CS 237B: Principles of Robot Autonomy II
Problem Set 2: Grasping and Manipulation
Due Feb 16th, 2024 11:59PM

Submission Instructions

You will submit your homework to Gradescope. Your submission will consist of:

1. a single pdf with your written answers for report questions (denoted by the  symbol). This must be typeset (e.g., LaTeX or Word).
2. a .zip folder containing your code for the programming questions (denoted by the  symbol).

Honor Code

- **Collaboration**: You may collaborate with other students on the homework. However, each student should independently write up their own solutions and **clearly list the names of their collaborators** in their write-up.
- **Committing code to GitHub**: Please do not push your homework code to public GitHub repositories (for example, a fork of our repository). If you wish to commit your solutions to GitHub, please create a **private repository** by making a new copy of our repository, rather than forking it.

Introduction

For this homework, you will explore different elements of grasping and manipulation that may be based on analytical or learning methods. In particular you will investigate the following:

1. How to test for force and form closure?
2. How to optimize grasp forces?
3. How to learn intuitive physics?

Further, in terms of software development, you will

- Use python and cvx to solve optimization problems
- Use Tensorflow

Starter code for this problem set has been made available online through github; to get started download the code by running `git clone https://github.com/PrinciplesofRobotAutonomy/CS237B_HW2.git`.

Install Additional Software Dependencies

Homework 2 continues to use the virtual environment from the previous homework with added dependencies. To maintain consistency within this class, we already listed all dependencies on `requirements.txt` in the git repository you cloned. Navigate to the repo and activate your virtual environment. For Anaconda:

```
$ conda activate cs237b
```

Once you're in the virtual environment, run:

```
$ pip install -r requirements.txt
```

Note for installation on M1 Macs (from the cvxpy website): If you experience an error when trying to install `cvxpy`, this may be because you:

1. do not have `cmake` installed via Homebrew. You can install this by running `brew install cmake`
2. need to install the ECOS 2.0.5 wheel by running `pip install ecos==2.0.5`

Now we're ready to go!

Problem 1: Form and Force Closure.

A desirable property for a grasp is that the contact forces applied by the hand are such that they prevent contact separation and unwanted contact sliding. A special class of grasps are closure grasps that can be maintained for every possible disturbance load.

In this problem, we will investigate two types of closure: form and force closure. For analyzing whether a grasp is in form closure, we only take the normal forces into account that are applied at each contact. The resulting wrenches have to positively span the wrench space such that the grasp can resist any disturbance wrench.

The analysis to decide whether a grasp is in force closure is similar to form closure. The difference is that now, we also take into account friction such that a full range of forces can be applied at each contact point.

(i)  Explain why every form closure grasp is also a force closure grasp.

(ii)  To achieve form closure for a 2D object, you need at least 4 contacts. To achieve form closure with a 3D object, you need at least 7 contacts. Explain in your own words why this is the case.

(iii)  Figure 1 shows five stationary fingers contacting a 2D, right-angled triangle. The object is in form closure and therefore force closure. For a planar object, you need at least four contacts to achieve form closure. Therefore, if we take away one finger, the object may still be in form closure. For which subsets of four fingers is the object still in form closure? