Advanced Robotic Manipulation Handout

CS327A (Spring 2017)

Problem Set #4

Due **Thurs, May** 26^{th}

Guidelines:

- This homework has both problem-solving and programming components. So please start early.
- In problems that require mathematical derivations, please try to be as detailed as possible so we can reward partially correct answers.
- For problems involving numerical answers, round off answers to two significant digits after the decimal point.
- We will be using the SAI2 (Simulation and Active Interfaces) software framework, developed in the Stanford Robotics Lab for this and future homeworks. Note that the SAI2-Simulation library is currently closed source. *So please do not forward or distribute.*
- Collaboration is allowed on this homework, but each student must submit their own original content (solution and code). In case you did collaborate, please note the names of other students you have worked with.

Submission Details:

We are using Gradescope for this class. So please submit your homework write-up through the Gradescope website directly. `https://gradescope.com/courses/7467`

For code submission, we'll be providing you a submission script with instructions soon.

Software update

The source code for this homework requires the latest version of SAI2-Common and the CS327A class repository.
To update the SAI2-Common repository, from within the `sai2-common` directory that you installed as part of homework 1, run

```
$ git pull
$ cd build && cmake -DCMAKE_BUILD_TYPE=Release .. && make && cd ../
```

To update the class repository, from within the `cs327a` directory, run

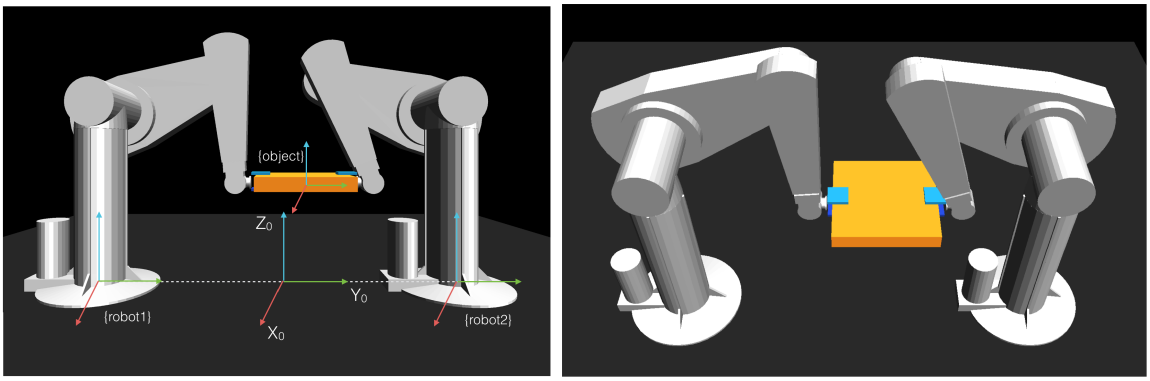
```
$ git stash
$ git pull --rebase
$ git stash pop
```

If you are asked to configure your computer for `git` with a username and email address, follow the instructions to do so, then run the above commands again.

Problem 1 - Centralized co-operative manipulation

For many manipulation tasks, a single robot might fall short to lift and move an object around due to limited motor capacity. As a result, multiple robots might be required to manipulate the object. We studied in class how the dynamics of the system changes when multiple arms grasp the same object, which we analyze through the representation of an *augmented object*. This representation can be used to design an artificial potential field at the object center of mass whose minimum value coincides with the desired manipulation objective. To realize the artificial potential field, the corresponding forces that need to be applied by each robot at its grasp point must be found. For this, we use the virtual linkage model which allows explicit accounting for the internal forces in the object. Finally, everything is put together to design the joint torques that achieves the required force at the end-effector of each robot.

(a) Set up



The co-operative manipulation set-up for this problem uses two PUMA robots as shown above. Each PUMA is outfitted with a gripper, which is modeled as a prismatic joint attached to the end-effector link.

The object to be manipulated is a box weighing $500g$, and the robots grip it at two ends along the Y-axis in the local frame. We model the object in simulation as a manipulator with six joints, three prismatic and three revolute. The last joint frame is co-located with the center of mass frame for the object.

