Aldohami Omar, Panoti Qirjako, Albaaj Mohamed

ROS Visualization and Detection of Virtual Rebar in Concrete Using Radar Scanner

Abstract

This paper proposes a robotic approach to perform on-site rebar detection in a construction building using an electromagnetic detector integrated in a robot arm end effector. The detected information is transmitted through the Ros2 platform, which serves as an interface for exchanging data between physical elements and devices. By accurately assessing the structural properties of the concrete wall, this approach offers a feasible automated solution for the construction industry. The flexibility of the Ros2 platform provides a new opportunity to implement such solutions in a construction environment.

1. Motivation

The reuse of building components from dismantled existing buildings is a great potential in the aim of reducing the Co2 emissions towards a more sustainable building construction industry.

The current stock of existing buildings of this typology is relatively large and it can be used as a resource from which the components can be reused. There is evidence of successfully completed building projects made out of existing concrete components and also attempts to use automation of the dismantling process through robotics implementation.

However, at the moment this practice is constrained only to particular cases and to small scale projects.

The major reason for this is the lack of knowledge concerning the structural properties of the existing components. The current practice for assessing these properties is based on manual on-site Non Destructive Tests (NDT) which require considerable time and resources and what is more important, they are limited to a certain number of specimens. This often causes limitations and uncertainties in a scenario when a new building has to be constructed with the existing elements coming from a dismantled building.

Creating an automated robotic process for performing Non Destructive Tests in existing concrete elements creates an opportunity to obtain more reliable test results with savings in time and resources.

This opportunity that the automation can offer together with the potential that this building typology presents with regard to sustainability, are the grounds of motivation for this paper, which elaborates this process in a robotic simulating platform.

2. Problem statement

Although the process of dismantling of building components is an already known process in the construction practice, reusing them and building a new construction out of them presents several challenges which go beyond the typical construction tasks.

The design discipline has an important role in order to properly plan, coordinate and implement this process. However, one issue associated with the design is the uncertainty related to the physical properties of the existing components when they are put in a new design context with different loading configuration.

One big challenge is to be able to accurately determine the properties of the existing components in order to reuse them safely in another building.

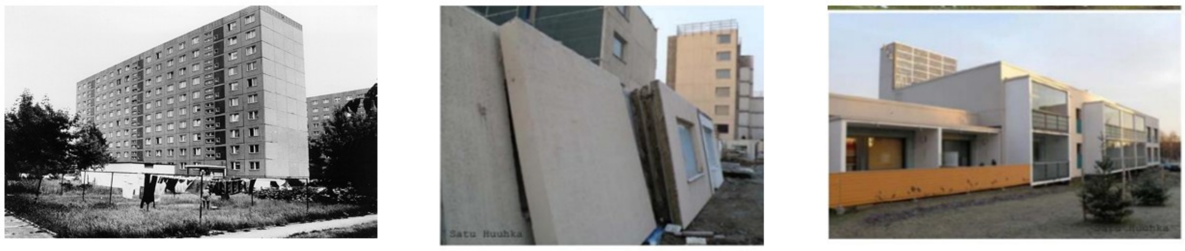

Existing building to be demolished
Dismantling of the components
New building

Figure1: Examples of project with dismantled buildings

2.1 On-site assessment and Non Destructive Tests (NDT)

The current practice for on-site assessment and Non-Destructive Tests (NDT) for existing concrete components consists mainly on manual processes which are labor intensive and require additional material and resources. Further, these manual tests are often performed on only some of the components of the structure (specimen) and do not include all of the elements of the building, which lowers the reliability of the results.

From the structural engineering point of view, many of the above mentioned challenges can be overcome if it is assured that the physical properties of existing components are accurately and quickly assessed and this information is easily available to the structural engineer. This would enable reliable structural verifications for the existing components when they are put in a new design context.

As an example, for a typical existing wall element there are several structural properties which if properly assessed, enable the structural verification of this particular element. Some of these properties are:

- Reinforcement rebar layout
- Cover depth
- Concrete strength

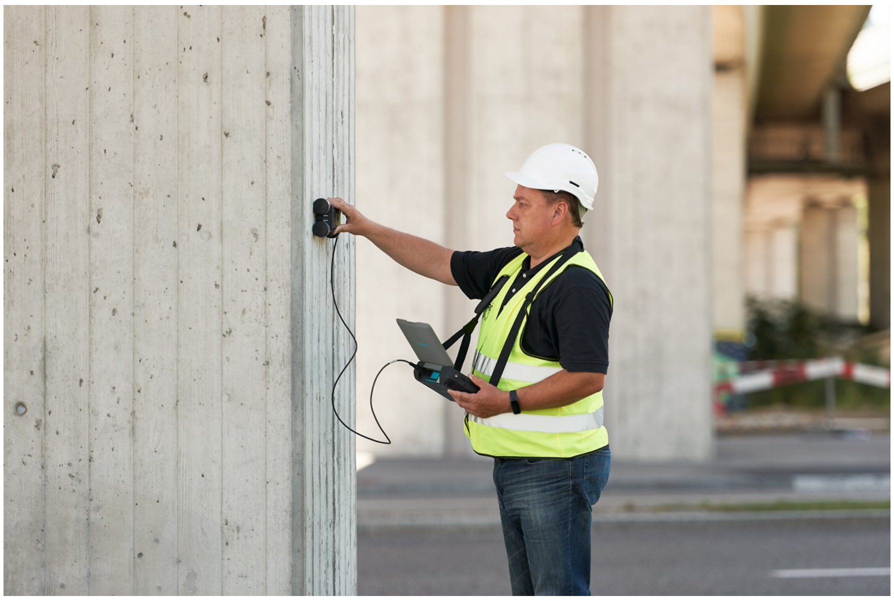


Figure 2: Manual assessment of the structural properties

If the values for these parameters (and the other parameters depending on these) are accurately assessed, a structural verification is possible which then can give insight to real capacity of an existing concrete wall.

Traditionally, there are two main types of steel rebar detectors: magnetic and electromagnetic. Magnetic detectors work by detecting the magnetic field created by the steel reinforcement in concrete. The detector's sensor contains a coil that emits a magnetic field, and when it detects a change in the field, it indicates the presence of steel reinforcement. Magnetic detectors are simple to use and inexpensive, but they have limited accuracy and are affected by the magnetic properties of the concrete, making them less reliable for non-magnetic concrete or complex reinforcement layouts.

