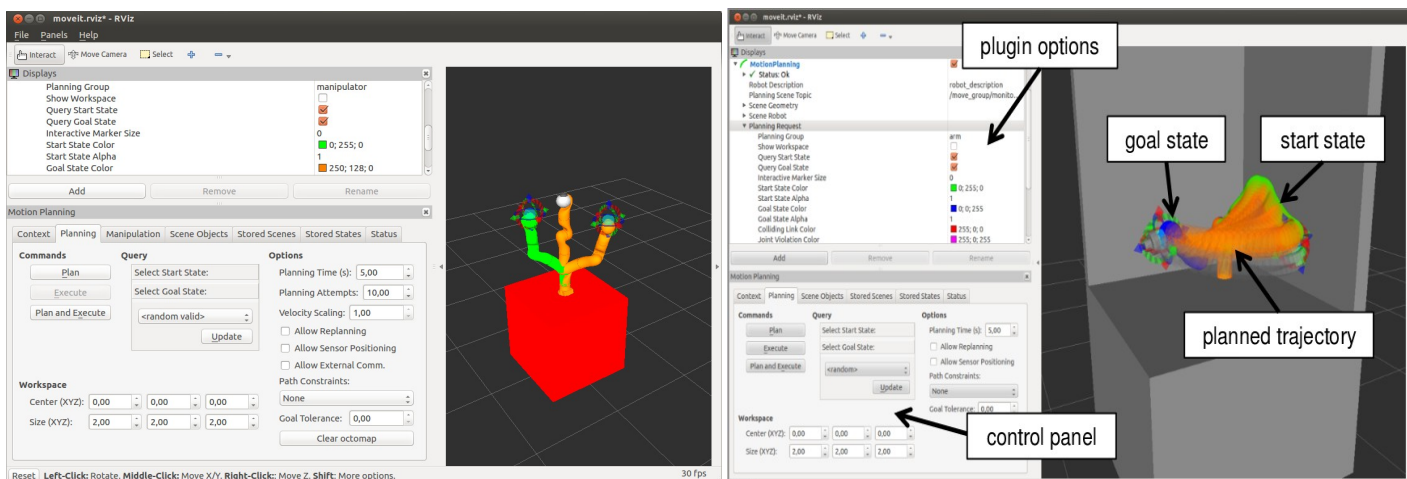


## Notions de Move it!, move\_group

'MoveIt!' est un ensemble d'outils permettant de faire la planification de mouvement ('motion planning'), de simuler une saisie ('grasping'), de prendre un objet et de le placer ('pick and place'), de faire de la détection de collisions ('collision checking'), de traiter des données de capteurs 3D ('perception'), de la navigation ...

Dans cette partie, on va s'intéresser plus particulièrement à la planification de mouvement.

Un 'plugin' nommé 'MoveIt! Rviz motion planning' a été ajouté dans l'outil de visualisation 'Rviz'. Celui-ci permet de définir graphiquement une position de départ et d'arrivée d'un robot, de lancer le calcul de la planification, d'afficher le résultat. On peut aussi le faire par programmation en utilisant la librairie C++ nommée 'move\_group'. La planification va tenir compte des contraintes exprimées dans le modèle du robot (par exemple : articulaire, vitesse ...).



<http://moveit.ros.org/documentation/concepts/>

Par défaut 'MoveIt' utilise :

- L'outil de planification OMPL ('Open Motion Planning Library' <http://ompl.kavrakilab.org/>).
- L'outil de détection de collisions FCL ('Flexible Collision Library' <https://github.com/flexible-collision-library/fcl> )
- L'outil de résolution de la cinématique inverse : The default IK solver ('Inverse Kinematic Solver') in MoveIt! Is a numerical jacobian-based solver.
- Une représentation de l'environnement, de sa perception 3D, par 'Octomap' (<http://wiki.ros.org/octomap>)

Ces outils sont vus comme des 'plugin' qui peuvent être utilisés par défaut, ou bien être développés

et interfacés par vos soins (par exemple on peut développer son propre outil de planification).

Tutorial Rviz (<http://wiki.ros.org/rviz> )

1- Utiliser le plugin 'MoveIt! Rviz motion planning'.

