Cartesian Interface in ROS Documentation

Lea Steffen, Stefan Scherzinger, Felix Exner

Sep 17, 2020

Contents

CHAPTER 1

Introduction

This document serves as a design documentation to create a Cartesian trajectory definition for ROS. As there have been many proposals in the past, but there is no standardized interface, this document tries to incorporate as many suggestions as possible to derive a common definition useful to as many people as possible.

1.1 Target of this project

This is part of a larger project that tries to achieve Cartesian trajectory execution on Universal Robots robotic arms. Expected outcome of this project is to provide a Cartesian equivalent of the control_msgs/FollowJointTrajectory.action.

Current ROS-based approaches often use motion planing to interpolate between individual Cartesian poses (e.g. MoveIt!, Descartes) or implement Cartesian pose tracking for dynamic targets (e.g. cartesian_controllers).

While these approaches work fine for many applications, there are also use cases that need a Cartesian trajectory approach. Especially in the industrial context, such as for welding or gluing applications, users classically define a tool path in Cartesian space that should be followed precisely.

As most robot vendors offer programming interfaces to define robot motions in Cartesian space it does make sense to also support these interfaces from ROS instead of always taking the detour of joint-based control. Cartesian control on the other hand introduces additional aspects that need consideration, such as resolving ambiguities in joint space that arise for obtaining identical Cartesian poses. A Cartesian trajectory interface needs to cover this as well.

1.2 Contents of this document

This document will start with a summary of existing suggestions for Cartesian interfaces known to us. A conclusion chapter will form a proposed interface taking those interfaces into account.

Additional to that we will have a look at native robot interfaces to get an overview how industrial vendors interface their robots.

1.3 Contribution

We would like to generate an interface suitable for as many people as possible. Therefore, any input is highly welcome! This project is hosted at FZI's github orgnanization

Please get in touch with us and enter the discussion. Either open a new issue if you want to commit on something or even write a Pull Request with a suggestion.

1.4 Requirements

From the motivation above and the shown possible use cases the following requirements are defined for the developed interface

- **Similar to control_msgs/FollowJointTrajectory.action**

  With this proposal we aim to offer Cartesian trajectory execution in terms of trajectories consisting of multiple waypoints, where motion between waypoints is interpolated in Cartesian space. This interface should be similar in use to the standardized joint trajectory interface. Therefore, not only a trajectory representation shall be developed, but also an action interface around it.

- **Include posture definitions**

  As mentioned above, when defining Cartesian poses there might be ambiguities in joint space for that pose. There should be a methodology included that helps resolving these ambiguities. This gives users control over repeatable robot motion.

- **Composable structure**

  Many use cases require tool activation / modification during trajectory execution, for example activating adhesive extrusion or a welding torch. With this in mind, the proposed trajectory interface should be extendible to introduce different aspects of trajectory execution such as adding IO commands to the trajectory. Example:

  ```
  #Trajectory
  TrajectoryPoint[] points
  IOCommand[] io_commands
  ```

- **Transparent error codes**

  When trajectory execution fails users should know the reason for that. With Cartesian motions additional error sources such as an IK solver not finding a solution are relevant and should therefore be included into the trajectory action definition. This has to be further investigated.

1.5 Project limitations

While this document proposes an interface for executing Cartesian trajectories there are a couple of aspects not being discussed inside this design document:

- **Trajectory-IO synchronization**

  While being mentioned earlier IO synchronization is not explicitly covered inside this document. As written, the interface should be designed in a way that it could easily be extended with such a feature, though.

- **Actual trajectory execution / interpolation**

  There are multiple steps involved between a Cartesian trajectory interface and actual motion execution. There are different strategies that can be implemented, where selection of such a strategy highly depends on the actual use case. This will not be part of this interface definition.

- **Trajectory planning (interface)**

  As stated above, this proposal's intention is to create a Cartesian counterpart of trajectory_msgs/JointTrajectoryPoint, trajectory_msgs/JointTrajectory and control_msgs/FollowJointTrajectory.action.

  Therefore, planning and parameterizing trajectories as for example MoveIt!'s computeCartesianPath() function are out of this project's scope.

CHAPTER 2

Existing Cartesian Interfaces in ROS

2.1 rep-I0003

REP for defining Cartesian paths. This is intended to be an interface for a Cartesian path planner, rather than a path execution.

Upstream URL: https://github.com/ros-industrial/rep/blob/master/rep-I0003.rst

2.1.1 Feature list

- Contains industrial primitives such as linear, circular, etc.)
- Uses a segment representation

2.1.2 Features required from hardware / driver

- Control interface
- Cartesian motion primitives

2.1.3 Message definition

```
[CartesianPath]
waypoint[N]
  constraints[K]
    ??
    ref frame
    tcp frame
    process parameters[]
segment[N-1]
  type (linear, joint, circular)
```

(continues on next page)

(continued from previous page)

```
  acceleration
  velocity
```

2.2 Willow Garage (2010)

Part of http://wiki.ros.org/trajectory_msgs/Reviews/Cartesian%20Trajectories_API_Review_2013_06_05

Specified inside http://wiki.ros.org/robot_mechanism_controllers/Reviews/Cartesian%20Trajectory%20Proposal%20API%20Review

2.2.1 Feature list

- Trajectory points defined by pose, twist and posture (joint configuration of joints defined in *posture_joint_names*, It is undecided where the *posture* should go. Into the trajectory points or as a separate field.)
- Contains path and goal tolerances in * absolute position distance * absolute rotation angular error * absolute linear velocity * absolute angular velocity
- *queue* field which is not really clear how it should be used

2.2.2 Features required from hardware / driver

- Control interface

2.2.3 Message definition

```
[CartesianTrajectoryGoal]
CartesianTrajectory trajectory
  Header header  # A stamp of 0 means "execute now"
  PoseStamped tool  # The frame which is being controlled
  string[] posture_joint_names
  CartesianTrajectoryPoint[] points
    duration time_from_start
    Pose pose
    Twist twist
    float64[] posture
  JointTrajectory posture  # For determining the redundancy
    std_msgs/Header header
    string[] joint_names
    trajectory_msgs/JointTrajectoryPoint[] points
      float64[] positions
      float64[] velocities
      float64[] accelerations
      float64[] effort
      duration time_from_start
CartesianTolerance path_tolerance  # Tolerance for aborting the path
  float64 position
  float64 orientation  # Permitted angular error
  float64 velocity
  float64 angular_velocity
```

(continues on next page)

(continued from previous page)

```
CartesianTolerance goal_tolerance  # Tolerance for when reaching the goal is␣
↪considered successful
bool queue
```

```
[CartesianTrajectoryResult]
int32 error_code  # 0 if successful
CartesianTrajectoryPoint cartesian_state
JointTrajectoryPoint joint_state
Twist pose_error
Twist twist_error
CartesianTolerance path_tolerance  # Current tolerance used for the path
```

```
[CartesianTrajectoryError]
int32 SUCCESSFUL = 0
int32 ROOT_TRANSFORM_FAILED
int32 TOOL_TRANSFORM_FAILED
int32 PATH_TOLERANCE_VIOLATED
int32 INVALID_POSTURE
```

2.2.4 Field details

Control (Tool) Frame

The *tool* field describes the control frame for this trajectory. The poses and twists of the trajectory will be applied in this frame, and the tolerances will be measured in this frame. The tool frame should be rigidly attached to the "tip" frame given in the controller configuration; the transform between the two will only be computed once.

Redundancy Resolution

Each cartesian trajectory point contains a posture, which is an array of joint positions for the joints listed in posture_joint_names. The controller attempts to track the posture in the nullspace of the cartesian movement. The posture value for each point is either the given value, or the previous posture value if the array is empty. The posture is linearly interpolated between trajectory points. If the posture array is empty in every point, then the posture is uncontrolled.

Tolerances

Tolerances are specified for the entire trajectory (path_tolerance) and for the success conditions (goal_tolerance). In both, a tolerance of 0 is interpreted as "unspecified", and a default tolerance (such as a parameter to the controller) is used. A tolerace of -1 means "no tolerance" and the corresponding field is ignored when tolerances are checked.

There are two possible ways to handle the path tolerance:

1. Abort if the path tolerance is violated
2. Stall the desired and allow the controller to catch up if the path tolerance is violated.

Option 1 is the most straightforward to implement, but more difficult to use. I'm pretty sure I can implement option 2 by stalling the time used for computing the desired point. I'm considering making this choice a parameter of the controller so the user can choose either behavior.

