ROBOTICS

Application manual

Automatic Path Planning

Trace back information:
Workspace Main version a631
Checked in 2024-11-29
Skribenta version 5.6.018

Application manual
Automatic Path Planning

Document ID: 3HAC092826-001
Revision: A

The information in this manual is subject to change without notice and should not be construed as a commitment by ABB. ABB assumes no responsibility for any errors that may appear in this manual.

Except as may be expressly stated anywhere in this manual, nothing herein shall be construed as any kind of guarantee or warranty by ABB for losses, damage to persons or property, fitness for a specific purpose or the like.

In no event shall ABB be liable for incidental or consequential damages arising from use of this manual and products described herein.

This manual and parts thereof must not be reproduced or copied without ABB's written permission.

Keep for future reference.

Additional copies of this manual may be obtained from ABB.

Original instructions.

Table of contents

This page is intentionally left blank

Overview of this manual

About this manual

This manual describes the functionality of the software *Automatic Path Planning*.

> **Note**
>
> It is the responsibility of the integrator to conduct a risk assessment of the final application.
>
> It is the responsibility of the integrator to provide safety and user guides for the robot system.

References

External references

- gRPC, 2022. Available:
  https://grpc.io/
- gRPC in .Net. Available:
  https://learn.microsoft.com/en-us/aspnet/core/tutorials/grpc/grpc-start?view=aspnetcore-7.0&tabs=visual-studio

User documentation from ABB Robotics

Reference	Document ID
Application manual - Functional safety and SafeMove	*3HAC066559-001*
Operating manual - RobotStudio	*3HAC032104-001*
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC065038-001*

Revisions

Revision	Description
A	First edition.

This page is intentionally left blank

1 Safety

Risk assessment

It is the responsibility of the integrator to conduct a risk assessment of the final application.

Verify the safety functions

Before the robot system is put into operation, verify that the safety functions are working as intended and that any remaining hazards identified in the risk assessment are mitigated to an acceptable level.

This page is intentionally left blank

2 Description of *Automatic Path Planning*

2.1 *Automatic Path Planning*

Introduction to *Automatic Path Planning*

The *Automatic Path Planning* is a self-contained path planning server built for Windows and Linux. The software addresses both offline and online applications that can benefit from kinematics, collision-checking, and automatic path planning services.

> **CAUTION**
>
> *Automatic Path Planning* will produce collision-free paths under the assumption that the virtual model is an accurate representation of the robot cell, and that the kinematic model of the robot has no errors. It is up to the user verify the accuracy of the model and provide *Automatic Path Planning* with safety margins that are large enough to compensate for errors in the kinematics and the virtual model.
>
> SafeMove can be used to set up safe zones for areas where people can be present, or obstacles that are not represented in the virtual model.

After adding a robot, attachments to the robot (for example, a robot tool from a CAD model), obstacles from CAD models or point clouds, the server will, on request, return a collision-free path from a starting target (RobTarget or JointTarget in RAPID) to a goal target. The server will try to find the shortest path from the start to the end. The returned path is a sequence of targets that are to be sent to the robot controller. The path does not have a speed parameter, as the server is a geometrical path planner and has no notion of motion time. Since the path is optimized in the joint space it is singularity-free. Furthermore, the zones of the targets are optimized to be as large as possible so that the motion is smooth, efficient, and fast.

The time needed by the server to generate a path depends on the complexity of the problem, the number of obstacles, and the available CPU performance, and can range from a few tens of milliseconds to some seconds. The server uses multi-threading to speed-up computations, and the user can configure the number of threads that can be used by the server.

The API

The file *cfree.proto* is the specification of the server API. The available services are listed in the file.

The API uses lowerCamelCase for server queries, UpperCamelCase for message types, and snake_case for message fields.

Supported robots

The following robots are supported by the path planning server:

- Six DOF elbow IRB robots (`Elbow`), for example, IRB 5710, CRB 1100
- Six DOF parallel rod robots (`ParallelRod`), for example, IRB 8700

Continues on next page

- GoFa robots (`ElbowWristOffset`), CRB 15000
- Single arm YuMi (`RedundantRobots`), IRB 14050

The name within parentheses is the corresponding enum name in the proto file. The difference between the `Elbow` type and the `ElbowWristOffset` type is that the former has a spherical wrist, while the latter has a z offset in the wrist.

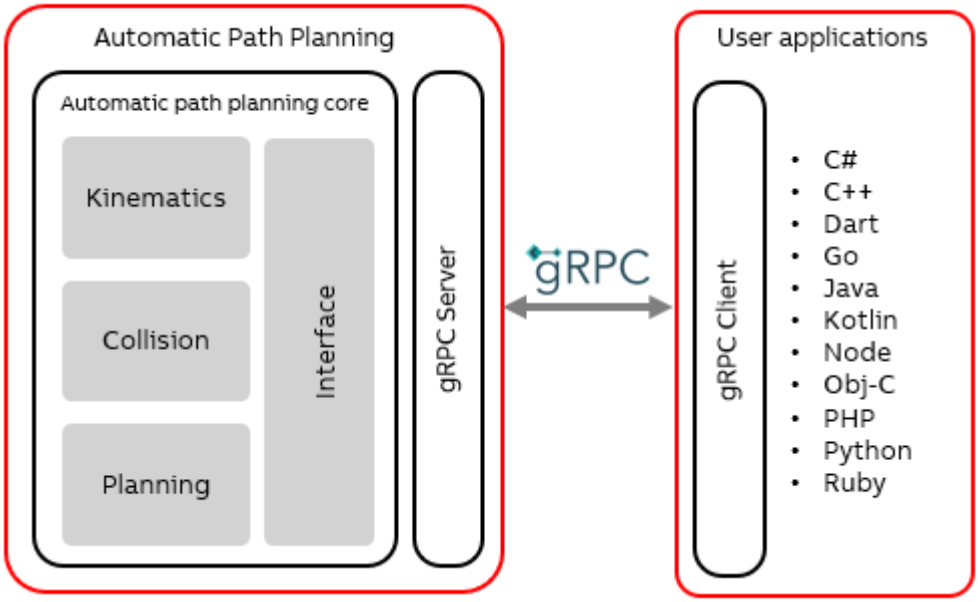
Paint robots are not supported.

2.2 Communication with the server

Introduction

The *Automatic Path Planning* is using gRPC, an open-source high performance Remote Procedure Call (RPC) framework from Google™ that can run in any environment. The client can be implemented in any language supported by the gRPC including C++, C#, Python, and Java. The server has no relation to RobotWare.

The following figure shows an overview of the server architecture. Tests show communication overhead with gRPC to be around 1ms for most services. The latency will depend on the message size. Large message loads (for example, sending a large point cloud) will result in higher latencies. Messages in gRPC are encoded in binary, using Google Protobuf.



xx2400001455

Client API generation

gRPC provides tools to automatically generate interfaces in several supported languages from a specification in proto format. The provided `cfree.proto` file can be used to generate the API interface to the server automatically. After that, setting up a client to establish communication with the server requires little effort. The remaining work for integrating the *Automatic Path Planning* in an application consists of converting data types from the application to the proto messages, constructing queries, calling the services and parsing the received response.

A fully documented Python client with 3D visualization capabilities is provided with the server.

Continues on next page

Network recommendations

The default behavior for the path planning is to listen on the port `localhost:50051`. It is recommended to not communicate with the *Automatic Path Planning* server over a remote connection. Therefore, the server and the client should be executing on the same computer.

Server usage in parallel

The *Automatic Path Planning* core and its interfaces are designed to be thread safe, therefore multiple clients can access the server simultaneously. However, in current form, functions such as collision-checking and path-planning require resources that cannot be used in parallel and are therefore access-locked during execution of those services. Therefore, there is little benefit in calling collision-checking and path planning services in parallel to gain compute time advantage. However, multiple instances of the server (with different IP and/or port of course) can be executed in parallel on the same system. Note that the path planning is done in multiple threads.