Notre objectif est de créer un paquet 'MoveIt!' qui nous permettra de tester une planification de trajectoire du bras 'kuka' entre 2 poses. Cette planification fera en sorte d'éviter les collisions avec l'environnement, les auto collisions, tout en tenant compte des limites articulaires définies dans le modèle du robot.

Pour se faire, on va utiliser un assistant graphique, pour générer les fichiers paramètres nécessaires à 'MoveIt!'.

[http://docs.ros.org/indigo/api/moveit\\_setup\\_assistant/html/sources/doc/tutorial.txt](http://docs.ros.org/indigo/api/moveit_setup_assistant/html/sources/doc/tutorial.txt)

Lancer l'assistant :

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

Suivre ces options :

Cliquer sur le bouton '**Create new moveit configuration package**'

Cliquer sur le bouton '**Browse**' et sélectionner le dossier '**model**' du paquet '**kuka\_lwr\_description**'. Dans ce dossier sélectionner le fichier '**platform.urdf.xacro**' qui contient la description 'urdf' du robot.

Cliquer sur le bouton '**Load files**'

- A ce stade, vous devez voir apparaître le robot sur la fenêtre de droite.

\* Onglet '**Self-Collisions**' cliquer sur le bouton '**Regenerate Default Collision Matrix**' (Permet de calculer/vérifier les collisions entre 'links')

\* Onglet '**Virtual Joints**' cliquer sur le bouton '**Add Virtual Joint**' :

- name = '**FixedBase**'
- child link = '**box**'
- Parent Frame name = '**world**'
- Joint Type = **fixed**

Cliquer sur le bouton '**Save**'

(Permet d'attacher le robot au monde)

\* Onglet '**Planning Groups**' cliquer bouton '**Add group**' :

- name = '**manipulator**'
- Kinematic solver = **kdl\_kinematics\_plugin/KDLKinematicsPlugin**
- cliquer sur bouton '**Add Kin. Chain**'
- cliquer sur '**box**' puis sur bouton '**choose selected**' de '**base link**'
- cliquer sur '**kuka\_lwr\_7\_link**' puis sur bouton '**choose selected**' de '**tip link**'
- cliquer sur bouton '**Save**'

(Permet de créer un groupe contenant une partie du robot et de lui associer un 'plugin' solveur cinématique basé sur la librairie KDL <http://www.orocos.org/kdl>)

\* Onglet '**Robot Poses**' – Rien facultatif pour cet exemple

(Permet de définir des poses spécifiques du robot, qui pourront être utilisées avec le 'plugin rviz')

\* Onglet '**End effectors**' – Rien facultatif pour cet exemple  
(Permet de définir un effecteur au bout du bras, dans notre cas il n'y en a pas)

\* Onglet '**Passive Joints**' – Rien facultatif pour cet exemple  
(Permet de définir quels 'joints' ne bougent pas, dans notre cas il n'y en a pas)