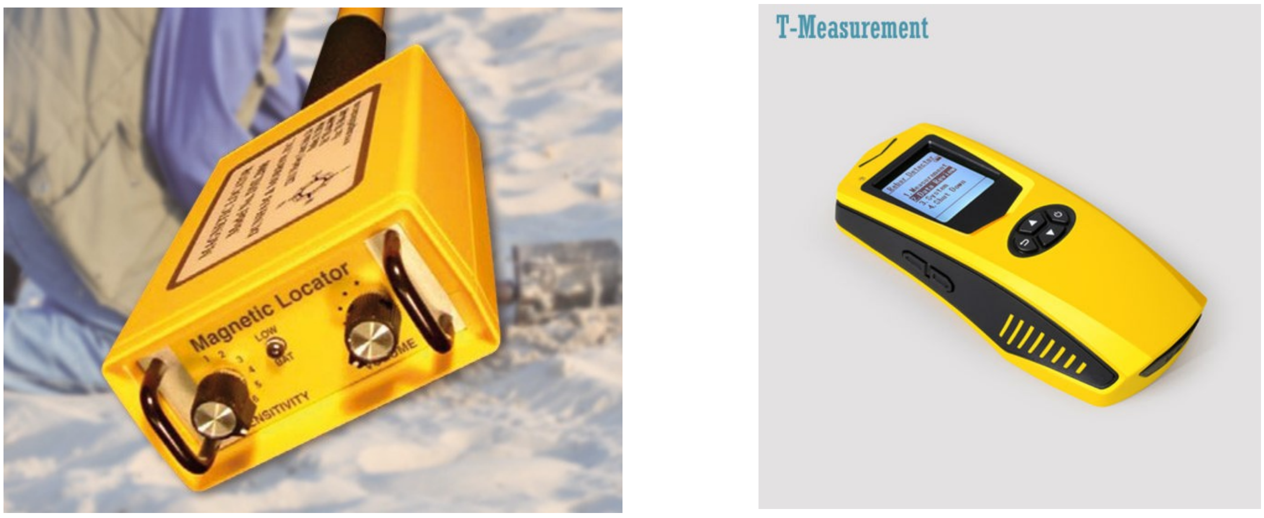


Figure 3: On the left magnetic detector and on the right electromagnetic steel detector.

Electromagnetic detectors, on the other hand, work by emitting an electromagnetic field that interacts with the steel reinforcement in concrete. The detector's sensor detects changes in the electromagnetic field and uses the information to determine the presence, depth, and approximate size of the steel. Electromagnetic detectors are more accurate and reliable than magnetic detectors, making them the preferred choice for most construction projects. However, they are more expensive and require more advanced skills to use effectively.

3. Goal

The idea presented in this paper proposes a robotic application for assessment of the internal reinforcement of an existing concrete wall which has an existing predefined reinforcement layout.

More specifically, it focuses on the process of rebar detection through the use of a radar scanner which will scan the surface of the wall and detect the location of the reinforcement along the wall. The final result is the horizontal distance of each rebar with regard to its consecutive element.

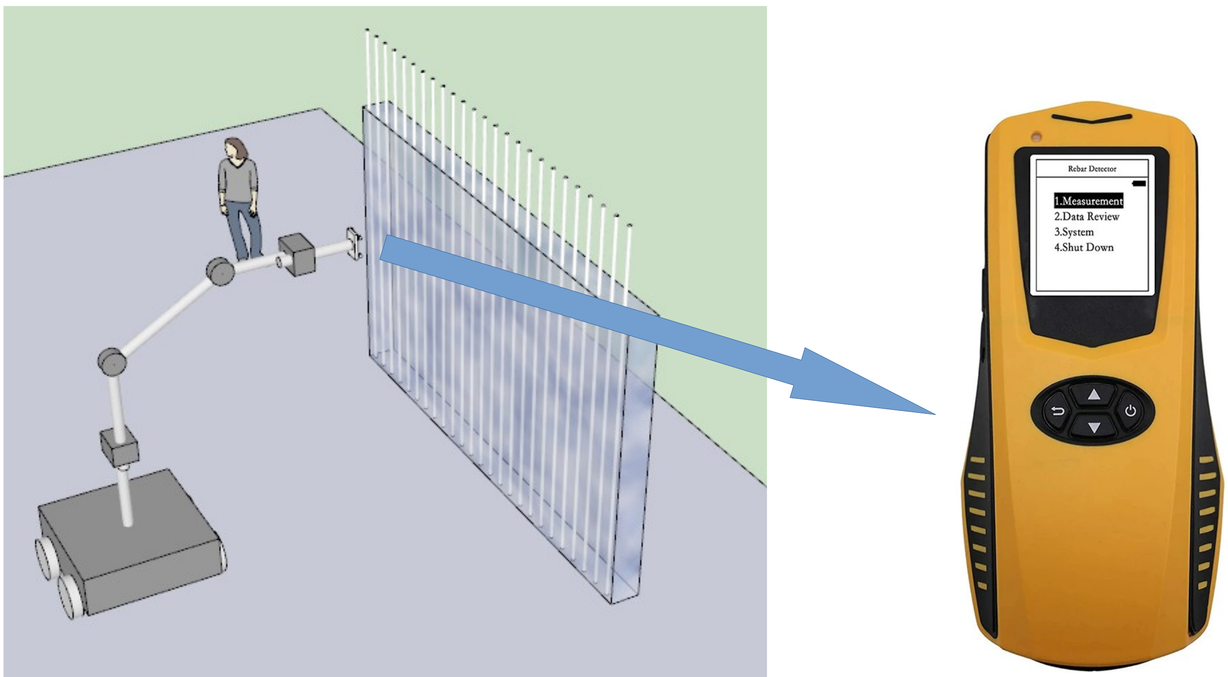


Figure4: Proposed conceptual idea

The proposed scenario foresees a robotic arm integrated with a laser scanner and a rebar detection device which is able to move along the length of the wall. During its movement the scanning of the wall surface is performed and the location of the rebar position is retrieved. This data then is transferred in real time to external applications.

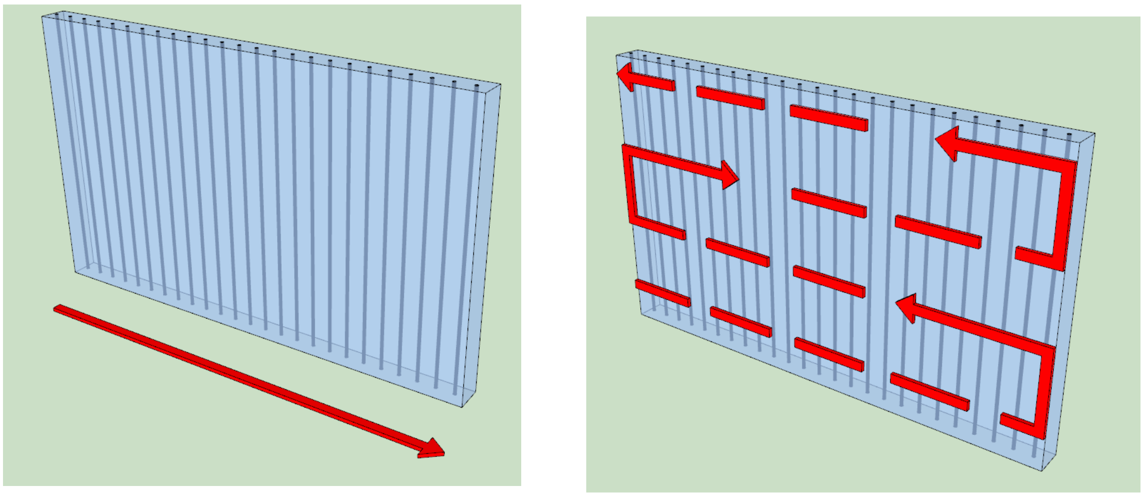


Figure 5: The scanning path

The end effector movement shall be provisioned to start from a fixed point of origin in the bottom part of the wall and then to proceed in the same height along the length of the wall. Then the effector will repeat this path backwards but with an increased height. This process continues until the whole surface of the wall is scanned.

