Robotics Homework

Industrial Automation mod. B

Claudia Vinci

Viviana Casale

Mary Aliberto

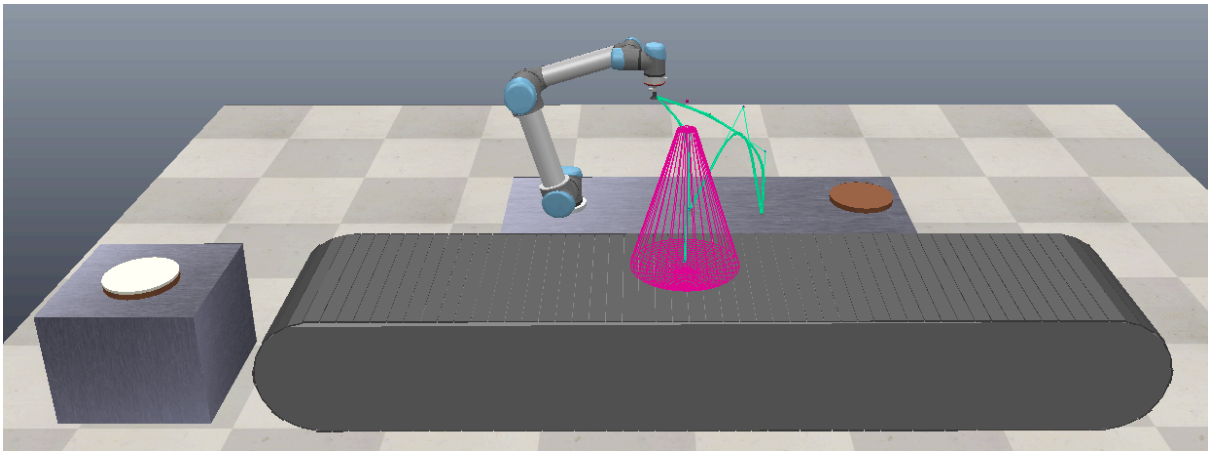
Index

Scenario

The chosen scenario simulates an automated pick-and-place process using a UR5 robotic arm and a conveyor belt system. The simulation represents an industrial setup where sponge cakes are transported, detected, picked, and placed precisely by the robot.

The process begins with the conveyor belt moving the sponge cakes toward a proximity sensor positioned at the pick-up zone. Once the sensor detects an incoming sponge cake, the conveyor automatically stops, allowing the UR5 manipulator to execute a pick operation. The UR5 follows a predefined pick path, adjusting its end-effector alignment using feedback from a Baxter vacuum cup proximity sensor to ensure accurate grasping.

After successfully picking up the sponge cake, the robot moves along a dynamically generated place path to deposit the item onto a placement area, where another proximity sensor verifies the correct position. Once placement is completed, the robot follows a return path back to its initial pose, while new sponge cakes are automatically spawned, and the conveyor belt restarts, enabling the simulation of continuous operation.



Demonstration Video Link:

pick&place.avi

Requirements

- CoppeliaSim Edu, Version ^4.10.0 (rev. 0)

Coppelia models

- **Conveyor belt:** *equipment/conveyors/generic conveyor (belt)*
- **Robotic Arm:** *robots/non-mobile/UR5*

  The UR5 is a six-degree-of-freedom collaborative robotic arm developed by Universal Robots. It is designed for a wide range of industrial and research applications, offering a balance between flexibility, precision, and ease of integration. With a reach of approximately 850 mm and a payload capacity of 5 kg, the UR5 is well-suited for tasks such as pick-and-place operations, machine tending, packaging, and laboratory automation.

  (https://www.universal-robots.com/media/50588/ur5_en.pdf)

- **End-effector (vacuum cup):** *components/grippers/Baxter Vacuum Cup*

  The Baxter Vacuum Cup is a pneumatic gripping device designed to handle delicate or irregularly shaped objects using suction-based adhesion. Instead of applying mechanical pressure like traditional grippers, the vacuum cup creates a negative pressure that securely holds an object's surface, making it ideal for soft or deformable materials.

  In this simulation, the Baxter Vacuum Cup is used as the end-effector of the UR5 robotic arm to manipulate sponge cakes. Because sponge cakes are light, soft, and fragile, conventional grippers could easily damage or deform them. The vacuum cup, on the other hand, ensures a gentle and uniform grip without applying excessive force, maintaining the integrity of the product during handling.

  The system also integrates a proximity sensor within the vacuum cup, allowing the robot to detect the exact position of the sponge cake before suction is activated.

Scene setup

1. Adding the Conveyor Belt

Insert the conveyor model (*Add → Models → Conveyor* or a custom one). Position and orient it along the desired direction of movement. Then, configure its dynamic properties: nominal speed (`0.5 m/s`), direction, acceleration (`0.5 m/s`). The conveyor provides a continuous flow of objects (cakes) toward the pick-up area.

2. Adding the First Table (along the conveyor's long side)

Insert a table (*Add → Shapes → Cuboid*) and size it according to the UR5 workspace. Place it next to the long side of the conveyor, aligned with the robot's working zone. The adequate position ensures that the robot has easy access to the conveyor for the pick operation.

3. Positioning the UR5 on the First Table

Place the UR5 model and place its base on the created table. Properly orient the UR5 to ensure smooth, collision-free motion.

4. Adding the Second Table (on the conveyor's short side)

Insert another table and position it along the conveyor's short side, before the start of the belt (the loading zone). This table will hold the template cake model used for spawning cakes onto the conveyor.

5. Create Sponge Cake templates

To create the Sponge Cake Model (single cylinder) add a cylindrical primitive shape in the scene (*Add → Primitive shape → Cylinder*). Set its dimensions to represent the sponge cake portion (e.g. diameter: 24 cm, height: 2 cm). Place the sponge cake template model on the first table, and assign it a unique alias (e.g., `Spongecake_template`) so the script can duplicate it during the simulation, since it will be the reference for spawning sponge cakes on the pick table.

To create the Cake Model (double cylinder), add two cylinders to the scene to represent the two layers of the cake: bottom layer (sponge base) and top layer (representing the cream layer) setting its height to 1 cm, placed one on top of the other. Group the two cylinders (*Select both → Edit → Grouping/Merging → Group selected shapes*) so they behave as a single object. Rename this grouped model assigning a unique alias, such as `Sponge_arriving_template` on the second table. This will be the reference for spawning cakes on the conveyor belt.

6. Adding the Baxter Vacuum Cup as the End-Effector

Attach the Baxter Vacuum Cup model as UR5 End-Effector. To do so, it is necessary to insert the Baxter Vacuum Cup model inside */UR5/.../link7_visible/connection*, in order to properly assemble the vacuum cup as end-effector.

Since the Baxter Vacuum Cup comes with a proximity sensor, rotate the sensor itself so that its positive z axis (the detection axis) points out of the vacuum cup.

The vacuum cup provides a soft, non-invasive grip suitable for delicate items like sponge cakes, preventing damage during handling.

7. Adding a Conical Proximity Sensor Above the Conveyor

Create a Proximity Sensor (*Add → Sensors → Proximity Sensor*) and set its detection volume to a cone shape. Position the sensor above the conveyor and rotate it so the cone points downward toward the belt's surface. Set a proper detection distance (range) depending on the conveyor and UR5 height, and adjust the radius to cover a portion of the belt slightly larger than the sponge cake's size, ensuring reliable detection even in case of small lateral displacements. It is also possible to increase the face count, for smoother cone geometry.
This sensor detects when a sponge cake arrives at the pick&place position, triggering the conveyor stop and the robot pick action.

