### How to select the expected path

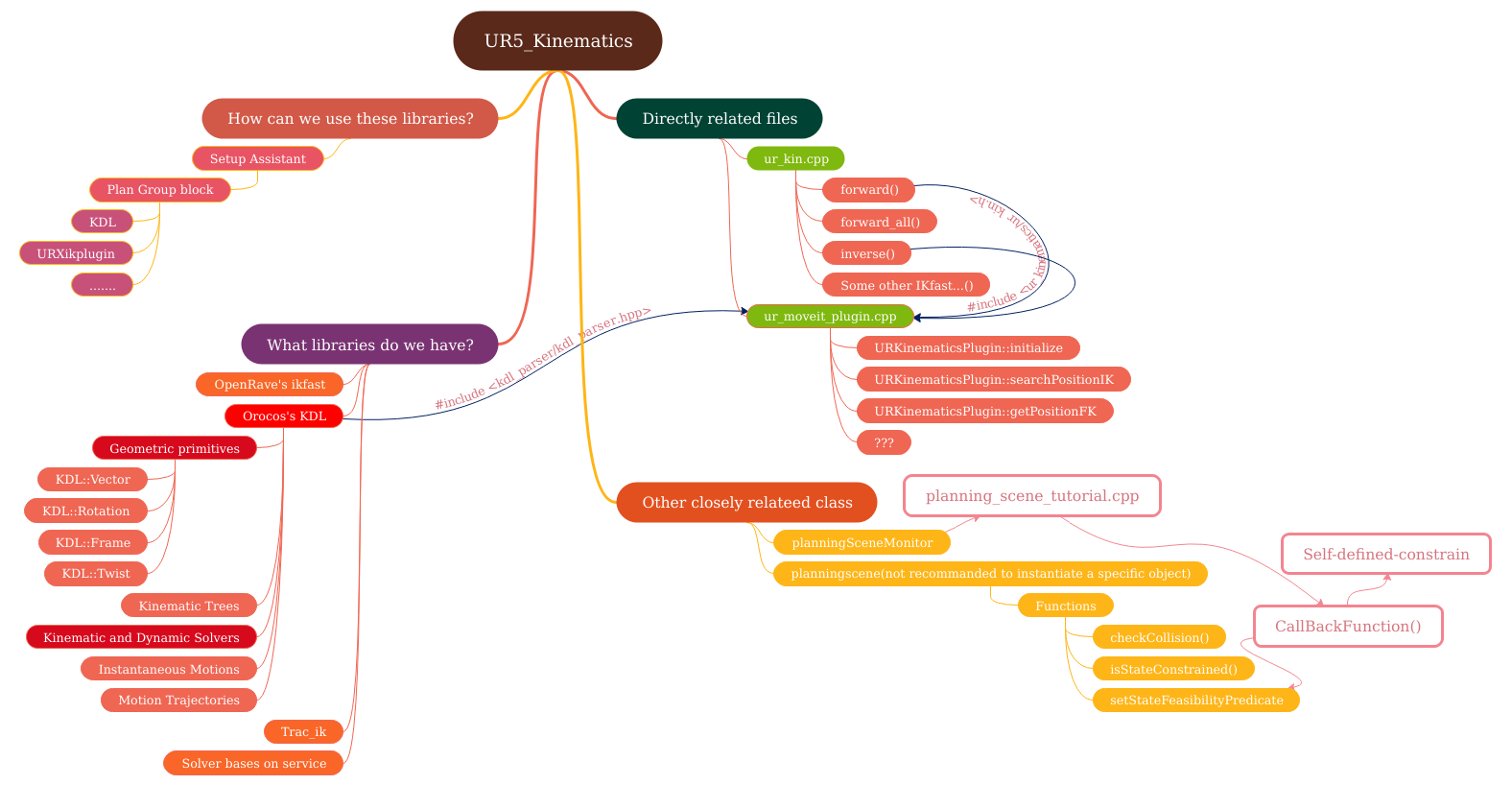
## [Description of the task]

The aim of this task is to find a way to select the expected and also valid solution manually. Also, if given a constrain to the robot, for example, if we want it to deliver a glass of water from one place to another, the glass must remain steady on its way to the expected spot, which means the robot's wrist joint angle must be pre-defined and remain unchanged during the whole route, we would like the robot follows the constrain while it attempts to find valid solutions.