In this way it is ensured that the scanning process is extended in all of the points of the surface which is an important aspect in the reliability of the results.

A main advantage of this path configuration is that even difficult points can be reached compared to a manual process performed by a human where this would not be possible. Further, it can detect even complex hidden reinforcement layouts that may exist in a wall component.

Another important point to mention is that this would also allow access for testing in dangerous places where the human access in not permitted such as in contaminated areas or in buildings previously hit by earthquakes where the assessment of the existing capacity is of primary importance.

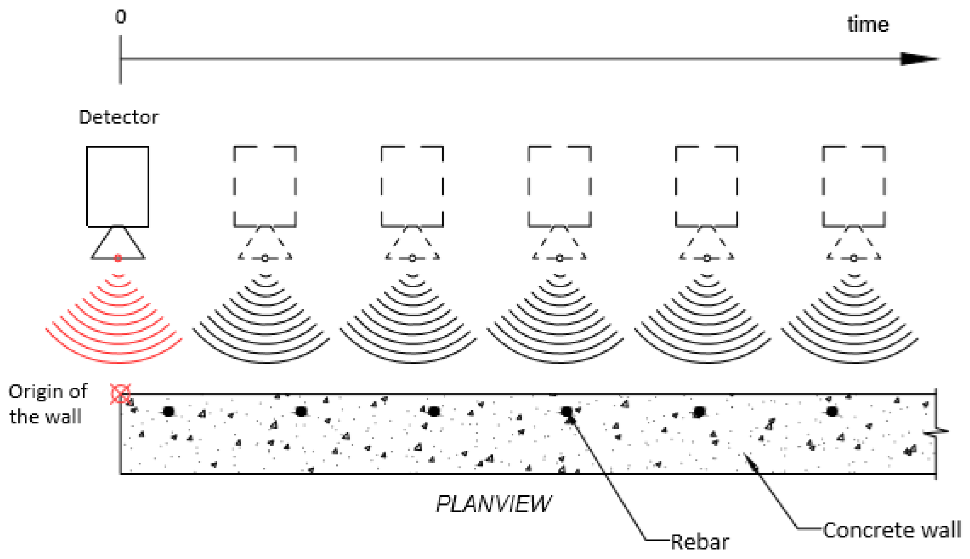


Figure 6: Detection of rebars along the wall direction

The working principle of the proposed device consists on a rebar detector which starts to move along the wall direction starting from a point of origin. This point is previously defined as the extreme edge of the wall, which will be measured through a laser scanner device. Starting from the point of origin the rebar detector continues moving until it detects the rebar. After that moment the detector gives a signal and continues moving until it detects the next rebar. By knowing the location of the two bars and the point of origin of the wall, their distance can be calculated.

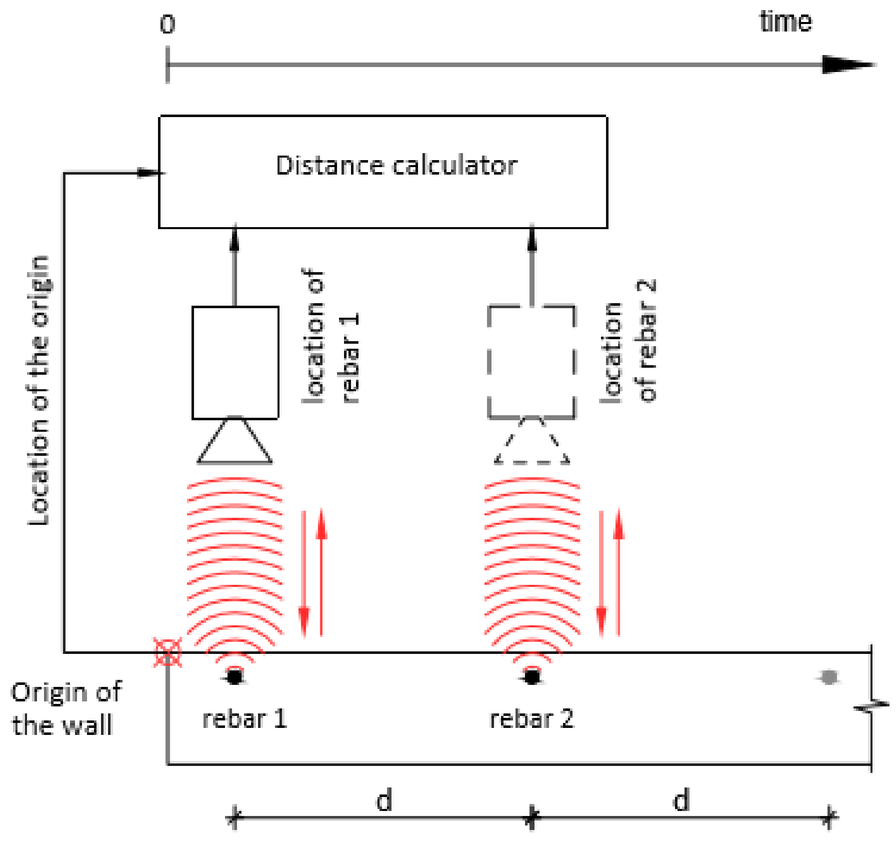


Figure 7: Schematic of working principle

5 Robotic Implementation

5.1 Use of Ros2

The Ros2 platform is chosen to simulate the virtual environment and the infrastructure of the communication between the physical objects and the physical properties to be evaluated.

In this platform, the robot device is modeled with all the required kinematics allowing it to move in the environment and to perform the processes.

Further, the environment representing the existing situation is created. This environment is composed of wall and bars modeled as markers.

The simulation of the whole process can be divided in two main parts.

The first part consists on the communication of the bars with the end effector by publishing messages which indicate their position. In the same time the end effector will receive these messages by subscribing to the relevant topic. This represents the assumption where a radar scanning device scans the rebar and the presence of the rebar is confirmed.

The second part consist on controlling the robot while scanning the surface of the wall and on the calculation module for the distance between bars. While the end effector moves along the surface of the wall it will continuously publish its location.

A custom node is designed which will listen to these messages and perform the calculation of the distance as described in the following paragraph.

5.2 Mathematical model

The mathematical model used in the code is the distance formula, which calculates the shortest distance between a point and a line in 3-dimensional space as shown in Figure 8 below. The distance formula is expressed as the square root of the sum of the squares of the difference between the x, y, and z coordinates of the point and the line point minus the square of the dot product of the difference between the x, y, and z coordinates of the point and the line direction divided by the magnitude of the line direction.

