Robotics

link alla presentazione: https://docs.google.com/presentation/d/1JyFYiXMonm4g-7YWPQ-7adeU57kF14Oo8nyBMW297a4/edit?usp=sharing

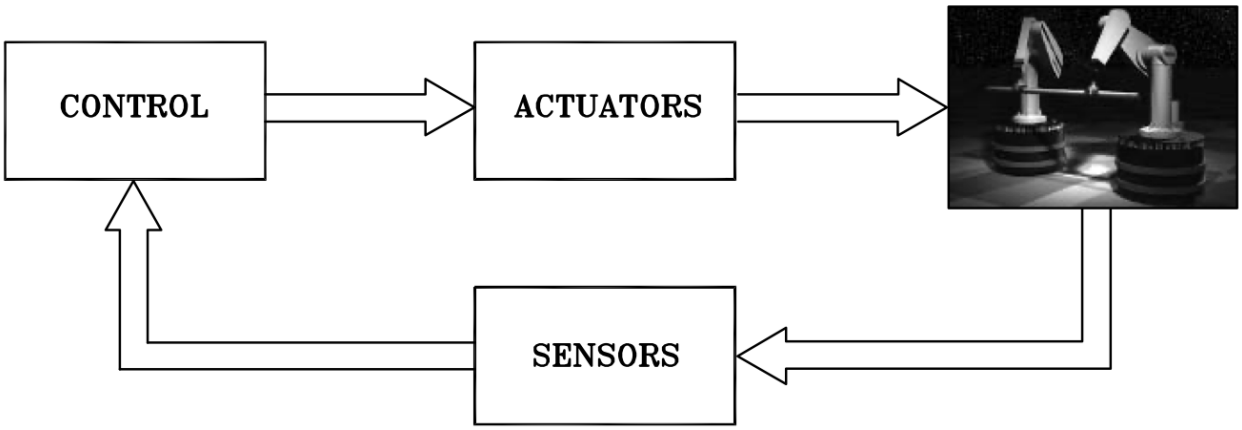
Introduction

Robotics is commonly defined as the science of studying the intelligent connections between perception and action. In this project we experienced both aspect of robotics and connected them to set up a system with the capability of seeing an object and move it, according to the varoius tasks.

It is important to notice that robots are the combination of many different systems, in particular:

- **perception system**, responsible for the acquisition of data useful to decide where to move. It is composed of different sensors, both internal and external. Internal sensors are used to give a sort of proprioception, while external ones are used to collect information about the world around the robot. In our case ZED2 camera was used to collect images and point-clouds of the table and environment under, and around the robot.
- **control system**, responsible for deciding, based on the information given by perception system, where and how to move the joints. Trajectory planning is an important part, whose goal is to generate timing laws for joints, given the description of the desired motion. Modelling the system is highly required to perform this task.
- **actuation system**, responsible for the physical and mechanical motion of the robot. It has to do with mechanical components as servomotors, drivers, transmission and so on. We will not consider these aspect in this project.

These three systems communicate among each other to obtain the final result.



The robot used for the project is classified as a manipulators, indeed it is a sequence of rigid bodies, called links, connected by articulations, called joints. Since there is only a sequence of links between the ends of the chain, it is said an *open kinematic chain*.

In our case all the joints are said to be revolute, since they provide a rotational motion between the links connected, moreover for this reason the arm is said antropomorphic. It is interesting to notice that every joint gives one degree of freedom (DOF) and to position and orient and object, at least six are required. Not by chance UR5 has six joints. In anthropomorfic manipulators, considering the similarities with the human body, the second joint is called shoulder and the third one is called elbow, since it connects forearm (second link) and arm (first link).

In order to control the robot, modelling is highly required since it enables the use of linear algebra. The most important mathematical concept is represented by **rotation matrices**. Their importance is given by the fact that they have three different geometrical meaning; they describe the mutual rotation between coordinate frames, the coordinate trasformation between point in different frames, and also the rotation on vectors in the same frame.

