7MR10070: Software and Robotic Integration

Semester 2

Final Project

Written by Alexandros Megalemos

# ROS-Slicer integration and robotic control

# Introduction

Before describing why integrating image processing and robotic simulation is useful in medical applications, we need to describe if they are useful on their own.

## Image Processing

Image processing (i.e. segmentation, registration, image guided systems) has proven to be a vital tool in the medical field. Some of its uses include:

* Improving/Restoring the quality of images in order to help physicians pick out important details/diseases in them
* Through machine learning and neural network techniques, extracting features from the image and learning how to detect said details/diseases automatically or separating different tissues from each other again in order to assist physicians
* In combination with machine learning techniques, assisting untrained physicians in figuring out how pick out certain details in unseen scenarios by learning how to generate new examples[1]
* Assisting surgeons in surgeries by providing visual guidance for procedures

## Robot Simulation

There is a huge need of robotic assistance in the medical field. Whether it is in surgery or in assistive robotics, robots will start playing a bigger role in medicine. There is a multitude of reasons for this and some of them are:

* Robots provide high precision movement
* Assuming a model/procedure has been figured out, creating a new robot takes far less time than training a new physician/surgeon
* Fills in certain roles that people lack interest in doing so themselves (such as nursing positions)
* Reduces the bus factor[[1]](#footnote-1) the field currently suffers from by heavily relying on individual people

Given the sensitive subject and that the intended use of robots can have long lasting or even fatal consequences if something does not go as planned, simulations help figure out certain problems before using a robot on an actual person. More specifically, robotic simulations can be used to:

* Determine if the specifications provided satisfy the task at hand
* Identify any design flaws early on
* Provide the ability to test multiple configurations to decide which configuration is better for the task
* By isolating software and hardware components, help figure out which part of the robot might cause/is causing an issue

## Integrating the two

Integrating image processing and robotic simulation allows us to optimise our system, improving performance both in terms of accuracy and execution time. For example, using image processing techniques we can plan a path or choose the best starting point in a surgery simulation and then have the robot perform the task based on that optimisation. Furthermore, using proper image processing techniques, we can recognise important/vulnerable areas that the robot can be programmed to take better care around.

## Developing an end-to-end pipeline

Developing an end-to-end pipeline is probably one of the most important parts of robotic and imaging integration. By developing an end-to-end pipeline, we can figure out certain problems that will arise to individual parts of the pipeline right from the beginning. For example, if we know that a specific part of our image processing algorithm consistently underperforms (and can’t fix it), we can configure our robot to take that into account. This would not have been possible if the two parts of the system were designed in isolation.

Moreover, designing the architecture and how different modules and components interact with each other, we allow for better scaling of the system when new features need to be added. Furthermore, proper design and implementation can help make certain components re-usable, speeding up the process of development as well as allow other developers/organisations to take advantage of said components in order to solve greater problems faster.

After designing the pipeline, we can then use automated (and manual) validation techniques in order to assess the accuracy and robustness of our whole system. Using unit testing we can see how each individual part performs and once we are sure every part works well on its own, we can start using integration testing. Here all (or multiple) components are tested together, thus ensuring that our framework can work well from one end to the other. Of course, we must still be cautious as if we have not designed our tests or model correctly, we can miss important issues in the system and under the false assumption of passing tests go into manufacturing which might be a waste of time and money. In even worse cases, we can cause serious harm to a person in a real-life scenario.

# Methodology

## 3D slicer path planning

The path planning aspect of the pipeline is separated into 4 different tasks, performed in the following order:

1. Avoidance of a critical structure
2. Placement of the tool into a target structure
3. Trajectory is below a certain length
4. Maximizing distance from critical structures

The path planning steps are executed in the order specified above. We start with the least complex (in terms of time and space) algorithm and work our way down to the most complex. Therefore:

1. First, create the OBB trees required for each task
2. Then, filter for targets that are within the hippocampus
3. After that, filter for entry/target trajectories that do not pass through the ventricles
4. After that, we filter for entry/target trajectories that do not pass through blood vessels
5. Finally, we filter so that only trajectories of a certain angle (degrees) are accepted

By doing this, we rule out most of the trajectories before we reach the more expensive checks in our overall algorithm. This is clearly demonstrated by the total time taken of ~25 seconds, whereas before filtering just for valid angles would take more than 100 seconds.

**Note:** Where pseudocode is provided, it is written in a generic form and does not contain any language/library specific syntax. Moreover, the provided .vtk files were converted to labelmaps using the “Model to Label Map” module.

### Avoidance of a critical structure

We transform the input node to an IKJ matrix and then loop over each target image. To decide if a target point is within our target area, we iterate over all the target points and retrieve the pixel value for each one. If its value is greater than zero (or 1), it is a valid target. The function used to do this is called **getFilteredTargets(targets, area)** and should accept any area we want to filter for.

Time complexity:

Pseudocode:

For each point in target points:

Get the coordinates (x, y, z) of the point

Retrieve the pixel value of (x, y, z) in the image

If pixel value == 1:

Add the point to the list

### Placement of the tool into a target structure

We first create an oriented bounding box tree (OBBTree) of the ventricles. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we reject the path. The function used to do this is called **getTrajectoriesAvoidingArea(entriesAndTargets, area)** and should accept any area we want to avoid. It is important to note that this function uses **isPassThroughArea(tree, entry, target)** which is where the actual intersection check is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Time complexity:

Pseudocode:

For each entry point in entry points:

For each target point in target points:

Get the line between the two points

Get the points of the line

validLine = true

For each point on the line:

If point passes through ventricle:

validLine = false

break

if validLine:

add (entry, target) point to valid points list

### Trajectory is below a certain length

We first create an oriented bounding box tree (OBBTree) of the blood vessels. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we reject the path. The function used to do this is called **getTrajectoriesAvoidingArea(entriesAndTargets, area)** and should accept any area we want to avoid. It is important to note that this function uses **isPassThroughArea(tree, entry, target)** which is where the actual intersection check is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Time complexity:

Pseudocode:

For each entry point in entry points:

For each target point in target points:

Get the line between the two points

Get the points of the line

validLine = true

For each point on the line:

If point passes through blood vessel:

validLine = false

break

if validLine:

add (entry, target) point to valid points list

### Maximizing distance from critical structures

We first create an oriented bounding box tree (OBBTree) of the cortex. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we create a line perpendicular to the intersection. We then create two vectors, one for our entry/target pair and one for the intersecting points and calculate the angle between the two. If the angle is below the specified limit (55), we accept the path. The function used to do this is called

**getTrajectoriesWithSpecifiedAngle (entriesAndTargets, area, specifiedAngle)** and should accept any area we want check the angles for. It is important to note that this function uses **isValidAngle()** which is where the actual check for the angle is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Time complexity:

Pseudocode:

For each entry point in entry points:

For each target point in target points:

Get the line, lineET, between the two points

For each point on line:

If point connects with the cortex:

Get perpendicular line where lineET passes through the cortex

Calculate the angle between the two lines

If angle < 55:

Add (entry, target) point to valid points list

## OpenIGTLink

In order to send messages between the two we use the OpenIGTLink protocol. OpenIGTLink defines the message format that is used to transfer data between Slicer and ROS. The message consists of a header, an extended header, the content and some meta data. Furthermore, we need to define an importer in order to interpret the above message format. In order to achieve two-way communication, we an define an exporter that sends the current location of the end effector back to Slicer. Finally, we define a Calibrator in order to correctly transform the points between Slicer and ROS.

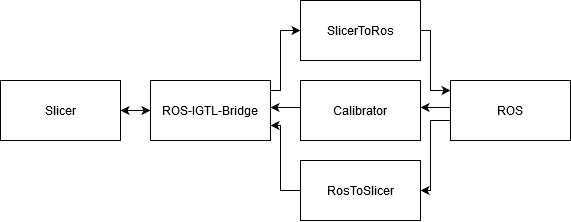


Figure 1: Connection between Slicer and ROS

## Data transfer from 3D Slicer to ROS: SlicerToRos.py

The data transfer from 3D Slicer to ROS works using the slicerToRos.py file. It takes either an entry point (named ‘Entry’) or a target point (named ‘Target’) and works as follows:

* If the point is the entry point, it simply moves to the point
* If the point is the target point, it moves the robot to the target point by creating a line/trajectory between the two. This is done by breaking said line to smaller points and forming a cartesian path. The reason it is implemented this way (the straight line) is because we are moving the robot’s end effector (i.e. a needle) in a straight line through the brain

Furthermore, it converts points from millimetres (Slicer’s configuration) to meters (ROS’ configuration). Finally, the kinematics are solved and executed through the use of the move\_it framework.

## Data transfer from ROS to 3D Slicer: RosToSlicer.py

This is a rather simpler implementation. The exporter reads the robot’s current pose in real time, using the move\_it package and sends the coordinates of the end effector to Slicer. This is mainly done for validation/sanity checks.   
Furthermore, similarly to the importer, it converts the points from meters (ROS’s configuration) to millimetres (Slicer’s configuration)

## ROS (to move the robot)

For simplicity, we use the robot provided by move\_it package in “demo.launch”. It provides:

* As mentioned above, a RobotModel that is built for ROS and visualised in RVIZ
* A kinematics model for the robot as well as various kinematic solvers to automate the calculation of kinematics based on the used RobotModel

The core classes of this are RobotModel and RobotState.

The RobotModel contains relations between all links and joints including their limits (collision, safety limits, etc), as defined by the URDF and SDRF files.

The RobotState contains information about the robot at any point in time. It is used to obtain kinematic information about the robot depending on its current state.

## Calibration (to translate points between ROS and Slicer)

The calibration (and therefore the transformation) occurs after we have calculated the entry and target points using the Path Planner module. This is done using the following steps:

1. Create 8 markups/points around the provided models (i.e. the critical structures and the cortex, forming a bounding box) in Slicer.
2. Create 8 points that form a bounding box in ROS, using the same measurements as the bounding box we created in Slicer (note: These are correctly hard coded and might have to be changed for completely different configurations).
3. Then we run the Calibration.py script, which prompts the user to save each calibration point in Slicer, one by one.
4. We then go to Slicer, where we can build a linear transformation matrix using the Fiducial Registration Wizard which is found in the IGT extension/module. The Fiducial Registration Wizard maps the 8 fiducials created in Slicer with the 8 points sent by ROS through the calibrator.
5. Then, using the transformation matrix, we transform the critical structures, the cortex and the optimal entry-target pair and pass them over to ROS. In order to ensure that our nodes are transformed, we need to use the ‘harden’ functionality of Slicer.

It is worth noting that we scale the models (critical structures and so on) to match our robot’s reachable workspace. We could have instead resized our robot to match the structures, but this would have led to a less generalised solution.

