FRA333: Kinematics of Robotics

Lab 4: Trajectory Tracking System

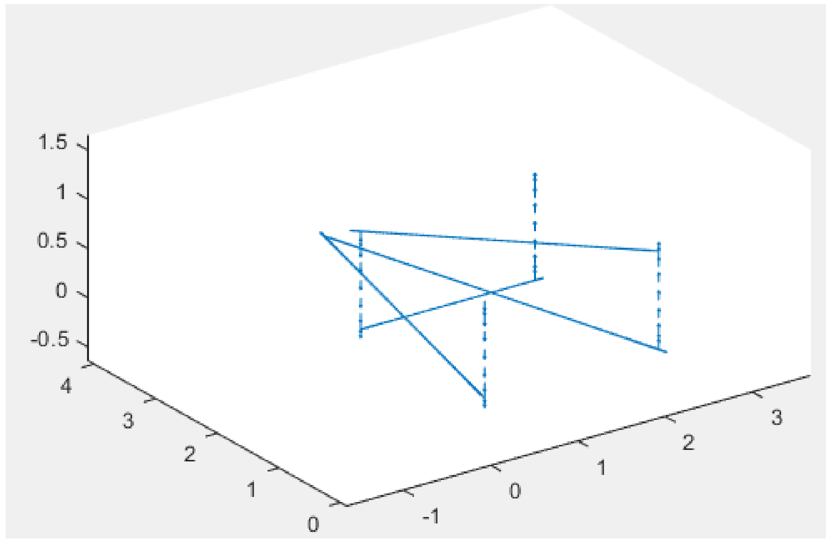
Objective

Objective

- Draw a sequence of letters “F”, “I”, “B”, and “O” in rviz using a robot in gazebo.

Objective

- Draw a sequence of letters “F”, “I”, “B”, and “O” on a 2D plane in rviz using a robot in gazebo.
- Draw a letter
  - Moving the end-effector along a path that resembles the letter
  - When it draws, the end-effector’s path must remain parallel to the plane.
  - The end-effector should retreat away from the drawing plane when it’s not drawing.



Objective

- Draw a letter
- Moving along a sequence of straight line
- Given a sequence of via points, move the end-effector linearly in the taskspace

Objective

- Given a sequence of via points, move the end-effector linearly in the taskspace
  - Track the reference joint trajectory
  - Generate the reference joint trajectory based on 2 via points
  - Scheduling the via points

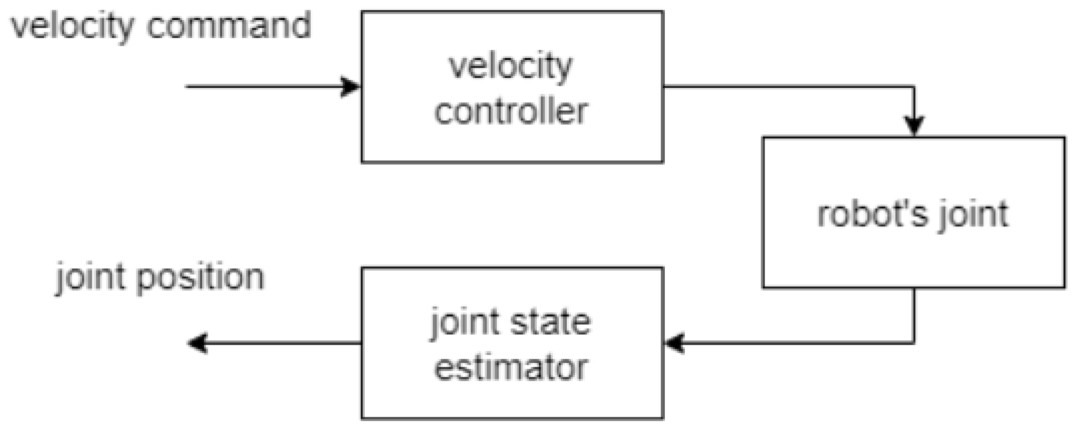
Objective

- Given a sequence of via points, move the end-effector linearly in the taskspace
  - Trajectory Tracking
  - Trajectory Generation
  - Trajectory Scheduling

Trajectory Tracking

Low-level Interface

If the low-level motion control system has done its job, the closed-loop control system will allow us to fully control the velocity of each joint. Furthermore, the low-level state estimation system will allow us to observe the position in joint space.



Control Policy and its objective

Based on the given assumptions, the goal of trajectory tracking control is to design a control policy $\pi(\cdot)$ that force a chosen state q to follow a desired reference trajectory q_r.

We can assume that the behavior of each joint can be represented by the following equation.

$$\dot{q} = \pi(t, q)$$

where $\dot{q}$ is the joint velocity, $\pi(\cdot)$ is a high-level control policy that depends on time t and the feedback joint position q.

