GrabPoseCalculator

For calculating the grabpose the output of find_object ROS service is used. The available data contains two arrays, one with the location (x, y, z), and one with the rotation in quaternions (x, y, z, w). the instance of the object is also known.

In short, the gripper will be placed a pre-set distance away from the object (“self.approach_distance”). This pose will be in line with the grab pose itself. There will always be many possible grab poses, the program finds the most vertical among of them. These functions can be found in bin_picking_sequencer.py.

The most vertical vector is chosen so the gripper doesn’t bump into the walls of the bin. To find this vector the rotated x, y, z vectors (as seen in red, green and blue respectively) are interpreted from the quaternion rotation. This translates the unit vectors of the object with respect to its base origin to the respect of the world (Figure 2)

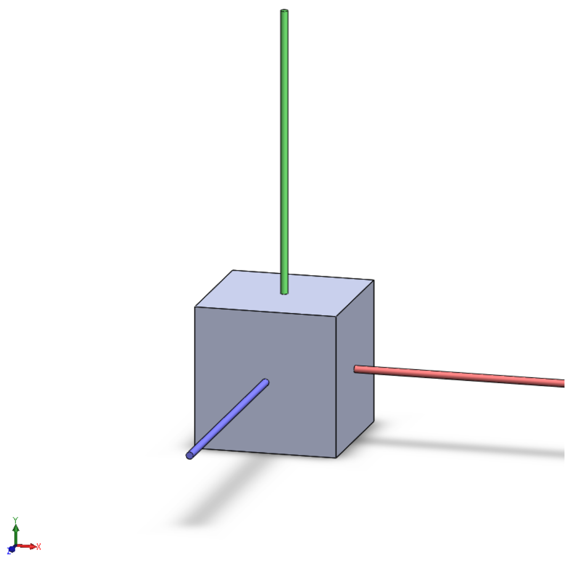

Figure 1, vectors match origin

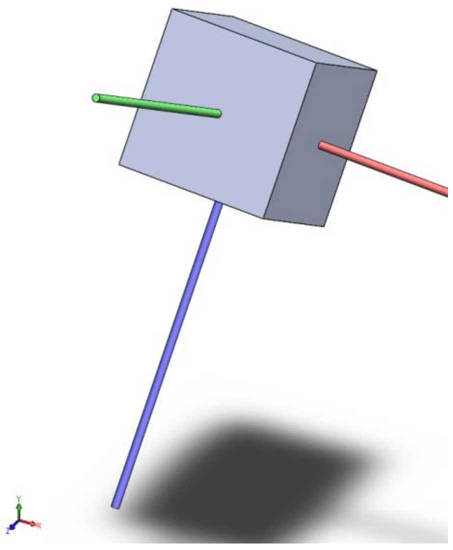

Figure 2, vectors rotated with respect to origin

These vectors are negated if pointed down and compared to each other to find the vector that’s pointing up the most. In case of Figure 2 that would be the negative blue vector. This vector is then used to calculate a point a set distance above the object in line with this vector. The gripper will go to this approach point first and move along the vector to the object. In case of cylindrical objects there will be more than 6 vectors to chose from. How this is handled is explained in GetGraspPoseUsingRodrigues.

DeterminePoses

```
89. def determinePoses(self, object_pose, object_type):
90.        """
91.        Calculate the approach pose and the grasp pose for an object
92.        object_pose -- geometry_msgs/PoseStamped object pose
93.        object_type -- string, object name
94.        returns -
    - approach pose (geometry_msgs/PoseStamped), grasp pose (geometry_msgs/PoseStamped)
95.        """
96.        self.publishPose(object_pose.pose, self.objectPosePublisher)
97.
98.        quaternion_object = Quaternion(
99.               object_pose.pose.orientation.w,
100.              object_pose.pose.orientation.x,
101.              object_pose.pose.orientation.y,
102.              object_pose.pose.orientation.z
103.          )
104.
105.          # Get the approach unit vector and pose rotation
106.          approach, rotation = self.approachCalculator(quaternion_object, object_type)

107.
108.          # Calculate the approach position:
109.          # approach * approach distance + object position
110.          approach_position = np.array(approach) * self.approach_distance + np.array([

111.              object_pose.pose.position.x,
112.              object_pose.pose.position.y,
113.              object_pose.pose.position.z
114.          ])
115.
116.          # Create Pose message for approach
117.          approach_message = self.makePoseMessage(approach_position, rotation)
118.
119.          # Adjust grasp orientation
120.          object_pose.pose.orientation = approach_message.orientation
121.
122.          self.publishPose(approach_message, self.approachPosePublisher)
123.          self.publishPose(object_pose.pose, self.graspPosePublisher)
124.
125.          return approach_message, object_pose
```

