MECEE 4602: Introduction to Robotics, Fall 2019

End-effector Forces and Statics

In previous lectures, we've talked about joint positions, end-effector positions, and the relationship between them (Forward Kinematics, Inverse Kinematics). We then talked about velocities, both for the joints and the end-effector (Differential Kinematics). In the context of trajectories we then discussed joint accelerations. Now it's time to talk about forces.

In this lecture, we look mainly at how forces and torques applied at the end-effector translate to joint torques (for revolute joints) and forces (for prismatic joints).

1 End-effector and Joint Forces

Consider the well-known relationship between joint velocities $\dot{\boldsymbol{q}}$ and end-effector velocity $\dot{\boldsymbol{x}}$:

$$\boldsymbol{J}\dot{\boldsymbol{q}} = \dot{\boldsymbol{x}} \tag{1}$$

As we know, a similar relationship applies to infinitesimal displacements around the current configuration:

$$\boldsymbol{J}\Delta\boldsymbol{q} = \Delta\boldsymbol{x} \tag{2}$$

Consider now that we have an **external force $\boldsymbol{\tau}$** applied **to the end-effector**, and counterbalanced by a set of **generalized joint forces $\boldsymbol{\lambda}$** (which are torques for revolute joints or forces for prismatic joints). If the manipulator is in static equilibrium, and we assume no other forces are acting on the system, the principle of virtual work tells us that the total work (force times displacement) performed by these forces must be zero, hence:

$$\boldsymbol{\lambda}^T\Delta\boldsymbol{q} - \boldsymbol{\tau}^T\Delta\boldsymbol{x} = 0 \tag{3}$$

$$\boldsymbol{\lambda}^T\Delta\boldsymbol{q} = \boldsymbol{\tau}^T\Delta\boldsymbol{x} \tag{4}$$

from which we can derive:

$$\boldsymbol{\lambda}^T\Delta\boldsymbol{q} = \boldsymbol{\tau}^T\boldsymbol{J}\Delta\boldsymbol{q} \tag{5}$$

and finally:

$$\boldsymbol{J}^T\boldsymbol{\tau} = \boldsymbol{\lambda} \tag{6}$$

Note again that we are assuming no other forces are active on the system. This is never truly the case. Even if the robot is indeded in static equilibrium, we must at least consider the effect of gravity. If the robot is moving, we must consider dynamic effects (acceleration, inertia) which we will discuss in the next lecture. In such cases, Eq. (6) can have a slightly different interpretation: it gives us the component of joint forces that arise in response to externally applied forces to the end-effector. Other joint forces exist, but they are balanced by other effects such as gravity, inertia, etc.

2 Jacobian Transpose

Eq. (6) is very important. It tells us that **the Jacobian transpose transforms end-effector forces to joint forces**. Note that, at first, this can be somewhat counterintuitive, since the Jacobian itself operates in the other direction, transforming joint velocities to end-effector velocities.

	joint space		end effector space
position	$\boldsymbol{q}$	$\xrightarrow{\text{FK}}$ $\xleftarrow{\text{IK}}$	$\boldsymbol{x}$
velocities	$\dot{\boldsymbol{q}}$	$\xrightarrow{\boldsymbol{J}}$	$\dot{\boldsymbol{x}}$
forces	$\boldsymbol{\lambda}$	$\xleftarrow{\boldsymbol{J}^T}$	$\boldsymbol{\tau}$

Or, in equation form:

$$\boldsymbol{J}\dot{\boldsymbol{q}} = \dot{\boldsymbol{x}} \tag{7}$$

$$\boldsymbol{J}^T\boldsymbol{\tau} = \boldsymbol{\lambda} \tag{8}$$

The second equation tells us that **if $\boldsymbol{J}^T$ has a null-space, there are some end-effector forces that produce no corresponding joint forces**. We can look at the following cases:

- for an over-actuated or fully-actuated robot, if the robot is in a singular configuration, ($\boldsymbol{J}$ is rank-defficient) then $\boldsymbol{J}^T$ has a null space. Intuitively, this means that the robot can not move the end-effector along some direction; end-effector forces applied along the same direction will have no result on the joints.
- an under-actuated robot (fewer joints than end-effector degrees of freedom) will have a $\boldsymbol{J}^T$ that has more columns than rows; such a matrix always has a null-space, hence for an under-actuated robot there will always be end-effector forces that produce no effect on the joints.