2.3 Banachowicz

Proposed by Konrad Banachowicz

Part of http://wiki.ros.org/trajectory_msgs/Reviews/Cartesian%20Trajectories_API_Review_2013_06_05

2.3.1 Feature list

- Multiple possible effector names
- Multiple trajectories in one goal (One for each effector?)
- Trajectory points consist of pose and twist
- Contains impedance for each effector with * Center of compliance * stiffness * damping
- Contains path and goal tolerances in * absolute position distance * absolute rotation angular error * absolute linear velocity * absolute angular velocity
- Contains posture information to handle redundancies (for each effector???)
- Contains a nullspace impedance

2.3.2 Features required from hardware / driver

- Control interface
- Impedance control

2.3.3 Message definition

```
[CartesianTrajectoryGoal]
Header header  # A stamp of 0 means "execute now"
string[] effector_names
CartesianTrajectory[] trajectory
  Pose tool  # The frame (offset ?) which is being controlled or it can be treated as␣
↪separate effector ?
  CartesianTrajectoryPoint[] points
    duration time_from_start
    Pose pose
    Twist twist
CartesianImpedance[] impedance
    Pose center_of_compliance # or it can be treated as separate effector ?
    TBD stiffness % cartesian stiffness
    TBD damping % damping ratio
CartesianTolerance[] path_tolerance  # Tolerance for aborting the path
  float64 position
  float64 orientation  # Permitted angular error
  float64 velocity
  float64 angular_velocity
CartesianTolerance[] goal_tolerance  # Tolerance for when reaching the goal is␣
↪considered successful
string[] joint_names
JointTrajectoryPoint[] posture  # For determining the redundancy
JointImpedance[] nullspace_impedance
```

(continues on next page)

(continued from previous page)

```
  float64[] stiffness
  float64[] damping
```

2.4 Banachowicz2

Proposed by Konrad Banachowicz, 2nd version

Part of http://wiki.ros.org/trajectory_msgs/Reviews/Cartesian%20Trajectories_API_Review_2013_06_05

2.4.1 Feature list

- Trajectory points consist of pose and twist
- Contains impedance for each trajectory point
  - target_frame_id
  - stiffness (6D)
  - damping (6D)
- Contains path tolerance in
  - translation
  - orientation
  - twist
  - wrench
- Contains posture information to handle redundancies (for each effector???)
- Contains a nullspace impedance

2.4.2 Features required from hardware / driver

- Control interface
- Impedance control

2.4.3 Message definition

```
[CartesianTrajectory]
Express trajectory of frame target_frame_id in relation to frame header.frame_id .
Header header # stamp - trajectory start time, frame_id - trajectory reference frame
string target_frame_id # target controlled frame
CartesianTrajectoryPoint[] points
  duration time_from_start
  Pose pose
  Twist twist
```

```
[CartesianImpedance]
# Parameters of spring-damper located between frame header.frame_id and frame target_
↪frame_id.
Header header # stamp - trajectory start time, frame_id - spring-damper base frame
string target_frame_id # spring-damper end frame
CartesianImpedancePoint[] points
  duration time_from_start
  CartesianStiffness stiffness
    Vector3 translational
    Vector3 rotational
  CartesianDamping damping
    Vector3 translational
    Vector3 rotational
```

```
[CartesianConstraints]
# Constraint the relation between header.frame_id and target_frame_id.
Header header # stamp - time of constraint activation
string target_frame_id
duration time_from_start # duration of constraint activation
TranslationConstraint translation
OrientationConstraint orientation
Twist twist
Wrench wrench
```

```
[CartesianTrajectoryGoal]
CartesianTrajectory[] trajectory
CartesianImpedance[] impedance
CartesianConstraints[] path_constraints  # Tolerance for aborting the path
JointTrajectory[] posture  # For determining the redundancy
JointImpedance[] nullspace_impedance
```

2.5 ROS-Answers question by arennuit

https://answers.ros.org/question/196954/cartesian-trajectory-description-which-message-type

2.5.1 Feature list

- Contains accelerations

2.5.2 Features required from hardware / driver

- Control interface

2.5.3 Message definition

Note: The source itself does not contain a message definition, so this is an interpretation of the plain text.

```
[CartesianTrajectoryGoal]
string reference_frame # could also come through header
CartesianTrajectoryPoint[] points
```

(continues on next page)

(continued from previous page)

```
  duration time_from_start
  Pose pose
  Twist twist
  Twist acceleration
```

2.6 Pilz

Not used in a message style but implemented in C++ code. Repo: https://github.com/PilzDE/pilz_industrial_motion

2.6.1 Feature list

- Contains accelerations

2.6.2 Features required from hardware / driver

- Control interface

2.6.3 Message definition

```
[CartesianTrajectory]
string group_name # MoveIt! planning group name
string link_name
CartesianTrajectoryPoint[] points
  Duration time_from_start
  Pose pose
  Twist velocity
  Twist acceleration
```

2.7 Descartes Trajectory

- http://wiki.ros.org/descartes_trajectory
- https://github.com/ros-industrial-consortium/descartes/tree/melodic-devel/descartes_trajectory/include/descartes_trajectory
- https://github.com/ros-industrial/rep/blob/master/rep-I0003.rst

2.7.1 Feature list

- Contains reference implementations for path sepcification and interface for more customized methods
- Offers three implementations for Trajectory Points
  - Joint point [JointTrajectoryPt]: Represents a robot joint pose. It can accept tolerances for each joint
  - Cartesian point [CartTrajectoryPt]: Defines the position and orientation of the tool relative to a world coordinate frame. It can also apply tolerances for the relevant variables that determine the tool pose.

  - AxialSymmetric point (5DOF) [AxialSymmetricPt]: Extends the CartTrajectoryPt by specifying a free axis of rotation for the tool. Useful whenever the orientation about the tool's approach vector doesn't have to be defined.

- Hybrid Trajectories possible
- All trajectory points take an optional TimingConstraint

2.7.2 Features required from hardware / driver

2.7.3 Message definition

```
[JointTrajectoryPt]
TolerancedJointValue joints
Frame tool
Frame wobj
TimingConstraint timing



[CartTrajectoryPt]
Frame wobj_base
TolerancedFrame wobj_pt
Frame tool_base
TolerancedFrame tool_pt
double pos_increment
double orient_increment
TimingConstraint timing



[AxialSymmetricPt]
double x
double y
double z
double rx
double ry
double rz
double orient_increment
FreeAxis axis
TimingConstraint timing
```

2.8 MoveIt!

Implemented in moveit_msgs: https://github.com/ros-planning/moveit_msgs/blob/master/msg/CartesianTrajectory.msg

2.8.1 Feature list

- Contains accelerations

2.8.2 Features required from hardware / driver

- Control interface

2.8.3 Message definition

```
[CartesianTrajectory]
Header header

# The name of the Cartesian frame being tracked with respect to the base frame␣
↪provided in header.frame_id
string tracked_frame

CartesianTrajectoryPoint[] points
  CartesianPoint point
    geometry_msgs/Pose pose
    geometry_msgs/Twist velocity
    geometry_msgs/Accel acceleration
  duration time_from_start
```

2.9 johnmichaloski

Upstream URL: https://github.com/johnmichaloski/ROS/blob/master/nistfanuc_ws/src/cartesian_trajectory_msg/msg/CartesianTrajectoryGoal.msg

This is very similar to *Willow Garage (2010)*.

2.9.1 Feature list

- Trajectory points defined by pose, twist and posture (joint configuration of joints defined in *posture_joint_names*, It is undecided where the *posture* should go. Into the trajectory points or as a separate field.)
- Contains path and goal tolerances in * absolute position distance * absolute rotation angular error * absolute linear velocity * absolute angular velocity
- *queue* field which is not really clear how it should be used

2.9.2 Features required from hardware / driver

- Control interface

2.9.3 Message definition

```
[CartesianTrajectoryGoal]
std_msgs/Header header  # A stamp of 0 means "execute now"
geometry_msgs/PoseStamped tool  # The frame which is being controlled
std_msgs/String[] posture_joint_names
CartesianTrajectoryPoint[] points
  std_msgs/Duration time_from_start
  geometry_msgs/Pose pose
```

(continues on next page)

(continued from previous page)

```
  geometry_msgs/Twist twist
  std_msgs/Float64[] posture
  std_msgs/Float64 velocity
  std_msgs/Float64 acceleration
  std_msgs/Float64 jerk
std_msgs/Duration time_from_start
geometry_msgs/Pose pose
geometry_msgs/Twist twist
std_msgs/Float64[] posture
#trajectory_msgs/JointTrajectory posture  # For determining the redundancy

CartesianTolerance path_tolerance  # Tolerance for aborting the path
  float64 position
  float64 orientation  # Permitted angular error
  float64 velocity
  float64 angular_velocity
CartesianTolerance goal_tolerance  # Tolerance for when reaching the goal is␣
↪considered successful
  float64 position
  float64 orientation  # Permitted angular error
  float64 velocity
  float64 angular_velocity
std_msgs/Bool queue
```

```
[CartesianTrajectoryResult]
std_msgs/Int32 error_code  # 0 if successful
CartesianTrajectoryPoint cartesian_state
trajectory_msgs/JointTrajectoryPoint joint_state
geometry_msgs/Twist pose_error
geometry_msgs/Twist twist_error
CartesianTolerance path_tolerance  # Current tolerance used for the path
```

```
[CartesianTrajectoryError]
int32 SUCCESSFUL = 0
int32 ROOT_TRANSFORM_FAILED
int32 TOOL_TRANSFORM_FAILED
int32 PATH_TOLERANCE_VIOLATED
int32 INVALID_POSTURE
```

2.10 FZI

Our internal Cartesian trajectory definition.

