Project: Kinematics Pick & Place

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Kinematic Analysis

1. Run the forward_kinematics demo and evaluate the kr210.urdf.xacro file to perform kinematic analysis of Kuka KR210 robot and derive its DH parameters.

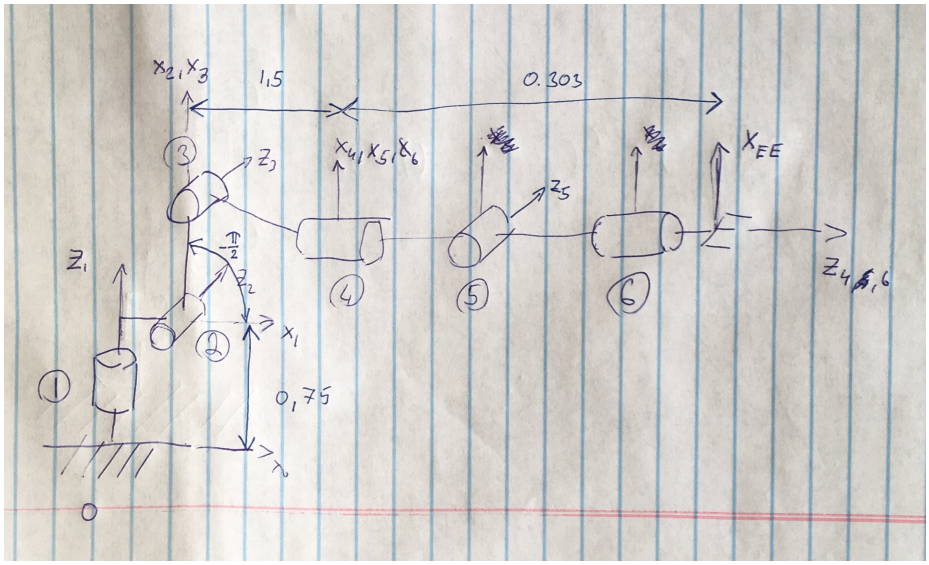
During this part we derived description of the robot (kinematic chain). The reason we us DH is that it is a system that ensures that if people use it the same way they can easily exchange information about robots etc. Also it simplifies the implementation since the transformation from DH table to transformation matrices is very simple.

Here is a DH parameters table.

Links	alpha(i-1)	a(i-1)	d(i-1)	theta(i)
0->1	0	0	0.75	q1
1->2	- pi/2	0.35	0	-pi/2 + q2
2->3	0	1.25	0	q3
3->4	-pi/2	-0.054	1.5	q4
4->5	pi/2	0	0	q5
5->6	-pi/2	0	0	q6
6->EE	0	0	0.303	0

Let's walk through derivation briefly

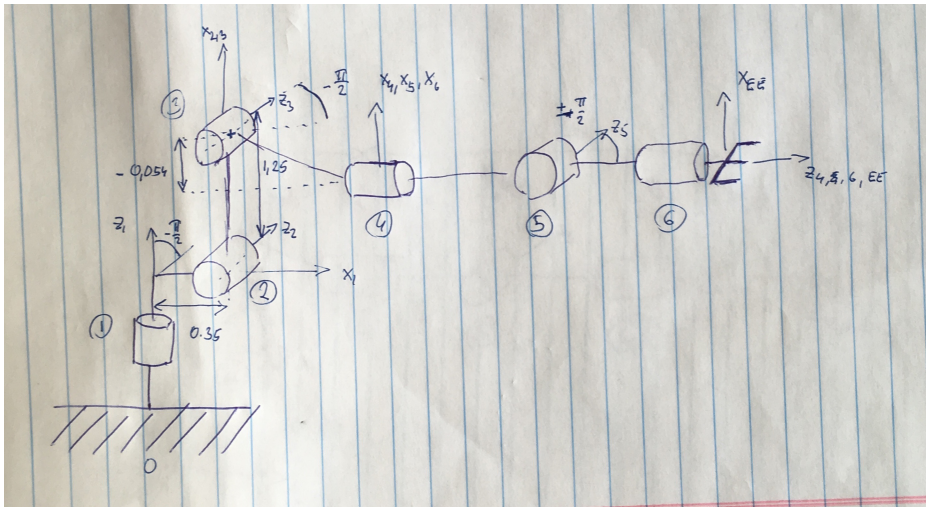
Here is pictures of the kinematic joints. Important is to pick z and x axes for the joints.