The goal is for the following limit to be true.

$$\lim_{t \to \infty} \big(q_\mathrm{r}(t) - q(t)\big) = 0$$

Trajectory Tracking Control

One of the solutions is to use the following control policy.

$$\pi(t, q) = \dot{q}_{\mathrm{r}}(t) + K_p \cdot (q_{\mathrm{r}}(t) - q) + K_i \cdot \int_{\tau=0}^{t} (q_{\mathrm{r}}(\tau) - q) \ \mathrm{d}\tau$$

where K_p and K_i are proportional and integral gains, respectively.

Convergence of error

It can be proven that the error between the desired trajectory q_{r} and the actual chosen state q will approach 0 as time goes to infinity. If one substitutes the policy back to the differential equation, the error dynamics appears to be the following.

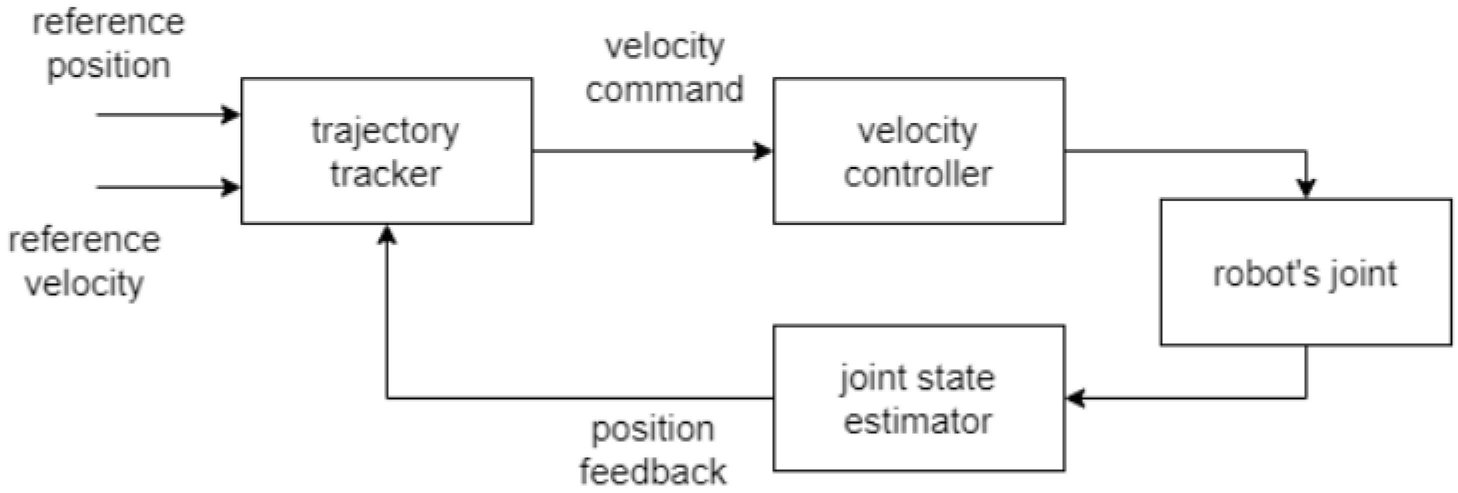
$$(\ddot{q}_{\mathrm{r}} - \ddot{q}) + K_p \cdot (\dot{q}_{\mathrm{r}} - \dot{q}) + K_i \cdot (q_{\mathrm{r}} - q)$$

This is a second-order homogenous differential equation with respect to the error $(q_{\mathrm{r}} - q)$. If the gains are chosen to be positive, the error will approach 0 as time goes to infinity.

Conclusion on Trajectory Tracker

$$\pi(t, q) = \dot{q}_{\mathrm{r}}(t) + K_p \cdot (q_{\mathrm{r}}(t) - q) + K_i \cdot \int_{\tau=0}^{t} (q_{\mathrm{r}}(\tau) - q) \, \mathrm{d}\tau$$

The proposed policy will guarantee the convergence of the error. One requires not only the position of the reference trajectory $q_{\mathrm{r}}(t)$ but also the reference velocity $\dot{q}$_r.