To represent this kind of information, it is possible to use *Euler Angles* in order to decrease the redundancy introduced by rotation matrices, since they are composed of nine parameters highly correlated. If considering planar scenarios, only two parameters are required, the matrix is part of the class of SO(2); if considering a three-dimensional space, three parameters are required, so the class is SO(3). The main combination of angle used is RPY, or better *Roll-Pitch-Yaw*, taking after the aeronautical field. In this representation, only three parameters must be specified to describe the complete rotation, one per axis:

$$\phi = [\varphi \ \ \vartheta \ \ \psi]^T$$

The first parameter represent the rotation around the z-axis (roll), the second one the rotation around the y-axis (pitch) and the third one the rotation around the x-axis (yaw). The resulting rotation matrix is obtained by premultiplication of the matrices obtained for the specified angles, because a fixed frame system has been chosen.

$$R(\phi) = R_z(\varphi)R_y(\vartheta)R_x(\psi)$$
$$= \begin{bmatrix} c_\varphi c_\vartheta & c_\varphi s_\vartheta s_\psi - s_\varphi c_\psi & c_\varphi s_\vartheta c_\psi + s_\varphi s_\psi \\ s_\varphi c_\vartheta & s_\varphi s_\vartheta s_\psi + c_\varphi c_\psi & s_\varphi s_\vartheta c_\psi - c_\varphi s_\psi \\ -s_\vartheta & c_\vartheta s_\psi & c_\vartheta c_\psi \end{bmatrix}$$

It is possible to compute the inverse of this matrix, with different formulas, according to different ranges of ϑ. The only problem is when c_ϑ, since it is not possible to find the correct values of the angles, but only its sum or difference.

Rotations can be espressed in many other ways, for example it is possible to describe a rotation with an angle and an arbitrary rotation axes different from the canonical ones, or even using *unit quaternions*.

Angle-axis representation
Considering $\omega \in \mathbb{R}^3$ to be an axis of rotation with unit magnitude and $\theta \in [0, 2\pi)$ to be an angle, any orientation $R \in SO(3)$ is equivalent to a rotation about the fixed axis ω by an angle θ.
It is possible to obtain ω and θ from R

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad \omega = \frac{1}{2sin\theta} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \quad \theta = cos^{-1}\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right)$$

Homogeneous transformations
Rotation matrices only represents rotations, while it is also useful to consider translations in order to trasform coordinates or vectors in different frames with different origins.
Homogeneous transformations has been introduced for this purpose. It is useful to consider $\tilde{p}$, that is a vector in tridimensional space with a 1 added at the end. The homogenous transformation matrix is:

$$A_1^0 = \begin{bmatrix} R_1^0 & o_1^0 \\ O^T & 1 \end{bmatrix}$$

These are part of a special *Euclidean group* called SE(3). Thanks to this, it is possible to transform a vector in the form of $\tilde{p}$, in another frame, multiplying by the homogenous transformation matrix. The inverse of the matrix is given by the inverse of each component. Differently from rotation matrix, the orthogonality property doesn't hold so the inverse matrix is different from the transposed matrix. Homogenous transofrmation matrices can be composed easily with product operation.

Vision

To properly set the vision mechanisms of the project, we decided to use a Ros service. In services, both a client and a server are involved; when the server is running, it waits for client requests and properly returns a response, without terminating after that, but waiting for other requests. In our project, the server is automatically started when running the robot simulation. It can be disabled from the params.py file. On the other hand, the client script, returning information about the blocks on the table, is automatically started when needed during the execution of the task, generally at the beginning. Yolo, IPC and eutistic strategies are used in the server script in order to succesfully satisfy the vision task.

Yolo - You Only Look Once

Yolo is a tool developed to perform real-time object detection on custom dataset of objects. Object detection means not only classification, but also localization of the subjects. Yolo represents the state of art from 2015, since it is faster than the algorithms used before, such as sliding window object detection, R-CNN, Fast R-CNN and Faster R-CNN.

R-CCN means Region-based Convolutional Neural Network. The first step of this algorithm is finding a manageable number of candidate bounding box, using selective search and looking for areas where pixels are correlated to each other based on a number of factors. The extracted regions are called *Region of Interest* (RoI). Every RoI is cropped with a fixed size and passed into the Convolutional Neural Network to predict the class and improve the boundig box precision. This algorithm is slow and even its evolutions can't be compared with Yolo in terms of speed and precision.