Visualization

Currently *Automatic Path Planning* does not provide native visualization tools. This is in part because of the various use-cases for which it is designed, which makes having a visualization concept that can be integrated in all these applications infeasible. The path planning server does however provide services to retrieve all the required data for visualization of the robot, obstacles, and paths. These services are:

- `getObstacleCollisionBodies`: To retrieve the obstacles (all obstacles or those only active for a specific robot) as triangular mesh objects.
- `getRobotCollisionBodies`: To retrieve the robot links and attachments (tools, load) at a given target as triangular mesh objects.
- `getVoxels`: To retrieve the voxel-based representation, if present.
- `collisionCheck`: To perform collision checks at one or more given targets. It returns a list of robot links/attachments and obstacles that are in collision, so that appropriate visualization such as highlighting colliding links/obstacles can be done.
- `interpolatePath`: To interpolate a path consisting of a series of targets with zones. This, together with the `forwardKinematicsTCP` service, can be used to accurately visualize the TCP trace of, for instance, a `MoveAbsJ` path with zones.
- `forwardKinematics`: To retrieve the pose of each robot link at a given target, to animate the robot links (which in turn are retrieved using the `getRobotCollisionBodies` service) along a path of targets.

Continues on next page

Logging of server requests

For development and debugging of online applications, the requests-logging service can be utilized to enable the complete logging of all called services. When requests-logging is enabled, a file is produced by the server that can be used later to replicate all the calls to the server. This includes adding robots and obstacles. Therefore, requests-logging should be enabled before any other service is called so that the added robot and obstacles are also logged. Furthermore, parsing functions are provided in the Python client package to read a log file and replicate the scenario.

2.3 Network security

Network security

This product is designed to be connected to and to communicate information and data via a network interface. It is your sole responsibility to provide, and continuously ensure, a secure connection between the product and to your network or any other network (as the case may be).

You shall establish and maintain any appropriate measures (such as, but not limited to, the installation of firewalls, application of authentication measures, encryption of data, installation of anti-virus programs, etc) to protect the product, the network, its system and the interface against any kind of security breaches, unauthorized access, interference, intrusion, leakage and/or theft of data or information. ABB Ltd and its entities are not liable for damage and/or loss related to such security breaches, any unauthorized access, interference, intrusion, leakage and/or theft of data or information.

3 Setting up *Automatic Path Planning*

3.1 Installation

Overview of the installation

1. Download[I] the distribution package for *Automatic Path Planning* from library.abb.com
2. Install the software on a device.
3. Activate the license.
4. The distribution package contains some examples that can be used for testing and getting started.

I https://library.abb.com/r?cid=9AAF630578

Activating the license key

The licensing for *Automatic Path Planning* works in a similar way as RobotStudio. The license is connected to the device, allowing *Automatic Path Planning* to run on it. Once a key has been used, it cannot be used on any other device.

Activating over internet (preferred)

1. Open a terminal/command line prompt on the target device in the install directory for *Automatic Path Planning*.
2. Initialize the license system: `cfree -init`

   This is done once per device.

   In a Windows multi-user environment this should be done from an admin prompt.
3. Activate the device: `cfree -a [activation-key]`

   You should get a response stating that the license was successfully activated.

Activating without internet connection

1. Open a terminal/command line prompt on the target device in the install directory for *Automatic Path Planning*.

   Or type `cfree -h` for help on available command line parameters.
2. Initialize the license system: `cfree -init`

   This is done once per device.

   In a Windows multi-user environment this should be done from an admin prompt.
3. Activate the device: `cfree -a [activation-key]`

   You should get a response stating that the license was successfully activated.
4. Create a license request file: `cfree -licrequest <key> <file[.licreqx]>`
5. Copy the file to a USB device or similar.
6. On a machine with internet connection, go to http://manualactivation.e.abb.com/
7. Upload the license request file.

Continues on next page

8 Copy the returned license file to the USB device.

9 On the target device, import the license file: `cfree -importlicense <license-file>`

You should get a response stating that the license was successfully activated.

Starting the server

1 Open a terminal/command line prompt on the target device in the install directory for *Automatic Path Planning*.

2 Type `cfree -unlock` to start the server.

3 Type `cfree -h` for help on available command line parameters.

3.2 Adding robots and attachments

Adding robots

1. To add a robot, use the service `addRobot`.

   The service returns an integer ID for the added robot, and a list of collision body IDs (one for each robot link geometry) that can be used in robot-related services like collision checking or path planning.
2. To get the parameters required for the robot, use the export functionality in RobotStudio. This function exports a robot, and additional geometric objects, to a file format that can be loaded by *Automatic Path Planning*.
3. The parameter `min_dist_self_collision` specifies how close the links of the robots can get to each other before a self-collision is determined. This can be important if there is uncertainty in robot attachments, for instance a load picked up by the robot. As the load will be part of the robot, the minimum self-collision distance parameter will dictate how close it can get to the robot links. The minimum allowed value is 0.001 meters.

Adding robot attachments

After the robot is created, robot attachments can be added as collision bodies to any frame of the robot. The collision bodies can be added with `addCollisionBody` or `addRobotTool`.

1. To add a robot attachment with the service `addCollisionBody`, specify the following parameters:
   - `robot_id`
   - `frame_number` (see *Frame numbering on page 23*)
   - `frame_offset` is a pose that can be used to specify how the collision body should be attached relative to the robot frame.
2. To add a robot tool with the service `addRobotTool`, specify the following parameters:
   - An optional name (to simplify debugging and coding). The default value is `tool0`.
   - `frame_offset` is a pose that can be used to specify how the collision body should be attached relative to the robot frame. This corresponds to `tframe` in the RAPID data type `tooldata`.
   - An optional collision body modelling the tool.
3. The server returns an ID for the added tool that can be used in the `switchTool` service.

Continues on next page

Switching robot tool

It is possible to add multiple tools to a robot and switch between them using the `switchRobotTool` service and the ID of the desired tool.

> **Note**
>
> The TCP of the switched tool will then be used in forward and inverse kinematics. Any collision body of the tool will be taken into account in collision checking and path planning.

Switching tools can be particularly useful in online applications, where the tool addition (which may include costly convex decomposition) is done once and at runtime tools are only switched.

3.3 Setting up obstacles

Adding a mesh obstacle

The service `addCollisionBody` can be used to add to the scene an obstacle or a robot attachment that consists of a set of convex hulls. Alternatively, `generateConvexDecomposition` can be used to send a generic mesh, from which first the convex hull decomposition is performed, and the resulting convex hulls are added as the collision body. In either case, a unique integer is returned as the ID of the collision body that can be used in services such as `removeCollisisonBody` or `activateCollisionBody`. A collision body can be given a name, which is helpful in debugging and improves messages about collisions that the path planning server might return. Note that the server will not check or require that the provided name is unique.

The `CollisionBody` message has an optional field, `min_dist_override`, that, if set, overrides the global minimum allowed distance to obstacles set by a path-planning query. This is useful if the collision body represents an obstacle that the robot is allowed/required to move closer to compared to other obstacles, for example, a table or a fixture. The minimum allowed value for the field `min_dist_override` is 0.001 m.

When an obstacle is added to the server, it is checked whether the obstacle is within a sphere that approximates the workspace of the robot. If the obstacle is outside the sphere, the obstacle will be ignored in collision checking of that robot. If the geometry of the robot changes, this sphere-check is repeated for all added obstacles, so that when for example a longer tool is added to the robot, a previously ignored obstacle might become active because of being within the robot's workspace.

In applications where there are multiple obstacles with the same geometry but at different locations, it is recommended to add a single collision body to the `addCollisionBody` and fill-in the optional `replication_poses` field with the poses of the obstacles. This increases the efficiency of the service.

Tip

If defining several collision bodies, it is more efficient to define them all at once with a single call to `addCollisionBody` than to call the service repeatedly for each collision body.

Deactivating or removing a mesh obstacle

Use the service `activateCollisionBody` together with one or more collision body IDs to activate or deactivate a collision body. A deactivated collision body is ignored in collision checking and path planning.

It is also possible to remove a collision body from the scene if it is no longer needed by using the service `removeCollisionBody`.