Possible implementation in ROS2

$$\pi(t,q) = \dot{q}_{\mathrm{r}}(t) + K_p \cdot (q_{\mathrm{r}}(t) - q) + K_i \cdot \int_{\tau=0}^{t} (q_{\mathrm{r}}(\tau) - q)\ \mathrm{d}\tau$$

Subscription: $\mathbf{q}$ →/joint_states, $\mathbf{q}_{\mathrm{r}}$ → /reference/joint_states

Publisher: π → /joint_velocity_controller/commands

Service Server: $\mathrm{enableTracker}$ →/enable

Timer: publish the control policy at fixed rate

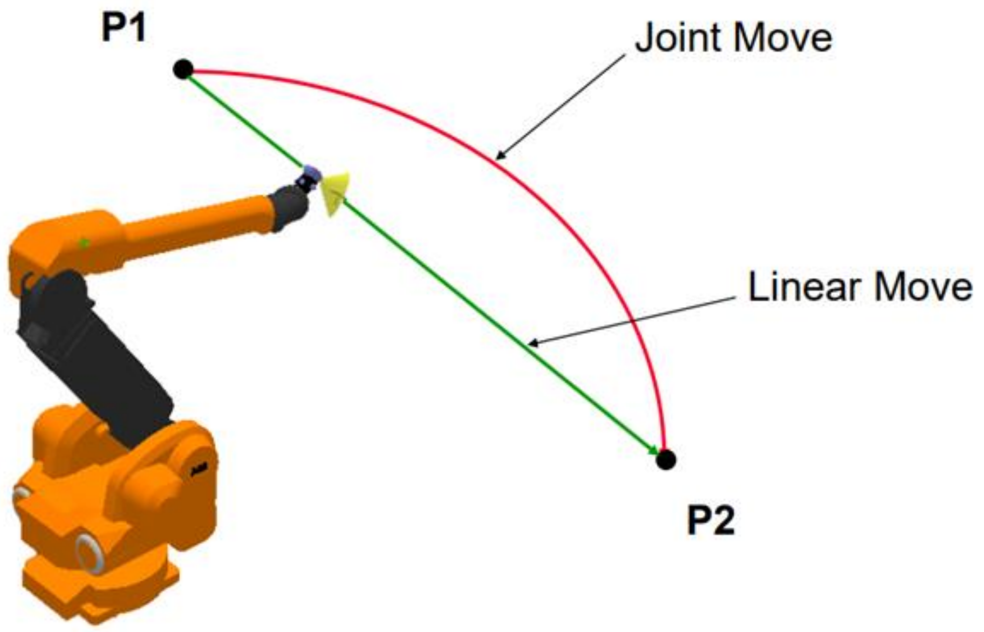
YAML Config : K_p, K_i

Trajectory Generation

Mode of operation

The objective of trajectory generation is to generate a time-dependent profile of position and velocity (and on occasion acceleration) that satisfies certain criteria. In robotics applications, a trajectory can be generated in either joint space or taskspace.

Therefore, most industrial robot has at least 2 modes of operation, joint mode and linear mode (each brand of robot call them differently, but we will them as joint mode and linear mode).



Joint Mode : Linear Interpolation

In joint mode, the trajectory profile $\mathbf{q}_r(t)$ is generated by interpolating between 2 desired joint configuration, which can be described as follows.

$$\mathbf{q}_{\mathrm{r}}(t) = \left(1 - \alpha(t)\right) \cdot \mathbf{q}_{\mathrm{i}} + \left(\alpha(t)\right) \cdot \mathbf{q}_{\mathrm{f}}$$

where $\mathbf{q}_{\mathrm{i}}$ and $\mathbf{q}_{\mathrm{f}}$ are the initial and final configurations, respectively.

For the trajectory to be valid, the parameter trajectory $\alpha(t)$ must satisfy the following properties.

$$\alpha(0) = 0$$
$$\alpha(T) = 1$$

where T is the duration of the trajectory.

Joint Mode : Velocity Constraints

In addition to the position profile, we can also obtain the velocity profile by differentiating the position, which results in the following expression.

$$\dot{\mathbf{q}}_{\mathrm{r}}(t) = \dot{\alpha}(t) \cdot (\mathbf{q}_{\mathrm{f}} - \mathbf{q}_{\mathrm{i}})$$

At this point, we have many options to constrain the velocity of the profile. One of the simplest approaches is to set the velocity at the initial time and final time to be zero. Hence, the following constraints must be true.

$$\dot{\alpha}(0) = 0$$
$$\dot{\alpha}(T) = 0$$

One can apply the same principle to constrain acceleration. The parameter profile $\alpha(t)$ can be anything as long as it satisfies all the constraints mentioned above.

Linear Mode : Linear Interpolation

Most industrial robots consist of at least 3 degrees of freedom, which allow them to manipulate their end-effector's positions. The word "linear" in "linear mode" refers to the path of the end-effector's position.

Hence, the trajectory profile $\mathbf{p}_\mathrm{r}(t)$ is generated by interpolating between 2 desired position, which can be described as follows.

$$\mathbf{p}_\mathrm{r}(t) = \big(1 - \alpha(t)\big) \cdot \mathbf{p}_\mathrm{i} + \big(\alpha(t)\big) \cdot \mathbf{p}_\mathrm{f}$$
$$\dot{\mathbf{p}}_\mathrm{r}(t) = \dot{\alpha}(t) \cdot (\mathbf{p}_\mathrm{f} - \mathbf{p}_\mathrm{i})$$

where $\mathbf{p}_\mathrm{i}$ and $\mathbf{p}_\mathrm{f}$ are the initial and final positions, respectively.