This is bit different to the other approaches as it doesn't require the user to specify detailed timing and / or velocity information in each waypoint. Instead, a maximum velocity and acceleration can be specified in both, translation and rotation domain.

2.10.1 Feature list

- Global max velocities and accelerations instead of timing information
- Contains interpolation distance
- Contains offsets to shift the trajectory in Cartesian space

2.10.2 Features required from hardware / driver

- Control interface
- Trajectory interpolation
- Calculate timings and velocities / accelerations in waypoints

2.10.3 Message definition

```
[CartesianPoseTrajectoryGoal]
fzi_manipulation_msgs/CartesianExecutionConfig params
  string reference_frame
  string endeffector_frame
  float64 acceleration_lin
  float64 acceleration_rot
  float64 velocity_lin
  float64 velocity_rot
  float64 interpolation_lin
  float64 interpolation_rot
  geometry_msgs/Pose trajectory_offset
  bool has_trajectory_offset
  geometry_msgs/Pose tcp_offset
  bool has_tcp_offset
geometry_msgs/Pose[] points
```

2.10.4 Field details

Interpolation

The used interpolation distances are currently used to create setpoints between the ones specified to get closer to the desired Cartesian linear motion when this is being executed using a joint-based backend, e.g. by using an inverse kinematics solver for each setpoint and creating a joint-based trajectory out of that.

Trajectory Offset

The trajectory offset offsets the trajectory relative to the specified **reference_frame**. This can be useful in situations where a trajectory shall be executed in contact with a surface but it should be executed a couple of centimeters above the surface for testing purposes. This is equivalent to applying the specified transformation on the **reference_frame** before executing the trajectory.

TCP Offset

The tcp offset shifts the trajectory relative to the specified **endeffector_frame**. This is useful for example if a trajectory is specified directly ontop of an object's surface, but the endeffector should keep a distance to the surface at all times, e.g. to make room for a gluing layer being applied. This is equivalent to applying the specified transformation on the **endeffector_frame** before executing the trajectory.

2.11 Gijs van der Hoorn

https://github.com/gavanderhoorn/common_msgs/pull/1/files

2.11.1 Feature list

- Contains accelerations
- Contains wrenches
- Contains posture information to handle redundancies

2.11.2 Features required from hardware / driver

- Control interface

2.11.3 Message definition

```
[LinearTrajectory]
std_msgs/Header header
string child_frame_id
LinearTrajectoryPoint[] points
  Duration time_from_start
  Pose pose
  Twist velocity
  Accel acceleration
  Wrench wrench
  sensor_msgs/JointState configuration # optional
```

2.11.4 Field details

The following list is not complete. Please see the linked PR for more information.

header.frame_id

The trajectory describes the motion of this frame relative to *header.frame_id*.

child_frame_id

Each point in the trajectory specifies at least a valid pose and a time at which that pose must be reached by *child_frame_id*.

time_from_start

Time (in seconds) at which the system state encoded in this trajectory point is to be attained, relative to the start of trajectory execution.

Required field.

It is an error to not initialise this field.

configuration

Preferred joint space configuration to achieve *pose*.

Optional field.

This field is encoded as a *sensor_msgs/JointState* to allow for the greatest flexibility when describing joint configurations (compared to bitmasks or lists of booleans/integers).

Only the *name* and *position* fields of the JointState message are used. Values in other fields are ignored.

Joints present in the system, but for which no values are provided will be considered unrestricted wrt possible IK solutions (ie: so called 'free' joints).

Leaving this field empty in case of kinematic systems for which multiple IK solutions exist for a given pose, in general or because of their by-design underconstraint nature (ie: in case of (hyper-)redundancy), could lead to motion discontinuities (ie: mid-motion configuration changes) as IK solvers may not provide the closest or preferred solutions automatically for those types of systems.

Example of a 6D kinematic configuration with all joints given values:

```
configuration.name = ['joint_1', 'joint_2', ..., 'joint_6']
configuration.position = [0.1092, 0.4012, ..., 1.6323]
```

Drivers moving robots to this trajectory point are required to use an IK solution either identical to or as close as possible to this joint configuration.

Encoding a "configuration bitmask" (ie: shoulder, elbow, wrist) typically used in industrial robotics into a 'configuration' field may be done as follows:

```
configuration.name = ['joint_2', 'joint_3', 'joint_4']
configuration.position = [0.0, 0.0, 3.1415]
```

Note: if only 'joint_4' would have been specified, the shoulder and elbow joints would not have been constrained, leading to potentially different solutions being used.

Also note: as *configuration* encodes joint angles instead of binary states, turn numbers and configuration flags can be expressed as a single joint angle.

2.12 Rethink Robotics Intera SDK

Cartesian trajectories for Rethink robots

Table 1: Vendor specifics

Teach pendant	Integrated into the robot
Programming / simulation software	Intera 5
Software	Intera SDK
Programming language	Python
Relevant hardware	Robots Sawyer and Baxxter

Rethink robots support a native ROS interface. The ROS master runs on the robot.

Further reading

- ros_github
- ros_manual

- wiki

2.12.1 Feature list

- Contains accelerations
- For Cartesian segments, the joint positions are used as nullspace biases
- Joint interpolation or cartesian interpolation
- Hybrid Trajectories not possible
- All trajectory points take an optional WaypointOptions
- Contains path and goal tolerances in joint space
- Contains for each waypoint: * Max joint speed ratio * Max joint acceleration * Max linear speed * Max linear acceleration * Max rotational speed * Max rotational acceleration * Corner distance
- Allows to specify active endpoint

2.12.2 Features required from hardware / driver

2.12.3 Message definition

Trajectories have some global options and a list of waypoint. Each waypoint itself can also have some local options.

```
[Waypoint]
# Representation of a waypoint used by the motion controller

# Desired joint positions
# For Cartesian segments, the joint positions are used as nullspace biases
float64[] joint_positions

# Name of the endpoint that is currently active
string active_endpoint

# Cartesian pose
# This is not used in trajectories using joint interpolation
geometry_msgs/PoseStamped pose

# Waypoint specific options
# Default values will be used if not set
# All waypoint options are applied to the segment moving to that waypoint
WaypointOptions options
```

```
[WaypointOptions]
# Optional waypoint label
string label

# Ratio of max allowed joint speed : max planned joint speed (from 0.0 to 1.0)
float64 max_joint_speed_ratio

# Slowdown heuristic is triggered if tracking error exceeds tolerances - radians
float64[] joint_tolerances

# Maximum accelerations for each joint (only for joint paths) - rad/s^2.
```

(continues on next page)

(continued from previous page)

```
float64[] max_joint_accel

############################################################
# The remaining parameters only apply to Cartesian paths

# Maximum linear speed of endpoint - m/s
float64 max_linear_speed

# Maximum linear acceleration of endpoint - m/s^2
float64 max_linear_accel

# Maximum rotational speed of endpoint - rad/s
float64 max_rotational_speed

# Maximum rotational acceleration of endpoint - rad/s^2
float64 max_rotational_accel

# Used for smoothing corners for continuous motion - m
# The distance from the waypoint to where the curve starts while blending from
# one straight line segment to the next.
# Larger distance:  trajectory passes farther from the waypoint at a higher speed
# Smaller distance:  trajectory passes closer to the waypoint at a lower speed
# Zero distance:  trajectory passes through the waypoint at zero speed
float64 corner_distance
```

```
[Trajectory]
# Representation of a trajectory used by the engine and motion controller.

# optional label
string label

# Array of joint names that correspond to the waypoint joint_positions
string[] joint_names

# Array of waypoints that comprise the trajectory
Waypoint[] waypoints

# Trajectory level options
TrajectoryOptions trajectory_options
```

```
[TrajectoryOptions]
# Trajectory interpolation type
string CARTESIAN=CARTESIAN
string JOINT=JOINT
string interpolation_type

# True if the trajectory uses interaction control, false for position control.
bool interaction_control

# Interaction control parameters
intera_core_msgs/InteractionControlCommand interaction_params

# Allow small joint adjustments at the beginning of Cartesian trajectories.
# Set to false for 'small' motions.
bool nso_start_offset_allowed
```

(continues on next page)

(continued from previous page)

```
# Check the offset at the end of a Cartesian trajectory from the final waypoint␣
↪nullspace goal.
bool nso_check_end_offset

# Options for the tracking controller:
TrackingOptions tracking_options

# Desired trajectory end time, ROS timestamp
time end_time

# The rate in seconds that the path is interpolated and returned back to the user
# No interpolation will happen if set to zero
float64 path_interpolation_step
```

```
[TrackingOptions]
# Minimum trajectory tracking time rate:  (default = less than one)
bool     use_min_time_rate
float64  min_time_rate

# Maximum trajectory tracking time rate:  (1.0 = real-time = default)
bool     use_max_time_rate
float64  max_time_rate

# Angular error tolerance at final point on trajectory (rad)
float64[] goal_joint_tolerance

# Time for the controller to settle within joint tolerances at the goal (sec)
bool     use_settling_time_at_goal
float64  settling_time_at_goal
```

CHAPTER 3

Vendor interfaces for Cartesian motions

In section *Existing Cartesian Interfaces in ROS* we had a closer look at existing Cartesian interface definitions and suggestions. While most of them showed a great similarity to existing joint-based interfaces, there's one that sticks out. *rep-I0003* suggests an interface that is closer to how most robot manufacturers define their own interfaces. To pickup this thought, we had a closer look at different vendor interfaces to check for similarities.

While existing interfaces such as the control_msgs/FollowJointTrajectory.action are widely accepted inside the ROS community it requires quite some effort to be filled correctly. Users have to provide positions, velocities and timings for each waypoint significantly affecting the path the robot is following between waypoints. To handle this, users usually leverage advanced tools such as MoveIt! to fill in all these constraints while giving away full control of the actual path the robot's joints and end effector will take.

Most robot manufacturers offer programming interfaces that focus more on point-to-point motions and how interpolation between those waypoints is done. Some of them already expose their full motion command set to ROS, e.g. *Doosan*. Having a closer look at how different manufacturers handle motion commands and how they are interfaced, obviously makes sense in order to create a standardized ROS interface for defining trajectories in such a way.

As a result of this we would like to extend *rep-I0003* to not only offering such an interface at planning level but also as a direct control interface for industrial robots.

3.1 KUKA

Cartesian trajectories for the KUKA robots (KRC/KRL).

Table 1: Vendor specifics

Teach pendant	"KCP" (KUKA Control Panel) or smartPAD
Programming / simulation software	OrangeEdit editor / KUKA simulator Sim Pro
Software	KUKA System Software (KSS)
User interface	KUKA smartHMI (smart Human-Machine Interface)
Programming language	KRL (KUKA Robot Language)
Relevant hardware	KR C2 / KR C3 / KR C4 and probably others

Further reading

- manual_collection
- manual_slides
- manual_advanced

3.1.1 Trajectory composition

Cartesian trajectories can be composed in three ways (see manual_slides p. 23-32).:

- **Linear Cartesian motions LIN**

```
[LIN]
$VEL.CP = 0.5
PTP Start point
LIN End point
```

- **Circular motions CIRC**

```
[CIRC]
$VEL.CP = 0.5
PTP Start point
CIRC Auxiliary point , Endpoint, CA Angle
```

- **Joint space interpolation PTP**
  - joint space movement to a given goal, which can be specified in joint space or in Cartesian space.
  - controller calculates the necessary angle differences for each axis
  - Preferred motion if a high TCP speed is wanted and the interpolation between both waypoints doesn't have to follow a predefined path.

```
[PTP]
$VEL.CP = 0.5
PTP Start point
PTP Auxiliary point C_PTP
PTP End point
```

- **Additional SLIN**
  - The *Spline Linear* motion uses splines between linear motions

  SCIRC
  - The *Spline Circular* motion uses splines between circular motions

  SPTP
  - The *Spline Point to Point* motion is similar to PTP but it allows continuous spline motions.

3.1.2 Waypoint representation

(see kuka_system_software and manual_slides)

- Linear

```
[LIN]
X 1000.00
Y 0.00
Z 1000.00
A 90.00
B 0.00
C 90.00
```

- Circular

```
[CIRC]
P1[]
  X 1000.00
  Y 1.00
  Z 1000.00
  A 90.00
  B 0.00
  C 90.00
P2[]
  X 1000.00
  Y -1.00
  Z 1000.00
  A 90.00
  B 0.00
  C 90.00
CA 180
```

- Point 2 Point

```
[PTP]
POS []
  X 1000.00
  Y 0.00
  Z 1000.00
  A 90.00
  B 0.00
  C 90.00
  S 6
  T 50
```

```
[PTP]
AXIS []
  A1 0
  A2 -90
  A3 90
  A4 90
  A5 0
  A6 -180
```

- Spline

```
[SPLINE]
SPL
  X 102
  Y 1
SPL
  X 104
```

(continues on next page)

(continued from previous page)

```
  Y 0
SPL
  X 204
  Y 0
```

- Angles of rotation of the robot coordinate systems
- S and T specify a robot's position unambiguously if more than one axis position is possible for the same point in space (because of kinematic singularities). This is often written in integer form, thus the values above.
  - **S (status):** 3-bit binary value describing the robot's configuration with predefined criteria
  - **T (turn):** direction of a turn. 6-bit binary value, containing flip bits for each axis (0 when axis >= 0 deg, 1 when axis < 0 deg)

Angle	rotation axis
A	Z
B	Y
C	X

3.1.3 Trajectory parameterization and execution

(see manual_advanced)

Specification of velocity

- Speed of TCP can be set within a move instructions in % by the 'vel' argument.
- For Continuous path motions ([LIN], [CIRC]) the velocity is constant from start to end.
- Realtive Joint Velocity can be set by: *setJointVelocityRel(0.3)*
- KUKA operation mode influence velocity

Mode	description	velocity
T1	Manual Reduced Velocity	max of 250mm/s
T2	Manual High Velocity	as programmed
AUT	Automatic	as programmed
EXT	Automatic external	as programmed
CPR	Safe Operation	max of 250mm/s

specification of acceleration

Relative Joint Acceleration can be set by: *setJointAccelerationRel(0.5)*

Blending

(source Angerer and *Vistein*)

- Blending is enabled by the *advance run mechanism* enabling planning the next motion while executing a motion.
- To activate blending a motion needs to be marked as blendable by adding a keyword to the motion instruction. *C PTP'for PTP motions and 'C_DIS*, *C_VEL* or *C_ORI* for motions in operation space.

- Blending between all motion types is supported. It is even possible to blend a PTP (joint space) into a LIN (Cartesian space) and vice versa.
- Blending can be done by defining a blend radius
  - as a relative value: *IMotion.setBlendingRel(0.2)*
  - in millimeters: *IMotion.setBlendingCart(20)*

Parallel IO operations

No information found so far

Online (real-time) trajectory modifications

Robot Sensor Interface (RSI) (see RobotSensorInterface)

- supported since KRC-4 controller
- influence the position of the robot by external sensors.
- robot position can be influenced by external sensors through overlaying a programmed motion with external control, like position correction from a sensor-based system
- default 4 ms cycle time for accepting set point, hence external controller requires hard real-time
- usually correction data is provided in relative values and applied directly to the running program. However, as absolute values are possible, the robot can be controlled externally while a KRL program only providing a fixed start position runs in the background.
- communication between KUKA and external controller via UDP/IP on a dedicated network segment
- *RSI context* is a library with RSI objects for configuration of the signal flow
- *RSI monitor* offers online a visualization of the RSI signals.

3.2 Universal Robots (UR)

Cartesian trajectories for Universal Robots (CB3 / e-Series).

Table 2: Vendor specifics

Teach pendant	UR+ / URcaps
Programming / simulation software	
Software	
User interface	PolyScope
Programming language	UR Script (similar to Python)
Relevant hardware	CB3 (UR3, UR5, UR10), e-Series (UR3e, UR5e, UR10e, UR16e)

Further reading

- manual_collection
- conveyor_tracking
- dynamic_force_control

3.2.1 Trajectory composition

Programming is done with move instructions (movement types) that move the robot to specified targets.