Compile and run the starter file under `cs327a/hw4/p1-main.cpp`.

```
$ cd bin
$ pushd ../build && cmake -DCMAKE_BUILD_TYPE=Release .. && make && popd
$ ./hw4-p1
```

You should see the robots idling with the object held. This is because a basic open loop controller with gravity compensation has already been implemented for you. The robot starts out running this controller and then switches to your designed controller once the grasp is stable, which usually takes only a few milliseconds.

(b) Manipulator Jacobians at object COM

In previous homeworks, the robot base frame was located at the world base frame. However, in this problem, the two robots are each located at a distance of $0.6m$ apart from the world frame along the world Y-axis, but oriented alike.

To obtain the augmented object model, you will first have to obtain the basic Jacobian for each robot at the center of mass (COM) of the object. The simulation provides you the object COM frame as the variable `object_com_frame` which is measured in the world frame. You are also provided the fixed transformations `robot1_base_frame` and `robot2_base_frame` which are the transformations from the world frame to the base frame of robot 1 and 2 respectively.

Fill in the section titled `FILL ME IN: Manipulator Jacobians` to calculate `robot1_j0_objcom_world` and `robot2_j0_objcom_world`, the basic Jacobians for the robots calculated in the end-effector frame (link `end-effector`) at the current object COM location and measured in the world frame.

Hints:

- The transformations are stored as variables of type `Eigen::Affine3d` which behave similar to 4×4 transformation matrices. For example, you can multiply them together to get chained transformations. You can also get the inverse of a transform `T` as `T.inverse()`. Lastly, you can access the translation part as a `Eigen::Vector3d` object through `T.translation()` and the rotation part as a `Eigen::Matrix3d` through `T.linear()`.
- The transformation from the base frame to a link frame can be obtained through the following function in the `ModelInterface` class:

```
/**
* @brief transformation from base to link, in base coordinates,
* for the last updated configuration
* @param T Transformation matrix to which the result is computed
* @param link_name name of the link where to compute the transformation matrix
*/
void transform(Eigen::Affine3d& T, const std::string& link_name);
```

- Note that the `ModelInterface` objects `robot1` and `robot2` return frame-dependent measurements such as position and velocity in the robot's base frame, and not in the world frame. You will have to use the constant frame transformations `robot1_base_frame` and `robot2_base_frame` from the world frame to each of the robot's base frames, to measure such quantities in the world frame.
- We have updated the `ModelInterface` to include a function that returns the basic Jacobian in the correct order, i.e. $[J_v^T \;\; J_w^T]^T$. Note, however, that this function has a different name, `J_0(...)` to avoid confusion.

```
/**
* @brief Full jacobian for point on link, relative to base (id=0)
* for the last updated configuration in the form [Jv; Jw]
* @param J Matrix to which the jacobian will be written
* @param link_name the name of the link where to compute the jacobian
```

```
* @param pos_in_link the position of the point in the link where
* the jacobian is computed (in local link frame)
*/
void J_0(Eigen::MatrixXd& J,
    const std::string& link_name,
    const Eigen::Vector3d& pos_in_link);
```

(c) Augmented object model

Update the section titled `FILL ME IN: Augmented object model` to calculate `augmented_object_inertia`, the augmented object inertia at the object COM, and `augmented_object_p`, the augmented object gravity vector at the object COM.

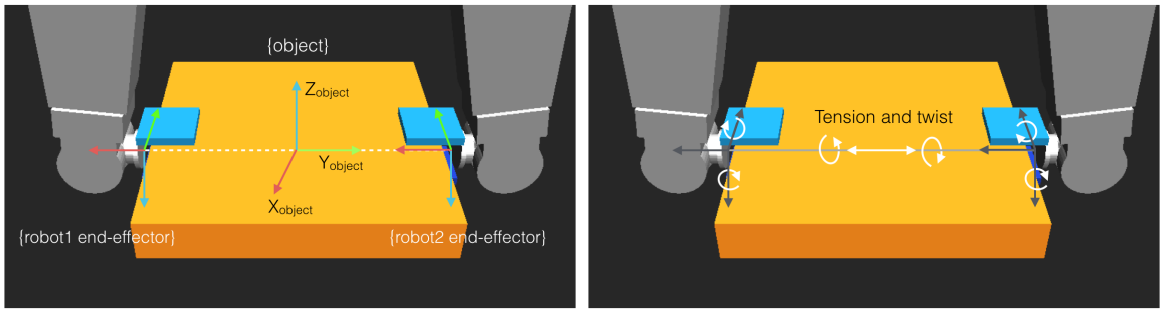
The object's inertia at its COM frame is already computed for you and saved as the variable `object_inertia`.

