Task priority assignment with collision avoidance

Stefano De Filippis & Marco Menchetti

Sapienza - University of Rome

Redundancy resolution problem

Find Robot command in order to execute a series of tasks. The problem can be formalized as:

$$Ax = b$$

with $A = \begin{bmatrix} A_1^T & A_2^T & \dots & A_l^T \end{bmatrix}$ and $b = \begin{bmatrix} b_1^T & b_2^T & \dots & b_l^T \end{bmatrix}$.

$$\sum_{k=1}^{l} m_k \leq n$$

How to solve redundancy?

The non prioritized solution is $\overline{x} = A^{\#} b$ if tasks not l.i. we need to accommodate conflicting tasks

1. Siciliano and Slotine approach: task projection in null space of higher order task
2. Flacco Matrix: separation of redundancy resolution from assignment of correct order

Flacco Matrix

From base solution we can extrapolate:

- contribution of each task
- task null space

useful information for reordering.

IDEA: find a matrix F that allows us to get these information and impose correct priority

$$x = A^{\#} F b$$

RESULTS: applying F we should get same solution as Siciliano and Slotine.

What is the structure of F?

Generally F has the following structure:

- It is block lower triangular
- It has I on diagonal if task l.i. to higher priority task
- It has 0 blocks on diagonal if task l.d. to higher priority tasks
- It has coefficient of dependency in the left side of rows

How to compute F and final solution?

We can use Gauss Jordan elimination with pivot square matrices

Algorithm

1: Use QR decomposition on A
$\rightarrow A^{\#} = QR^{-T}$
2: Initialize F
3: **for all** row_j **do**
4: $\quad row_j \leftarrow P^{\#} * row_j$
5: $\quad row_i \leftarrow row_i - block_{ij} * row_j$
$(i < j)$
6: **end for**
7: $F \leftarrow F^T$
8: $x \leftarrow (QR^{-T} * F * b)$

$$\overline{F} = \begin{array}{c c} & \begin{array}{cccc} m_1 & m_2 & & m_l \end{array} \\ \begin{array}{c} m_1 \\ m_2 \\ \\ m_l \end{array} & \begin{pmatrix} R_{11} & \star & \dots & \star \\ 0 & R_{22} & \dots & \star \\ 0 & 0 & \dots & \star \\ 0 & 0 & \dots & R_{ll} \end{pmatrix} \end{array}$$

How to compute F: code

Code

```
while(i < m){
        /* In j I keep the index of the final row and column of the current task I am working on
*/
        j = i + tasksDim(i_taskDim) - 1;
        rows = tasksDim(i_taskDim);
        col = tasksDim(i_taskDim);
        /* I compute the pseudoinverse of the pivot block matrix of the corresponding to the current task I am working on*/
        MatrixXf pR = damped_pinv(bF.block(i,i,rows,col),lam,eps);
        last = m-i;
        /* I execute the first step of the Gauss Jordan elimination in order to try to have an identity matrix as a pivot block*/
        bF.block(i,i,rows,last) = pR * bF.block(i,i,rows,last);
        /* I execute the second step pf the Gauss Jordan elimination in order to try to nullify the block corrisponding to same block column as the current task
           but preceding block rows*/
        bF.block(0,i,i,last) = bF.block(0,i,i,last) - bF.block(0,i,i,col) * bF.block(i,i,rows,last);
        i = j + 1;
        i_taskDim = i_taskDim + 1;
}
```

Why priority?

- Decomposition of problems in many tasks.
- Most problems can't be solved by just one task.
- Error is kept on the tasks that can't be executed **EXACTLY**
- More natural and smoother behavior.

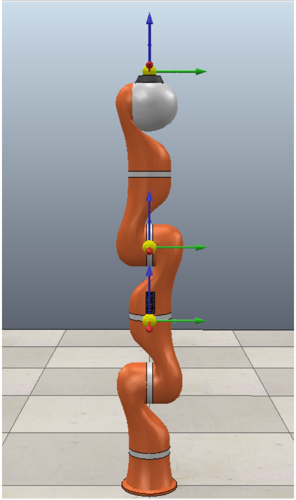
Collision avoidance. How?

1. **Control points**

Distributing a certain amount of points along the structure of the manipulator we can keep track of its distance from the obstacles and how it changes with respect to the pose.

2. **Collision avoidance**

Whenever a control point is within a certain radius from the surface of the obstacles we start to push it away until we move in a "safe zone".