in line 98 the rotation is defined as a quaternion. This ensures the pyquaternion library can use it. Next the approach is calculated in “approachCaclulator.

ApproachCalculator

```
129.    def approachCalculator(self, quaternion_object, object_type):
130.            """
131.            Calculate the approach unit vector (grasp diretion) and grasp orientation
132.            quaternion_object -- Quaternion of the rotation of the object
133.            object_type -- Name of the object
134.            return -- Unit vector (array), grasp orientation (Quaternion)
135.            """
136.            rotated_x = np.array(quaternion_object.rotate([1, 0, 0]))
137.            rotated_y = np.array(quaternion_object.rotate([0, 1, 0]))
138.            rotated_z = np.array(quaternion_object.rotate([0, 0, 1]))
139.
140.            preffered_rotation, rod_vector, cylindrical_axis = self.getPrefferedRotation(object_type, rotated_x, rotated_y, rotated_z)
141.
142.            if cylindrical_axis is not None and abs(preffered_rotation[2]) < self.z_limit:
143.                return self.getGraspPoseUsingRodrigues(preffered_rotation, rod_vector)
144.            else:
145.                return self.getGraspPoseUsingAxis(rotated_x, rotated_y, rotated_z)
```

In line 136-138 the quaternion rotation is applied to the three unit vectors x, y and z. for example, the rotated_x value will contain an array [x,y,z] which describes where the x axle of the object is pointed, with reference to the base of the robot. This is done to make the rotation data easier to compute with.

The program treats cylindrical objects different than cubic objects. In the config file each object can be assigned a cylindrical axis if it has one. This data is obtained with GetPreferredRotation in line 140.

GetPreferredRotation

```
    def getPrefferedRotation(self, object_type, rotated_x, rotated_y, rotated_z):
            """
            Get the preffered rotation axis. This is determined by the config
            object_type -- name of the object
            rotated_x -- X rotation vector
            rotated_y -- Y rotation vector
            rotated_z -- Z rotation vector
            returns -- preffered rotation vector, perpendicular vector, cylindrical axis
            """
            if object_type in self.pick_up_config:
                cylindrical_axis = self.pick_up_config[object_type]
            else:
                cylindrical_axis = None

            preffered_rotation = rotated_x
            rod_vector = rotated_z

            if cylindrical_axis == 'y':
                preffered_rotation = rotated_y
                rod_vector = rotated_x
            elif cylindrical_axis == 'z':
                preffered_rotation = rotated_z
                rod_vector = rotated_y

            return preffered_rotation, rod_vector, cylindrical_axis
```

This function assigns the cylindrical_axis from the config file and rotates the preferred_rotation and rod_vector accordingly. These axis are dependent on how the object is drawn with respect to its own origin

When the object does have a cylindrical axis defined, line 142 tests if the absolute vertical component of this vector is higher than self.z_limit (in this case 0.8). Because the vector will always be length one, this will be the case if the cylinder is pointing straight up until ~35° from the Z axis (**Error! Reference source not found.**). When this is the case the object will be treated as a cubic object and use the function on line 145. We will first focus on an object with a cylindrical axis and self.z_limit < 0.8. The program will use the function on line 143.

