Robotics Internship
Report 2

Working with Franka Panda and the Vicon System

VRIJE
UNIVERSITEIT
BRUSSEL

Roulios Charalampos
VUB Intern
Studies in NTUA

Contents

Chapter 1

Vicon

Vicon is a developer of motion capture products and services for the life science, entertainment, and engineering industries. Motion capture (mocap) is the process of recording the movement of objects or people.

The Vicon motion capture system is a passive optical system. This technique uses retroreflective markers that are tracked by the infrared cameras by reflecting the light that is generated near the cameras' lens. The cameras emit the infrared light towards the direction they are facing, the markers reflect the light and by receiving the reflected light, the cameras can determine the markers' position coordinates relative to a fixed frame. The camera's threshold can be adjusted so only the bright reflective markers will be sampled, ignoring skin and fabric.

1.1 Set-up of Franka Panda

The basic set-up of Panda research looks like this:

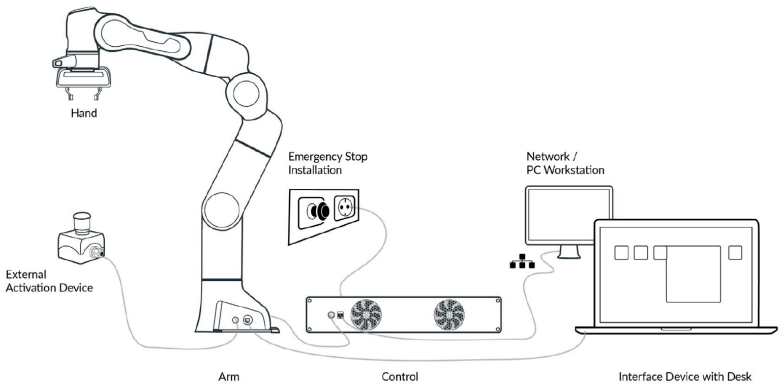


Figure 1.1: Connection scheme to control Franka Panda using Desk or FCI.

- The Arm is connected to the Control via a connection cable.

- Between the Control and the power supply there is an emergency off installation that will cut the power to Panda research in case of emergency. (not at R&MM lab)
- At the base of the Arm an external activation device is connected, in order to deliberately activate movements of the Arm.
- When using Desk: an interface device with web browser is connected to the base of the Arm via an Ethernet cable.
- When using FCI: a PC workstation (i.e. a network Ethernet interface) is connected to the Control via an Ethernet cable.

Since the PC workstation (the Probo desktop) we use to control Franka Panda via FCI has only one Ethernet port, we had to create a local wifi network called adhocname, so that the computer that controls the robot can receive information from the Vicon desktop that receives the markers' positions.
To receive this information, we also have to make a bridge between Vicon and Ros, and therefore we also have to install the Ros package for Vicon (Vicon_bridge) and afterwards include the Vicon C++ library in our program. The steps we needed to follow in order to read Vicon data in a C++ program are the following:

- Clone and make the Vicon_bridge package in catkin_ws, from this github repository: https://github.com/ethz-asl/vicon_bridge, https://github.com/ethz-asl/vicon_bridge
- Include the Vicon header file in the .cpp program that will read the Vicon data:

```
#include "vicon_bridge/Markers.h"
```

- Open the CMakeLists.txt file of the directory in which the source code of the program is located and add *vicon_bridge* in *catkin REQUIRED COMPONENTS*:

```
13 find_package(catkin REQUIRED COMPONENTS
14   roscpp
15   rospy
16   std_msgs
17   message_generation
18   vicon_bridge
19 )
```

Figure 1.2: Adding vicon_bridge in the CMakeLists.txt

- Now you are ready to build the source file.

1.2 Create adhoc wifi network on Vicon computer

To create this adhoc wifi network, go through the following steps.

1. Turn on the Vicon computer on which the Nexus software is installed.
2. Turn on the synch box.

Figure 1.3: Vicon synch box

3. To start the *adhocname* wifi from the Vicon computer, you have to press "Windows+X", click on the *Windows Powershell (Admin)*, and type the following command:

   ```
   netsh wlan start hostednetwork
   ```

   as visualized in Fig. 1.4. You can also press the "up arrow" key till you find it.

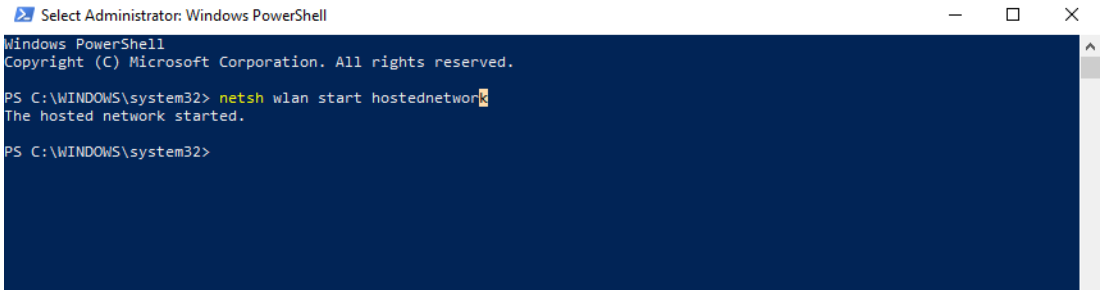


Figure 1.4: Activating *adhocname* wifi

1.3 Setup the overall system

First of all, we have to calibrate the Vicon system.

1. Double click on the Vicon Nexus Software.
2. Mask the cameras and reflective objects in the room that may not be taken into account as reflection markers. Moreover, masking a camera will identify all the other sources-reflection of infrared light including the light emitted from the other cameras.

As a result, before you mask the cameras you need to make sure that no reflecting markers are visible from their perspective.

3. Calibrate cameras. To do so, first of all you need to turn on the wand by toggling its on/off switch:

Figure 1.5: Vicon wand

To start calibrating the cameras click "start" at the "Calibrate Cameras" panel and then start waving the wand towards the cameras. While calibrating, try to wave the wand mostly in an area which all cameras can see, since this would help the cameras to identify their relevant position and minimize the measurement error.

4. Set the volume origin (you can find more information for the procedure, specifically for the franka robot, in section 5.2 of Louis' report).

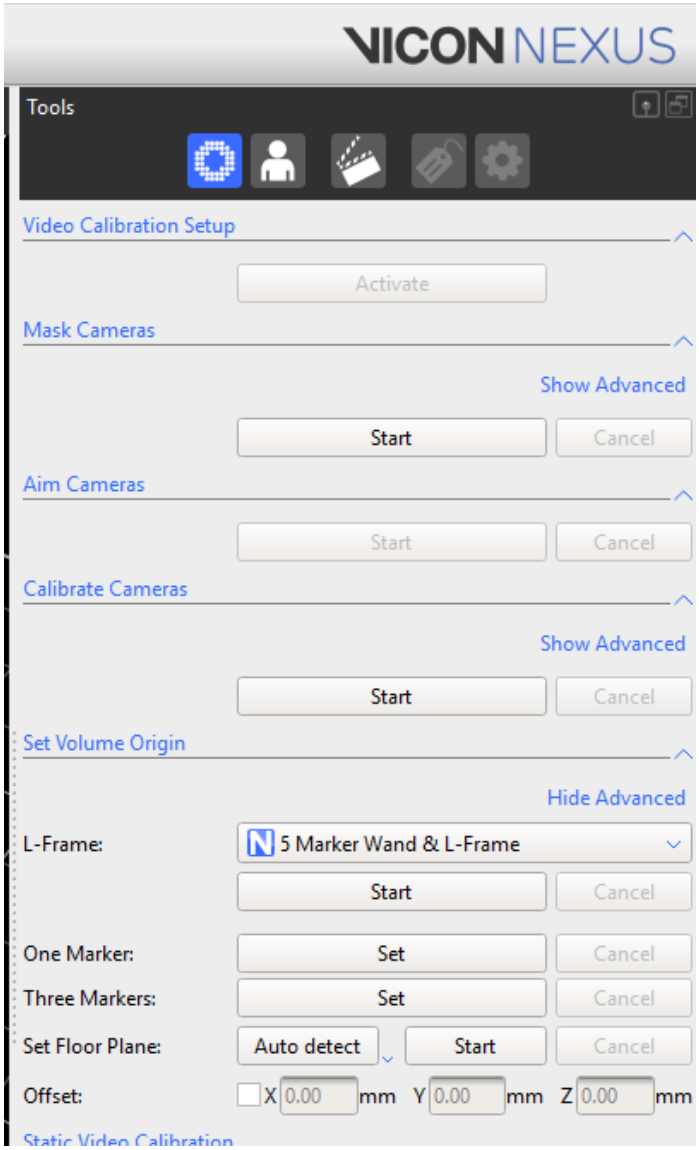


Figure 1.6: Masking and calibrating cameras, and setting the volume origin

5. Select the right session and subjects according to the experiment you want to execute. More info for specific experiments later in the document.