## [Overview of achievement]

After going deep down into the source code in ur\_kin.cpp and ur\_moveit\_plugin.cpp, I have got a primary view on the mechanism of solving the inverse kinematics and filtering out the best(most reasonable) solution in at most eight solutions. However, since at the most beginning I did not use the ROS editor “Roboware-Studio”, it became nearly impossible for me to uncover the whole mechanism of ROS's solution selecting process, which contains lots of other deeper interconnections in dozens of forms like using the callback-function and subscribing the tf trees and so on. I need more time to uncover the mask of ROS even with the powerful tool Roboware-Studio.

## [Mystery that has been uncovered]



### 1. The multiple solver libraries of ROS

We have lots of solver libraries in ROS which can be found when we are using the Setup Assistant. The most typical one that we are using is the KDL, which is based on the numerical solution. However, in my opinion, UR5 in fact uses analytical solution. (To draw this conclusion, we can open the file in the package called “universal\_robot” and the subdirectory called “ur\_kinematics”, and then check the inverse() function defined in the ur\_kin.cpp) We only use something useful provided by KDL such as the KDL tree and some defined classes within KDL.

### 2.The usage of ur\_kin.cpp

Although it contains not very much codes, it is the key of solution-finding while we are using UR5. It defines these important functions: forward(), forward\_all() ,inverse(), ComputeFK() and ComputeIK(). Also, some other funcions like to\_mat44() and from\_mat44() are also defined, but since they are closely related to IKfast method, which goes beyond my ability, I will not mention them in details.

forward():

Given the six joints' value, compute the transfer matrix T6\_0(homogeneous transfer matrix) accordingly.

forward\_all():

Given the six joints' value, compute six transfer matrices T1\_0 T2\_0 T3\_0 ...(homogeneous transfer matrix) accordingly.

inverse():

Given the homogeneous transfer matrix, compute six joints' value accordingly. However, there are eight different set of joints and they are placed in a 8\*6 array.

ComputeFK() and ComputeIK():

They belongs to IKfast\_API, with the help of to\_mat44() and from\_mat44(), combining the forward() and inverse(), compute the solutions for later use.

### 3. The usage of ur\_moveit\_plugin.cpp

In a very abstract description, it combines the ur\_kin.cpp and the multiple functions provided by KDL together and calculate the solution. However, it remains unknown that who uses it, in what kind of ways, and the dependences between other files and it.

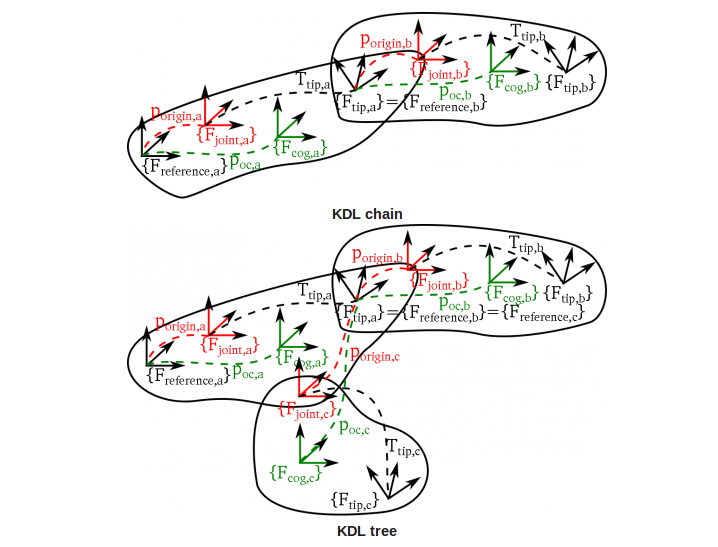
It imports some classes and functions provides by KDL like KDL::JntArray(which is a data type of vector defined by KDL), and the KDL tree(describe the connection between joints on the code level).

