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.

### 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 (data transfer from 3D Slicer to ROS)

TODO

## OpenIGTLink (data transfer from ROS to 3D Slicer)

TODO

## ROS (to move the robot)

TODO

## Whole system

TODO

# Validation

## 3D slicer path planning

TODO

## OpenIGTLink (data transfer from 3D Slicer to ROS)

TODO

## OpenIGTLink (data transfer from ROS to 3D Slicer)

TODO

## ROS (to move the robot)

TODO

## Whole system

TODO

# Conclusions

TODO

# 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

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)