Robotics Software Engineer Technical Assessment

This assessment is divided into three steps, designed to build upon each other. They will give you a feel for the type of work we do and help us understand your skills in areas crucial to this role. The project should be completed within one week.

Please submit your work as a link to a single public Git repository containing the solutions for all steps. We expect to see a clear and logical commit history that reflects your progress through the tasks.

We appreciate you taking the time to complete this challenge and are excited to see your work.

Assessment Overview

This assessment will guide you through setting up a development environment, solving a core robotics math problem, and finally, visualizing a robot model—all common tasks for a Robotics Software Engineer at our company.

Step 1: Development Environment with Docker

Tasks:

1. **Create a Dockerfile:**
    - Start from an official ubuntu:22.04 base image.
    - Inside the Dockerfile, perform all the necessary steps to install the latest **ROS2 LTS release (Humble Hawksbill)**. We recommend the ros-humble-desktop variant.
    - Ensure the ROS2 environment is automatically sourced (e.g., in .bashrc) so that ROS2 commands are available upon entry.
    - Install also the c++ linear algebra “Eigen”
    - Install also the yaml-cpp library
    - Install also the “rviz_visual_tools” ros2 package”
2. **Create a docker-compose.yml file:**
    - This file should define a service that builds your Dockerfile.
    - It should mount a local src directory (which will later contain your ROS2 packages) into a ROS2 workspace (e.g., /ros2_ws/src) inside the container.
    - Configure it to keep the container running after launch (e.g., using tty: true).

Deliverables for Step 1:

- A Dockerfile at the root of your repository.
- A docker-compose.yml file at the root of your repository.
- A README.md with a section explaining how to build and run the Docker container.

Step 2: C++ ROS2 Service for Linear Algebra

Tasks:
You have to implement a ros2 service and 2 c++ nodes, one that creates a server and the second the client. The client should load from a yaml file an M x 3 matrix A (M >= 3) and a 3d vector b. The client node should call the server with request the matrix A and vector b. When the server node receives a request, it should find the least squares solutions of ||Ax - b||. Employ the "Eigen" c++ library to find the solution. Then, the solution x should be rotated by a random 3d rotation matrix R' and displaced by a random 3d vector d'. The server should then send as response the rotated-and-displaced solution x' along with R' and d'. When the client receives the response, it should use R' and d' to apply the inverse transformation to x' and get x. Then, the client should publish x to a topic.

The server should also define a subscriber in the topic that the client publishes and a separate thread that waits to be notified by a condition variable. When a message is received, the server should notify the thread that waits on the condition variable. When the thread that waits on the condition variable is notified, the thread should print the message that was received from the client's topic, and then wait to be notified again for future messages.

For the request and response of the service, use the appropriate existing messages from the geometry_msgs ros package.

Place the service in a separate package, and the server and client ros nodes in a common package.

Deliverables for Step 2:

- The complete linear_algebra_service package inside the src directory.
- Your README.md should be updated with instructions on how to:
  - Build the new package within your Docker container (using colcon build).
  - Run the service server and client nodes and verify that they work properly.

Step 3: URDF Visualization and Robot State Control

Tasks:

1. **Integrate the UR20 Model:**
   - Add a new ROS2 package to your workspace that contains the URDF for the Universal Robots UR20. The official ROS2 description can be found at UniversalRobots/Universal_Robots_ROS2_Description. You can include this as a git submodule or provide clear instructions in your README on how to clone it into the src directory.
   - Add as a gripper a fixed link with a box geometry (0.3 x 0.4 x 0.2), connected through a fixed joint at the flange of the robot.
2. **Create a State Publisher Node:**

- Create a new ROS2 package named ur20_display.
- Inside this package create a node that should read from the parameter server a joint configuration and publish that joint configuration as a sensor_msgs/msg/JointState. It should also create a tf listener that reads the transform of the gripper link relative to the robot's elbow link (Tf_elbow_gripper), the transform of the elbow link relative to the world frame (Tf_world_elbow) and the transform of the gripper relative to the world frame (Tf_world_gripper). Convert these transforms to "Eigen::Isometry3d" and use Tf_elbow_gripper and Tf_world_elbow to verify that Tf_world_gripper is correct. Then use the rviz_visual_tools package to publish Tf_world_gripper as a x-y-z oriented frame and also publish a text label with the name "Tf_elbow_gripper".
- **Bonus 1**: in the same node, create a random goal joint configuration and generate a periodic trajectory from the current to the goal configuration (e.g. using a sinusoid with a period of your choice). Generate a trajectory for 1.5 periods. In a for loop, publish each point of the periodic trajectory you created as a sensor_msgs/msg/JointState message and add appropriate sleep between each loop iteration to visualize the animated trajectory in rviz.
- **Bonus 2:** Publish the periodic trajectory you created to a topic of your choice. Create a python node (in the same* or in a separate package) that listens to this topic, and when it receives a trajectory message, it plots the trajectory for each joint (e.g. using matplotlib)
  *this can be a little tricky...

3. **Create a Launch File:**
   - Develop a ROS2 launch file (.launch.py) that:
     - Loads the UR20's URDF from the description package.
     - Starts the standard robot_state_publisher node.
     - Starts your own created node (passing also to it as parameter a joint configuration) from the ur20_display package.
     - Starts RViz2, pre-configured with a provided .rviz file to display the robot model.
4. **Create an RViz Configuration:** Save an RViz configuration (.rviz) that is set up to visualize the UR20 model correctly. Add any other required plugins (e.g. to visualize any markers published through the rviz_visual_tools)

Note: Normally you cannot visualize stuff that run inside your container. To allow your container to have access to your local display, check out this:

https://wiki.ros.org/docker/Tutorials/GUI
(the 1st proposed solution is sufficient)

Deliverables for Step 3:

- The ur20_display package, including the C++ node, launch file, and .rviz file.
- Your README.md must be updated with:
  - Instructions on how to get the URDF dependency.
  - A single command to run your launch file and visualize the robot in the specified pose.

- A screenshot of the UR20 in RViz in its final configuration.

Final Submission

Ensure your Git repository is well-organized with a clear commit history and that the final README.md file contains all the requested information and instructions.

If you have any questions, please don't hesitate to reach out. We look forward to seeing your solution.

Best regards,

The Progressive Robotics Team

General comments:

Use the ros logging to print INFO and DEBUG messages.

Add documentation to your code.

Follow OOP and best coding practices.

No clarifications will be provided. In case you are unsure what to do or you think something is missing, feel free to improvise.

The aim of this assignment is to assess your knowledge and understanding on basic fields, tools and concepts required for this job.

If you are unfamiliar with some concepts, it is a good opportunity to dwell on them and demonstrate your ability to learn quickly and fast.

Use of AI tools is allowed strictly only if you understand the code you wrote. During the live interview you will be asked to explain and apply on the fly changes to the code you submitted, so if you do not fully grasp your code, you will not be able to cope with the live questions. It would be really awkward and unpleasant both for you and us to find out that you actually “cheated”. Only if you believe that something was blocking you from proceeding to the next question/assignment, can you use AI tools, but state clearly where you did so. Be honest and frank (those qualities are greatly valued)