This page is intentionally left blank

4 Reference information

4.1 Robots and frame numbering

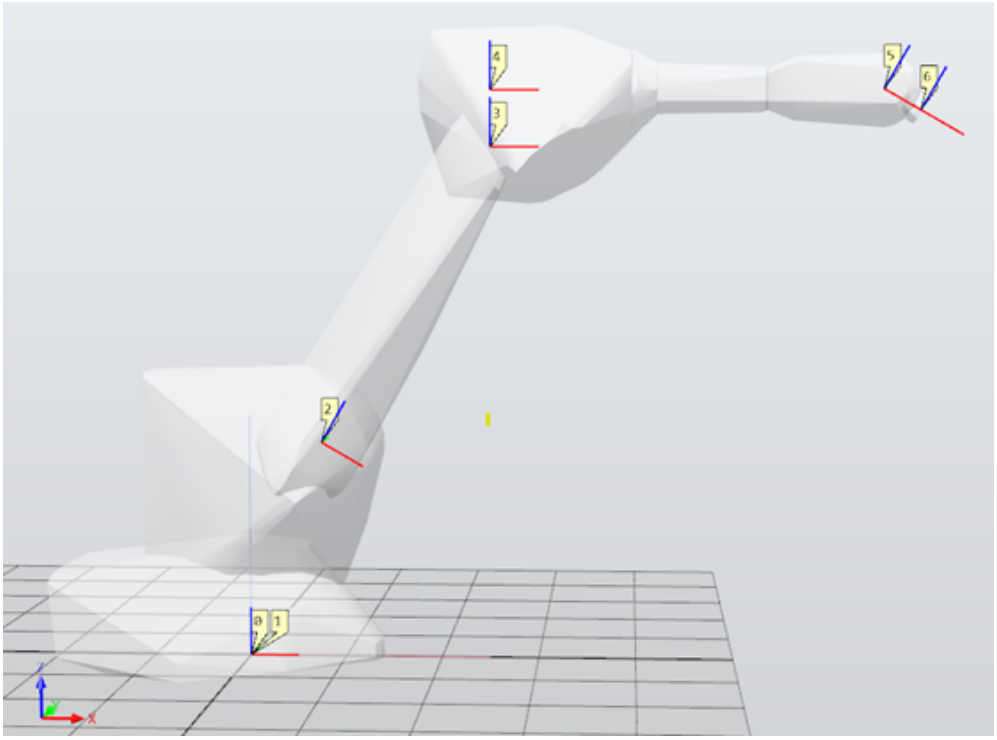
Introduction

A robot in *Automatic Path Planning* consists of a kinematic model and collision bodies. A user must provide all the model parameters, together with the collision bodies, to the path planning server when adding a robot.

See also *Supported robots on page 11*.

Frame numbering

The kinematic model for a robot consists of sequence of moving coordinate frames. Each frame is numbered according to a convention that is specific to each robot variant. When adding collision bodies that model robot links, each collision body is attached to the corresponding frame using the frame number. The path planning server uses the same frame numbering convention as the *Collision Avoidance* functionality in RobotWare. See the following figure for an example of numbering convention used for elbow robots. For the elbow robots the convention is easy to remember: frame number *i* moves with joint *i* and frame *0* correspond to the base frame.



xx2400001770

Continues on next page

Robot mounted on linear axis

The *Automatic Path Planning* supports mounting any robot onto a gantry structure consisting of up to three linear axes. This is useful if the robot is mounted on a linear track motion or hanging from a gantry. The linear axes are specified using the field `gantry_params` in `RobotParameters`.

> **Note**
>
> Possible self-collisions between the linear axes and the robot must be specified in the field `self_collisions` in `RobotParameters`.

4.2 Kinematics

Introduction

Automatic Path Planning provides services for kinematics operations such as forward and inverse kinematics. These services can be used to avoid the need for communication with the robot controller and RAPID operations that may have a high latency for an application running on a PC. The latency of using these services will likely be dominated by the communication overhead of 1-2 ms per message, where a message can contain an array of targets as inputs to kinematics operations.

Forward kinematics

There are two services for forward kinematics.

Service	Description
`forwardKinematicsTCP`	Expects an array of JointTargets and returns an array of RobTargets, one per each JointTarget and ordered as the sent array of JointTargets. It also returns an array of statuses, reporting the success of the forward kinematic operation. Forward kinematics can fail, for example, if the JointTarget is outside the joint limits of the robot.
`forwardKinematics`	Expects an array of JointTargets and returns an array of pose arrays, one pose per robot joint. This service can be used along with the `getRobotCollisionBodies` service to visualize a robot at a given JointTarget.

Both services also return a status per target, as forward kinematics can fail for certain robot variants.

Inverse kinematics

The `inverseKinematics` service expects an array of RobTargets and returns an array of JointTargets and an array of statuses ordered as the sent RobTarget array. When inverse kinematics fails for a target, the status will specify the type of failure as one of the following:

- ERR_INVKIN_POS_OUT_OF_REACH: RobTarget is outside of robot reach
- ERR_INVKIN_WRIST_SINGULARITY: RobTarget is in wrist singularity
- ERR_INVKIN_WCP_SINGULARITY: RobTarget is in shoulder singularity
- ERR_STATE_LIMITS: No solution was found within robot joint limits.

Note

7 DoF and the GoFa robots rely on iterative inverse kinematics which will have a higher latency than other robots.

4.3 Collision body geometries

Introduction

In *Automatic Path Planning*, the physical world is represented as sets of collision bodies. Both robot links and obstacles are represented with collision bodies. Each collision body has a unique integer ID that is generated when the body is added to the server. A collision body includes a geometrical representation that is used in collision checking.

There are two types of geometrical representations available, mesh and voxels. Collision checking with mesh geometries is computationally very efficient and should be considered as the best option when possible. However, when dealing with an unstructured environment where perception is involved, a voxel-based representation is used.

Continues on next page

4.3.1 Mesh

Introduction

A mesh geometry, or to be more precise a triangular mesh geometry, is a common collision geometry representation that consists of vertices (float vector of x, y, and z in meters) and triangles (a triangle is represented by a vector of 3 integers, where each integer represents the index of a vertex). The proto message Mesh consists of two arrays:

```
message Mesh
{
  repeated double vertices = 1;
  repeated int32 triangles = 2;
}
```

The vertices array is the concatenation of all the vertices' positions, that is, `[v0_x, v0_y, v0_z, v1_x, v1_y, v1_z, … ]`. The triangles array contains the concatenation of all the triangles. Each triangle consists of 3 vertex indices, for example, `[v0, v1, v2, v9, v3, v5, ….]`. For instances, the 8 vertices and 12 triangles of a 0.2 m cube centered at the origin can be:

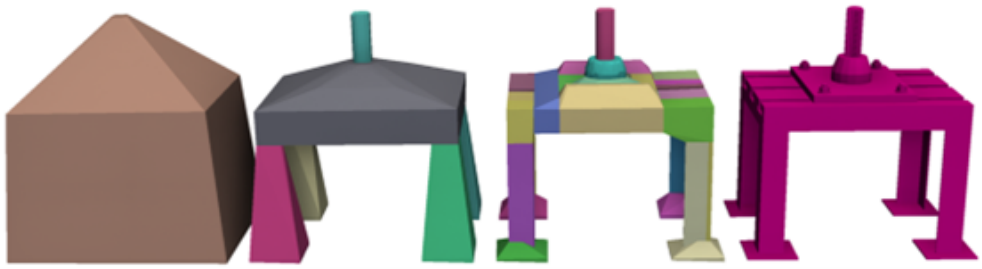
Vertices: [-0.1, -0.1, 0.1, 0.1, -0.1, 0.1, -0.1, 0.1, 0.1, 0.1, 0.1, 0.1, -0.1, -0.1, -0.1, 0.1, -0.1, -0.1, -0.1, 0.1, -0.1, 0.1, 0.1, -0.1]

Triangles: [0, 1, 2, 1, 3, 2, 2, 3, 7, 2, 7, 6, 1, 7, 3, 1, 5, 7, 6, 7, 4, 7, 5, 4, 0, 4, 1, 1, 4, 5, 2, 6, 4, 0, 2, 4]

Convex decomposition

Although it is possible to check whether two generic mesh geometries are in collision, collision checking can be performed more efficiently if the meshes are convex. Therefore, *Automatic Path Planning* uses an approximation of generic mesh geometries referred to as convex decomposition, where a generic mesh object is approximated by a number of convex hulls. The higher the number of convex hulls, the tighter the approximation will be, as shown in the following figure.