- **Linear Cartesian motions MoveL**
  - tool moves in a straight line. To keep moving linearly between waypoints each joint performs a more complicated motion.
  - parameters (pose, a=1.2, v=0.25, t=0, r=0):
    - pose: target pose
    - a: tool acceleration in m/s^2
    - v: tool speed in m/s
    - t: time in s
    - r: blend radius in m
- **Circular motions MoveC**
  - tool moves on a circular arc segment to from current pose to target pose. Path point *pose_via* defines the arch's shape
  - parameters (pose_via, pose_to, a=1.2, v=0.25, r=0, mode=0):
    - pose_via: path point
    - pose_to: target pose
    - a: tool acceleration in m/s^2
    - v: tool speed in m/s
    - r: blend radius (of target pose) in m
    - mode: (0: Unconstrained / 1: Fixed mode)
- **Joint space interpolation MoveJ**
  - tool moves in a curved path interpolated in joint space. Each joint reaches location simultaneously. Preferred motion if a high TCP speed is desired.
  - parameters (q, a=1.4, v=1.05, t=0, r=0):
    - q: joint positions
    - a: joint acceleration of leading axis in rad/s^2
    - v: joint speed of leading axisin rad/s
    - t: time in s
    - r: blend radius in m
- **Additional MoveP**
  - tool moves linearly with constant speed with circular blends. Command can be extendend by a Circle move consisting of two waypoints.
  - parameters (pose, a=1.2, v=0.25, r=0):
    - pose: target pose
    - a: tool acceleration in m/s^2
    - v: tool speed in m/s

* r: blend radius in m

3.2.2 Waypoint representation

Individually taught points have the following representation:

```
X
Y
Z
Rx (roll)
Ry (pitch)
Rz (yaw)
```

3.2.3 Trajectory parameterization and execution

Specification of velocity

The robot's speed is defined in form of an argument by the move command.

- For *MoveJ* in deg/s: **maximum joint speed**
- For *MoveL* in mm/s: **desired tool speed**

Global specification of velocity are done separately for joint and TCP speed. The limits, which depend on the robot version, are stated in the table below. The actual speed limits also depend on the robot configuration.

Function	Description	Limit
Joint speed	Max. angular joint speed	180 ○/s[1]
TCP speed	Max. speed of the robot TCP	5000 mm/s

Specification of acceleration

The acceleration of the robot's motions is defined in form of an argument by the move command. Depending on the chosen movement type either the joints' or TCP's acceleration is definable.

- For *MoveJ* in deg/s^2: **joint acceleration**
- For *MoveL* in mm/s^2: **tool acceleration**

Blending

- Circular blending is part of **MoveP**. The blend radius' size is by default a shared value between all the waypoints. A smaller blend radius leads to sharper and a biger radius to smoother paths.
- Blending can also be done by defining a blend radius for waypoints. In this case the trajectory blends around the waypoint, allowing the robot arm not to stop at the point.

Parallel IO operations

Can be triggered at certain points in the robot's path

[1] Wrist joints of UR3 have max. angular speed of 360°/s and shoulder joints of UR10 have max angular speed of 120 °/s.

Online (real-time) trajectory modifications

- path offset
  - a robot motion can be superimposed with a Cartesian offset
  - Cartesian path offset is specified by the script function *path_offset_set(offset, type)*
    - offset: Pose specifying the translational and rotational offset
    - type: Specifies which coordinates to apply (*BASE*,'TCP', *MOTION*, *BASE_INVERTED*)
  - possible applications:
    - imposing a weaving motion onto a welding task
    - compensating for moving the robot base while following a trajectory
- dynamic force control (see dynamic_force_control)
  - provides control of the force parameters dynamically at runtime
  - function to set robot to force mode: *force_mode(task_frame, selection_vector, wrench, type, limits)*
- conveyor tracking (see conveyor_tracking)
  - adjusts a robot's trajectory to a moving conveyor
  - available for linear and circular conveyors
  - CB3 and e-Series controller can decode signals at up to 40kHz

3.3 Fanuc

Cartesian trajectories for Fanuc robots

Teach pendant	FANUC iPendant touch
Programming / simulation software	ROBOGUIDE
Software	
User interface	Fanuc iHMI (Intelligent Human Machine Interface)
Programming language	FANUC Karel (derived from Pascal)
Relevant hardware	R-30iA or R-J3iC (controller)

Further reading

- manual_collection
- manual_slides
- roboguide_help
- reference_manual
- Karel

3.3.1 Trajectory composition

Programming is done with move instructions (robot movement types). (see manual_slides p. 15-16):

- **Linear Cartesian motions LINEAR**: controlled movement of the TCP in a straight line from position A to B

- **Circular motions CIRCULAR**: The TCP follows a circular arc from the initial position to the destination
- **Joint space interpolation JOINT**: basic robot motion with nonlinear toolpath. Tool speed is determined with % of the maximum speed.

3.3.2 Waypoint representation

Points are described with position coordinates x,y, z and rotations w, p, r.

```
x
y
z
w (x-axis rotation)
p (y-axis rotation)
r (z-axis rotation)
```

3.3.3 Trajectory parameterization and execution

(see reference_manual)

Specification of velocity

As motions are initiated and controlled in TP the user can only adapt the robot's motion speed with TP. System configurations and overrides influence the velocity additionally.

specification of acceleration

Specification of acceleration can be done via the following variables:

- acceleration time is fixe and proportional to the programmed speed.
- **$USEMAXACCEL**: enables 'fast acceleration' feature

Blending

Taught positions can either be fly-by points, or stop points:

- **FINE**: motion stops robot arm briefly at each way point
- **CNT** (continuous): robot approaches to the point with a distance specified by the CNT value without ever actually reaching the point, so the robots arm moves in a continuous trajectory
- **CR** (corner radius): like CNT, but specifying a radius for corner rounding allows to precisely define the shape of the blended motion

Parallel IO operations

No information found so far

Online (real-time) trajectory modifications

Dynamic Path Modifier (DPM)

- dynamic path modification using sensor data, so robot's path can be adapted in real-time
- an external sensor provides postion and orientation offset for the next destination
- applicable to multiple groups
- possible applications:
  - Weave operations
  - Stationary tracking
  - Orientation control

J519 (Stream Motion)

- **external protocol for:**
  - path trajectory planning
  - near-real time streaming of the path trajectory to the robot
  - enabling highly flexible and dynamic applications

R912 (Remote Motion Interface)

- drip-feed for TP programs

3.4 Doosan

Cartesian trajectories for Doosan robots

Table 3: Vendor specifics

Teach pendant	Teach Pendant
Programming / simulation software	Teach Pendant + DART platform on user PC
Software	FlexPendant SDK, Microsoft CE + Visual Studio
Programming language	DRL
Relevant hardware	Robots M0609, M1509, M1013 and M0617

Doosan robots support a ROS interface and wrap many of the DRL functionalities with ROS services.

Further reading

- ros_github
- ros_manual
- manuals after free registration

3.4.1 Trajectory composition

The teaching of waypoints is done either in jog-operation or hand-guiding operation. Doosan programming features a combination of basic instructions with a skill-based task composition.

- **Linear Cartesian motions movel**
  - Move tool linearly to a specified target.

  - Possible arguments: point, velocity, acceleration, time, blending radius, and others

- **Circular motions movec**
  - Move in an arc via a point to a target point.
  - Possible arguments: point, point, velocity, acceleration, time, blending radius, and others
- **Joint space interpolation movej**
  - Move to the specified joint position.
  - Possible arguments: target joint angles, velocity, acceleration, time, and others. Note that this command uses the different target type **posj**.

  movejx
  - Move to the specified point with joint interpolation. Similar to **movel**, but without the guarantee of a linear motion result in Cartesian space.
  - Possible arguments: point, velocity, acceleration, time, radius for blending, and others. Additionally, users specify the solution space with a three-bit flag, representing shoulder (lefty vs righty), elbow (below vs above) and wrist (flip vs no flip).

  movesj
  - Move along a spline curve path with joint interpolation, connecting various joint-based waypoints.
  - Possible arguments: list of joint positions, velocity, acceleration, time, and others.
- **Additional movesx**
  - Move along a spline curve from the current point to the target via waypoints.
  - Possible arguments: List of points, velocity, acceleration, time, and others.

  moveb
  - Move along a list of path segments (lines, circles) with constant velocity. Segments are blended.
  - Possible arguments: list of points, velocity, acceleration, time, and others.

  move_spiral
  - Motion along a spiral trajectory on a plane, which is perpendicular to a specified axis.
  - Possible arguments: Revolutions, final spiral radius, and others

  move_periodic
  - Sine-based motion per axis.
  - Possible arguments: Amplitude, period, and others

All move commands have an asynchronous variant, e.g. **amovel** corresponds to **movel**, that allows the user to run other commands in parallel, i.e. the main thread continues executing instructions. The blending parameter is not available for these asynchronous move commands. Triggering concurrent motion commands is caught with errors.