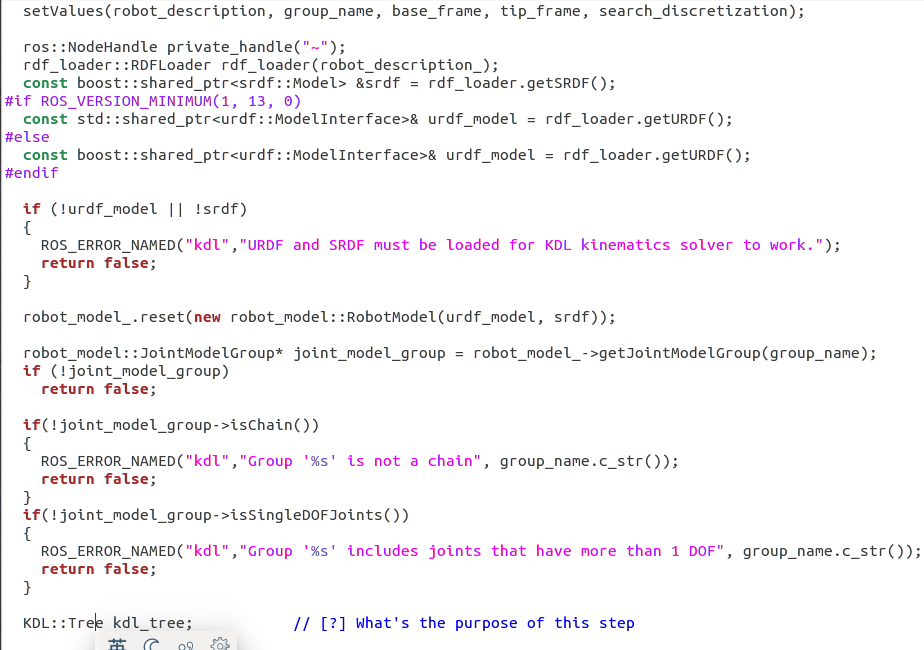
The way it defines functions is to put them into a namespace called ur\_kinematics, and defines a class called URKinematicsPlugin. Although there are lots of functions in this files, three of them are essentially important:

URKinematicsPlugin::initialize()

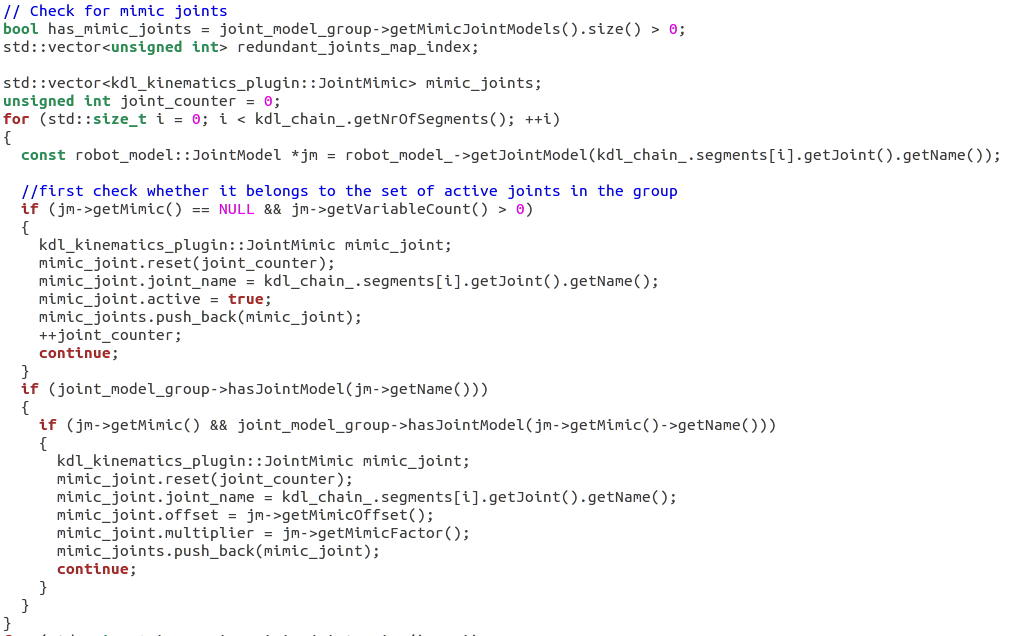
It performs these functionality:

1. Transform the robot model described in URDF and SRDF into a code-describing level using KDL::Tree.

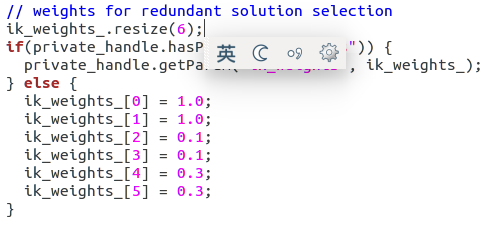




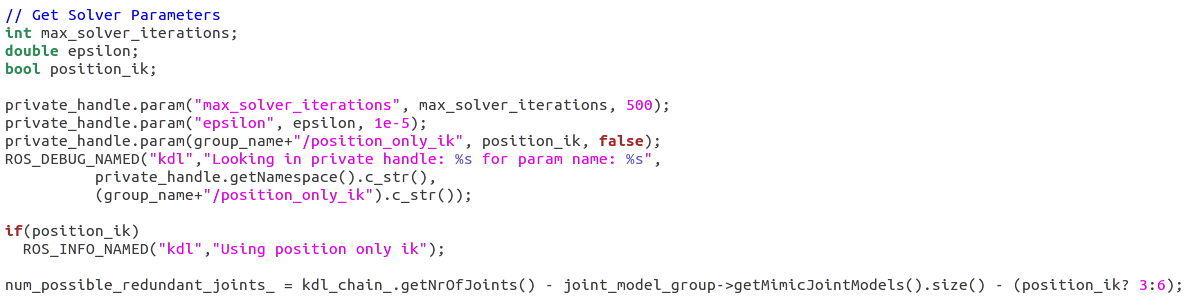
b. Defines the redundant joints that helps to simplifies the process of finding the solutions.



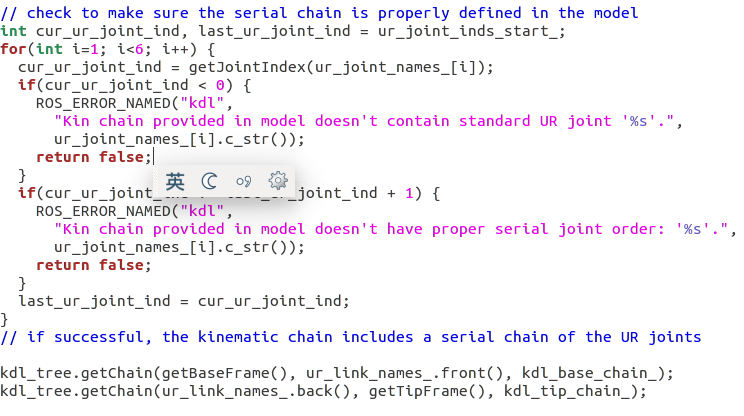
c. Defines the weights for redundant solution selection(for later solution-selecting)



d. Get solver parameters defined by the user



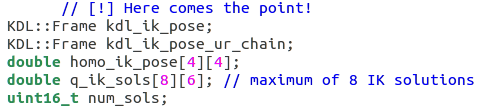
e. Check to make sure the serial chain is properly defined in the model and create the kdl\_tree

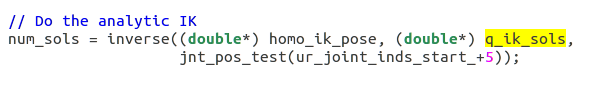


URKinematicsPlugin::searchPositionIK():