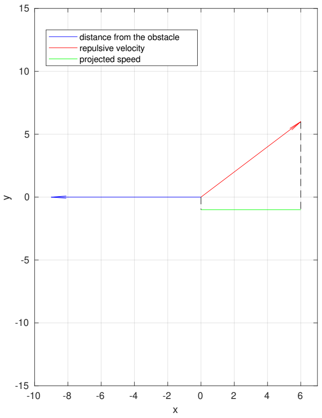


Collision avoidance. How?

2. How do we push?

We change approach whether the control point is on the e-e or on the structure:

- For the end-effector we add to the task velocity, a cartesian velocity pointing away from the obstacle.
- For the ones on the structure we add the projection of a cartesian velocity, along the distance of the control point from the obstacle (1 DOF).



Tasks

We know why to prioritize Tasks, but which are the ones we are going to use?

1. A cartesian positioning task (i.e. we want our e-e to behave in a certain way)
   - **3 DOFs**
2. An orientation task used to simulate any kind of auxiliary task
   - **1 DOFs**

3,4 Two collision avoidance task, each one on 1 DOF
   - **2 DOFs**

In the end we saturated 6 out of all the 7 DOFs of the manipulator.

Task 1: Cartesian positioning

Cartesian positioning means we want the end effector to execute a given trajectory in $\mathbb{R}^3$.

Path used:

- A linear path
- A point-to-point motion path

The associated jacobian J_1 is the analytical jacobian of the direct kinematics.

Task 1: Collision avoidance

- We have **4** tasks occupying **6** DOF and we can't another cartesian positioning task. How is it performed?

IDEA:

Add another repulsive velocity to the desired one, pointing away from the obstacle!

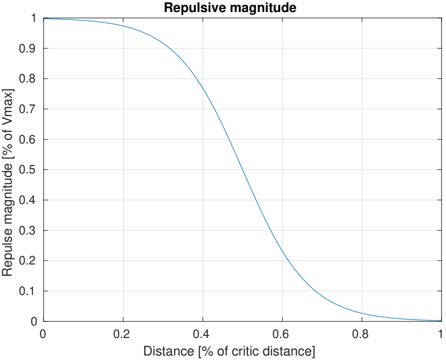
- In this way the *Flacco Matrix* will handle the exact joint velocities so as to execute the sum of the two.
- This won't change the jacobian of the task.

Repulsive velocity

How do we choose?

We want the repulsive velocity to satisfy a certain amount of properties:

- Maximum admissible cartesian velocity at distance $d = 0$ from the obstacle $\rightarrow V_{max}$.
- Smooth descending curve $\rightarrow \alpha$.
- Zero velocity after a given distance from the obstacle $\rightarrow \rho$.



Hence:

$$v(P, O) = \frac{V_{max}}{1 + e^{(\|D(P,O)\|(2/\rho)-1)\alpha}}$$

Task 2: Link orientation

The "orientation task" tries to keep constant the elevation of the third link axis. We need to define it as a vector in $\mathbb{R}^3$:

$$p_l(q) = p_5(q) - p_4(q)$$

Applying a coordinate transformation into spherical ones we can easily get the expression of the elevation (dropping the dependencies on q):

$$\phi = \arccos\left(\frac{p_{l,z}}{\parallel p_l \parallel}\right)$$

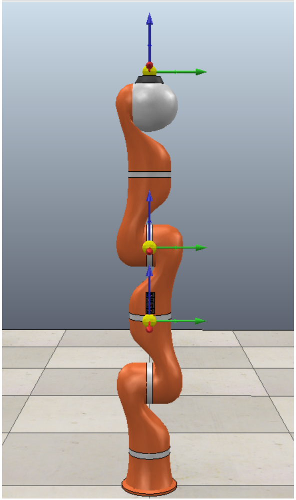
Denoting $p = \frac{p_{l,z}}{\|p_l\|}$ we can get the expression of the associated jacobian as:

$$\dot{\phi} = \frac{\partial \phi}{\partial p}\frac{\partial p}{\partial q}\dot{q}$$

Tasks 3,4: Control points

We chose the following:

- The end-effector
- The origin of the DH reference frame associated with the 4^{th} joint
- The third is on the second link axis at a given distance from the origin of the DH reference frame associated to the 2^{nd} joint (i.e. half link axis' length off the shoulder)



Tasks 3,4: Collision avoidance

As already introduced we can't perform collision avoidance for the control points using all the three component of the distance vector. So what?