3.4.2 Waypoint representation

Individually taught points (type **posx**) have the following field representation:

```
x
y
z
w (z-direction rotation of reference coordinate system)
p (y-direction rotation of w rotated coordinate system)
r (z-direction rotation of w and p rotated coordinate system)
```

Individual points can be saved in various reference frames.

3.4.3 Trajectory parameterization and execution

Specification of velocity

- In form of a speed argument to move instructions. The speed is valid until the next point
- Global adjustments of task space velocity with the **set_velx** function. This value will be used as default for **movel, movec** and **movesx** if nothing is specified.
- Global, trajectory-wide setting with **change_operation_speed** as a percentage of the current speed setting

Specification of acceleration

- In form of a max acceleration argument to move instructions.
- Global adjustments of task space acceleration with **set_accx**. This is also taken as default for the move commands **movel, movec** and **movesx**.

Blending

- Can be started and stopped with **begin_blend** and **end_blend**, respectively. Upon activation, all points get blended during execution.
- Alternatively, segments can be executed with **moveb**, which is a constant velocity blending motion for a path of given move segments.

Parallel IO operations

- I/O operations are managed independently of trajectory execution
- Users can trigger them e.g. with the asynchronous move instructions for individual segments.

Online (real-time) trajectory modifications

- Supports compliant trajectory execution, in which preference is given to force control over motion control for in-contact tasks
- Trajectories can be modified with the threaded **alter_motion** function with a cycle time of 100ms.

3.5 Franka Emika

Cartesian trajectories for Franka Panda

Table 4: Vendor specifics

Programming / simulation software	Franka Desk, ROS
Software	Franka Control Interface (FCI)
User interface	Franka Desk
Programming language	C++
Relevant hardware	Panda with libfranka 0.7.1

Further reading

- franka ros
- documentation

3.5.1 Trajectory composition

Franka has a dual way of programming its panda robot: First, there's the graphical-based programming using a browser interface (Desk) in which users compose tasks with different apps. The apps range from simple to complex and can be customized and shared with a community. Some apps are payed, limiting this overview to the default available functionality.

- **Linear Cartesian motions Cartesian Motion**
  - Move linearly along a list of points, allowing to set blending, velocity and acceleration.
- **Additional Relative Motion**
  - Move relative to the current pose in a direction specified as vector.

  Line
  - Move along a line with specified velocity for a certain duration. The direction vector is taught with two points.

  Lissajous Figures
  - Realize lissajous figures in a plane. Allows to specify motion and amplitude in both directions.

  Spiral
  - Move in a spiral pattern in a plane. Provides configuration parameters, such as duration, width, etc.

The second way to realize Cartesian trajectories is directly over the Franka Control Interface (FCI) (Research) that provides full control of the robot in form of motion executors

- **joint position**
- **joint velocity**
- **Cartesian pose**
- **Cartesian velocity**

and controllers

- **External controller**: Users can command torques directly to the robot
- **Internal joint impedance**: Implicitly handled without user commands
- **Internal Cartesian impedance**: Implicitly handled without user commands

Motion executors and controllers can be combined to realize hybrid behavior.

Additionally, the robot has a ROS-control interface that uses FCI and provides users with a base for own ROS controller developments. According to the hardware interface conventions of ROS-control, Franka offers:

Joint-based interfaces for reading and writing

- **hardware_interface::VelocityJointInterface**
- **hardware_interface::PositionJointInterface**
- **hardware_interface::EffortJointInterface**

and custom Cartesian interfaces for reading and writing with

- **franka_hw::FrankaPoseCartesianInterface**
- **franka_hw::FrankaVelocityCartesianInterface**

All interfaces operate on a 1kHz cycle.

When using these research interfaces, users have the freedom (and necessity) to implement motion types (e.g. with ROS controllers) on top of raw target and feedback signals from the robot.

3.5.2 Waypoint representation

For users not having access to Franka's Desk environment, it's not straight forward to identify the waypoint representation. However, at least on the research interface, a full robot state is available. Subsets of this state could potentially be saved as waypoints when writing own motion commands.

Apart from the last commanded values, the state contains

```
joint level signals:
   - motor angles
   - motor angle derivatives
   - joint angles
   - joint angle derivatives
   - joint torque
   - joint torque derivatives
   - estimated external torque
   - joint collisions/contacts

Cartesian level signals:
   - Cartesian pose
   - load parameters
   - external wrench
   - Cartesian collision
```

3.5.3 Trajectory parameterization and execution

Since Franka's FCI allows users to implement any desired behavior themselves, the following list is limited to the possible configurations available for the apps-based approach.

Specification of velocity

- In form of a percentage of the robot's maximal velocity. Is done when configuring instances of motion types

Specification of acceleration

- Also in form of a percentage of the maximal values. Is parameterized during setup of the motion types

Blending

- Can be configured in the **Cartesian Motion** app

Parallel IO operations

- Specific apps trigger operations, such as **Modbus Wait**, **Modbus Out** and **Modbus Pulse**

Online (real-time) trajectory modifications

- Can be achieved implicitly through active impedance control and additional forces set with **Apply Force**
- External forces and commanded forces can overlay Cartesian motion types and alter the trajectories

3.6 ABB

Cartesian trajectories for ABB robots (IRC5 controllers)

Table 5: Vendor specifics

Teach pendant	FlexPendant
Programming / simulation software	RobotStudio
Software	FlexPendant SDK, Microsoft CE + Visual Studio
Programming language	RAPID
Relevant hardware	Robots of the IRC5 controller

Further reading

- manual_rapid

3.6.1 Trajectory composition

Programming is done with move instructions (motion types) that move the robot to specified targets.

- **Linear Cartesian motions MoveL**
  - Move linearly to a specified target.
  - Possible arguments: target point, speed, coordinate system, duration until point (replaces speed), and others
- **Circular motions MoveC**
  - Build circular, open motion arcs, using a via-point and end point.
  - Possible arguments: point on circle, target point, coordinate systems, duration, and others
- **Joint space interpolation MoveJ**
  - Move the robot to specified points using joint interpolation. All joints will reach their destination at the same time.

  - Possible arguments: Target point, speed, zone, tool, and others. The tool center point is the point moved to the destination.

3.6.2 Waypoint representation

Individually taught points (type **robtarget**) have the following field representation:

```
trans
rot
robconf
extax
```

Individual fields

- **trans**: x, y, z (position of tcp)
- **rot**: q1, q2, q3, q4 (orientation in quaternion notation)
- **robconf**: cf1, cf4, cf6, cfx (axis configuration of the robot for possibly ambiguous axes). Each field (integer) indicates the *configuration quadrant* for the numbered axis and is counted in positive or negative quarter revolutions of 90° starting from zero:

```
...
-3 = axis is in (-270°, -180°)
-2 = axis is in (-180°, -90°)
-1 = axis is in (-90°, -0°)
 0 = axis is in (+0°, +90°)
 1 = axis is in (+90°, +180°)
 2 = axis is in (+180°, +270°)
...
```

- **extax**: [eax_a, eax_b, eax_c, eax_d, eax_e, eax_f] (list of up to six external hardware axes)

Different coordinate systems for point representations are possible, such as world or various object coordinate systems.

3.6.3 Trajectory parameterization and execution

Specification of velocity

- In form of a speed argument to move instructions during teaching of fly-by points. The speed is valid until the next point
- Path segment specific with **VelSet** (overrides global velocity settings until reset). Can be either specified as percentage of the current global velocity or can be set to become the new max velocity.
- Global adjustments with **motsetdata**, affects all points

Specification of acceleration

- Path segment specific accelerations with **AccSet** (overrides global acceleration until reset). Provokes slower acceleration and deceleration (percentage) of the global settings.
- Global adjustments with **motsetdata**, affects all points. This also includes adjustment of ramping accelerations etc.

Blending

- Taught positions can either be fly-by points, or stop points
- During **MoveL**, fly-by points are automatically blended, leading to adjusted *corner paths* (parabolas). Stop points are exactly passed.
- The blending configuration is handled with *zone data* that specifies how corner paths are realized.
- A parameterization of different zones allows to design corner paths in which tool orientation and Cartesian position can be started and stopped Individually.

Parallel IO operations

- **MoveLDO**: Move linearly and trigger an I/O operation at the target's middle corner path
- **MoveCDO**: Move in a circle and trigger an I/O operation at the target's middle corner path

Online (real-time) trajectory modifications

- Offsets to paths can be realized with **CorrWrite** and special correction generators. No information on real-time capability found.

