Project-1: Robotic Manipulator

Generated by Doxygen 1.8.17

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 CollisionBox Struct Reference

Struct containing the information of an object that we want to check collisions with.

`#include <collision_handler.h>`

Public Attributes

- Eigen::Matrix3d rotation_matrix

  rotation matrix we use to translate a point to the CollisionBox frame
- Eigen::Vector3d translation

  translation vector we use to translate a point to the CollisionBox frame
- Eigen::Vector3d extents

  vector containing length, width and height of the CollisionBox
- std::string box_name

  Collsion Box name.

3.1.1 Detailed Description

Struct containing the information of an object that we want to check collisions with.

The documentation for this struct was generated from the following file:

- collision_handler.h

3.2 image_processor.imageProcessor Class Reference

Public Member Functions

- def __init__ (self)
- def receive_pointcloud (self, msg)
- def receive_image_L (self, msg)
- def get_pointCloud_region (self, x1, y1, x2, y2)
- def object_detection (self, path_to_model, path_to_weights)
- def show_detections (self)
- def pose_estimation (self, models_dir)
- def show_objects_poses (self)

Public Attributes

- img_L
  image captured from the camera left eye
- **point_cloud**
- **classes**
- **boxes**
- **confidences**
- **class_ids**
- **number_of_detections**
- objects_relative_points
  Object points respect to object's center.
- objects_relative_pixels
  Object points relative to pixel coordinate system.
- objects_rotations
  list of rotation matrices from World to Object coordinates
- objects_positions
  list of traslation vectors from World to Object coordinates
- w_R_c
  Rotation matrix from Camera to World frame.
- x_c
  Traslation vector from Robot to Camera frame.
- base_offset
  Translation vector from World to Robot base frame.
- cameraMatrix
  intrisic parameters of the camera
- distCoeffs
  intrinsic distortion coefficients of the camera

3.2.1 Detailed Description

```
Class to detect and classify objects using YOLO5.
```

3.2.2 Constructor & Destructor Documentation

3.2.2.1 __init__()

```
def image_processor.imageProcessor.__init__ (
              self )
```

```
Initialize the ImageProcessor.
```

3.2.3 Member Function Documentation

3.2.3.1 get_pointCloud_region()

```
def image_processor.imageProcessor.get_pointCloud_region (
              self,
              x1,
              y1,
              x2,
              y2 )
```

```
Retrieve the point cloud region.

Args:
    x1 (int): x-coordinate of the top-left corner.
    y1 (int): y-coordinate of the top-left corner.
    x2 (int): x-coordinate of the bottom-right corner.
    y2 (int): y-coordinate of the bottom-right corner.

Returns:
    tuple: points regarding the specified region in camera coordinates, same points in pixel coordinates.
```

3.2.3.2 object_detection()

```
def image_processor.imageProcessor.object_detection (
              self,
              path_to_model,
              path_to_weights )
```

```
Detect objects in the image and store the results in class variables:

boxes : bounding boxes of the objects.
confidences : confidence score of each detection.
class_ids : classes associated yolo indexes.
classes : class names in string format.
number_of_detections : number of detected objects.

Args:
    path_to_model (string): absolute path to the Yolov5 model.
    path_to_weights (string) : absolute path to the weights file.
```

3.2.3.3 pose_estimation()

```
def image_processor.imageProcessor.pose_estimation (
              self,
              models_dir )
```

```
Instantiate a PoseEstimater object for each detection and run pose estimation on it

Args:
    models_dir (string) : Absolute path to .stl mesh folder
```

3.2.3.4 receive_image_L()

```
def image_processor.imageProcessor.receive_image_L (
              self,
              msg )
```

```
Receive the left camera image.

Args:
    msg (Image): Image message.
```

3.2.3.5 receive_pointcloud()

```
def image_processor.imageProcessor.receive_pointcloud (
              self,
              msg )
```

```
Receive and store the point cloud data.

Args:
    msg (PointCloud2): Point cloud message.
```

3.2.3.6 show_detections()

```
def image_processor.imageProcessor.show_detections (
              self )
```

```
Show the image with detections on using openCV libraries
```

3.2.3.7 show_objects_poses()

```
def image_processor.imageProcessor.show_objects_poses (
              self )
```

```
Display the detected rotation on the objects as cartesian axes with openCV
```

The documentation for this class was generated from the following file:

- image_processor.py

3.3 Pos_manager Class Reference

pos_manager is a class that we use to publish joint angles for the real/simulated robot

```
#include <pos_manager.h>
```

Public Member Functions

- Pos_manager (ros::NodeHandle node)

  Constructor for a new Pos_manager object, differentiate between real and simulated robot, and soft/rigid gripper.
- void send_Reference (Eigen::VectorXd q_des, double diameter=0, Eigen::VectorXd qd_des=Eigen::VectorXd(), Eigen::VectorXd tau_ffwd=Eigen::VectorXd())

  Function used to publish robot instructions, calls send_full_joint_state() or send_reduced_joint_state(), depending on how much information we can publish.

Private Member Functions

- void initFilter (const int &size)

  Function that initializes a filter (used fon gripper fingers in simulation)
- Eigen::VectorXd secondOrderFilter (Eigen::VectorXd input, const double rate, const double settling_time)

  Function that applies the filter on a given input (gripper joint in simulation)
- void send_full_joint_state (Eigen::VectorXd q_des, double diameter=0, Eigen::VectorXd qd_des=Eigen::VectorXd(), Eigen::VectorXd tau_ffwd=Eigen::VectorXd())

  Function used to publish when using "torque" control mode in simulation.
- void send_reduced_joint_state (Eigen::VectorXd q_des, double diameter=0)

  Function used to only publish joint and gripper positions.

Private Attributes

- ros::NodeHandle pos_manager_node

  node we use to instantiate the publishers
- bool real_robot

  boolean representing real or simulated robot (true = REAL ROBOT, false = SIMULATION)
- bool gripper_sim

  boolean to see if we want to utilize the gripper during simulation
- bool soft_gripper

  boolean differentiating between soft and rigid gripper (true = SOFT GRIPPER, false = RIGID GRIPPER)
- int number_of_fingers

  number of fingers decided from the soft_gripper variable
- int number_of_joints

  number of joints of the ur5 robot exluding the gripper fingers (in our case its always 6)
- std::string control_type

  string representing the control type ("position" or "torque", we only use position)
- sensor_msgs::JointState jointState_msg_sim

  message we publish in the send_full_joint_state() function
- std_msgs::Float64MultiArray jointState_msg_robot

  message we publish in the send_reduced_joint_state() function
- ros::Publisher pub_des_jstate

  publisher for joint states (subscribes to a different topic, depending on real_robot value)

- ros::ServiceClient gripper_client

  service client to make a service call to move_gripper() when using the real robot
- ros_impedance_controller::generic_float srv

  message we use for the rosservice call to move_gripper()
- Eigen::VectorXd filter_1

  first filter used on secondOrderFilter()
- Eigen::VectorXd filter_2

  second filter used on secondOrderFilter()

3.3.1 Detailed Description

pos_manager is a class that we use to publish joint angles for the real/simulated robot

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Pos_manager()

```
Pos_manager::Pos_manager (
            ros::NodeHandle node )
```

Constructor for a new Pos_manager object, differentiate between real and simulated robot, and soft/rigid gripper.

Parameters

node	ros node we want this object to have reference to

3.3.3 Member Function Documentation

3.3.3.1 initFilter()

```
void Pos_manager::initFilter (
            const int & size )  [private]
```

Function that initializes a filter (used fon gripper fingers in simulation)

Parameters

size	size of the filter

3.3.3.2 secondOrderFilter()

```
Eigen::VectorXd Pos_manager::secondOrderFilter (
            Eigen::VectorXd input,
            const double rate,
            const double settling_time )  [private]
```

Function that applies the filter on a given input (gripper joint in simulation)

Parameters

input	vector we want to filter
rate	frequency at which the filter is being applied
settling_time	settling time of the filter

Returns

Eigen::VectorXd filtered output obtained from the input vector

3.3.3.3 send_full_joint_state()

```
void Pos_manager::send_full_joint_state (
            Eigen::VectorXd q_des,
            double diameter = 0,
            Eigen::VectorXd qd_des = Eigen::VectorXd(),
            Eigen::VectorXd tau_ffwd = Eigen::VectorXd() )  [private]
```

Function used to publish when using "torque" control mode in simulation.

Parameters

q_des	joint angle we want to publish
diameter	diameter for opening or closing the gripper
qd_des	joint velocity we want to publish
tau_ffwd	joint effort we want to publish

3.3.3.4 send_reduced_joint_state()

```
void Pos_manager::send_reduced_joint_state (
            Eigen::VectorXd q_des,
            double diameter = 0 )  [private]
```

Function used to only publish joint and gripper positions.

Parameters

q_des	joint angle we want to publish
diameter	diameter for opening or closing the gripper

3.3.3.5 send_Reference()

```
void Pos_manager::send_Reference (
            Eigen::VectorXd q_des,
            double diameter = 0,
            Eigen::VectorXd qd_des = Eigen::VectorXd(),
            Eigen::VectorXd tau_ffwd = Eigen::VectorXd() )
```

Function used to publish robot instructions, calls send_full_joint_state() or send_reduced_joint_state(), depending on how much information we can publish.

Parameters

q_des	joint angle we want to publish
diameter	diameter for opening or closing the gripper
qd_des	joint velocity we want to publish
tau_ffwd	joint effort we want to publish

The documentation for this class was generated from the following files:

- pos_manager.h
- pos_manager.cpp

3.4 image_processor.PoseEstimator Class Reference

Public Member Functions

- def __init__ (self, object_points, object_label, path_to_models)
- def draw_registration_result (self, source, target, transformation)
- def preprocess_point_cloud (self, pcd, voxel_size)
- def prepare_data (self, voxel_size, object_points, path_to_models, object_label)
- def execute_global_registration (self, source_down, target_down, source_fpfh, target_fpfh, voxel_size)
- def refine_registration (self, source, target, result_ransac, voxel_size)
- def run_pose_estimation (self)

Public Attributes

- object_points
  detected object points from the depth sensor
- object_label
  object name
- path_to_models
  absolute path to the mesh folder
- do_draw
  Set True to see intermediate point cloud plots.
- voxel_size
  scaling factor to tune functions with
- **target_fpfh**

3.4.1 Detailed Description

```
Class to estimate the rotation of the objects.
```

3.4.2 Constructor & Destructor Documentation

3.4.2.1 __init__()

```
def image_processor.PoseEstimator.__init__ (
              self,
              object_points,
              object_label,
              path_to_models )
```

```
Initialize the PoseEstimator.

Args:
    object_points (list): List of object points.
    object_label (str): Label of the object.
    path_to_models (str): Path to the .stl meshes.
```

3.4.3 Member Function Documentation

3.4.3.1 draw_registration_result()

```
def image_processor.PoseEstimator.draw_registration_result (
              self,
              source,
              target,
              transformation )
```

```
Draw the registration result.

Args:
    source (PointCloud): Source point cloud (detected one).
    target (PointCloud): Target point cloud (artificial extracted from the mesh).
    transformation (ndarray): Transformation matrix between the 2 point clouds.
```

3.4.3.2 execute_global_registration()

```
def image_processor.PoseEstimator.execute_global_registration (
              self,
              source_down,
              target_down,
              source_fpfh,
              target_fpfh,
              voxel_size )
```

```
Execute global registration using RANSAC.

Args:
    source_down (PointCloud): Downsampled source point cloud.
    target_down (PointCloud): Downsampled target point cloud.
    source_fpfh (ndarray): Source FPFH features.
    target_fpfh (ndarray): Target FPFH features.
    voxel_size (float): Voxel size.

Returns:
    RegistrationResult: Result of the global registration.
```