(d) Grasp matrix

Update the section titled `FILL ME IN: Grasp matrix` to calculate `G`, the grasp matrix describing the relationship between the forces and moments applied at the two robot end-effectors, and the resultant and internal forces in the object.

Hint:

- Since there are only two robots grasping the object, the internal forces are given by a single tension along the line joining the end-effector frames, and five internal moments. The internal moments are comprised of the end effector moments about X and Z axis in the object COM frame, as well as a twisting moment about the Y axis. The figure below qualitatively shows these quantities:



(e) Force control

Design a controller that tracks the following desired object configuration:

$$\begin{aligned} x_{object} &= 0.0 \\ y_{object} &= 0.15 \sin\left(\frac{2\pi t}{3}\right) \\ z_{object} &= 0.4 \\ \lambda_{0,obj} &= 1.0 \\ \lambda_{1,obj} &= 0.0 \\ \lambda_{2,obj} &= 0.0 \\ \lambda_{3,obj} &= 0.0 \end{aligned}$$

In addition, your controller should maintain an internal tension of $-20N$ (that is, a compression of $20N$) between the end-effectors, and zero internal moments.

Note that once you have set the joint torques for each robot, `tau1` and `tau2`, an additional gripper force of $15N$ is added to each robot by the starter code. This is to maintain a rigid grip on the object. You should not need to change this.

Also note that if your controller attempts to set a desired torque of $\geq 200Nm$ on any joint, the starter code prints a message saying `"Torque override. User asked torques beyond safe limit."` and switches to a default safe controller to prevent unwanted instability in the simulation. Similarly, if a robot loses grip on the object, the starter code prints a message saying `"Torque override. Robot 1 or 2 dropped object."` and switches to the default safe controller.

Hints:

- Instead of compensating for gravity on the augmented object, you can compensate for each robot separately, and for the object alone at the operational point. The following controller structure will achieve it:

$$\begin{aligned} \Gamma_1 &= {}^{world}J_{1,ee}^T \, {}^{world}F_{1,ee} + g_1 \\ \Gamma_2 &= {}^{world}J_{2,ee}^T \, {}^{world}F_{2,ee} + g_2 \\ \begin{bmatrix} f_{1,ee} \\ f_{2,ee} \\ m_{1,ee} \\ m_{2,ee} \end{bmatrix} &= G^{-1} \begin{bmatrix} \Lambda_{\oplus} F^*_{motion} + p_{obj} \\ t_{int} \\ \tau_{int} \end{bmatrix} \end{aligned}$$

  where ${}^{world}J_{1,ee}$ is the basic Jacobian for the first robot at its end-effector, measured in the world frame. ${}^{world}F_{1,ee} = [f_{1,ee}^T \; m_{1,ee}^T]$ is the desired force and moment vector at the end-effector for the first robot in the world frame. F^*_{motion} is the unit mass control force on the augmented object and p_{obj} is the object gravity vector available through the variable `object_p`. t_{int} and τ_{int} are the internal tension and moments respectively. All other symbols have their standard meaning.

- Note that the simulation for this problem has been slowed down by a factor of 10 to allow smooth control with feed-forward forces alone at the end-effector. This should not affect your code at all, but you will notice that the robot's movements are quite sluggish.

Problem 2 - Further reading on whole-body control

Read the paper 'Synthesis of Whole-body Behaviors through Hierarchical Control of Behavioral Primitives', Sentis L., Khatib O., IJHR, 2006. It is available online here:
cs.stanford.edu/groups/manips/publications/pdfs/Sentis_2005_IJHR.pdf

Write a critical summary of no more than a couple short paragraphs discussing the strengths and weaknesses of the paper. Attach any notes you take while reviewing the maths in the paper (just reviewing the key results is fine).