The Yolo approch si completely different, even if it is based on CNN too. Every image is divided into cells, forming a grid. After passing the image in the CCN, every cell will have its own feature vector, forming a feature map. The feature vector contains the coordinates of the bounding-box, the probability that the cell contains an object and the class score, telling the probability of being part of a specific class, for each class present in the system. If more than one are needed to describe a cell, they are easily concateneted, forming a longer vector for each cell. this explains the name of the algorithm, indeed it is necessary to pass

the image through the neural network just one time. To find where objects are, all the cells with an objectness score lower than a chosen trashold are deleted and cells with a class score probability similar are unified to form a bounding box. It is clear that many bounding box will appear around an object, to select the most appropriate and accurate, *NMS algorithm* is used, which means Non-Maxima Suppression. It is also possible to use the Soft-NMS to deal with situations where object with the same class overlap but are distinct.

For our project e decided to use the fifth version and the largest model available in order to increase the precision, given that the blocks are quite similar.
To use YOLOv5 we had to perform a training using a custom dataset: for each image, there is a corresponding line in a text file that describes position and class of the bounding box containing the object to detect.
Clearly, the larger the dataset, the better the result, so we trained the model with a dataset of almost 600 images. After the training, YOLO neural network is ready to be used for the detection of the blocks. We use it to detect the rough position of the objects that will be used to analyze the pointcloud, in the next part of the process.

ICP - Iterativa Closest Point

In order to understand class and orientation of the blocks detected by YOLO, we decided to use the ICP technique. The main difference from YOLO is that ICP uses pointclouds instead of plain
images.

The problem that ICP solves is the alignment of point clouds, without conditions on the dimentions of them. The first step is to connect each point of the transalted point clound, with the nearest point of the original one. The second step is to calculate the center of mass of both, the complete translated point clound and the one formed by the nearest point found in the original one and align them. The third step is consists of finding the better rotation in order to minimaze the distance between the points of both point cloud. These steps can be iterated many times in order to obtain a precise result.
For this project we created pointcloud templates for each object.
Everytime an object is found by YOLO, the corresponding area of the pointcloud is compared with the templates usign IPC algorithm and the one that best satisfies the chosen KPI gives both class and orientation of the block.
The position of the block is obtained by computing the 3d bounding box of the block's pointcloud and calculating its center.

Kinematics

Forward Kinematics

The aim of direct or forward kinematics is to compute the pose of the end-effector as a function of the joint variables. This can be expressed as an homogeneous transformation matrix with:

$$T_e^b(q) = \begin{bmatrix} n_e^b(q) & s_e^b(q) & a_e^b(q) & p_e^b(q) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where **q** is a nx1 vector containing the joint variables, p_e is called *position vector* and connects the base frame with the end effector frame; a_e is the *approach vector*, it is a unit vector with a direction approaching the object; s_e is called *sliding*, it is normal to a_e in the direction of opening of the end effector; n_e is a unit vector, normal to a_e and s_e. To calculate this values it is possible to use a geometric approach, it will be easy if convenient choices are made.

In case of an open chain manipulator, as the one used in the project, fixing a frame attached to each link, the homogeneous transformation matrix is obtained by multiplying all the matrices that transform a frame in one of the following link, as follows:

$$T_n^0(q) = A_1^0(q1)A_2^1(q2)...A_n^{n-1}(q_n)$$

Generally the formula used is:

$$T_b^e(q) = T_0^b T_n^0(q) T_e^n$$

where T_0^b and T_e^n are fixed.

In this kind of operation it is convenient to use conventions in order to simplify the calculation to be made. The most known convention is *Denavit-Hartenberg* and it is used to define the relative position and orientation of consecutive links.

Inverse Kinematics

The inverse kinematics problem consists of the determination of the joint variables corresponding to a given end-effector position and orientation. The solution to this problem is of fundamental importance in order to transform the motion specifications, assigned to the end-effector in the operational space, into the corresponding joint space motions that allow execution of the desired motion. Unlike direct kinematics, the inverse kinematic

problem is much more complex. It may have multiple, infinite or no admissible solutions and it is not always possible to find a *closed-form* solution as the equations are in general non linear, and when it is, they require good algebraic or geometric intuitions.