3.4.3.3 prepare_data()

```
def image_processor.PoseEstimator.prepare_data (
              self,
              voxel_size,
              object_points,
              path_to_models,
              object_label )
```

```
Generate the 2 point clouds as open3d friendly pointclouds and preprocess them by:
downsampling, estimate normals, extract FPFH features.

Args:
    voxel_size (float): Voxel size to scale hyper parameters.
    object_points (list): List of object points.
    path_to_models (str): Path to the .stl mesh.
    object_label (str): Label of the object.

Returns:
    tuple: Tuple containing the source, target, downsampled source and target, and FPFH features.
```

3.4.3.4 preprocess_point_cloud()

```
def image_processor.PoseEstimator.preprocess_point_cloud (
              self,
              pcd,
              voxel_size )
```

```
Preprocess the point cloud.

Args:
    pcd (PointCloud): Point cloud.
    voxel_size (float): Voxel size to scale hyper parameters.

Returns:
    tuple: Tuple containing the downsampled point cloud and FPFH features.
```

3.4.3.5 refine_registration()

```
def image_processor.PoseEstimator.refine_registration (
              self,
              source,
              target,
              result_ransac,
              voxel_size )
```

```
Refine the registration using ICP.

Args:
    source (PointCloud): Source point cloud.
    target (PointCloud): Target point cloud.
    result_ransac (RegistrationResult): Result of the global registration.
    voxel_size (float): Voxel size.

Returns:
    RegistrationResult: refined registration result.
```

3.4.3.6 run_pose_estimation()

```
def image_processor.PoseEstimator.run_pose_estimation (
              self )
```

```
Run the pose estimation algorithm.

Returns:
    ndarray: Rotation matrix of the object.
```

The documentation for this class was generated from the following file:

- image_processor.py

Chapter 4

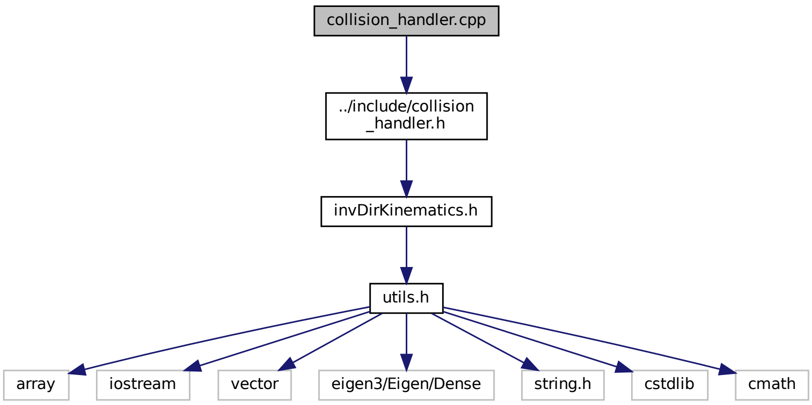
File Documentation

4.1 collision_handler.cpp File Reference

Implementation of the functions used for collision control.

`#include "../include/collision_handler.h"`

Include dependency graph for collision_handler.cpp:



Functions

- void get_joint_info (const int joint_index, const VectorXd &config, Vector3d &joint_pos, Matrix3d &joint_rot, Vector3d &joint_length)

    Get specific joint information (joint_pos, joint_rot, joint_length), knowing the current configuration.
- MatrixXd get_square_matrix_plane (int dimension, float w)

    Function used to get a set of 4 points (vertices of a square) in the same plane (around the robot arm)
- std::vector< Vector3d > get_joint_collision_points (const double square_spacing, const double box_extents, const Vector3d &joint_pos, const Matrix3d &joint_rot, const Vector3d &joint_length)

    Get the points that we use to evaluate collisions for a specific joint.
- bool **violates_joint_limits** (const VectorXd &conf)
- std::vector< CollisionBox > define_world_cBoxes ()

Function used to create the "fixed" collision boxes (Table, ...) that will not change during the robot motion.
The collision box for the arm is evaluated for every configuration we check collisions on.

- bool isPointInsideBox (const Eigen::Vector3d &point, const CollisionBox &box)

  Function that checks if a given point is inside of a CollisionBox object.

- bool checkCollisions (const MatrixXd &joint_configs, int conf_index)

  Function that checks for collisions with the world or the robot itself (by building CollisionBox objects around the arm's links).
  This control is done every 10 configurations.

4.1.1 Detailed Description

Implementation of the functions used for collision control.

4.1.2 Function Documentation

4.1.2.1 checkCollisions()

```
bool checkCollisions (
            const MatrixXd & joint_configs,
            int conf_index )
```

Function that checks for collisions with the world or the robot itself (by building CollisionBox objects around the arm's links).
This control is done every 10 configurations.

Parameters

joint_configs	configurations of the joints during a computed trajectory
conf_index	index of the final configuration we are evaluating

Returns

true if the trajectory doesn't have any collisions

4.1.2.2 define_world_cBoxes()

```
std::vector<CollisionBox> define_world_cBoxes ( )
```

Function used to create the "fixed" collision boxes (Table, ...) that will not change during the robot motion.
The collision box for the arm is evaluated for every configuration we check collisions on.

Returns

std::vector<CollisionBox> List of CollisionBox objects

4.1.2.3 get_joint_collision_points()

```
std::vector<Vector3d> get_joint_collision_points (
            const double square_spacing,
            const double box_extents,
            const Vector3d & joint_pos,
            const Matrix3d & joint_rot,
            const Vector3d & joint_length )
```

Get the points that we use to evaluate collisions for a specific joint.

Parameters

square_spacing	spacing between each set of 4 points along the robot's link
box_extents	distance (divided by 2) that the 4 points have from eachother
joint_pos	position of the joint
joint_rot	rotation of the joint
joint_length	length of the joint along the 3 dimensions (X, Y, Z)

Returns

std::vector<Vector3d> vector containing a set of points (built around the arm) that we'll use to check for collisions

4.1.2.4 get_joint_info()

```
void get_joint_info (
            const int joint_index,
            const VectorXd & config,
            Vector3d & joint_pos,
            Matrix3d & joint_rot,
            Vector3d & joint_length )
```

Get specific joint information (joint_pos, joint_rot, joint_length), knowing the current configuration.

Parameters

joint_index	index of the joint we want informations about
config	current configuration of the joints
joint_pos	position of the joint with regards to the robot base frame, obtained with the same method as Direct kinematics
joint_rot	rotation of the joint with regards to the robot base frame, obtained with the same method as Direct kinematics
joint_length	length of the joint along the three axis (X, Y, Z)

4.1.2.5 get_square_matrix_plane()

```
MatrixXd get_square_matrix_plane (
            int dimension,
            float w )
```

Function used to get a set of 4 points (vertices of a square) in the same plane (around the robot arm)

Parameters

dimension	axis that we want our square to be perpendicular to, so that our square is built around the robot's link
w	length representing half of the square's side

Returns

MatrixXd matrix containing the 4 vertices of the square

4.1.2.6 isPointInsideBox()

```
bool isPointInsideBox (
            const Eigen::Vector3d & point,
            const CollisionBox & box )
```

Function that checks if a given point is inside of a CollisionBox object.

Parameters

point	point we'll use to check for collisions
box	object we want to know collisions on

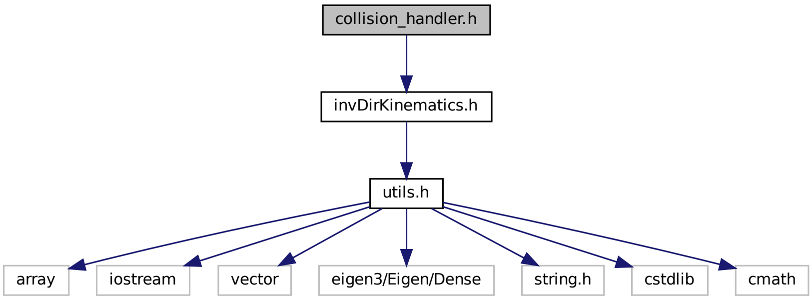
Returns

true if the point is inside the CollisionBox (we have a collision)

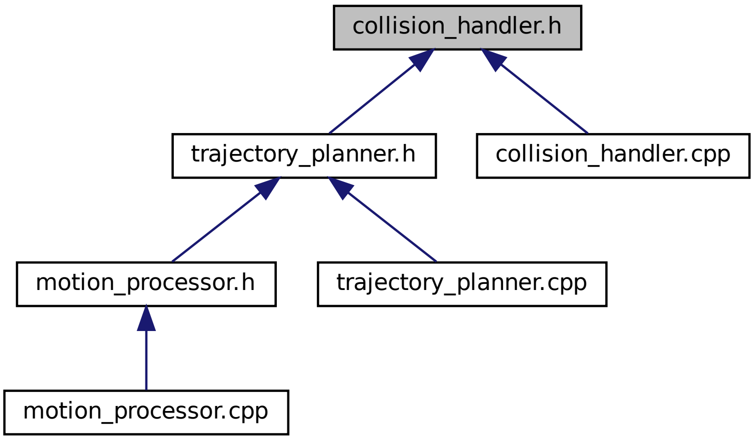
4.2 collision_handler.h File Reference

Header with the declaration of the functions implemented in collision_handler.cpp.

`#include "invDirKinematics.h"`
Include dependency graph for collision_handler.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct CollisionBox

  Struct containing the information of an object that we want to check collisions with.

Functions

- bool isPointInsideBox (const Eigen::Vector3d &point, const CollisionBox &box)

  Function that checks if a given point is inside of a CollisionBox object.
- std::vector< CollisionBox > define_world_cBoxes ()

  Function used to create the "fixed" collision boxes (Table, ...) that will not change during the robot motion. The collision box for the arm is evaluated for every configuration we check collisions on.

- void get_joint_info (const int joint_index, const VectorXd &config, Vector3d &joint_pos, Matrix3d &joint_rot, Vector3d &joint_length)

    Get specific joint information (joint_pos, joint_rot, joint_length), knowing the current configuration.
- MatrixXd get_square_matrix_plane (int dimension, float w)

    Function used to get a set of 4 points (vertices of a square) in the same plane (around the robot arm)
- std::vector< Vector3d > get_joint_collision_points (const double square_spacing, const double box_extents, const Vector3d &joint_pos, const Matrix3d &joint_rot, const Vector3d &joint_length)

    Get the points that we use to evaluate collisions for a specific joint.
- bool checkCollisions (const MatrixXd &joint_configs, int conf_index)

    Function that checks for collisions with the world or the robot itself (by building CollisionBox objects around the arm's links).
    This control is done every 10 configurations.

4.2.1 Detailed Description

Header with the declaration of the functions implemented in collision_handler.cpp.

4.2.2 Function Documentation

4.2.2.1 checkCollisions()

```
bool checkCollisions (
            const MatrixXd & joint_configs,
            int conf_index )
```

Function that checks for collisions with the world or the robot itself (by building CollisionBox objects around the arm's links).
This control is done every 10 configurations.

Parameters

joint_configs	configurations of the joints during a computed trajectory
conf_index	index of the final configuration we are evaluating

Returns

true if the trajectory doesn't have any collisions

4.2.2.2 define_world_cBoxes()

```
std::vector<CollisionBox> define_world_cBoxes ( )
```

Function used to create the "fixed" collision boxes (Table, ...) that will not change during the robot motion.
The collision box for the arm is evaluated for every configuration we check collisions on.

Returns

std::vector<CollisionBox> List of CollisionBox objects

4.2.2.3 get_joint_collision_points()

```
std::vector<Vector3d> get_joint_collision_points (
            const double square_spacing,
            const double box_extents,
            const Vector3d & joint_pos,
            const Matrix3d & joint_rot,
            const Vector3d & joint_length )
```

Get the points that we use to evaluate collisions for a specific joint.

Parameters

square_spacing	spacing between each set of 4 points along the robot's link
box_extents	distance (divided by 2) that the 4 points have from eachother
joint_pos	position of the joint
joint_rot	rotation of the joint
joint_length	length of the joint along the 3 dimensions (X, Y, Z)

Returns

std::vector<Vector3d> vector containing a set of points (built around the arm) that we'll use to check for collisions

4.2.2.4 get_joint_info()

```
void get_joint_info (
            const int joint_index,
            const VectorXd & config,
            Vector3d & joint_pos,
            Matrix3d & joint_rot,
            Vector3d & joint_length )
```

Get specific joint information (joint_pos, joint_rot, joint_length), knowing the current configuration.