Inverse Kinematics

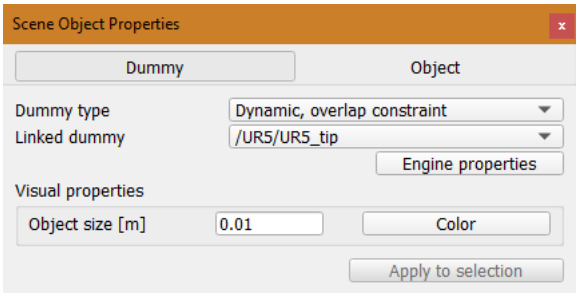
In this simulation, the UR5 robotic arm is controlled using the principle of inverse kinematics (IK). Inverse kinematics is a computational method used to determine the joint angles required for the robot's end-effector (in this case, the vacuum cup) to reach a specific position and orientation in space.

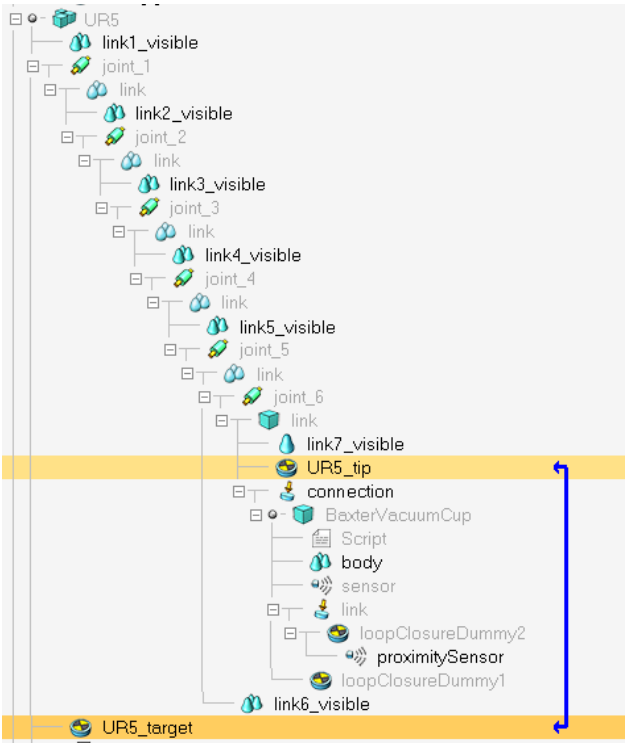
While forward kinematics calculates the end-effector's pose from known joint angles, inverse kinematics works in the opposite direction — starting from the desired target pose and computing the corresponding joint configurations that achieve it.

In CoppeliaSim, this is implemented by associating the UR5 arm with a kinematic chain described with a tip dummy attached to the end-effector which follows a target dummy. The simulation continuously solves the IK equations so that the robot's joints automatically adjust to keep the tip (the end-effector) aligned with this moving target.

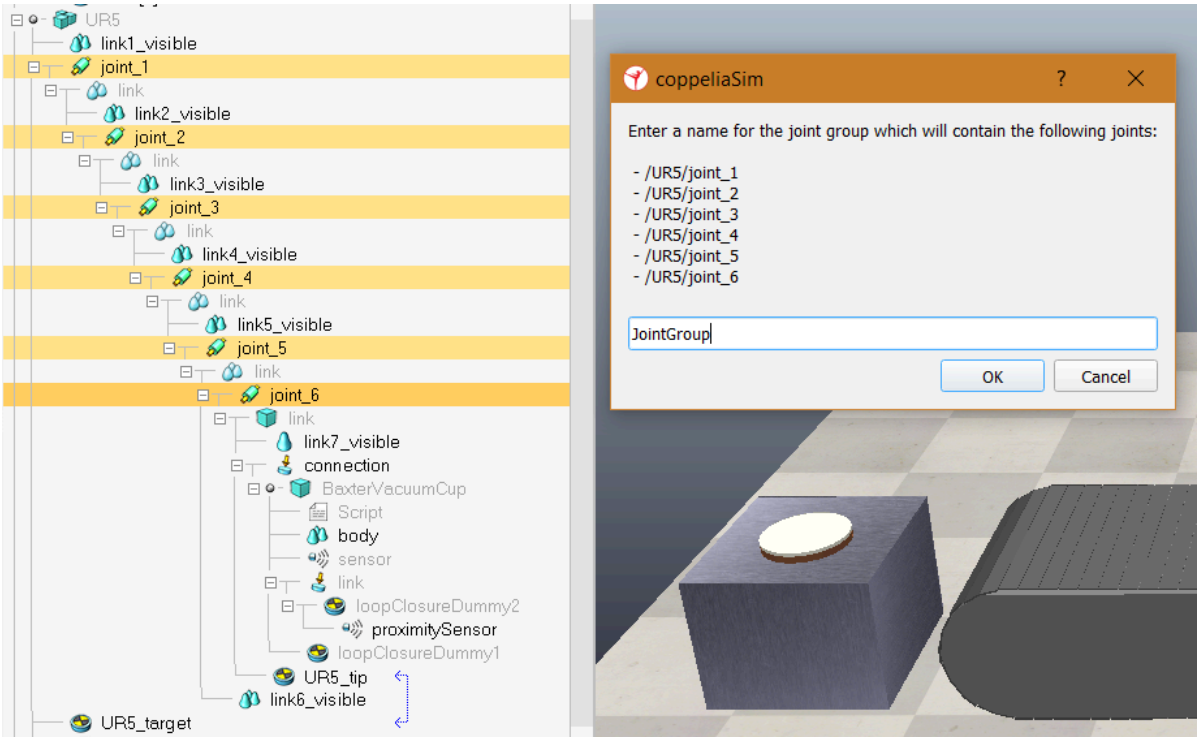
First, the tip has to be attached to the end-effector. Thus, add a *dummy object* (`UR5_tip`) to the last link (the link inside the last joint). Then, to define the target add another dummy object (`UR5_target`) inside */UR5*.

The tip and the target have to be linked in order to make the tip follow the target, and thus move the UR5. To do so, double click on one of the two dummy objects and select the other one in the "*Linked dummy*" section.

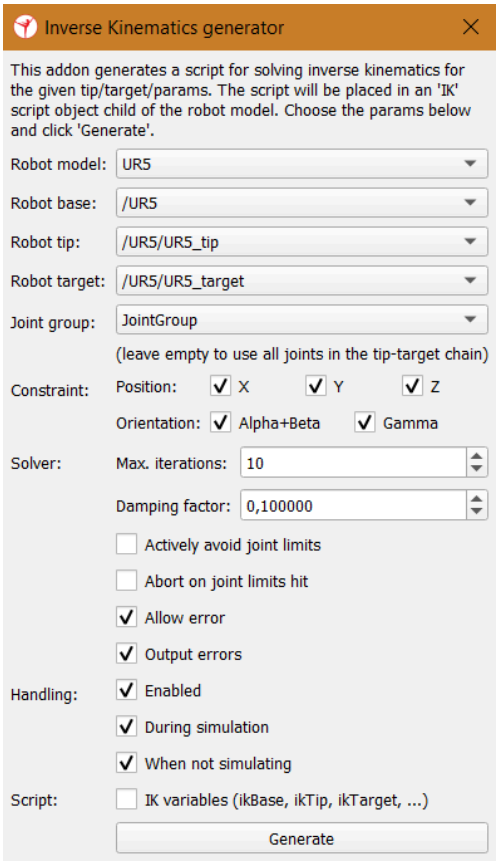




Next, in order to generate the Inverse Kinematics, a joint kinematic chain has to be created. To do so, select all the joints and then go to *Modules/Kinematics/Define Joint Group*.