In this first picture I highlighted d and theta parameters.

d is defined as distance between neigbouring x axes (x_i and x_i+1) along the z_i+1 axes. In the picture are 3 instances. The values can be found in the xacro files. Note that although joints 4,5,6 are drawn far apart their x axes are actually coincident. This simplifies some values + it actually fits the physical implementation of the wrist.

theta are defined as angle between x_i and x_i+1 along z_i+1 in the right hand sense. Since all joints are revolute this parameters are all variables but we need to augment one angle by a value -pi/2 for it to work (between x2,x3).



In this picture I did the same for the parameters alpha and a.

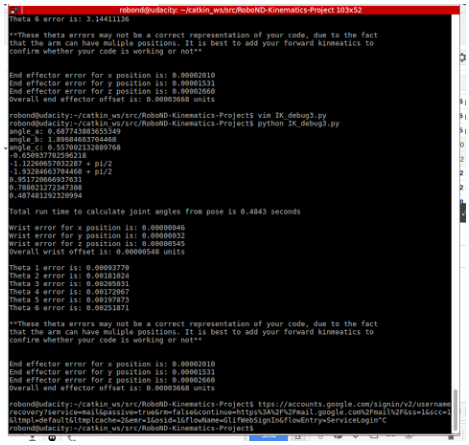
a is defined as distance between z_i and z_i+1 along x_i. There are three nnonzera places marked in the picture. Note maybe the -0.054. It is zero since it goes down from 3 to 4.

The alphas are defined as angle between z_i and z_i+1 about x_i again in the right hand sense. There is a lot of these values in the picture since we have adjacent joints usually rotated by 90 degrees. I marked sevearl values in the picture. Notice z4-z5 and z5-z6. They are marked at the same spot but it has different signs.

What is interesting about this particular setup is that there is many parallel/collinear links which significantly simplifies derivation of parameters. Also the wrist center (4-6) is in one place again causing a lot of parameters to be zero.

Derivation of DH parameters is not difficult but definitely requires practice to be able to perform it without errors. One place where I struggled in this part was the selection of axis between points 1-2 which leads to additional pi/2. Only later I realized that I did not follow the DH rules properly and this orientation of this axis is actually forced by definition based on selection of previous axes.

After acquainting ourself with the environment I derived the DH parameters and implemented forward and backward kinematics using the IK. Here is the output of the debug script which was essential to make it work.



2. Using the DH parameter table you derived earlier, create individual transformation matrices about each joint. In addition, also generate a generalized homogeneous transform between base_link and gripper_link using only end-effector(gripper) pose.

Benefit of DH parameters is that it simplifies generation of transformation matrices since they can be constructed automatically as seen in the code. By wrapping the matrix generation into a function we can easily write something like

```
def ht_matrix_from_dh(alpha, a, d, q):
  return Matrix([[cos(q),               -sin(q),           0,           a],
                 [sin(q) * cos(alpha), cos(q)*cos(alpha), -sin(alpha), -sin(alpha) * d],
                 [sin(q) * sin(alpha), cos(q)*sin(alpha),  cos(alpha),  cos(alpha) * d],
                 [0                  , 0                , 0          ,  1             ]])
```

We then can generate individual transformations directly from DH table

```
T0_1 = ht_matrix_from_dh(alpha0, a0, d1, q1).subs(dh_parameters)
..
..
T6_EE
```

And then we can create transformation between arbitrary points.

```
T0_EE = T0_1 * T1_2 .... * T6_EE
```

3. Decouple Inverse Kinematics problem into Inverse Position Kinematics and inverse Orientation Kinematics; doing so derive the equations to calculate all individual joint angles.

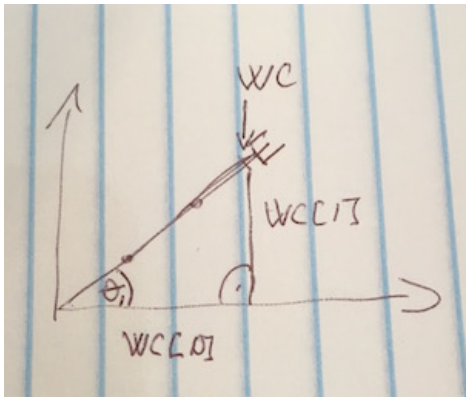
Outline of solution

The solution goes like this.