Parameters

joint_index	index of the joint we want informations about
config	current configuration of the joints
joint_pos	position of the joint with regards to the robot base frame, obtained with the same method as Direct kinematics
joint_rot	rotation of the joint with regards to the robot base frame, obtained with the same method as Direct kinematics
joint_length	length of the joint along the three axis (X, Y, Z)

4.2.2.5 get_square_matrix_plane()

```
MatrixXd get_square_matrix_plane (
            int dimension,
            float w )
```

Function used to get a set of 4 points (vertices of a square) in the same plane (around the robot arm)

Parameters

dimension	axis that we want our square to be perpendicular to, so that our square is built around the robot's link
w	length representing half of the square's side

Returns

MatrixXd matrix containing the 4 vertices of the square

4.2.2.6 isPointInsideBox()

```
bool isPointInsideBox (
            const Eigen::Vector3d & point,
            const CollisionBox & box )
```

Function that checks if a given point is inside of a CollisionBox object.

Parameters

point	point we'll use to check for collisions
box	object we want to know collisions on

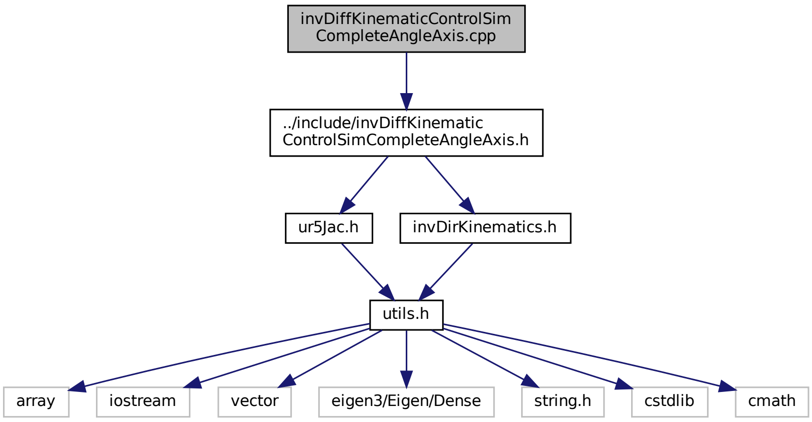
Returns

true if the point is inside the CollisionBox (we have a collision)

4.3 invDiffKinematicControlSimCompleteAngleAxis.cpp File Reference

Implementation of the functions we use to compute the Differential kinematic.

```
#include "../include/invDiffKinematicControlSimCompleteAngleAxis.h"
```
Include dependency graph for invDiffKinematicControlSimCompleteAngleAxis.cpp:



Functions

- Vector3d computeOrientationErrorW (Matrix3d w_R_e, Matrix3d w_R_d)

  Function that computes the orientation error of the end effector, using Angle axis instead of Euler angles.
- Matrix3d get_optimal_Kphi (const VectorXd &start_cfg, const VectorXd &end_cfg, const float base_factor)

  Compute Kphi used in the invDiffKinematicControlCompleteAngleAxis() function.
- VectorXd invDiffKinematicControlCompleteAngleAxis (const VectorXd &q, const Vector3d &xe, const Vector3d &xd, const Vector3d &vd, const Matrix3d &w_R_e, const Vector3d &phid, const Vector3d &phiddot, const Matrix3d &Kp, const Matrix3d &Kphi)

  Computes the joint velocities we to use for the Differential kinematics.
  This is called by invDiffKinematicControlSimCompleteAngleAxis() for every configuration in a trajectory.
- std::tuple< MatrixXd, MatrixXd, MatrixXd > invDiffKinematicControlSimCompleteAngleAxis (const MatrixXd &xd, const MatrixXd &phid, const VectorXd &TH0, const VectorXd &THf, const double minT, const double maxT, const double Dt)

  Computes the differential kinematic over a received trajectory.

4.3.1 Detailed Description

Implementation of the functions we use to compute the Differential kinematic.

4.3.2 Function Documentation

4.3.2.1 computeOrientationErrorW()

```
Vector3d computeOrientationErrorW (
            Matrix3d w_R_e,
            Matrix3d w_R_d )
```

Function that computes the orientation error of the end effector, using Angle axis instead of Euler angles.

Parameters

w_R←_e	current end effector rotation (rotation matrix)
w_R←_d	desired end effector rotation (rotation matrix)

Returns

Vector3d orientation error between current and desired end effector rotation

4.3.2.2 get_optimal_Kphi()

```
Matrix3d get_optimal_Kphi (
            const VectorXd & start_cfg,
            const VectorXd & end_cfg,
            const float base_factor )
```

Compute Kphi used in the invDiffKinematicControlCompleteAngleAxis() function.

Parameters

start_cfg	start joint configuration
end_cfg	finish joint configuration
base_factor	base factor added to the computed Kphi (to make sure its never 0)

Returns

Matrix3d matrix containing the values for Kphi we are going to use on the orientation error

4.3.2.3 invDiffKinematicControlCompleteAngleAxis()

```
VectorXd invDiffKinematicControlCompleteAngleAxis (
            const VectorXd & q,
            const Vector3d & xe,
            const Vector3d & xd,
            const Vector3d & vd,
            const Matrix3d & w_R_e,
            const Vector3d & phid,
            const Vector3d & phiddot,
            const Matrix3d & Kp,
            const Matrix3d & Kphi )
```

Computes the joint velocities we to use for the Differential kinematics.
This is called by invDiffKinematicControlSimCompleteAngleAxis() for every configuration in a trajectory.

Parameters

q	
xe	current end effector position
xd	desired end effector position
vd	desired end effector velocity
w_R↩ _e	current end effector rotation (rotation matrix)
phid	desired end effector rotation (euler angles)
phiddot	desired end effector angular velocity
Kp	scaling factor for the position error
Kphi	scaling factor for the orientation error obtained from get_optimal_Kphi()

Returns

VectorXd joint velocities corrected using the position and orientation error

4.3.2.4 invDiffKinematicControlSimCompleteAngleAxis()

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> invDiffKinematicControlSimCompleteAngleAxis (
            const MatrixXd & xd,
            const MatrixXd & phid,
            const VectorXd & TH0,
            const VectorXd & THf,
            const double minT,
            const double maxT,
            const double Dt )
```

Computes the differential kinematic over a received trajectory.

Parameters

xd	matrix containing end effector positions at each time step of the trajectory
phid	matrix containing end effector rotations at each time step of the trajectory
TH0	joint configuration at the start of the trajectory
THf	joint configuration at the end of the trajectory
minT	time of the start of the trajectory
maxT	time of the end of the trajectory
Dt	sampling time

Returns

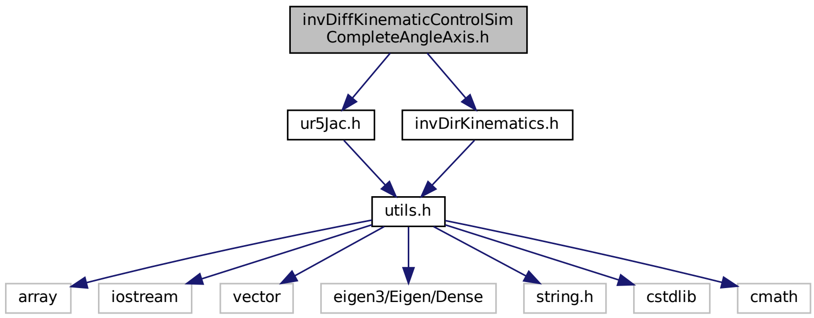
std::tuple<MatrixXd, MatrixXd, MatrixXd> corrected trajectory (joint positions, end effector positions, end effector rotations)

4.4 invDiffKinematicControlSimCompleteAngleAxis.h File Reference

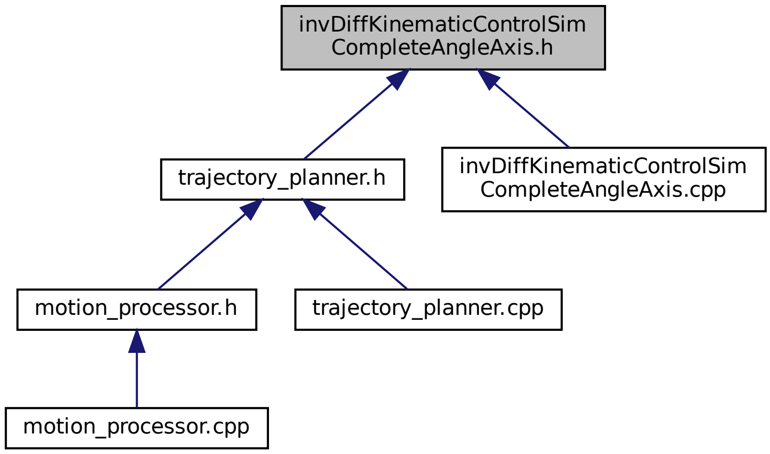
Header file containing the declaration of the functions we use to compute the Differential kinematic.

```
#include "ur5Jac.h"
#include "invDirKinematics.h"
```

Include dependency graph for invDiffKinematicControlSimCompleteAngleAxis.h:



This graph shows which files directly or indirectly include this file:



Functions

- std::tuple< MatrixXd, MatrixXd, MatrixXd > invDiffKinematicControlSimCompleteAngleAxis (const MatrixXd &xd, const MatrixXd &phid, const VectorXd &TH0, const VectorXd &THf, const double minT, const double maxT, const double Dt)

  Computes the differential kinematic over a received trajectory.
- VectorXd invDiffKinematicControlCompleteAngleAxis (const VectorXd &q, const Vector3d &xe, const Vector3d &xd, const Vector3d &vd, const Matrix3d &w_R_e, const Vector3d &phid, const Vector3d &phiddot, const Matrix3d &Kp, const Matrix3d &Kphi)

  Computes the joint velocities we to use for the Differential kinematics.
  This is called by invDiffKinematicControlSimCompleteAngleAxis() for every configuration in a trajectory.
- Matrix3d get_optimal_Kphi (const VectorXd &start_cfg, const VectorXd &end_cfg, const float base_factor)

  Compute Kphi used in the invDiffKinematicControlCompleteAngleAxis() function.
- Vector3d computeOrientationErrorW (Matrix3d w_R_e, Matrix3d w_R_d)

  Function that computes the orientation error of the end effector, using Angle axis instead of Euler angles.

4.4.1 Detailed Description

Header file containing the declaration of the functions we use to compute the Differential kinematic.

4.4.2 Function Documentation

4.4.2.1 computeOrientationErrorW()

```
Vector3d computeOrientationErrorW (
            Matrix3d w_R_e,
            Matrix3d w_R_d )
```

Function that computes the orientation error of the end effector, using Angle axis instead of Euler angles.

Parameters

w_R_e	current end effector rotation (rotation matrix)
w_R_d	desired end effector rotation (rotation matrix)

Returns

Vector3d orientation error between current and desired end effector rotation

4.4.2.2 get_optimal_Kphi()

```
Matrix3d get_optimal_Kphi (
            const VectorXd & start_cfg,
            const VectorXd & end_cfg,
            const float base_factor )
```

Compute Kphi used in the invDiffKinematicControlCompleteAngleAxis() function.

Parameters

start_cfg	start joint configuration
end_cfg	finish joint configuration
base_factor	base factor added to the computed Kphi (to make sure its never 0)

Returns

Matrix3d matrix containing the values for Kphi we are going to use on the orientation error

4.4.2.3 invDiffKinematicControlCompleteAngleAxis()

```
VectorXd invDiffKinematicControlCompleteAngleAxis (
            const VectorXd & q,
            const Vector3d & xe,
            const Vector3d & xd,
            const Vector3d & vd,
            const Matrix3d & w_R_e,
            const Vector3d & phid,
            const Vector3d & phiddot,
            const Matrix3d & Kp,
            const Matrix3d & Kphi )
```

Computes the joint velocities we to use for the Differential kinematics.
This is called by invDiffKinematicControlSimCompleteAngleAxis() for every configuration in a trajectory.

Parameters

q	
xe	current end effector position
xd	desired end effector position
vd	desired end effector velocity
w_R_e	current end effector rotation (rotation matrix)
phid	desired end effector rotation (euler angles)
phiddot	desired end effector angular velocity
Kp	scaling factor for the position error
Kphi	scaling factor for the orientation error obtained from get_optimal_Kphi()

Returns

VectorXd joint velocities corrected using the position and orientation error

4.4.2.4 invDiffKinematicControlSimCompleteAngleAxis()

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> invDiffKinematicControlSimCompleteAngleAxis (
            const MatrixXd & xd,
            const MatrixXd & phid,
            const VectorXd & TH0,
            const VectorXd & THf,
            const double minT,
            const double maxT,
            const double Dt )
```

Computes the differential kinematic over a received trajectory.