The service `generateConvexDecomposition` enables the users to send a generic mesh object, along with an approximation level specified by the desired number of convex hulls and receive an array of mesh objects, each representing a convex hull.



xx2400001456

Three convex decompositions of a generic mesh shown on the right, with increasing number of convex hulls (1, 6, 32).

Continues on next page

4.3.1 Mesh
Continued

Practical notes

Note that a higher number of convex members will result in longer collision checking times per obstacle. Therefore, use the lowest number of convex hulls that suits the application requirement. Note that this is especially important for robot attachments (for example, robot tool) because any robot collision geometry will have to be checked against all the obstacles and a robot attachment with more than a few convex hulls can have a significant impact on planning time.

The convex decomposition algorithm is rather time consuming and must be avoided in real-time applications. The higher the accuracy of the decomposition, the longer the procedure will take. It is recommended to use the *Automatic Path Planning* to generate convex decomposition and save the results to disk, such that at runtime, the decomposed collision geometry can be set directly and avoid calling the convex decomposition service.

4.3.2 Voxelized environment (point cloud)

Introduction

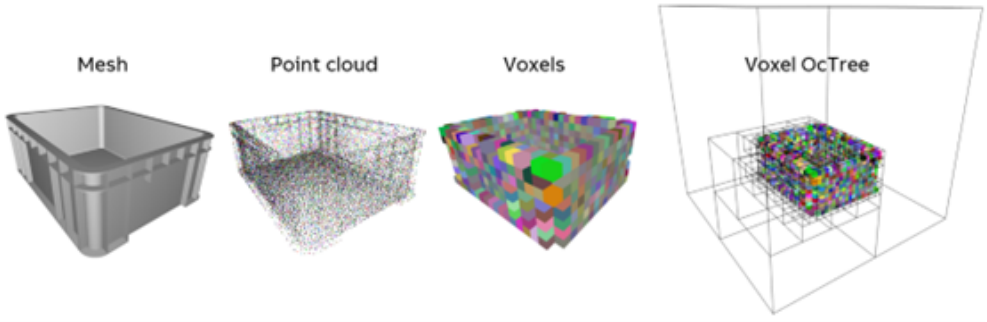
In some applications the CAD model of the robot environment (or parts of it) is either not available or not possible to design and localize in advance. In these cases, a vision system can be used instead to generate a point cloud of the robot workspace. To address such applications, the path planning server uses a voxelized representation of point clouds that reduces collision checking computations. A voxel is the 3D counterpart of a pixel, and therefore shaped as a cuboid. The voxel representation of a shape is a discretized version of the geometry. The smaller the size of voxels, the more accurate the discretization will be, but also the more costly memory/compute requirements.

Sending point clouds

Only one collision body with voxels is allowed at a time. Sending a new voxelized environment overwrites a previously sent one.

There are 2 ways to send point clouds to the automatic path planning server:

- Point cloud: In this approach the x-y-z coordinates of each point are sent to the server. Although this is the most straight forward approach, when a large number of points are present, the size of the message to the server can become large which increases communication latencies.
- OcTree: An OcTree (Octant Tree) is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space (a cube) by recursively subdividing it into eight octants. Using OcTree, the parts of the workspace that are empty can be efficiently represented, see the following figure. Furthermore, using this encoding the message's size can be greatly reduced, so less time is spent on communication.



xx2400001457

An example of point cloud, voxel, and voxel OcTree representations.

4.4 Collision checking

Check for collision at targets

Use the service `collisionCheck` to check whether the robot is colliding with itself or with an obstacle at the provided array of targets. The service replies with an array of `CollisionCheckData`, with one element for each specified robot target. The returned information includes details such as which robot collision body is colliding with which obstacle collision body. If a voxel-based representation is used, then a list of voxels in collision with the robot is also returned.

The minimum-distance between each colliding pair is also returned. If the objects are penetrating, then the reported distance is zero, so no negative values.

A `min_distance_to_obstacles` parameter can be provided, to specify below which link-obstacle distance that an obstacle collision is identified. The tolerance for self-collisions is determined by the parameter `min_dist_self_collision`, which is set by the service `addRobot`. For information about minimum distances, see *Minimum allowed distance on page 33*.

Path collision check

Similar to collision check at targets the `PathCollisionCheck` service can be used to check collision status of a list of `MoveCommand` path. In contrast to the collision check, the response includes only a status per path that signifies whether the path is collision-free or not and no more details are provided.

4.5 Path planning

4.5.1 Path characteristics

Introduction

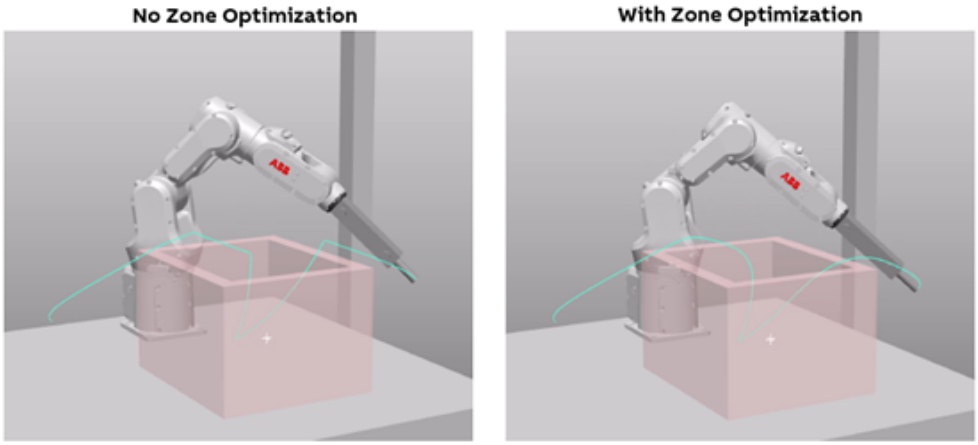
The purpose and main service of *Automatic Path Planning* is to generate collision-free paths for point to point or pick and place tasks. The service relies on efficient algorithms developed to produce optimal paths (as in shortest path) within a short computation time. In the following we discuss the characteristics of path planning in *Automatic Path Planning*.

MoveCommand-based

Collision-free paths found by *Automatic Path Planning* comprise Move commands (`MoveAbsJ` for point-to-point planning, and `MoveAbsJ` and `MoveL` for pick and place planning). These commands can be directly converted to the corresponding RAPID instructions and sent to the robot controller to benefit from the superior motion and accuracy performance offered by ABB robots.

Optimized zones

The generated joint space path includes the largest collision-free zone sizes per `JointTarget`. This unique feature of the path planning server leads to paths that are highly smooth (see the following figure) and have faster cycle time and potentially lower energy consumption.



xx2400001458

Short lengths

The path planning objective is to find the shortest possible collision-free path from start to goal. However, it cannot be guaranteed that the globally shortest path can be found within a limited time. The path length is generally improved by increasing `optimality_effort`, but the planning time will then increase.

Continues on next page

4.5.1 Path characteristics
Continued

Joint space

The collision-free path returned by *Automatic Path Planning* is an array of JointTargets each with a zone size, to be sent to the controller by the user as a series of `MoveAbsJ` instructions. This means that the path is singularity free. Furthermore, in most cases, the shortest joint space path can lead to smoother and faster motion.

No redundant via points

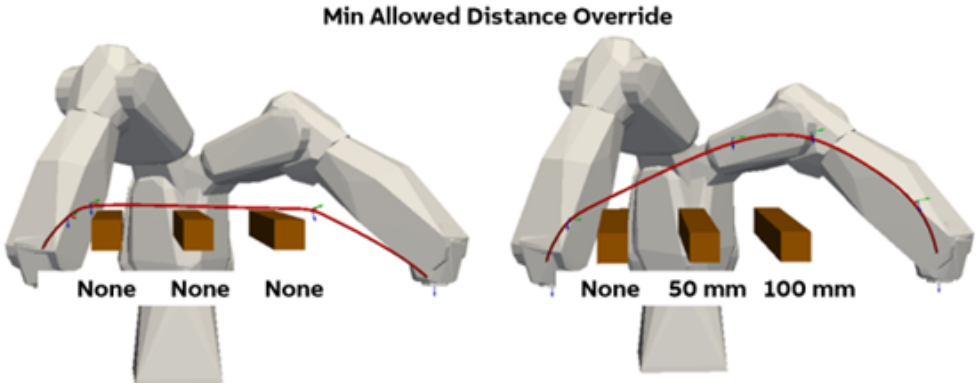
An *Automatic Path Planning* path is processed to have shorter length, while having fewer number of via points. As a result, redundant via points are removed when it is possible to make a shortcut between non-consecutive points.