- From the planner we receive the poses of the end effector
- Our robot falls into a wide category of robots whose IK can be solved geometrically. This has an advantage that the solver can be very fast since it is in closed form. But we have to find specific solution for this particular kimenatic chain
- We get position of the end effector
- From this we compute the position of wrist center
    - This is done by using inverse transformations from the forward kinematics part from wrist center to end effector
- If we know position of our wrist center we can solve theta 1-3 by applying some trigonometric functions
- Theta 4-6 are taken from relation of wrist center to end effector by extracting angles from the rotation matrix

Theta 1

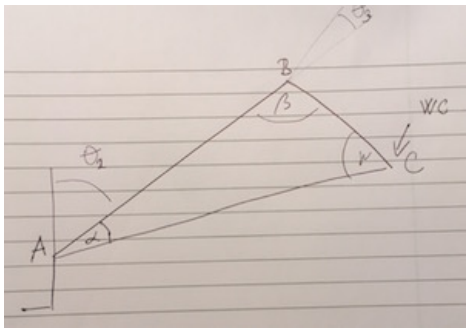
Seen from above we can simply calculate theta 1 by using atan2 directly from the known position of the wrist in x y plane.




```
theta_1 = atan2(wc[0], wc[1])
```

Triangle ABC

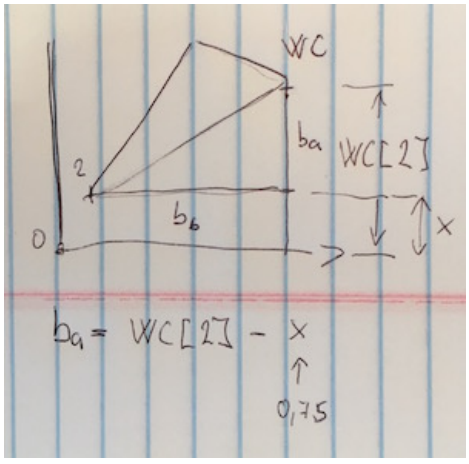
Both theta 2 and theta 3 can be calculated from a triangle A (joint 2), B(joint 3), C (wrist center).



First we calculate the sides of a triangle and then use cosine sentences to get the corresponding angles.

Side b

First let's grab side b. It can be calculated by pythagorean theorem from sides b_b and b_a on the following picture.



b_a we get almost directly from the position of wrist center. **b_b** can be calculated by another application of pythagoreas from x and y position of wrist center. Both need to be accounted for the fact that they are measured from point 2 and not the origin.

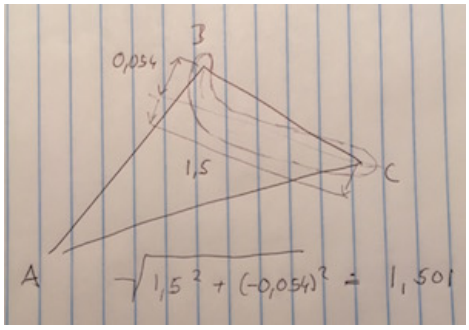
Side c

Side C can be used directly from the DH params or xarco file.

Side a

Just taking the side A from DH parameters directly yields decent results but we are introducing systematic error. I noticed after looking at the provided solution. It works roughly like this.

The error comes from the fact that the link represented by c is not a straight link but has a curve or "sag" to it. The corrected length of the side c is simply acquired by computing the hypotenuse of that link for the DH parameters (the sag in the following picture is really exagerrated).



The second correction comes to the angle. And is acquired by for example atan2 from the sides.

```
theta_2 = atan2(-0.054, 1.5) = -0.036
```

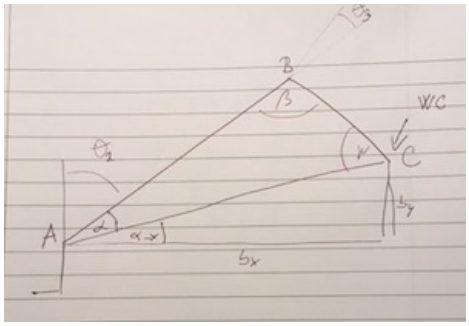
Theta 2

When we have all the pieces we can simply get theta

```
theta_2 = pi/2 - angle_a - angle_a_x
```

