Institute for Cognitive Systems
Technische Universität München
M.Sc. Simon Armleder

Modeling and control of legged robots
Tutorial 4: Whole-body control

1 Introduction

In this tutorial we will explore whole body control of humanoid robot based on the Task Space Inverse Dynamics formulation (TSID).
When multiple tasks have been assigned to a robot and can not be realized at the same time, a trade needs to be made among them. This problem is known as **Redundancy Resolution**.
Let a task be represented by a task variable $\mathbf{y}(\mathbf{q}) \in \mathbb{R}^m$ such as e.g. position, orientation, joint pose, Cartesian pose, etc. The performance of a controller acting in that space is measured w.r.t some reference by the error $\mathbf{e} = \mathbf{y}(\mathbf{q}) - \mathbf{y}_\mathsf{d}(t)$, where $\mathbf{y}_\mathsf{d}(t)$ is some desired signal. Since the control input of the robotic system is typically assumed on torque level $\boldsymbol{\tau}$ we can only instantaneously influence joint accelerations $\ddot{\mathbf{q}}$. Therefore, taking the second-order time derivative of our task variable gives us the linear relation

$$\mathbf{J}\ddot{\mathbf{q}} + \dot{\mathbf{J}}\dot{\mathbf{q}} = \ddot{\mathbf{y}} \tag{1}$$

Here $\mathbf{J} = \partial\mathbf{y}/\partial\mathbf{q}$ is the differential mapping (Jacobian) that relates task and joint space.
Now, to steer the error $\mathbf{e}(t)$ towards zero, lets impose the second order linear dynamics

$$\ddot{\mathbf{e}} = -\mathbf{K}\mathbf{e} - \mathbf{D}\dot{\mathbf{e}} \tag{2}$$

where $\mathbf{K}, \mathbf{D} \in \mathbb{R}^{m\times m}$ are the controllers feedback gains. Plugging Eq. (1) into this dynamics gives us

$$\mathbf{J}\ddot{\mathbf{q}} + \dot{\mathbf{J}}\dot{\mathbf{q}} - \ddot{\mathbf{y}}_d = -\mathbf{K}\mathbf{e} - \mathbf{D}\dot{\mathbf{e}} \tag{3}$$

$$\mathbf{J}\ddot{\mathbf{q}} = \ddot{\mathbf{y}}_\mathsf{d} - \mathbf{K}\mathbf{e} - \mathbf{D}\dot{\mathbf{e}} - \dot{\mathbf{J}}\dot{\mathbf{q}} \tag{4}$$

$$\mathbf{J}\ddot{\mathbf{q}} = \mathbf{a} \tag{5}$$

So, the problem of minimizing the task error becomes affine when expressed in terms of robot accelerations. Since this system is subject to noise an exact solution to Eq. (5) rarely exists. Instead, we try to minimize the error as best as possible using a quadratic loss

$$\min_{\ddot{\mathbf{q}}} \|\mathbf{J}\ddot{\mathbf{q}} - \mathbf{a}\|_2^2 \tag{6}$$

Now, of cause, the robot can not produce arbitrary accelerations but instead is subject to its own dynamics

$$\min_{\boldsymbol{\tau},\ddot{\mathbf{q}}} \|\mathbf{A}\ddot{\mathbf{q}} - \mathbf{a}\|_2^2 \quad \text{s.t.} \quad \mathbf{M}\ddot{\mathbf{q}} + \mathbf{h} = \mathbf{S}^\mathsf{T}\boldsymbol{\tau} \tag{7}$$

here $\mathbf{A}$, $\mathbf{a}$ are called the task Matrix and task vector respectively.
Embedding multiple tasks can be simply achieved by adding more than one objective into the controller formulation. This is called **weighted approach** and leads to

$$\min_{\mathbf{y}=[\dot{\mathbf{v}},\boldsymbol{\tau}]} \sum_k w_k \|\mathbf{A}_k\mathbf{y} - \mathbf{a}_k\|_2^2 \quad \text{s.t.} \begin{bmatrix}\mathbf{M} & -\mathbf{S}^\mathsf{T}\end{bmatrix}\mathbf{y} = -\mathbf{h} \tag{8}$$

where w_k are user defined weights that determine the importance of tasks.