4.5.2 Minimum allowed distance

Introduction

To ensure a safe distance in the collision-free path planning, a parameter called `min_distance_to_obstacles` can be set. This parameter guarantees that none of the robot's links or attachments come closer to obstacles than the specified distance while moving along the path. It is important that the start and goal targets also adhere to this minimum distance. Otherwise, the server will return a collision error. To avoid numerical issues during collision checking, the smallest allowed safety margin is 1 mm.

To allow for more granular control over the safety margin, when adding obstacles or robot attachments such as tools and loads, a `min_allowed_distance_override` parameter can be provided. In such cases when checking the collision state of the specified obstacle or robot attachment, the larger of the override and the global min distance are chosen. This is visualized in the right hand figure, the obstacles from left to right have none, 50 mm, and 100 mm set as their `min_allowed_distance_override` parameter when they are added to the server. When a path planning query with `min_distance_to_obstacles` of 2 mm is sent to the server the path is generated such that the robot links do not get closer than 2, 50, and 100 mm respectively to the obstacles from left to right.



xx2400001460

On the right hand figure, the obstacles from left to right have none, 50 mm, and 100 mm set as their `min_allowed_distance_override` parameter when they are added to the server.

Path planning computation effort and duration

Finding an optimum path is an optimization problem may take infinite time. The automatic path planning server provides four parameters that can be used to shape the computational effort spent on a path planning query.

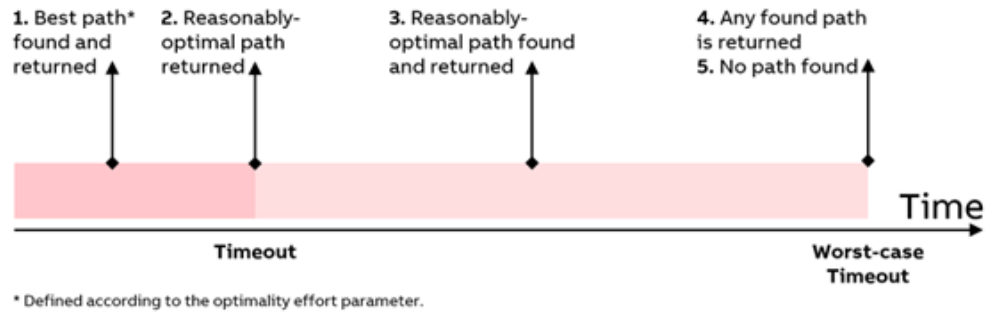
- Number of threads: The automatic path planning server can use multiple threads to speed up the planning computations. It is recommended to use at least 4 threads.
- Optimality effort: A value between 0 and 1 that specifies how much computational effort should be spent on finding an optimal path (referred to

Continues on next page

as "best path"). For online applications where computation time is limited a value of 0.1 is recommended. For offline applications, a value of 0.5 is recommended.

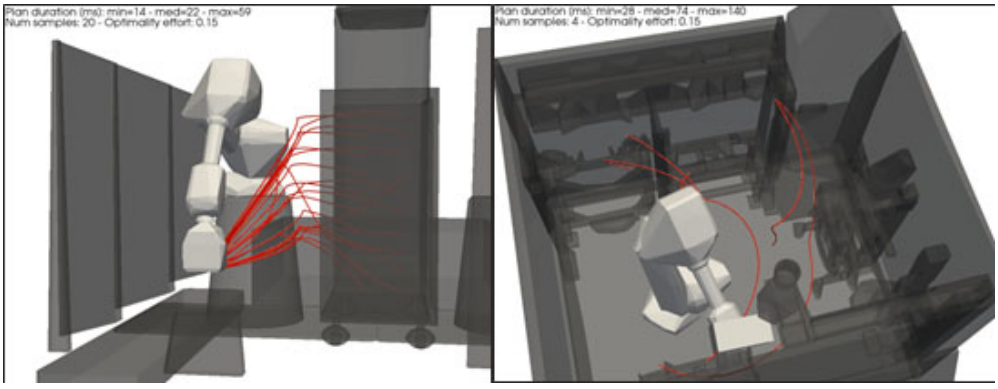
- Timeout: The path planner has internal criteria for having found a reasonably-optimal path. When the timeout is expired, when the reasonably optimal criteria are met, the server does not search anymore and returns the found path.
- Worst-case timeout: If the path planner's criteria for having found a reasonably optimal path are not met by the time the timeout has expired, the server will search until either a reasonably optimal path is found, or the expiration of `timeout_worst_case` is reached.

The following figure shows five possible outcomes of a path planning query. The timeout and optimality effort parameters should be set so that in most queries the best path (defined based on the specified optimality effort parameter) is found before the timeout is reached.



xx2400001771

It is not possible to provide a general rule of thumb for how long a planning query may take to solve, as that depends on the complexity of the problem and the performance of the available hardware. Complexity can arise from environment representation (a large number of collision bodies, robot attachments with many convex hulls, or a large number of voxels), or the difficulty of finding a collision-free path (for example, very tight clearances at some point along the path from start to goals).



xx2400001461

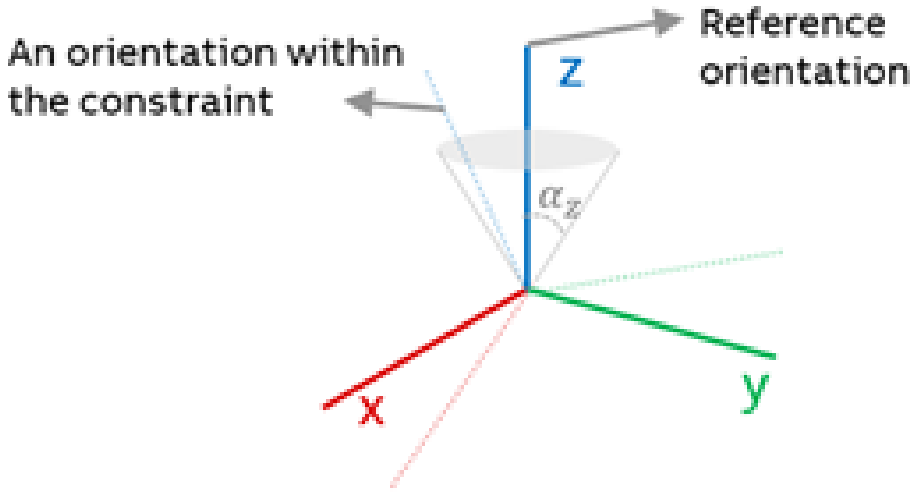
Two examples of planning duration with an optimality effort of 0.15 and 6 threads utilized on an 8-core Core i7-8700 3.7 GHz CPU. Median planning duration for the

Continues on next page

cell on the left with around 27 obstacles (32 convex hulls) was 22 ms, and for the cell on the right with 47 obstacles (314 convex hulls) was 74 ms.

Path orientation constraint

Some applications may require that the orientation of the TCP be constrained along the collision-free path. This is supported by *Automatic Path Planning* and can be specified in a path planning query by setting the orientation constraint parameter. An orientation constraint is specified as three cone angles in radians and in the range of [0, Pi]. Each angle represents the allowed deviation from an axis of a reference orientation. For example, a cone angle of 0.1 radian for Z means that along the path the TCP z axis will not deviate more than 0.1 radians from the z axis of the TCP at the start target of the planning query. If the cone angle is -1, then there is no constraint on the corresponding tool-frame axis. If more than 2 axes are constrained, then the orientation is fixed. This means, at via points the orientation is equal to that of the start target, and the segments between via points can have deviation in orientation up to the specified angles. Note that using very small cone angles will increase the required planning time. Also note that while *Automatic Path Planning* guarantees the path via points to be within the orientation limits, it does not provide such a guarantee for the segments connecting the via points. So at times, there may be some violation of the constraint when moving from one via point to another.



xx2400001462

Representation of an orientation constraint described as α_z radians around the z axis of the tool.