Note that this is almost the same as joint mode. Therefore, the choice of parameter profile α and the duration T can be determined by the same approach as the one in joint mode.

Linear Mode : Inverse Kinematics

The position trajectory $\mathbf{p}_\mathrm{r}$ is in the taskspace while the trajectory tracking control requires the reference in the joint space. Both inverse position kinematics and inverse velocity kinematics are required to compute the necessary trajectory profile.

$$\mathbf{q}_\mathrm{r}(t) = \mathbf{IPK}\big(\mathbf{p}_\mathrm{r}(t)\big)$$
$$\dot{\mathbf{q}}_\mathrm{r}(t) = \mathbf{IVK}\big(\mathbf{q}_\mathrm{r}(t), \dot{\mathbf{p}}_\mathrm{r}(t)\big)$$

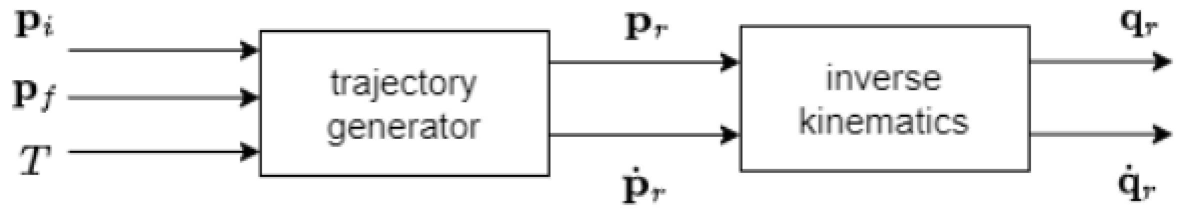
Conclusion on Trajectory Generation

If you are interested in drawing a straight line in taskspace, then the interpolation must be done in the taskspace using the following equation.

$$\mathbf{p}_\mathrm{r}(t) = \left(1-\alpha(t)\right)\cdot\mathbf{p}_\mathrm{i} + \left(\alpha(t)\right)\cdot\mathbf{p}_\mathrm{f}$$
$$\dot{\mathbf{p}}_\mathrm{r}(t) = \dot{\alpha}(t)\cdot(\mathbf{p}_\mathrm{f}-\mathbf{p}_\mathrm{i})$$

After that, the generated trajectory must be converted to joint space.

$$\mathbf{q}_\mathrm{r}(t) = \mathbf{IPK}\left(\mathbf{p}_\mathrm{r}(t)\right)$$
$$\dot{\mathbf{q}}_\mathrm{r}(t) = \mathbf{IVK}\left(\mathbf{q}_\mathrm{r}(t), \dot{\mathbf{p}}_\mathrm{r}(t)\right)$$



Possible Implementation in ROS2

$$\mathbf{p}_\mathrm{r}(t) = \left(1 - \alpha(t)\right) \cdot \mathbf{p}_\mathrm{i} + \left(\alpha(t)\right) \cdot \mathbf{p}_\mathrm{f}$$
$$\dot{\mathbf{p}}_\mathrm{r}(t) = \dot{\alpha}(t) \cdot (\mathbf{p}_\mathrm{f} - \mathbf{p}_\mathrm{i})$$

Subscription: $\langle [\mathbf{p}_\mathrm{i}, \mathbf{p}_\mathrm{f}], T\rangle$ →/via_points

Publisher: $\langle \mathbf{p}_\mathrm{r}, \dot{\mathbf{p}}_\mathrm{r}\rangle$ →/reference/task_states

Timer: publish the trajectory at fixed rate

$$\mathbf{q}_\mathrm{r}(t) = \mathbf{IPK}\left(\mathbf{p}_\mathrm{r}(t)\right)$$
$$\dot{\mathbf{q}}_\mathrm{r}(t) = \mathbf{IVK}\left(\mathbf{q}_\mathrm{r}(t), \dot{\mathbf{p}}_\mathrm{r}(t)\right)$$

Subscription: $\langle \mathbf{p}_\mathrm{r}, \dot{\mathbf{p}}_\mathrm{r}\rangle$ →/reference/task_states

Publisher: $\langle \mathbf{q}_\mathrm{r}, \dot{\mathbf{q}}_\mathrm{r}\rangle$ →/reference/joint_states