For our UR5 manipulator, a closed-form solution exists, and for a given pose, there exist in general 8 possible solutions, corrisponding to shoulder left/right, elbow up/down and wrist up/down.

It is possible to use this solution in the motion control algorithm, planning a path in the operational space and invert every point to obtain the corrisponding joint positions, but we opted for another strategy.

Differential Kinematics

Differential kinematics gives the relationship between the joint velocities $\dot{q}$ and the corresponding end-effector linear and angular velocity, respectively $\dot{\mathbf{p}}_\mathbf{e}$ and ω. This mapping is described by a matrix, termed **geometric Jacobian**, which depends on the manipulator configuration q.

Jacobian matrix

The Jacobian matrix is a linear approximation of the forward kinematics of our robotic manipulator at the current configuration. It gives an approximate change in end effector pose for a change in joint angle configuration.

$$v_e = \begin{bmatrix} \dot{\mathbf{p}}_\mathbf{e} \\ \omega \end{bmatrix} \dot{q} = \begin{bmatrix} J_p(q) \\ J_\omega(q) \end{bmatrix} \dot{q} = J(q)\dot{q}$$

where v_e is the $r \times 1$ vector of end-effector velocity (r is the number of operational space variables necessary to specify a given task), $\dot{q}$ is the $n \times 1$ vector of joint velocities (n is the number of DoFs of the structure), and J is the $r \times n$ Jacobian matrix.
In our case, $r = n = 6$ so J is square.

The Jacobian matrix has two components, one for the linear velocity J_p and one for the angular velocity J_ω.
The contribution to both linear and angular velocity from each joint is linear, i.e. we can solve for the contribution from each joint and take the entire sum as the linear and angular velocity of the end effector. Each term represents the contribution of a single joint i to the end-effector velocity when all the other joints are still.
The contribution for the linear velocity from joint i depends on the axis of rotation z_i, the

end effector position p_e, the joint position p_i, and the rate of change in angle $\dot{\theta}_i$ (for revolute joints $q_i = \theta_i$).
The axis of rotation is the *z-axis* since we are following the Denavit-Hartenberg convention.

$$J_p = \sum_{i=1}^{n} [z_{i-1} \times (p_e - p_{i-1})] \qquad J_w = \sum_{i=1}^{n} z_{i-1}$$

Inverse Differential Kinematics
Now that we can calculate the Jacobian matrix, we need to solve the IK problem for the change in joint angles which achieves that velocity.

$$\dot{q} = J^{-1}(q) v_e$$

which can only be solved if $J(q)$ is non-singular.

If the initial manipulator posture $q(0)$ is known, joint positions can be computed by integrating velocities over time. The integration can be performed in discrete time by the use of numerical techniques.
The simplest technique is based on the Euler integration method: given an integration interval Δt, if the joint positions and velocities at time t_k are known, the joint positions at time $t_{k+1} = t_k + \Delta t$ can be computed as

$$q(t_{k+1}) = q(t_k) + \dot{q}(t_k)\Delta t = q(t_k) + J^{-1}(q(t_k)) v_e \, \Delta t$$

Inverse Kinematics algorithms
It is useful to define an error vector which represent the translation and rotation “distance” from the end-effector’s current pose T_e to the desired pose T_d:

$$\mathbf{e} = \begin{pmatrix} e_p \\ e_o \end{pmatrix} \in \mathbb{R}^6$$

where the top three rows correspond to the translational error in the world frame, the bottom three rows correspond to the rotational error in the world frame.

While the error in the position is pretty straightforward, as

$$e_p = p_d - p_e$$

The rotational error is trickier.
Given R_d and R_e, the rotation matrices that represent the desired and current orientation

of the end-effector w.r.t. the world frame, we can define R_d^e

$$R_d = R_e R_d^e \rightarrow R_d^e = R_e^{-1} R_d$$

as the rotation to be applied to the current orientation R_e to obtain the desired orientation R_d.
It is helpful to change the representation of the matrix R_d^e using, for example, the angle-axis representation. Once θ and ω are defined, we can use the vector $\theta\omega$ but since it is expressed in the end-effector frame, we need to rotate it in order to be expressed in the world frame, the same as the Jacobian and end-effector position are

$$e_o = R_e \theta \omega$$

A possible algorithm takes the error term $\mathbf{e}$ to set v_e in the inverse differential equation at each time step

$$v_e = K\mathbf{e}$$

where K is a proportional gain term which controls the rate of convergence to the goal, typically a diagonal matrix to set gains for each task-space DoF. This control scheme will cause the error to asymptotically decrease to zero.
Using this method we can get the end-effector to travel in a straight line, in the robot's task space, towards some desired end-effector pose.