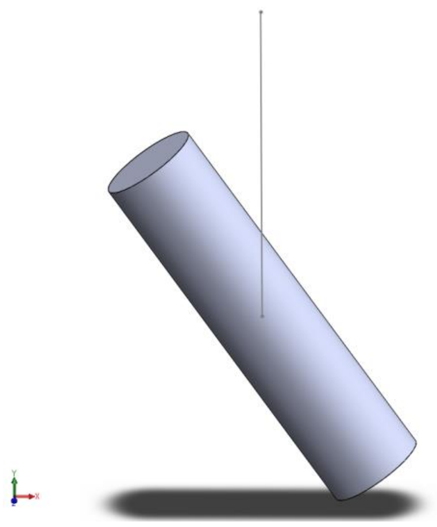
Figure 3, cylinder at 35 degree from Z axis

GetGraspPoseUsingRodrigues

```
193.    def getGraspPoseUsingRodrigues(self, preffered_rotation, rod_vector):
194.          """
195.          Get the grasping position using rodrigues rotation.
196.          Only suitable for cyclindrical objects. The highest point above the object is used
197.          as the grasping approach.
198.          preffered_rotation -- The axis to rotate around
199.          rod_vector -- Axis perpendicular to preffered_rotation
200.          returns -- Unit vector, grasp orientation
201.          """
202.          rospy.loginfo('Determining grasping poses using rodrigues rotation')
203.
204.          rodr = self.rodriguesRotation(rod_vector, preffered_rotation)
205.
206.          if preffered_rotation[2] < 0:
207.              preffered_rotation = -1 * preffered_rotation
208.
209.          array = np.zeros([3, 3])
210.          array[0] = -rodr
211.          array[1] = np.cross(rodr, preffered_rotation)
212.          array[2] = preffered_rotation
213.
214.          array = np.rot90(np.fliplr(array))
215.          quad = Quaternion(matrix=array)
216.
217.          return rodr, quad
```

When the cylinder lies ‘flat’ the object has to be picked up from the side. This can still be treated as a cubic object, but because the object has a round surface, infinitely valid grasps can be made in between the 90° cubic axis. A circle will be drawn from the midpoint of the object as seen in Figure 4. The vector from the centre to the highest point on the circle will be used as the approach vector. This is calculated in the rodriguesRotation function in line 204.

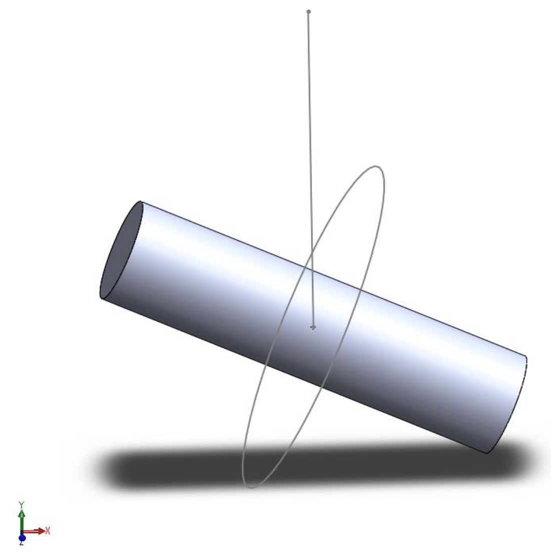

Figure 4, cylindrical object at 65 degree from Z axis

RodriguesRotation

```
268.   def rodriguesRotation(self, v, k):
269.       """
270.       Rotates vector v about unitvector k according to Rodrigues' rotation formula and
    returns highest point
271.       """
272.       temp = np.zeros(shape=(100,3))
273.       for i in range(1,100):
274.           temp[i] = v * math.cos(2*math.pi/i) + np.cross(k,v) * math.sin(2*math.pi/i)
   + k * np.dot(k,v) * (1-math.cos(2*math.pi/i))
275.       comp = np.zeros(shape=(3))
276.
277.       # Finds vector with highest z component
278.       for r in range(1,100):
279.           if temp[r-1][2] > comp[2]:
280.               comp = temp[r-1]
281.
282.       return comp
```

the function is based on Rodrigues' rotation formula:

$$v_{rot} = v\cos\theta + (k \times v)\, sin\theta + k(k \cdot v)(1 - cos\theta)$$

Which rotates a known vector v about a unit vector k with angle θ.

