of the project a “training environment” was developed consisting of a variety of different geometrical shapes with different colours in different positions. A total of 4 different sets was created (folder: table_images) to “feed” the object detection algorithm and provide a basis to choose objects for the pick and place task from. For the project the OpenCV library was used. Several sub-tasks are performed by the program/algorithm to give the robotic arm the information which object to select and which position to pick (object centre).
Performed sub-tasks:
1. Compute the contour centre
2. Recognition of various shapes (basic geometries)
3. Colour labelling of the shape
4. Object identification output format

1.) Computation of the contour centre - object_detection.py

First the object centre is determined. With OpenCV some image pre-processing is required in order to perform the task:
- Grayscale Conversion
- Blurring: increase the contour detection accuracy by reducing image noise (here: Gaussian Blur)
- Binarization: by thresholding

Now, the image consists of a black background with white objects in contrast.
The next step is the contour detection to detect the outline of an object to determine its centre by using “cv2.findcontours()”. “cv2.moments()” is able to find the centre coordinates of the objects.

2.) Recognition of basic geometries - shapedetector.py

The technique in use for the geometry detection in our project is contour approximation which is already implemented in OpenCV with the “cv2.approxPolyDP()” method. The number of vertices of the determined contour is used to output the geometric shape. Three vertices for a triangle, four vertices for a square, five vertices for a pentagon. Is a certain number of vertices reached/exceeded, the detected object is defined as a circle.

3.) Colour labelling of the shape - colorlabeler.py

The determined colours are defined in a colour dictionary. With a ColorLabeler class the found colours are checked against the defined values in the dictionary and are given the colour closest to the defined colour values.

4.) Object identification output format

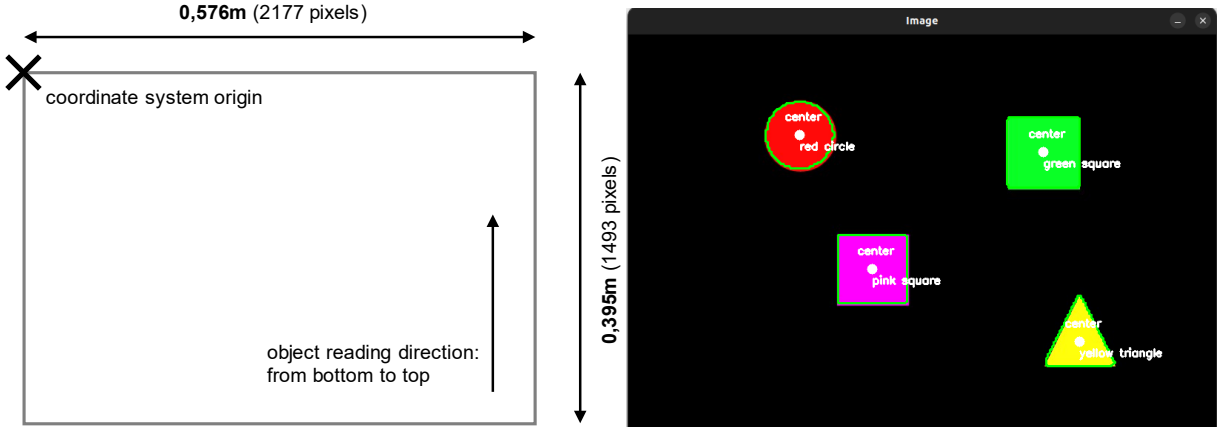


Figure 1: Object identification input picture specifications → Object identification output

After the succesful object identification process the individual object information are transferred to the “object_marker_publisher” node as well as the “inverse_kinematics” node to continue the pick and place task procedure. The detection and coordinate output sequence for the individual objects is predefined by OpenCV. The coordinate system origin is set in the top left corner of the given picture. The object detection direction order is from the bottom to the top of the picture.
The object properties (object shape, object color and x/y coordinates) of the given picture are output as a string message:

```
data: triangle yellow 379 336 square pink 617 291 square green 177 229 circle red 485
187
```

Figure 2: Object identification output format

3 Environment - Robot model and staffage objects

1.) Robot model - urdf_model

The IRB 1200 is a robot from the latest generation of ABB Robotics' 6-axis industrial robots, with a payload of 5 to 7 kg. It has been specifically designed for the manufacturing industry where flexible, robot-based automation is in use. This robot is used for the bin picking task in the project. The geometry of the robotic arm was implemented as Collada format (.dae). Some of the Unified Robot Description Format (URDF) was adapted/modified according to the specific needs of the project proposal. Special attention should be given to the possible work area and the movement type of the robot. In the following table (excerpt from the robot data sheet) the robots joint value limits are specified:

Position of movement	Type of movement	ABB IRB 1200-5/0.9
axis 1	rotational movement	+170° to -170°
axis 2	arm movement	+130° to -100°
axis 3	arm movement	+70° to -200°
axis 4	wrist movement	+270° to -270°
axis 5	tilt movement	±130° (not Hygienic robots) ±128° (Hygienic robots)
axis 6	rotational movement	Default: +400° to -400° Maximum revolution: ±242 (extended by changing the parameter values in the software)

Figure 3: Joint value limits and types of motion

Note: For all used joints the joint type in the URDF-file is defined as revolute. The minimal and maximal values for the rotation range must be converted into radians.

2.) Staffage objects

Apart from the robot model two additional categories of objects are placed to complete the environment.

Basic geometries – object_marker node

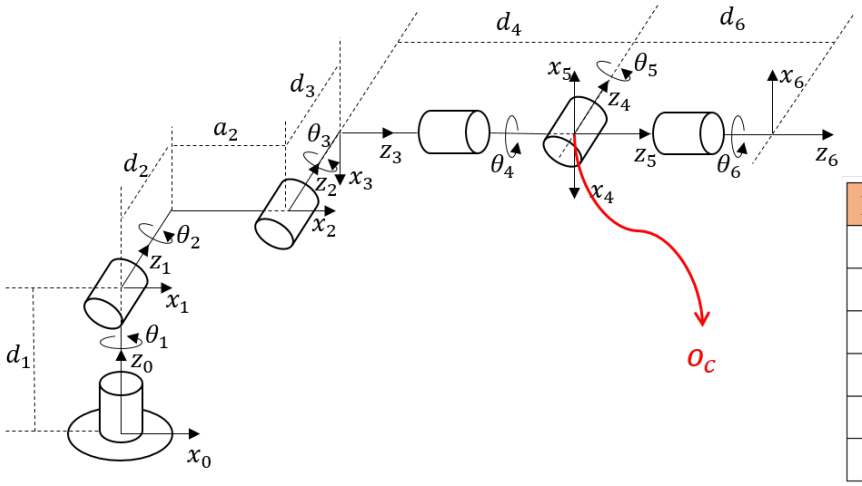
The basic geometries (triangle, cylinder, spuare, spuare) are placed into the environment according to the object's detected centre during the object identification process. Depending on the loaded image set the object's colour and position is determined. The objects are published as a marker array, a mesh resource is used for the triangle.

Environment objects – environment_marker node

In addition, a table, a box for the starting position of the objects and a "target area" is placed into the scene for the robot to pick the objects from and place the objects at. The mesh resources (Collada files) are located in the "environment" folder of the package.

4 Movement planning - kinematics and path planning inverse_kinematics.py

1.) Inverse kinematics



Link	d_i	θ_i	a_i	α_i
1	d_1	θ_1^*	0	$-\pi/2$
2	d_2	θ_2^*	a_2	0
3	d_3	θ_3^*	0	$-\pi/2$
4	d_4	θ_4^*	0	$-\pi/2$
5	0	θ_5^*	0	$-\pi/2$
6	d_6	θ_6^*	0	0