Then, by clicking on *Modules/Kinematics/Inverse Kinematics Generator* it is possible to generate the inverse kinematics (IK) by selecting: the robot model, the robot base, the tip, the target and the defined joint group.



The resulting IK is a set of nonlinear equations (managed inside the "customization script" created when generating the IK) that describe the relationship between the robot's joint variables and the desired end-effector pose (its position and orientation in 3D space), and they are solved by the IK solver each time the target moves, finding the adequate joint variables that allow the tip to follow the target (hence, the desired movement of the UR5).

Paths definition

In CoppeliaSim, a path represents a predefined trajectory in 3D space that an object (such as a robot's end-effector or target) can follow during the simulation. It is composed of a series of control points, which define the spatial positions and orientations that make up the shape and smoothness of the path.
A *path* is essentially a **curve** or that connects multiple key points in space, called control points. Each control point defines a **specific pose** along the trajectory, which includes:

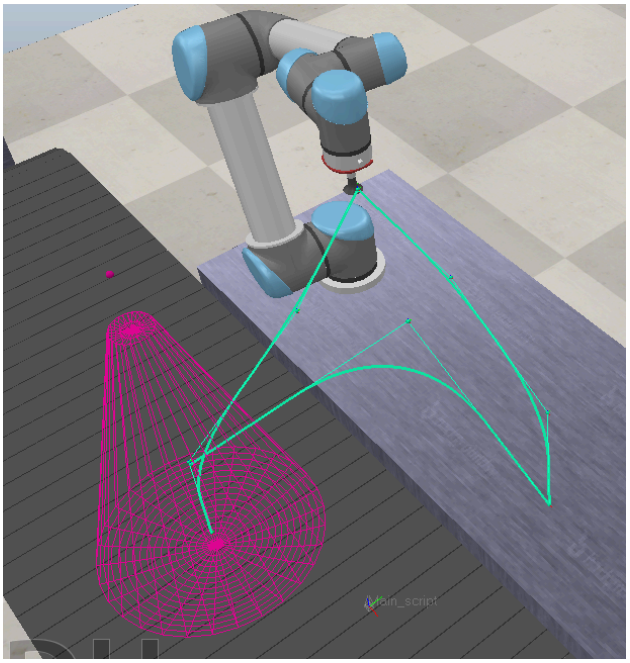
- Position coordinates (x, y, z) — defining where the point is in space,
- Orientation (q_1, q_2, q_3, q_4) — a quaternion defining how the object should be rotated.

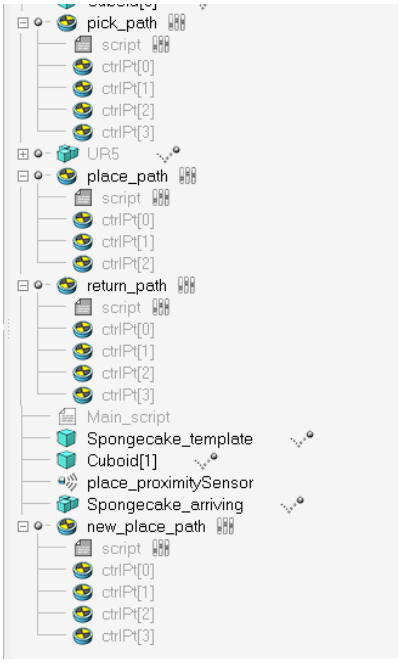
By interpolating between these points, CoppeliaSim can generate a continuous motion trajectory that can be followed smoothly by an object such as the `UR5_target`.

Paths can be created manually by clicking *Add* → *Path* → *Open* or *Closed* to insert a new path object into the scene, and then open the Path properties panel to add, delete, or move control points.

In this simulation paths are used to guide the robot's target dummy (`UR5_target`) along smooth motion trajectories:

- ***Pick path:*** the approach and grasp movement toward the sponge cake.
- ***Place path:*** the transfer and placement motion toward the arriving cake.
- ***Return path:*** the motion returning the robot to its initial position.





Pick paths

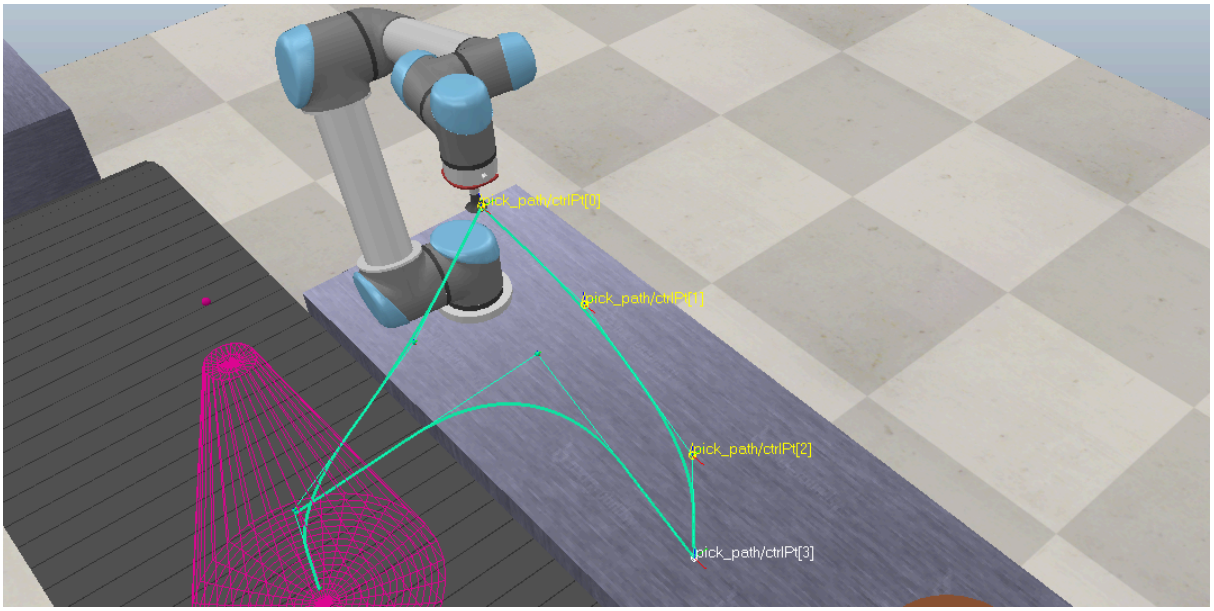
The first path defined is the pick path, which describes the movement necessary to pick up the sponge cake layer to place on top of the arriving cake.

It is composed of four control points:

- The ***first control point*** is placed at the same position as the UR5 tip, ensuring that the path starts exactly from the robot's current end-effector pose.
- The ***last control point***, on the other hand, is located precisely at the pickup target, i.e., the point where the sponge cake layer must be grasped. This represents the goal position of the pick motion.
- The ***two intermediate control points*** are positioned strategically to create the desired curved trajectory, allowing for a smooth approach toward the object and avoiding abrupt or linear movements that could cause collisions or unstable motion.

All control points share the same orientation, aligned with both the tip and target dummies.

This uniform orientation ensures that the end-effector maintains a constant pose throughout the entire motion, keeping the Baxter Vacuum Cup perpendicular to the sponge cake surface during approach and grasp.