PROJECT!

We can project the same repulsive velocity we used on task 1, on the distance from the obstacle, obtaining a "repulsive speed" we will call v.

$$v = \eta^T \dot{r_o} = \eta^T J_i \dot{q} = J_{c,i} \dot{q}$$

J_i is the jacobian at the i-th control point.

Reordering

Stack

We organized the tasks in a vector (here `stack`) where the first element is the lower priority one and the last is the higher priority one.

This is divided in two parts:

1. The first part contains the task which are defined *critic*.
2. The second one contains the tasks whose cost is high enough to be in a safe position.

Criticality

In order to evaluate which position a task should take within the `stack`, we need a method to compute a "general cost".

Reordering: cost & execution

- In this application almost each task is associated with a control point so an easy cost function is the minimum distance of the points from the obstacles within the workspace.
- The second task is not easy to associate a control point with, so we assigned to it a constant cost which is high enough to never make it critical.

The execution of the reordering algorithm is performed as such:

Algorithm

```
1: for all non critic tasks do
2:     reorder by cost
3:     if any task cost ≤ critic distance then
4:         augment number of critic tasks
5:     end if
6: end for
7: for all critic tasks do
8:     reorder by cost
9: end for
```

Reordering: code

Code

```
/*REORDERING*/
int initial{danger+1}, final{sizeMax};
// danger+1 is the number of tasks w/ priority lower than the cartesian task
// sizeMax is the total size of the stack
for (int j = 0; j < 2; ++j) {
        // first iteration is for the relaxed sub-vector
        // second one is for the critic sub-vector
        //
        // After swapping the first, if there is any critic situation, we will
        // augment the length of the critic sub-vector and reorder that, knowing that the added components
        // are actually critic ones.
        for (int i = initial; i < final; ++i) {
            float min{distT[i]};
            int minK{i};
                for (int k = i; k < final; ++k) {
                // Find the minimum
                    if(distT[k] < min) {
                        min = distT[k];
                        minK = k;
                    }
                }
                // Replace the minimum
                if(min < distance_warning) {
                        distT.goUpTo(minK, i);
                        switched = true;
                } //else switched = false;
        // update only if it is in the first iteration on j
        // i.e. if we are sorting the non critical vector
                danger += distT[i] < distance_critic && j == 0;
        }
        // reset the values so that we sort from the beginning up to danger
        // that means we include also the critic tasks coming from the non critical part
        initial = 0;
        final = danger;
}
```

