AALBORG UNIVERSITY
DENMARK

Object Manipulation and Task Planning: Mini Project Report

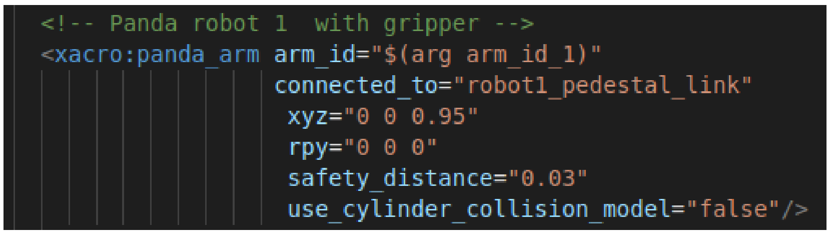
Group 853

İbrahim Doruk Kızıloklu
Mahed Dadgostar
Victor Bjorholm
Kutay Shentyurk

Lecture 1:Building a Robot Simulation Environment in ROS

In the first lecture we built our simulation environment with urdf(unified robotics description format).It is a domain specific modeling language and it allowed us to store information about robot kinematics and dynamics.We used the Franka Emika Robot Arm in the simulation environment . All the visualization of this Robot has been designated in the franka_description file ,visual meshes are imported as DAE ,these can have their own colors and materials and collision meshes are imported as STL files.For simpler configuration we used xacro files to manage the urdf files.

We start to structure the environment by making the world interface to move everything according to that origin.Then we get the visuals of the Fib 14 map inside of a xacro file and link the map to the world interface. After that we import the robot arm and hand xacro . Made a pedestal for robot arms to stand in the factory at the level of automation bands connect the pedestal to the world interface then connect the robot arm into the pedestal .Then add the coordinate and orientation parameters and collision avoidance we set the cylindrical collision model into false meaning that default mesh based collision model will be used for the robot.Xacro parameters of robot arm example is given below :



```
<!-- Panda robot 1  with gripper -->
<xacro:panda_arm arm_id="$(arg arm_id_1)"
                 connected_to="robot1_pedestal_link"
                  xyz="0 0 0.95"
                  rpy="0 0 0"
                  safety_distance="0.03"
                  use_cylinder_collision_model="false"/>
```

To form the factory each conveyor module has been added from the aau_lab_models file . All the dae files and the urdf files have been given by the lecturer.The conveyors are only used for visualization. They don't have the functionality of a real belt. So for object detection we put the object directly in front of the robot arm on top of the conveyors instead of putting on a starting point of a belt , moving the belt and detecting the object with a camera and then stopping the belt to grasp the object. These functionalities are crucial for object detection and grasping in a factory environment and can be added in the future.The visualization of the factory can be seen in the figure 1.

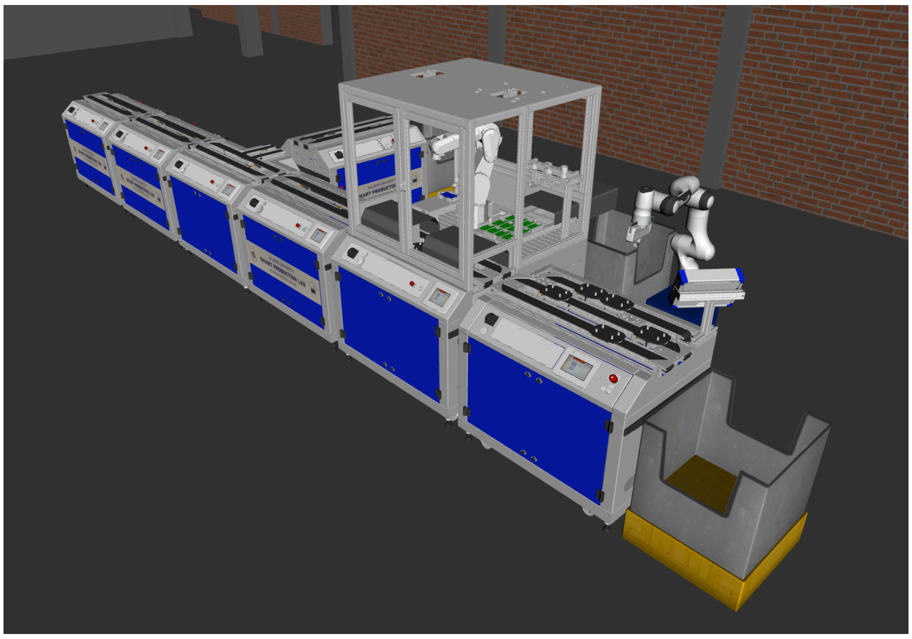
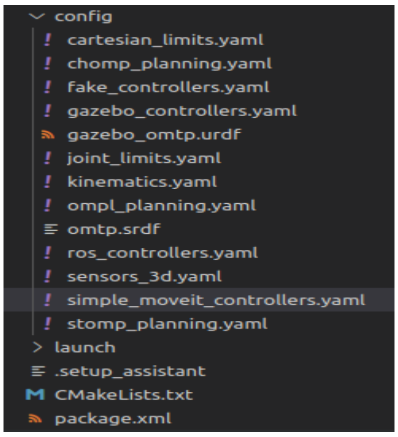