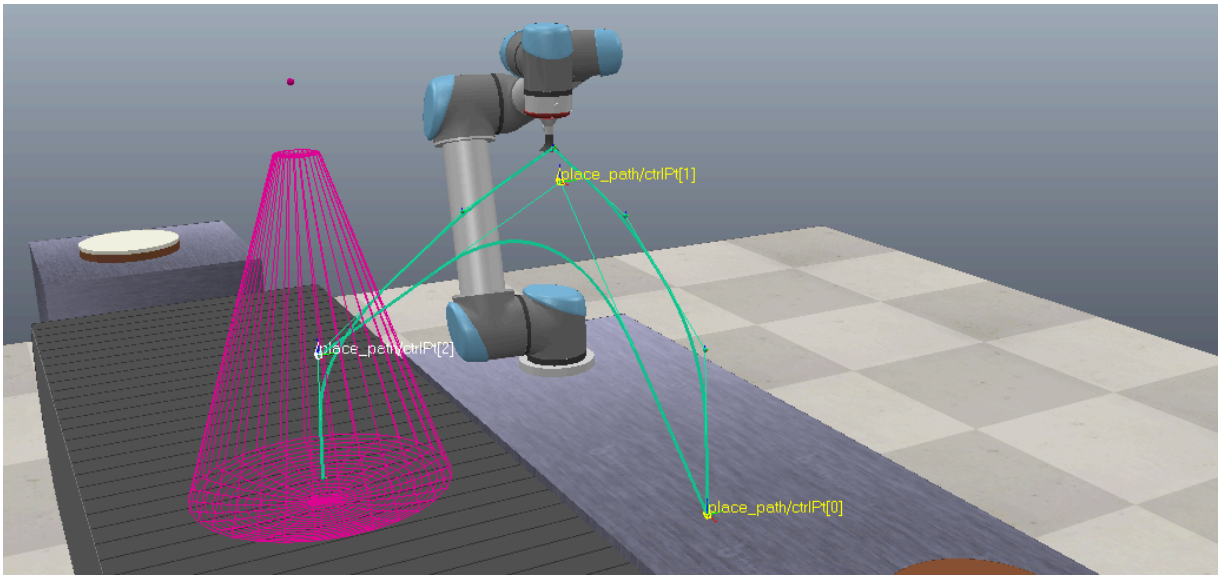
Place path

The place path defines the transfer and placement motion toward the arriving cake.

Unlike the static pick path, the place path does not represent a fixed motion. Instead, it provides a reference structure (a predefined shape with control points) that the simulation script uses to adapt the robot's placement trajectory to the real-time position of the arriving cake on the conveyor.

The place path template is composed of three control points, where the first one is at pick target position, and the other two serve as intermediate points to create the desired curve.

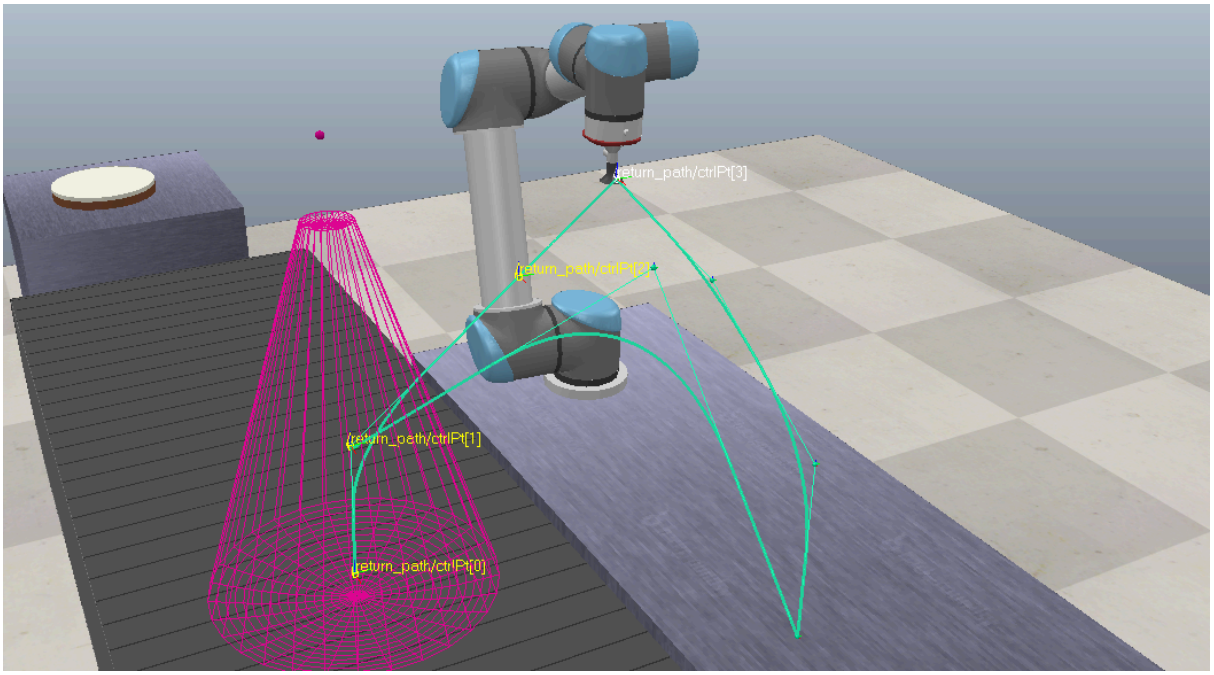
During the simulation, when the proximity sensor detects an incoming cake, the script retrieves the position of that object and uses the template path to build a new adjusted path. The final control point of this new path is automatically updated so that it matches the exact location of the detected cake, ensuring the robot places the picked sponge layer precisely on top of it. This dynamically generated trajectory is saved as **`new_place_path`** and temporarily used for the placement motion.



Return path

The return path defines the static trajectory that the UR5 follows to move back to its initial position after completing the placement operation. Its purpose is to reset the robot's configuration, preparing it for the next pick-and-place cycle, thus the path connects the final position of the placement motion with the robot's initial rest pose.

Its three control points are arranged to ensure a natural retraction movement — lifting the end-effector away from the workspace before returning to the starting position above the table.



Lua script

In this simulation, all scene components — including the UR5 robot, conveyor belt, proximity sensors, and vacuum gripper — perform their specific tasks in a coordinated and automated manner.
Their behavior and interactions are managed entirely through the ***Main Script***, which serves as the central control unit of the simulation.

The Main Script continuously monitors the sensor readings (sensing phase) and, based on these inputs, executes the corresponding actions (actuation phase).
This includes starting or stopping the conveyor belt, activating the pick-and-place sequence of the UR5, generating dynamic paths for accurate alignment, and handling the spawning and removal of sponge cake objects.

Through this structure, the Main Script ensures synchronized operation among all elements in the environment, allowing the system to function as an integrated and autonomous process.

Initialization

The initialization state starts by calling the init system call. The ***`sysCall_init()`*** function is executed once at the beginning of the simulation and is responsible for initializing all the main components and parameters required for the system to operate correctly.

During this phase, the script retrieves the ***handles*** of all relevant scene objects — such as the UR5 robot target, the conveyor belt, proximity sensors, and the sponge cake templates — and stores them for later use.

In addition, `sysCall_init()` sets the initial values of ***key variables*** and ***flags***, such as motion velocities, control states (e.g., `picked`, `completed`, `conv_stopped`), and the initial positions of the robot.

It also defines the ***reference poses*** used for spawning the sponge cake objects and calls the ***spawn functions*** to generate the first pair of cakes at their respective starting positions (pickup and conveyor).