Timer: publish the trajectory policy at fixed rate

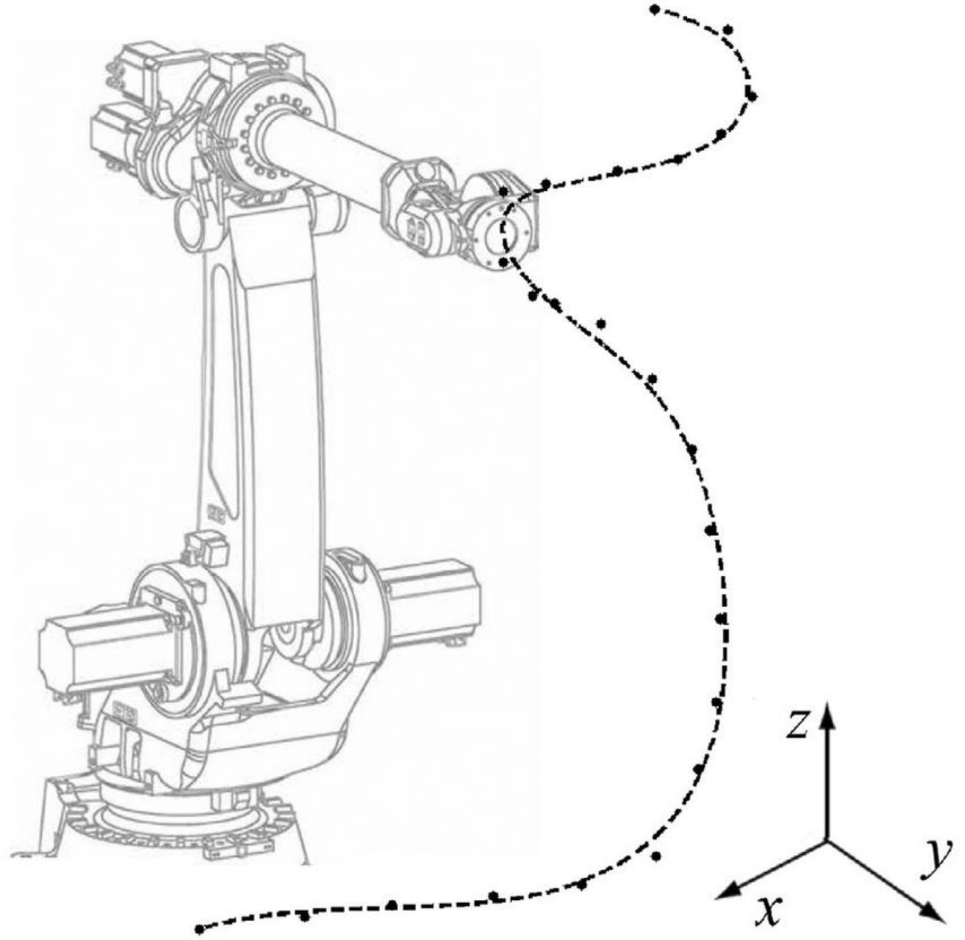
Arguments : “inverse”

Trajectory Scheduling

Scheduling via points

If multiple via points are given, there must be a process that allows the robot to approach each via point accordingly. This process is known as “scheduling”.

Since the change in via points does not occur continuously, a "via point scheduler" must be designed as a discrete-state system. One of the simplest implementation is to use finite state machine.



Proximity Detection

Let enableTracker be a Boolean control signal that enable a trajectory tracker when the value is true, and false otherwise. The idea is to enable the tracker until the robot reaches the last via point.

In addition to enabling the tracker, a scheduler should update a new via point when the current position $\mathbf{p}$ is relatively close to the current via point $\mathbf{p}_\mathrm{f}$. This results in the following guard condition.