6. Make sure the status of the Nexus software is live and not offline.

Then we have to make sure the robot can receive all the information coming from the Vicon system.

1. Go to the desktop with which we control the Franka Panda robot and select *adhocname* from the wifi network list.

2. Open a terminal and type:

```
roslaunch vicon_bridge vicon.launch
```

Now the computer should receive data from the Vicon system.

3. To check if the stream of data is properly received, you can open another terminal and type:

```
rostopic echo /vicon/markers
```

You should see a real time print of all the visible-for-the-camera markers' positions.

Normally you will receive information of the Vicon system at a rate of $100Hz$.

Chapter 2

Teleoperation

The goal of the teleoperation experiment is to control the robot arm with your own arm, i.e. teleoperate the robot arm. For this purpose, we created a subject (a pattern) of 5 markers that are placed on the arm. As can be seen in Fig. 2.1, the markers are positioned on the elbow, the wrist, the hand, the thumb, and the index.

Figure 2.1: Markers positioned on the arm. From top to bottom: elbow, wrist, hand, thumb, and index.

The program has four functionalities.

- The end-effector has to move in the same direction as the wrist does, see Fig. 2.2.
- The gripper can be opened and closed by moving the thumb and index away from each other or to each other, see Fig. 2.3.
- The 6th joint can be controlled by moving the wrist up and down, see Fig. 2.4.
- The 7th joint can be controlled by spinning the thumb and index, see Fig. 2.5.

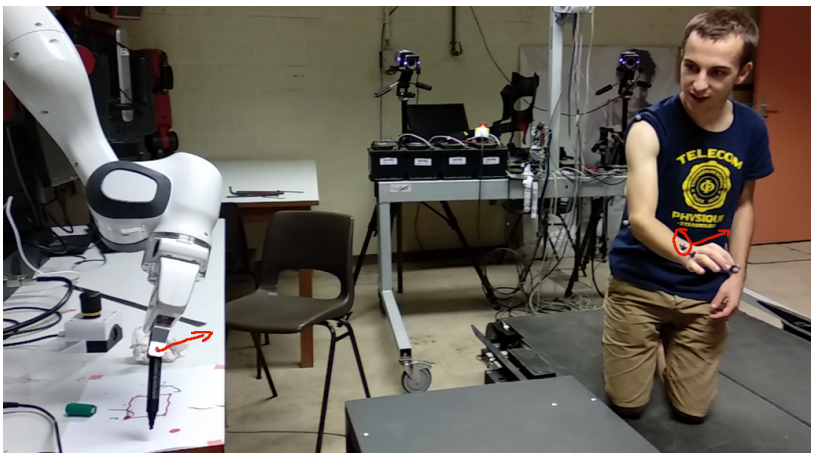
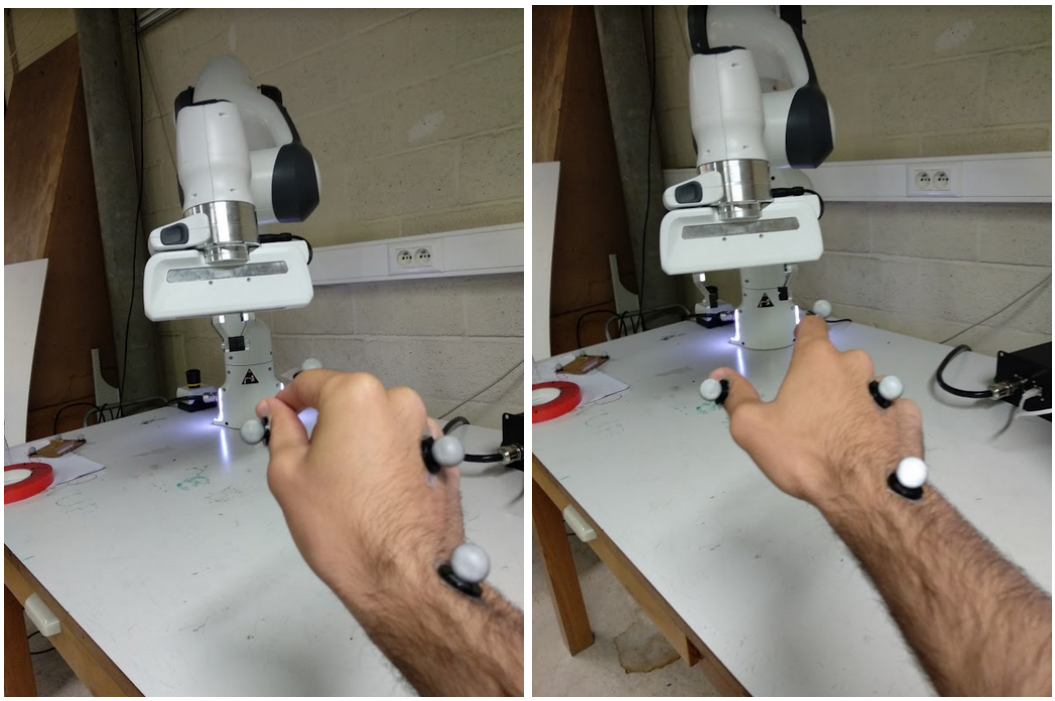


Figure 2.2: The end-effector follows the translational movement of the wrist.



(a) Closed gripper **(b)** Open gripper

Figure 2.3: The gripper opens and closes when the thumb and index are moving away from eachother or to eachother respectively.

(a) Wrist down

(b) Wrist up

Figure 2.4: The 6th joint is controlled by moving the wrist up and down.

Figure 2.5: The 7th joint is controlled by spinning the thumb and index.

2.1 Setting up the experiment

To be able to properly run this experiment using the Vicon system you will need to follow the next steps:

1. Make sure the robot is turned on and that you have setup the Vicon system as described in Chapter 1.

2. Select New_session1 as the current session in the Nexus software in the Vicon computer and make sure only *MyArm* is checked in the subject list, as visualized in Fig. 2.6.

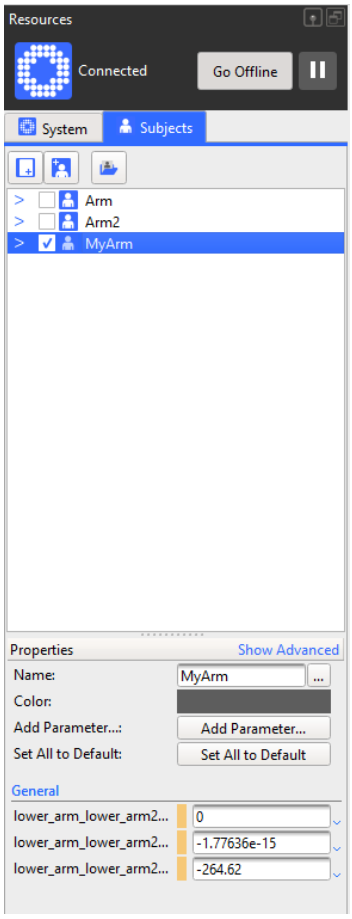


Figure 2.6: Selecting the appropriate session and subjects.