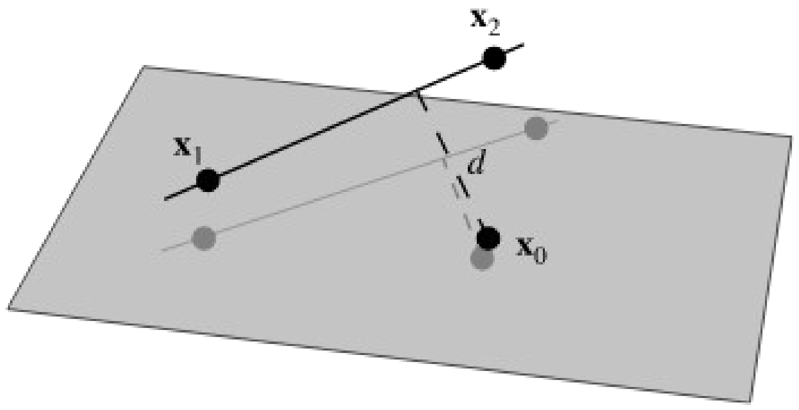


Figure 8: Distance between a point and a line in 3d formula was used.

The head of the robot is represented as a point in space with x, y, and z coordinates, which are obtained from the odometry system. The steel rebars are represented as lines with known line points and line directions in the 3-dimensional space. The distance from the head of the robot to each steel rebar is calculated using the formula for the distance between a point (x0, y0, z0) and a line defined by a point (x1, y1, z1) and a direction vector (vx, vy, vz) which is defined by:

```
distance = sqrt(sum([(point[i] - line_point[i])**2 for i in range(3)]) - (sum([(point[i] - line_point[i]) * line_direction[i] for i in range(3)])2 / sum([i2 for i in line_direction])))
```

where "sum" is the sum of elements in a list and "sqrt" is the square root.

In this code, the line points list holds the starting points of the steel bars, and the line directions list holds the direction vectors of the steel bars. The values of x, y, z, which represent the location of the robot's head, are obtained using the TransformListener class from the tf2_ros package. These values are used to calculate the distance between the robot's head and each line representing a steel bar.

If the calculated distance is less than 0.2 m, it means that the robot's head is close enough to the steel bar to detect it. The 0.2 distance assumption was assumed to represent a boundary zone surrounding the rebar inside which the rebar is detected. The head takes measurements and performs detections while hovering on the surface area of the wall.

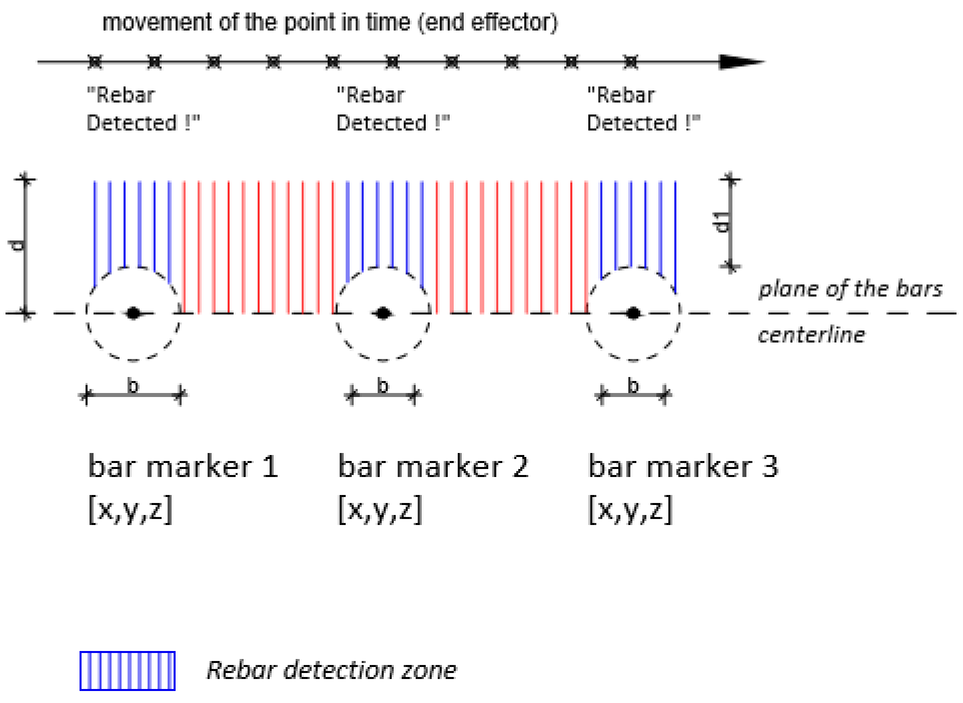


Figure 9: The principle of calculation algorithm

6. The Robot’s Definition Through the URDF file

The robot system consists of two main components: the mobile base (automobile model) and the manipulator arm (Kuka robot arm model). The Kuka robot arm is a multi-jointed arm with six links, which can move in a wide range of motion to manipulate objects. Each joint is actuated by a motor that allows precise control of the arm's movement. The six links are connected by five joints, which are capable of rotating around an axis to allow for different degrees of freedom.

The base link of the Kuka robot arm is fixedly connected to the base of the automobile model, and the entire robotic arm can rotate around this joint. This allows the robot to move and manipulate objects in a wide range of locations, as the mobile base can be positioned in different areas, while the robotic arm can reach objects in a large workspace.

The base model consists of a chassis and four wheels that are connected to the chassis through continuous joints, which allow for continuous motion of the wheels. The continuous joints provide smooth and uninterrupted movement, which is essential for mobile platforms to move efficiently in different environments.

The Kuka robot arm model consists of six **links**, which are connected by five joints. The links, in order from the base to the end-effector (the part of the arm that performs tasks), are:

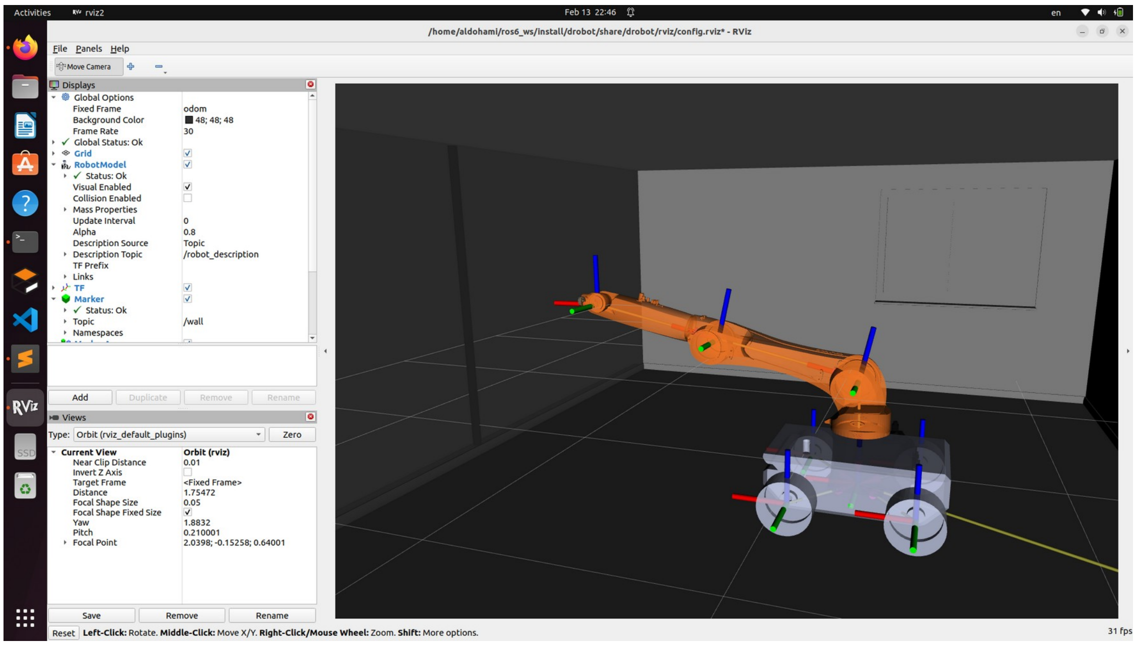
- **Base**: This is the lowest link of the robot arm, which is fixed to the mobile base.