# Validation

## 3D slicer path planning

For each part of the Path Planner module, following overall process was used:

1. Pick a small subset of the data
2. Visually look for a trajectory / point that is obviously valid for a task and for one that is obviously invalid for a task. For example, when filtering for targets within the hippocampus, the invalid point could be one outside the image.
3. Run the algorithm
4. Inspect the output in slicer
5. Increase the subset of data gradually

Furthermore, we use unit testing to provide automated tests as well. These can be found in PathPlanner.py and are the following:

**testLoadAllData(path):** check that data has been loaded successfully**testGetFilteredHippocampusValidTargets():** check that targets are correctly filtered down to hippocampus targets

**testGetFilteredHippocampusInvalidTargets():** check that targets are correctly rejects invalid targets

**testAvoidBloodVesselsDilateValidPath()**: check that the algorithm accepts a path that doesn’t pass through the blood vessels dilate **testAvoidBloodVesselsDilateInvalidPath()**: check that the algorithm rejects a path that passes through the blood vessels dilate

**testAvoidBloodVesselsValidPath()**: check that the algorithm accepts a path that doesn’t pass through the blood vessels **testAvoidBloodVesselsInvalidPath()**: check that the algorithm rejects a path that passes through the blood vessels

**testAngleValidPath():** check that the algorithm accepts a path that hits the cortex at the correct angle

**testAngleInvalidPath():** check that the algorithm rejects a path that hits the cortex at an incorrect angle**testCountRejectedTrajectories(True):** To count rejected trajectories and time each part. This is a slow test**testAllTogether():** Just to see if everything is able to run together (pseudo test for task 4). This is a slow test

## Slicer Scene

The images concerning the Slicer scene contain many elements.

* Green dots with blue background: The bounding box of the brain/critical structures
* Red dots: All the possible entry points
* Cyan dots: All the possible target point
* Yellow background surrounding a dot: The current position of the end-effector (as sent by ROS)
* Green dot: Best entry point
* Blue dot: Best target point

## Position Validation

### Default Position

TODO

### At Entry position

Figure 2: End effector at entry point (view 1)

Figure 3: End effector at entry point (view 2)

Here we have both a top and bottom view of the brain. As mentioned in the instructions above, we can see the yellow background around the entry point (green dot) which represents the end-effectors position. The yellow background is set only when we hit the “Send” button through the IGT extension

### At Target Position

Figure 4: End effector at target point (view 1)

Figure 5: End effector at target point (view 2)

Here we have both a top and bottom view of the brain. As mentioned in the instructions above, we can see the yellow background around the target point (blue dot) which represents the end-effectors position. The yellow background is set only when we hit the “Send” button through the IGT extension

## Ros Scene

### At default position

Figure 6: Robot at default position (RVIZ)

Here we can see our robot in its default position. The part part represents the needle/drill that will be used to enter the brain

### At Entry position

Figure 7: Robot's end effector at entry point (RVIZ)

Here the cyan dot represents the entry point where the end-effector is currently position. The robot moves here once we send the entry point to the robot.

### At Target Position

Figure 8: Robot's end effector at target point (RVIZ)Figure 8: Robot's end effector at target point (RVIZ)

Finally, we once we send the target point (the blue point within the brain), we can see that the robot correctly moves in a straight like and enters the brain at the desired point as requested.

## ROS (to move the robot)

### Input Validation

### Valid Input (moves to a point)

Figure 9: Robot's end effector at a random point (RVIZ)

This figure shows our robot moved to a random position from the default one. This is to validate that the kinematics work as intended

### Invalid Input (can’t get to the point)

Figure 10: Invalid position sent to ROS

This figure shows the error message the robot publishes if an invalid configuration is requested. If the kinematics can’t be resolved, the robot simply passes message stating so.

## Whole system

1. Use the PathPlanner to calculate the entry and target pair/trajectory
2. Initialise connection between Slicer and ROS (step by step)
3. Set the 8 points for the bounding box on Slicer
4. Set the 8 points for the bounding box in ROS
5. Send the 8 points from ROS to Slicer
6. Calculate the transformation matrix
7. Transform the critical structures, cortex and entry-target pair
8. Send critical structures and cortex to ROS
9. Send Entry and verify that it moved to entry point
10. Check that end effector is at the entry point on Slicer
11. Send Target and verify it moved to target point
12. Check that end effector is at the target point on Slicer
13. Done
14. Mention that scene is available with everything in the repository and should be ready to run

# Conclusions

## What was achieved

Through this project, we managed to correctly calculate the best trajectory that avoids critical structures based on the length, angle and distance from critical structures. This is achieved by employing various image processing techniques that optimise the traversal between points within large data structures using packages such as VTK, numpy and Slicer. Furthermore, we describe and set up a connection between Slicer 3D and ROS in order to perform the operation. Using the move\_it package, we successfully command a robot and transfer points and information between Slicer and ROS. We’ve also learned how to create a custom RobotModel even though we use the one provided by move\_it for simplicity.

## Future improvements

While the overall system performs the desired task, there is still a lot of room for improvement in various parts of the pipeline.

### Path planner module

The path planner module seems to work well. Still, we could have further optimised our code in order to speed up the process. Another point worth mentioned is the way we calculate the best trajectory. Here we weigh all critical structures as equal. Perhaps here we could have used a weighting system to better calculate the distance from multiple structures instead. Furthermore, after calculating the optimal entry-target, we could have automatically sent it to ROS but perhaps this might be risky if this was a real-case scenario.

### RobotModel

Here we could have designed a simpler robot whose sole purpose was to perform brain surgery (which includes a needle for example) instead of using a more generalised robot

### Calibration

An obvious improvement here is an automated bounding box calculator as we are currently choosing the points manually.

### Importer & Exporter

These are mostly fine. Here we could have split the code in more files/classes in order to make some parts reusable and separate some of the robot’s repositioning logic and IGTL logic.

### Slicer

Here we could have created a model (i.e. an .stl file) of the robot that can be loaded in Slicer in order to better visualise how the robot moves during the operation instead of simply looking at where the end effector goes.

### Overall

As mentioned above, while everything works and the pipeline is complete, we could have automated some of the manual parts (such as the calibration and sending points to ROS). Although, perhaps it is better this way as if it was a real case scenario, we would have to oversee each step individually in order to make sure everything is done safely.