Fig1: Factory Env

Lecture 2: Manipulation with MoveIt

In this lecture we set up our robot arm for easier manipulation configuration with the help of MoveIt. With the help of the MoveIt Setup Assistant we manage to set up positions for each joint of the robot arm and two tool joints individually.First we load the xacro of our factory to the setup assistant .Then we edit the setup assistant parameters according to the video tutorials.On the Controller section we only use effort_controller/JointTrajectory controller for the hand and arm . FollowJointTrajectory that mentioned in the tutorials has been implemented in the noetic version in the config file as simple_moveit_controllers.yaml . These differences while migrating from melodic to noetic should be taken into consideration.Moveit Config file should look like in the Figure2 for noetic versions.



```
config
  cartesian_limits.yaml
  chomp_planning.yaml
  fake_controllers.yaml
  gazebo_controllers.yaml
  gazebo_omtp.urdf
  joint_limits.yaml
  kinematics.yaml
  ompl_planning.yaml
  omtp.srdf
  ros_controllers.yaml
  sensors_3d.yaml
  simple_moveit_controllers.yaml
  stomp_planning.yaml
launch
.setup_assistant
CMakeLists.txt
package.xml
```

Fig2:Config File for Moveit

Then we define a couple of positions for grasping movement. We made robot_ready and up positions similar to tutorials mentioned and robot_grasp position and robot_place positions for the grasping action.After defining the trajectories we define the PID gains for each joint.Optimal gain values for each joints PID depends on the application. In this lecture we configure this values easily with rqt_reconfigure.

For the grasping sequence we implement the pick and place code with the following steps:

1.Bring the arm to a robot ready_position and open the gripper .
2.Place the arm on top of the box and give a certain space on the z axis .
3.Move the arm closer until the box is between tool fingers.
4.Grasp the object.
5. Place it on the designated position.

Lecture 3: Object Detection and Grasping

Factory placement and move configurations have been done in the previous lectures. In this lecture we place objects in front of the manipulator and detect it with different cameras to take the position of the object, transform the pose into a reference frame according to the robot arm and grasp it.We use a logical camera to get the position and orientation of the object. We do not have a logical camera in the real world so we should use a 2D/3D camera in order to get the real world outputs. We implemented a 2D camera for object detection in the lecture 5,6 and we need to put a 2D camera into our xacro file and then reconfigure the move files to be able to see the camera in our environment.The camera publishes images on the /camera1/image_raw topic and camera information on the /camera1/camera_info topic.

We implement a transform_object_pose node which outputs the logical camera's messages. The logical_camera_callback function checks if the logical camera has detected a box object. If it has, the script initializes the movement commander for the robot arm and hand groups, and moves the robot arm to a predefined "robot_ready" position.

The detected box object's pose is transformed from the logical_camera1_frame coordinate frame to the world coordinate frame using a TransformListener. The script then computes a Cartesian path for the robot's end effector to follow towards the object. The robot arm executes the planned trajectory and moves towards the box object.