- **Shoulder**: This link is connected to the base through the base-shoulder joint and can rotate around the vertical axis.
- **Arm 1**: This link is connected to the shoulder through the shoulder-arm1 joint and can rotate around the horizontal axis.
- **Middle**: This link is connected to the arm1 link through the arm1-middle joint and can also rotate around the horizontal axis.
- **Arm 2**: This link is connected to the middle link through the middle-arm2 joint and can rotate around the horizontal axis.
- **Head**: This is the end-effector of the robot arm, which is connected to the arm2 link through the arm2-head joint. It is the part of the robot that interacts with the environment or objects it is manipulating.

The five **joints** that connect the links are:

1. **Base-shoulder joint**: This joint connects the base to the shoulder and allows the arm to rotate around the vertical axis.
2. **Shoulder-arm1 joint**: This joint connects the shoulder to arm1 and allows the arm to rotate around the horizontal axis.
3. **Arm1-middle joint**: This joint connects arm1 to middle and allows the arm to rotate around the horizontal axis.
4. **Middle-arm2 joint**: This joint connects middle to arm2 and allows the arm to rotate around the horizontal axis.
5. **Arm2-head joint**: This joint connects arm2 to the head, which is the end-effector of the robot.

These links and joints work together to give the robot arm the ability to reach and manipulate objects in different locations and orientations. The mobility of the mobile base allows the robot arm to be positioned in different areas, while the manipulator arm can reach objects in a wide workspace.



```
Global Options
  Fixed Frame            odom
  Background Color       48; 48; 48
  Frame Rate             30
Global Status: Ok
Grid                     ✓
RobotModel               ✓
  Status: Ok
  Visual Enabled         ✓
  Collision Enabled
  Mass Properties
  Update Interval        0
  Alpha                  0.8
  Description Source     Topic
  Description Topic      /robot_description
  TF Prefix
  Links
TF                       ✓
Marker                   ✓
  Status: Ok
  Topic                  /wall
  Namespaces
```

```
Type: Orbit (rviz_default_plugins)   Zero
Current View             Orbit (rviz)
  Near Clip Distance     0.01
  Invert Z Axis
  Target Frame           <Fixed Frame>
  Distance               1.75472
  Focal Shape Size       0.05
  Focal Shape Fixed Size ✓
  Yaw                    1.8832
  Pitch                  0.210001
  Focal Point            2.0398; -0.15258; 0.64001
```

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.

Figure 10: Robotic model imported to RVIZ with assigned links and joints in URDF file.

7. Control Package of Robot

The control package for a robot involves multiple nodes and scripts working together to ensure that the robot moves as intended. These nodes and scripts have specific roles, from managing the life-cycle of the package to ensuring that RVIZ displays the robot's movements accurately. For example, the Jointstate_joystick.py node plays a critical role in translating the user's joystick commands into joint movements for the robot. On the other hand, the Setup.py script prepares the package for use and provides a convenient interface for starting and stopping the package. Additionally, the Meshes folder stores files containing 3D geometry for the robot and its components, which RViz uses to visualize the robot's movements and appearance accurately. Together, these nodes, scripts, and files work to ensure that the robot moves as intended and that users can interact with it effectively.

7. 1 Jointstate_joystick.py

Jointstate_joystick.py is a Python node that communicates with other nodes in the robot's control package. This node has two primary responsibilities. The first aim is to read the current joint positions of the robot and share them with other nodes. This allows other nodes in the package to know the robot's current state, such as where the arm is positioned and the orientation of the head.
The second responsibility of Jointstate_joystick.py is to translate the user's joystick commands into joint movements for the robot.
To achieve this, the node maps specific joystick buttons to the required changes in joint positions and robot movement. For example, axes buttons of the joystick are used to move the robot in translational directions, while axes [2] and [3] are used to control the arm, as shown in the snippet below.

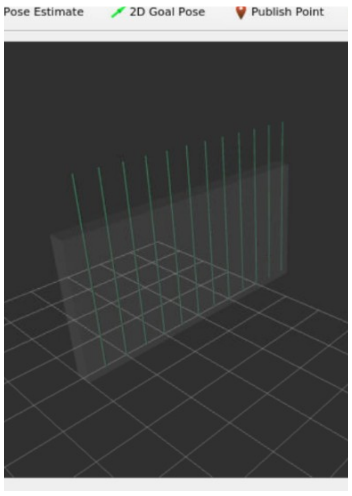
```python
def getJoystickInput(self, msg):
        self.x += self.scale_factor * msg.axes[1]
        self.y += self.scale_factor * msg.axes[0]
        self.sa = max(-90, min(18, self.sa + msg.axes[2]))
        self.a2h = max(-50, min(70, self.a2h + msg.axes[3]))
```

This node plays a crucial role in the low-level control of the robot's joints and overall movement. By communicating with the other nodes and receiving joystick commands, Jointstate_joystick.py ensures that the robot moves in the intended direction and performs the desired actions. Without this node, controlling the robot's movements would be more challenging, and the robot may not respond as intended to the user's input.

7. 2 Custom Nodes

Wall.py

Node	Type	Topic
wall.py	Marker	“wall”



This node is responsible for creating parts of the environment related to the existing wall to be scanned. It consist on a Marker object with geometrical properties related to a wall element. The geometrical representation is achieved through a mesh file in a separate folder.

bar.py

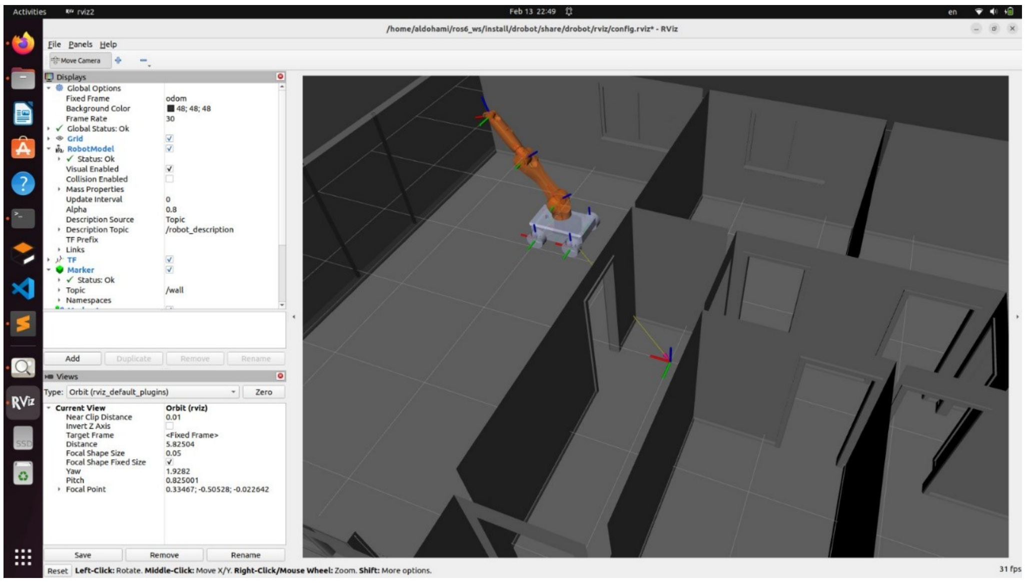
Node	Type	Topic
bar.py	MarkerArray	“bar”

This node represents the rebar elements inside the wall which have to be detected by the end effector device. It is modeled as a marker array which publishes MarkerArray messages defining the location of the bars in the wall.