The final modification to this control formulation appears when external contacts are presents. If the contacts cannot be violated they need to be included as a constraint that eliminates any acceleration at the contact. Given the known position of a contact in the kinematic chain of the robot, the second order time derivative of the contact position leads to

$$\mathbf{J}\dot{\mathbf{v}} + \dot{\mathbf{J}}\mathbf{v} = \mathbf{0} \tag{9}$$

where $\mathbf{J}$ is also called the contact Jacobian.
Adding in these additional contact constraints gives the final control formulation:

$$\min_{\mathbf{y}=[\dot{\mathbf{v}},\mathbf{W},\boldsymbol{\tau}]} \sum_k w_k \left\|\mathbf{A}_k\mathbf{y} - \mathbf{a}_k\right\|_2^2 \quad \text{s.t.} \begin{bmatrix} \mathbf{J} & \mathbf{0} & \mathbf{0} \\ \mathbf{M} & -\mathbf{J}^\mathsf{T} & -\mathbf{S}^\mathsf{T} \end{bmatrix} \mathbf{y} = \begin{bmatrix} -\dot{\mathbf{J}}\mathbf{v} \\ -\mathbf{h} \end{bmatrix} \tag{10}$$

Note: that contact wrenches $\mathbf{W}$ are part of the decision variables and are selected by the solver.

2 Homework

You should build on the solution of the previous tutorial. The goal is to apply whole-body control to our humanoid robot. We will make use of the open-source library TSID.

- Install the new library: `sudo apt install robotpkg-py3*-tsid`
- In the template you can find two new scripts `tsid_wrapper.py` and `config.py`. First start by checking the files.

  The file `config.py` contains all robot specific settings for solving Eq. (10). You will find:

  - task weights w_i
  - task gains $\mathbf{K}_p, \mathbf{K}_d$
  - names of robot frames we want to control
  - homing posture
  - definitions of feet / hand contacts

  The file `tsid_wrapper.py` contains a wrapper class that sets up the controller formulation and offers many setter/getter to turn on / turn off / modify task references. Per default the following tasks are added to the formulation in Eq. (10):

  - contact task on the left and right foot, formulated as a constraint (hard contact model)
  - center of mass task (COM) to move the COM of the robot
  - angular momentum task (AM) to reduce unwanted angular momentum to zero
  - torso task to keep the robots torso in a upright position (orientation reference)
  - foot motion tasks on the left and right foot, to move the feet if there is not contact
  - a posture task that keep the robot close to a desired posture

TSID has more tasks implemented, e.g. to limit actuation torques, velocities, joints. The function in `TSIDWrapper` allow you to

- update the formulation to compute the next command: `update()`
- add and remove contacts e.g. `add_contact_RF()`
- add and remove motion e.g. `remove_motion_RF()`
- set references e.g. `set_RF_pose_ref()`
- get task states (pose, vel) `get_RF_3d_pos_vel_acc()`

Here `RF` stands for right foot similar functions exist for other robot end-effectors.

a Exercise 1: Getting the robot to stand

The first task is to load the robot and the controller formulation inside a new script and let the robot stand (no motions). This is the most basic task and should work out of the box. For this, we are working in the script `t4_standing.py`

- Like the last time we need a robot class `Talos` that derives from `Robot`. This time we will load the robot with a floating base `useFixedBase=False` and the `pin.RobotWrapper` inside is no longer needed since TSID has its own robot wrapper.
- Again, add a ROS publisher for the robots joint state in the `publish` function
- Now, since the floating base is no longer fixed add a TF-Broadcaster that publishers the transformation `T_b_w` between `"base_link"` and `"world"`
- now edit the main function
  - Instantiate the `TSIDWrapper`
  - Instantiate the simulator `PybulletWrapper`
  - Then, instantiate `Talos` and give it the model from `TSIDWrapper` and the joint configuration from `conf.q_home`
  - Now, create a while loop to run the simulation. We need to update the simulator, the robot and finally, the TSID controller. The `TSIDWrapper.update(...)` will solve the problem in Eq. (10) and return the solution torque and acceleration. Feed the torque to our robot.
  - Finally, call the robot's `publish` function at 30 Hz. You can get the base-to-world transformation from TSID with the `baseState()` function.