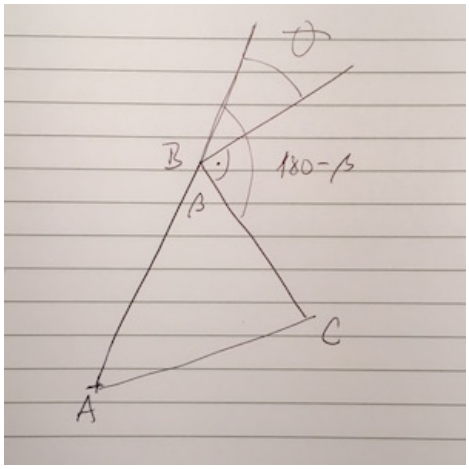
where angle_a_x is acquired by

```
angle_a_x = atan2(b_y, b_x)
```

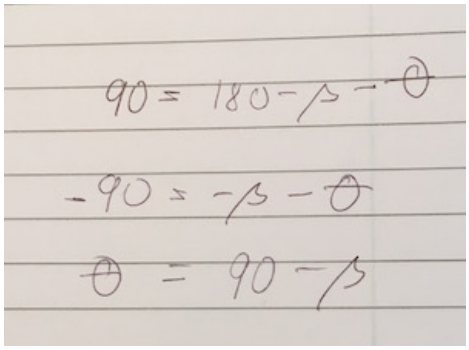


Theta 3

Theta 3 can be coputed by realizing that Theta 3 and **reflected beta** has to equal `pi/2`



This can be expressed in equation as

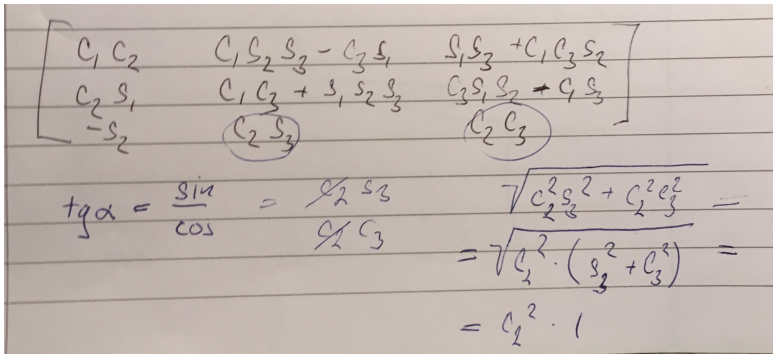


Here in the derivation I omit the small correction of the beta angle discussed before caused by the sag of the link.

Theta 4-6

From known rotation matrix we want to extract euler's angles. This is done by 2 tricks. Generally we want to use atans as much as possible due to unambiguity compared to arcsin.

The trigonometric tricks are illustrated in the following image.



One is leveraging the identity

```
tg(3) = sin(3)/cos(3) = (c2 s3)/(c2 c3)
```

The second one is using the idea

```
cos(x) = sqrt(cos^2(x)sin^2(y) + cos^2(x)cos^2(y))
       = sqrt(cos^2(x) * (sin^2(y) + cos^2(y))

using substitution (sin^2(y) + cos^2(y)) = 1

       = sqrt(cos^2(x) * 1)
       = cos(x)

and similar to the above can write

tg(2) = sin(2)/ sqrt(cos^2(x)sin^2(y) + cos^2(x)cos^2(y))
```

Here the example uses Z Y X matrix. The real code actually uses R3_EE. First we calculate

```
T3_EE = T3_4 * T4_5 * T5_6 * T6_EE
```