3. Place the markers on your arm at the positions shown in Fig. 2.1. Note that for different people, different "arm subjects" need to be made, since the distance between the markers is important for Vicon to recognize the subject. If the markers' placement is correct, then your arm should appear like in Fig. 2.7 in the Nexus software.

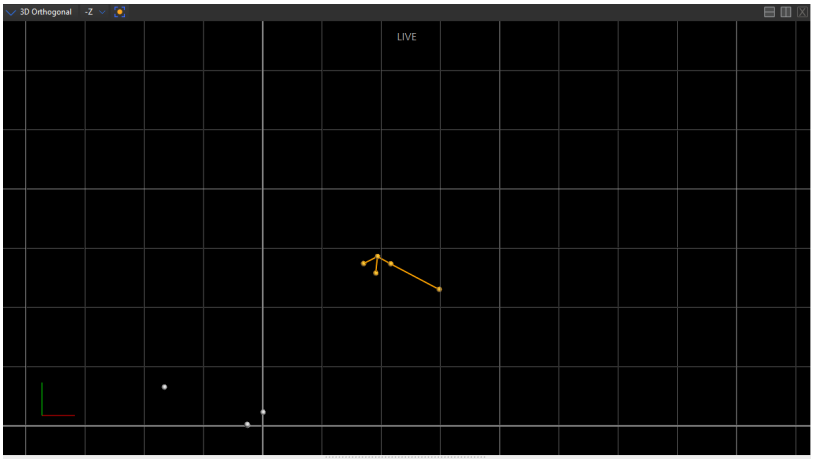


Figure 2.7: Arm subject in Nexus

4. You are ready to run the program, called *Vicon_Teleoperation.cpp*, located in:

   `/home/franka/libfranka/build/examples/Harry_Controllers`

   Make sure that your arm is in a position that is fully visible for the camera system before the program starts.

2.2 Working principle

2.2.1 Using ROS to receive data from Vicon

In all Vicon experiments' source code, ROS and Vicon libraries are included. At the beginning of each program, you will find this command, which creates a new subscriber node:

```
ros::init(argc, argv, "vicon_subscriber");
```

Additionally, in each Vicon related source code, you will find a function called **subscribe_target_motion** and a function called **MarkersCallback**.

subscribe_target_motion
The function purpose is to subscribe to the *vicon/markers* topic where the data are streamed and call the *MarkersCallback* function, each time a new ROS message has been published to the subscriber.
This function is the same in every program and looks like this:

```
void subscribe_target_motion() {
  // subscription stuff
  ros::NodeHandle n ;

  sub = n.subscribe("vicon/markers",10,MarkersCallback);
```

```
  ros::Rate loop_rate(2000);  // 2 kHz
  while (ros::ok()) {
    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

MarkersCallback

This function is used to do some calculations and store values to some global variables each time a new message arrives.
For example, the *Vicon_pick_and_place* program contains a *MarkersCallback* function that stores the received XYZ position of one marker in the **pd** vector after changing the units from milimeters to meters (since vicon publishes the markers' position in mm). This function also closes or opens the gripper, depending on the variables *stop* and *release* which values are altered within the control loop.

```
void MarkersCallback(const vicon_bridge::Markers::ConstPtr& msg)
{
  x = -msg->markers[marker].translation.x;
  y = -msg->markers[marker].translation.y;
  z = msg->markers[marker].translation.z;

  dist = distance(x/1000,y/1000,z/1000,-offsetx,-offsety,-offsetz);

  if(x!=0 && y!=0 && z!=0 && dist < max_dist){
    pd(0) = x/1000 + offsetx;
    pd(1) = y/1000 + offsety;
    pd(2) = z/1000 + offsetz;
  }

  if (stop)
    {
      if(!release){
          gripper->grasp(0.04, target_gripper_speed, target_gripper_force, 1, 1);
          stop = false;
        }
      else {
        gripper->move(target_gripper_width, target_gripper_speed);
        stop = false;
      }
      release = !release;
    }
}
```

Important: The *subscribe_target_motion* function (and as a result the callback function as well) needs to run in parallel with the control loop, since it practically is an infinite while loop that waits for new messages. To run this function separately, we need to run it in another thread. To do so we need to include the *thread* library and add the following command in the start of the source code:

```
static std::thread t1(subscribe_target_motion);
```

2.2.2 Control the translational movement of the end-effector

The desired Cartesian position is given by the wrist (marker) relative to the initial position of the robot. The position of the wrist is updated at a rate of 100 Hz. The goal is to let the robot end-effector reach this position with a minimal error by giving torques to the system.

Defining at the k$^{\text{th}}$ sample time the desired Cartesian position as $\mathbf{p_{d_k}}$, the current Cartesian position as $\mathbf{p_k}$, the desired Cartesian end-effector velocity as $\dot{\mathbf{p}}_{\mathbf{d_k}}$, the acceleration as $\ddot{\mathbf{p}}_{\mathbf{d_k}}$, the Jacobian matrix as $\mathbf{J}$, and the pseudo-inverse of the Jacobian matrix as $\mathbf{J}^+$, the desired joint space velocities can be calculated.

$$\mathbf{e_k} = \mathbf{p_{d_k}} - \mathbf{p_k} \tag{2.1}$$

$$\ddot{\mathbf{p}}_{\mathbf{d_k}} = K_p \mathbf{e_k} - K_d \dot{\mathbf{p}}_{\mathbf{k}} \tag{2.2}$$

$$\dot{\mathbf{p}}_{\mathbf{d_{k+1}}} = \dot{\mathbf{p}}_{\mathbf{d_k}} + \ddot{\mathbf{p}}_{\mathbf{d_k}} \Delta t \tag{2.3}$$

$$\dot{\mathbf{q}}_{\mathbf{d_k}} = \mathbf{J}^+ (\dot{\mathbf{p}}_{\mathbf{d_{k+1}}} + K_e \frac{\mathbf{e_k}}{\|\mathbf{e_k}\|}) \tag{2.4}$$

Note that the term $K_e \mathbf{e_k}$ adds speed proportional to the direction of the error (since it is divided by its norm) and is only added to minimize the error from the desired position.

Note that the use of this extra term is not mandatory, since a PD controller is already used before, but by selecting the right (a relatively small value since a higher value will cause the end-effector to shake between the desired point) K_e value, the robot will reach the reference position with very high precision. Using a higher K_p value would probably have the same result but then the whole movement from the beginning to the desired position would speed up, something that may not be wanted.

After calculating the desired joint velocities $\dot{\mathbf{q}}_\mathbf{d}$, the torque for each joint can be computed as

$$\boxed{\tau = Kt \frac{(\dot{\mathbf{q}}_\mathbf{d} - \dot{\mathbf{q}})}{\Delta t} + \tau_\mathbf{ext}} \tag{2.5}$$

The $Kt \frac{(\dot{\mathbf{q}}_\mathbf{d} - \dot{\mathbf{q}})}{\Delta t}$ is practically the joint acceleration (scaled by the K_t constant which tunes the intensity of the movement) and the $\tau_\mathbf{ext}$ term are the external torques measured by the sensors.
Compensating the external torques improves the whole movement. Without this term in Equation (2.5), the torques would be computed only looking to the individual joint velocities, not taking into account the motion of the other joints. Also, by compensating the

external torques, the robot becomes completely stiff to outer forces. If we would not compensate the external torques, then all the joints would react like springs if an external force would be applied on it and also lifting an object would result to a weird behaviour since its gravity would apply external force to the robot.

The controller is be visualized in the following block diagram.

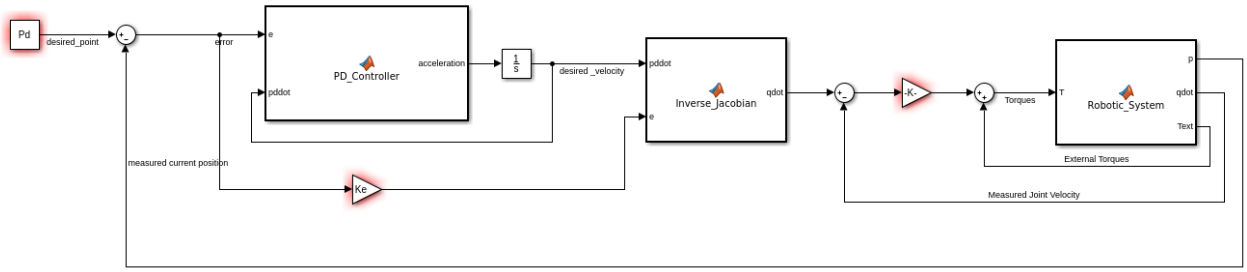


Figure 2.8: Block diagram of the feedback loop

The code that is responsible for the previous calculations is shown below:

```
for(i = 0; i<3; i++) err(i) = pd(i) - current_pose[12+i];

for(i = 0; i<3; i++) acceleration[i] = Kp*err(i) - Kd*prev_vel(i);

for(i = 0; i<3; i++) new_vel(i) = prev_vel(i) + acceleration[i] * period.toSec();

Jac = createJac(model.zeroJacobian(franka::Frame::kEndEffector, robot_state));

JacPlus = pinv(Jac); //pseudoinverse

qdot = JacPlus * (new_vel + Kerr * err);
current_q = robot_state.q;
current_qdot = robot_state.dq;

for(i=0; i<5; i++) {
  Torque[i] =  Kt * (qdot(i)-current_qdot[i])/0.001 + robot_state.tau_ext_hat_filtered[i];
}
```

Figure 2.9: Equations in C++ code

Note that in the last for loop, only the first 5 torques are computed, since the last two joints are controlled individually by the movement of the users hand. Furthermore, *Armadillo* C++ library is used for the calculations that include matrices.

2.2.3 Control of the 6th and 7th joint motion

The torque equations in the code for the last two joints, are located right after the *for* loop for the previous 5 torques and are shown below:

```
Torque[5] = Kp7 * (angle6 - current_q[5]) - Kd7 * current_qdot[5];
Torque[6] = Kp7 * (angle7 - current_q[6]) - Kd7 * current_qdot[6];
```

As you can see, a classic PD control is used for the last two joints, where *current_q[i]* is the currently measured joint angle and **angle6** and **angle7** are the desired joint angles which

are calculated in the *MarkersCallback* function each time a new Vicon message arrives.

angle6

For the *angle6* desired angle, the euclidean distance between the wrist and hand marker, visualized in Fig. 2.10, is calculated. This distance is scaled to an angle with the following equation:

```
angle6 = 0.065 * (80-dist) + 3.14/1.6;
```

The value 80 of the scaling is chosen since the distance of the two markers, while the hand maintains a straight position is around $80mm$, the 0.065 value converts this distance from millimeters to rad and the $\frac{3.14}{1.6}$ angle was mostly tuned experimentally, so that when the program starts, the end effector would have an orientation vertical to the ground. As a result, moving your wrist upwards, will decrease the distance between those two markers and also change the angle6 reference variable.

Figure 2.10: Distance from wrist to hand used to calculate angle6.

angle7

The *angle7* desired angle is defined as the angle between the vector (only in the XY plane) that is formed from by the thumb and index and the vector that is formed by the elbow and wrist, multiplied by a scale factor. This is depicted in Fig. 2.11.

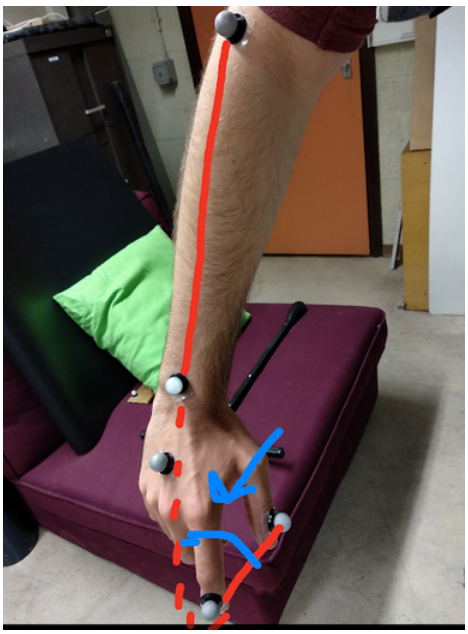


Figure 2.11: Angle that is calculated for the angle7

For the calculation of the angle between two vectors in the XY plane, the following function is used:

```
inline float angle(double Max, double May, double Mbx, double Mby,
double Mcx,double Mcy,double Mdx,double Mdy){
  Vax= Max-Mbx ;
  Vay= May-Mby ;
  Vbx= Mcx-Mdx ;
  Vby= Mcy-Mdy ;
  return atan2(Vay,Vax) - atan2(Vby,Vbx); //atan2(Vay,Vax);
}
```

Angle7 is computed in the following code, scaled by a factor of two and subtracted by a reference angle so that the last joint will stay at its starting position when the program starts and you don't move your hand:

```
angle7 = 2 * angle( -msg->markers[4].translation.x,
                                -msg->markers[4].translation.y,
                                -msg->markers[3].translation.x,
                                -msg->markers[3].translation.y,
                                -msg->markers[0].translation.x,
                                -msg->markers[0].translation.y,
                                -msg->markers[1].translation.x,
                                -msg->markers[1].translation.y) - angle7_ref;
```

Chapter 3

Obstacle Avoidance

This experiment is practically an early stage of the ERG method. The goal for the robot is to reach both a (possibly moving) desired position and orientation and simultaneously avoid (possibly moving) obstacles, but without taking into account joint angles or torque limits. The Vicon room is used here so that we can easily change the desired positions and obstacle positions, even while the robot is moving. The robot is also trying to avoid obstacles, not only close to the end-effector, but also close to the last 4 joints. You can see more clearly the program's functionality in the following figures.

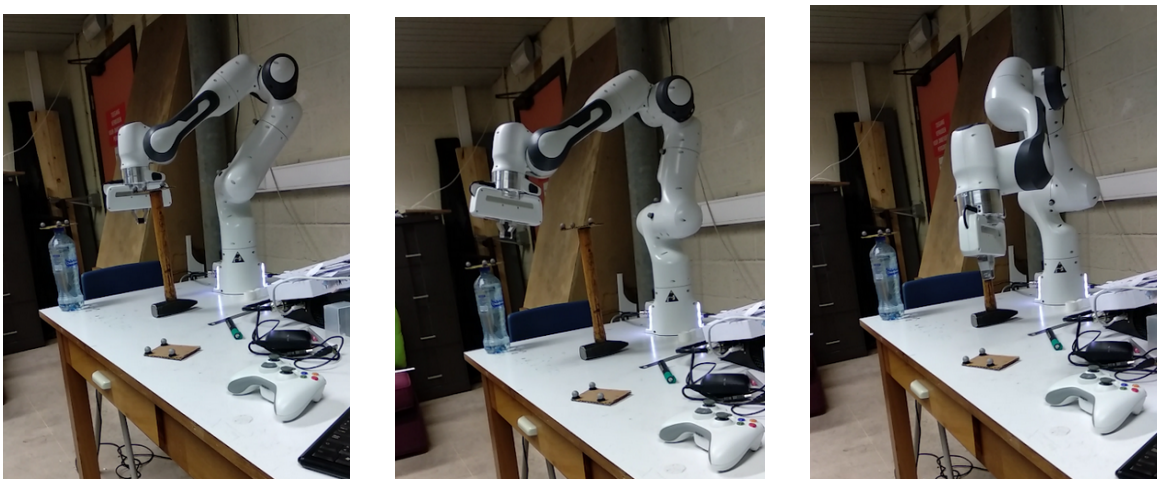


Figure 3.1: Avoiding a hammer and a bottle in order to reach the 3-marker frame

Figure 3.2: Avoiding a bottle that is close to the arm

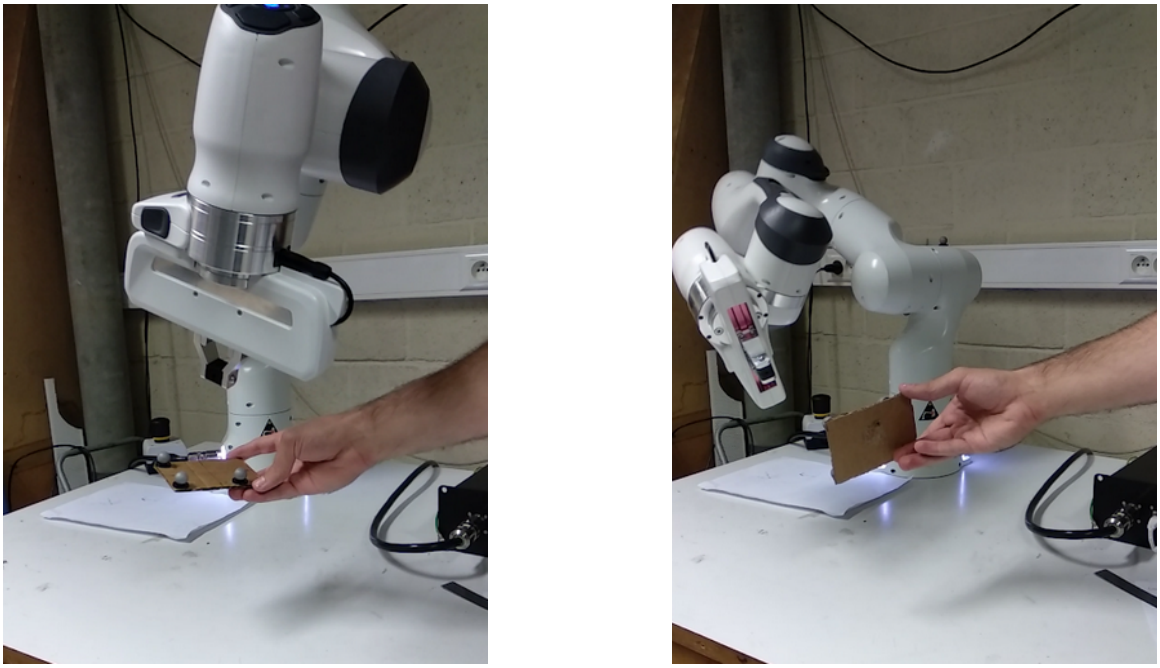


Figure 3.3: Robot following the orientation of the reference frame

3.1 Setting up the experiment

To execute this experiment you need to:

1. Make sure the robot is turned on and that you have setup the Vicon system as described before.
2. Select *New_session2* as the current session in the Nexus software and make sure all 3 frames are checked in the subject list.
3. Place the 3 frames in locations that are visible from the cameras.
4. You are ready to run *Vicon_obstacle_avoidance* program.

3.2 Working principle

3.2.1 Reaching the desired end-effector position

The code and the equations responsible for making the robot go to a desired end-effector position, are the same as the *teleoperation* program but with some small adjustments, so that the equations fit those in the ERG method.
The main difference is that this time the Cartesian velocities are calculated using the following formula:

$$\mathbf{V_{attr}} = K_{attr}(\mathbf{p_{d_k}} - \mathbf{p_k}) \tag{3.1}$$

Equation (3.1) actually sets the desired velocity proportional to the error from the desired Cartesian end-effector position and describes the attraction field since the end-effector will tend to reach the desired position.
To transform these Carthesian velocities to joint velocities and to subsequently compute the torques, use Equations (2.4) and (2.5) respectively. The only difference for (2.4) is that here the full Jacobian (6x7 and not 3x7) is used since we also need to change the orientation.
Note also that other K_p and K_d gains are used to ensure that the robot doesn't accelerate too fast in the beginning and that it reaches the desired position with a negligible error. In the following equations, *total_dist* is the initial distance, from the starting to the desired point and *dist* is the distance from the current end-effector position to the desired position (so $\frac{dist}{total\_dist}$ is the percentage of the distance the robot has still to travel). Moreover, the controller will initially have PD gains of $K_p = 10,\ K_d = 3$ (so that it will start relatively slowly) and near the desired position the gains will be equal to $K_p = 50,\ K_d = 10$ (for a stronger and more precise control).