4.5.3 Goals

Introduction

A path planning query is often from the current position of the robot to a single goal target. *Automatic Path Planning* allows for more elaborate goal definitions that can simplify application logics and/or enhance motion performance or efficiency.

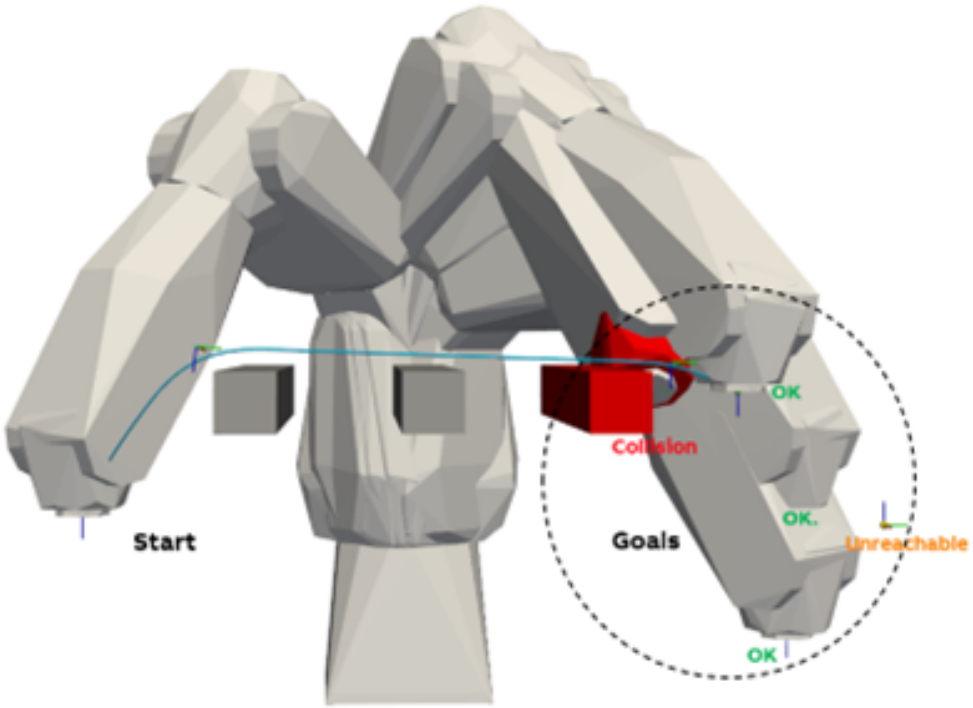
Planning goal with RobTarget kinematic configuration

A RobTarget includes not only the Pose of the TCP but also the kinematic configuration parameters that specify one of the multiple inverse kinematics solutions. When specifying a RobTarget as a planning goal, the kinematic configuration parameters can be left unspecified, such that the path planning selects valid parameters that lead to the shortest path from the start.

If only the CFX parameter of the kinematic configuration is set, then cf1, cf4, and cf6 will be chosen by the path planner. Note that the `arm_angle` parameter shall always be set for 7 DOF robots. See *Technical reference manual - RAPID Instructions, Functions and Data types*.

Multiple goal targets

A collision-free path planning query consists of a single start target and an array of goal targets. The generated collision-free path will reach only one of the specified goals, and often the goal that results in the shortest path from the start. Goals that are kinematically infeasible or are not collision-free will be discarded. A visual example is shown in the following graphic.



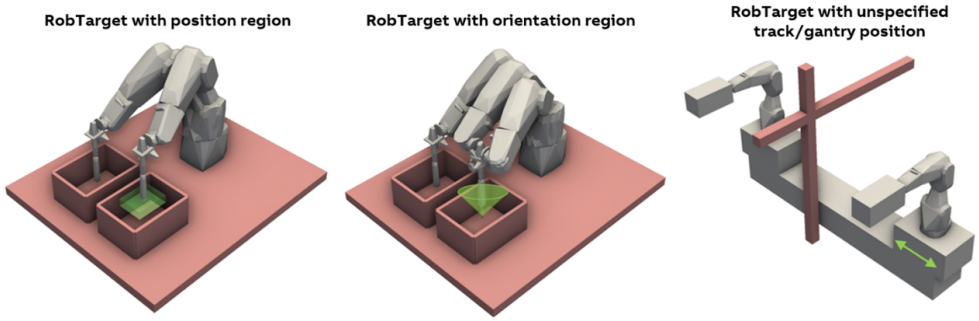
xx2400001463

Continues on next page

Example of planning with multiple goals. 5 goal targets are specified from which 3 are valid but 1 is in-collision, and 1 is out of reach. *Automatic Path Planning* will find the shortest path connecting the start to a valid goal.

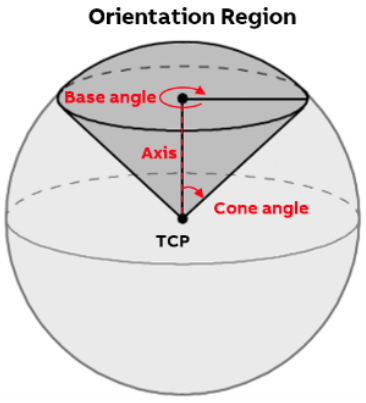
Goal regions

A goal can be turned from a fully specified robot target into a region. *Automatic Path Planning* will try to find a valid target within the goal region that results in the shortest path from the start. A valid target is kinematically feasible, collision-free, and satisfies constraints such as path orientation constraint, if present. Goal regions can optionally be specified in planning targets in addition to a reference RobTarget/JointTarget. The following goal regions are available.



xx2400001772

- Position region: Specifies a box region centered at the TCP position of the provided reference target. The path planner selects a valid goal with TCP position within the box region and with the same TCP orientation as that of the provided reference.
- Orientation region: Specifies a spherical cone around the provided reference as shown in the figure. The path planner selects a valid goal with TCP position matching that of the reference and an orientation that falls within the cone region. The base angle parameter of the orientation region can be used to limit rotations around the cone axis, for instance to protect cables/hoses of flange attachments.



xx2400001773

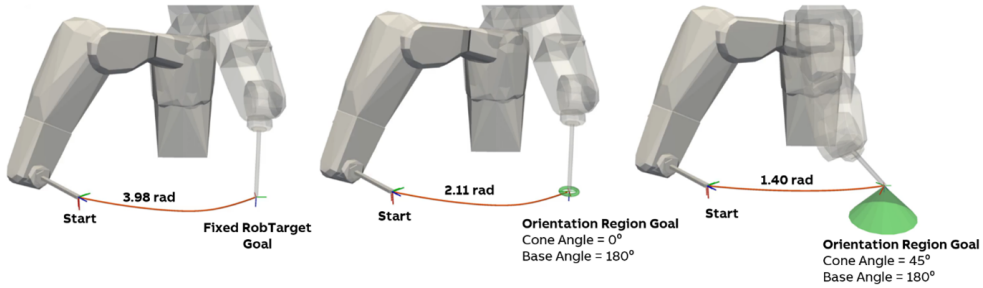
- RobTarget Linear axes-mounted robots: Normally to plan for a linear axes-mounted robot with a RobTarget, the position of the linear axis needs to be specified. *Automatic Path Planning* allows for not specifying those

Continues on next page

4.5.3 Goals
Continued

positions and only specifying the RobTarget of a goal. The path planner will find linear axis positions that lead to the shortest path from the start.

When the application requirement allows for it, using goal regions can lead to significant path length reduction as shown in the following figure.



xx2400001774

4.5.4 Pick-and-place planning

Introduction

The service `planCollisionFreePickAndPlace` is a powerful service that allows for planning of multiple picks in one single call to the server.