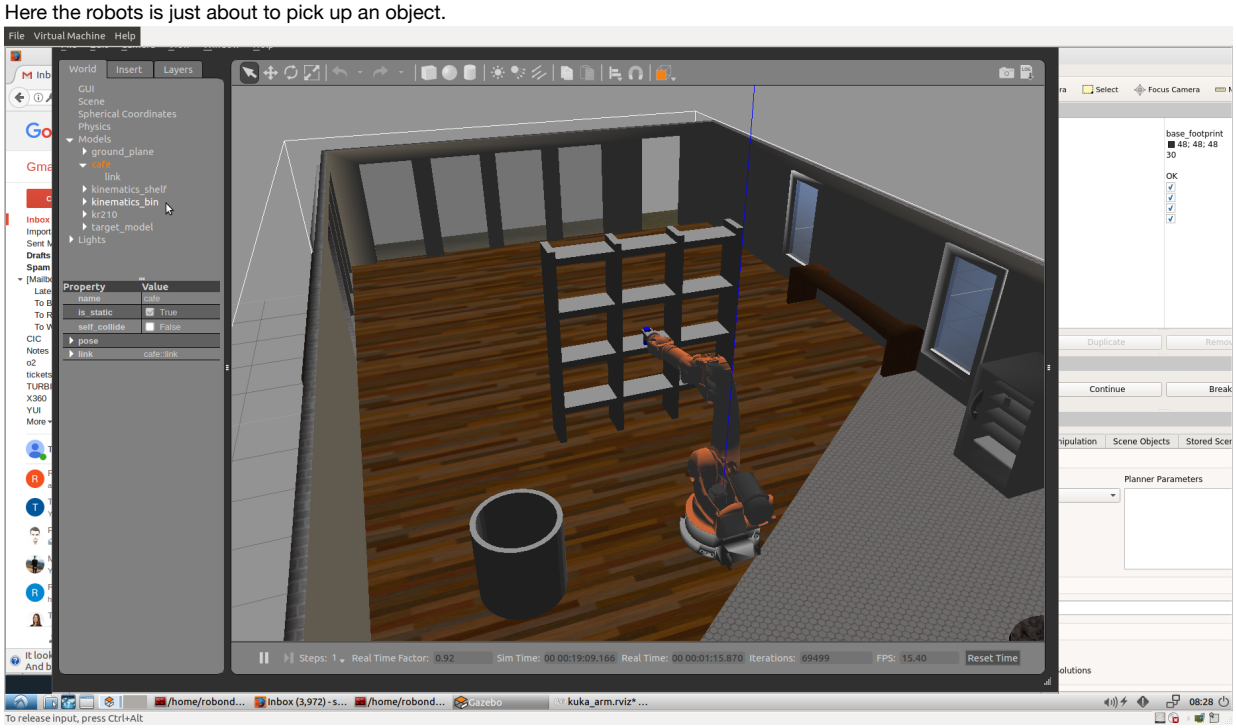
Then we turn into rotational by croppping and use sympy to simplify. Then we apply exactly the same approach as above.

Project Implementation

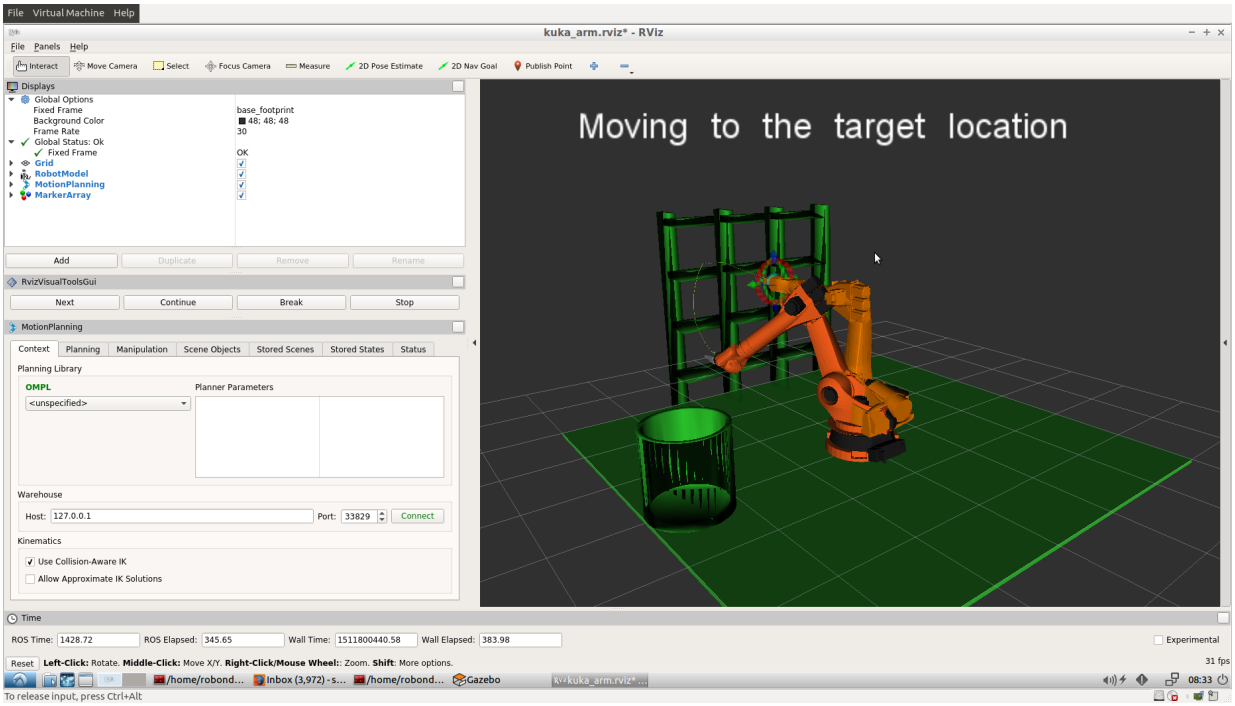
1. Fill in the `IK_server.py` file with properly commented python code for calculating Inverse Kinematics based on previously performed Kinematic Analysis. Your code must guide the robot to successfully complete 8/10 pick and place cycles. Briefly discuss the code you implemented and your results.