```
Kp = 10 + (1 - dist/total_dist) * 40;
Kd = 3 + (1 - dist/total_dist) * 7;
```

3.2.2 Avoiding Obstacles by repulsion fields

In this experiment, there are 3 frames used. One of them is the desired frame and the other two play the role of spherical obstacles that the robot needs to avoid.
In the following figure, the "obstacle" frames are placed on the top of a bottle and a hammer and the desired frame is placed on the table.

Figure 3.4: Obstacle and desired frames

To avoid the obstacles, the distance between the end-effector and the centers of the "spheres", as well as the distances between the last four joints and the "spheres" are calculated. Then, for each obstacle and each joint, a repulsion velocity vector $\mathbf{V_{obst}}$ is computed which has the opposite direction from the vector that connects the joint position and the obstacle and is similar to a barrier function:

$$barrier(\epsilon, dist, offset) = \begin{cases} \frac{(\epsilon - dist)}{dist^2 + offset}, & dist < \epsilon \\ 0, & dist > \epsilon \end{cases} \tag{3.2}$$

The *offset* variable is only used to make sure that the barrier function doesn't get a very high value when the distance tends to 0, since then the barrier function would go to infinity which would cause a very sudden and unwanted reaction from the robot. The repulsion and attraction field is visualized in Figure 3.5 where the top bottle is the center of the sphere.

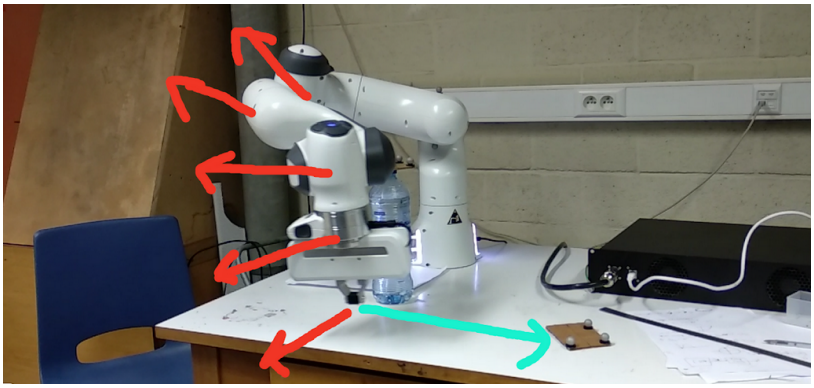


Figure 3.5: Repulsion fields (red) and attraction field (blue)

Note that in order to calculate the repulsion velocity of each joint position, a different

Jacobian matrix is required since we need the position of the joint to move and not always the end-effector. When a new Cartesian field velocity (either repulsion or attraction) is computed, it transforms to a joint space velocity and gets added to an accumulator velocity variable. As a result, the desired joint velocity in the end will be the sum of all the previous repulsion and attraction velocities and which is used to calculate the torques.

3.2.3 Following the frame's orientation

The orientation in the XY plane:
The $4x4$ ent-effector pose matrix (which is available through the libfranka library) for the end-effector frame, provides not only information about the position of the end-effector, but also about its orientation (relative to the base frame of the robot). More specifically, the pose matrix can give us the end-effector orientation vector, relative to the base frame of the robot, by calling the pose method of the Model C++ class.

The orientation vector is visualized for two cases in Fig. 3.6. In the first case (left) the robot is vertical to the table plane (which is the same as the XY plane) and its end-effector is facing downwards, the orientation vector should be equal to $\overrightarrow{Or} = (0, 0, -1)$ relative to the base frame.

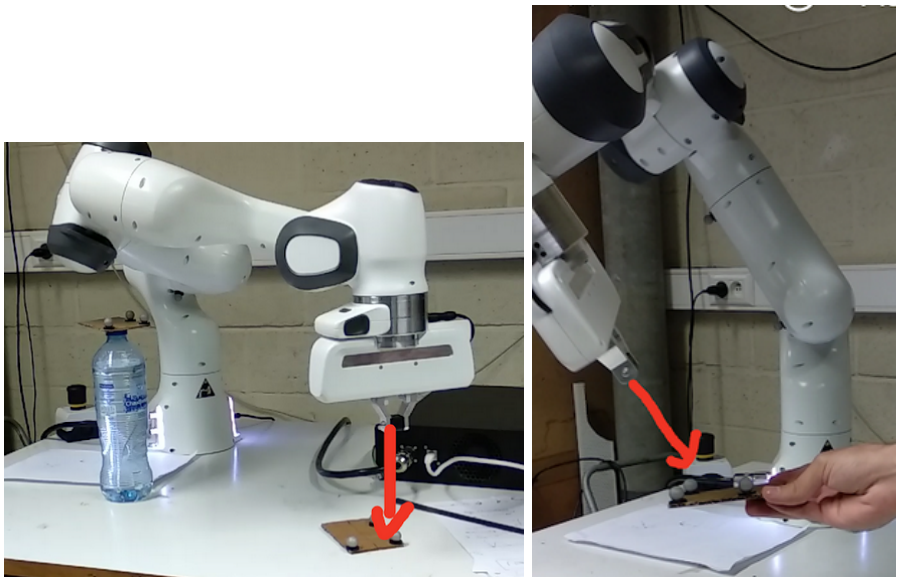


Figure 3.6: Orientation vector

Next step is be to calculate the vector vertical to the desired frame, so that we will be able to write a control law in order to align those two vectors. The vector vertical to the desired frame is computed by calculating the cross-product of the vectors that are formed by the three markers, as shown in the following figure.

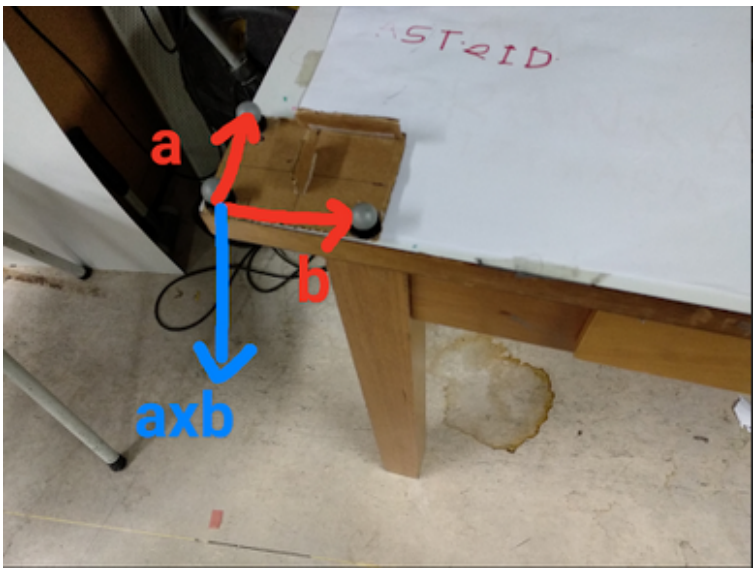


Figure 3.7: Vector vertical to the frame

After having the desired frame and the end-effector orientation vector, we need to find the angles between the projections on the XZ and YZ plane and the **z** vector (the angles on the XZ and YZ plane). After finding those angles, we use a PD control law for the X and Y angular velocities with the following equation, in order to align the current and the desired orientation vector:

```
new_vel(i) = prev_vel(i-3) + (KpOrr * (desired_angle[i-3] - angle1)
-
KdOrr * prev_vel(i-3)) * period.toSec();
```

where angle1 is the current angle on each axis, and i takes the values 3 and 4 (*new_vel(3) and new_vel(4)* represent the angular velocities on the X and Y axis respectively, while *new_vel(5)* is the Z angular velocity, which will be calculated separately later to control the gripper orientation). Note that the velocity vector this time has 6 instead of 3 elements since we need to include the angular velocities. The Jacobian also needs to be a 6x7 matrix.

The orientation of the gripper:
Previously, we controlled the XY angle so that the end-effector orientation vector matches the desired frame's orientation vector. Now we need to control a last angle, in order to align the gripper with two markers of the frame. As indicated in the next figure, the angle between those two vectors is computed and then, using the same form of PD control, the angle is demanded to become zero.

Figure 3.8: Angle between the gripper and the markers

Chapter 4

Object Stack

This experiment is similar to the *Vicon_obstacle_avoidance* but this time there are three desired frames instead of one and there are no obstacles. The goal is for the robot to properly grab the three frames (that are considered to be objects), whose position and orientation is not predefined, and put them at a specific position and orientation inside a box as shown in Fig. 4.1.

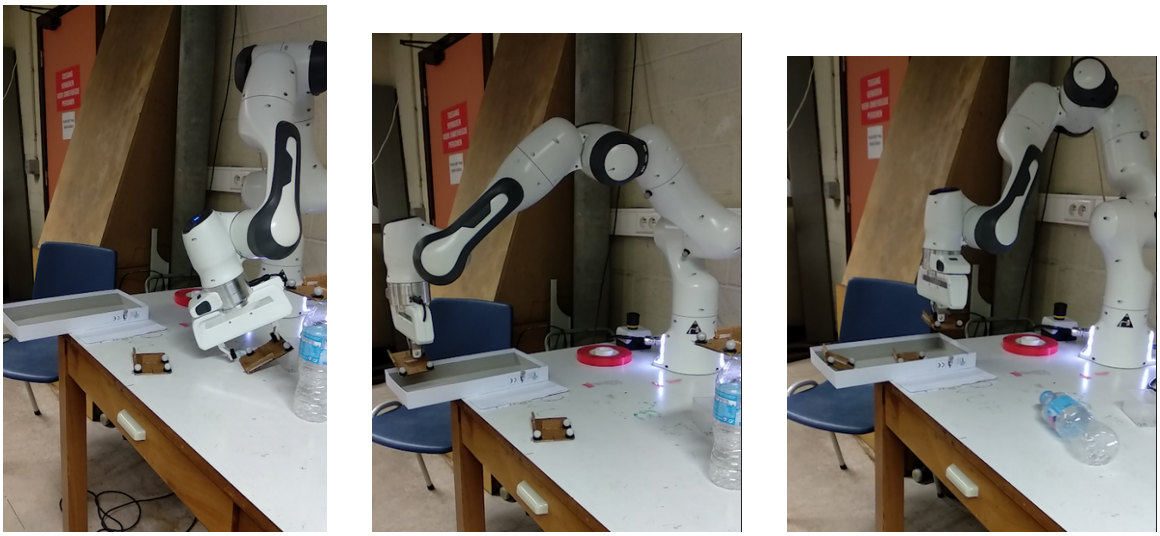


Figure 4.1: Placing the frames (i.e. objects) inside a box

4.1 Setting up the experiment

The procedure you need to follow in order to execute this experiment is the same as for the *Vicon_sobstacle_avoidance*.

4.2 Working principle

The way the robot reaches the desired position and orientation is in the same as the previous program, see Section 3.2. The robot in this program follows the following steps.

1. Firstly, the robot tries to reach the first desired position, which is $8cm$ higher from the center of one of the frames. Vicon system represents each frame and each marker on the frame with numbers. In this program, the robot will initialy go to the first frame that Vicon publishes (which is a specific one), then the second and lastly the third, but the code can be easily modified so that the robot will attempt to go to the frame closer to it.

2. After reaching this position, the desired position changes so that the robot gets closer to the object, at a distance around $1cm$.

3. When the robot is at the right position and orientation, the gripper closes in order to grasp the object.

4. Then, the desired position changes once again, this time to a predefined releasing position above the box. Note that for this motion, extra velocity on the z-axis is added so that the robot performs a curved motion before reaching the box. This part will be described later in more detail.

5. Upon reaching the desired position above the box, the robot releases the object and repeats the same procedure for the other frames.

In order to perform a curved motion when it's time to put the object to the box, we need to add an offset on the Z-position of the robot. The Z_offset needs to be a function similar to Fig. 4.2.

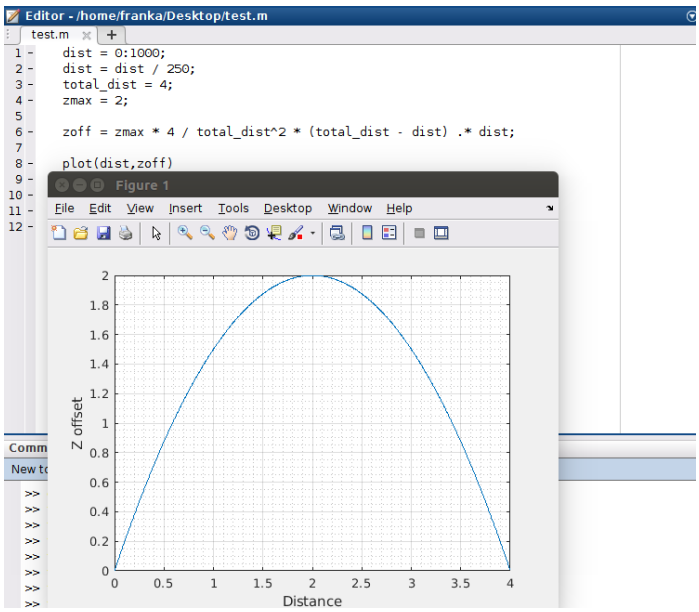


Figure 4.2: Offset on the Z axis

In Fig. 4.2 you can see the parabolic function and its equation. Zmax is the maximum z value we want to reach and total_dist is the initial distance from the desired point. Note that we need this offset only for around half the distance, so that the end-effector will go upwards and then it will go down again since the desired position will be lower. By differentiating the previous equation (because we need to transform the position offset to velocity) we can have the following *if* statement for the velocity offset (the values 4 and 8 are a result of the previous differentiation, while the 0.45 value is chosen so that this offset is only applied when the robot has to travel more than the 45% of its total distance and was chosen experimentally):