One of the advantages for using the marker array for the bars is the flexibility it offers to create different pattern of wall reinforcement as it may be the case in the real life.

environment.py

Node	Type	Topic
environment.py	Marker	“environment”



```
Global Options
  Fixed Frame            odom
  Background Color       48; 48; 48
  Frame Rate             30
Global Status: Ok
Grid                     ✓
RobotModel               ✓
  Status: Ok
  Visual Enabled         ✓
  Collision Enabled
  Mass Properties
  Update Interval        0
  Alpha                  0.8
  Description Source     Topic
  Description Topic      /robot_description
  TF Prefix
  Links
TF                       ✓
Marker                   ✓
  Status: Ok
  Topic                  /wall
  Namespaces

Type: Orbit (rviz_default_plugins)
Current View             Orbit (rviz)
  Near Clip Distance     0.01
  Invert Z Axis
  Target Frame           <Fixed Frame>
  Distance               5.82504
  Focal Shape Size       0.05
  Focal Shape Fixed Size ✓
  Yaw                    1.9282
  Pitch                  0.825001
  Focal Point            0.33467; -0.50528; -0.022642

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.
```

This node creates a more realistic scene in Rviz by presenting a space with several walls in different plan configuration, where the robot can move until it reaches the objective which is the wall to be scanned.

7. 3 Launch.py

Launch.py is a Python node responsible for starting and stopping other nodes in the package. It is the primary node responsible for managing the life/cycle of the entire package. This node launches other nodes required for the robot to work, confirming that all the nodes required for the package to work are started and running as expected. It also provides an easy way to stop all the nodes in the package when needed. When the package stops, Launch.py shuts down all the other nodes, ensuring that the robot is not left in an unintended state.
The get_package_share_path function is used to get the path of the package, and the os.path.join function is used to create the path of the URDF and RVIZ files.

The Node function is used to specify the nodes to be launched, including the jointstate_joystick, robot_state_publisher, joy_node, rviz2, static_transform_publisher, wall, bar, and enviroment.
The executable parameter specifies the name of the node to be launched, while the output parameter specifies where the output of the node should be printed. The namespace parameter is used to group the nodes together, and the arguments and parameters parameters are used to pass arguments and parameters to the nodes. Finally, the LaunchDescription object is returned, and the ld.add_action function is used to add the nodes to the launch file. The resulting launch file can be executed using the ros2 launch command.

7.4 Config.rviz

Config.rviz node ensures that RViz displays the robot's movements and 3D geometry correctly. It ensures the names and options used in other nodes are consistent and accurate for visualization. The node is responsible for configuring RViz to display the robot and its movements accurately.

7.5 Setup.py

Setup.py: This script prepares the package to run on the command line. It brings in the necessary nodes and functions, sets up the package dependencies, and sets up the command-line interface for starting and stopping the package. The script is responsible for setting up the package for use and provides a convenient interface for starting and stopping the package.
The data_files argument is a list of tuples that specifies additional files to be included in the package distribution. Each tuple contains a relative path to the directory where the file(s) should be installed, and a list of files matching a glob pattern.
(os.path.join('share', package_name, 'meshes'), glob.glob('meshes/*.dae')),
for instance reading the meshes files, it searches for the location where mesh files with .dae extension are located, and it's using the glob module to identify all such files in the meshes directory. os.path.join function is used to join different parts of the path to create a complete file path. The resulting file paths will be used to copy these mesh files to a directory under the package's installation directory.

7.6 Distance_Calculator.py

This code is used o simulate the detection of steel rebars in reinforced concrete walls. The code uses the ROS Client Library (rclpy) and the ROS Transform Library (tf2_ros) to handle the transformations of data between different coordinate frames, as seen below.

```
import rclpy
import tf2_ros
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener
import tf_transformations as tf_trans
import tf2_ros
import tf2_geometry_msgs
from tf2_msgs.msg import TFMessage
from rclpy.node        import Node
from math import sqrt
import csv
```

The code defines a class named "dist_calc", which is a subclass of the "Node" class from the rclpy library. This class is responsible for performing the rebar detection simulation:

```
class dist_calc(Node):

    def __init__(self, name):
        super().__init__(name)

        self.sub = self.create_subscription(TFMessage, '/tf',
self.print_translation_values, 10)

        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)

        self.rebar_detections = []
```

The **init** method of the dist_calc class is responsible for setting up the required variables and subscriptions. A subscription is created to the "/tf" topic, which is a standard topic in ROS for transmitting transform information between coordinate frames.

The callback function for this subscription, "print_translation_values", will be called whenever a new message is received on the "/tf" topic. The method also sets up a buffer to store the transform information and a transform listener to receive the transform data.

```
def print_translation_values(self, msg):
        line_points = [[3.6, 0.0, 0.5], [3.6, 1.0, 0.5], [3.6, -1, 0.5]]
        line_directions = [[0.000001, 0.000001, 1], [0.000001, 0.000001,
1], [0.000001, 0.000001, 1]]

        try:
            transform = self.tf_buffer.lookup_transform("odom","head",
rclpy.time.Time())
            x = transform.transform.translation.x
            y = transform.transform.translation.y
            z = transform.transform.translation.z
            t = self.get_clock().now()
            point = [x, y, z]
            for i in range(3):
                line_point = line_points[i]
                line_direction = line_directions[i]
                distance = sqrt(sum([(point[i] - line_point[i])**2 for i in
range(3)]) - (sum([(point[i] - line_point[i]) * line_direction[i] for i in
range(3)])**2 / sum([i**2 for i in line_direction])))
                if distance < 0.2:
                    self.rebar_detections.append([t, x, y, z, distance])
                    print("Time: {}, Translation: x = {}, y = {}, z = {},
Distance to line = {}".format(t, x, y, z, distance))
                    print("Rebar is detected")

            with open("/home/aldohami/Desktop/rebar_detections.csv", "w",
newline="") as csvfile:

                writer = csv.writer(csvfile)
                writer.writerow(["time", "x", "y", "z", "distance"])
                writer.writerows(self.rebar_detections)

        except (tf2_ros.LookupException, tf2_ros.ConnectivityException,
tf2_ros.ExtrapolationException):
            self.get_logger().info("Error")
```

The print_translation_values method is the callback function for the "/tf" topic subscription. This method is responsible for performing the rebar detection simulation. It starts by looking up the latest transform information between the "odom" and "head" frames using the "lookup_transform" method of the TransformListener object. This method retrieves the latest transform information from the buffer and uses it to calculate the current position of the "head" frame.