This initialization step ensures that all objects are properly referenced, all motion paths are ready for use, and the simulation environment is fully prepared for the sensing and actuation cycles that follow.

```lua
function sysCall_init()
    sim = require('sim')

    ---------- CONVEYOR BELT ----------
    conveyor = sim.getObject("/conveyor")
    belt_runVel = 0.5   -- velocit? del conveyor belt

    --------- PICK AND PLACE ------------
    UR5_target = sim.getObject('/UR5/UR5_target')
    pick_path = sim.getObject('/pick_path')
    place_path = sim.getObject('/place_path')
    baxter_cup = sim.getObject('../BaxterVacuumCup/link')
    baxter_prox_sens = sim.getObject('../BaxterVacuumCup/sensor')
    spongecake_template = sim.getObject('/Spongecake_template')
    sponge_arriving_template = sim.getObject("/Sponge_arriving_template")
    place_prox_sens = sim.getObject('/place_proximitySensor')
    UR5_run_vel = 0.5
    UR5_init_pos = sim.getObjectPosition(UR5_target)
    pick_path_initialized = false
    posAlongPlacePath = 0

    -- flags
    picked = false
    completed = false
    place_detected = 0
    conv_stopped = true -- inizia da fermo
    alignment = false
    baxter_detected = 0

    ---------- RETURN TO INITIAL POSITION ---------
    return_path = sim.getObject('/return_path')
    posAlongReturnPath = 0

    ---------- SPONGECAKE SPAWN AND DELETION ----------
    sponge_pose = {0.275, 0.236, 0.41, 0., 0., 0., 1.}
    sponge_arriving_pose = {-0.8, -0.225, 0.422, 0., 0., 0., 1.}

    spongecake = spawn_spongecake(spongecake_template, "Spongecake_pick", sponge_pose)
    spongecake_arriving = spawn_spongecake(sponge_arriving_template, "Spongecake_arriving", sponge_arriving_pose)

end
```

Sensing and actuation

The simulation's control logic is structured around two key system calls: `sysCall_sensing()` and `sysCall_actuation()`, which are executed cyclically at each simulation step.
Together, they manage the continuous interaction between the robot, sensors, and conveyor, ensuring coordinated and autonomous behavior of the entire system.

- ***`sysCall_sensing()`*** handles the ***data acquisition phase***, where the script reads the state of all proximity sensors and other relevant inputs from the environment.
  This includes detecting the presence of objects on the conveyor and verifying the alignment between the vacuum cup and the sponge cake.
  The gathered information is then used to update the system's internal state variables (e.g., `place_detected`, `baxter_detected`), which determine the next control actions.

- ***`sysCall_actuation()`*** manages the ***execution phase***, using the updated sensor data to decide and perform the appropriate actions.
  It controls the conveyor's motion, the UR5's pick-and-place operations, and the return sequence to the initial position. Through this function, the robot and all

scene elements respond dynamically to sensor feedback, allowing for a fully automated and reactive process.

Together, these two system calls form a closed sensing–actuation loop, enabling real-time interaction between perception and motion control within the simulation environment.

Conveyor

The conveyor belt's operation is controlled automatically through the defined functions `start_conveyor_belt()`, `stop_conveyor_belt()`, and the `sysCall_actuation()` loop. This logic allows the conveyor to start or stop dynamically depending on the detection of an incoming cake by the proximity sensor.

The ***`start_conveyor_belt()`*** function activates the conveyor when no object is detected by the sensor and the belt is currently stopped. It prints a confirmation message and sets the flag `conv_stopped` to `false`.
The conveyor's velocity is then updated by writing into its custom buffer property using the built-in method:

`sim.setBufferProperty(conveyor,'customData.__ctrl__',sim.packTable({vel=belt_runVel}))`

This line sends the target speed value (`belt_runVel`) to the conveyor's control script, causing it to move.

The ***`stop_conveyor_belt()`*** function stops the conveyor as soon as the proximity sensor detects an object on the belt (a cake ready for pickup). It prints a confirmation message and sets `conv_stopped` to `true`.
The same buffer property is updated, but this time the velocity is set to zero (`vel=0.0`).

```
78 function start_conveyor_belt()
79     print("torta completata, riparte il conveyor")
80     conv_stopped = false
81     sim.setBufferProperty(conveyor, 'customData.__ctrl__', sim.packTable({vel=belt_runVel}))
82 end
83
84 function stop_conveyor_belt()
85     print('torta rilevata, fermo il conveyor')
86     conv_stopped = true
87     sim.setBufferProperty(conveyor, 'customData.__ctrl__', sim.packTable({vel=0.0}))
88 end
89
```

This stops the belt's motion; the stop isn't immediate since the simulation conveyor belt has a certain acceleration.

The ***sysCall_actuation()*** function determines when to start or stop the conveyor based on two variables:

- *place_detected* → indicates whether the proximity sensor has detected an object.
- *conv_stopped* → indicates whether the conveyor is currently moving or not.

The logic is as follows:

1. If no object is detected (`if place_detected == 0`) and the conveyor is stopped (`if conv_stopped`), the script calls `start_conveyor_belt()`.
   → The belt starts moving to bring a new cake into the pickup area.

2. If an object is detected (`if place_detected == 1`) and the conveyor is running (`if not conv_stopped`), the script calls `stop_conveyor_belt()`.
   → The belt stops so that the robot can perform the pickup operation safely.

```
43 function sysCall_actuation()
44     -- START - Faccio muovere il conveyor belt
45     -- non ho rilevato nulla e il conveyor era fermo
46     if place_detected == 0 and conv_stopped then
47         start_conveyor_belt()
48     -- ho rilevato qualcosa e il conveyor era in movimento
49     elseif not conv_stopped and place_detected == 1 then
50         stop_conveyor_belt()
51     end
```

This simple conditional control ensures that the conveyor moves only when needed (to deliver new cakes), and it stops automatically whenever a cake reaches the pickup area.

Detection

The detection phase corresponds to the ***sysCall_sensing()*** function, which is executed at every simulation step to update the state of both proximity sensors:

- *baxter_prox_sens*, which is built-in to the Baxter vacuum cup, used to align with the sponge cake layer in the pick position.
- *place_prox_sens*, placed above the conveyor belt to detect the incoming cake.

```
function sysCall_sensing()

    if not completed and not picked then
        place_detected, _, place_det_point = sim.readProximitySensor(place_prox_sens)
    end
    baxter_detected, _, baxt_det_point, baxt_detectedObj = sim.readProximitySensor(baxter_prox_sens)
end

```

The first `if` condition ensures that the place sensor (`place_prox_sens`) is only read when the robot has not yet completed the task and has not already picked the sponge cake. This avoids redundant detections during the motion phases.

The command `sim.readProximitySensor(sensor_handle)` returns whether the sensor detects an object (`1` if detected, `0` otherwise), as well as the detection point coordinates and the detected object handle.
The variable `place_detected` is therefore used as a boolean flag to indicate the presence of a sponge cake on the conveyor.

The second sensor (`baxter_prox_sens`) continuously checks for objects in the pick area, updating the variables `baxter_detected` and `baxt_detectedObj`.

The code inside the ***sysCall_actuation()*** defines what happens after the sensors detect an object.

```
    if conv_stopped and place_detected == 1 then
        if not picked then
            initSimulationTime = sim.getSimulationTime()
            UR5_vel = UR5_run_vel
            pick_spongecake()
        else
            UR5_vel = UR5_run_vel
            place_spongecake()
        end
    else
        if completed then
            UR5_vel = UR5_run_vel
            return_to_initial_pos()
        end
    end
```

The first condition (`if conv_stopped and place_detected == 1 then`) verifies that the conveyor is stopped and that the proximity sensor detected a sponge cake. When both are true, the robot can safely start the pick-and-place operation.