Lecture 4: Behavior Design with State Machines

In lecture 4 of the course, the students are tasked with controlling a gripper using a behavior application, namely FlexBE. Cloning the FlexBE application from its git repository, and unzipping the lecture 4 files into catkin workspace, a catkin_make is done. The lecture files include a module with a Franka robot. In order to run the module from the lecture files, one needs to undergo the process of Moveit setup assistant, and generate Moveit configuration module for the .urdf.xacro provided. We utilized the previously generated panda config module named "panda_v1" in the git repository. We simply modified the launch file so that it takes the omtp factory as the xacro file, and the corresponding moveit configurations.

Private configuration and userdata of the behavior is defined in the same way described in lecture videos. We added some more parameters in order to be able to control the end effector tool, such as joint names and values for certain group poses just like the homing of the robot arm group. We first developed a behavior that detected the part, calculated the joint configuration values, and moved to the position. The camera state that detected the object position had some errors, and we fixed it by manually inputting the message type of the camera topic. The grasp state we coded is a derivative of the state *MoveitToJointDynState* from FlexBE manipulation states. We changed it so that it takes the gripper joints and group as input, and the goal position. The goal position is passed as a parameter instead of an input from an earlier state since the position is either open or close. We use it at two different places. First at the very beginning of the behavior to open the tool, and close it right after the robot hand moves to the position to pick the object. Later, it moves back to a position called home. The state sequence can be seen in Figure (kutay1).

One problem we had was the fact that control fail occurred when the gripper was commanded to go to the closed pose because it requires the gripper to crash the object to actually end up at the position. Therefore, we overcome the problem by connecting the control failed outcome from the "closed" state to "move to home" state, which might cause problems with control but the control failure occurs rarely with the gripper. How the state sequence is connected can be seen in Figure .

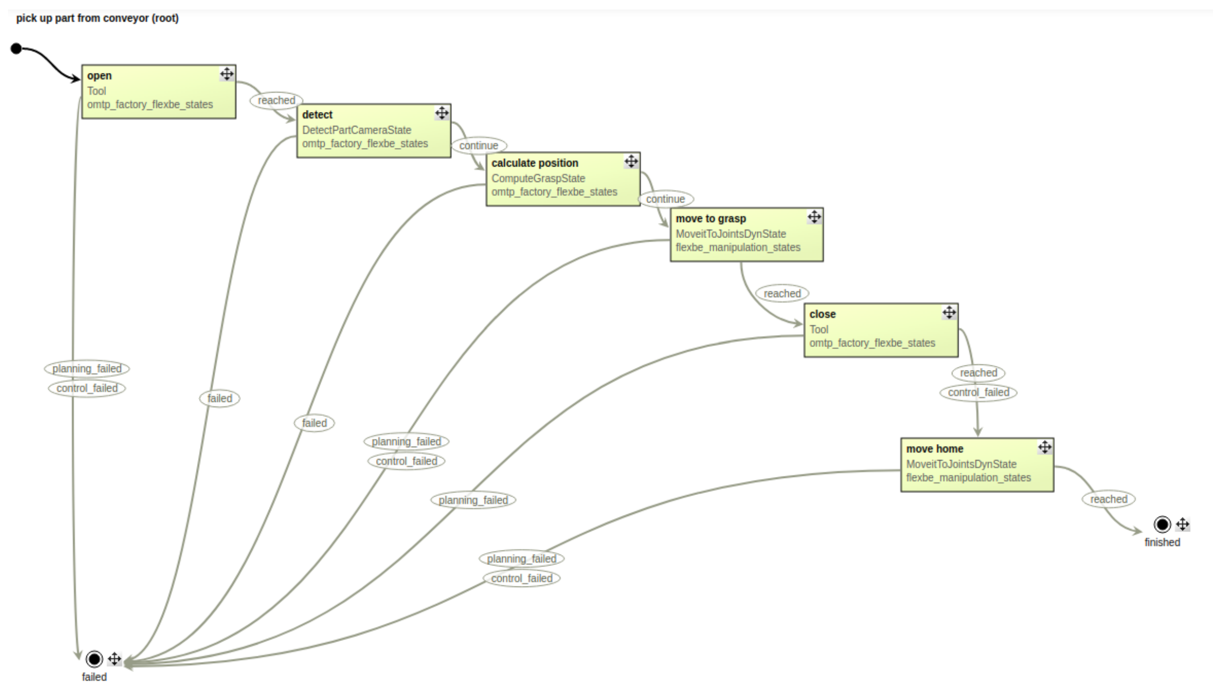


Figure : The state sequence of FlexBE Behavior

Lecture 5-6: CNNs in Practical Robotic Application

CNNs are neural networks that are commonly used in robotic applications, most commonly with vision applications. The CNN application for deciding whether people are wearing masks or not. The videos from the colab application can be seen in the lecture files in git repository. What is done there is that we are simply training a CNN using images of people with a mask and not. As a result the neural network is able to detect whether a person is wearing a mask.