The line_points and line_directions lists are predefined geometries representing the hypothetical locations of the rebars. The method then iterates over these predefined geometries and calculates the distance between the current position of the "head" frame and the hypothetical rebar. If the distance is less than 0.2, the method considers it a detection of the rebar and appends the time, position, and distance information to the rebar_detections list.

Finally, the rebar_detections list is written to a CSV file on the disk for later analysis. The information stored in the file includes the time of detection, the x, y, and z position of the "head" frame, and the calculated distance to the rebar.

The main method initializes the ROS system and creates an instance of the dist_calc class. The rclpy.spin method is called to start the ROS event loop, and the destroy_node method is called to properly shut down the ROS system when the event loop is exited, as shown below.

```
def main(args=None):
    rclpy.init(args=args)
    clac = dist_calc("distance_calculator")
    rclpy.spin(clac)
    clac.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

The script “joinstate_joystick.py”,as explained earlier, creates a ROS node called control_your_robot which subscribes to a joystick input topic and publishes to two topics: joint_states and odom (odometry).

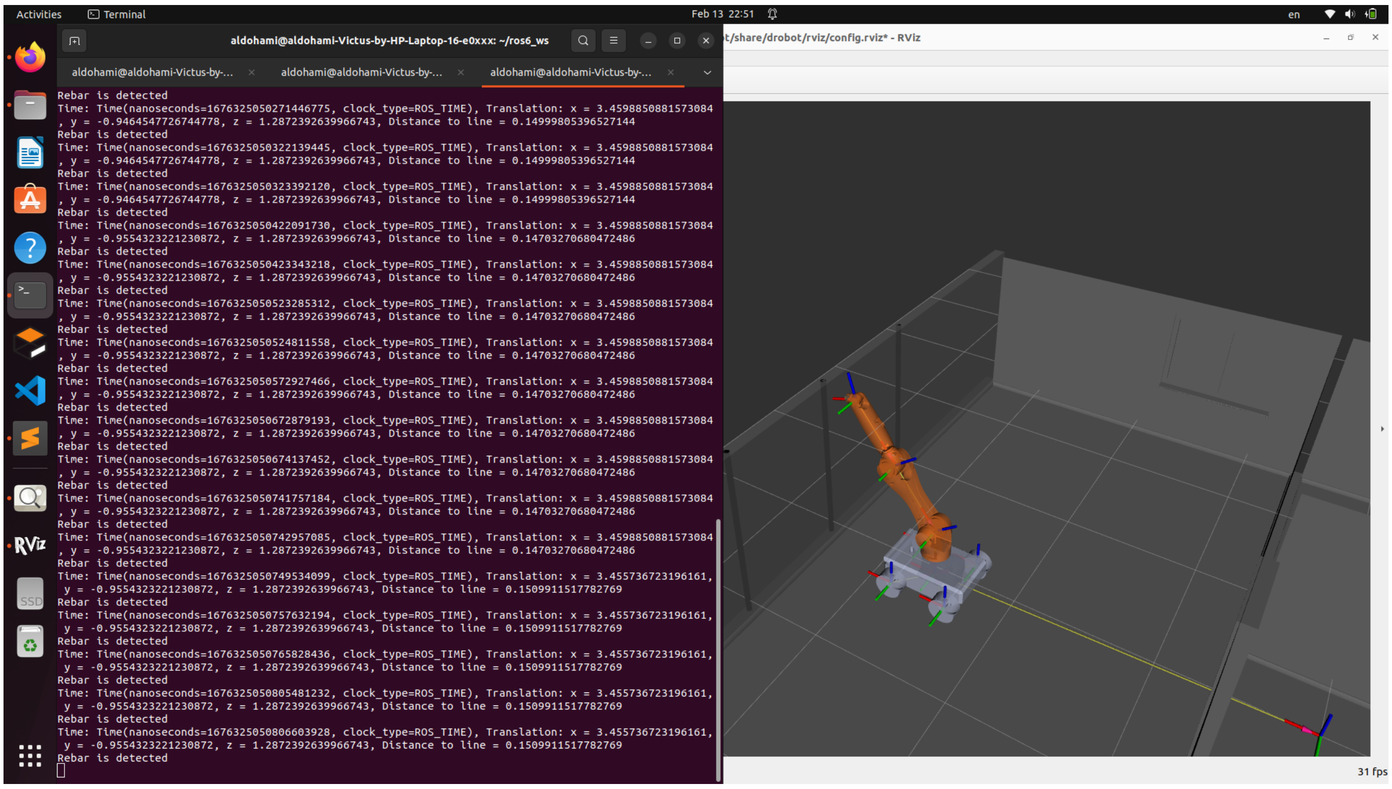
The joystick input topic provides the movement control for the robot in the x and y direction and the rotations around the z and y axes (sa and a2h). The node scales these inputs and updates the odom_trans object, which represents the transformation from the odom frame to the base_link frame. This transformation is then broadcasted using the odom_broadcaster.

The joint_state object represents the joint angles for the robot and is updated with the latest rotation values for sa and a2h. The joint_state object is then published on the joint_states topic.

The main function initializes the ROS system, creates an instance of the control_your_robot node, and starts the ROS spin loop to keep the node running.

By providing a way to control the position and orientation of the robot, this script allows for more dynamic and realistic simulations in the rebar detection scenario. The updated position and orientation can be used by the rebar detection algorithm to generate more accurate results.

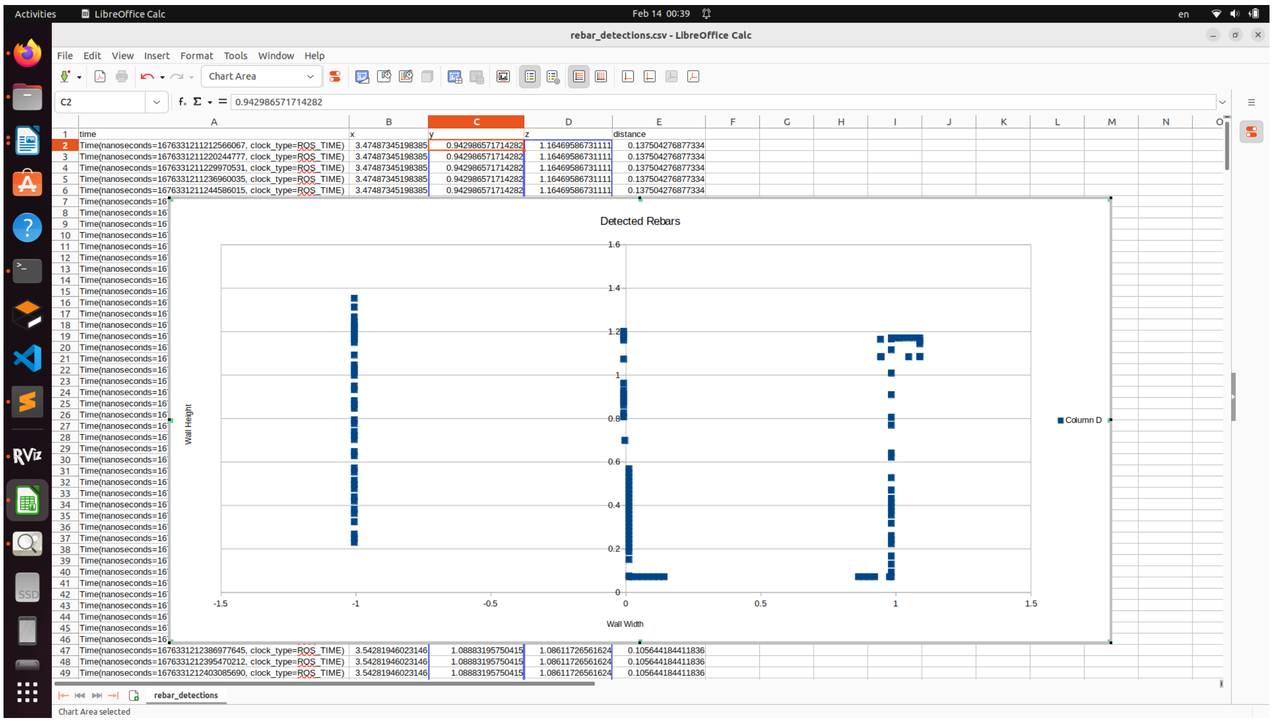
The benefits of this simulation for the construction industry include the ability to efficiently and accurately detect rebars in reinforced concrete walls without having to physically inspect the walls. This can save time and resources and also reduce the risk of injury to workers. The simulation can also be used to test and refine rebar detection algorithms and equipment before they are deployed in the field.