# Code / Git Repository

https://github.com/Meldanen/kcl/tree/master/robotics/final

# References

[1] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, “Autoencoding beyond pixels using a learned similarity metric,” *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 4, pp. 2341–2349, 2016.

# Appendix

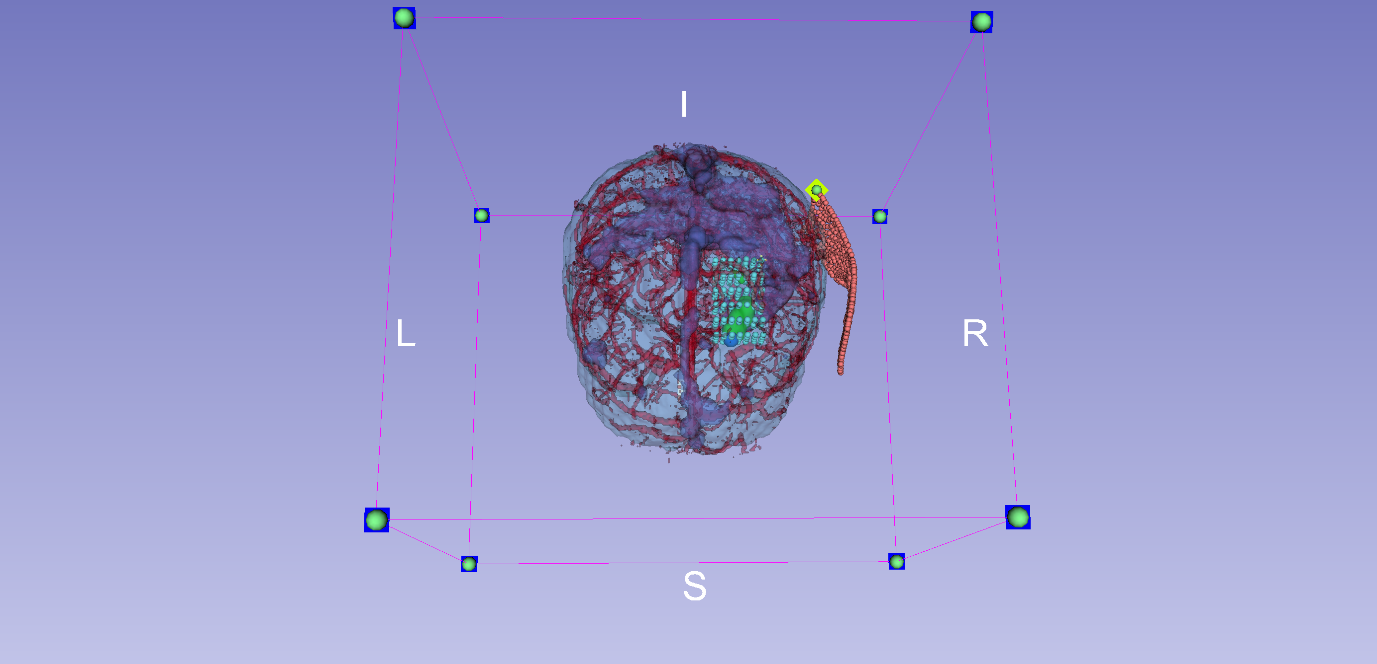


Figure : End effector at entry point (view 1)

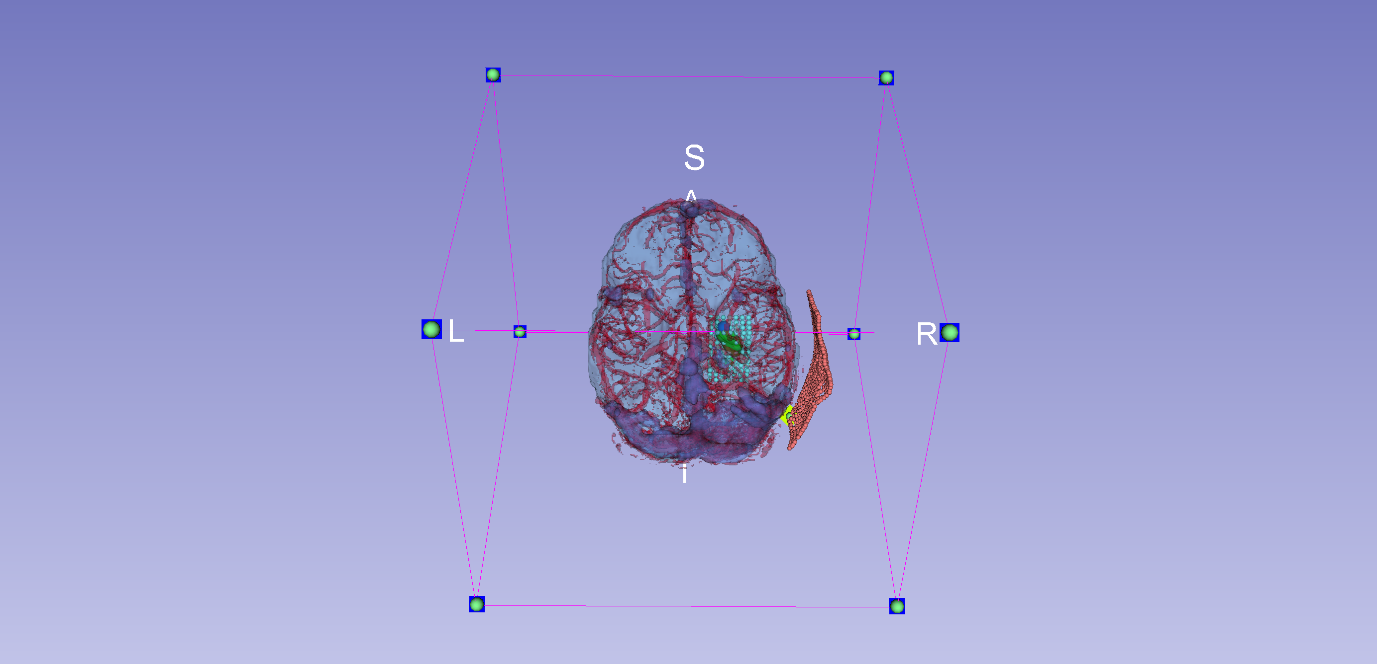


Figure : End effector at entry point (view 2)