Figure 4: Overview 6-DoF robotic arm structure for geometric method

Inverse kinematics is the opposite to forward kinematics and presents the use of kinematic equations to determine the joint values if the desired end position of the end effector (gripper) is known. Amongst others, especially two possibilities arise, to solve the inverse kinematics problem for a 6-DoF robotic arm. On the one hand there is the possibility to obtain numerical solutions through iterative methods. However, the computation speed is rather slow. On the other hand, one can also obtain analytical solutions through geometric methods. In our case the geometric method was applied to solve the inverse kinematics. A prerequisite for this strategy to be applicable is that the last three joints are revolving joints (→ spherical wrist) as well as their axis of rotation intersecting in one point (→ wrist centre). With these two prerequisites given, the inverse kinematics problem is divided into the two subproblems of determining the individual joint values for the position and orientation of the end effector. The robotic arm structure is displayed in Figure 4. z3, z4, z5 are the rotation axes of the three revolute joints that form the spherical wrist and intersect in a point o_c. When the three revolute joints rotate, the position of o_c remains unchanged (→ the position is determined only by the first three joints q1, q2, q3). With this information as starting point the most important code components of the inverse kinematics are explained in the following table:

Component	Explanation
`class State():`	The class holds the joint angles, positions and orientations of the different links of the robotic arm; along with the rotation matrices between the links. The class also holds the end-effector position and orientation.
`def euler2R (roll, pitch, yaw):`	The method takes roll, pitch, and yaw angles as inputs and returns the corresponding rotation matrix.
`def initial():`	The method computes the initial rotation matrices between the links of the arm using the euler2R method.
`def inverse_kinematics():`	The method calculates the joint angles required to place the end-effector at a given position and orientation. The function first calculates the rotation matrix of the end-effector with respect to the world frame and then performs inverse kinematics calculations to find the joint angles.

Figure 5: Inverse kinematics code components

2.) Path planning

The path planning for the robotic arm is also executed within the “inverse_kinematics” node. It outputs the path/movement between start, object (centre) and target position. Every movement between two points is divided into 50 sub-movements to create a smooth robot motion.

Component	Explanation
`def move_joints (self,qgoal,qinitial):`	The method executes a movement from initial to target joint values, slicing the movement into 50 steps.
`def movement_sequence(self):`	The method iterates over the objects in the scene, uses safe points to move towards them, publishes the grasp status of the objects and places them in their target location.