```
if (transistion_state && dist/total_dist > 0.45) new_vel(2) -=
1.0*(zmax*4/total_dist - zmax*8*dist / (total_dist*total_dist)) * period.toSec();
```

Moreover, in this experiment there is a high chance that the last joint will reach its angle limit, resulting the program to stop. In order to avoid this problem, we check if the desired angle will cause the last joint to exceed its angle limit and if so, we change the desired angle by adding or subtracting 180 degrees.

Chapter 5

Pick and Place

In this experiment, the robot goes to a desired position given by a frame, grabs an object at this position, and then goes to a second frame and releases the object. This experiment can also be considered as a human-robot interaction experiment, since the positions where the robot grabs and releases the object, is actually the human hand that will give or receive the object. Some pictures of the experiment can be seen below.

Figure 5.1: Transferring an object from one person to another

5.1 Setting up the experiment

The procedure you need to follow in order to execute this experiment is the same as for the *Vicon_obstacle_avoidance*.

5.2 Working principle

The outputs of the control loop in this experiment are Cartesian Velocities and not torques, so we need to calculate the desired velocities (we started making this program by using torques, but torques added some difficulty so we switched to Cartesian Velocities). Moreover, in this experiment there is no desired orientation so we only compute the three Carte-

sian velocities, using the same "attraction" equations as the two previous programs. The robot follows the following steps.

1. Firstly the robot goes to one of the two frames (user's hand).
2. As long as the hand of the user is not moving and the robot has slowed down enough, the robot grabs the object.
3. Then the robot transfers to the second frame (hand) and when it slows down, it releases the object.
4. The same process repeats till the user manually stops the program ($Ctrl + C$).

Note that again the Z offset is added so that the robot performs a curved motion when going from person 1 to person 2 with the object.

Chapter 6

Inverted Pendulum

In this experiment, we place a long metal stick vertically on the end-effector of the robot and the goal is to make the robot stabilize the stick while the desired end-effector position is possible changing.

6.1 Setting up the experiment

To execute this experiment you need to:

1. Make sure the robot is turned on and that you have setup the Vicon system as described before.

2. Select *New_session3* as the current session in the Nexus software and make sure the stick checked in the subject list.

3. Place (or move) the three markers on the stick as indicated in Fig. 6.1 and make sure that the markers are very well aligned.

   For the purposes of the experiment, we put three markers on the stick. Actually, only two markers are needed to measure the angle of the stick w.r.t. the robot and to use those measurements in the feedback stabilizing control loop to control the robot, but a third marker is required so that the stick can be recognized by Vicon.
   The stick with the three markers on it can be seen in Fig. 6.1. In this figure, you can also see a fourth marker at the base of the stick. This markers is used to have a better contact (curved surface instead of flat surface) with the robot's end-effector.

Figure 6.1: The pendulum stick with the three markers

4. Check if the stick is visible from the cameras. If the stick is blinking in the nexus program, change its orientation or its position. If the problem still exists then you probably need to recalibrate the cameras.

5. You are ready to run *Vicon_inverted_pendulum2* program.

Important: For this experiment high precision is needed in order to be executed successfully. This means that if the markers on the stick are not very well aligned or if the stick is blinking (either because of bad calibration or because the stick is occluded from the cameras) in the Nexus software, then the robot will not be able to balance it properly.

The functional program is called *Vicon_inverted_pendulum2*. You can run the program without needing to place the stick on the robot from the beginning. While the program is running, you can place the stick on the robot and the robot will start moving only when the stick is vertical. In the first 7 seconds the robot will try to stay in its original position, but after the 7 seconds, the desired position changes and the robot will try to move to the new position and also balance the stick.

Figure 6.2: The first success of the experiment

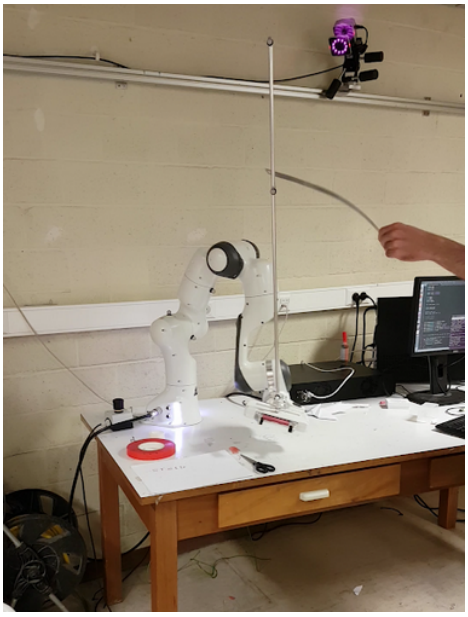


Figure 6.3: Hitting the stick with a ruler while the program is running

6.2 Working principle

This experiment is limited to the XY plane, so any movement of the end-effector on the z-axis is unwanted. To make sure that the robot remains in the same Z position, we simply use the following equation for the velocity on the z axis:

```
new_vel(2) = 7.5 * (z_ref - current_pose[14]);
```

New_vel(2) is the desired velocity on the Z axis, z_ref is the initial Z position of the robot which we want to maintain and current_pose[14] is the current Z position of the robot. Furthermore, it is also needed that the last joint angle doesn't change in the middle of the experiment, so it is separately controlled to remain at its initial angle:

```
Torque[6] = 10 * (last_joint_ref - current_q[6]) - 3 * current_qdot[6];
```

For the movement on the XY plane, the equations of the *Vicon_teleoperation* program are used, where the acceleration is computed from (2.2) and then we use integration to determine the desired velocities.