Problem 3 - Whole body control

The whole-body control strategy allows us to decompose the control effort into goals of different priority to achieve them in a mutually consistent manner. At the same time, the strategy is computationally efficient as it avoids the need to discover globally consistent plans in the configuration space in spite of uncertainty in the robot's environment.

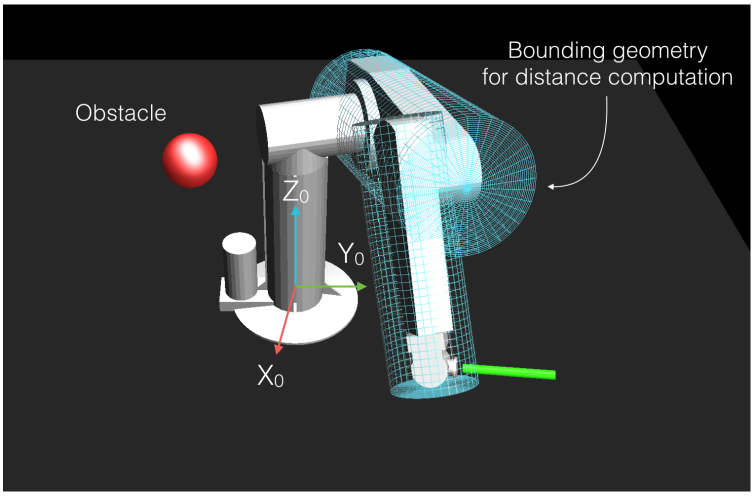
For this problem, you will explore a whole-body controller for the familiar PUMA robot. The robot's goals are decomposed into three parts - maintaining a position task trajectory at the end-effector, keeping all joints close to a desired posture and maintaining a safe distance from a moving obstacle in the environment.

Let the total applied control torque be Γ. We will design three separate controllers Γ_c, $\Gamma_{t|c}$ and $\Gamma_{p|t|c}$, where Γ_c is the control torque applied to maintain a safe distance which we treat as a constraint (constraint consistent task controller). $\Gamma_{t|c}$ is the control torque applied to track the desired end-effector position trajectory while not violating the constraint. Finally, $\Gamma_{p|t|c}$ is the control torque applied to maintain a desired joint posture while not violating the constraint and not disturbing the task (constraint and task consistent posture controller). Putting together the individual controllers, we will obtain

$$\Gamma = \Gamma_c + \Gamma_{t|c} + \Gamma_{p|t|c} + g$$

where g is the overall feed-forward compensation for gravity.

Shown below is a snapshot of the setup:



(a) Setup

To pull and execute the starter code for this problem, run from within the `cs327a` directory:

```
$ git stash
$ git pull --rebase
$ git stash pop
```

```
$ cd bin
$ pushd ../build && cmake -DCMAKE_BUILD_TYPE=Release .. && make && popd
$ ./hw4-p3
```

The robot should simply fall under gravity as no control torques are currently being applied.

Keyboard control of obstacle

The red sphere visible when you run the starter code is a phantom object with no physical properties that can be moved around in the Y_0 - Z_0 plane with arrow keys. `Left` and `Right` arrow keys move the obstacle along +ve and −ve Y_0 axis, whereas the `Up` and `Down` arrow keys move the obstacle along +ve and −ve Z_0 axis.

(b) Constraint controller design

The constraint considered in this problem is to maintain a safe distance from the red sphere. One way to do so is to create an artificial potential field.

Let the surface of the sphere be located at a distance d from the surface of the closest link, link i on the PUMA. Also let $x_{c,i}$ be the location of the point on link i closest to the sphere. Finally, let c be the location of the center of the sphere.

Design a potential function $U_c(d)$ whose gradient produces a repulsive force field in the direction $n_c = (c - x_{c,i})/\|c - x_{c,i}\|$. Refer to page 4 on the "Lecture10 - Elastic planning" handout for a possible design. Choose a cut-off distance d_0 such that the potential as well as its gradient are zero for $d \geq d_0$.

The control torques to maintain a safe distance from the sphere can then be written as

$$\Gamma_c = J_c^T \nabla U_c + J_c^T \Lambda_c (-k_{vc} J_c \dot{q})$$

where the additional term on the right is to damp motions in the constraint direction. J_c is the constraint Jacobian given by

$$J_c = n_c^T J_v \left(x_{c,i}\right)$$

where $J_v \left(x_{c,i}\right)$ is the linear velocity Jacobian at the point $x_{c,i}$ on link i.

