# Task F

DeNiro is a humanoid robot and each of his arms has 7 DOF just like a human arm. Performing movements in space never requires more than 6 DOF. Therefore, with at least one additional DOF than required, when performing movement in space, **kinematic redundancy** exists. Additionally, when moving between points in space, such as the circular path in Task E, infinitely many velocity profiles could be used. This exemplifies **trajectory redundancy**.

How these **redundancies** are solved has a major impact on the optimality of the motion. Alternatively, by adding a secondary motion, the redundancy can be resolved. To do this a **Null Space Projection Method** can be used. This will ensure that the redundancy resolution will be within the a set of values which do not impact the end effectors position. This set of values is called the null space. Thus, joint velocities , which enable Task 2, are added to joint velocities which enable Task 1 without compromising the task.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

In our case, we want to keep the end-effector stationary whilst avoiding the red ball which appears by the elbow joint. To do this, we can find a factor N which can guarantee that the resultant of will be in the Null Space. This factor, known as the **Null Space Projector,** can be seen as a map, transforming a motion to be within the null space . The null space is additive, thus for the particular solution which holds the end effector stationary, adding the product has no effect.

|  |  |  |
| --- | --- | --- |
|  |  |  |

# Task G i

The VelocityController class creates variable joint\_des within the constructor. This variable is a hardcoded NumPy array which stores a joint space where DeNiro’s arm and the red ball do not collide. This variable is accessed using self.joint\_des and stored as new variable q\_desired within the end effector’s inverse kinematics method (ee\_IK\_solver).

With the desired joint space accessed, and the current joint space known, the change in time can be used to define the incremental changes in joint velocity towards q\_desired.

|  |  |  |
| --- | --- | --- |
|  |  |  |

These are then slowed for visualization using a scaling factor. Lastly, equation is applied to add the motion to whilst ensuring the motion occurs within the null space.

Running the task with ‘python velocity\_controller.py nullspace’ it is clear to see the arm moving out of contact with the ball whilst the white brick stays in position.

Applications of redundancy resolution, mentioned within Task F, can allow motions to be optimised. Different objectives such as minimised effort (power) or maximise manipulability can be solved. One option uses the pseudo inverse of the Jacobian to minimise actuation velocity by minimising the sum of squared joint velocities. This, in turn, minimises energy consumption.

There are many other wise to resolve redundancies such as gradient descent. Here, the factor to be optimised is chosen by the robot operator (*or the robot!*) as a cost function which has a cost function landscape composed of gradients. This landscape can be moved across by altering joint velocities. By moving along the steepest path, a local minimum can be found. While this local minimum is not guaranteed to be optimal, strategies such as choosing multiple start values or altering the gradient descent algorithm can improve the chances of finding a global minimum. This all depends on correctly defining the cost function.

# Task G ii

Task G ii involves generating finding using a method which ensures that only the joints up until the elbow joint moves. This first requires the Jacobian of the elbow to be calculated. This is done within the link\_jacobian method. The following variables are created:

1. q\_elbow - The points of the current joint values up to the elbow joint.
2. nj - The number of joints to the elbow are counted.
3. nj\_tot - The total number of joints are saved.
4. jacobian – Uses Python’s Kinematics and Dynamics library (PyKDL) to set up a PyKDL variable the right size for the elbow
5. q\_kdl - A joint array the right size for the number of joints to the elbow is created with PyKDL. The joint array is populated with values from the current joint values to the elbow.

Having found these variables, we can access another function from KDL called ChainJntToJacSolver which was saved as \_jac\_kdl\_link within VelocityController’s constructor. This takes the q\_kdl and jacobian variables to output a populated jacobian which can then be converted into a NumPy array.

The attained matrix is a 6 x 4 because it only includes joints up to the elbow. However, it must be multiplied by the **Null Space Projector** from equation . This projector is obtained from the Jacobian and thus a square matrix of size DOF x DOF which in our case is 7 x 7. As a result, to calculate , must be 6 x 7. As such, we must pad out our array with zeros. This is done using the NumPy.pad() method. As a result, applying the padded matrix will make the elbow act as the end effector. All joints past the elbow will experience the same angular and linear velocity as the elbow just as an object held by the end effect have the same velocity.

Finally, because only the translational linear components of the jacobian are required – just the first three rows of the matrix are returned.

The pseudo-inverse of this 7 x 3 matrix multiplied by vel\_elbow (a circular velocity defined for the elbow) to give where are non-zero integers. The latter four joint velocities are zero because we are trying to move the elbow out of contact with the ball, without changing any other joint velocities. Then, by multiplying and inserting to equation

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  | are non-zero integers and is within the null space |  |
|  |  |  |

Therefore, finding with only joints up to the elbow moving (G ii) or all the joints moving (G i), the result can require motion in all joints.