Jaco for smart pick ’n’ place

Smart Robotics Project

Eros Bignardi matr. 162415
Alessandro Castellucci matr. 162423

The idea

Object **quality check** for small industrial contexts: shape and mass evaluation of small objects with **motion planning** and **collision avoidance**.

Main goal: our robot should be aware of the context in which it operates and of the objects that it handles to perform its industrial work.

Key parts:

- computer vision algorithm for object shape detection
- mass evaluation by a force sensor
- motion path planning with collision avoidance

Robot choice: JACO

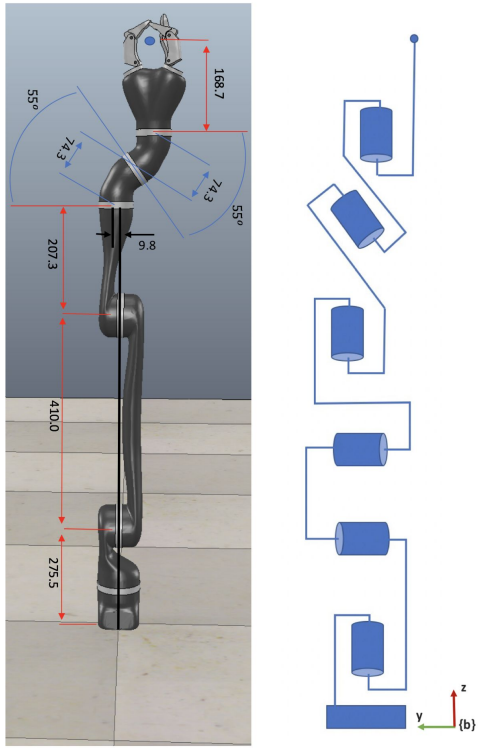
For our project we have chosen the JACO robot (6DOF) with its 3 fingers hand.

It's **lightweight**, **handy** and relatively **cheap**.

It's mainly used for the assistance to people with disabilities, **academic research** and small industrial contexts.

Being a 6DOF robot, it's able to pick small objects easily and with efficiency.

Overall weight: 5.4 kg
Continuous payload: 1.6 kg



Left: Physical dimensions of the Jaco arm. Right: Schematics of the Jaco arm. {b} is the frame of the base of the Jaco arm, which we define to be the world of the other links and joints.

Robot Environment and Simulation

The robotics simulator **CoppeliaSim**, with integrated development environment, is based on a **distributed control architecture**: each object/model can be individually controlled via an embedded script, a **plugin**, a ROS node, a **remote API client**, or a custom solution. This makes CoppeliaSim very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, **Python**, Java, **Lua**, Matlab or Octave.

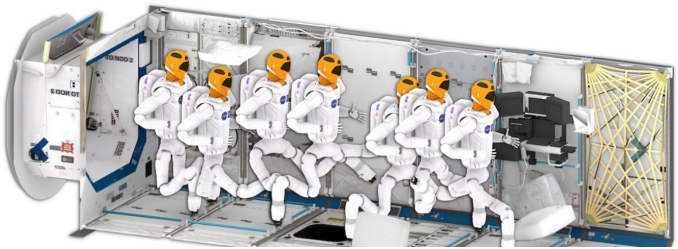
CoppeliaSim is used for fast algorithm development, factory automation simulations, fast prototyping and verification, robotics related education, remote monitoring, safety double-checking, as digital twin, and much more.

CoppeliaSim
from the creators of V-REP

BULLET PHYSICS LIBRARY

Lua

The Open Motion Planning Library



High Level Architecture

1 - **Object generation** on conveyor (cuboid or cylinder)