- ***Pick Phase***

  If the sponge cake has not yet been ***picked*** (`not picked`), the system records the current simulation time (`InitSimulationTime`), which is necessary to correctly retrieve the target position along the path. Then it sets the robot's motion velocity (`UR5_vel = UR5_run_vel`), and calls the function `pick_spongecake()`.
  This initiates the path following that moves the UR5 toward the pickup point.

- ***Place Phase***

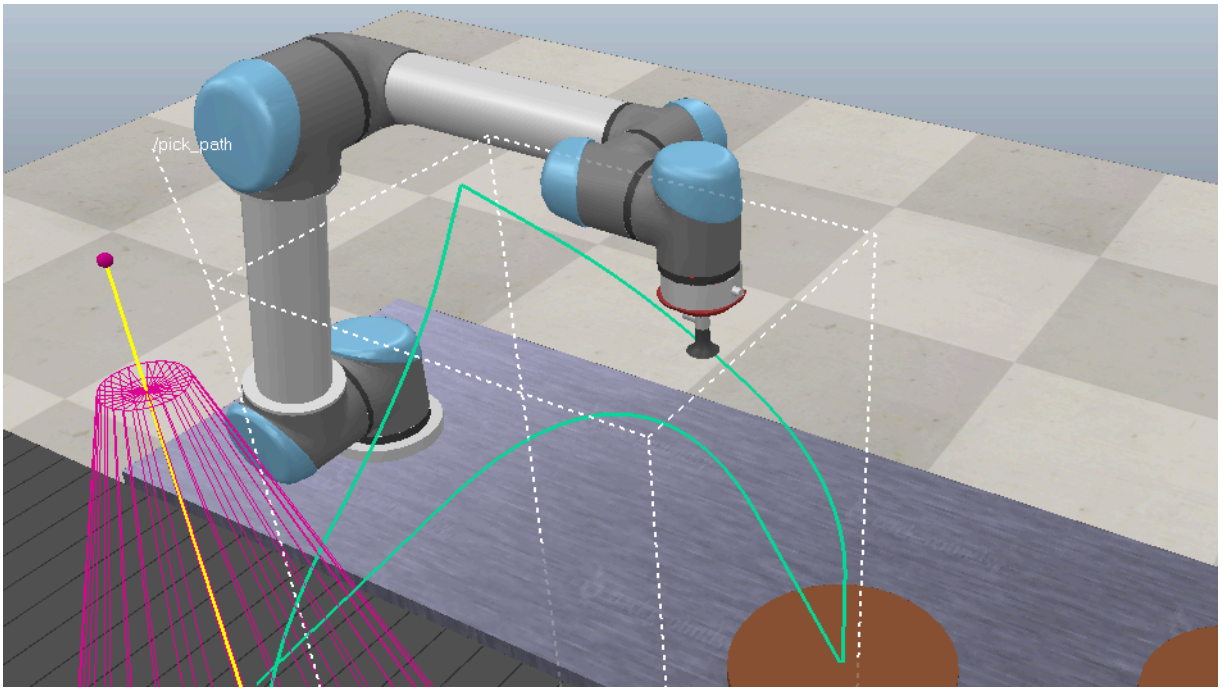
  Once the sponge cake has been picked, the flag `picked` becomes `true`. In the next iteration, the same condition will lead to the placement phase (`place_spongecake()`), moving the robot to position the layer on the target cake.

- ***Return Phase***

  When the placement operation is ***completed***, the flag `completed` becomes `true`.
  The robot then executes `return_to_initial_pos()` to move back to its starting position, ready for the next cycle.

Pick

The ***`pick_spongecake()`*** function manages the pick phase of the simulation, where the UR5 robot moves along a predefined pick path to grab the sponge cake using the Baxter Vacuum Cup.



The `pick_spongecake()` function first retrieves the path data from the object property `'customData.PATH'`

If this is the ***first execution***, it initializes variables such as:

- `previousSimulationTime` (to measure time progression), which is initialized with the `InitSimulationTime`, set right before calling the pick function. This step is important to retrieve the correct position for the target along the path, since position is determined by using the equation $x = x_0 + v(t-t_0)$
- `posAlongPickPath`, which is the current position along the path, initialized to zero.
- `pick_path_initialized` is set to true so that initialization only happens once.

Then the `Matrix` object stores all the path information (positions and quaternions). From this data two vectors are extracted:

- `pathPositions`, the position vectors for the robot's movement.
- `pathQuaternions`, the Orientation quaternions used to maintain the correct end-effector alignment.

The function also calculates the `totalLength` of the path using `sim.getPathLengths` function, by passing the path positions and the reference coordinates (3, x-y-z)

```
90  function pick_spongecake()
91      pathData = sim.unpackDoubleTable(sim.getBufferProperty(pick_path, 'customData.PATH'))
92      if not pick_path_initialized then
93          previousSimulationTime = initSimulationTime
94          posAlongPickPath = 0
95          pick_path_initialized = true
96      end
97
98      local m = Matrix(#pathData // 7, 7, pathData)
99      pathPositions = m:slice(1, 1, m:rows(), 3):data()
100     pathQuaternions = m:slice(1, 4, m:rows(), 7):data()
101     pathLengths, totalLength = sim.getPathLengths(pathPositions, 3)
102
```

In the next cycle, once the initialization step is already performed, the current position along the path is updated proportionally to the robot's velocity and elapsed time.

Each time the position is updated, it is checked whether the end of the path has been reached or not.

If `posAlongPickPath >= totalLength`, meaning that the robot has reached the ***end of the path***:

- The velocity is set to zero (`UR5_vel = 0`).

- If the Baxter proximity sensor detects the sponge cake, the code performs an auto-alignment: it calculates a correction vector between the positions of the vacuum cup (`baxter_cup`) and the sponge (`spongecake`).
  This correction adjusts the target's position to ensure accurate pickup even with small misalignments.

- The sponge cake is attached to the vacuum cup using `sim.setObjectParent(spongecake, baxter_cup)`

- The system sets `picked = true`, marking the pick phase as completed. By doing this, in the next actuation call, the system will fall in the placement state.