Linear algebra side-note: we can formalize the notion that the direction in which external forces produce no result on the joints is the same as the direction that the robot can not move the end-effector in. Assume a set of non-zero external end-effector forces $\boldsymbol{\tau}_0$ that produces no joint result:

$$\boldsymbol{J}^T\boldsymbol{\tau}_0 = 0 \tag{9}$$

This is equivalent to:

$$\boldsymbol{\tau}_0^T\boldsymbol{J} = 0 \tag{10}$$

Now assume that there is some joint movement $\boldsymbol{q}_0$ that produces motion in the direction $\boldsymbol{\tau}_0$. This would imply:

$$\boldsymbol{J}\boldsymbol{q}_0 = \boldsymbol{\tau}_0 \tag{11}$$

Multiplying by $\boldsymbol{\tau}_0^T$ and using (10) we obtain

$$\boldsymbol{\tau}_0^T\boldsymbol{J}\boldsymbol{q}_0 = \boldsymbol{\tau}_0^T\boldsymbol{\tau}_0 \tag{12}$$

$$0 = \boldsymbol{\tau}_0^T\boldsymbol{\tau}_0 \tag{13}$$

which contradicts our initial assumption.

Generally, the end-effector forces that produce no joint torques lie in the null-space of $\boldsymbol{J}^T$, which is known to be the orthogonal complement of the image of $\boldsymbol{J}$.

3 Examples

To illustrate, we will use a redundant design: the 3-link planar robot.

We recall (from Lecture 5) the Jacobian formulation for this robot:

$$\boldsymbol{J} = \begin{bmatrix} -l_3 S_{123} - l_2 S_{12} - l_1 S_1 & -l_3 S_{123} - l_2 S_{12} & -l_3 S_{123} \\ l_3 C_{123} + l_2 C_{12} + l_1 C_1 & l_3 C_{123} + l_2 C_{12} & l_3 C_{123} \end{bmatrix} \tag{14}$$

Again, we will use a Python function to easily compute the Jacobian. As before, we define the following function in a file called "jacobians.py":

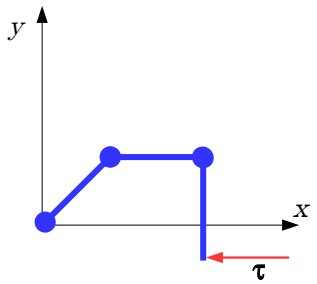
```
import numpy
from math import *
def Jac3link(q,l):
  J=numpy.zeros((2,3))
  J[0][0] = -l[2]*sin(q[0]+q[1]+q[2]) - l[1]*sin(q[0]+q[1]) - l[0]*sin(q[0])
  J[0][1] = -l[2]*sin(q[0]+q[1]+q[2]) - l[1]*sin(q[0]+q[1])
  J[0][2] = -l[2]*sin(q[0]+q[1]+q[2])
  J[1][0] =  l[2]*cos(q[0]+q[1]+q[2]) + l[1]*cos(q[0]+q[1]) + l[0]*cos(q[0])
  J[1][1] =  l[2]*cos(q[0]+q[1]+q[2]) + l[1]*cos(q[0]+q[1])
  J[1][2] =  l[2]*cos(q[0]+q[1]+q[2])
  return J
```

Then, we use it in iPython like this:

```
import math
import numpy
import jacobians
J=jacobians.Jac3link((math.pi/4.0, -math.pi/4.0, -math.pi/2.0),(1,1,1))
JT = numpy.transpose(J)
```

We notice that $\boldsymbol{J}^T$ has more rows than columns. This means it will only have a null-space if it is singular.

Pose 1: $\boldsymbol{q} = [\frac{\pi}{4}, \frac{-\pi}{4}, \frac{-\pi}{2}]^T, l_1 = l_2 = l_3 = 1$



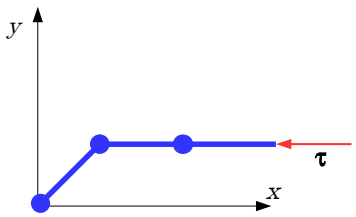
$$\boldsymbol{J}^T = \begin{bmatrix} 0.29 & 1.71 \\ 1.0 & 1.0 \\ 1.0 & 0.0 \end{bmatrix} \tag{15}$$

Let's solve for the joint torques created by an end-effector force $\boldsymbol{\tau} = [-1, 0]^T$.

```
tau=(-1,0)
l=numpy.dot(JT,tau)
print l
[-0.29 -1.   -1.  ]
```

Note that we get a small negative torque around the first joint, and bigger negative torques around the the second and third joints.

