Summer Industrial Immersion Report

Smetankin Sergei

Contents

1 Main goal

NEUROROBOT is robot-assisted TMS navigation system for the brain for neurorehabilitation and pre-radiation preparation of patients. During the procedure, the robot is surrounded by additional equipment: chairs, monitors, TMS devices, etc. All of the above items change their location or state over time depending on the convenience of the doctor and the patient. It is equally important to note that the equipment is expensive, so it is necessary to reduce the risk of damage to it. The main objective was to implement a collision avoidance system into the project, which would help in real time in telecontrol mode to keep the equipment intact and ensure patient safety in the event of possible operator errors (prevent the robot from colliding with objects when controlled by a joystick).

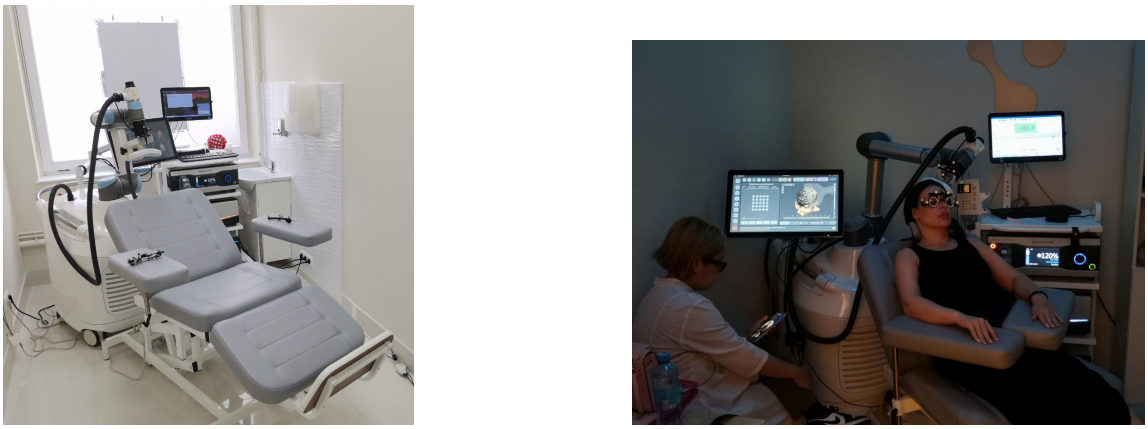


Figure 1: NeuroBot – Robot-assisted navigation device for TMS of the brain in neurorehabilitation

2 Workflow

2.1 The original plan and its modifications

Initially, it was planned to use an algorithm based on the *Artificial Potential Field*, i.e. to add a repulsive force to the end-effector when approaching objects that the robot should not collide with. But it turned out that in this case, the robot would be affected by unwanted accelerations that prevented the robot from being positioned correctly. Therefore, it was decided to adapt the "*Threshold distance method*", implemented in the default *moveit_servo* from the *Moveit* framework and based on the *FCL* library, to our needs.

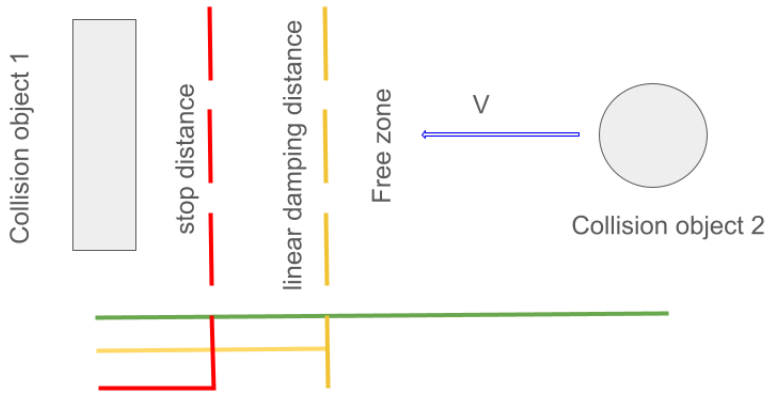


Figure 2: Threshold distance method

The basic idea of the "threshold distance method" is that there are three zones of motion of objects, depending on the distance between them. In the first zone (green), objects approach each other without changing their speed, in the second (yellow), the vector of their approach speed is multiplied by a certain scale coefficient, linearly dependent on the distance, and in the third (red), the motion stops and the approach speed becomes zero, which prevents objects from colliding.