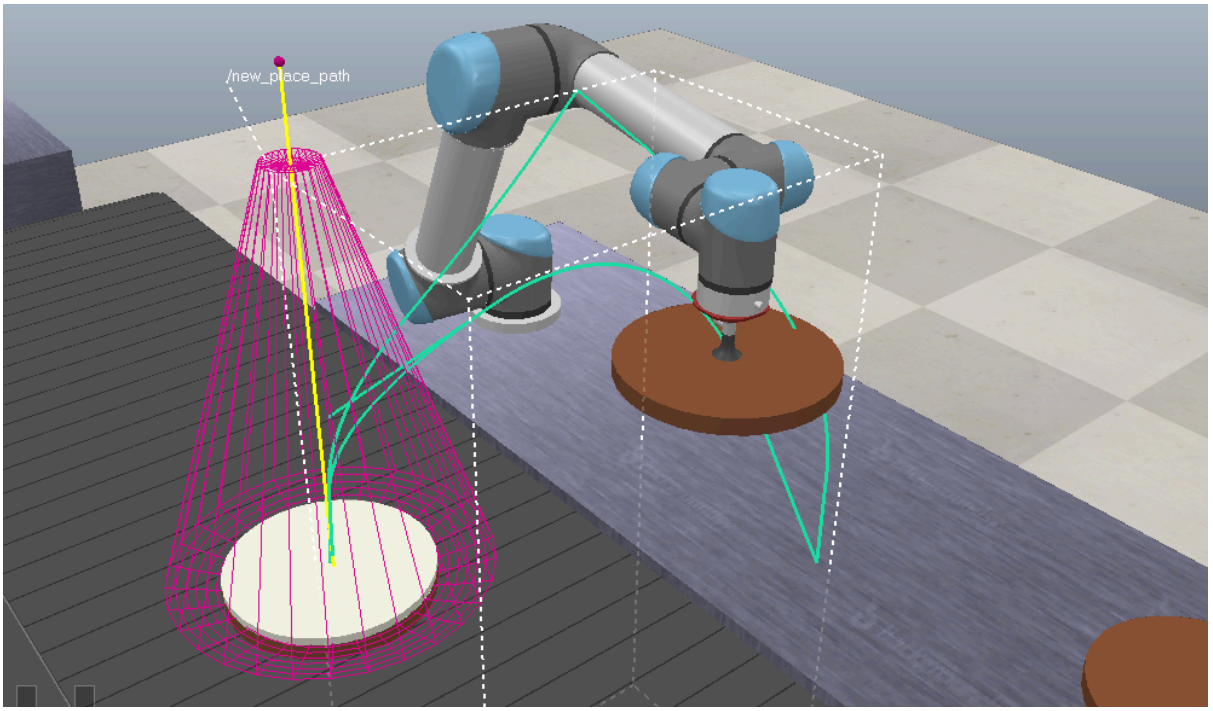
Otherwise, it interpolates the next position and orientation along the path using `sim.getPathInterpolatedConfig()`.Then it updates the UR5 target position (`sim.setObjectPosition()`) and orientation (`sim.setObjectQuaternion`) in real-time, ensuring smooth motion along the pick path.

Finally, the previous simulation time is updated for the next cycle (the next movement along the path until the end is reached).

```
    local t = sim.getSimulationTime()
    posAlongPickPath = posAlongPickPath + UR5_vel * (t - previousSimulationTime)

    if posAlongPickPath >= totalLength then
        UR5_vel = 0
        if baxter_detected == 1 and baxt_detectedObj == spongecake then
            local p = sim.getObjectPosition(baxter_cup, -1)
            local sp = sim.getObjectPosition(spongecake, -1)
            local correction = {sp[1] - p[1], sp[2] - p[2], 0}
            sim.setObjectPosition(UR5_target, -1, {p[1] + correction[1], p[2] + correction[2], p[3]})
            print(string.format("Auto-allineamento pick: delta X=%.3f delta Y=%.3f", correction[1], correction[2]))
        end
        picked = true
        sim.setObjectParent(spongecake, baxter_cup)
        pick_path_initialized = false
    else
        posAlongPickPath = posAlongPickPath % totalLength

        local pos = sim.getPathInterpolatedConfig(pathPositions,
        pathLengths, posAlongPickPath)

        local quat = sim.getPathInterpolatedConfig(pathQuaternions,
        pathLengths, posAlongPickPath, nil, {2, 2, 2, 2})

        -- Move the UR5 Target
        sim.setObjectPosition(UR5_target, pos, pick_path)
        sim.setObjectQuaternion(UR5_target, quat, pick_path)
    end
    previousSimulationTime = t
end
```

Place

The ***place_spongecake()*** function manages the placement phase of the process, where the UR5 robot moves along a dynamically generated path to correctly place the sponge cake layer onto the arriving cake base. It combines path creation, dynamic alignment, and motion execution.



At the beginning of the function, the script checks whether a new alignment is required, by checking the `alignment` and `place_detected` boolean flags. This means that when the proximity sensor detects the presence of the arriving cake and the alignment has not yet been performed, a new placement path is generated.

The ***control points*** of the existing template path (`place_path`) are retrieved by using the `sim.getObjectsInTree()` function, which gives all the children of the specified object (in this case, the path).

Then, the ***position*** of the arriving cake (`spongecake_arriving`) is obtained and the detection point of the proximity sensor (`place_det_point`) is converted from sensor space to world coordinates using:

```
sim.multiplyVector(sim.getObjectMatrix(place_prox_sens,-1),place_det_point)
```

A new ***fourth control point*** **(**`ctrlPt_4`**)** is created, ensuring the Z-coordinate corresponds to the detected position, while the X and Y match the arriving cake's position. This ensures correct vertical alignment between the UR5 and the arriving cake.

The new set of control points (`new_ctrlPts`) is ***flattened*** using a utility function, and then used to create a new placement path, to which an alias (`new_place_path`) is assigned.

Finally, the `alignment` flag is set to true, meaning the robot is now ready to move along this dynamically adjusted trajectory.

```
function place_spongecake()
    if not alignment and place_detected == 1 then
        local ctrlPts = sim.getObjectsInTree(place_path, sim.object_dummy_type, 1)
        sponge_pos = sim.getObjectPose(spongecake_arriving)
        place_pos = sim.multiplyVector(sim.getObjectMatrix(place_prox_sens, -1), place_det_point)
        ctrlPt_4 = {sponge_pos[1], sponge_pos[2], place_pos[3], sponge_pos[4], sponge_pos[5], sponge_pos[6], sponge_pos[7]}
        new_ctrlPts = {sim.getObjectPose(ctrlPts[1]), sim.getObjectPose(ctrlPts[2]), sim.getObjectPose(ctrlPts[3]), ctrlPt_4}
        flat_ctrlPts = flatten(new_ctrlPts)
        new_place_path = sim.createPath(flat_ctrlPts)

        sim.setObjectAlias(new_place_path, "new_place_path")
        alignment = true
    end
    pathData = sim.unpackDoubleTable(sim.getBufferProperty(new_place_path, 'customData.PATH'))
    local m = Matrix(#pathData // 7, 7, pathData)

    pathPositions = m:slice(1, 1, m:rows(), 3):data()
    pathQuaternions = m:slice(1, 4, m:rows(), 7):data()
    pathLengths, totalLength = sim.getPathLengths(pathPositions, 3)
```

The movement along the path works the same as in the `pick_spongecake()` function. What changes are the updates performed as the end of the path is reached; thus, when `posAlongPlacePath >= totalLength:`