Pose 2: $\boldsymbol{q} = [\frac{\pi}{4}, \frac{-\pi}{4}, 0]^T, l_1 = l_2 = l_3 = 1$



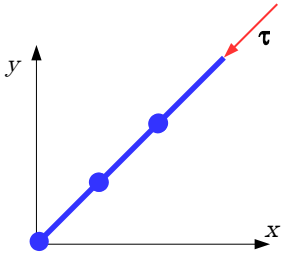
$$\boldsymbol{J}^T = \begin{bmatrix} -0.71 & 2.71 \\ 0.0 & 2.0 \\ 0.0 & 1.0 \end{bmatrix} \tag{16}$$

Solving for the joint torques created by the same end-effector force $\boldsymbol{\tau} = [-1, 0]^T$ yields:

```
tau=(-1,0)
l=numpy.dot(JT,tau)
print l
[ 0.71  0.    0.  ]
```

Note that the only joint that supports the external end-effector force is the first one.

Pose 3: $\boldsymbol{q} = [\frac{\pi}{4}, 0, 0]^T, l_1 = l_2 = l_3 = 1$



$$\boldsymbol{J}^T = \begin{bmatrix} -2.12 & 2.12 \\ -1.41 & 1.41 \\ -0.71 & 0.71 \end{bmatrix} \tag{17}$$

$\boldsymbol{J}^T$ is now singular. We can check that a force applied in the direction $\boldsymbol{\tau} = [-1, -1]^T$ produces no joint torques.

```
tau=(-1,0)
l=numpy.dot(JT,tau)
```

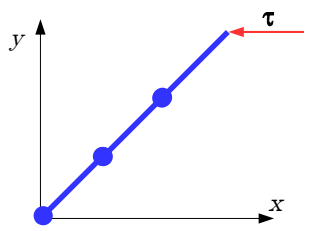
```
print l
[-0. -0. -0.]
```

More formally, recall that when performing a Singular Value Decomposition, the basis vectors of the null-space are given by the columns of $\boldsymbol{V}$ (rows of $\boldsymbol{V^T}$) that correspond to a zero singular value, in this case the 2nd column.

```
U,s,VT = numpy.linalg.svd(JT)
print s
[ 3.74  0.  ]
print VT
[[ 0.71 -0.71]
 [ 0.71  0.71]]
```

Note that end-effector forces in different directions can still produce joint torques. If we try again the previous joint force $\boldsymbol{\tau} = [-1, 0]^T$ we obtain:

```
tau=(-1,0)
l=numpy.dot(JT,tau)
print l
[ 2.12  1.41  0.71]
```



Discussion: What happens to the end-effector forces that are **not** counterbalanced by joint torques? After all, if the robot is quasi-static equilibrium, those forces must go somewhere...

4 Cartesian Control using the Jacobian Transpose

Cartesian (position) control refers to achieving a requested position goal for the end-effector, usually expressed as a change w.r.t the current position. In previous lectures we have seen how we can achieve that by computing and commanding a joint velocity that achieves the needed end-effector velocity. We do this by inverting the Jacobian; this control method is thus often referred to as J-inverse control:

$$\boldsymbol{J}\dot{\boldsymbol{q}} = \dot{\boldsymbol{x}} \tag{18}$$

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}\dot{\boldsymbol{x}} \tag{19}$$

$$\tag{20}$$

The Jacobian transpose gives us another way to do Cartesian control, referred to as J-transpose control. This method can be applied if our robot allows us to command joint torques $\boldsymbol{\lambda}$. We simply

command a set of joint torques that result in an end-effector force pointing towards the goal. This set of joint torques is straightforward to compute with the same equation we've used so far in this lecture:

$$\boldsymbol{\lambda} = \boldsymbol{J}^T \boldsymbol{\tau} \tag{21}$$

Note that this is an approximation of real-life behavior: applying these torques to the motors does not guarantee the end-effector will move in the desired direction, as many other factors come into play. Some of these include gravity, inertia, friction, etc.; we will learn about some of them in next lectures. However, in practice, J-transpose control tends to work well.

Some of the main differences between J-transpose and J-inverse Cartesian control are:

- J-transpose control does not require matrix inversion, a fairly expensive operation. If computation is done on a modern computer, this is less of a factor, but on smaller embedded CPUs it could still matter. Also note that if we want to achieve a secondary objective in the Jacobian nullspace, we still need to compute the pseudo-inverse.
- J-transpose control does not require special care around singularities, since the Jacobian is not inverted. If end-effector movement is requested in a direction that is impossible, the robot will simply not execute that.
- J-transpose control explicitly regulates end-effector forces that the robot applies. This allows us to control how much force the end-effector will apply if it encounters an obstacle on the way to the goal.
- For best results, J-transpose needs a robot with joint torque control available. Most available robots do not have torque sensors, and torque control is approximated via motor current control. This can be a poor approximation, resulting in poor tracking behavior. For many robots of this type, J-inverse control provides better tracking behavior (is better at following the goal).