We consider the constraint "active" if $d < d_0$ when the sphere gets too close to the PUMA, else we consider it "inactive". When the constraint is "inactive", we set $\Gamma_c = 0$.

(c) Task controller design

For this problem, we consider the task of maintaining the end-effector of the PUMA along the following desired trajectory:

$$\begin{aligned} x_{ee,d} &= 0.7 \\ y_{ee,d} &= 0.2 + 0.4 \sin\left(\frac{2\pi t}{6}\right) \\ z_{ee,d} &= 0.5 \end{aligned}$$

Design a PD controller such that the end-effector follows the above trajectory while not violating the constraint. That is, if the constraint is "active", the task control torque should not produce any acceleration in the constraint direction. In other words, we require that

$$J_c A^{-1} \Gamma_{t|c} = 0$$

when the constraint is "active".

The following template might be useful:

$$\Gamma_{t|c} = J_{t|c}^T \Lambda_{t|c} F_{motion}^*$$

where $J_{t|c}$ is the constraint consistent task Jacobian. If the constraint is "active", then

$$J_{t|c} = J_t N_c$$

else, if the constraint is "inactive"

$$J_{t|c} = J_t.$$

The constraint consistent task space inertia $\Lambda_{t|c}$ is given by $\left(J_{t|c} A^{-1} J_{t|c}^T\right)^{-1}$.

Note: The end-effector tip is located at $[-0.2 \;\; 0 \;\; 0]$ in the last joint frame, and is available through the variable `ee_pos_local` in the starter code. So, you have to use the following commands from the model interface to evaluate the position and linear velocity Jacobian at the tip of the end-effector.

```
// calculate end-effector tip position
Eigen::Vector3d ee_x;
robot->position(ee_x, ee_link_name, ee_pos_local);

// calculate end-effector tip linear velocity Jacobian
Eigen::VectorXd Jv(3, robot->dof());
robot->Jv(Jv, ee_link_name, ee_pos_local);
```

Optional: The above controller will work as expected, but it can be made better. Currently, we allow the constraint torques Γ_c to disturb the task by producing accelerations given by $J_t A^{-1} \Gamma_c$ which are non-zero as long as the constraint is "active". Try to modify F_{motion}^* to compensate for these accelerations.

(d) Posture controller design

Finally, design a posture PD controller $\Gamma_{p|t|c}$ that holds the following joint angles:

$$q_d = \begin{bmatrix} 0.00 \\ 5.90 \\ 3.70 \\ 1.57 \\ 1.75 \\ 3.14 \end{bmatrix}.$$

However, ensure that the posture control torques produce no acceleration either in the constraint or in the task co-ordinates. That is, we require

$$J_c A^{-1} \Gamma_{p|t|c} = 0 \quad \text{and} \quad J_t A^{-1} \Gamma_{p|t|c} = 0$$

(e) Implementation

Now implement your controller in the portion marked as `FILL ME IN` in `cs327a/hw4/p3-main.cpp`. You might find the following variables useful, which are already evaluated for you in every control cycle:

```
// center of sphere in robot base frame
Eigen::Vector3d sphere_center;

// position of the closest point on the robot to the sphere,
// in robot base frame
Eigen::Vector3d closest_point;

// distance from closest point to the surface of the sphere
double closest_distance;

// name of the link on which the closest point is located
std::string closest_link_name;

// linear velocity Jacobian at the point closest to the
// surface of the sphere
Eigen::MatrixXd Jv_closest_point(3, robot->dof());

// desired end effector positions
Eigen::Vector3d ee_pos_des;

// desired joint positions
Eigen::VectorXd q_des(robot->dof());
```

Note: The distance computations are implemented not directly between the mesh representations of the PUMA links and the sphere, but instead with bounding cylinders around the links. This is to improve computational efficiency. We also consider only the two largest links of the PUMA, the upper arm and the lower arm.

Once you have your controller running, try moving the sphere around to test its robustness. Write a brief summary of the observed behavior.