$$\|\mathbf{p}_\mathrm{f} - \mathbf{FPK}(\mathbf{q})\| \leq \varepsilon$$

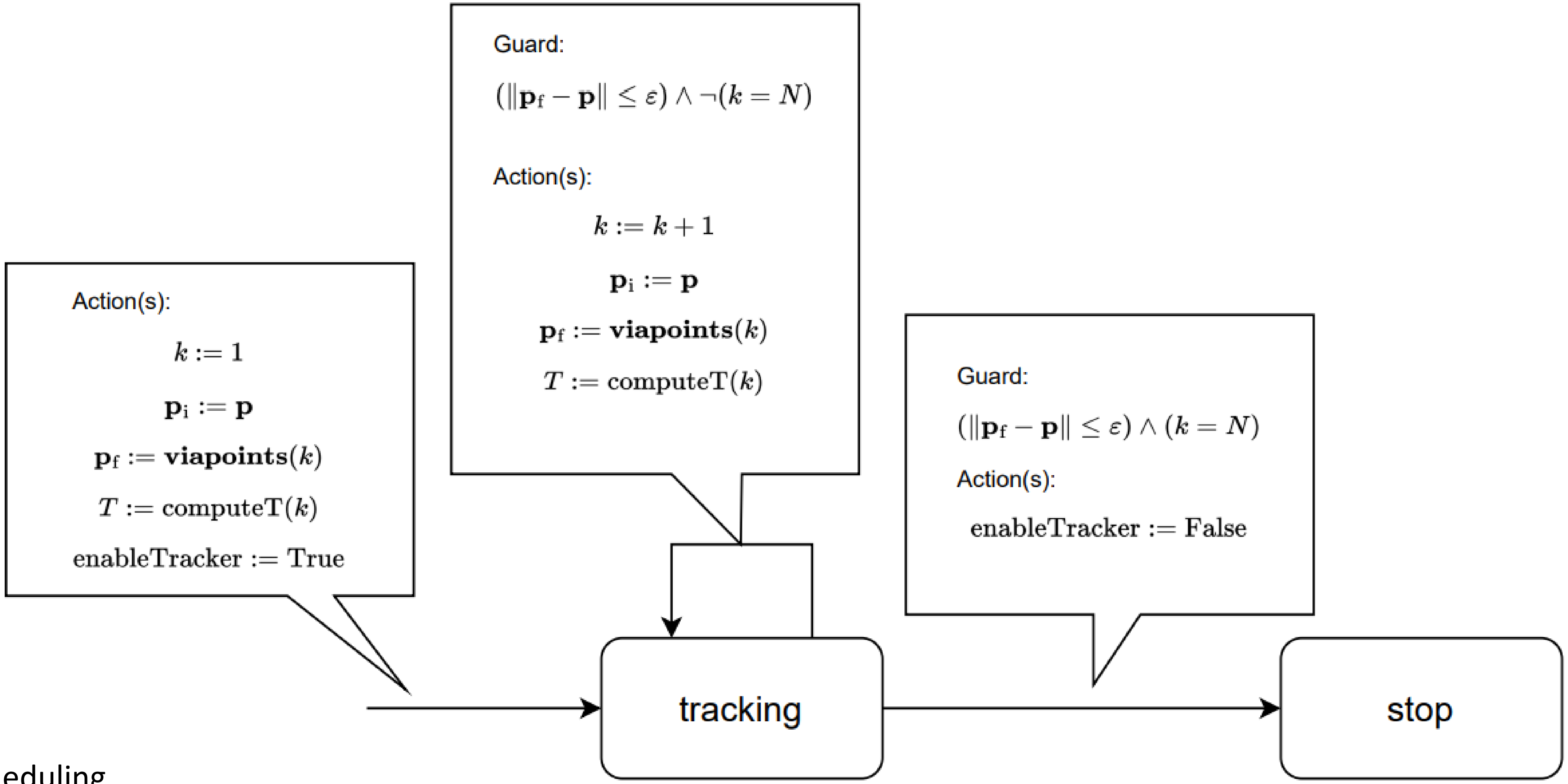
where ε is the positional tolerance.

We may call the above guard condition as “proximity detection” in taskspace.

One can also check proximity in joint space.

$$\|\mathbf{IPK}(\mathbf{p}_\mathrm{f}) - \mathbf{q}\| \leq \varepsilon_q$$

Finite State Machine


Guard:
$(\|\mathbf{p}_\mathrm{f} - \mathbf{p}\| \leq \varepsilon) \land \neg(k = N)$
Action(s):
$k := k + 1$
$\mathbf{p}_\mathrm{i} := \mathbf{p}$
$\mathbf{p}_\mathrm{f} := \mathbf{viapoints}(k)$
$T := \mathrm{computeT}(k)$
Action(s):
$k := 1$
$\mathbf{p}_\mathrm{i} := \mathbf{p}$
$\mathbf{p}_\mathrm{f} := \mathbf{viapoints}(k)$
$T := \mathrm{computeT}(k)$
$\mathrm{enableTracker} := \mathrm{True}$
Guard:
$(\|\mathbf{p}_\mathrm{f} - \mathbf{p}\| \leq \varepsilon) \land (k = N)$
Action(s):
$\mathrm{enableTracker} := \mathrm{False}$
tracking
stop

Conclusion of Trajectory Scheduling

This implies that the schedule requires a list of via points as well as the current position feedback from the system in order to determine an appropriate via points $\mathbf{q}_\mathrm{i}$ and $\mathbf{q}_\mathrm{f}$ and the duration T.

Possible Implementation in ROS2 (mostly pub/sub)

Subscription: $\mathrm{hasReached}$ →/hasReached

Publisher: $\langle [\mathbf{p}_\mathrm{i}, \mathbf{p}_\mathrm{f}], T \rangle$ →/via_points

Service Client: $\mathrm{enableTracker}$ →/enable

Subscription Callback: publish the updated via points when proximity is detected.