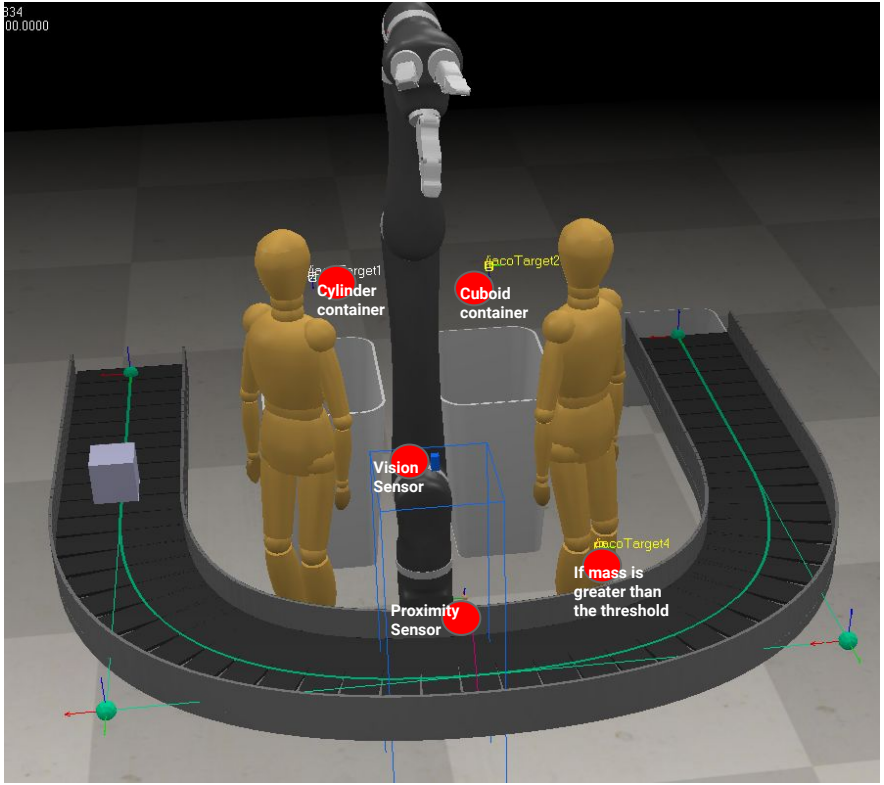
2 - **Proximity sensor** stops the conveyor when object is detected

3 - Fixed **vision sensor** → shape detection

4 - Jaco lifts the object and evaluates its mass by a **force sensor** between the arm and the hand

5 - Three possible destinations with **motion path planning** (collision avoidance) based on object shape and mass

6 - Cycle restarts



Proximity Sensor (ray type)

sim.readProximitySensor reads the state of a proximity sensor.

The conveyor remains off until the proximity sensor detects an object or when Jaco is performing the picking and placing operation.

When the sensor detects an object it stops the conveyor and calls the vision script and the main robot script through **signals communication**.

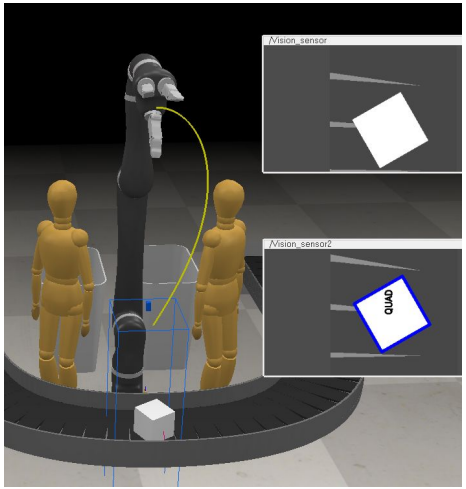
Lua return values	**result**: detection state (0 or 1), or -1 if sim.handleProximitySensor was never called, or if sim.resetProximitySensor was previously called. **distance**: distance to the detected point **detectedPoint**: table of 3 numbers indicating the relative coordinates of the detected point **detectedObjectHandle**: handle of the object that was detected **detectedSurfaceNormalVector**: normal vector (normalized) of the detected surface. Relative to the sensor reference frame.

```
Proximity_sensor=sim.getObject('./conveyor/Proximity_sensor')
conveyorHandle=sim.getObject('./conveyorSystem')
camera=sim.getObject('./Vision_sensor')
sim.setInt32Signal("jaco_grip", 0)

while true do
    result,_,_,objHandle,_ = sim.readProximitySensor(Proximity_sensor)
    if(result>0 or sim.getInt32Signal("jaco_grip")==1) then
        sim.writeCustomTableData(conveyorHandle,'__ctrl__',{vel=0})
        sim.setStringSignal("objHandle", objHandle)
        sim.setInt32Signal("jaco_grip",1)
        sim.wait(1.0)
    else
        sim.writeCustomTableData(conveyorHandle,'__ctrl__',{vel=0.05})
    end
end
```