Figure 6: Path planning code components

5 Data transfer and Communication

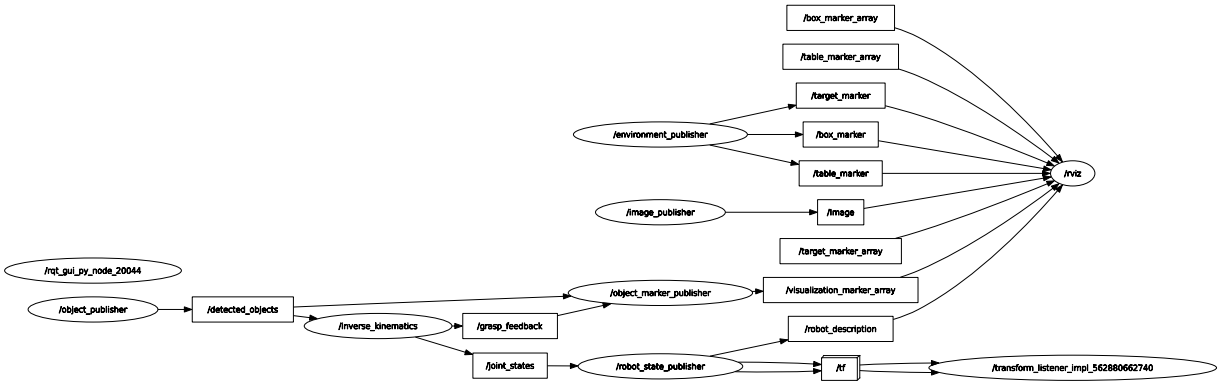


Figure 7: rqt_graph - overview data transfer and communication

The following graph shows the project’s data transfer and communication accomplished with publishers and subscribers. The object identification from the selected image set is done inside the “object_publisher” node and serves as the main input for the other nodes. It publishes the object shapes, colours and their x and y coordinates to the /detected_objects topic. This topic is subscribed to by the “object_marker_publisher” and “inverse_kinematics” node. The “object_marker_publisher” creates Marker() objects from this information and publishes the markers as a MarkerArray() message to the /visualization_marker_array topic, allowing them to be displayed in RVIZ. The “inverse_kinematics” node uses the detected objects to compute the required joint values to move the robot to the object coordinates, publishes the joint values to the /joint_states topic and also publishes a /grasp_feedback to the “object_marker_publisher” node. This feedback is used to attach the object markers to the end effector when the robot reaches the objects position and releases them again when the target position of the object is reached.
The “robot_state_publisher” node subscribes to the /joint_states topic and publishes coordinate frames to the transform library of ROS2. Furthermore, an “image_publisher” node is used to publish an image message that displays the selected image set in RVIZ. The “environment_publisher” node publishes three separate markers for the table, the starting box of the objects and the target area, all of which are displayed in RVIZ. The corresponding Collada files are located in the “environment” folder of the package.

6 Conclusion - Challenges and future work

Object Detection

For the application in a real-life scenario two conditions for the Object Detection became apparent which could be improved and therefore leave room for future work. On the one hand it is the detection of complex shapes/geometries and on the other hand it is the colour labelling.

Complex Shapes

Currently the geometries are determined by the number of their vertices. A star for example has ten, so does a decagon, depending on the defined threshold value it could also be identified as a circle. Here a more sophisticated contour detection/shape matching method would be appropriate, e.g.: Hu moments.

Colour labelling

In the beginning, the colours are determined as an RGB tuple inside a dictionary. Only the pre-defined colours can be detected. The background also has an influence on the detection reliability. Black or white backgrounds provide a contrast and increase the detection accuracy. For real-life implementation and future work a more versatile and more reliable colour detection approach is suggested.

Environment - Robot model and staffage objects

Robot Model

For subsequent work the “design” of the robot model could be done with a more sophisticated approach using e.g. Gazebo (simulation suite with focus on the realistic representation of physics). RVIZ is currently in use to display the robotic arm. However, this tool is limited in terms of possibilites and predominatly suitable for visualisation purposes. With Gazebo the user takes the step from visualisation to simulation, but at the same time the development requirements increase drastically. To describe the specifications of the robot the XML-based “SDF” (Simulation Description Format) format is required. The addition of properties like mass, inertia or the centre of gravity of the different links increases the possibilities for simulation, but also the effort for implementation.

Movement planning - kinematics and path planning

Inverse kinematics

Obtaining analytical solutions through geometric methods:
Not all inverse kinematic problems can be solved with the applied analytical approach. Some necessary prerequisites have to be met in order to “qualify” for this approach (see also: Chapter 4 Movement planning - kinematics and path planning). Also, more testing would be required to test the reliability of the implemented approach outside of the scope of this project (future research question: Can the correct joint values be computed for every given end effector position?). In general, a more sophisticated mathematical understanding is required. For this reason, the possibility to use MoveIt2 as a framework for the inverse kinematic calculations was also investigated.

MoveIt2:

For future work one could switch from using geometric methods to solve the inverse kinematics problem of a robotic arm to MoveIt2. MoveIt2 is a ROS2 robotic manipulation platform and allows for a more “automated” movement planning.
For the setup/installation process follow along the official MoveIt2 documentation.
documentation link: https://moveit.picknik.ai/humble/index.html
Initially, MoveIt2 was the preferred approach to solve the inverse kinematics and path planning, but due to problems, explained in the following, we decided against it and our own “solver” was developed (see also: Chapter 4 Movement planning - kinematics and path planning). Several downsides of MoveIt2 became apperant immediately. The installation process is very time-consuming and tedious which leaves room for errors on both the user as well as the software side. Before launching the software and its extensions, a lot of troubleshooting is required which limits the user-friendliness. While following along the “Pick and Place with MoveIt Task Constructor” tutorial we ran into a problem while downloading the “MoveIt Task Constructor”. We created an issue on GitHub (Download MoveIt Task Constructor Fails #418 by donnoxin) and the MTC download and installation instructions were updated shortly after.
Apart from the lengthy set-up process the high hardware requirements to actually run MoveIt2 pose another problem. At least 16GB of RAM is recommended to run the software and execute tasks like building the MoveIt2 workspace. Especially during the development process the fast and repetitive execution of the software is required to faciliate target-oriented troubleshooting. The explained challenges limit those efforts quite drastically.
MoveIt2 can be describes as rather buggy/unreliable in general. Our personal observation was that running the program ten times resulted in an error eight times.