3.7 Yaskawa

Cartesian trajectories for the Yaskawa Motoman robots

Table 6: Vendor specifics

Teach pendant	Teach panel (programming pendant)
Programming / simulation software	Partly PLC support
Software	Customization of teach pendant: C++ and C#
Programming language	INFORM
Relevant hardware	Controllers YRC1000, DX100, NX100 and probably others.

Further reading

- operator's manual (YRC1000)
- inform language (YRC1000)
- on blending
- on circles and splines

3.7.1 Trajectory composition

Teaching trajectories is done on the basis of a point-2-point approach with different motion (interpolation) types:

- **Linear Cartesian motions MOVL**
  - For linear motion to the point
- **Circular motions MOVC**
  - For circular motion with three points for each arc

- **Joint space interpolation MOVJ**
  - For joint-interpolated motion to the point
- **Additional MOVS**
  - For spline motion with parabolic interpolation between three points

  IMOV
  - For incremental linear motion, starting from a point

There are also application specific routines that help in the process of trajectory programming while teaching points. These routines may include additional sensors, such as force-torque sensors for approaching surfaces or contour following. The final obtained trajectories, however, are still built with basic motion types.

3.7.2 Waypoint representation

Individually taught points have the following representation:

```
X
Y
Z
Rx (roll)
Ry (pitch)
Rz (yaw)
```

Different coordinate systems for point representations are possible.

3.7.3 Trajectory parameterization and execution

Specification of velocity

- Point-level: Execution speeds are taught on a per-point basis, applied from the previous to the current point, respectively. As an example, the speed taught at point P2 is applied from point P1 until P2. Users can also adjust speed as a percentage to the trajectory-global play speed.
- Trajectory-global: Users can choose and adjust different execution speeds on a discrete scala with the **SPEED** instruction

Specification of acceleration

- Point-level: Users can adjust acceleration **ACC** and deceleration **DEC** as *adjustment ratios* in the range of 20% - 100% for each point

Blending

- Is done with setting a position level (PL) incrementally for **MOVL** points, with PL from 0 = no bending radius to 8 = max. bending radius

Parallel IO operations

- Due to synchronous execution, users apply IO operations immediately before/after move instructions.

Online (real-time) trajectory modifications

- No information found so far

3.8 Summary of vendor interfaces

This section will summarize the different vendor interfaces to explicitly show common patterns that we could build an interface upon.

3.8.1 Segment interpolation methods

	linear Cartesian	**circular** Cartesian	**spline** Cartesian	**joint**
KUKA	yes	yes	yes	yes
UR	yes	yes	no	yes
Fanuc	yes	yes	no	yes
Doosan	yes	yes	yes	yes
Franka Emika	yes	(yes)	(yes)	(yes)
ABB	yes	yes	no	yes
Yaskawa	yes	yes	yes	yes

The investigated interfaces differ in the amount and richness of provided motion types, such that the table above summarizes the best supported subset.

Apart from Franka Emika, all investigated robot interfaces could natively execute trajectories composed of *linear*, *circular* and *joint* interpolated motion segments. A possible Cartesian trajectory definition could therefore build upon these three types.

Franka Emika is a special case here, as they do not offer all functionality in an open programming language, but they have pre-built solution modules that help users getting their tasks done. Therefore, such interfaces do exist, but in a different manner than the other brands and should not veto the trajectory definition.

3.8.2 Parameterization of trajectory executions

For each vendor interface, We looked at four parameter categories that will be important for trajectory definitions: **Velocity**, **Acceleration**, **Blending**, and **Parallel IO** commands.

The table below summarizes the findings to highlight commonly supported configuration by each vendor. We use our own abbreviations to keep the table clear.

For velocity / accelerations, users can set:

```
(a) Global target parameter for whole trajectory
(b) Local target parameter for motion or waypoint
(c) Global limit for whole trajectory
(d) Local limit for motion or waypoint
(e) Percentage of globally defined parameter
```

For blending, users can set:

```
(1) Radius per waypoint
(2) Radius per motion
(3) Global radius for all motions or waypoints
(4) Advanced radii definitions
```

	Velocity	Accelera-tion	Blend-ing	Parallel IO
KUKA	(a) (e)	(e)	(1) (2) (4)	??
UR	(b) (c) (d)	(b)	(1) (3)	Synchronized with waypoints
Fanuc	(a) (e)	(c)	(1)	??
Doosan	(a) (b) (e)	(c) (d)	(1)	In parallel with asynchronous motions
Franka Emika	(e)	(e)	(2) (4)	User-custom with apps
ABB	(a) (b)	(a) (c) (e)	(1) (4)	Separate motion commands with IOs triggered in the middle
Yaskawa	(a) (b)	(e)	(1)	Directly before or after motion commands

Most interfaces provide similar parameterization of velocity and accelerations. However, blending is differently implemented by the vendors. The (4) = *Advanced radii definition* mentioned in the column *Blending* refers to blending for Cartesian motions for Franka Emika, blending through detailed corner paths for ABB and blending between joint and Cartesian motion for KUKA.

The double ?? indicate that we didn't find sufficient information on this subject.

3.8.3 Specification of waypoints

	Waypoint Representation	Posture definition
KUKA	xyz-rpy	3 bit configuration, 6 bit turn directions
UR	xyz-rpy	qnear
Fanuc	xyz-rpy	3 bit configuration
Doosan	xyz-rpy (zyz)	3 bit configuration
Franka Emika	user's choice	user's choice
ABB	xyz-quat	quadrants for axes
Yaskawa	xyz-rpy	3 bit configuration

Franka's waypoint representation for the app-based approach was not obvious during this review. However, their fully exposed interface with low-level access to joint drivers for the FCI-based approach offers users to implement specific representations themselves. We used *user's choice* to indicate this.

Both KUKA and ABB offer more detailed posture definitions with respect to the other vendor interfaces. A least denominator of 3 bit posture configurations is therefore meaningful for a common interface with UR being an exception, as they use a notation of a nearby joint configuration.

Posture definition is discussed in section *Conclusion / Proposed Interface*. We suggest to follow the definition there, as it should be possible to map them to the vendors' representations.

ABB stands out with their quaternion representation for orientation, avoiding potential gimbal locks. Due to the unique mapping from angle-sequence representations to quaternions, however, all observed vendor interfaces will work nicely with a roll-pitch-yaw notation.

Proposed interface

To be determined. . . .

- Integration into / combination with REP-I0003?
  - Open questions?
    * How to integrate into ROS control?

CHAPTER 4

Conclusion / Proposed Interface

Cartesian trajectory definitions have long been a complicated topic in ROS. The investigated definitions showed that quite some differences exist about generality vs expressiveness vs ease of use, not to mention the somewhat *orthogonal* industrial way of doing things. In fact, most people will agree that it's probably impossible to cover every detail, to meet all requirements of all possible users of such an interface. Nevertheless, with this document having a lot of information in one place, we believe that there are sufficient similarities to start a new trial.

If you don't want to read the reasoning, you can jump to the *proposed interface* directly.

4.1 Components

Here's our proposal for Cartesian Trajectory Definitions. We present the new message types step by step and explain our design decisions, basing them on conclusions from the previous section *Existing Cartesian Interfaces in ROS*.

4.1.1 CartesianTrajectoryPoint

One common thing in all existing proposals is a Cartesian trajectory point definition. This would be fairly similar to the trajectory_msgs/JointTrajectoryPoint message.

A jerk is added to the trajectory point definition, as well, so controllers executing a Cartesian trajectory can provide a smoother trajectory execution. As there is currently no message available to encode this, a custom message will be provided initially. Essentially, it would be a copy of `geometry_msgs/Accel` but reusing this would be semantically incorrect. There is also an open discussion on adding Jerks to `geometry_msgs` which would be the preferable solution.

Listing 1: CartesianTrajectoryPoint.msg

```
duration time_from_start
geometry_msgs/Pose pose
geometry_msgs/Twist twist
geometry_msgs/Accel acceleration
<to_be_determined_msgs>/Jerk jerk
```

The definition above doesn't contain any *frame_id* or *timestamp* information raising the need to contain this into a parent message.

Postures

When assembling Cartesian points to a trajectory additional posture information could be given to specify the desired joint configuration in case of multiple possible solutions coming from an IK solver or planner.

In a first naive attempt we define posture information inside a separate message to decouple it from the geometric waypoint definition. A given joint configuration defines the configuration being close to the desired IK solution. Multiple solutions can be checked for similarity to the given configuration.