YAML config.: lists of via points

$$\|\mathbf{p}_\mathrm{f} - \mathbf{p}\| \leq \varepsilon$$

Subscription:$\mathbf{p}$ →/task_states, $\langle [\mathbf{p}_\mathrm{i}, \mathbf{p}_\mathrm{f}], T \rangle$ →/via_points

Publisher: $\mathrm{hasReached}$ →/hasReached

Timer: check proximity at fixed rate, but only publish when it is detected.

YAML Config.: ε

$$\mathbf{p} = \mathbf{FPK}(\mathbf{q})$$
$$\dot{\mathbf{p}} = \mathbf{FVK}(\mathbf{q}, \dot{\mathbf{q}})$$

Subscription: $\langle \mathbf{q}, \dot{\mathbf{q}} \rangle$ →/joint_states
Publisher:$\langle \mathbf{p}, \dot{\mathbf{p}} \rangle$ →/task_states
Timer: publish the task states at fixed rate
Arguments : “forward”

Possible Implementation in ROS2 (service)

Service Server: $\mathrm{hasReached}$ →/hasReached

Service Client: $\langle [\mathbf{p}_\mathrm{i}, \mathbf{p}_\mathrm{f}], T \rangle$ →/via_points,

$\mathrm{enableTracker}$ →/enable

Server Callback: publish the updated via points when proximity is detected.

YAML config.: lists of via points

$$\|\mathbf{p}_\mathrm{f} - \mathbf{p}\| \leq \varepsilon$$

Subscription:$\mathbf{p}$ →/task_states, $\langle [\mathbf{p}_\mathrm{i}, \mathbf{p}_\mathrm{f}], T \rangle$ →/via_points

Service Client: $\mathrm{hasReached}$ →/hasReached

Timer: check proximity at fixed rate, but only publish when it is detected.

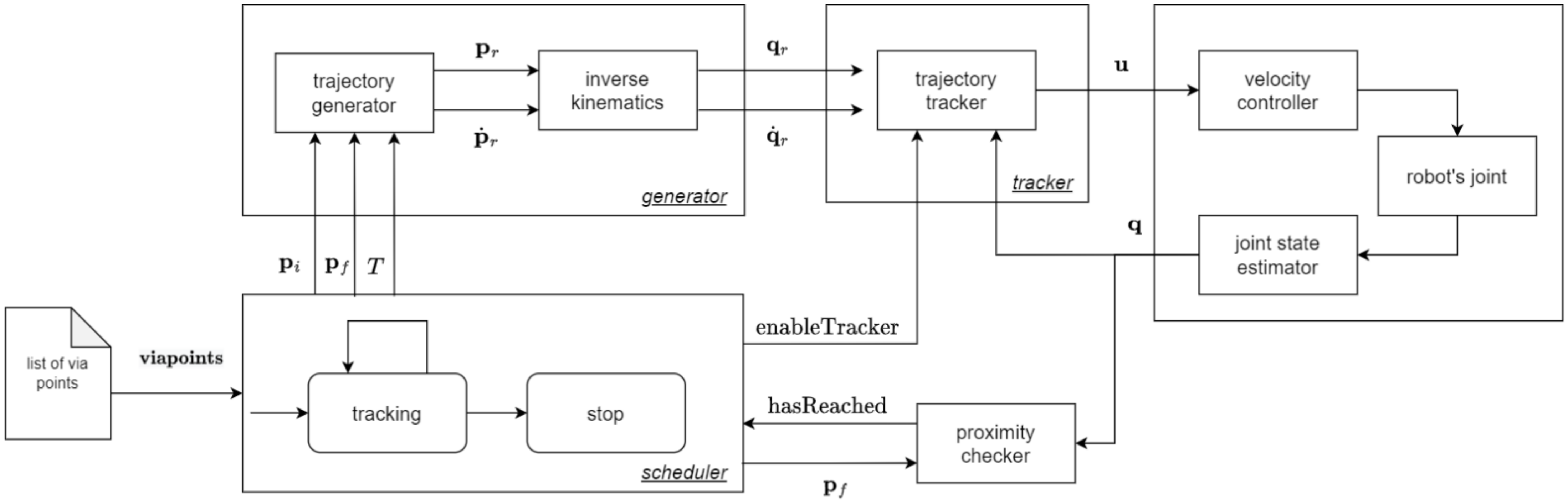
YAML Config.: ε

$$\mathbf{p} = \mathbf{FPK}(\mathbf{q})$$
$$\dot{\mathbf{p}} = \mathbf{FVK}(\mathbf{q}, \dot{\mathbf{q}})$$

Subscription: $\langle \mathbf{q}, \dot{\mathbf{q}} \rangle$ →/joint_states
Publisher:$\langle \mathbf{p}, \dot{\mathbf{p}} \rangle$ →/task_states
Timer: publish the task states at fixed rate
Arguments : “forward”

Overall Architecture

Overall Architecture



Overall Architecture

Implementation

…

Overall Architecture

Implementation in ROS2

… It’s your task to figure this out …

LAB 4 / (60)

Step 1: Setting up velocity controller (5)

In the controller configuration file (yaml), define a new velocity controller. Make sure that its type is "velocity_controllers/JointGroupVelocityController“ In the file, set the parameters to be the following.