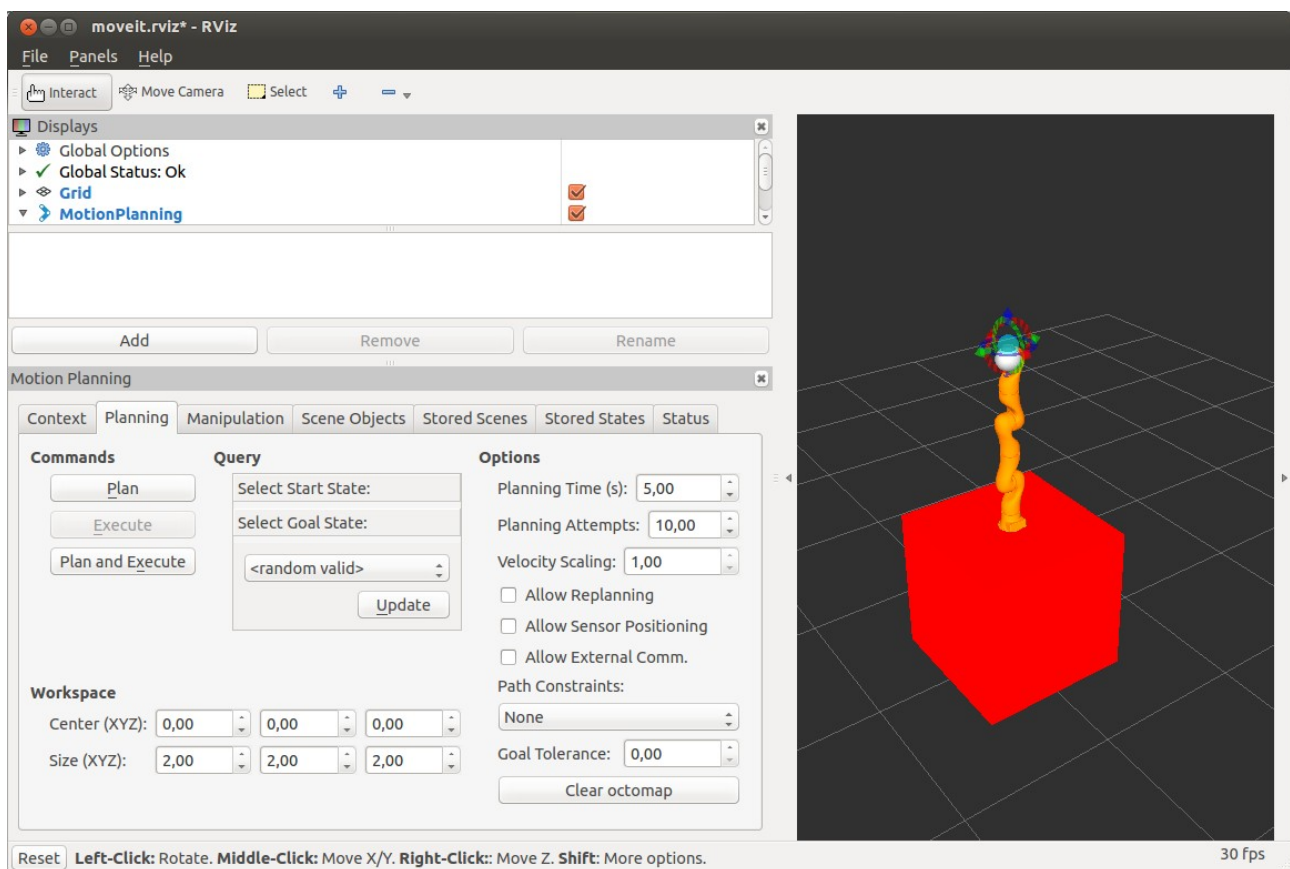
\* Onglet '**Configuration files**' cliquer sur bouton '**browse**', sélectionner le dossier '**src**' du '**workspace**' et créer un nouveau dossier nommé '**kuka\_lwr\_moveit**'.  
- cliquer sur bouton '**Generate package**'.  
- cliquer sur le bouton '**Exit Setup Assistant**'  
(Permet de créer tous les fichiers de configuration nécessaires à 'MoveIt!' dans le dossier sélectionné)

Remarque : Si vous souhaitez modifier la configuration générée dans le paquet '**kuka\_lwr\_moveit**', il faut relancer l'assistant, cliquer sur le '**Edit Existing Moveit ...**', cliquer sur le bouton '**Browse**' et choisir le dossier '**kuka\_lwr\_moveit**' du '**workspace**' puis cliquer sur le bouton '**Load Files**'.

Relancer une compilation 'catkin' de votre 'workspace'.

Lancer le 'launch' du paquet généré :  
roslaunch **kuka\_lwr\_moveit** demo.launch

'Rviz' se lancera avec le plugin 'Moveit', permettant de faire des simulations de trajectoires.



Pour simuler une trajectoire :

Dans la Fenêtre '**Displays**' déployer l'option '**Planning Request**' puis cocher l'option '**Query Start State**' (Observer que le 'start state color' sera par défaut en vert et le 'goal state color' en orange).

\* Onglet '**Context**' :

- Si la librairie de planification par défaut qui est 'OMPL' ( <http://ompl.kavrakilab.org/> ) a bien été chargée, vous devez voir le mot 'OMPL' en vert.

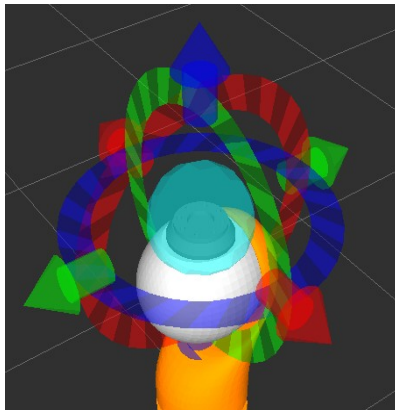
\* Onglet '**Planning**' :

→ '**Select Start State**' (position de départ) : Se servir des poignées ('markers') pour bouger le bras en vert.