```
new_vel(i) = new_vel(i) + acceleration[i] * period.toSec();
```

After calculating the desired velocities, they get transformed into torques the same way as before (2.5)?. Now, the acceleration is the basic term for controlling the stick, since it describes the virtual force on the end-effector which is also the input of the feedback system. The acceleration of the end-effector is a sum of 5 different terms and is set equal to:

```
acceleration[i] = -Kmove * (pd[i] - current_pose[12+i]) + Kang * (ref[i] - angl[i])
+ Kangvel * ang_vel[i] + Kvel * avg[i]/20 - Kd * new_vel(i);
```

Now we will analyze each term in the previous equation.

- All variables that start with a K (Kmove, Kang, etc.) are gains.
- The term $-Kmove * (pd[i] - current\_pose[12 + i])$ is responsible for keeping the end-effector position to a desired point ($pd[i]$). Note that since Kmove is a positive constant, the minus in front of the equation makes the position feedback positive, which is necessary for the stability of the pendulum, despite the fact that usually positive feedback leads to unstable behavior. The theoretical reason for this behaviour, is that selecting the eigenvalues of the linearized system so that they have a negative real part, will lead to a negative $Kmove$ gain. Another way to think of it, is that if the $Kmove$ gain was positive, then the robot would tend to move directly towards the desired position and the pendulum would fall towards the opposite direction. What we need is exactly the opposite, so that the pendulum would fall towards the desired position and then the robot would try to follow the pendulum in order to balance it and ultimately the robot itself will transfer to the desired position.

- The term $Kang * (ref[i] - angl[i])$ is responsible for keeping the stick straight (with zero angle from the z-axis). The constant $Kang$ needs to have a relatively high value, since we have to ensure that the robot will move very fast when the pendulum is not straight, in order to prevent it from falling.
  The *ref[i]* variable is an almost zero reference that needs to be tuned to compensate the inaccuracies of the marker placement on the stick. These values can only be tuned experimentally and more specifically by running the experiment and making sure that the robot doesn't tend to go to a specific direction when releasing the pendulum.
  The *angl* array is a two-element array for the XY angles of the stick from the Z-axis. Actually, since the angle is going to be very small through the experiment, the length of the vector (projected on the XZ and XY plane) between the bottom and the top marker of the stick will equal to the *angl* instead (the stabilizing control will not allow the stick angle to be more than a few degrees and moreover we assume that $\sin x \approx x$ if x is small).

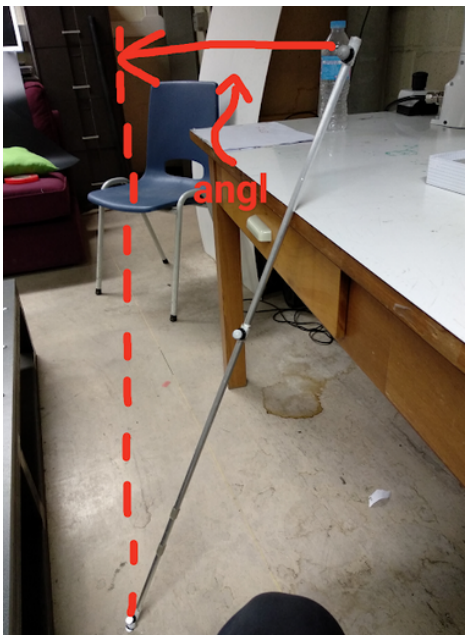


Figure 6.4: The angl variable (there are two angl variables, one for the X and one for the Y axis)

- The $Kangvel * ang\_vel[i]$ term is responsible for speeding up the end-effector proportionally to the angular velocity of the stick.
- The $-Kd * new\_vel(i)$ is the damping K_d term.
- The $Kvel * avg[i]/20$ is an extra term that speeds up the end-effector proportionally to the stick's linear velocity. This term is useful because it guaranties that the stick will not accelerate forever in any direction since the end-effector will always become "faster" and reach the stick. Increasing $Kvel$ makes the end-effector very shaky, causing also the stick to shake. To solve this problem, we take the speed of the middle

marker as the stick velocity, since it appears to be the most stable to vibrations and also, instead of using the speed itself, we filter it, by averaging the last 20 recorded velocities of the middle marker. That's why the variable is called *avg* and its divided by 20.

The *Kmove* gain is used to stabilize the end-effector position in a desired place. The end-effector position does some small oscillations between the desired point, since it also tries to stabilize the stick. To reduce the oscillations, a relatively high value of *Kmove* is needed, but if the desired position changes, a high *Kmove* value may lead to a sudden unwanted behavior. This is why a variable, according to the distance from the desired position, *Kmove* is used:

```
Kmove = max((80 - 190 * dist), 25.0);
```

6.3 Teleoperation for the desired reference

Next step was to create another program, that instead of having a predefined desired position as an input, the desired end-effector position is given by the user's arm, like in the *Vicon_teleoperation* program. The user, apart from moving the the end-effector position can also control the last joint angle with his thumb and index. Some pictures of the pendulum experiment with teleoperation can be seen below.

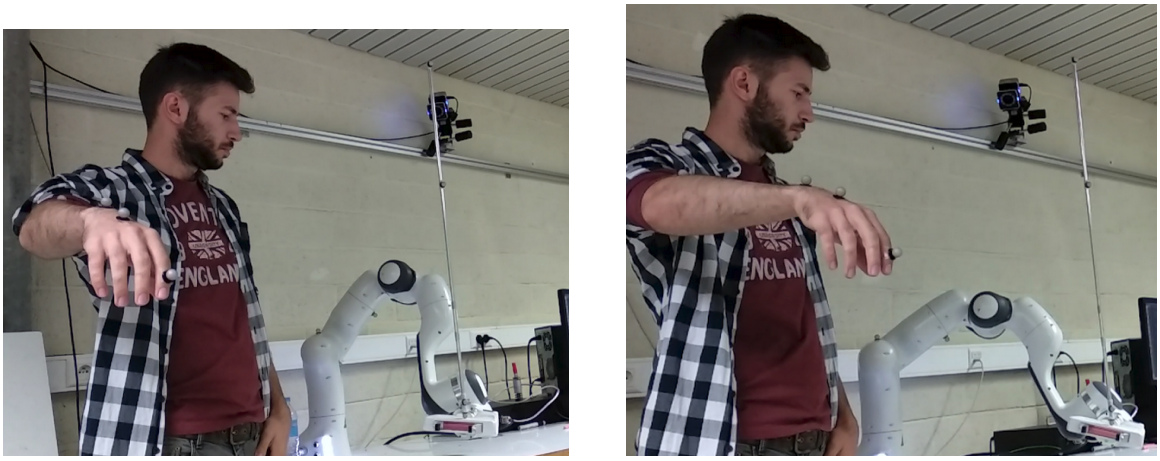


Figure 6.5: Controlling the inverted pendulum with the user's arm

6.3.1 Setting up the experiment

To set up this experiment, you need to follow the same steps as the *Vicon_inverted_pendulum2* program and also:

- Place the markers to your arm as described in the *Vicon_teleoperation* experiment.
- Also check MyArm in the subject list (Session3, Nexus software).

6.3.2 Working principle

The *Vicon_inverted_pendulum_teleop* program is a simple mix of the *Vicon_inverted_pendulum2* program that stabilizes the stick and from the part of the *Vicon_teleoperation* program that changes the desired XYZ and last joint position, according to the movement of your arm.

6.4 Xbox controller for the desired reference

Similar to the *Vicon_inverted_pendulum_teleop.cpp* program which controls the position of the robot's end-effector with the user's arm while the robot balances the pendulum, there has been created another program called *Vicon_inverted_pendulum_xbox.cpp*, which controls the end-effector position with the Xbox device. Note that in this program, the user can only control the XY position and the last joint angle with the Xbox controller and not for example the gripper state, as in the *Xbox.cpp* program.

To run this experiment, you need to set up the Vicon system in the same way as for the *Vicon_inverted_pendulum_teleop.cpp* program and also ensure that the Xbox controller is properly connected with the computer.