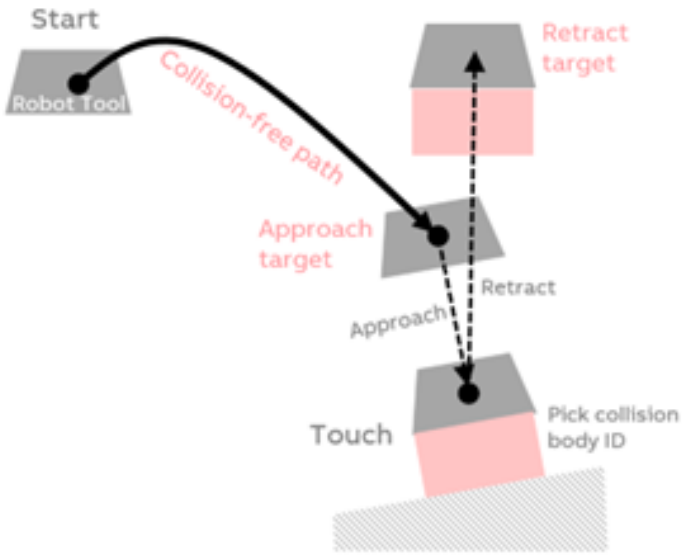
Some applications such as pick-and-place include interacting with the environment. For instance, picking up an object resting on a surface may mean that just before picking the object, the robot needs to touch (that is, come in collision with) the object. Also, just after the pick, the object, that is now part of the robot, will be in collision with the resting surface. Therefore, pick motions are often defined by an approach and retract targets above the object and commonly use a `MoveL` to reduce the chance of unforeseen collision or disturbing the environment. Since a collision-free path by definition cannot have a start or goals where the robot is in collision with the environment, the automatic path planning server uses special touch templates to pick or place objects. Furthermore, there may be several possible grasp poses to pick an object and a path planner should identify the best valid pose among them. To address these 2 needs, the automatic path planning server offers a pick-and-place planning service. A pick-and-place planning query provides a high degree of flexibility by includes an array of touch queries, where a touch query can specify picking or placing one or multiple objects. For instance, a single pick-and-place query can specify picking 3 objects at once and placing them in 3 different locations and going to a final target and the path planner finds the collision-free path between these steps.

A minimal pick-and-place query is shown in the following figure, where an object is picked by the robot tool. The query includes a start target, and a single touch query. The touch query includes the description of the approach and retract movements (for example if the movement is linear Cartesian or in Joint space) and the ID of the collision body to be picked at the single provided target. Upon receiving such a query, the path planner generates valid approach/retract/pick targets, plans a collision-free path from the start to the approach, and returns the overall path from the start to the retract target to the user. The returned path is divided into segments to allow for inserting other commands such as opening the gripper before an approach segment, and so on. Note that, after the conclusion of the planning,

Continues on next page

4.5.4 Pick-and-place planning
Continued

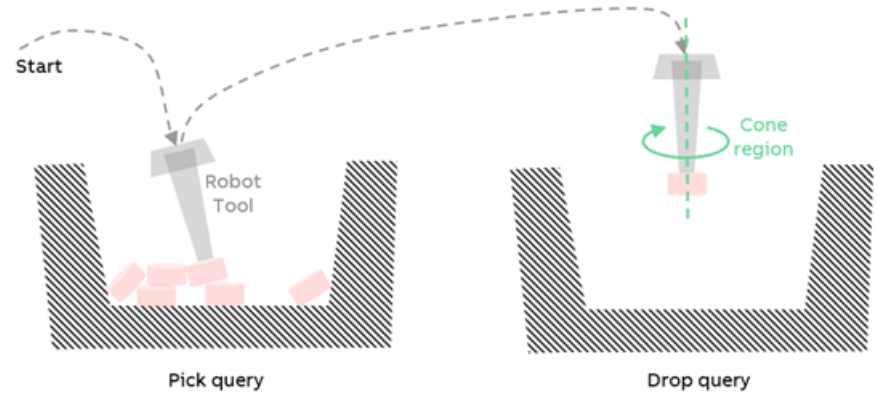
the picked obstacle is attached to the robot and will be considered in collision checking unless it is placed or removed by the user.



xx2400001464

Example 1

Example of picking an object where an obstacle collision body becomes a robot attachment through approach and retract motions.



xx2400001778

Example 2

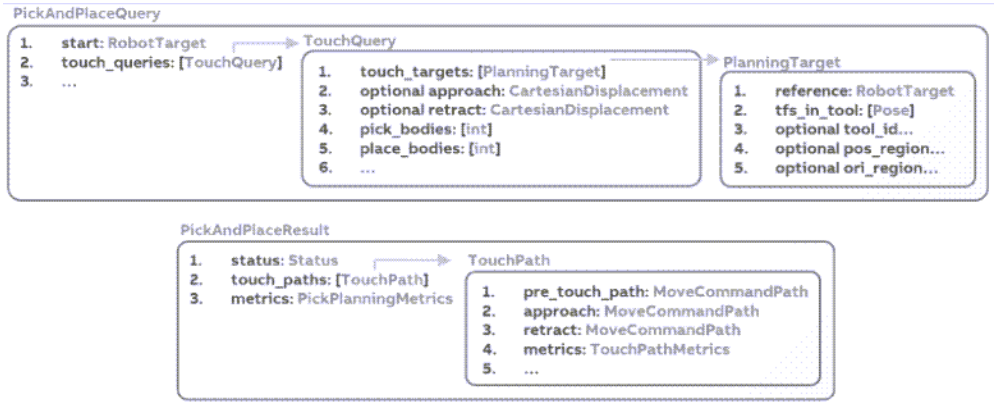
Example of a pick and place planning query with 5 touch queries where query 0 specifies picking 2 objects that are placed in queries 2 and 3. Queries 1 and 4 specify via points (targets that the robot should visit but that include no pick or place).

Continues on next page

Touch query specification

The following figure shows the structure of pick-and-place planning query and result. A pick-and-place planning query comprises an array of touch queries that describe the steps of a pick-and-place procedure. For instance, a two-step procedure of picking an object from one location and placing it at another location includes two touch queries, the first of which describes the pick step and the second the place step.

The structure of pick and place planning query and response



xx2400001776

Contents of a touch query

- An array of `PlanningTargets` from which a single target will be selected by the planner for the touchpoint. The selected target must be valid (in terms of kinematics, collision criteria, and constraints). Furthermore, the planner tries to find the target that results in the shortest overall path from the start.
- Optional approach and/or retract moves specified by parameters such as length, direction, interpolation type (resulting in `MoveL` or `MoveAbsJ`) etc. The planner tries to generate valid (in terms of kinematics, collision, and constraints) approach/retract targets from the specification. Note that if no approach is specified in the query, the planner attempts to plan a collision-free path from the start to the touchpoint.
- An optional array of bodies to be picked (converted from an obstacle to a robot attachment) at the touchpoint.
- An optional array of bodies to be placed (detached from the robot and converted to an obstacle) at the touchpoint.
- Other parameters affecting the path planning and collision checking.

Continues on next page

4.5.4 Pick-and-place planning
Continued

Specifying multiple pick/place poses

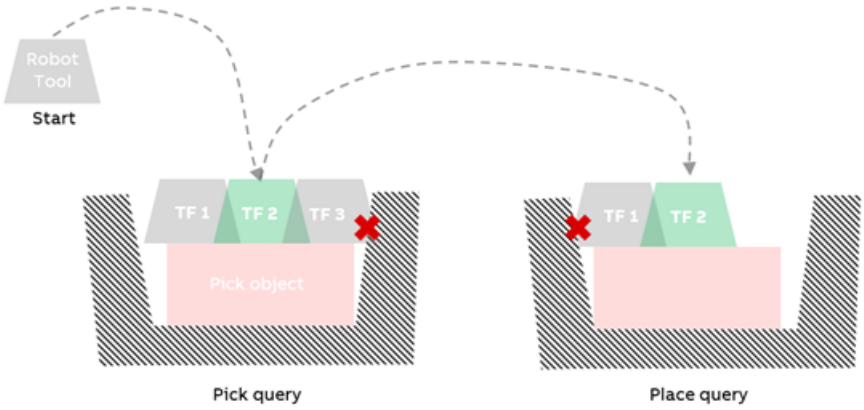
In many pick and place applications it is possible to pick/place an object with several poses. For instance, when multiple grasping candidates are provided by a vision system for picking an item, all these poses can be provided to the Automatic Path Planning server such that the best is selected for the path. A touch query comprises an array of PlanningTargets each of which can specify multiple targets. The planner stops searching after having generated a valid path to a planning target starting from the first one. Therefore, the user can specify priority of provided planning targets by their order in the `touch_queries` array.