  If everything is alright, the robot should be standing and holding its desired posture (`conf.q_home`). You should also be able to visualize the robot in RVIZ.

b Exercise 2: Getting the robot to balance on foot

Now that TSID is powering the robot, lets program some behaviours by changing the references of the many tasks that are loaded into the TSID controller formulation. Work in the script `one_leg_stand.py`. Here we are trying to stand on one leg. This requires us to shift the centre of mass (COM) over one leg. Then, release the contact on the other leg. And finally, activate a motion on the free leg to lift it from the ground.

- First, copy over your `Talos` class as it remains the same.
- Then, instantiate everything as before inside the main function
- Now, change the reference of the COM Task to the XY position of the right foot (Note: the COM height should remain the same!). For this, get the current com position from TSID `comState().pos()` and current right foot (RF) position `get_placement_RF().translation`. Then set the new COM reference `setComRefState(p_com)`.
  Since we have low gains and dampening, this will slowly drag the COM towards the new goal. (Note: You can also use a spline here.)
- After 2 seconds have past, the COM should be fully shifted to the right. Remove the left foot contact `remove_contact_LF()` and set its new reference to be 0.3 meters over the ground. Use: `get_placement_LF(), set_LF_pose_ref()`.
- Finalize by adding the publisher and update calls as in the last exercise.

Now the robot should be standing on one leg.

c Exercise 3: Getting the robot to do squats

- Continue from the task before but copy everything into a new script called `squating`. After 4 seconds, we start changing the robot's height with a sinusoidal function. Modify the main function to add a sinus with amplitude $a = 0.05$ m and frequency of $f = 0.5$ Hz. Also, compute the derivatives and call `setComRefState(p_com, v_com, a_com)` to update the COM task.

d Exercise 3: Adding arm motions

- Now its time to extend the Controller formulation and add two more motion tasks (TaskSE3Equality) for the right and left hand. Check the `TSIDWrapper` init function (Line 252) that shows you how a new motion task on the right foot is added to the controller formulation. Repeat the same for the left and right hand. You can find the required frame names and gains inside `config`.
  Don't add the tasks immediately to the formulation (no `formulation.addMotionTask()`). We will activate them when the hand should start moving.
- In the main function, after 8 seconds have passed, start moving the right hand on a circle. The center of the circle is at $\mathbf{c} = \begin{bmatrix}0.4 & -0.2 & 1.1\end{bmatrix}^\mathsf{T}$ and has a radius of $r = 0.2$ meters. The frequency should be $f = 0.1$ Hz. The circle should be located in the $Y - Z$ plane in front of the robot. If you want can first use `simulator.addGlobalDebugTrajectory(X,Y,Z)` to visualize it in pybullet.
- Now update in every iteration the right-hand reference with `set_RH_pos_ref`. Set the linear part from the circle $\mathbf{p} = \mathbf{c} + \begin{bmatrix}0 & r\cos(\omega t) & r\sin(\omega t)\end{bmatrix}^\mathsf{T}$.
- Since we don't care about the orientation, you can set its gains to zero, before activating the right-hand task:

```
rightHandTask.setKp(100*np.array([1,1,1,0,0,0])
rightHandTask.setKp(2.0*np.sqrt(100)*np.array([1,1,1,0,0,0])
```

Institute for Cognitive Systems
Technische Universität München
M.Sc. Simon Armleder

e Exercise 4: Do some plotting

- Finally, plot (Pos, vel, acc) your COM reference, the computed COM by TSID (basically what the model thinks) and the COM computed by pybullet (the simulator). You can use functions such as:
  `TSIDWrapper.comState()`
  `robot.baseWorldPosition(), robot.baseWorldLinearVeloctiy()`
  `TSIDWrapper.comReference()`