Parameters

xd	matrix containing end effector positions at each time step of the trajectory
phid	matrix containing end effector rotations at each time step of the trajectory

Parameters

TH0	joint configuration at the start of the trajectory
THf	joint configuration at the end of the trajectory
minT	time of the start of the trajectory
maxT	time of the end of the trajectory
Dt	sampling time

Returns

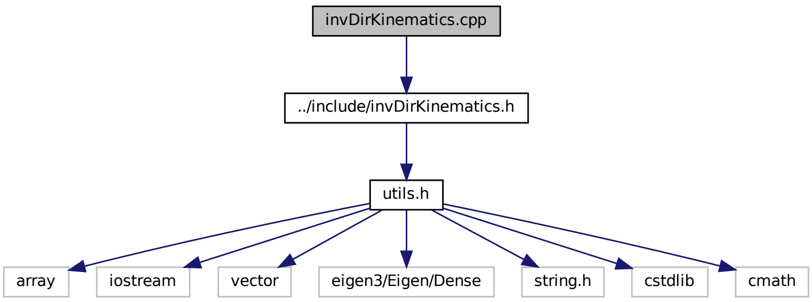
std::tuple<MatrixXd, MatrixXd, MatrixXd> corrected trajectory (joint positions, end effector positions, end effector rotations)

4.5 invDirKinematics.cpp File Reference

Implementation of the functions we use to compute the Direct and Inverse kinematics.

`#include "../include/invDirKinematics.h"`

Include dependency graph for invDirKinematics.cpp:



Functions

- VectorXd **A** (6)
- VectorXd **D** (6)
- Matrix4d T10 (double th1)

    Returns the transformation matrix from base frame to joint 1, given the angle we are using.
- Matrix4d T21 (double th2)

    Returns the transformation matrix from joint 1 to joint 2, given the angle we are using.
- Matrix4d T32 (double th3)

    Returns the transformation matrix from joint 2 to joint 3, given the angle we are using.
- Matrix4d T43 (double th4)

    Returns the transformation matrix from joint 3 to joint 4, given the angle we are using.
- Matrix4d T54 (double th5)

    Returns the transformation matrix from joint 4 to joint 5, given the angle we are using.
- Matrix4d T65 (double th6)

    Returns the transformation matrix from joint 5 to the last joint, given the angle we are using.
- std::tuple< Vector3d, Matrix3d > ur5Direct (VectorXd Th)

    Implementation of Direct kinematic for the ur5.
- MatrixXd ur5Inverse (Vector3d p60, Matrix3d R60)

    Implementation of Inverse kinematic for the ur5.

4.5.1 Detailed Description

Implementation of the functions we use to compute the Direct and Inverse kinematics.

4.5.2 Function Documentation

4.5.2.1 T10()

```
Matrix4d T10 (
            double th1 )
```

Returns the transformation matrix from base frame to joint 1, given the angle we are using.

Parameters

th1	angle between base frame and joint 1

Returns

Matrix4d Computed transformation matrix

4.5.2.2 T21()

```
Matrix4d T21 (
            double th2 )
```

Returns the transformation matrix from joint 1 to joint 2, given the angle we are using.

Parameters

th2	angle between joint 1 and joint 2

Returns

Matrix4d Computed transformation matrix

4.5.2.3 T32()

```
Matrix4d T32 (
            double th3 )
```

Returns the transformation matrix from joint 2 to joint 3, given the angle we are using.

Parameters

th3	angle between joint 2 and joint 3

Returns

Matrix4d Computed transformation matrix

4.5.2.4 T43()

```
Matrix4d T43 (
            double th4 )
```

Returns the transformation matrix from joint 3 to joint 4, given the angle we are using.

Parameters

th4	angle between joint 3 and joint 4

Returns

Matrix4d Computed transformation matrix

4.5.2.5 T54()

```
Matrix4d T54 (
            double th5 )
```

Returns the transformation matrix from joint 4 to joint 5, given the angle we are using.

Parameters

th5	angle between joint 4 and joint 5

Returns

Matrix4d Computed transformation matrix

4.5.2.6 T65()

```
Matrix4d T65 (
            double th6 )
```

Returns the transformation matrix from joint 5 to the last joint, given the angle we are using.

Parameters

th6	angle between joint 5 and the last joint

Returns

Matrix4d Computed transformation matrix

4.5.2.7 ur5Direct()

```
std::tuple<Vector3d, Matrix3d> ur5Direct (
            VectorXd Th )
```

Implementation of Direct kinematic for the ur5.

Parameters

Th	vector containing the joint configuration

Returns

std::tuple<Vector3d, Matrix3d> position and rotation of the end effector

4.5.2.8 ur5Inverse()

```
MatrixXd ur5Inverse (
            Vector3d p60,
            Matrix3d R60 )
```

Implementation of Inverse kinematic for the ur5.

Parameters

p60	end effector position
R60	end effector rotation

Returns

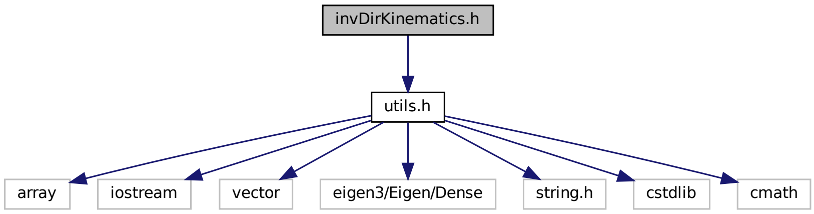
MatrixXd

4.6 invDirKinematics.h File Reference

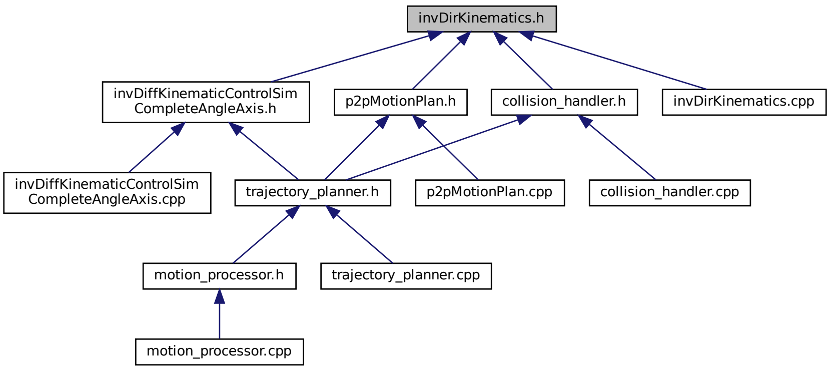
Header file containing the declaration of the functions we use to compute the Direct and Inverse kinematics.

`#include "utils.h"`

Include dependency graph for invDirKinematics.h:



This graph shows which files directly or indirectly include this file:



Functions

- MatrixXd ur5Inverse (Vector3d p60, Matrix3d R60)

    Implementation of Inverse kinematic for the ur5.
- std::tuple< Vector3d, Matrix3d > ur5Direct (VectorXd Th)

    Implementation of Direct kinematic for the ur5.
- Matrix4d T10 (double th1)

    Returns the transformation matrix from base frame to joint 1, given the angle we are using.
- Matrix4d T21 (double th2)

    Returns the transformation matrix from joint 1 to joint 2, given the angle we are using.
- Matrix4d T32 (double th3)

    Returns the transformation matrix from joint 2 to joint 3, given the angle we are using.
- Matrix4d T43 (double th4)

    Returns the transformation matrix from joint 3 to joint 4, given the angle we are using.
- Matrix4d T54 (double th5)

    Returns the transformation matrix from joint 4 to joint 5, given the angle we are using.
- Matrix4d T65 (double th6)

    Returns the transformation matrix from joint 5 to the last joint, given the angle we are using.

4.6.1 Detailed Description

Header file containing the declaration of the functions we use to compute the Direct and Inverse kinematics.

4.6.2 Function Documentation

4.6.2.1 T10()

```
Matrix4d T10 (
            double th1 )
```

Returns the transformation matrix from base frame to joint 1, given the angle we are using.

Parameters

th1	angle between base frame and joint 1

Returns

Matrix4d Computed transformation matrix

4.6.2.2 T21()

```
Matrix4d T21 (
            double th2 )
```

Returns the transformation matrix from joint 1 to joint 2, given the angle we are using.

Parameters

th2	angle between joint 1 and joint 2

Returns

Matrix4d Computed transformation matrix

4.6.2.3 T32()

```
Matrix4d T32 (
            double th3 )
```

Returns the transformation matrix from joint 2 to joint 3, given the angle we are using.

Parameters

th3	angle between joint 2 and joint 3

Returns

Matrix4d Computed transformation matrix

4.6.2.4 T43()

```
Matrix4d T43 (
            double th4 )
```

Returns the transformation matrix from joint 3 to joint 4, given the angle we are using.

Parameters

th4	angle between joint 3 and joint 4

Returns

Matrix4d Computed transformation matrix

4.6.2.5 T54()

```
Matrix4d T54 (
            double th5 )
```

Returns the transformation matrix from joint 4 to joint 5, given the angle we are using.

Parameters

th5	angle between joint 4 and joint 5

Returns

Matrix4d Computed transformation matrix

4.6.2.6 T65()

```
Matrix4d T65 (
            double th6 )
```

Returns the transformation matrix from joint 5 to the last joint, given the angle we are using.

Parameters

th6	angle between joint 5 and the last joint

Returns

Matrix4d Computed transformation matrix

4.6.2.7 ur5Direct()

```
std::tuple<Vector3d, Matrix3d> ur5Direct (
            VectorXd Th )
```

Implementation of Direct kinematic for the ur5.

Parameters

Th	vector containing the joint configuration

Returns

std::tuple<Vector3d, Matrix3d> position and rotation of the end effector

4.6.2.8 ur5Inverse()

```
MatrixXd ur5Inverse (
            Vector3d p60,
            Matrix3d R60 )
```

Implementation of Inverse kinematic for the ur5.