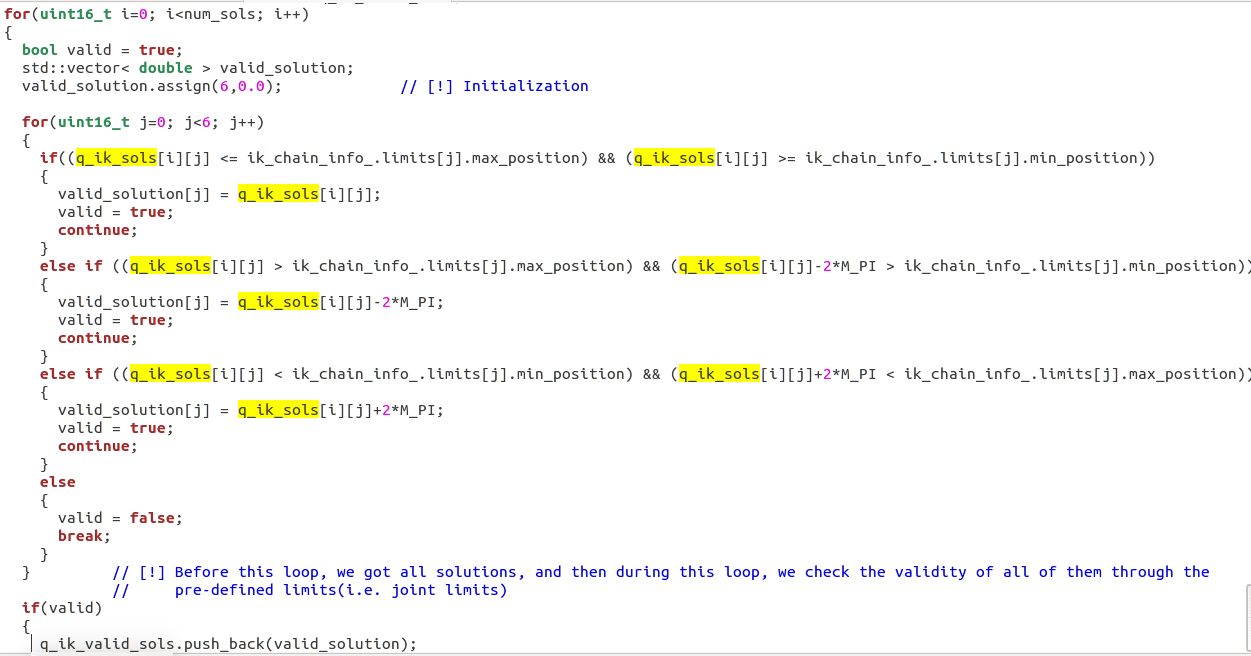
It is the key function to compute the inverse kinematics in this file. Let's break it down into several pieces:

a. compute the joints's value using inverse() defined in ur\_kin.cpp

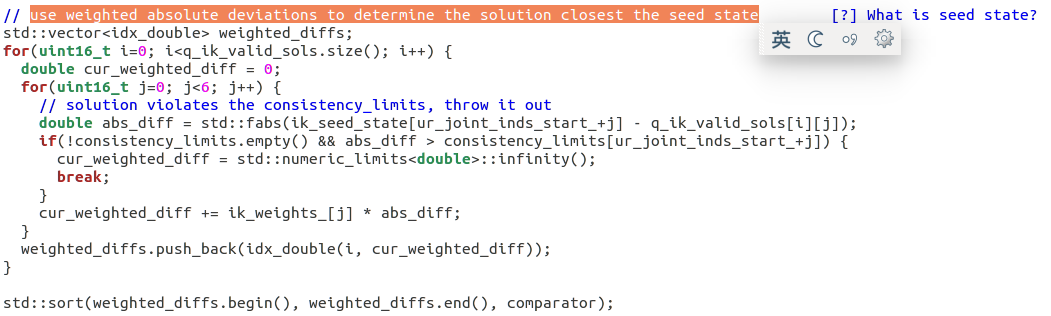




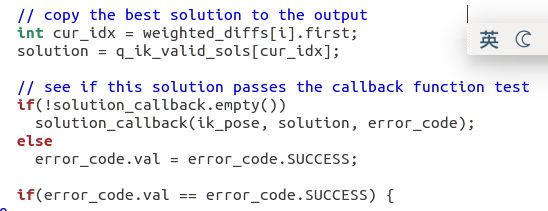
b. Filter\_1 ---> through the pre-defined joints's constrains



c. Filter\_2 ---> use weighted absolute deviations to determine the solution closest the seed state