2.2 Problem 1.

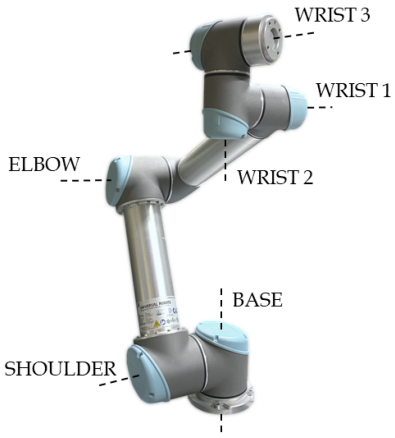
Description: When controlling the joystick, strange behavior of the robot was noticed, namely: it was not always possible to avoid collisions.

Cause of the problem: After a long search for the cause, it was suggested that the algorithm was slow. After that, a widget was written on *Qt* to measure the time of action of the *FCL* library and it was found that the time spent on the necessary calculations to avoid collisions is more than necessary for control at the frequency of 125Hz, embedded in the robot. (I was told about the default 125Hz by the project engineers).

Off-topic:

How collision avoidance algorithms work in general

- Initially, AllowedCollisionMatrix is built -- a collision matrix that tells which objects can potentially collide and which cannot. This is intended to reduce computational costs.
- Different algorithms calculate the distances between objects, in accordance with the collision matrix.
- Then, the minimum distance is selected from these distances and, depending on it, we can judge whether we have collisions or not.



		0	1	2	3	4	5	6	7	8	9	10	11
base_link_inertia	0	-	-	-	-	-	-	1	1	-	-	-	-
coil_link	1	-	-	1	-	1	1	-	-	-	1	1	1
flange_link	2	-	1	-	-	1	1	-	-	-	1	1	1
forearm_link	3	-	-	-	-	-	-	-	-	1	1	-	-
ndi_marker_link	4	-	1	1	-	-	1	-	-	-	1	1	1
sensor_body	5	-	1	1	-	1	-	-	-	-	1	1	1
shoulder_link	6	1	-	-	-	-	-	-	1	1	-	-	-
tms_base	7	1	-	-	-	-	-	1	-	-	-	-	-
upper_arm_link	8	-	-	-	1	-	-	1	-	-	-	-	-
wrist_1_link	9	-	1	1	1	1	1	-	-	-	-	1	1
wrist_2_link	10	-	1	1	-	1	1	-	-	-	1	-	1
wrist_3_link	11	-	1	1	-	1	1	-	-	-	1	1	-

Figure 3: Robot and collision matrix example

Solution: It was necessary to somehow speed up the calculations of distances necessary to avoid collisions. After searching, two ways were proposed:

1. Implementing the *Bullet* Library
2. Offloading Computations to NVIDIA GPUs with *cuRobo: CUDA Accelerated Robot Library*

(spoiler: the first method worked, so there was no need to implement the second one)

3 Implementing the *Bullet* Library

After modifying the original moveit_servo, I was able to make the new collision avoidance method via OOP C++ easy to integrate into the project. I implemented the *Bullet* library, already modified for ROS, and measured the speed of calculations. The speed gain was **×30**, which allowed me to fit within the 125 Hz refresh rate.