Another algorithm considers the equation

$$v_e = v_d + K\mathbf{e}$$

that compared to the previous one, adds a feedforward term. The *feedforward* action provided by v_d injects information about the intended movement. Feedforward handles parts of the control actions we already know must be applied to make the system track a reference.

Our implementation
We implemented an algorithm that is similar to the first explained but with a small but important modification. We know that the Jacobian matrix is a linear approximation of the forward kinematics of our robotic manipulator at the current configuration. It gives an approximate change in end-effector pose for a change in joint angle configuration

$$\Delta S \approx J \Delta \theta$$

Based on the Jacobian inverse method, we solve for some $\Delta\theta$ which will move the end-effector along the desired velocity.
As this assumption relies on a linear approximation, it fails for large s or θ. We must scale the step size down to an appropriate level so our assumption approximately holds.

We have to be careful as some nearby points in Cartesian space will be further away in joint space than others.
For this reason, we directly scale the change in joint angles. We set some $\Delta\theta_{max}$ and then multiplicatively scale the maximum $\dot{q}$ to that value. We also take care to check if the maxmimum value in $\dot{q}$ is less than $\Delta\theta_{max}$ to prevent oscillation problems (overshooting).

$$\dot{q} = \dot{q}\frac{\Delta\theta_{max}}{max(\dot{q})}$$

This is equivalent to solving the desired joint velocities, running the manipulator on those velocities for a short while, before calculating the next joint velocities to maintain the desired trajectory.

The control algorithm must check for convergence. The first idea of a convergence check is to calculate the distance between the current pose and desired pose, stopping if it is below some threshold. This has the flaw that some desired poses may be unreachable. The control algorithm in this case will oscillate around some minimum distance point to the desired pose, never terminating.
A more desirable convergence check is to instead compute the difference between a previous and current pose. This solves the oscillation problem and causes the control algorithm to terminate either when the desired pose is reached or when it gets as close as possible to the desired pose.

Kinematic singularities
The Jacobian is, in general, a function of the configuration q; those configurations at which J is rank-deficient are termed *kinematic singularities*. To find the singularities of a manipulator is of great interest for the following reasons:
- Singularities represent configurations at which mobility of the structure is reduced, i.e., it is not possible to impose an arbitrary motion to the end-effector
- When the structure is at a singularity, infinite solutions to the inverse kinematics problem may exist
- In the neighbourhood of a singularity, small velocities in the operational space may cause large velocities in the joint space

Internal singularities occur inside the reachable workspace and are generally caused by the alignment of two or more axes of motion, or else by the attainment of particular end-effector configurations. These singularities constitute a serious problem, as they can be encountered anywhere in the reachable workspace for a planned path in the operational space.

The inverse differential equation can be computed only when the Jacobian has full rank. Hence, it becomes meaningless when the manipulator is at a singular configuration; in such a case, the system $v_e = J\dot{q}$ contains linearly dependent equations.
It is important to underline that the inversion of the Jacobian can represent a serious inconvenience not only at a singularity but also in the neighbourhood of a singularity.

A rigorous analysis of the solution features in the neighbourhood of singular configurations can be developed by resorting to the singular value decomposition (SVD) of matrix J. It is possible to check the *minimum singular value* of the matrix before the inversion in order to understand how well-conditioned it is. If the msv is above a certain threshold, we invert the Jacobian, otherwhise we need to use a different method. One of these is the so called *damped least-square inverse*.

$$J^* = J^T(JJ^T + k^2I)^{-1}$$

where k is a damping factor that renders the inversion better conditioned. The large joint angle changes required to move in near-singular directions are penalized to favor the smaller changes necessary to move in other directions, trading a small off-track motion with bounded velocities.