```
Rebar is detected
Time: Time(nanoseconds=1676325050271446775, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9464547726744778, z = 1.2872392639966743, Distance to line = 0.14999805396527144
Rebar is detected
Time: Time(nanoseconds=1676325050322139445, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9464547726744778, z = 1.2872392639966743, Distance to line = 0.14999805396527144
Rebar is detected
Time: Time(nanoseconds=1676325050323392120, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9464547726744778, z = 1.2872392639966743, Distance to line = 0.14999805396527144
Rebar is detected
Time: Time(nanoseconds=1676325050422091730, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050423343218, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050523285312, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050524811558, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050572927466, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050672879193, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050674137452, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050741757184, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050742957085, clock_type=ROS_TIME), Translation: x = 3.4598850881573084
, y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.14703270680472486
Rebar is detected
Time: Time(nanoseconds=1676325050749534099, clock_type=ROS_TIME), Translation: x = 3.455736723196161,
 y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.1509911517782769
Rebar is detected
Time: Time(nanoseconds=1676325050757632194, clock_type=ROS_TIME), Translation: x = 3.455736723196161,
 y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.1509911517782769
Rebar is detected
Time: Time(nanoseconds=1676325050765828436, clock_type=ROS_TIME), Translation: x = 3.455736723196161,
 y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.1509911517782769
Rebar is detected
Time: Time(nanoseconds=1676325050805481232, clock_type=ROS_TIME), Translation: x = 3.455736723196161,
 y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.1509911517782769
Rebar is detected
Time: Time(nanoseconds=1676325050806603928, clock_type=ROS_TIME), Translation: x = 3.455736723196161,
 y = -0.9554323221230872, z = 1.2872392639966743, Distance to line = 0.1509911517782769
Rebar is detected
```

Activities
Terminal
Feb 13 22:51
aldohami@aldohami-Victus-by-HP-Laptop-16-e0xxx: ~/ros6_ws
t/share/drobot/rviz/config.rviz* - RViz
31 fps

Figure 11: The robot scanning detecting rebars inside the wall.



	A	B	C	D	E
1	time	x	y	z	distance
2	Time(nanoseconds=1676331211212566067, clock_type=ROS_TIME)	3.47487345198385	0.942986571714282	1.16469586731111	0.137504276877334
3	Time(nanoseconds=1676331211220244777, clock_type=ROS_TIME)	3.47487345198385	0.942986571714282	1.16469586731111	0.137504276877334
4	Time(nanoseconds=1676331211229970531, clock_type=ROS_TIME)	3.47487345198385	0.942986571714282	1.16469586731111	0.137504276877334
5	Time(nanoseconds=1676331211236960035, clock_type=ROS_TIME)	3.47487345198385	0.942986571714282	1.16469586731111	0.137504276877334
6	Time(nanoseconds=1676331211244586015, clock_type=ROS_TIME)	3.47487345198385	0.942986571714282	1.16469586731111	0.137504276877334
7	Time(nanoseconds=16				
8	Time(nanoseconds=16				
9	Time(nanoseconds=16				
10	Time(nanoseconds=16				
11	Time(nanoseconds=16				
12	Time(nanoseconds=16				
13	Time(nanoseconds=16				
14	Time(nanoseconds=16				
15	Time(nanoseconds=16				
16	Time(nanoseconds=16				
17	Time(nanoseconds=16				
18	Time(nanoseconds=16				
19	Time(nanoseconds=16				
20	Time(nanoseconds=16				
21	Time(nanoseconds=16				
22	Time(nanoseconds=16				
23	Time(nanoseconds=16				
24	Time(nanoseconds=16				
25	Time(nanoseconds=16				
26	Time(nanoseconds=16				
27	Time(nanoseconds=16				
28	Time(nanoseconds=16				
29	Time(nanoseconds=16				
30	Time(nanoseconds=16				
31	Time(nanoseconds=16				
32	Time(nanoseconds=16				
33	Time(nanoseconds=16				
34	Time(nanoseconds=16				
35	Time(nanoseconds=16				
36	Time(nanoseconds=16				
37	Time(nanoseconds=16				
38	Time(nanoseconds=16				
39	Time(nanoseconds=16				
40	Time(nanoseconds=16				
41	Time(nanoseconds=16				
42	Time(nanoseconds=16				
43	Time(nanoseconds=16				
44	Time(nanoseconds=16				
45	Time(nanoseconds=16				
46	Time(nanoseconds=16				
47	Time(nanoseconds=1676331212386977645, clock_type=ROS_TIME)	3.54281946023146	1.08883195750415	1.08611726561624	0.105644184411836
48	Time(nanoseconds=1676331212395470212, clock_type=ROS_TIME)	3.54281946023146	1.08883195750415	1.08611726561624	0.105644184411836
49	Time(nanoseconds=1676331212403085690, clock_type=ROS_TIME)	3.54281946023146	1.08883195750415	1.08611726561624	0.105644184411836

Figure 14: Detected rebars graph exported to CSV file

8. Conclusion

In this paper a simulation of a robotic process is performed in a scenario taking place inside a construction site. The process simulates a robot which performs on-site assessment non-destructive test conducted for existing concrete elements part of the building.

The goal is to automate this process not only for increasing the effectivity of the process itself but also to accurately assess the structural properties of the building element with high degree of information quality. This information can then be used in external domains such as Building Information Modeling as a digital asset.

Application of the Ros platform allowed for a feasible approach by making possible establishing an interface of communication between the robot and the typical building elements such as reinforcement bars.

By adapting the nodes available in Ros and its open source features, a simulation of the robotic process was possible, with the final result obtaining the required data.

Further work is required related to physical simulation of the devices such as electromagnetic metal detectors and their integration with external measurements devices.