The OMTP application of the CNNs is object detection. The provided weights for the lecture are capable of detecting a number of objects. The object we need to detect is a banana. The input to the neural network is the image pixels. In order to obtain images, a camera is placed in the OMTP factory environment. The implementation was provided within lecture files. However, the way it is done is that simply calling the plugin in the .urdf.xacro file, and indicating which topic to publish the images to. We had to generate a new Moveit configuration module because when you add the camera to an urdf, it needs to be made visible, therefore changing the module name in .launch file to the previously generated Moveit module did not work. The module named “cam_lecture5” is Moveit configuration module created for the OMTP environment with camera, and in order to keep the Rviz displaying the camera, the rviz configuration setting in .launch file is kept the same. The rviz part of the file is as follows:

```
 <include file="$(find cam_lecture5)/launch/moveit_rviz.launch">
   <arg name="rviz_config" value="$(find omtp_lecture5)/rviz/
moveit.rviz"/>
 </include>
```

One of the problems faced is that if the banana is placed on top of a Festo module from OMTP factory, the CNN cannot detect the banana, and instead detects other objects such as train. That is because the top view of festo modules resemble the data fed to the neural network as a train. In order to be able to use it with festo modules, a new set of trained weights is required.

Lecture 7-8:Robots in Contact (DMP)

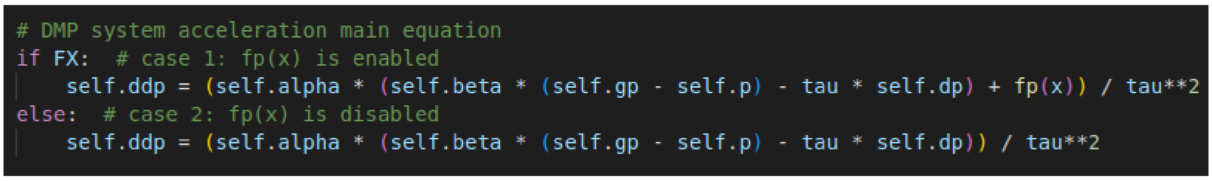
Dynamic Movement Primitives have been used in these lectures to move the robot arm with a human like desired trajectory.We are imitating a desired path with basis functions to guide the real system.For this method we will use something similar to PD control signal plus a forcing function.Our system acceleration main equation is given in the figure.

$$\ddot{y} = (\alpha_z(\beta_z(g - y) - \dot{y}) + f)\tau$$

α_z=damping(D) , β_z=stiffness(K) , g = target position ,

y=current position,τ=temporal scaling term,f=time dependent forcing term.

The implementation of DMP is in the code below.



```
# DMP system acceleration main equation
if FX:  # case 1: fp(x) is enabled
    self.ddp = (self.alpha * (self.beta * (self.gp - self.p) - tau * self.dp) + fp(x)) / tau**2
else:  # case 2: fp(x) is disabled
    self.ddp = (self.alpha * (self.beta * (self.gp - self.p) - tau * self.dp)) / tau**2
```

Fig3:DMP Implementation

In order to understand the movement of the main equation we need to write the forcing function as well.Forcing function helps us form the to move on the desired trajectory.Canonical system of forcing function can be defined as :

$$f(x) = \frac{\Sigma_{i=1}^{N} w_i \psi_i(x)}{\Sigma_{i=1}^{N} \psi_i(x)} * x(g - y_0)$$

w=weights,ψ=superposition of gaussian terms

When x reaches zero at the end of the movement the forcing function becomes zero
To move a large distance equation itself scales up the forcing function so the influence of the forcing term is sufficient enough.To choose the weights so that the forcing function matches desired trajectory we use least squares in the code and optimize the weights for each joint of the manipulator.