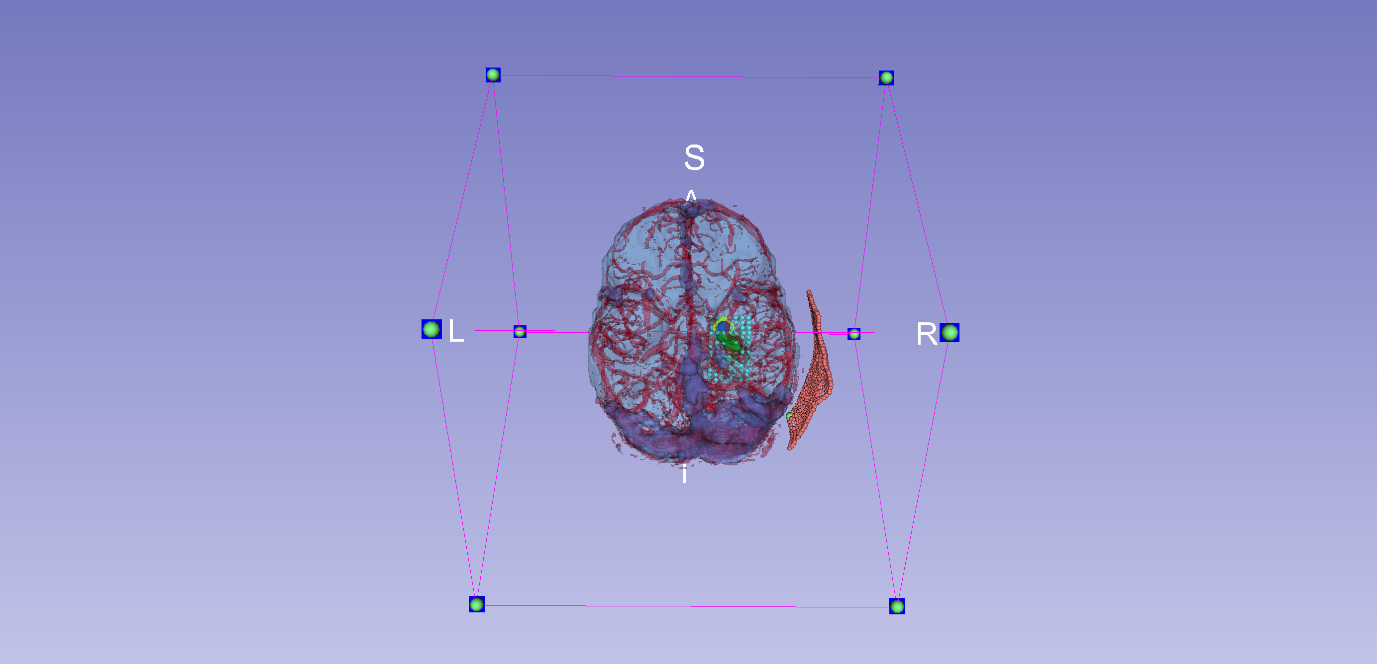


Figure : End effector at target point (view 1)