→ '**Select Goal State**' (position à atteindre) : Se servir des poignées ('markers') pour bouger le bras en orange.

(Vous pouvez aussi utiliser l'option 'random valid' puis cliquer sur le bouton 'update')

→ Enfin : '**Plan**'.



Red = Position en X,  
Green = Position en Y,  
Blue = Position en Z

Remarque : Si vous souhaitez ne plus afficher la position initiale du bras, il faut décocher 'Show Robot Visual' de l'option 'Scene Robot' de la fenêtre 'Displays'.

2- Simuler une collision en important un objet dans la scène.

Dans l'onglet '**Scene Objects**' cliquer sur le bouton '**Import File**' et choisir un des fichiers au format 'collada' (extension '.dae') du dossier :

**'tp\_ros\_master\_robotique/Ressources/kuka/object\_collision'**

Si vous obtenez un message d'erreur lors de l'import, c'est un bug dû au nom du dossier ... copier le dossier 'object\_collision' sur votre 'Bureau' par exemple et relancer l'import.

Vous pouvez modifier le facteur d'échelle de l'objet et sa position, en modifiant le 'slider scale' et les valeurs 'position' et 'rotation'. Vous pouvez aussi agir graphiquement sur le 'marker' de l'objet.

Placer l'objet entre la position de départ du bras et sa position à atteindre.

Dans l'onglet '**Context**' cliquer sur le bouton '**Publish Current Scene**' afin de prendre en compte ce nouvel objet importé.

Puis cliquer sur le bouton '**Plan**' de l'onglet '**Planning**'

Observer le déplacement du bras qui doit éviter l'objet.

Attention : si après le 'Plan' vous obtenez un message '**failed**', soit la position demandée avec l'objet à éviter n'est pas atteignable (limites articulaires), soit il y a déjà une collision avec une partie du bras (symbolisée en rouge).

2- Utiliser l'Api C++ '**move\_group**' pour interagir avec 'MoveIt!'.

Créer un nouveau paquet nommé 'my\_first\_move\_group' avec des dépendances :

```
catkin_create_pkg my_first_move_group catkin cmake_modules interactive_markers moveit_core moveit_ros_perception
moveit_ros_planning_interface pluginlib roscpp std_msgs
```

cmake\_modules : [http://wiki.ros.org/cmake\\_modules](http://wiki.ros.org/cmake_modules)

interactive\_markers : [http://wiki.ros.org/interactive\\_markers](http://wiki.ros.org/interactive_markers)

moveit\_core : [http://wiki.ros.org/moveit\\_core](http://wiki.ros.org/moveit_core)

moveit\_ros\_perception : [http://wiki.ros.org/moveit\\_ros\\_perception](http://wiki.ros.org/moveit_ros_perception)

moveit\_ros\_planning\_interface : [http://wiki.ros.org/moveit\\_ros\\_planning\\_interface](http://wiki.ros.org/moveit_ros_planning_interface)

pluginlib : <http://wiki.ros.org/pluginlib>

Créer un fichier c++ nommé '**test\_random.cpp**' permettant la planification du bras à une position aléatoire :

```
#include <moveit/move_group_interface/move_group.h>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "test_random_node", ros::init_options::AnonymousName);

    // start a ROS spinning thread
    ros::AsyncSpinner spinner(1);
    spinner.start();

    // setup the group you would like to control and plan for.
    move_group_interface::MoveGroup group("manipulator");

    // specify that our target will be a random one
    group.setRandomTarget();

    // plan the motion and then move the group to the sampled target
    group.move();

    // wait for shutdown
    ros::waitForShutdown();
}
```

Ici on crée un 'MoveGroup' qui correspond au '**Planning Groups manipulator**' défini précédemment avec l'assistant :

```
move_group_interface::MoveGroup group("manipulator");
```

On lui spécifie une position aléatoire à atteindre du bout du bras :

```
group.setRandomTarget();
```

On demande d'effectuer la planification et le mouvement :

```
group.move();
```

Modifier le fichier '**CMakeLists.txt**' en ajoutant ceci :

```
add_executable(test_random_node src/test_random.cpp)
```

```
target_link_libraries(test_random_node ${catkin_LIBRARIES})
```


Lancer dans 2 terminaux différents, 'rviz avec le plugin MoveIt!' et ce nouveau 'node' :

```
roslaunch kuka_lwr_moveit demo.launch  
roslaunch my_first_move_group test_random_node
```

Observer le mouvement planifié du bras dans 'Rviz'.