Listing 2: NaiveCartesianPosture.msg

```
float64[] joint_values
```

The user has to make sure that the number of entries given in the `posture` array match the number of joints similar to the trajectory_msgs/JointTrajectoryPoint message. *Gijs van der Hoorn* proposed to use a sensor_msgs/JointState message for posture information. However, in contrast to that we propose to use a plain `float64[]` field instead of a full joint state in order to prevent redundant information / containing a lot of unused fields. This on the other hand raises the requirement to specify the joint names on a higher level as mentioned above.

The posture definition proposed above is raises a couple of questions / concerns as discussed inside #5.

Should the field be made optional?

- There might be kinematic structures that only result in one single IK solution for a given Cartesian pose.
- Posture constraints are not necessarily known or relevant when defining / executing a trajectory. The user might not have a preference for a specific configuration in which case a mandatory posture definition will force the user to pick one instead.

In order to give users the choice instead of enforcing posture definitions, we propose to have posture definitions optional.

Should partial posture specifications be supported?

- It might be beneficial to allow partial posture definitions such as a shoulder lift joint or elbow joint value only. However, allowing this will require to have the joint names available at this stage, as well.
- This would effectively allow both, a "flip bits" approach as used by many vendors, as well as a "qnear" approach where the user defines a (partial) joint configuration which is close to the desired configuration.
- By allowing partial posture definitions, users can choose to specify selected joints only, while leaving the other joints for optimization e.g. by an IK solver or planner. Especially when combined with a `CartesianTolerance` this would be a rather powerful feature.

As a result, we propose to allow partial posture definitions.

Should postures be made a member of `CartesianTrajectoryPoint`?

- One argument against that is that `CartesianTrajectoryPoint` should be a pure geometric representation of a setpoint independent of any robot configuration. This way, a sequence of `CartesianTrajectoryPoint` objects could be reused and applied to different robot kinematics and /

or use-cases. Posture definition should be a part of the trajectory execution but not the trajectory definition. However, as raised in #5 (comment), we should not mix up tool paths and robot trajectories. As this proposal is about robot trajectories, `CartesianTrajectoryPoint` instances should be treated as trajectory setpoints, not tool paths and therefor it makes sense to incorporate the posture definition into the setpoint.

- If posture is not included into the `CartesianTrajectoryPoint` structure, there has to be a way of matching posture definitions to trajectory setpoints. For this, either a unique identifier for each waypoint would be needed or users would have to provide a posture definition for each waypoint to get a 1-to-1 mapping. This would however conflict with the posture definition being optional for each waypoint. Additionally, there would have to be additional code required checking that each waypoint has a corresponding posture definition.
- If the posture configuration is defined for each `CartesianTrajectoryPoint` it can be left empty for each waypoint by simply not defining it. Thus, if a user chooses not to define any posture, no additional action would be required. If postures would be stored in a parallel datastructure on trajectory level, users would have to define an empty posture for each waypoint individually.

For the sake of usability we propose to include the posture definition into the `CartesianTrajectoryPoint`. This comes with the cost of a CartesianTrajectoryPoint being coupled to a specific kinematic setup, though.

Should `posture_joint_names` be a member of `CartesianPosture`?

- Defining joint names in each `CartesianPosture` would effectively increase the amount of redundant information in case of a fully defined posture specification for each waypoint, which motivated us to exclude it from our naive posture definition above.
- Integrating joint names into the posture definition adds the possibility to define partial posture constraints, e.g. only requiring shoulder and elbow configuration.

As reasoned above partial posture definitions are a desired feature which is why including the joint names into the posture definition is required.

Posture definition

With the reasons above, we propose the following `CartesianPosture` to be included into `CartesianTrajecoryPoint`:

Listing 3: CartesianPosture.msg

```
# Posture joint names may reflect a subset of all available joints (empty posture␣
↪definitions are
# also possible). The length of posture_joint_names and posture_joint_values have to␣
↪be equal.

string[] posture_joint_names
float64[] posture_joint_values
```

4.1.2 CartesianTrajectory

To get a trajectory from multiple `CartesianTrajectoryPoint` objects the next container is a trajectory object consisting of multiple trajectory points.

Listing 4: CartesianTrajectory.msg

```
# header.frame_id is the frame in which all data from CartesianTrajectoryPoint[] is␣
↪given
Header header
CartesianTrajectoryPoint[] points
string controlled_frame
```

At this stage we include a time stamp through the `header` message. Note that `header` also includes a `frame_id`, which is the assumed reference frame for the data given in `points`. The link that shall follow the trajectory is specified with `controlled_frame`. Some of the existing proposals use a `geometry_msgs/Pose` field to express the points' reference frame. However, we think that using names as identifiers makes this interface more versatile, because it delegates possible lookups to where this information is easier available.

4.1.3 CartesianTolerance

In the investigated interfaces tolerances are often proposed as scalar values for each of [position, orientation, velocity, angular velocity]. In contrast we propose specifying constraints for each axis individually by using 3-dimensional datatypes:

Listing 5: CartesianTolerance.msg

```
geometry_msgs/Vector3 position_error
geometry_msgs/Vector3 orientation_error
geometry_msgs/Twist twist_error
geometry_msgs/Accel acceleration_error
```

With this definition users can define tolerances per axis, where rotational constraints are meant to be angle differences in the local coordinate system. Therefore we use `geometry_msgs/Vector3` instead of `geometry_msgs/Pose` for `position_error` and `orientation_error`.

4.1.4 CartesianTrajectoryAction

Similar to the control_msgs/FollowJointTrajectory action we propose an action interface for executing Cartesian trajectories.

Listing 6: FollowCartesianTrajectory.action

```
CartesianTrajectory trajectory
CartesianTolerance path_tolerance
CartesianTolerance goal_tolerance
duration goal_time_tolerance

---

int32 error_code
int32 SUCCESSFUL = 0
int32 INVALID_GOAL = -1 # e.g. illegal quaternions in poses
int32 INVALID_JOINTS = -2
int32 OLD_HEADER_TIMESTAMP = -3
int32 PATH_TOLERANCE_VIOLATED = -4
int32 GOAL_TOLERANCE_VIOLATED = -5
int32 INVALID_POSTURE = -6
```

(continues on next page)

(continued from previous page)

```
string error_string

---

Header header
string controlled_frame
CartesianTrajectoryPoint desired
CartesianTrajectoryPoint actual
CartesianTolerance error
```

For the result and feedback we again are following the methods from joint-based trajectory execution. The errors get extended by a posture-related error flag.

4.2 TLDR; Proposed interface

As elaborated in the previous section we propose the following action interface

Listing 7: FollowCartesianTrajectory.action

```
CartesianTrajectory trajectory
  # header.frame_id is the frame in which all data from CartesianTrajectoryPoint[] is␣
↪given
  Header header
  CartesianTrajectoryPoint[] points
    duration time_from_start
    geometry_msgs/Pose pose
    geometry_msgs/Twist twist
    geometry_msgs/Accel acceleration
    <to_be_determined_msgs>/Jerk jerk
    CartesianPosture posture
      string [] posture_joint_names
      float64[] posture_joint_values
  string controlled_frame
CartesianTolerance path_tolerance
  geometry_msgs/Vector3 position_error
  geometry_msgs/Vector3 orientation_error
  geometry_msgs/Twist twist_error
  geometry_msgs/Accel acceleration_error
CartesianTolerance goal_tolerance
  geometry_msgs/Vector3 position_error
  geometry_msgs/Vector3 orientation_error
  geometry_msgs/Twist twist_error
  geometry_msgs/Accel acceleration_error
duration goal_time_tolerance

---

int32 error_code
int32 SUCCESSFUL = 0
int32 INVALID_GOAL = -1 # e.g. illegal quaternions in poses
int32 INVALID_JOINTS = -2
int32 OLD_HEADER_TIMESTAMP = -3
int32 PATH_TOLERANCE_VIOLATED = -4
int32 GOAL_TOLERANCE_VIOLATED = -5
```

(continues on next page)

(continued from previous page)

```
int32 INVALID_POSTURE = -6

string error_string

---

Header header
string controlled_frame
CartesianTrajectoryPoint desired
    duration time_from_start
    geometry_msgs/Pose pose
    geometry_msgs/Twist twist
    geometry_msgs/Accel acceleration
    <to_be_determined_msgs>/Jerk jerk
CartesianTrajectoryPoint actual
    duration time_from_start
    geometry_msgs/Pose pose
    geometry_msgs/Twist twist
    geometry_msgs/Accel acceleration
    <to_be_determined_msgs>/Jerk jerk
CartesianTrajectoryPoint error
    duration time_from_start
    geometry_msgs/Pose pose
    geometry_msgs/Twist twist
    geometry_msgs/Accel acceleration
    <to_be_determined_msgs>/Jerk jerk
```

Note: For readability reasons we left the commonly-used ROS messages collapsed.

CHAPTER 5

ToDos