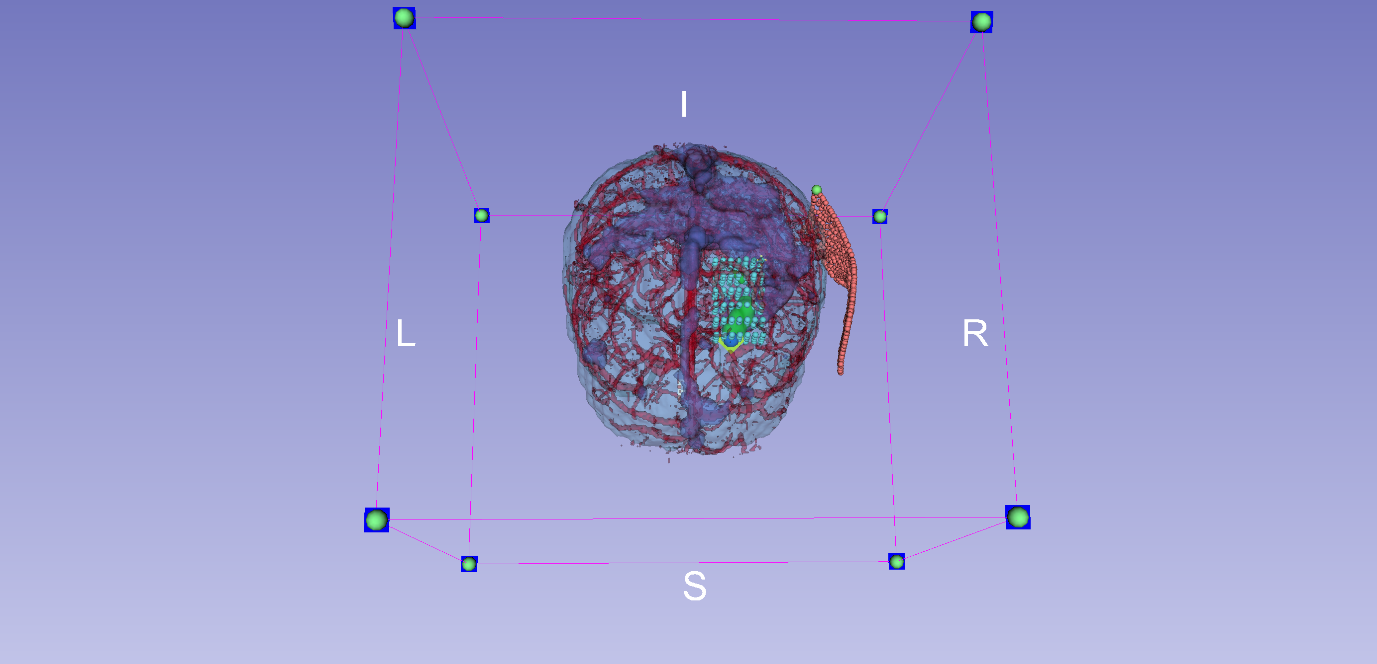


Figure : End effector at target point (view 2)

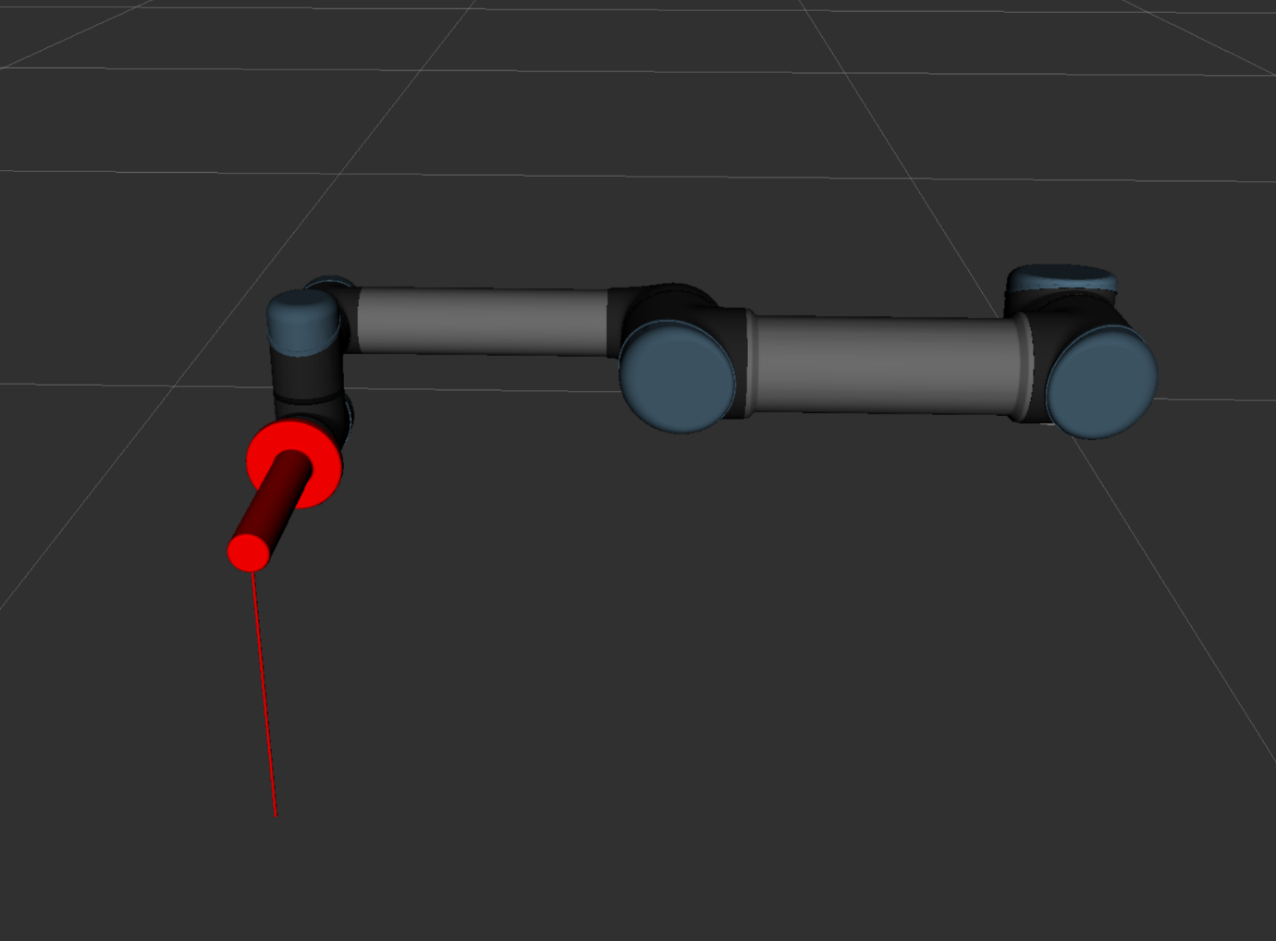


Figure : Robot at default position (RVIZ)