A planning target can have an optionally specified tool id. For instance, it may be possible to grasp a ring from the outside with an open gripper or from the inside with a closed gripper. In such a case 2 planning targets can be provided, one with the tool id of the closed gripper and the other with the tool id of the open gripper.

Several pick/place poses can be specified within a single planning target. To do so one can provide a reference `RobTarget` or `JointTarget` and an array of poses, referred to as `tfs_in_tool` for transformations in the tool coordinate frame. These poses transform the reference's pose in the tool coordinate frame. An example is shown below where the pick query has a planning target with 3 `tfs_in_tool` shown as TF1, TF2, and TF3. TF1 is an identity transform and TF2 and TF3 are translations along a single axis. As depicted in the image, only one of the transforms results in collision-free pick and place targets.

Example

This is an example of specifying multiple `tfs_in_tool`. The query has 2 touch queries, one for pick and the other for place. The pick query has a single planning target with 3 `tfs_in_tool`, one of which results in robot colliding with the environment at pick and the other results in collision at place. Therefore, the last remaining transform is chosen by the planner.

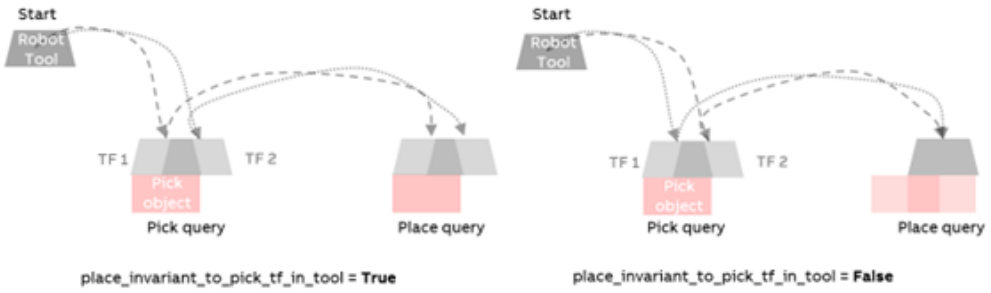


xx2400001795

An advantage of providing the transforms relative to a reference is that the planner can ensure a fixed place pose for the item if queried. That is, if it is of interest to place the picked object always in the same location despite the pick `tf_in_tool`, as shown to the left in the following figure, the `place_invariant_to_pick_tf_in_tool` flag of the pick-and-place query should

Continues on next page

be set to *True*. When the flag is set to *False* the place target is not transformed as shown on the right and the picked object is placed in different locations.

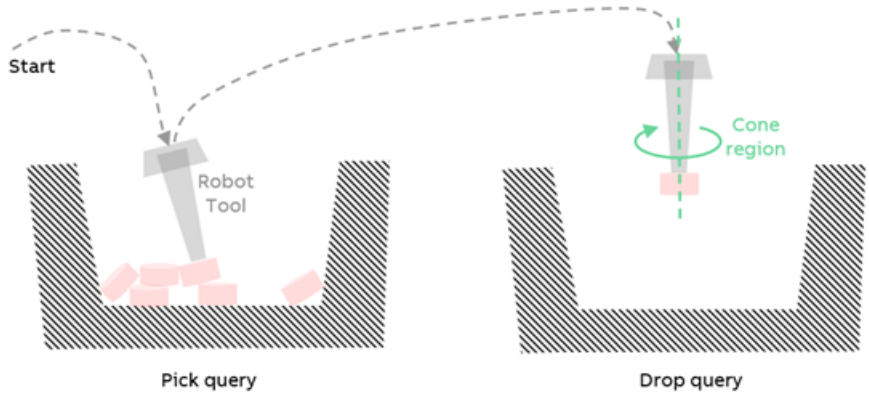


xx2400001465

Another way of specifying flexibility to improve robustness and/or productivity is to use the goal regions field of a planning target. For instance, when dropping an item in a bin the orientation around the vertical axis can be left free by specifying a cone region as shown in the following example. Relaxed targets can result in a significantly shorter path.

Example

Example of dropping an item in a bin while exploiting the cone region specification to free the orientation around the vertical axis.



xx2400001778

In applications where several items are located at once, an array of pick-and-place queries can be sent to the server in one request, one query per item. This can reduce the communication overhead. Using the parameter `num_queries_to_solve`, it can be specified after how many solved queries the planning should stop. When set to 1, the planner starts from the first query and returns as soon as one query is solved.

Continues on next page

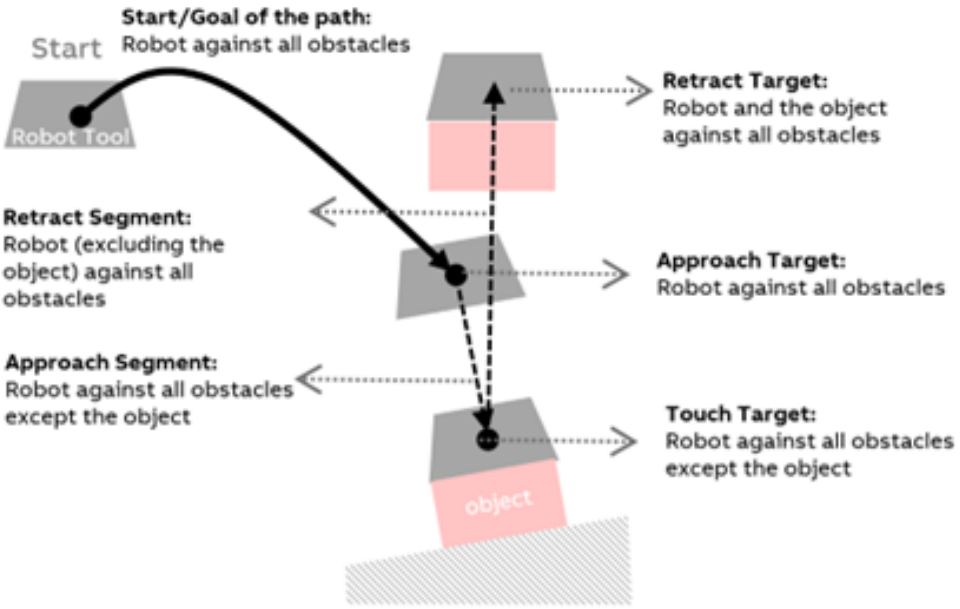
4.5.4 Pick-and-place planning
Continued

Touch query collision checks

A touch query includes various collision checks to ensure no undesired collision occurs while allowing for the touch between the picked item and the robot. These collision checks are shown in the following figure. What can add further complexity is different collision thresholds (allowed minimum distances) in the query.

- In the simplest case the `min_distance_to_obstacles` parameter of the `PlanningParameters` of the pick-and-place query can be used for all collision checks.
- Optionally, the user can set a different `PlanningParameters` for each touch query that override the `PlanningParameters` of the pick-and-place query.
- Furthermore, the user can specify a different collision threshold to be used at the touch target and the approach/retract segments. This can be specified by setting the `min_distance_at_touch` field of a touch query. If set to a negative value, no collision check is performed at the Touch target and the retract/approach segments. This should be done with caution and only in extraordinary cases.

Note that when `min_distance_at_touch` is specified, at the approach and retract targets, the strictest collision criterion must hold. For instance, if the pick-and-place query has a `min_distance_to_obstacles` of 10 mm but the `min_distance_at_touch` is set to 5 mm, at the approach target the stricter 10 mm threshold must hold for the path planning to succeed.



xx2400001779

Index

ABB AB
Robotics & Discrete Automation
S-721 68 VÄSTERÅS, Sweden
Telephone +46 10-732 50 00

ABB AS
Robotics & Discrete Automation
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 22 87 2000

ABB Engineering (Shanghai) Ltd.
Robotics & Discrete Automation
No. 4528 Kangxin Highway
PuDong New District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

ABB Inc.
Robotics & Discrete Automation
1250 Brown Road
Auburn Hills, MI 48326
USA
Telephone: +1 248 391 9000

abb.com/robotics

3HAC092826-001, Rev A, en