In the first exercise we test the DMP for a given trajectory for a couple of different scenarios.We change the tau(temporal scaling term) and compare the different scales if the manipulator can perform the desired trajectory.
To get the joint-space and cartesian-space trajectory we implement the second exercise.we get the necessary joint , cartesian positions and angle orientations .First we train the joint-space DMP according to these parameters and plot the results .Then we train the quaternions and plot the results.On the third exercise we use Mujoco simulation the simulate the trajectory of dmp that we generate to move 0.3 meter in cartesian coordinate. Necessary videos and plots are in the github file of lecture 7.

Lecture 9 :Deep Reinforcement Learning for Robot Control

In this exercise, we attempted to solve the CartPole problem, one of the most well-known control tasks, using two distinct reinforcement learning algorithms: Deep Q-Network (DQN) and Proximal Policy Optimization (PPO).

The CartPole problem is a classic control task in reinforcement learning. The goal is to balance a pole on a cart by applying forces to the cart. The agent get a reward of +1 for every time step it keeps the pole upright, and the episode terminates when the pole falls or the cart moves too far from the center.

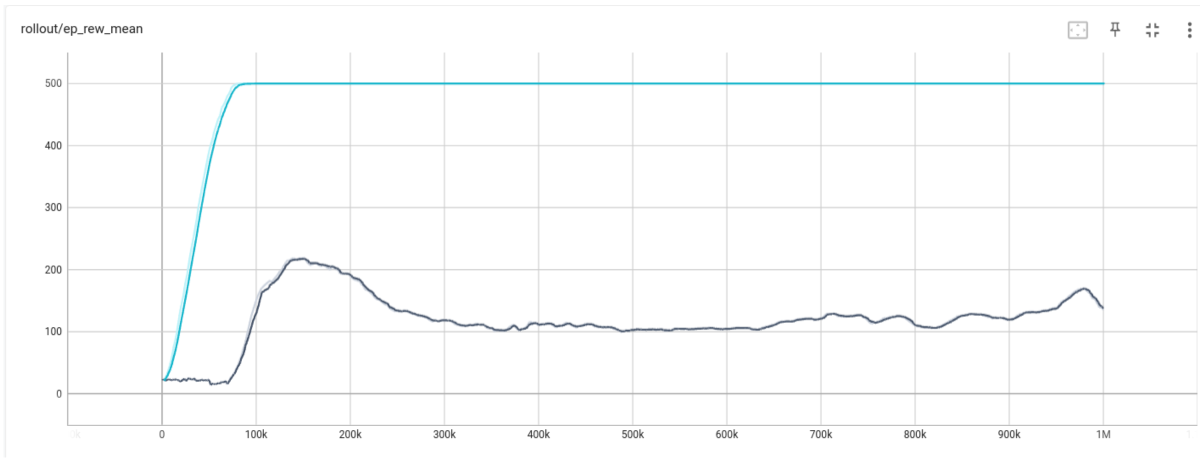
The environment used in this project is the CartPole-v1 environment from the OpenAI Gym library. It consists of a cart that can move horizontally with a pole hinged to the cart. The agent can apply forces to the cart, either pushing it to the left or to the right.

The two algorithms used in this implementation are:

DQN: Deep Q-Network (DQN) combines Q-learning with deep neural networks to approximate the Q-function, which maps state-action pairs to expected future rewards. The agent selects actions by choosing the one that maximizes the Q-value.

PPO: Proximal Policy Optimization (PPO) is a policy gradient algorithm that optimizes the agent's policy by maximizing the expected sum reward while making sure that the new policy does not depart too far from the old policy.This is achieved using a clipped objective function that penalizes large policy updates.

To train a new model, set the “algorithm” variable in the main function to either 'dqn' or 'ppo'. Set the train_model_flag variable to True for training or False for loading a pre-trained model. TensorBoard can be used to monitor the training progress of the agent. by using the command “tensorboard --logdir logs” in the python file directory we can observe the training process of the agent. Here is the image from the tensorboard that shows PPO(light blue) agent performs much better that DQN(black) agent that could not converge to the maximum optimal value in the timesteps of 1 million.