Path planning

In the environment of the project the implemented path planning approach was sufficient and the collision-free movement of the robot could be ensured by setting several safe points along the path. However, adding additional objects inside the environment increase the level of complexity. Not only the self-collision checking is required then, but also a full collision checking in order to avoid collisions with the environment objects (e.g.: wall) as well.

References

Figure 3: Working range and type of motion
ABB (2022 September 21). Robotics Produktspezifikation IRB 1200 [Online].
Available:
https://library.e.abb.com/public/52b343c24a894176a5f3c3617a9e6be3/3HAC046982%20PS%20IRB%201200%20on%20IRC5-de.pdf?x-sign=AfgivypTv6NgZr4SpBa4TAqpklfd8LlaKcUOebYb1WUE7SAIj6DN2PKejY1Kyw6/ ,
last accessed 2023/02/10.

Figure 4: Overview 6-DoF robotic arm structure for geometric method
Gao Yichao. inverse kinematics [Online].
Available:
https://gaoyichao.com/Xiaotu/?book=math_physics_for_robotics&title=inverse_kinematics ,
last accessed 2023/02/10.

Contact information

timur.kuzu@rwth-aachen.de - Timur Kuzu
dominik.leitner@rwth-aachen.de - Dominik Leitner
orhun.oezcan@rwth-aachen.de - Orhun Oezcan

Supervisor:
picchi@ip.rwth-aachen.de - Davide Picchi

RWTH AACHEN
Chair of Individualized Production in Architecture
Construction Robotics

```
bin_picking_project
├── bin_picking_project
│   ├── image_processing
│   │   ├── __init__.py
│   │   ├── colorlabeler.py
│   │   └── shapedetector.py
│   ├── __init__.py
│   ├── environment_marker.py
│   ├── image_publisher.py
│   ├── inverse_kinematics.py
│   ├── object_detection.py
│   └── object_marker.py
├── config
│   ├── params1.yaml
│   ├── params2.yaml
│   ├── params3.yaml
│   └── params4.yaml
├── environment
│   ├── env_box.dae
│   ├── env_table.dae
│   ├── target_position.dae
│   └── triangle.dae
├── launch
│   ├── launch1.py
│   ├── launch2.py
│   ├── launch3.py
│   └── launch4.py
├── resource
│   └── bin_picking_project
├── start_sricpts
│   ├── icons
│   │   ├── V3_BUILD_ICON.svg
│   │   └── V3_START_ICON.svg
│   ├── build_package.sh
│   ├── start_set_1.sh
│   ├── start_set_2.sh
│   ├── start_set_3.sh
│   └── start_set_4.sh
├── rviz
│   └── config.rviz
├── table_images
│   ├── set_01.jpg
│   ├── set_02.jpg
│   ├── set_03.jpg
│   └── set_04.jpg
├── test
│   ├── test_copyright.py
│   ├── test_flake8.py
│   └── test_pep257.py
├── urdf_model
│   ├── meshes
│   │   ├── base.dae
│   │   ├── body1.dae
│   │   ├── body2.dae
│   │   ├── body3.dae
│   │   ├── body4.dae
│   │   ├── body5.dae
│   │   ├── body6.dae
│   │   └── __init__.py
│   └── robot_model.urdf
├── package.xml
├── setup.cfg
└── setup.py
```

Figure 8: bin_picking_project - folder structue