In our case, we have to deal with a shoulder singularity, that is when the wrist point lies on or is in the neighbourhood of axis $z0$. This situation can happen quite often as the base of the robot is placed well inside our workspace and passing through this area can happen easily. It is of great importance to deal with this type of situations during the planning part.

Trajectoy planning

Path and trajectory

The minimal requirement for a manipulator is the capability to move from an initial posture to a final assigned posture. The transition should be characterized by motion laws requiring the actuators to exert joint generalized forces which do not violate the saturation limits.

A **path** denotes the locus of points in the joint space, or in the operational space, which the manipulator has to follow in the execution of the assigned motion; a path is then a pure

geometric description of motion. On the other hand, a **trajectory** is a path on which a timing law is specified, for instance in terms of velocities and/or accelerations at each point.

Path primitives
Our path primitives for the position are of two types: straight line and Bezier curve. The line path is a simple interpolation between the start and end point. We employed the line segments for short paths, including rotational only movements, and to handle the approach and landing phase to the blocks. The Bezier curve is a cubic Bezier curve and is used in the longer paths and it is calibrated through the control points to avoid passing into the singularity zone, for example when going from one side of the table to the other.
The curves are both defined in the operational space and a constant step ds of 5cm is used to generate the points.

For what concerns the orientation, it is specified in terms of the rotation matrix of the end-effector frame with respect to the base frame. To generate a trajectory, however, a linear interpolation on the unit vectors of the matrix describing the initial and final orientation does not guarantee orthonormality at each step.
For this reason, we implemented a slerp, spherical linear interpolation. It describes an interpolation (with constant angular velocity) along the shortest path (a.k.a. geodesic) on the unit hypersphere between two quaternions. It is then just necessary to transform the initial and final rotation matrices into quaternions and then the resulting quaternion slerp back to rotation matrices.
At the beginning, we have initial and final orientations q_i and q_f expressed as quaternions after a simple transformation from the rotation matrices. We know that $q_f = q_i * q$, where q is the rotation that brings q_i to q_f, then $q = q_i^{-1} * q_f$. From q, we extract the information about the angle of rotation as $cos(\theta) = real(q)$ and then

$$q_m = \frac{q_i sin((1-t)\theta) + q_f sin(t\theta)}{sin\theta} \qquad \text{where } t \in [0,1]$$

In order to generate the final trajectory, once the path is defined, it is necessary to define a timing law. We implemented our trajectories with constant speed and angular velocity.

Task management

Task description

The main goal of the project is to move different-shaped blocks placed on the working area from a random initial configuration to a final known configuration. The blocks can lay on every flat face and upside down and they have to be placed in their “natural” posture. This

involves simple pick-and-place operations, as well as a sequence of steps in order to rotate the block in the final configuration.

One of the tasks, the most complete and complex as it includes the other ones, consists in building a pre-defined “castle” of blocks.

Finite State Machine

The whole procedure can be seen as a finite state machine that process the blocks, one by one, to achive the final goal.

The first part involves a vision system, consisting of a ZED2 depth-camera, that is employed to find position, orientation and classify the blocks that are placed on the working area and have to be processed.

Then, each block, if not already, is rotated in its natural configuration and placed in the final position in the castle, starting from the ones in the bottom layer of the castle and going upwards. To rotate the blocks, the arm approaches at 45° and then operates a rotation of 90° to complete the operation. This was necessary as the manipulator can not grasp the blocks directly at 90° because it crushes on the table (the blocks are too small) and it is not possible to use the border of the table since the arm can not reach that position.

Handling special cases

Initially, the blocks are spawned randomly across the working area. It is not possible, though, to place them right underneath the shoulder of the manipulator and in an area that is approximately 12 cm in radius from it as the arm can not reach those positions due to the shoulder singularity.
In addition to this, we defined a second restricted area of radius of about 30 cm because when we tilt the end-effector at 45° and the arm bends towards the inside of this area, the manipulator is still very close to a singular configuration. For this reason, in this area, we only apporach blocks with the gripper perpendicular to the table and we move the block outside of this area (finding a free spot distant enough form other blocks) and then process it.
This procedure is executed also when the blocks are near the bottom of the table (where the box of the electronics is) as when the arm bends at 45°, it might collide with the box.