Vision Script

The vision script has the task of detecting the object shape and transmits the information to the main robot script for the motion planning task. The images are acquired through an **orthogonal vision sensor** placed above the conveyor and the proximity sensor.




```python
def shapeDetection(resolution, image):
    shape = 'NONE'
    img = np.array(image).astype(np.uint8)
    img = img.reshape(resolution[0], resolution[1], 3).transpose([1, 0, 2])
    img_copy = img.copy()
    gray = cv.cvtColor(img_copy, cv.COLOR_BGR2GRAY)
    _, threshold = cv.threshold(gray, 145, 255, cv.THRESH_BINARY)
    contours, _ = cv.findContours(threshold, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

    for contour in contours:
        approx = cv.approxPolyDP(contour, 0.01 * cv.arcLength(contour, True), True)
        cv.drawContours(img_copy, [contour], 0, (0, 0, 255), 5)
        x = 0
        y = 0
        # finding center point of shape
        M = cv.moments(contour)
        if M['m00'] != 0.0:
            x = int(M['m10'] / M['m00'])
            y = int(M['m01'] / M['m00'])

        # putting shape name at center of each shape
        if len(approx) == 3:
            shape = 'TRIANGLE'
            cv.putText(img_copy, 'TRIANGLE', (x, y), cv.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 0), 2)
        elif len(approx) == 4:
            shape = 'QUAD'
            cv.putText(img_copy, 'QUAD', (x, y), cv.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 0), 2)
        elif len(approx) == 5:
            shape = 'PENTA'
            cv.putText(img_copy, 'PENTA', (x, y), cv.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 0), 2)
        elif len(approx) == 6:
            shape = 'HEXA'
            cv.putText(img_copy, 'HEXA', (x, y), cv.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 0), 2)
        else:
            shape = 'CIRCLE'
            cv.putText(img_copy, 'CIRCLE', (x, y), cv.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 0), 2)

    out_img = img_copy.transpose([1, 0, 2]).reshape(resolution[0] * resolution[1] * 3).tolist()
    return out_img, shape
```

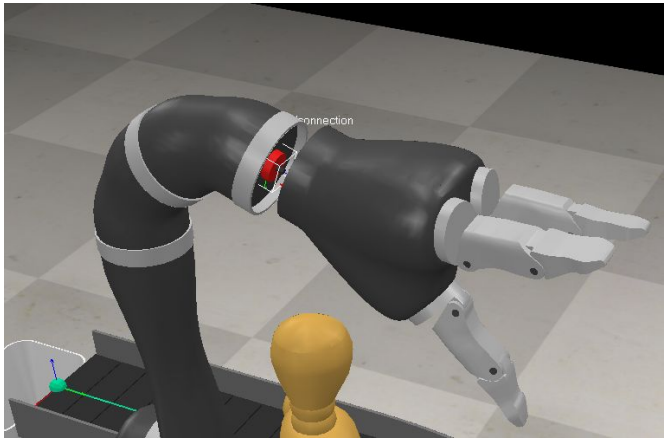
Force Sensor

In order to weigh the object, a force sensor is placed between the robot's hand and arm.

The reading is done through the function `sim.readForceSensor` which returns the following values:

Lua return values	**result**: bit-coded (same as for the C-function return value) **forceVector**: table holding 3 values (applied forces along the sensor's x, y and z-axes) **torqueVector**: table holding 3 values (applied torques about the sensor's x, y and z-axes)

To retrieve in the simplest way the mass, we align the approach axis of the hand with the Z axis of the object, and after the grasping we simply use the **Newton's second law**.



Motion Planning

To placing the grasped objects to the desired positions we use the motion path planning thanks to the **OMPL** library.