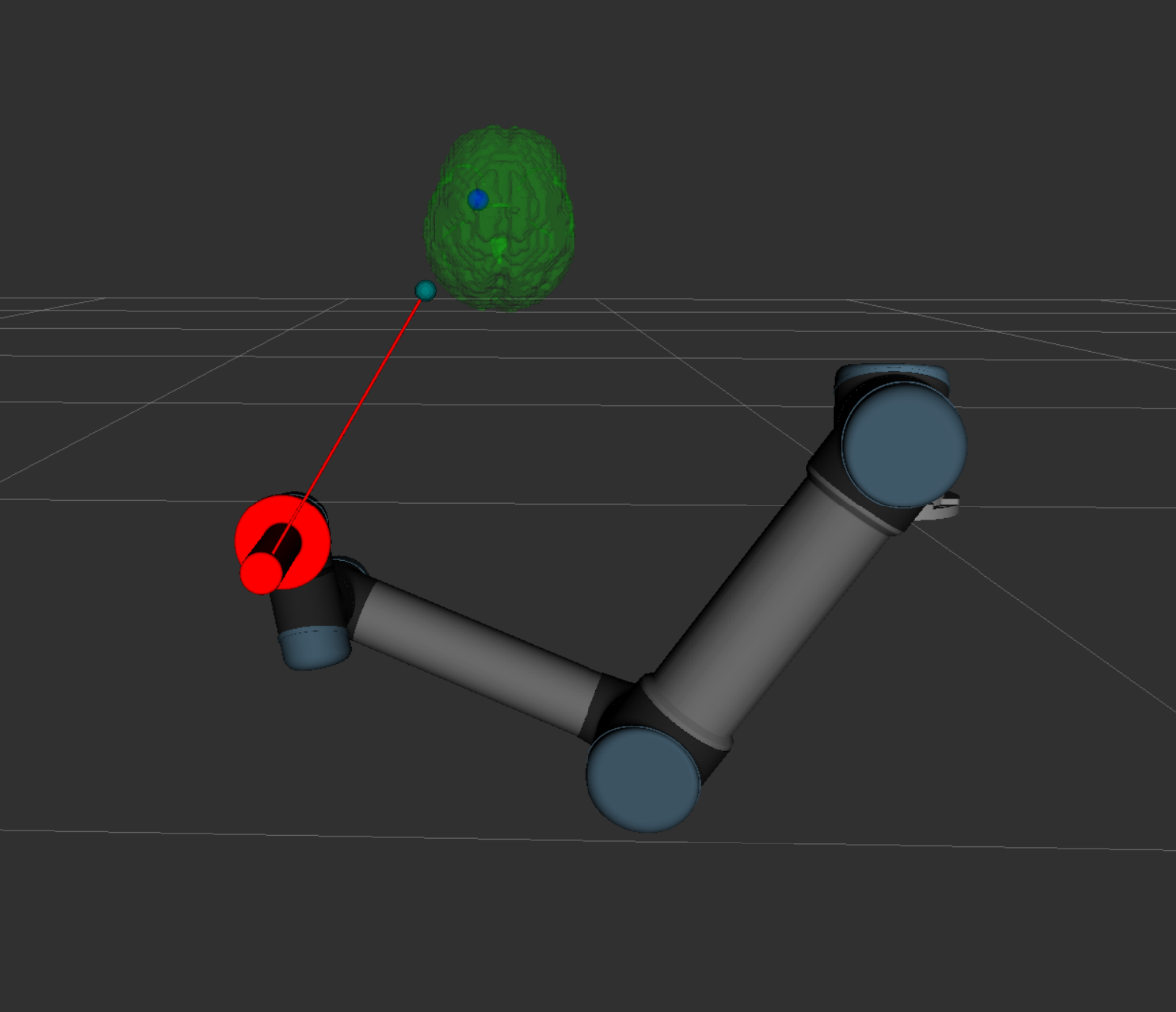


Figure : Robot's end effector at entry point (RVIZ)