In line 272-275 a vector is rotated about the cylindrical axis in 100 steps of 100th of a full rotation. In line 278-280 the most vertical of these vectors is found by finding the object with the highest z component.

This vector is placed in a rotation matrix (line 209-212), along with the preferred rotation. Note that this preferred rotation is flipped before it is placed in the matrix in line 206 if it is pointing down. A unit vector perpendicular to these vectors is calculated using the np.cross function and used as the last vector in the rotation matrix.

In line 214 the array is mirrored diagonally so the pyquaternion can use it. In line 215 the array is converted to a quaternion rotation.

The function returns the vector rodr, which is a unitvector pointing from the centre of the object in line with the path the gripper has to take. This is used in line 110. The vector is multiplied by self.approach_distance (in this case 0.3m) and added to the object position. This creates a location 30cm above the object. This will be the first position the gripper moves to. The object location being the second.

For cubic objects, or cylindrical object standing up (Figure 3, cylinder at 35 degree from Z axisFigure 3) an equivalent of the rodr vector can be found. This is done by the getGraspPoseUsingAxis function on line 145.

getGraspPoseUsingAxis

```
153.    def getGraspPoseUsingAxis(self, rotated_x, rotated_y, rotated_z):
154.            """
155.            Get the grasping position using the axis of the grasping object.
156.            The axis of the object that is the highest is the approach axis for grasping.

157.            rotated_x -- X rotation vector
158.            rotated_y -- Y rotation vector
159.            rotated_z -- Z rotation vector
160.            return Unit vector, grasp orientation
161.            """
162.            rospy.loginfo('Determining grasping poses using axis rotation')
163.
164.            array = np.zeros([3, 3])
165.            array[0] = rotated_x
166.            array[1] = rotated_y
167.            array[2] = rotated_z
168.
169.            if abs(array[0][2]) < abs(rotated_y[2]):
170.                array[0] = rotated_y
171.                array[1] = rotated_z
172.                array[2] = rotated_x
173.            elif abs(array[0][2]) < abs(rotated_z[2]):
174.                array[0] = rotated_z
175.                array[1] = rotated_x
176.                array[2] = rotated_y
177.
178.            pos = array[0]
179.
180.            if pos[2] < 0:
181.                pos = -pos
182.            else:
183.                array[0] = -array[0]
184.                array[1] = -array[1]
185.
186.                if array[0][2] < 0:
187.                    pos = -pos
188.
189.            array = np.rot90(np.fliplr(array))
190.            quad = Quaternion(matrix=array)
191.
192.            return pos, quad
```

To find the approach vector the rotated unit vectors are placed in an rotation matrix (line164-167). In line 169-173 the most vertical vector among the three will be found. This is done by searching the greatest |z| component of the rotated x, y and z. the most vertical vector will be placed in array[0]. The order of array[1] and array[2] are set and negated accordingly so the determinant of the array will be 1. This has to be 1 for the conversion to quaternions.

In line 180 the most vertical vector will be negated if it is pointing down, so the robot doesn't approach it from below.

In line 189 the array is mirrored diagonally so the pyquaternion can use it. In line 190 the array is converted to a quaternion rotation.

The function returns (pos, quad). pos is the most vertical vector pointing up, quad is the rotation the gripper has to assume.

The vector pos is used exactly the same as the vector rodr in line 110.

Finally the data will be placed in a message in line 117 using the makePoseMessage function.

MakePoseMessage

```
250.    def makePoseMessage(self, position, orientation):
251.        """
252.        Create a geometry_msgs/Pose message from coords and Quaternion
253.        position -- Array with the x, y, z coordinates ([x, y, z])
254.        orientation -- Quaternion object
255.        """
256.        message = geometry_msgs.msg.Pose()
257.        message.position.x = position[0]
258.        message.position.y = position[1]
259.        message.position.z = position[2]
260.        message.orientation.x = orientation[1]
261.        message.orientation.y = orientation[2]
262.        message.orientation.z = orientation[3]
263.        message.orientation.w = orientation[0]
264.
265.        return message
```

This function only formats the data so it can be used by moveit.