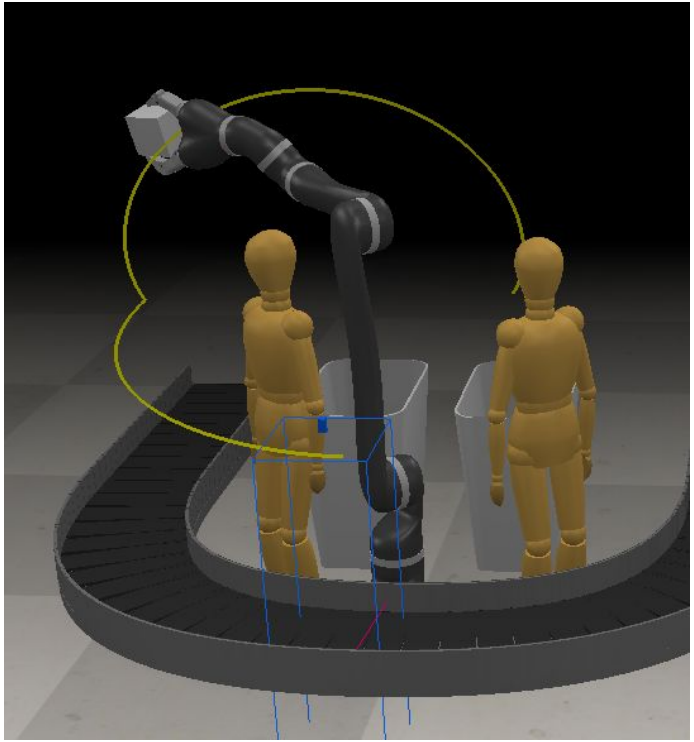
This library allows to use different algorithms that calculate different possible paths given the initial pose and the final pose **avoiding obstacles**.

```
function visualizePath(path)
function _getJointPosDifference(startValue,goalValue,isRevolute)
function _applyJoints(jointHandles,joints)
function generatePathLengths(path)
function getShiftedMatrix(matrix,localShift,dir)
function validationCb(config,auxData)
function findCollisionFreeConfig(matrix)
function findSeveralCollisionFreeConfigs(matrix,trialCnt,maxConfigs)
function getConfig()
function setConfig(config)
function getConfigConfigDistance(config1,config2)
function getPathLength(path)
function _findPath(startConfig,goalConfigs)
function findPath(startConfig,goalConfigs)
function generateIkPath(startConfig,goalPose,steps,ignoreCollisions)
function executeMotion(path,lengths,maxVel,maxAccel,maxJerk)
```

```
local task=simOMPL.createTask('task')
simOMPL.setAlgorithm(task,OMPLAlgo)
local jSpaces={}
for i=1,#jh,1 do
    local proj=i
    if i>3 then proj=0 end
    jSpaces[#jSpaces+1]=simOMPL.createStateSpace('j_space'..i,simOMPL.StateSpaceType.joint_position,jh[i],{jointLimitsL[i]},{jointLimitsH[i]},proj)
end
simOMPL.setStateSpace(task,jSpaces)
simOMPL.setCollisionPairs(task,collisionPairs)
simOMPL.setStartState(task,startConfig)
simOMPL.setGoalState(task,goalConfigs[1])
for i=2,#goalConfigs,1 do
    simOMPL.addGoalState(task,goalConfigs[i])
end
simOMPL.setup(task)
local l=nil
local res,path=simOMPL.compute(task,maxOMPLCalculationTime,-1,200)
if path then
    visualizePath(path)
    l=getPathLength(path)
end
simOMPL.destroyTask(task)
return path,l
```

1 - Create a path planning task
2 - Create the required state space
3 - Specify which entities are not allowed to collide
4 - Specify the start and goal states
5 - Compute path
6 - Destroy the path planning task

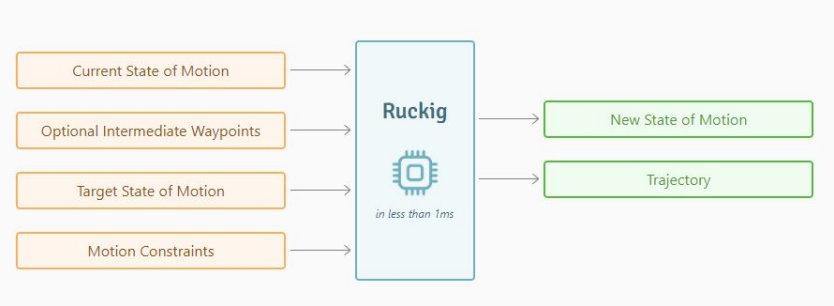
OMPL Planner

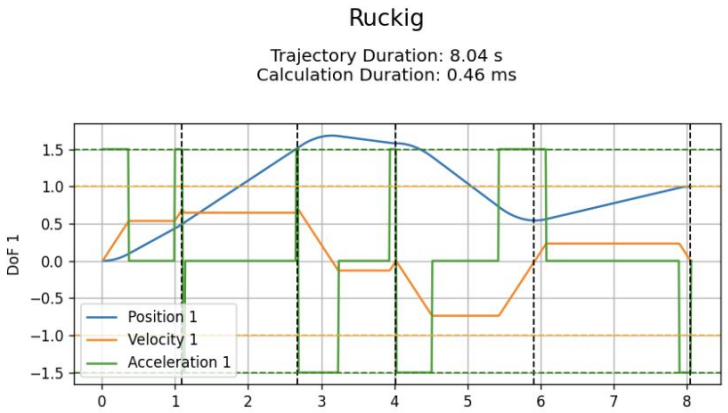


For motion path planning we used the **Lower Bound Tree Rapidly-exploring Random Trees** (LBTRTT) algorithm. It belongs to the sampling methods family.