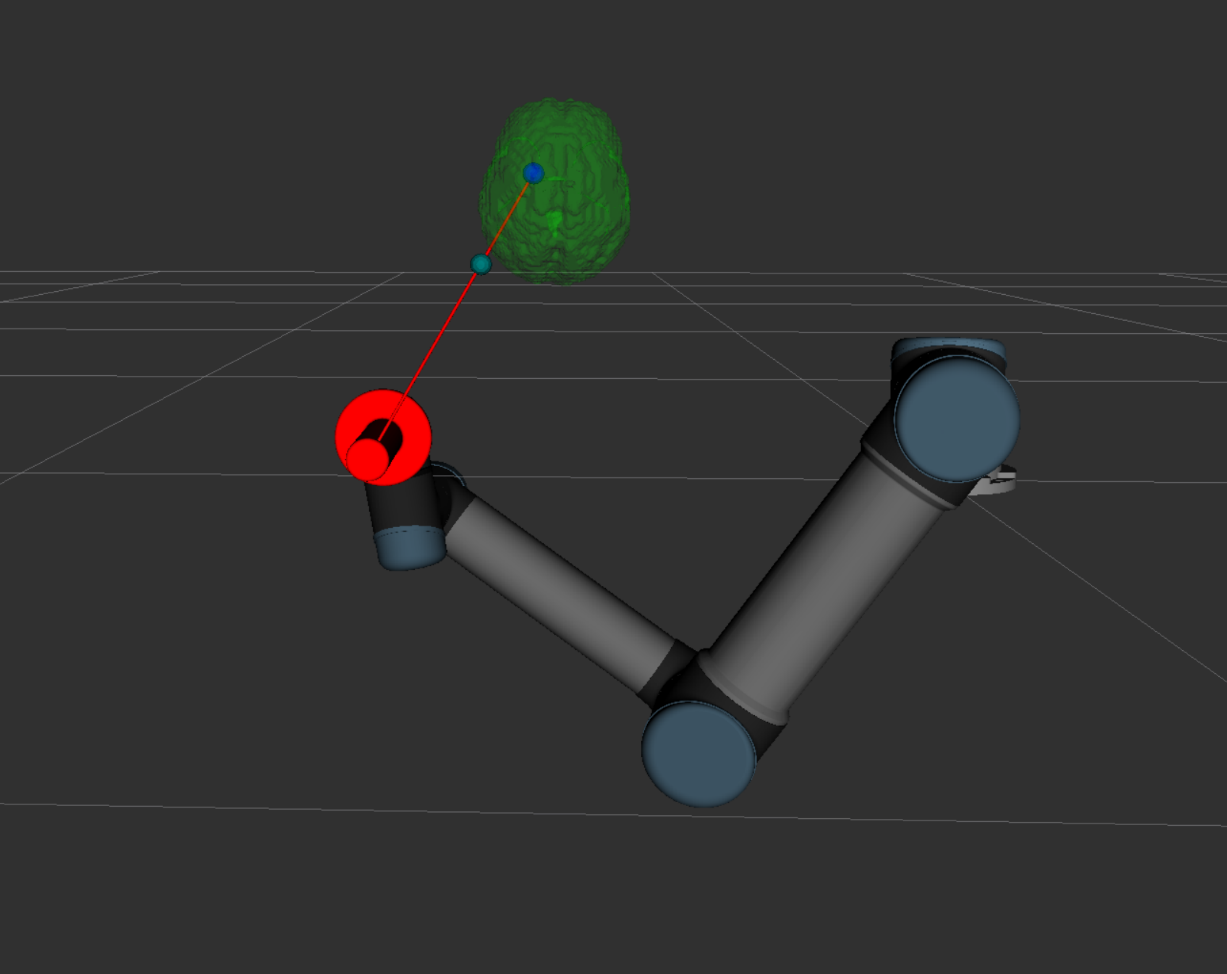


Figure : Robot's end effector at target point (RVIZ)

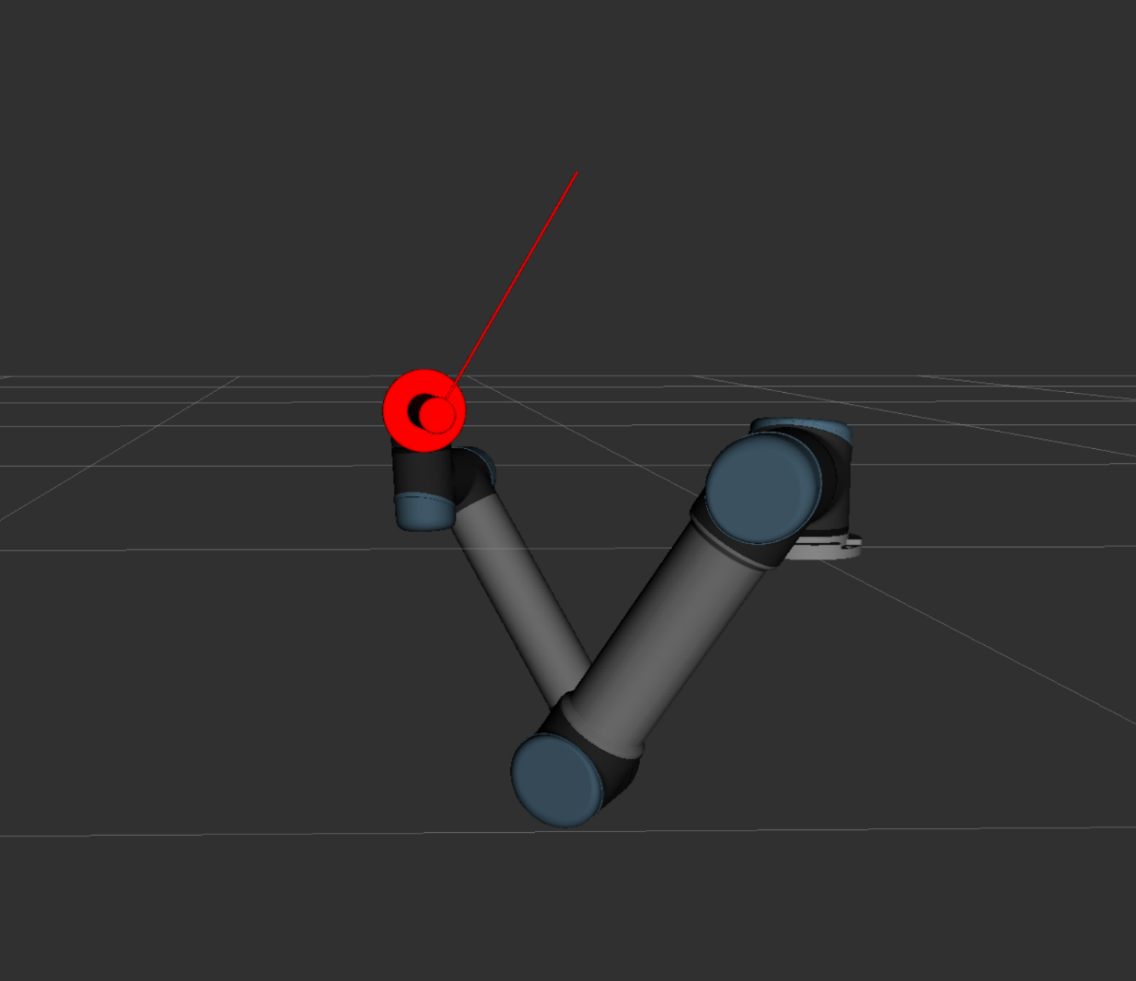


Figure : Robot's end effector at a random point (RVIZ)

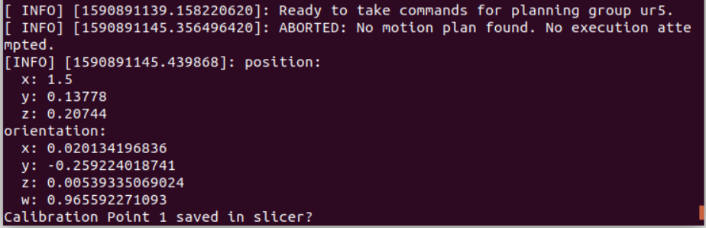


Figure : Invalid position sent to ROS

1. The risk resulting from information and capabilities not being shared among/by enough team members either because of lack of planning or long training times [↑](#footnote-ref-1)