Results: angular velocity interesting cases

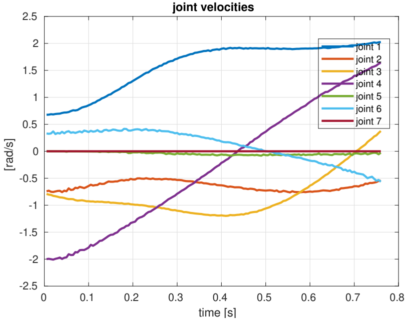


Figure: obstacle: **far**, control points not always active

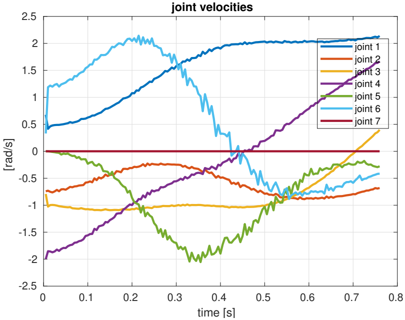


Figure: obstacle: **far**, control points always active

Results: angular velocity interesting cases

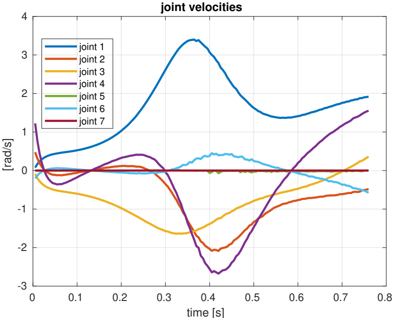


Figure: obstacle: **on**, control points not always active

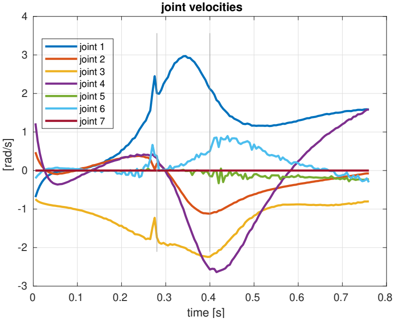


Figure: obstacle: **on**, control points always active

Results: ee task error in "obstacle on path" case

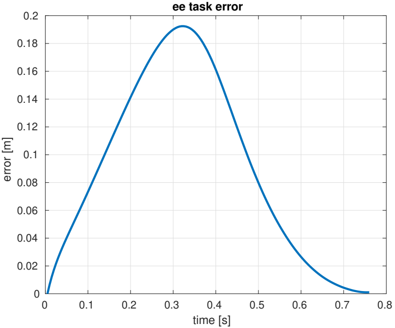


Figure: obstacle: **on**, control points not always active

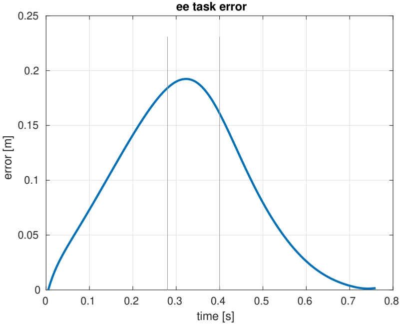


Figure: obstacle: **on**, control points always active

Results: cluttered environment

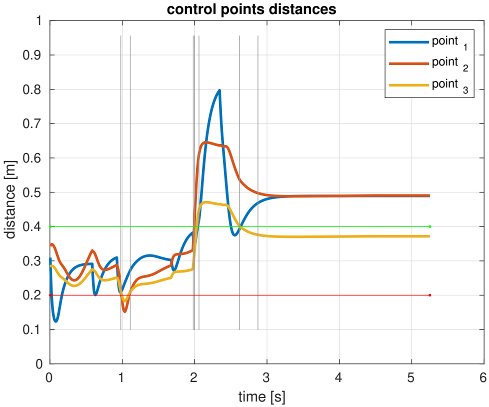


Figure: Control points' distances from 1st obstacle

Results: cluttered environment

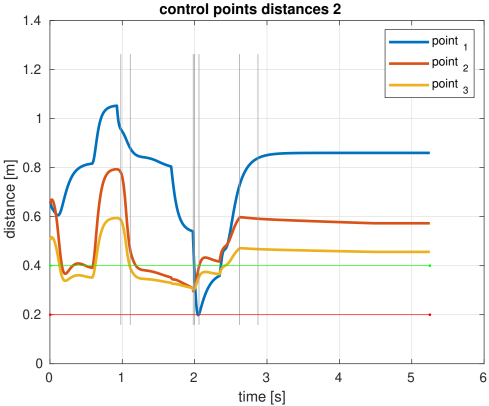


Figure: Control points' distances from 2^{nd} obstacle

Results: cluttered environment

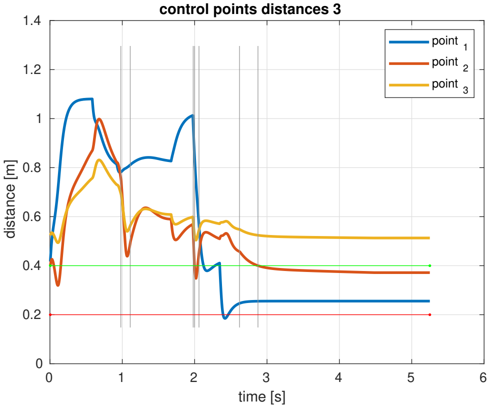


Figure: Control points' distances from 3^{rd} obstacle

Results: cluttered environment

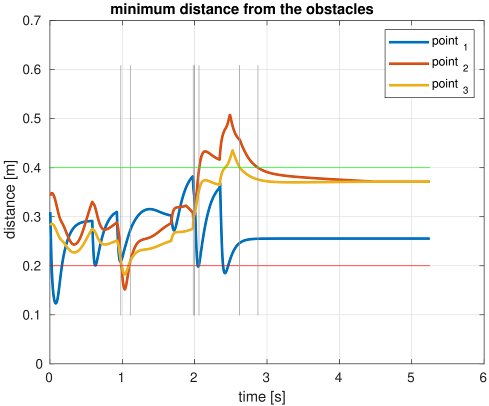


Figure: Minimum distance among all the obstacle