From the OMPL documentation: ***LBTRRT*** *(Lower Bound Tree RRT) is a near asymptotically-optimal incremental sampling-based motion planning algorithm. LBTRRT algorithm is guaranteed to converge to a solution that is within a constant factor of the optimal solution. The notion of optimality is with respect to the distance function defined on the state space we are operating on.*

Motion and trajectory generation





Ruckig is a next-generation motion planning library for robotic applications.

Ruckig is able to generate trajectories on-the-fly, and therefore allows robots and machines to react instantaneously to sensor input.

The core **Ruckig algorithm** calculates a trajectory to a target state (with position, velocity, and acceleration) starting from any initial state limited by velocity, acceleration, and jerk constraints.

simRuckigPos / sim.ruckigPos

Description	Executes a call to the Ruckig online trajectory generator. The Ruckig online trajectory generator provides instantaneous trajectory generation capabilities for motion control systems. This function prepares a position-based trajectory generation object, that can then be calculated with sim.ruckigStep. When this object is not needed anymore, remove it with sim.ruckigRemove. See also sim.ruckigVel, sim.moveToPose and sim.moveToConfig.

Kinematics, Dynamics and Control

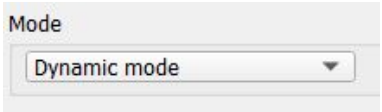
A joint can be in one of following 3 modes:

- **Kinematic mode**: the joint operates in kinematic mode and is not handled by the dynamics module. It can instantaneously change its linear/angular position (via sim.setJointPosition), display a specific motion profile (via sim.setJointTargetPosition or sim.setJointTargetVelocity), or be handled in a customized manner via a joint callback function
- **Dependent mode**: the joint is directly dependent of another joint via a linear equation. The joint is not handled by the dynamics module
- **Dynamic mode**: the joint is handled by the dynamics module, if it forms with its connecting items a valid configuration, i.e. is dynamically enabled. In that case, it can be controlled in various modes

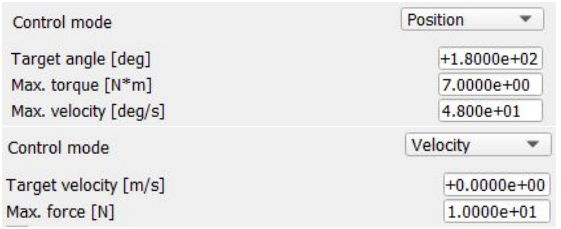
There are many different ways a joint can be controlled, depending on its joint mode and control mode, but one can differentiate between high-level control and low-level control:

High-level control is achieved mainly via specific API functions, such as sim.setJointPosition, sim.setJointTargetPosition, sim.setJointTargetVelocity or sim.setJointTargetForce. Depending on the joint mode and control mode, not all functions make sense, e.g. calling sim.setJointTargetForce on a kinematic joint, or calling sim.setJointTargetPosition on a dynamic joint in velocity control

Low-level control is best implemented via a joint callback function from within CoppeliaSim. In that case the joint should be in kinematic mode, or in dynamic mode and CoppeliaSim. The callback function will then constantly called by CoppeliaSim, in order to fetch new regulation values



Arm is position controlled
Hand is velocity controlled



Several control modes are supported:

- **Free (no control)**: the joint is free, i.e. non-motorized
- **Force/torque control**: the joint is controlled in force/torque, i.e. a constant force/torque is applied. Can be modulated with sim.setJointTargetForce
- **Velocity control**: the joint is controlled in velocity: the specified force/torque is applied until the desired velocity is reached. Optionally, a specific motion profile for the velocity can be applied. Can be modulated with sim.setJointTargetVelocity and sim.setJointTargetForce
- **Position control**: the joint is controlled in position: the specified force/torque is applied and the velocity is modulated accordingly, until the target position/angle is reached. Parameters can be adjusted with sim.setJointTargetPosition, sim.setJointTargetForce and sim.setObjectFloatParam
- **Spring-damper control**: the joint is controlled in position by modulating the exerted force/torque via a simple KC controller, trying to reach the zero displacement position/angle. Parameters can be adjusted with sim.setJointTargetPosition, sim.setJointTargetForce and sim.setObjectFloatParam
- **Custom control**: the joint calls a joint callback function for control, where the user can decide of a force/torque and target velocity in a flexible way

Demo

Future improvements

Our project is just a prototype and a concept idea, many things can be improved obviously, such as:

- Advanced object recognition with everyday objects
- More realistic vision sensor position
- Simulations in more complex environments, such as factories and contexts where people are dynamic

Thanks for the attention