- joints:
  - joint_1
  - joint_2
  - joint_3
- command_interface:
  - velocity
- state_interface:
  - velocity
  - effort

Also, in your xacro file, make sure that the command interface is velocity only, and the state interfaces are velocity and effort.

You can verify the set-up by spawning your robot in gazebo. Make sure you spawn "joint_state_broadcaster" and the correct controller. You should see a topic "[controller_name]/commands". You may publish a message to that topic, and the robot in gazebo should move at the given velocity.

Step 2: Developing a tracker (10)

You must create a new package called "[*robot_name*]_control". This is where you put your tracking system. In that package, create a Python executable called "tracker.py" that acts as a node "tracker". The tracker will be enabled when a user calls a service '/enable' (The choice of the interface is up to you).

If the tracker is enabled, the node should constantly publish an enabling message to "[controller_name]/commands" topic at a fixed rate until it becomes disabled.

Once it is disabled, the node should publish a message that corresponds to "stop" to "[controller_name]/commands" topic only once.

The tracker should subscribe '/joint_states' since it is required as a position feedback to the control policy.

The tracker should also subscribe reference position and reference velocity. You must choose an appropriate type for this topic, i.e. name: type:trajectory_msgs/JointTrajectoryPoint.

Do not hard-code the control gain. This should be implemented as ROS parameters in a configuration file called "tracker_config.yaml".

You can preliminarily verify this node by publishing a constant joint position and zero velocity.

Step 3: Modifying kinematics server (10)

Assume that you have functions/methods that performs forward kinematics and inverse kinematics for both position and velocity, modify your kinematics server so that it takes an argument. This argument will dictate whether it is inverse kinematics or forward kinematics.

Also modify the node so that, when the node is forward kinematics, it subscribes to a 'joint_states' topic and publishes the corresponding position to a 'end_effector_states' topic (you may use "sensor_msgs/JointState" as its type), but when the node is inverse kinematics, it subscribes to a 'end_effector_states' topicand publishes the corresponding joint configuration to a topic 'joint_states'.

In this lab, you need 2 kinematics servers. One of them is used for mapping the reference trajectory, and the other one is used as a part of determining proximity between the current position and the current via point.

Step 4: Developing a generator (10)

Create a Python executable called "generator.py" that acts as a node "generator". The generator should publish the reference position and velocity in taskspace to the corresponding topic at a fixed rate.

The generator must be given the information about the initial and final via points as well as the duration of a trajectory. You can design the topics to be anything you want as long as they provide all necessary information. (You may use trajectory_msg/JointTrajectory to capture all information in one message. But this is not mandatory.)

Make sure an "inverse" kinematics server is subscribing to the reference provided by the generator. This kinematics server will publish the corresponding joint states to the tracker.

Step 5: Developing a proximity detector (5)

Create a Python executable called "proximity_detector.py" that acts as a node "proximity_detector". There are several ways to implement a proximity check. You can check the proximity in taskspace or joint space.

If the check is done in taskspace, you need a "forward" kinematics server that subscribes to '/joint_states'. The detector itself will subscribe the position that being published from the server as well as the current via point from the scheduler.

If the check is done in joint space, you need an "inverse" kinematics server that subscribes to the current via point. The detector itself will subscribe the current '/joint_states' and the joint states from the kinematics server.

The detector can publish an empty message or send empty service 'hasReached' in order to trigger an update in the scheduler. This can be implemented as pub/sub or service.

Step 6: Developing a scheduler (10)

Create a Python executable called "scheduler.py" that acts as a node "scheduler". The scheduler obtains a list of via points from a "data" file (should be in .yaml) and publishes 2 via points and a duration.

The scheduler can be triggered by '/hasReached' topic, which will update its internal index accordingly. If the index reaches the end, the scheduler will publish '/enable' as "False".

There should not be any timer in this node since its behavior is strictly event-driven.

Step 7: Developing a generator (10)

```
Create a launch file that starts the following nodes.
    - Gazebo : simulator
    - Gazebo Spawner : spawn the robot in gazebo
    - Controller Manager Spawner : spawn joint_state_broadcaster
    - Controller Manager Spawner : spawn your velocity controller
    - Tracker
    - Inverse Kinematicss Server : Mapping referecne trajectory
    - Generator
    - Forward/Inverse Kinematics Server : Mapping current position
    - Proximity Detector
    - Scheduler
```