Remarque :

- Décocher les options 'Query Start State' et 'Query Goal State' de 'Planning Request'.



▼ Planning Request  
Planning Group  
Show Workspace  
Query Start State  
Query Goal State  
Interactive Marker Size



manipulator

Créer un autre fichier C++ nommé '**test\_custom.cpp**' permettant de spécifier une pose précise à atteindre :

```
#include <moveit/move_group_interface/move_group.h>
```

```
int main(int argc, char **argv)  
{
```

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

```
    // start a ROS spinning thread  
    ros::NodeHandle node_handle;  
    ros::AsyncSpinner spinner(1);  
    spinner.start();
```

```
    // setup the group you would like to control and plan for.  
    moveit::planning_interface::MoveGroup group("manipulator");
```

```
    // retrieve current position and orientation  
    geometry_msgs::PoseStamped robot_pose;  
    robot_pose = group.getCurrentPose();
```

```
    geometry_msgs::Pose current_position;  
    current_position = robot_pose.pose;
```

```
    geometry_msgs::Point exact_pose = current_position.position;  
    geometry_msgs::Quaternion exact_orientation = current_position.orientation;
```

```
    ROS_INFO("Current position : x=%f, y=%f, z=%f", exact_pose.x, exact_pose.y, exact_pose.z);  
    ROS_INFO("Current orientation : x=%f, y=%f, z=%f, w=%f", exact_orientation.x, exact_orientation.y, exact_orientation.z,  
    exact_orientation.w);
```

```
    // We can print the name of the reference frame for this robot.  
    ROS_INFO("Reference frame: %s", group.getPlanningFrame().c_str());
```

```
    // We can also print the name of the end-effector link for this group.  
    ROS_INFO("Reference frame: %s", group.getEndEffectorLink().c_str());
```

```
    // Planning to a Pose goal  
    // =====
```

```
    // We can plan a motion for this group to a desired pose for the  
    // end-effector.
```

```
    geometry_msgs::Pose target_goal;  
    target_goal.orientation.w = 1.0;
```

```

target_goal.orientation.x = 0.0;
target_goal.orientation.y = 0.0;
target_goal.orientation.z = 0.0;

target_goal.position.x = 0.3;
target_goal.position.y = 0.3;
target_goal.position.z = 1.5;
group.setPoseTarget(target_goal);

// call the planner to compute the plan and visualize it.
moveit::planning_interface::MoveGroup::Plan my_plan;
bool success = group.plan(my_plan);

ROS_INFO("Visualizing plan 1 (pose goal) %s",success?"":"FAILED");

// then move the group to the sampled target
group.move();

// wait for shutdown
ros::waitForShutdown();
return 0;
}

```

Modifier le fichier '**CMakelists.txt**' afin de prendre en compte ce nouveau fichier.  
Compiler et tester.

Créer un fichier C++ nommé '**test\_collision.cpp**' permettant de lancer une planification avec un objet ajouté dans la scène (ici une boîte) :

```

#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "test_collision_node");

    // start a ROS spinning thread
    ros::NodeHandle node_handle;
    ros::AsyncSpinner spinner(1);
    spinner.start();

    // setup the group you would like to control and plan for.
    moveit::planning_interface::MoveGroup group("manipulator");

    // setup a plan
    moveit::planning_interface::MoveGroup::Plan my_plan;

    // We will use this class to deal directly with the world.
    moveit::planning_interface::PlanningSceneInterface planning_scene_interface;

    // retrieve current position and orientation.
    geometry_msgs::PoseStamped robot_pose;
    robot_pose = group.getCurrentPose();

    geometry_msgs::Pose current_position;
    current_position = robot_pose.pose;

    geometry_msgs::Point exact_pose = current_position.position;
    geometry_msgs::Quaternion exact_orientation = current_position.orientation;