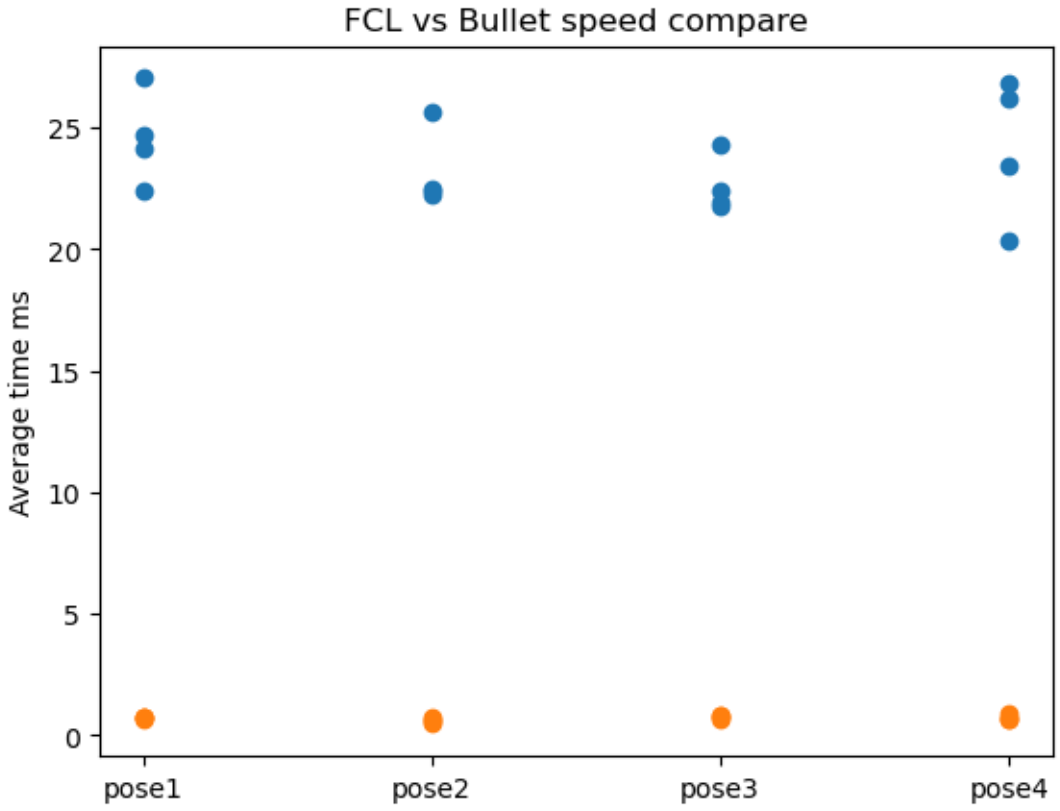


Figure 4: Comparison of the time it takes to calculate distances between objects in different robot positions for FCL (blue dots) and Bullet (orange).

3.1 Problem 2

Description: In some robot positions, the FCL and Bullet libraries gave different minimum distances at the output, which were necessary to avoid collisions. There was also an obvious error in the Bullet library - the zero distance was incorrect, and a significant gap between objects was visible in the simulation.

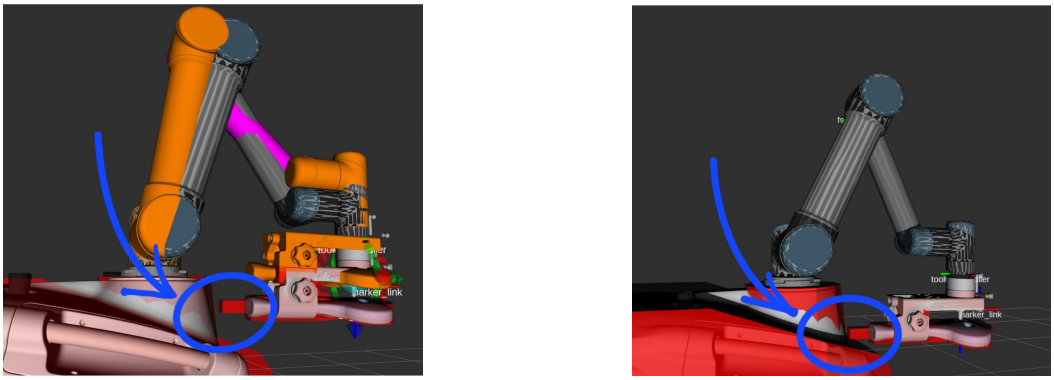


Figure 5: Position of the robot when the distance showed "0". We can see a gap for bullet (right picture)

Cause of the problem: As it turns out, *Bullet* doesn't work well with non-convex objects

Solution: It was necessary to modify the existing urdf files and collision matrix

4 Summary

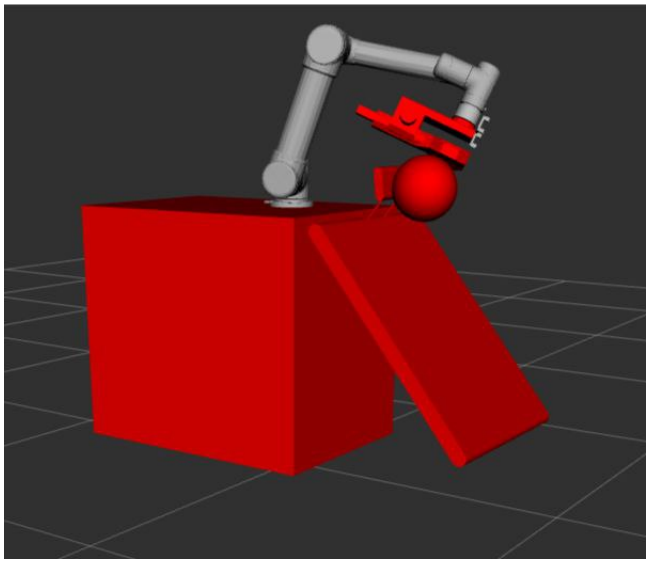


Figure 6: Final version of model with approximation of chair and head (collision shapes)

During the practice, an algorithm for avoiding collisions in the joystick control mode was implemented in the neurobot project. An optimal method was found that corresponded to the initial

task of correct positioning of the TMS device, and the problems that arose were consistently solved. This work will be included in subsequent software updates on the real robot.