The code seem to perform well after several iterations. Here are a couple of picture to illustrate it in action.

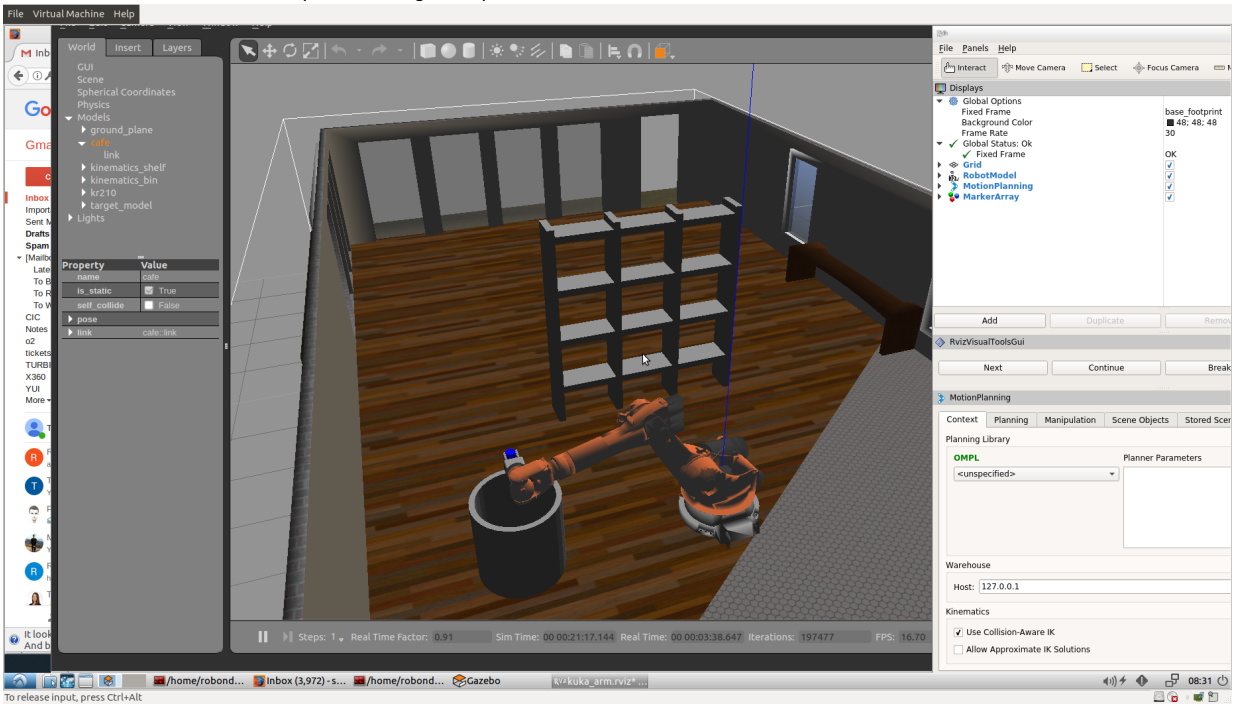
Here the robots is just about to pick up an object.