```

```

ROS_INFO("Current position : x=%f, y=%f, z=%f", exact_pose.x, exact_pose.y, exact_pose.z);
ROS_INFO("Current orientation : x=%f, y=%f, z=%f, w=%f", exact_orientation.x, exact_orientation.y, exact_orientation.z,
exact_orientation.w);

// We can print the name of the reference frame for this robot.
ROS_INFO("Reference frame: %s", group.getPlanningFrame().c_str());

// We can also print the name of the end-effector link for this group.
ROS_INFO("Reference frame: %s", group.getEndEffectorLink().c_str());

// Plan and move the robot to Home Position.
ROS_INFO("Go to Home Position");

geometry_msgs::Pose target_home;
target_home.orientation.w = 1.0;
target_home.orientation.x = 0.0;
target_home.orientation.y = 0.0;
target_home.orientation.z = 0.0;

target_home.position.x = 0.0;
target_home.position.y = 0.0;
target_home.position.z = 2.178500;

group.setPoseTarget(target_home);
bool success = group.plan(my_plan);

ROS_INFO("Visualizing plan 1 (pose home) %s",success?"":"FAILED");
group.move();

// Adding Collision Object
// =====
// First, we will define the collision object message.
moveit_msgs::CollisionObject collision_object;
collision_object.header.frame_id = group.getPlanningFrame();

// The id of the object is used to identify it.
collision_object.id = "box1";

// Define a box to add to the world.
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.1;
primitive.dimensions[1] = 0.5;
primitive.dimensions[2] = 1.0;

// A pose for the box.
geometry_msgs::Pose box_pose;
box_pose.orientation.w = 1.0;
box_pose.position.x = -0.3;
box_pose.position.y = 0.0;
box_pose.position.z = 2.0;

// Now, let's add the collision object into the world
ROS_INFO("Add an object into the world");
collision_object.primitives.push_back(primitive);
collision_object.primitive_poses.push_back(box_pose);
collision_object.operation = collision_object.ADD;

std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(collision_object);

planning_scene_interface.addCollisionObjects(collision_objects);

// Sleep so we have time to see the object in RViz.
sleep(2.0);

```



```

// Planning with collision detection can be slow. Lets set the planning time
// to be sure the planner has enough time to plan around the box. 10 seconds
// should be plenty.
group.setPlanningTime(10.0);

// Planning to a Pose goal
// =====
// We can plan a motion for this group to a desired pose for the
// end-effector.
geometry_msgs::Pose target_goal;
target_goal.orientation.w = 1.0;
target_goal.orientation.x = 0.0;
target_goal.orientation.y = 0.0;
target_goal.orientation.z = 0.0;

target_goal.position.x = -0.7;
target_goal.position.y = 0.2;
target_goal.position.z = 1.6;
group.setPoseTarget(target_goal);

// Now, we call the planner to compute the plan
// and visualize it.
success = group.plan(my_plan);

ROS_INFO("Visualizing plan 2 (pose goal) %s", success?"":"FAILED");

// then move the group to the sampled target.
group.move();

// wait for shutdown.
ros::waitForShutdown();
return 0;
}

```

Ajouter ce fichier dans '**CMakeLists.txt**', compiler et tester.

Créer un fichier C++ nommé '**test\_attach.cpp**' permettant de lancer une planification du bras avec un objet attaché :

```

#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "test_attach_node");

    // start a ROS spinning thread
    ros::NodeHandle node_handle;
    ros::AsyncSpinner spinner(1);
    spinner.start();

    // We will use this class to deal directly with the world.
    moveit::planning_interface::PlanningSceneInterface planning_scene_interface;

    // setup the group you would like to control and plan for.
    moveit::planning_interface::MoveGroup group("manipulator");

    // setup a plan
    moveit::planning_interface::MoveGroup::Plan my_plan;

