
 (Spring 2007-08)
Lab Assignment #4Due **Lab #4**

In this assignment you will implement an operational space controller. This will allow you to control the manipulator at the end-effector level. See **Operational Space Framework** as reference.

Operational Space Non-Dynamic

1. The following controls should use a simple PD controller that does not consider the dynamics of the manipulator except to compensate for gravity, i.e. $\Gamma = J^T F + G$. See **PD Control in Operational Space** for information about calculating F .
2. Implement the user command “**nhold**”.
3. Implement the user command “**ngoto** $x\ y\ z\ \dots$ ”. Use the velocity saturation method presented in class. This will allow you to move the end-effector in a straight line. This controller should satisfy the constraint $v < v_{max}$ (**g_dxmax**) for linear motion, and $\omega < \omega_{max}$ (**g_wmax**) for angular motion.

v_{max} and ω_{max} are in global variables **g_dxmax** and **g_wmax**, respectively.

4. Implement the user command “**ntrack** $x\ y\ z\ \dots$ ”. Use a cubic spline trajectory and non-dynamic control. This controller should satisfy the constraint $v < v_{max}$ for linear velocity, $a < a_{max}$ for linear acceleration, and $\omega < \omega_{max}$ for angular velocity. See **Cubic Splines in Operational Space** for tips.

Operational Space Dynamic

1. Create a function **void OpDynamics(const PrVector& force)** to calculate the joint torques from the force F , using the following relation:

$$\Gamma = J^T(\Lambda F + \mu + p)$$

If the manipulator is in the vicinity of a singularity, then it should revert to non-dynamic control:

$$\Gamma = J^T F + G$$

The global variable **g_singularities** is nonzero in a singular configuration.

Use this function to implement the following dynamic controllers. In the next lab, you will modify this function to provide better singularity control.

2. Implement the user command “**hold**”, using dynamic control.

3. Implement the user command “**goto** $x y z \dots$ ”, using dynamic control. This controller should satisfy the constraint $v < v_{max}$ for linear motion, and $\omega < \omega_{max}$ for angular motion. Use the velocity saturation method presented in class. This will allow you to move the end-effector in a straight line.
4. Implement the user command “**track** $x y z \dots$ ”. Use a cubic spline trajectory and dynamic control. This controller should satisfy the constraint $v < v_{max}$ for linear velocity, $a < a_{max}$ for linear acceleration, and $\omega < \omega_{max}$ for angular velocity.

Experimental Data

1. Collect data from position (.4, .2, .7, .8, 0, .6, 0) to position (.9, -.2, -.1, .5, .5, .5, .5), using the **ngoto**, **goto**, **ntrack**, **track**, and **xtrack** controllers. Examine the trajectory by plotting the end-effector coordinates (x,z), linear velocity, linear acceleration, and angular velocity. Compare the results.
2. Select **6-DOF (quaternions)** from the combo box in the bottom center, and use **track** to move from (.58, .135, .43, .71, 0, .71, 0) to (.41, -.02, .43, .5, .5, -.5, .5). Now select **6-DOF (euler angle y)** and use **track** to move from (.58, .135, .43, 0, 90, 0) to (.41, -.02, .43, -90, 90, -180). This is the same motion, but the notation is different. How are the trajectories different? (The y value of 0.135 might need to be changed to match the y value in the all-zero configuration.)

Make sure you put enough comments so that we can understand what you're doing!

Submit a report describing what you learned in the lab. Discuss special features in your code. Incorporate material from lectures.

PD Control in Operational Space

Because the end-effector can rotate in more than one dimension, the operational-space PD controller is a little more complicated than the joint-space PD controller.

In joint space, the velocity is the derivative of the position. But this is not the case for angular velocity. None of the notations used for angular position (quaternions, euler angles, direction cosines, axis-angle, etc.) can be differentiated to get the angular velocity $(\omega_x, \omega_y, \omega_z)$.

Thus, we cannot simply add $-k_p(\vec{\theta} - \vec{\theta}_d)$ and $-k_v(\vec{\omega} - \vec{\omega}_d)$, where $\vec{\theta}$ denotes the orientation configuration parameter. It would be adding apples and oranges. Depending on what configuration parameters you use, $\vec{\theta} - \vec{\theta}_d$ may not even have the same number of dimensions as $\vec{\omega} - \vec{\omega}_d$. Quaternions use four dimensions to represent θ , and direction cosines use nine.

The solution is to multiply the $\vec{\theta} - \vec{\theta}_d$ term by the “ E^+ ” matrix, which translates the $\vec{\theta} - \vec{\theta}_d$ differential into units that are compatible with $\vec{\omega}$. E^+ is the pseudo-inverse of the E matrix, which does the opposite conversion. In other words,

$$\dot{\vec{\theta}} = E\vec{\omega} \quad \vec{\omega} = E^+\dot{\vec{\theta}}$$

The E and E^+ matrices are already calculated for you, in the global variables **g_E** and **g_Einverse**, respectively. To simplify the matrix multiplication, we added a 3×3 identity matrix to the upper-left corner of E . That way, you can treat \vec{v} and $\vec{\omega}$ as a six-dimensional vector, rather than dealing with them as two vectors and then combining the result. For example, here are the E and E^+ matrices that you'll get if you use quaternions $(\lambda_0, \lambda_1, \lambda_2, \lambda_3)$ to represent the orientation:

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\lambda_1/2 & -\lambda_2/2 & -\lambda_3/2 \\ 0 & 0 & 0 & \lambda_0/2 & \lambda_3/2 & -\lambda_2/2 \\ 0 & 0 & 0 & -\lambda_3/2 & \lambda_0/2 & \lambda_1/2 \\ 0 & 0 & 0 & \lambda_2/2 & -\lambda_1/2 & \lambda_0/2 \end{bmatrix} \quad E^+ = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2\lambda_1 & 2\lambda_0 & -2\lambda_3 & 2\lambda_2 \\ 0 & 0 & 0 & -2\lambda_2 & 2\lambda_3 & 2\lambda_0 & -2\lambda_1 \\ 0 & 0 & 0 & -2\lambda_3 & -2\lambda_2 & 2\lambda_1 & 2\lambda_0 \end{bmatrix}$$

Thus, the control law for an unsaturated operational-space PD controller is simply:

$$F = -k_p(E^+(x - x_d)) - k_v(v - E^+\dot{x}_d)$$

Cubic Splines in Operational Space

Orientation also causes some problems with cubic splines.

Maximum velocity

According to handout 8, in order to constrain a cubic spline $x(t)$ to a maximum velocity of \dot{x}_{\max} , the duration t_f of the spline must meet the following condition:

$$t_f \geq \frac{3}{2\dot{x}_{\max}} |\vec{x}_f - \vec{x}_0|$$

(Technically, handout 8 showed that the above equation works when x_f and x_0 are scalars. But it works equally well in a vector space, as long as the magnitude $|\vec{x}_f - \vec{x}_0|$ accurately reflects the separation between \vec{x}_f and \vec{x}_0 .)

This equation works fine in joint space and in linear operational space, but it fails in angular operational space. Once again, the problem is that a differential position such as $|\vec{\theta}_f - \vec{\theta}_0|$ is not comparable to an angular velocity such as ω_{\max} . And once again, the solution is to multiply by the E^+ matrix:

$$t_f \geq \frac{3}{2\omega_{\max}} |E^+(\theta_f - \theta_0)|$$

Note: In the program, the easiest way to find $E^+(\theta_f - \theta_0)$ is to multiply $\mathbf{g_Einverse} * \mathbf{g_x}$ to get a 6×1 vector, and then extract the angular part.

We will not limit the angular acceleration, since it varies with \dot{E} and is significantly harder to estimate.

Desired velocity

At each servo tick along the trajectory, you will calculate the desired position and velocity. Typically, the desired velocity is the time-derivative of the desired position: $a_1 + 2a_2t + 3a_3t^2$. Once again, you will have to multiply this by E^+ to convert it to the same units as $\mathbf{g_dx}$.

Interpolation

When you create cubic spline trajectories in operational space, you should decide how to interpolate the orientation. For example, suppose you're using rotation matrices to represent your orientation. If your spline starts at orientation R_1 and ends at R_2 , and you're currently halfway through the spline, how do you find the halfway point between two rotation matrices?

The experiments in this lab were designed to keep the rotations small. You can get away with simply using linear interpolation on your configuration parameters – even though linearly interpolating between two rotation matrices will create intermediate matrices that aren't quite orthogonal (You can orthogonalize them via SVD method). We may invest the time and implement a robust method right away to better prepare your group for the final project.

Here are some of the ways of getting better results:

- Linear interpolation works fine on Euler angles, since all three parameters are independent. It also works well on the 3-parameter form of axis-angle notation.

In both cases, you might consider how the angles are aliased — a move from 359° to 1° can be done with a 2° rotation (from 359° to 361°) instead of a 358° rotation. On the other hand, this improvement makes it more likely that you'll hit a joint limit.

- Quaternions (Euler parameters) might provide the most “natural” way to interpolate orientation. However, linear interpolation does not work: if you interpolate $\lambda_0 \dots \lambda_3$ independently, the result will not obey the law $\lambda_0^2 + \lambda_1^2 + \lambda_2^2 + \lambda_3^2 = 1$.

The best way to interpolate quaternion is to note that rotation quaternions lie on the surface of a four-dimensional unit hypersphere. Draw a great-circle arc from the initial rotation, i.e. the point on the sphere, to the final rotation, measure the angle, and interpolate the angle along the arc:

1. Let ${}^0_i Q$ be the initial rotation, and ${}^0_f Q$ Be the final rotation. (This naming convention is analogous to ${}^0_i R$ for rotation matrices.)
 2. Note that ${}^0_f Q$ and $-{}^0_f Q$ are two aliases for the same rotation. To take the “short way around” (with the same caveats mentioned above), choose the alias for ${}^0_f Q$ that is closer to ${}^0_i Q$. In other words, make sure that ${}^0_f Q \cdot {}^0_i Q \geq 0$.
 3. Find ${}^i_f Q = {}^i_0 Q {}^0_f Q = \overline{{}^0_i Q} {}^0_f Q$, and convert it to the axis-angle notation (\vec{u}, ϕ) from the identity ${}^i_f Q = (\cos(\phi/2), \vec{u} \sin(\phi/2))$. $\overline{{}^0_i Q}$ denotes a conjugate or inverse of the quaternion.
 4. Interpolate the angle ϕ_n from 0 to ϕ . The interpolated quaternion is ${}^0_n Q = {}^0_i Q {}^i_n Q$, where ${}^i_n Q = (\cos(\phi_n/2), \vec{u} \sin(\phi_n/2))$.
- The 4-parameter form of axis-angle notation, in which the angle and axis are separate, does not lend itself well to interpolation. The best solution is probably to convert to the 3-parameter form or to quaternions.