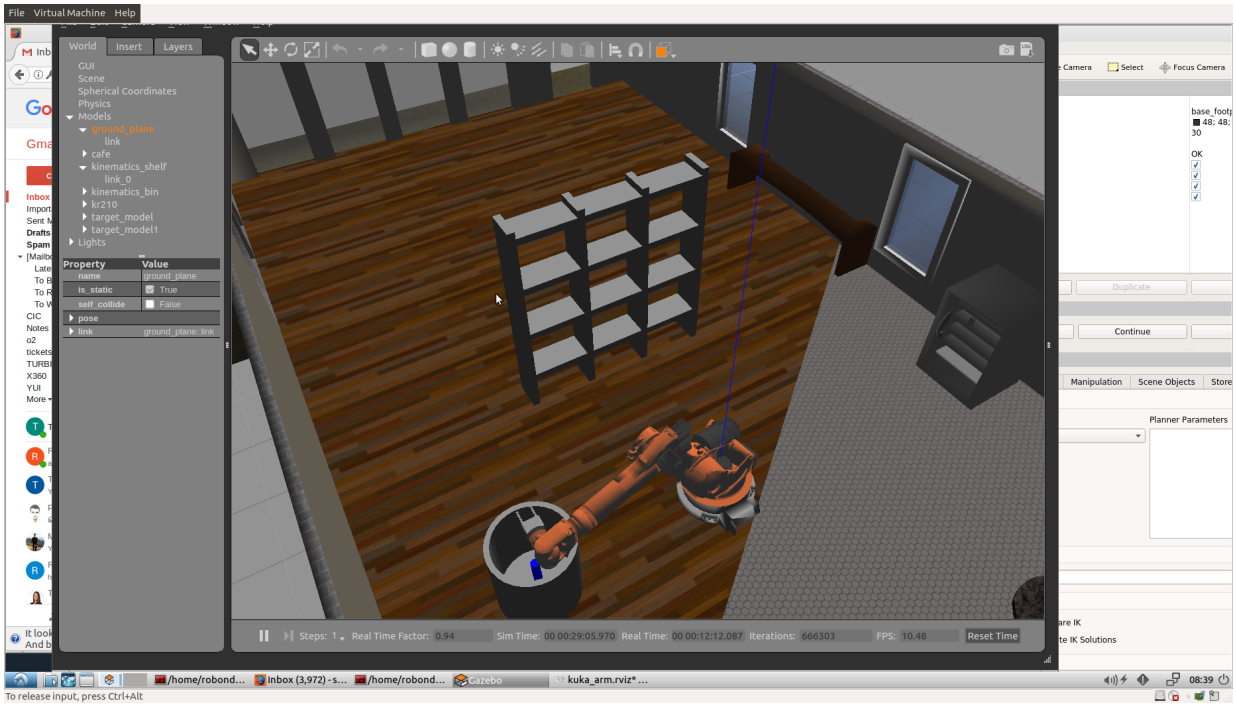


Here it is following the path planned by planner



And here it is before and after drop to the designated place





Video is included in the repo as well in file pick_cycle.mp4

Implementation difficulties

The implementation is not that difficult codewise if a person understands the underlying math which at times is a bit difficult since a lot of things are going on. I had the following issues.

Inversion stability

During the implementation I had one major issue and that was numerical stability of the implementations of inversion of matrices. I tried several implementations (LU decomposition, inversion, transposition) with different results on different computers. Unfortunately the notes are misleading here. During ~3 days of debugging I switched portions of the implementation for the walkthrough one and it still gave weird results until I switched the inverse for the transpose. Potentially something that might be improved in the future.

How it manifested in the project is that the end effector was spinning wildly. Since it picked cylinders occasionally and followed the path more or less I was not sure what is going on. Then I saw somebody's video posted on slack which did not have this behavior. By experimentation I was able to narrow it down to the inversion.

Cosine sentences

I had an issue with cosine sentences. I coded them wrong because of a typo and a complex number was introduced. This coupled with sympy simplification my IK_debug just hung up and never finished so I was not sure what is going on. Again I was not sure if this is a problem of my computer or maybe issue connected to the VM but I was able to debug the problem in the cosine sentences eventually and got rid of some more expensive sympy operations pushing the debug execution run below a second or two.

Theta 4-6

For this part I had to consult the walkthrough. I understood process of extracting the angles from typical matrices but took me a while to realize where the elements are coming from in this particular implementation and why. I saw a comment in the slack giving some tips so after then it was fairly easy to use sympy to come up with the matrix and apply the "tricks" described above to extract the angles.

Areas for improvements

Since this project is fairly big the areas are not so much improvements but more areas for further exploration. - I would definitely like to understand the whole chain. Especially how the planner works since sometimes it does not feel like the path planned is optimal - ROS + gazebo seems like a very powerful tool

Attributions

- Portions of the code are used from the project walkthrough