```

```

// retrieve current position and orientation.
geometry_msgs::PoseStamped robot_pose;
robot_pose = group.getCurrentPose();

geometry_msgs::Pose current_position;
current_position = robot_pose.pose;

geometry_msgs::Point exact_pose = current_position.position;
geometry_msgs::Quaternion exact_orientation = current_position.orientation;

ROS_INFO("Current position : x=%f, y=%f, z=%f", exact_pose.x, exact_pose.y, exact_pose.z);
ROS_INFO("Current orientation : x=%f, y=%f, z=%f, w=%f", exact_orientation.x, exact_orientation.y, exact_orientation.z,
exact_orientation.w);

// We can print the name of the reference frame for this robot.
ROS_INFO("Reference frame: %s", group.getPlanningFrame().c_str());

// We can also print the name of the end-effector link for this group.
ROS_INFO("Reference frame: %s", group.getEndEffectorLink().c_str());

// Plan and move the robot to Home Position.
geometry_msgs::Pose target_home;
target_home.orientation.w = 1.0;
target_home.orientation.x = 0.0;
target_home.orientation.y = 0.0;
target_home.orientation.z = 0.0;

target_home.position.x = 0.0;
target_home.position.y = 0.0;
target_home.position.z = 2.178500;

group.setPoseTarget(target_home);
bool success = group.plan(my_plan);

ROS_INFO("Visualizing plan 1 (pose home) %s",success?"":"FAILED");
group.move();

sleep(4.0);

// Adding/Removing Objects and Attaching/Detaching Objects
// =====
// First, we will define the collision object message.
moveit_msgs::CollisionObject collision_object;
collision_object.header.frame_id = group.getPlanningFrame();

// The id of the object is used to identify it.
collision_object.id = "box1";

// Define a box to add to the world.
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.4;
primitive.dimensions[1] = 0.1;
primitive.dimensions[2] = 0.4;

// A pose for the box.
geometry_msgs::Pose box_pose;
box_pose.orientation.w = 1.0;
box_pose.position.x = 0.6;
box_pose.position.y = -0.4;
box_pose.position.z = 1.2;

// Now, let's add the collision object into the world
ROS_INFO("Add an object into the world");
collision_object.primitives.push_back(primitive);

```

```

collision_object.primitive_poses.push_back(box_pose);
collision_object.operation = collision_object.ADD;

std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(collision_object);

planning_scene_interface.addCollisionObjects(collision_objects);

// Sleep so we have time to see the object in RViz.
sleep(4.0);

// Planning to a Pose goal
// =====
// We can plan a motion for this group to a desired pose for the
// end-effector.
geometry_msgs::Pose target_goal;
target_goal.orientation.w = 1.0;
target_goal.orientation.x = 0.0;
target_goal.orientation.y = 0.0;
target_goal.orientation.z = 0.0;

target_goal.position.x = 0.6;
target_goal.position.y = -0.27;
target_goal.position.z = 1.2;
group.setPoseTarget(target_goal);

// Now, we call the planner to compute the plan
// and visualize it.
success = group.plan(my_plan);

ROS_INFO("Visualizing plan 2 (pose goal) %s",success?"":"FAILED");
group.move();

// Get current position and orientation.
robot_pose = group.getCurrentPose();
current_position = robot_pose.pose;

exact_pose = current_position.position;
exact_orientation = current_position.orientation;

ROS_INFO("Current position : x=%f, y=%f, z=%f", exact_pose.x, exact_pose.y, exact_pose.z);
ROS_INFO("Current orientation : x=%f, y=%f, z=%f, w=%f", exact_orientation.x, exact_orientation.y, exact_orientation.z,
exact_orientation.w);

sleep(4.0);

// Now, let's attach the collision object to the robot.
ROS_INFO("Attach the object to the robot");
group.attachObject(collision_object.id);

// Sleep to give Rviz time to show the object attached (different color).
sleep(4.0);

// Plan, move the robot and object attached.
ROS_INFO("Move the robot and the object attached to the robot");
target_goal.position.x -= 0.2;
target_goal.position.y -= 0.2;
target_goal.position.z += 0.5;
group.setPoseTarget(target_goal);

success = group.plan(my_plan);
group.move();

sleep(4.0);

// Now, let's detach the collision object from the robot.
ROS_INFO("Detach the object from the robot");

```

```

group.detachObject(collision_object.id);

// Sleep to give Rviz time to show the object detached.
sleep(4.0);

// Now, let's remove the collision object from the world.
ROS_INFO("Remove the object from the world");
std::vector<std::string> object_ids;
object_ids.push_back(collision_object.id);
planning_scene_interface.removeCollisionObjects(object_ids);

// Sleep to give Rviz time to show the object is no longer there.
sleep(4.0);

// Plan, move the robot to Home Position.
ROS_INFO("Return to Home Position");
group.setPoseTarget(target_home);
success = group.plan(my_plan);

ROS_INFO("Visualizing plan 3 (pose home) %s",success?"":"FAILED");
group.move();

success = group.plan(my_plan);

// wait for shutdown
ros::waitForShutdown();
return 0;
}

```

Ajouter ce fichier dans '**CMakeLists.txt**', compiler et tester.