Parameters

p60	end effector position
R60	end effector rotation

Returns

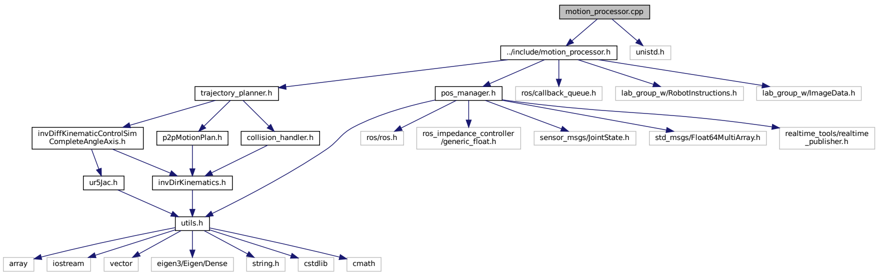
MatrixXd

4.7 motion_processor.cpp File Reference

Implementation of the ROS node motion_processor, that we use to compute and send trajectories to the robot.

```
#include "../include/motion_processor.h"
#include <unistd.h>
```

Include dependency graph for motion_processor.cpp:



Functions

- void recive_jstate (const sensor_msgs::JointState &jointState_msg)

    Callback function used for reading current jointStates from the /ur5/joint_states topic.
- void callback_instructions_sub (const lab_group_w::RobotInstructions &msg)

    Callback function used for reading the next position information received on the /task_planner/robot_instructions custom topic.
- void set_positions (double p0, double p1, double p2)

    Set the positions in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)
- void set_rotations (double r0_0, double r0_1, double r0_2, double r1_0, double r1_1, double r1_2, double r2_0, double r2_1, double r2_2)

    Set the rotations in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)
- MatrixXd get_trajectory ()

    Function used to obtain the trajectory we want to follow, based on the information received from the task planner node (task_planner.cpp)
- int **main** (int argc, char **argv)

4.7.1 Detailed Description

Implementation of the ROS node motion_processor, that we use to compute and send trajectories to the robot.

jointState_msgte_msg

4.7.2 Function Documentation

4.7.2.1 callback_instructions_sub()

```
void callback_instructions_sub (
            const lab_group_w::RobotInstructions & msg )
```

Callback function used for reading the next position information received on the /task_planner/robot_instructions custom topic.

Parameters

msg	RobotInstructions message received from the task planner node (task_planner.cpp)

4.7.2.2 get_trajectory()

```
MatrixXd get_trajectory ( )
```

Function used to obtain the trajectory we want to follow, based on the information received from the task planner node (task_planner.cpp)

Returns

Eigen::MatrixXd matrix containing the trajectory we'll send to the robot

4.7.2.3 recive_jstate()

```
void recive_jstate (
            const sensor_msgs::JointState & jointState_msg_sim )
```

Callback function used for reading current jointStates from the /ur5/joint_states topic.

Parameters

jointState_msg_sim	message obtained from the topic

4.7.2.4 set_positions()

```
void set_positions (
            double p0,
            double p1,
            double p2 )
```

Set the positions in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)

Set the positions in the "RobotInstructions" message we'll send to the motion node.

Parameters

p0	X
p1	Y
p2	Z

4.7.2.5 set_rotations()

```
void set_rotations (
            double r0_0,
            double r0_1,
            double r0_2,
            double r1_0,
            double r1_1,
            double r1_2,
            double r2_0,
            double r2_1,
            double r2_2 )
```

Set the rotations in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)

Set the rotations in the "RobotInstructions" message we'll send to the motion node (saved as a Rotation matrix)

Parameters

r0←_0	
r0←_1	
r0←_2	
r1←_0	
r1←_1	
r1←_2	
r2←_0	
r2←_1	
r2←_2	

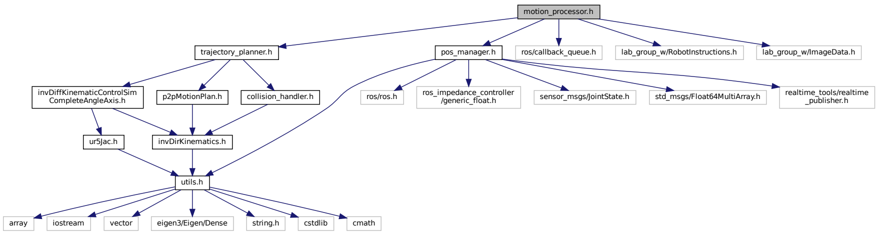
4.8 motion_processor.h File Reference

Header of the motion_processor.cpp file.

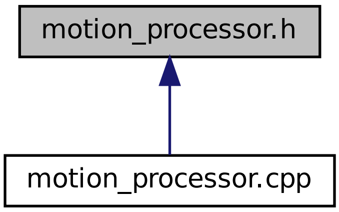
```
#include "trajectory_planner.h"
#include "pos_manager.h"
#include <ros/callback_queue.h>
#include "lab_group_w/RobotInstructions.h"
```

```
#include "lab_group_w/ImageData.h"
```

Include dependency graph for motion_processor.h:



This graph shows which files directly or indirectly include this file:



Functions

- void recive_jstate (const sensor_msgs::JointState &jointState_msg_sim)

    Callback function used for reading current jointStates from the /ur5/joint_states topic.
- std::vector< double > q (6, 0.0)

    vector where we save the joint values obtained in recive_jstate()
- void set_positions (double p0, double p1, double p2)

    Set the positions in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)
- void set_rotations (double r0_0, double r0_1, double r0_2, double r1_0, double r1_1, double r1_2, double r2_0, double r2_1, double r2_2)

    Set the rotations in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)
- void callback_instructions_sub (const lab_group_w::RobotInstructions &msg)

    Callback function used for reading the next position information received on the /task_planner/robot_instructions custom topic.
- Eigen::MatrixXd get_trajectory ()

    Function used to obtain the trajectory we want to follow, based on the information received from the task planner node (task_planner.cpp)

Variables

- std::vector< std::string > joint_names = {"shoulder_pan_joint", "shoulder_lift_joint", "elbow_joint", "wrist_↩ 1_joint", "wrist_2_joint", "wrist_3_joint"}

  vector containing the joint names, used to read the joint values in the recive_jstate() callback function
- ros::Subscriber rec_jstate

  subscriber to the /ur5/joint_states topic
- double diameter

  diameter used for the gripper
- const double loop_frequency = 1000.0

  Frequency used to coordinate with other nodes.
- const double dt = 1/loop_frequency

  variable used for synchronization (will be used in p2pMotionPlan.cpp)
- ros::Subscriber sub_instructions

  subscriber to the /task_planner/robot_instructions topic
- lab_group_w::RobotInstructions instructions

  message that the we receive from the task planner node (task_planner.cpp)
- ros::CallbackQueue instructions_CallbackQueue

  CallbackQueue used to coordinate when reading RobotInstructions from the /task_planner/robot_instructions topic.
- ros::CallbackQueue jstates_CallbackQueue

  CallbackQueue used to coordinate when reading jstates from the /ur5/joint_states topic.
- ros::ServiceClient gripper_client

  Set service client for move_gripper service.
- ros_impedance_controller::generic_float srv

  message used for the move_gripper service (containing the diameter for the gripper)

4.8.1 Detailed Description

Header of the motion_processor.cpp file.

Motion planner is the ROS node we use to move the robot from a starting position to a desired one (received from task_planner.cpp)

4.8.2 Function Documentation

4.8.2.1 callback_instructions_sub()

```
void callback_instructions_sub (
            const lab_group_w::RobotInstructions & msg )
```

Callback function used for reading the next position information received on the /task_planner/robot_instructions custom topic.

Parameters

msg	RobotInstructions message received from the task planner node (task_planner.cpp)

4.8.2.2 get_trajectory()

```
Eigen::MatrixXd get_trajectory ( )
```

Function used to obtain the trajectory we want to follow, based on the information received from the task planner node (task_planner.cpp)

Returns

Eigen::MatrixXd matrix containing the trajectory we'll send to the robot

4.8.2.3 recive_jstate()

```
void recive_jstate (
            const sensor_msgs::JointState & jointState_msg_sim )
```

Callback function used for reading current jointStates from the /ur5/joint_states topic.

Parameters

jointState_msg_sim	message obtained from the topic

4.8.2.4 set_positions()

```
void set_positions (
            double p0,
            double p1,
            double p2 )
```

Set the positions in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)

Parameters

p0	X
p1	Y
p2	Z

4.8.2.5 set_rotations()

```
void set_rotations (
            double r0_0,
            double r0_1,
            double r0_2,
            double r1_0,
            double r1_1,
            double r1_2,
            double r2_0,
            double r2_1,
            double r2_2 )
```

Set the rotations in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)

Parameters

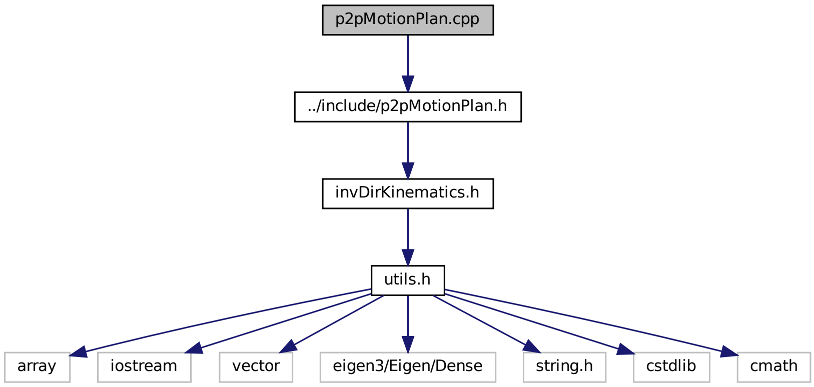
r0_0	
r0_1	
r0_2	
r1_0	
r1_1	
r1_2	
r2_0	
r2_1	
r2_2	

4.9 p2pMotionPlan.cpp File Reference

Implementation of the functions we use to compute the trajectories.

```
#include "../include/p2pMotionPlan.h"
```

Include dependency graph for p2pMotionPlan.cpp:



Functions

- std::tuple< MatrixXd, MatrixXd, MatrixXd > p2pMotionPlan (const VectorXd &xEs, const VectorXd &xEf, const double minT, const double maxT, const double dt, const int total_steps)

    Function that computes trajectories using a cubic polynomial.
- double limitJointAngle (double angle, double minAngle, double maxAngle)

    Function that checks if a recieved joint angle surpasses its limits and eventually modifies it.
    This is called for every joint by fix_joint_config()
- VectorXd fix_joint_config (const VectorXd &conf)

    Function used to "correct" joint angles if they go beyond their limits, the actual modification is done by calling limitJointAngle()
- std::tuple< MatrixXd, MatrixXd, MatrixXd > p2via2pMotionPlan (const std::vector< VectorXd > &conf, const std::vector< double > &times, const double dt, const int total_steps)

    Function that computes trajectories using a cubic polynomial, adapted to recieve multiple points to pass trough (NOT USED)

4.9.1 Detailed Description

Implementation of the functions we use to compute the trajectories.

4.9.2 Function Documentation

4.9.2.1 fix_joint_config()

```
VectorXd fix_joint_config (
            const VectorXd & conf )
```

Function used to "correct" joint angles if they go beyond their limits, the actual modification is done by calling limitJointAngle()

Parameters

conf	joint angles we want to check

Returns

VectorXd corrected joint angles

4.9.2.2 limitJointAngle()

```
double limitJointAngle (
            double angle,
            double minAngle,
            double maxAngle )
```

Function that checks if a recieved joint angle surpasses its limits and eventually modifies it.
This is called for every joint by fix_joint_config()

Parameters

angle	joint angle we want to check
minAngle	minimum value for the recieved joint
maxAngle	maximum value for the recieved joint

Returns

double corrected joint angle

4.9.2.3 p2pMotionPlan()

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> p2pMotionPlan (
            const VectorXd & xEs,
            const VectorXd & xEf,
            const double minT,
            const double maxT,
            const double dt,
            const int total_steps )
```

Function that computes trajectories using a cubic polynomial.

Parameters

xEs	starting end effector position
xEf	final end effector position
minT	start time for the motion plan
maxT	finish time for the motion plan
dt	sampling time
total_steps	number of configurations we'll sample

Returns

std::tuple<MatrixXd, MatrixXd, MatrixXd> computed trajectory (joint positions, end effector positions, end effector rotations)

4.9.2.4 p2via2pMotionPlan()

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> p2via2pMotionPlan (
            const std::vector< VectorXd > & conf,
            const std::vector< double > & times,
            const double dt,
            const int total_steps )
```

Function that computes trajectories using a cubic polynomial, adapted to recieve multiple points to pass trough (NOT USED)

Parameters

conf	points we want our trajectory to cover
times	times at which we want to reach each point
dt	sampling time
total_steps	number of configurations we'll sample