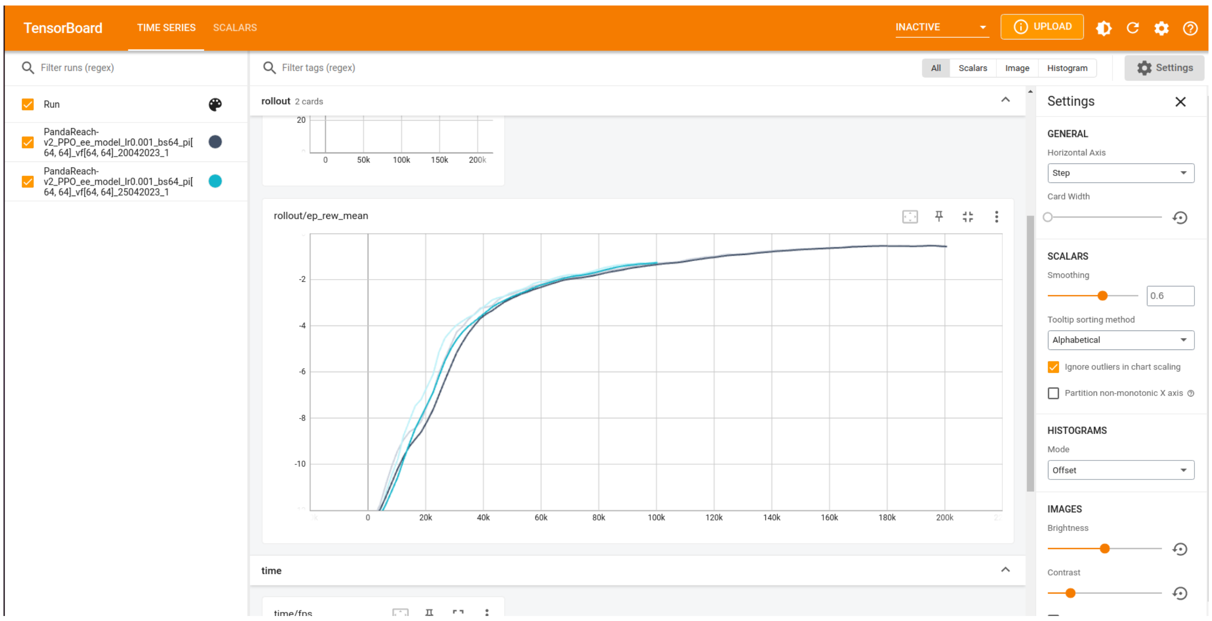
Lecture 10 : Deep Reinforcement Learning Part 2

In this exercise, an PPO agent was trained and tested in simulation environment of a Franka Panda using pybullet. You can find the main repository which provide the installation instruction for this exercise in this link.

To save and show the .gif of the simulation, the imageio package was installed using “pip install imageio”. A random agent was first tested in the environment by running “python panda_reach_random_agent.py”. To save the .gif, a list of frames rendered by the environment was created and the .gif file was saved using “imageio.mimwrite”.

The agent was then trained using the command “python panda_reach_train_agent.py”, which trained the agent for 200,000 steps and saved the model with the current timestamp within the name. The trained agent was tested for 200 steps using “python panda_reach_test_agent.py”, and a .gif was saved to visualize the test.

Tensorboard was also used to monitor the training process by running “tensorboard --logdir ./runs”. The Tensorboard output displayed two training processes, one for 200,000 steps and the other for 100,000 steps. A screenshot of the training process from tensorboard depicted below which shows that after around 180000 steps episode reward mean converges to a stable point:



Run
- ✓ PandaReach-v2_PPO_ee_model_lr0.001_bs64_pi[64, 64]_vf[64, 64]_20042023_1 ●
- ✓ PandaReach-v2_PPO_ee_model_lr0.001_bs64_pi[64, 64]_vf[64, 64]_25042023_1 ●

Settings

GENERAL
Horizontal Axis: Step

SCALARS
Smoothing: 0.6
Tooltip sorting method: Alphabetical
✓ Ignore outliers in chart scaling
☐ Partition non-monotonic X axis

HISTOGRAMS
Mode: Offset

IMAGES
Brightness
Contrast