Créer un fichier C++ nommé '**test\_path.cpp**' permettant de lancer une planification du bras selon un chemin ('Cartesian Path') :

```

#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

#include <moveit/robot_trajectory/robot_trajectory.h>
#include <moveit/trajectory_processing/iterative_time_parameterization.h>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "test_path_node");

    // start a ROS spinning thread
    ros::NodeHandle node_handle;
    ros::AsyncSpinner spinner(1);
    spinner.start();

    // setup the group you would like to control and plan for.
    moveit::planning_interface::MoveGroup group("manipulator");

    // setup a plan
    moveit::planning_interface::MoveGroup::Plan my_plan;

    // plan and move the robot to Home Position.
    ROS_INFO("Go to Home Position");

    geometry_msgs::Pose target_home;
    target_home.orientation.w = 1.0;

```

```

target_home.orientation.x = 0.0;
target_home.orientation.y = 0.0;
target_home.orientation.z = 0.0;

target_home.position.x = 0.0;
target_home.position.y = 0.0;
target_home.position.z = 1.8;

group.setPoseTarget(target_home);
bool success = group.plan(my_plan);

ROS_INFO("Visualizing plan 1 (pose home) %s",success?"":"FAILED");
group.move();

// sleep so we have time to see in RViz.
sleep(4.0);

// retrieve current position and orientation.
geometry_msgs::PoseStamped robot_pose;
robot_pose = group.getCurrentPose();

geometry_msgs::Pose current_position;
current_position = robot_pose.pose;

geometry_msgs::Point exact_pose = current_position.position;
geometry_msgs::Quaternion exact_orientation = current_position.orientation;

ROS_INFO("Current position : x=%f, y=%f, z=%f", exact_pose.x, exact_pose.y, exact_pose.z);
ROS_INFO("Current orientation : x=%f, y=%f, z=%f, w=%f", exact_orientation.x, exact_orientation.y, exact_orientation.z,
exact_orientation.w);

// We can print the name of the reference frame for this robot.
ROS_INFO("Reference frame: %s", group.getPlanningFrame().c_str());

// We can also print the name of the end-effector link for this group.
ROS_INFO("Reference frame: %s", group.getEndEffectorLink().c_str());

// Sleep so we have time to see in RViz.
sleep(4.0);

ROS_INFO("begin cartesian path ...");

// Cartesian Paths
// =====
// You can plan a cartesian path directly by specifying a list of waypoints
// for the end-effector to go through.
std::vector<geometry_msgs::Pose> waypoints;

geometry_msgs::Pose target_pose = current_position;

// move x by 1 cm.
for (int i=0; i<10; i++) {
    target_pose.position.x += 0.01;
    waypoints.push_back(target_pose);
}

// move y by 1 cm.
for (int i=0; i<10; i++) {
    target_pose.position.y += 0.01;
    waypoints.push_back(target_pose);
}

// move z by 1 cm.
for (int i=0; i<10; i++) {
    target_pose.position.z += 0.01;
    waypoints.push_back(target_pose);
}

```

```

ROS_INFO("final target_pose x = %f, y = %f, z = %f", target_pose.position.x, target_pose.position.y, target_pose.position.z);

// We want the cartesian path to be interpolated at a resolution of 1 cm
// which is why we will specify 0.01 as the max step in cartesian
// translation. We will specify the jump threshold as 0.0, effectively
// disabling it.
moveit_msgs::RobotTrajectory trajectory_msg;

// set Planning time to 30 seconds.
group.setPlanningTime(30.0);

double fraction = group.computeCartesianPath(waypoints,
                                              0.01, // step of 1 cm
                                              0.0, // jump_threshold
                                              trajectory_msg);

ROS_INFO("Visualizing plan 2 (cartesian path) (%.2f%% acheived)",
        fraction * 100.0);

// Sleep so we have time to see the object in RViz.
sleep(4.0);

ros::waitForShutdown();
return 0;
}

```

Ajouter ce fichier dans '**CMakeLists.txt**', compiler et tester.