Returns

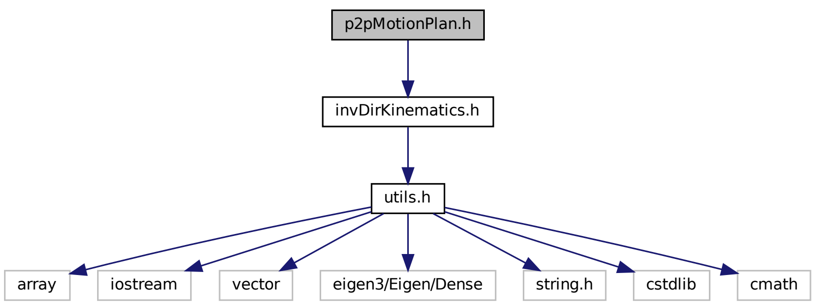
std::tuple<MatrixXd, MatrixXd, MatrixXd> computed trajectory (joint positions, end effector positions, end effector rotations)

4.10 p2pMotionPlan.h File Reference

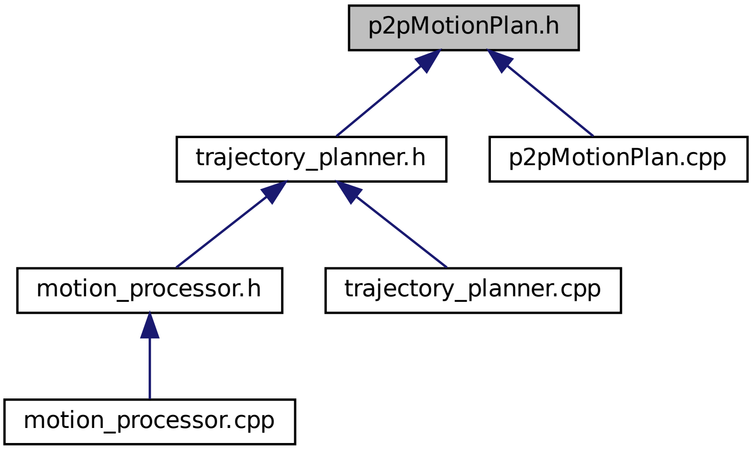
Header of the p2pMotionPlan.cpp file, used to compute trajectories.

```
#include "invDirKinematics.h"
```

Include dependency graph for p2pMotionPlan.h:



This graph shows which files directly or indirectly include this file:



Functions

- std::tuple< MatrixXd, MatrixXd, MatrixXd > p2pMotionPlan (const VectorXd &xEs, const VectorXd &xEf, const double minT, const double maxT, const double dt, const int total_steps)

    Function that computes trajectories using a cubic polynomial.
- std::tuple< MatrixXd, MatrixXd, MatrixXd > p2via2pMotionPlan (const std::vector< VectorXd > &conf, const std::vector< double > &times, const double dt, const int total_steps)

    Function that computes trajectories using a cubic polynomial, adapted to recieve multiple points to pass trough (NOT USED)
- VectorXd fix_joint_config (const VectorXd &conf)

    Function used to "correct" joint angles if they go beyond their limits, the actual modification is done by calling limitJointAngle()
- double limitJointAngle (double angle, double minAngle, double maxAngle)

    Function that checks if a recieved joint angle surpasses its limits and eventually modifies it. This is called for every joint by fix_joint_config()

4.10.1 Detailed Description

Header of the p2pMotionPlan.cpp file, used to compute trajectories.

4.10.2 Function Documentation

4.10.2.1 fix_joint_config()

```
VectorXd fix_joint_config (
            const VectorXd & conf )
```

Function used to "correct" joint angles if they go beyond their limits, the actual modification is done by calling limitJointAngle()

Parameters

conf	joint angles we want to check

Returns

VectorXd corrected joint angles

4.10.2.2 limitJointAngle()

```
double limitJointAngle (
            double angle,
            double minAngle,
            double maxAngle )
```

Function that checks if a recieved joint angle surpasses its limits and eventually modifies it.
This is called for every joint by fix_joint_config()

Parameters

angle	joint angle we want to check
minAngle	minimum value for the recieved joint
maxAngle	maximum value for the recieved joint

Returns

double corrected joint angle

4.10.2.3 p2pMotionPlan()

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> p2pMotionPlan (
            const VectorXd & xEs,
            const VectorXd & xEf,
            const double minT,
            const double maxT,
            const double dt,
            const int total_steps )
```

Function that computes trajectories using a cubic polynomial.

Parameters

xEs	starting end effector position
xEf	final end effector position
minT	start time for the motion plan
maxT	finish time for the motion plan
dt	sampling time
total_steps	number of configurations we'll sample

Returns

std::tuple<MatrixXd, MatrixXd, MatrixXd> computed trajectory (joint positions, end effector positions, end effector rotations)

4.10.2.4 p2via2pMotionPlan()

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> p2via2pMotionPlan (
            const std::vector< VectorXd > & conf,
            const std::vector< double > & times,
            const double dt,
            const int total_steps )
```

Function that computes trajectories using a cubic polynomial, adapted to recieve multiple points to pass trough (NOT USED)

Parameters

conf	points we want our trajectory to cover
times	times at which we want to reach each point
dt	sampling time
total_steps	number of configurations we'll sample

Returns

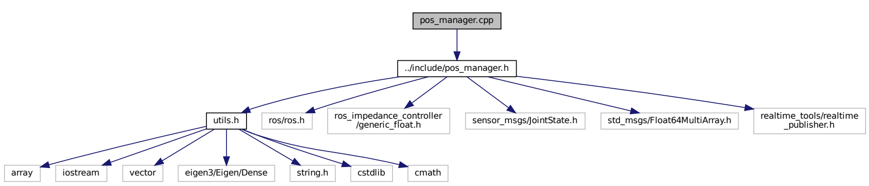
std::tuple<MatrixXd, MatrixXd, MatrixXd> computed trajectory (joint positions, end effector positions, end effector rotations)

4.11 pos_manager.cpp File Reference

Implementation of the class we use to comunicate with the real/simulated robot.

```
#include "../include/pos_manager.h"
```

Include dependency graph for pos_manager.cpp:



4.11.1 Detailed Description

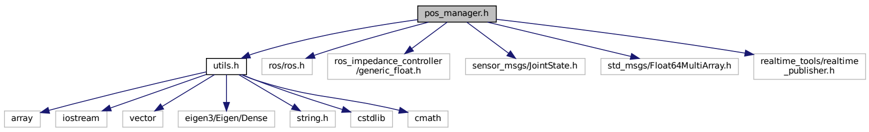
Implementation of the class we use to comunicate with the real/simulated robot.

4.12 pos_manager.h File Reference

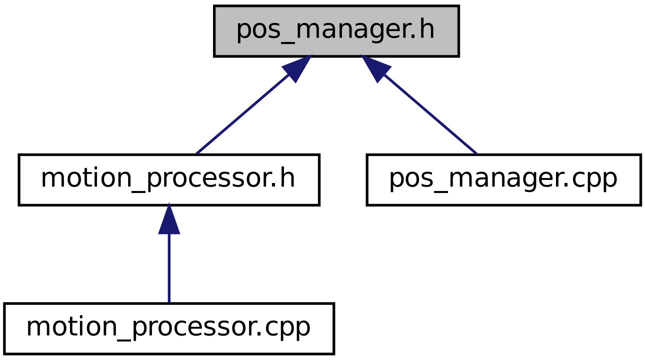
Header of the class Pos_manager, used to handle publishers.

```
#include "utils.h"
#include "ros/ros.h"
#include "ros_impedance_controller/generic_float.h"
#include <sensor_msgs/JointState.h>
#include <std_msgs/Float64MultiArray.h>
#include <realtime_tools/realtime_publisher.h>
```

Include dependency graph for pos_manager.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Pos_manager

    pos_manager is a class that we use to publish joint angles for the real/simulated robot

4.12.1 Detailed Description

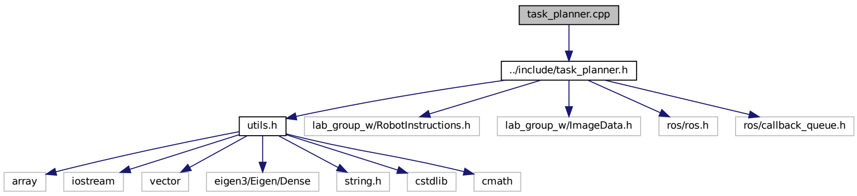
Header of the class Pos_manager, used to handle publishers.

4.13 task_planner.cpp File Reference

Implementation of the ROS node task_planner, that we use to send destinations to the motion node.

```
#include "../include/task_planner.h"
```

Include dependency graph for task_planner.cpp:



Functions

- void callback_sub (const lab_group_w::ImageData &msg)

  Callback function used to store the informations of the detected object obtained from the /image←Processor/processed_data custom topic.
- void choose_destination ()

  Function that decides where to put the blocks (calling set_positions()) depending on its class (saved inside instructions.class_type) received from the image detection node.
- void set_positions (double p0, double p1, double p2)

  Set the positions in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)
- void set_rotations (double r0_0, double r0_1, double r0_2, double r1_0, double r1_1, double r1_2, double r2_0, double r2_1, double r2_2)

  Set the rotations in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)
- void set_diameter (int class_type, bool soft_gripper)

  Set the gripper diameter in the "RobotInstructions" message that we send to the motion node.
- int **main** (int argc, char **argv)

4.13.1 Detailed Description

Implementation of the ROS node task_planner, that we use to send destinations to the motion node.

4.13.2 Function Documentation

4.13.2.1 callback_sub()

```
void callback_sub (
            const lab_group_w::ImageData & msg )
```

Callback function used to store the informations of the detected object obtained from the /image←Processor/processed_data custom topic.

Parameters

msg	ImageData message received from the vision node

4.13.2.2 choose_destination()

```
void choose_destination ( )
```

Function that decides where to put the blocks (calling set_positions()) depending on its class (saved inside instructions.class_type) received from the image detection node.

4.13.2.3 set_diameter()

```
void set_diameter (
            int class_type,
            bool soft_gripper )
```

Set the gripper diameter in the "RobotInstructions" message that we send to the motion node.

Parameters

class_type	class type of the detected object
soft_gripper	variable used to differentiate between soft and rigid gripper

4.13.2.4 set_positions()

```
void set_positions (
            double p0,
            double p1,
            double p2 )
```

Set the positions in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)

Set the positions in the "RobotInstructions" message we'll send to the motion node.

Parameters

p0	X
p1	Y
p2	Z

4.13.2.5 set_rotations()

```
void set_rotations (
            double r0_0,
            double r0_1,
            double r0_2,
            double r1_0,
            double r1_1,
            double r1_2,
            double r2_0,
            double r2_1,
            double r2_2 )
```

Set the rotations in the "RobotInstructions" message using the information we received from the task planner node (task_planner.cpp)

Set the rotations in the "RobotInstructions" message we'll send to the motion node (saved as a Rotation matrix)

Parameters

r0←_0	
r0←_1	
r0←_2	
r1←_0	
r1←_1	
r1←_2	
r2←_0	
r2←_1	
r2←_2	

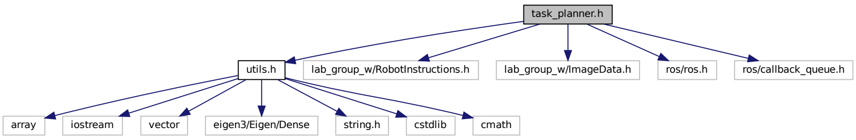
4.14 task_planner.h File Reference

Header with the declaration of the functions implemented in task_planner.cpp.