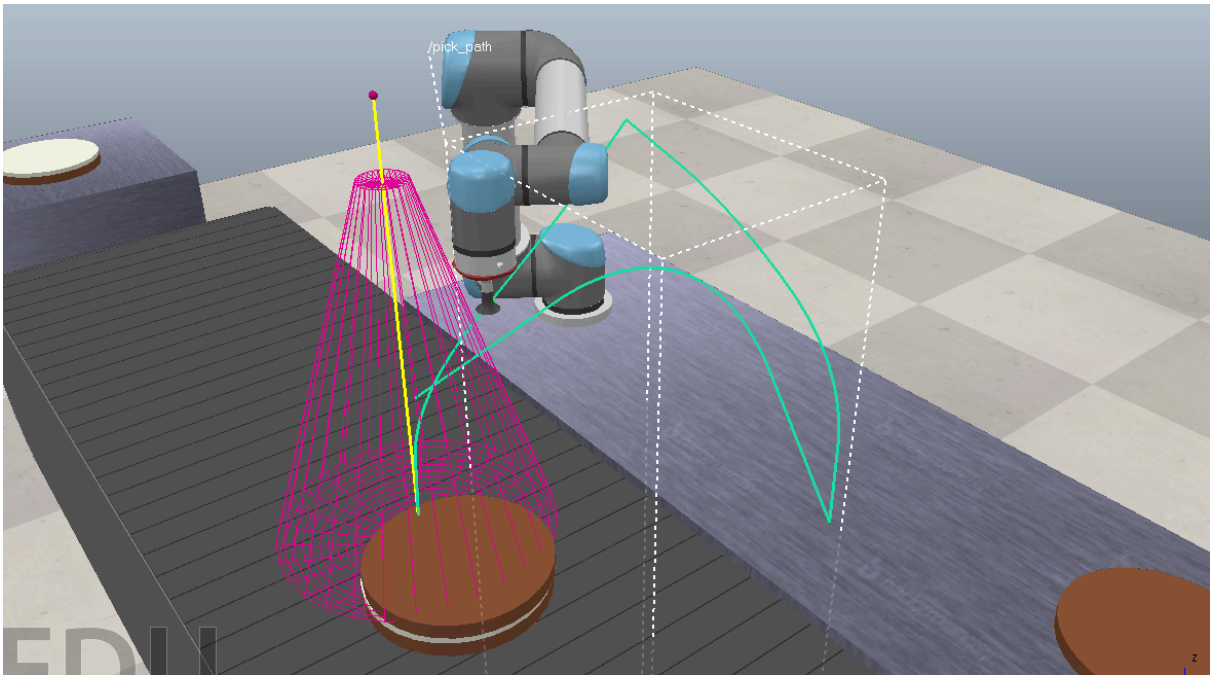
```
    if posAlongPlacePath >= totalLength then
        UR5_vel = 0
        picked = false
        completed = true
        alignment = false
        sim.setObjectParent(spongecake, spongecake_arriving)
        posAlongPlacePath = 0
        place_detected = 0
        delete_old_path(new_place_path)
    else
```

- The movement stops, so `UR5_vel` is set to zero.
- The sponge cake is released by reparenting it from the vacuum cup to the arriving cake object.
- Flags are updated:
    - `picked = false`, since the sponge cake layer has been released.
    - `completed = true`, since the place phase is done, so the cake is completed. This way, in the next actuation call, the system will fall into the return state.
    - `alignment = false`, since the system is ready for next detection and the creation of the new temporary path.

- The `posAlongPlacePath` variable is reset, so that the UR5 starts from the initial point when a new cake arrives.
- The `place_detected` variable is reset, so that the conveyor starts moving again (check the *Conveyor* section for the condition to call the `start_conveyor()` function), since the cake is completed and a new one can arrive.
- The temporary path is deleted using the ***`delete_old_path(new_place_path)`*** function to clean up the simulation.

Return

The ***`return_to_initial_pos()`*** function handles the return phase of the UR5 robot, moving it back to its initial position once the pick-and-place cycle is completed. It ensures that the robot resets properly and prepares the scene for the next iteration by deleting and respawning the sponge cake objects.



The `return_to_initial_pos()` function follows the same motion logic used in both the *pick* and *place* functions. Like them, it performs the movement of the UR5 robot along a predefined path by retrieving the path data, converting into matrices containing positions and quaternion and then by updating the robot's current position along the path (`posAlongReturnPath`). The interpolated position and orientation are then again computed using `sim.getPathInterpolatedConfig()` and applied to the robot's target (`UR5_target`).

The key difference lies in what happens when the robot reaches the end of the path.

In the return function, no manipulation occurs — instead, it performs a **system reset**:

```
192     if posAlongReturnPath >= totalLength then
193     -- Stop the movement
194         UR5_vel = 0
195         completed = false
196         posAlongReturnPath = 0
197         delete_spongecake()
198         spongecake = spawn_spongecake(spongecake_template, "Spongecake_pick", sponge_pose)
199         spongecake_arriving = spawn_spongecake(sponge_arriving_template, "Spongecake_arriving", sponge_arriving_pose)
200     else
```

- The robot stops (`UR5_vel = 0`).
- Control variables are reset to prepare for the next cycle; in particular, the `completed` flag is set to false, to prepare the system for a new cake, and the `posAlongReturnPath` is reset, in order to start again from the initial position of the return path in the next return cycle.
- New sponge cakes are spawned in their starting positions.
- The scene is reset by deleting the previous sponge cakes, calling the ***`delete_spongecake()`*** function.

Simulation loop

The continuous operation of the simulation is achieved through a cyclic process of deleting and respawning the sponge cake models. This mechanism ensures that each pick-and-place cycle can repeat indefinitely without manual intervention. Two specific functions handle the creation and removal of the sponge cakes.

The ***spawn_spongecake()*** function creates a new instance of a sponge cake from a given template using the method `sim.copyPasteObjects({handle}, 0)[1]`.
It then sets the new object's name (`object_name`) and pose (`pose`), effectively placing it in the desired position — either the pick area or the arriving position on the conveyor.
This allows the simulation to regenerate fresh sponge cakes for each new cycle.

```
216 function spawn_spongecake(handle, object_name, pose)
217     new_sponge = sim.copyPasteObjects({handle}, 0)[1]
218     sim.setObjectAlias(new_sponge, object_name)
219     sim.setObjectPose(new_sponge, pose)
220
221     return new_sponge
222 end
223
```

The ***delete_spongecake()*** function simply removes the two sponge cake objects currently present in the scene. It ensures that the workspace is cleared before new instances are created, preventing object accumulation or overlap.

```
247 function delete_spongecake()
248     sim.removeObjects({spongecake, spongecake_arriving}, true)
249 end
250
```

These two functions are called at the end of the ***return_to_initial_pos()*** function:

```
delete_spongecake()
spongecake = spawn_spongecake(spongecake_template, "Spongecake_pick", sponge_pose)
spongecake_arriving = spawn_spongecake(sponge_arriving_template, "Spongecake_arriving", sponge_arriving_pose)
```

This sequence effectively resets the environment, allowing the conveyor and UR5 to start a new pick-and-place operation exactly as in the previous cycle.

Project limitations and future works

Although the simulation successfully demonstrates the automated pick-and-place process, it includes some simplifications that make it only partially realistic when compared to a real-world implementation.

Both during the pick and place phases, the UR5 performs fine positional adjustments by directly retrieving the coordinates of the sponge cake object. This approach works perfectly in simulation, since the software environment allows direct access to every object's position and orientation.

In particular, the alignment correction implemented in the `pick` function adjusts the UR5's position by calculating the exact offset between the Baxter vacuum cup and the sponge cake. Similarly, in the place phase, the final control point of the path is adapted using the precise detected pose of the arriving sponge cake on the conveyor.

However, in a real system, such information would not be directly available. In a real-world application, these positional corrections should be based on sensor feedback rather than perfect positional data. This issue could be solved by adding:

- ***Vision systems*** (e.g., RGB-D cameras or 3D sensors) to locate the object's center in space;
- ***Multiple proximity sensors or encoder-based timing***, to estimate the center position when the object moves on a conveyor;
- ***Visual markers and computer vision algorithms*** for accurate localization.

Another simplification in the current simulation concerns the source of the sponge cake used in the pick phase. In the virtual setup, the sponge cake to be picked is already placed on the table at the start of each cycle, making it immediately available for the UR5 to grasp.

In a realistic production scenario, however, the sponge cake should arrive dynamically from a separate conveyor belt, synchronized with the one carrying the cakes to be completed.
This would require implementing a ***synchronization mechanism*** between the two conveyors, ensuring that:

- The sponge cake arrives at the pick position at the right time;
- The robot performs the pick and place operations within the proper timing of both conveyor movements.