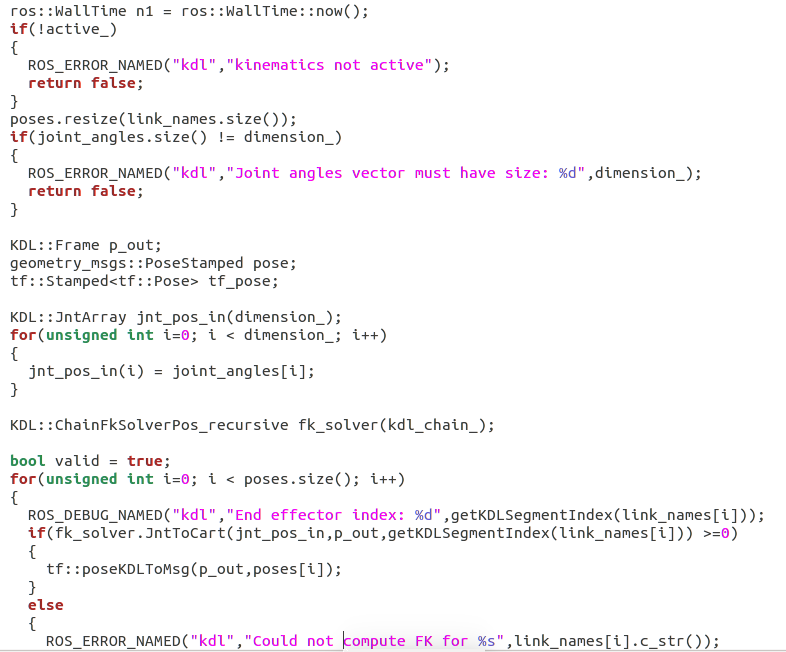


d. Filter\_3 ---> use the callback function(defined by user?) to test the validity of the solution



URKinematicsPlugin::getPositionFK():

Use fk\_solver() to transform the joints' value into a pose, and then publish it.

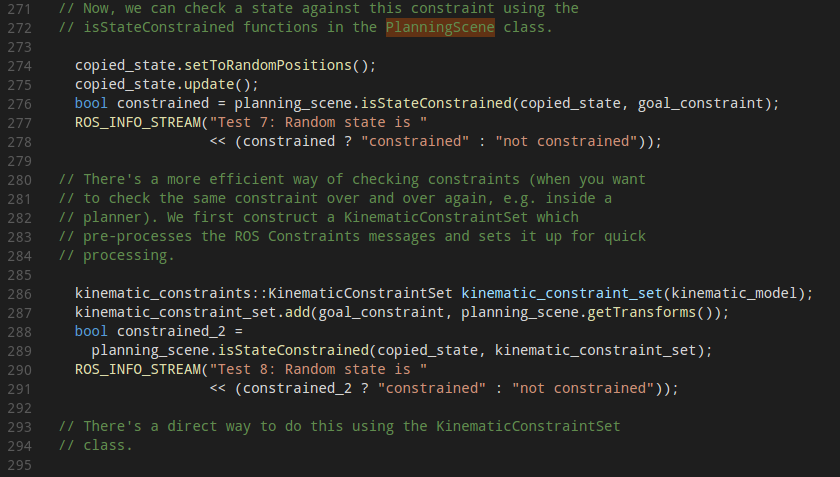


### 4. Other closely related files or class if we want to set constrains on our robot

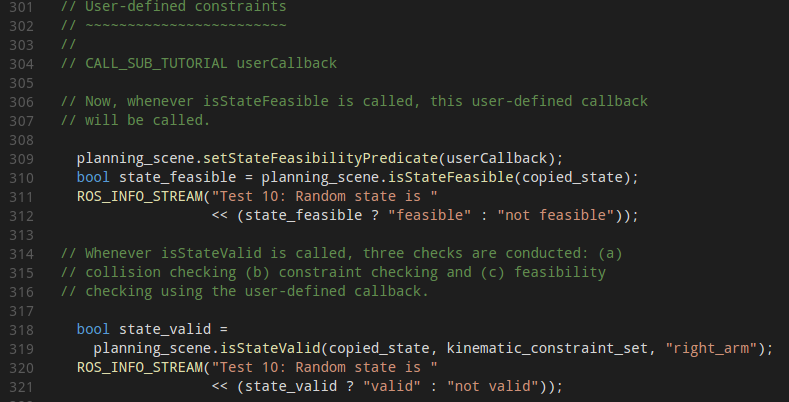
Closely related file: planning\_scene\_tutorial.cpp

Description:

In this file, it instantiate a special class planning\_scene::PlanningScene, which contains several key functions to check self-collision and other types of collisions.



What's more, it uses callback function to check whether a solution satisfies user's expectation.



## [Summary]

That's all I can uncover until now, it takes some more time to go deeper into the mechanism of the solution-selecting. Especially, it is a good idea to browse the ROS wiki to find some more details in areas like the usage of callback function and how can ROS connect multiple files together using nodes and msgs.