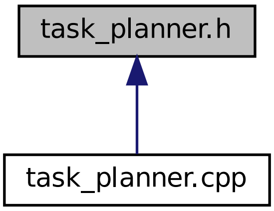
```
#include "utils.h"
#include "lab_group_w/RobotInstructions.h"
#include "lab_group_w/ImageData.h"
#include "ros/ros.h"
```

```
#include <ros/callback_queue.h>
```

Include dependency graph for task_planner.h:



This graph shows which files directly or indirectly include this file:



Functions

- void set_positions (double p0, double p1, double p2)

  Set the positions in the "RobotInstructions" message we'll send to the motion node.
- void set_rotations (double r0_0, double r0_1, double r0_2, double r1_0, double r1_1, double r1_2, double r2_0, double r2_1, double r2_2)

  Set the rotations in the "RobotInstructions" message we'll send to the motion node (saved as a Rotation matrix)
- void set_diameter (int class_type, bool soft_gripper)

  Set the gripper diameter in the "RobotInstructions" message that we send to the motion node.
- void choose_destination ()

  Function that decides where to put the blocks (calling set_positions()) depending on its class (saved inside instructions.class_type) received from the image detection node.
- void callback_sub (const lab_group_w::ImageData &msg)

  Callback function used to store the informations of the detected object obtained from the /imageProcessor/processed_data custom topic.

Variables

- lab_group_w::RobotInstructions instructions

  message that the task planner sends to the motion node (motion_planner.cpp)
- ros::Publisher pub_instructions

  *publisher for the **/task_planner/robot_instructions** custom topic (RobotInstructions message)*
- ros::Subscriber sub_image_data

  *subscriber to the **/imageProcessor/processed_data** custom topic (ImageData message)*
- double loop_frequency = 1000.

  Frequency used to coordinate with other nodes.
- ros::CallbackQueue imageData_CallbackQueue

  *CallbackQueue for the **/imageProcessor/processed_data** custom topic, used to wait for messages.*

4.14.1 Detailed Description

Header with the declaration of the functions implemented in task_planner.cpp.

task_planner is the ROS node we use to decide destinations for the robot arm

4.14.2 Function Documentation

4.14.2.1 callback_sub()

```
void callback_sub (
            const lab_group_w::ImageData & msg )
```

Callback function used to store the informations of the detected object obtained from the /image←Processor/processed_data custom topic.

Parameters

msg	ImageData message received from the vision node

4.14.2.2 choose_destination()

```
void choose_destination ( )
```

Function that decides where to put the blocks (calling set_positions()) depending on its class (saved inside instructions.class_type) received from the image detection node.

4.14.2.3 set_diameter()

```
void set_diameter (
            int class_type,
            bool soft_gripper )
```

Set the gripper diameter in the "RobotInstructions" message that we send to the motion node.

Parameters

class_type	class type of the detected object
soft_gripper	variable used to differentiate between soft and rigid gripper

4.14.2.4 set_positions()

```
void set_positions (
            double p0,
            double p1,
            double p2 )
```

Set the positions in the "RobotInstructions" message we'll send to the motion node.

Parameters

p0	X
p1	Y
p2	Z

Set the positions in the "RobotInstructions" message we'll send to the motion node.

Parameters

p0	X
p1	Y
p2	Z

4.14.2.5 set_rotations()

```
void set_rotations (
            double r0_0,
            double r0_1,
            double r0_2,
            double r1_0,
            double r1_1,
            double r1_2,
            double r2_0,
            double r2_1,
            double r2_2 )
```

Set the rotations in the "RobotInstructions" message we'll send to the motion node (saved as a Rotation matrix)

Parameters

r0← _0	
r0← _1	
r0← _2	
r1← _0	
r1← _1	

Parameters

r1←_2	
r2←_0	
r2←_1	
r2←_2	

Set the rotations in the "RobotInstructions" message we'll send to the motion node (saved as a Rotation matrix)

Parameters

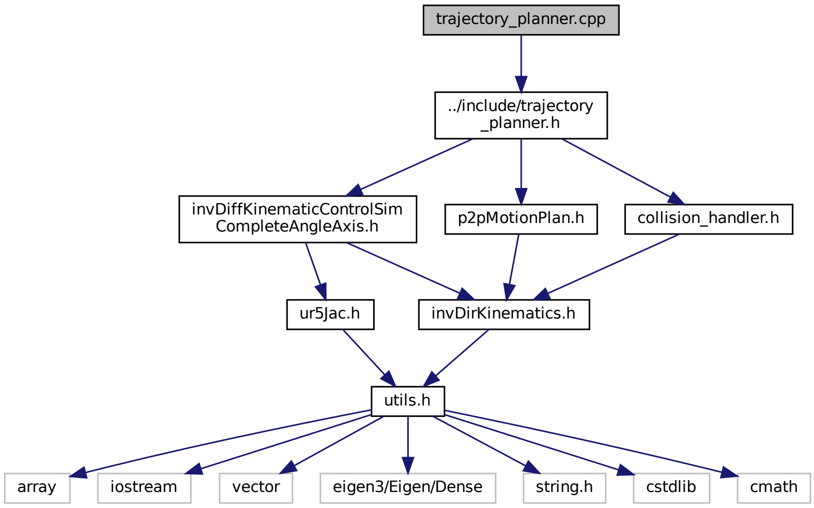
r0←_0	
r0←_1	
r0←_2	
r1←_0	
r1←_1	
r1←_2	
r2←_0	
r2←_1	
r2←_2	

4.15 trajectory_planner.cpp File Reference

Implementation of the functions we use to compute a trajectory for the robot arm.

`#include "../include/trajectory_planner.h"`

Include dependency graph for trajectory_planner.cpp:



Functions

- std::array< bool, 8 > exclude_invalid_confs (const MatrixXd &Th, const Vector3d &goal_point)

  Function that checks if the joint configurations obtained from inverse kinematic are vaild.
- std::array< int, 8 > sort_confs (const MatrixXd &Th, const VectorXd &q_0, const std::array< bool, 8 > &valid_config)

  Function that sorts the valid configurations from the one that has the minimum difference in overall joint angles with the starting configuration.
- double find_optimal_maxT (const VectorXd &q_0, const VectorXd &q_f, const double scaling_factor, const double minimum_time)

  Function that decides the time a trajectory should take to complete.
- MatrixXd get_best_config (const MatrixXd &Th, const VectorXd &q_0, const Vector3d &goal_point, const double dt)

  Returns the trajectory that we use to reach a goal point.

4.15.1 Detailed Description

Implementation of the functions we use to compute a trajectory for the robot arm.

4.15.2 Function Documentation

4.15.2.1 exclude_invalid_confs()

```
std::array<bool,8> exclude_invalid_confs (
            const MatrixXd & Th,
            const Vector3d & goal_point )
```

Function that checks if the joint configurations obtained from inverse kinematic are vaild.

Parameters

Th	joint configurations obtained from inverse kinematic
goal_point	point we want the end effector to reach

Returns

std::array<bool,8> array containing the validity of the configurations (true = VALID, false = NOT VALID)

4.15.2.2 find_optimal_maxT()

```
double find_optimal_maxT (
            const VectorXd & q_0,
            const VectorXd & q_f,
            const double scaling_factor,
            const double minimum_time )
```

Function that decides the time a trajectory should take to complete.

Parameters

q_0	joint configuration at the start of the trajectory
q_f	joint configuration at the end of the trajectory
scaling_factor	scaling factor that multiplies the maximum joint velocity in order to scale it down (we don't want the joint to move at maximum speed)
minimum_time	minimum time that is added to the computed trajectory time to have even more control over the final computed time

Returns

double maxT used to compute the trajectory between the two passed joint configurations

4.15.2.3 get_best_config()

```
MatrixXd get_best_config (
            const MatrixXd & Th,
            const VectorXd & q_0,
            const Vector3d & goal_point,
            const double dt )
```

Returns the trajectory that we use to reach a goal point.

Parameters

Th	joint configurations obtained from inverse kinematic
q_0	initial joint configuration
goal_point	point we want the end effector to reach
dt	time interval that we use to decide how many points a trajectory should have

Returns

MatrixXd containing the trajectory we'll use to reach the desired point

4.15.2.4 sort_confs()

```
std::array<int, 8> sort_confs (
            const MatrixXd & Th,
            const VectorXd & q_0,
            const std::array< bool, 8 > & valid_config )
```

Function that sorts the valid configurations from the one that has the minimum difference in overall joint angles with the starting configuration.

Parameters

Th	joint configurations obtained from inverse kinematic
q_0	initial joint configuration
valid_config	array containing the validity of the configurations (true = VALID, false = NOT VALID)

Returns

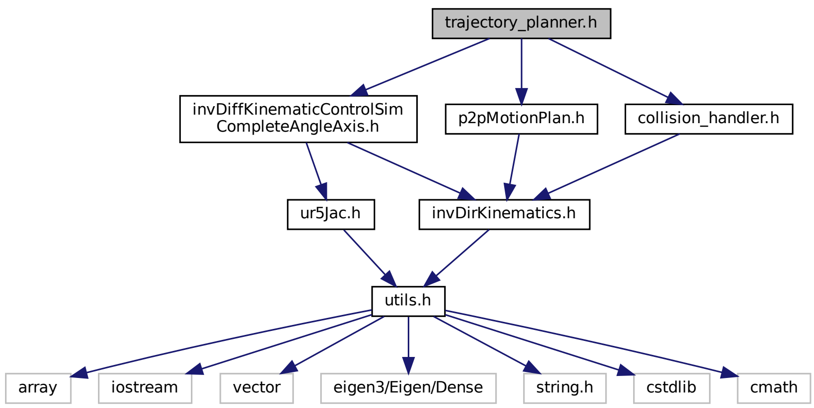
std::array<int, 8> sorted vector that contains the ordered indexes of the joint configurations inside Th

4.16 trajectory_planner.h File Reference

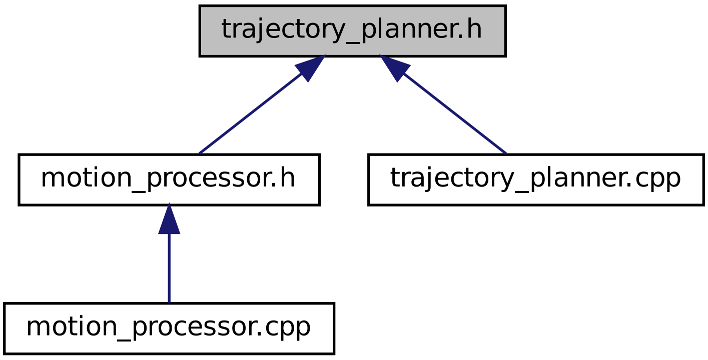
Header with the declaration of the functions implemented in trajectory_planner.cpp.

```
#include "invDiffKinematicControlSimCompleteAngleAxis.h"
#include "p2pMotionPlan.h"
#include "collision_handler.h"
```

Include dependency graph for trajectory_planner.h:



This graph shows which files directly or indirectly include this file:



Functions

- MatrixXd get_best_config (const MatrixXd &Th, const VectorXd &q_0, const Vector3d &goal_point, const double dt)

    Returns the trajectory that we use to reach a goal point.
- std::array< bool, 8 > exclude_invalid_confs (const MatrixXd &Th, const Vector3d &goal_point)

    Function that checks if the joint configurations obtained from inverse kinematic are vaild.
- std::array< int, 8 > sort_confs (const MatrixXd &Th, const VectorXd &q_0, const std::array< bool, 8 > &valid_config)

    Function that sorts the valid configurations from the one that has the minimum difference in overall joint angles with the starting configuration.
- double find_optimal_maxT (const VectorXd &q_0, const VectorXd &q_f, const double scaling_factor, const double minimum_time)

    Function that decides the time a trajectory should take to complete.

4.16.1 Detailed Description

Header with the declaration of the functions implemented in trajectory_planner.cpp.

4.16.2 Function Documentation

4.16.2.1 exclude_invalid_confs()

```
std::array<bool,8> exclude_invalid_confs (
            const MatrixXd & Th,
            const Vector3d & goal_point )
```

Function that checks if the joint configurations obtained from inverse kinematic are vaild.

Parameters

Th	joint configurations obtained from inverse kinematic
goal_point	point we want the end effector to reach

Returns

std::array<bool,8> array containing the validity of the configurations (true = VALID, false = NOT VALID)

4.16.2.2 find_optimal_maxT()

```
double find_optimal_maxT (
            const VectorXd & q_0,
            const VectorXd & q_f,
            const double scaling_factor,
            const double minimum_time )
```

Function that decides the time a trajectory should take to complete.

Parameters

q_0	joint configuration at the start of the trajectory
q_f	joint configuration at the end of the trajectory
scaling_factor	scaling factor that multiplies the maximum joint velocity in order to scale it down (we don't want the joint to move at maximum speed)
minimum_time	minimum time that is added to the computed trajectory time to have even more control over the final computed time

Returns

double maxT used to compute the trajectory between the two passed joint configurations

4.16.2.3 get_best_config()

```
MatrixXd get_best_config (
            const MatrixXd & Th,
            const VectorXd & q_0,
            const Vector3d & goal_point,
            const double dt )
```

Returns the trajectory that we use to reach a goal point.

Parameters

Th	joint configurations obtained from inverse kinematic
q_0	initial joint configuration
goal_point	point we want the end effector to reach
dt	time interval that we use to decide how many points a trajectory should have

Returns

MatrixXd containing the trajectory we'll use to reach the desired point

4.16.2.4 sort_confs()

```
std::array<int, 8> sort_confs (
            const MatrixXd & Th,
            const VectorXd & q_0,
            const std::array< bool, 8 > & valid_config )
```

Function that sorts the valid configurations from the one that has the minimum difference in overall joint angles with the starting configuration.

Parameters

Th	joint configurations obtained from inverse kinematic
q_0	initial joint configuration
valid_config	array containing the validity of the configurations (true = VALID, false = NOT VALID)

Returns

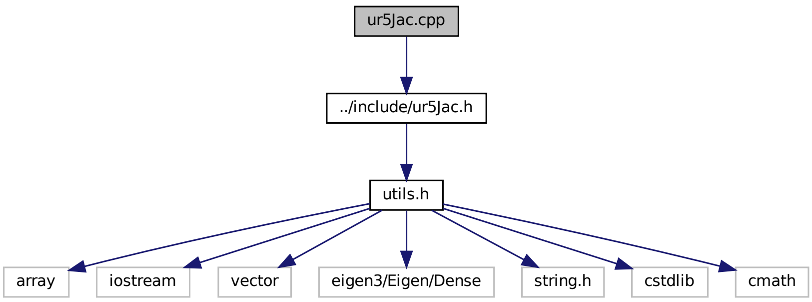
std::array<int, 8> sorted vector that contains the ordered indexes of the joint configurations inside Th

4.17 ur5Jac.cpp File Reference

implementation of the Jacobian for the ur5

```
#include "../include/ur5Jac.h"
```

Include dependency graph for ur5Jac.cpp:



Functions

- MatrixXd ur5Jac (VectorXd Th)

  Compute the jacobian matrix for a specified joint configuration.

4.17.1 Detailed Description

implementation of the Jacobian for the ur5

4.17.2 Function Documentation

4.17.2.1 ur5Jac()

```
MatrixXd ur5Jac (
            VectorXd Th )
```

Compute the jacobian matrix for a specified joint configuration.

19

Parameters

Th	joint configuration we want to compute the jacobian for

Returns

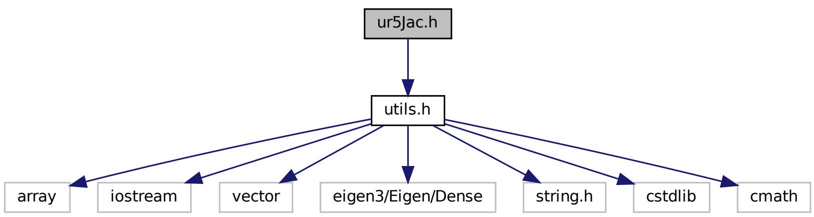
Resulting Jacobian Matrix

4.18 ur5Jac.h File Reference

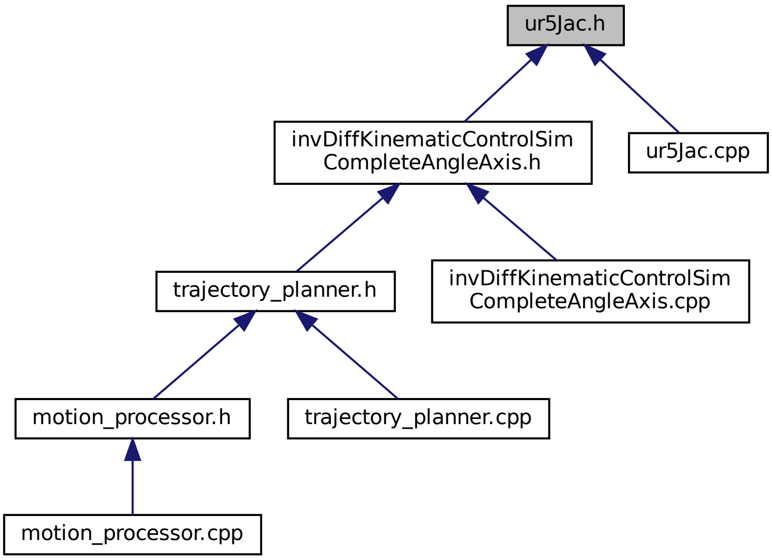
Header with the declaration of the function we use to compute the jacobian.

```
#include "utils.h"
```

Include dependency graph for ur5Jac.h:



This graph shows which files directly or indirectly include this file:



Functions

- MatrixXd ur5Jac (VectorXd Th)

  Compute the jacobian matrix for a specified joint configuration.

4.18.1 Detailed Description

Header with the declaration of the function we use to compute the jacobian.

4.18.2 Function Documentation

4.18.2.1 ur5Jac()

```
MatrixXd ur5Jac (
            VectorXd Th )
```

Compute the jacobian matrix for a specified joint configuration.

19

Parameters

Th	joint configuration we want to compute the jacobian for

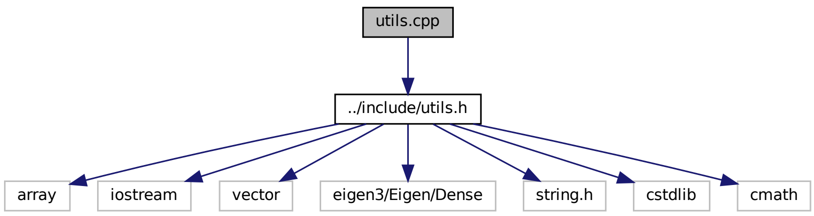
Returns

Resulting Jacobian Matrix

4.19 utils.cpp File Reference

Implementation of commonly used functions.

`#include "../include/utils.h"`
Include dependency graph for utils.cpp:



Functions

- Vector3d W_t_R_transform (const Vector3d &pos)

  Function that transforms a point from World frame to Robot frame.
- Vector3d R_t_W_transform (const Vector3d &pos)

  Function that transforms a point from Robot frame to World frame.
- Matrix3d eul2rotmFDR (const Vector3d &eulXYZ)

  Function that translates from Euler angles to rotation matrix.
- Vector3d rotm2eulFDR (const Matrix3d &R)

  Function that translates from rotation matrix to Euler angles.

4.19.1 Detailed Description

Implementation of commonly used functions.

4.19.2 Function Documentation

4.19.2.1 eul2rotmFDR()

```
Matrix3d eul2rotmFDR (
            const Vector3d & eulXYZ )
```

Function that translates from Euler angles to rotation matrix.

Parameters

eulXYZ	Euler angles we want to transform (XYZ vector)

Returns

Matrix3d containing the rotation matrix (ZYX matrix) obtained from the received Euler angles

4.19.2.2 R_t_W_transform()

```
Vector3d R_t_W_transform (
            const Vector3d & pos )
```

Function that transforms a point from Robot frame to World frame.

Parameters

pos	point we want to transform

Returns

Vector3d containing the point translated in the World frame

4.19.2.3 rotm2eulFDR()

```
Vector3d rotm2eulFDR (
            const Matrix3d & R )
```

Function that translates from rotation matrix to Euler angles.

Parameters

R	rotation matrix we want to transform (ZYX matrix)

Returns

Vector3d containing the Euler angles (XYZ vector) obtained from the received rotation matrix

4.19.2.4 W_t_R_transform()

```
Vector3d W_t_R_transform (
            const Vector3d & pos )
```

Function that transforms a point from World frame to Robot frame.

Parameters

pos	point we want to transform

Returns

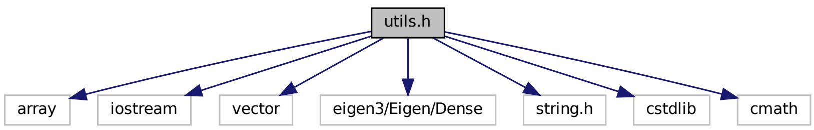
Vector3d containing the point translated in the Robot frame

4.20 utils.h File Reference

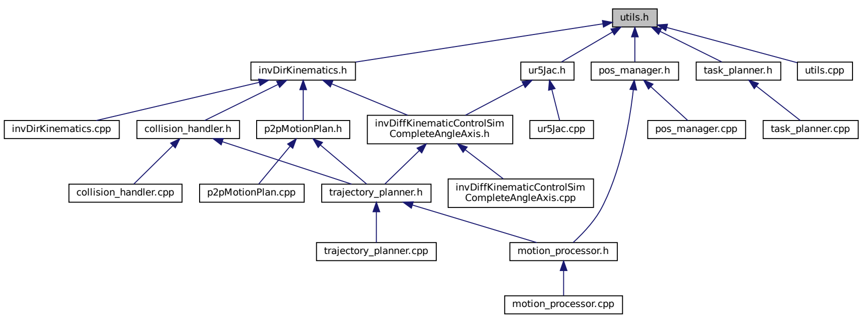
Header containing commonly used functions and libraries.

```
#include <array>
#include <iostream>
#include <vector>
#include <eigen3/Eigen/Dense>
#include <string.h>
#include <cstdlib>
#include <cmath>
```

Include dependency graph for utils.h:



This graph shows which files directly or indirectly include this file:



Functions

- Vector3d W_t_R_transform (const Vector3d &pos)

  Function that transforms a point from World frame to Robot frame.
- Vector3d R_t_W_transform (const Vector3d &pos)

  Function that transforms a point from Robot frame to World frame.
- Matrix3d eul2rotmFDR (const Vector3d &eulXYZ)

  Function that translates from Euler angles to rotation matrix.
- Vector3d rotm2eulFDR (const Matrix3d &R)

  Function that translates from rotation matrix to Euler angles.

4.20.1 Detailed Description

Header containing commonly used functions and libraries.

4.20.2 Function Documentation

4.20.2.1 eul2rotmFDR()

```
Matrix3d eul2rotmFDR (
            const Vector3d & eulXYZ )
```

Function that translates from Euler angles to rotation matrix.

Parameters

eulXYZ	Euler angles we want to transform (XYZ vector)

Returns

Matrix3d containing the rotation matrix (ZYX matrix) obtained from the received Euler angles

4.20.2.2 R_t_W_transform()

```
Vector3d R_t_W_transform (
            const Vector3d & pos )
```

Function that transforms a point from Robot frame to World frame.

Parameters

pos	point we want to transform

Returns

Vector3d containing the point translated in the World frame

4.20.2.3 rotm2eulFDR()

```
Vector3d rotm2eulFDR (
            const Matrix3d & R )
```

Function that translates from rotation matrix to Euler angles.

Parameters

R	rotation matrix we want to transform (ZYX matrix)

Returns

Vector3d containing the Euler angles (XYZ vector) obtained from the received rotation matrix

4.20.2.4 W_t_R_transform()

```
Vector3d W_t_R_transform (
            const Vector3d & pos )
```

Function that transforms a point from World frame to Robot frame.

Parameters

pos	point we want to transform

Returns

Vector3d containing the point translated in the Robot frame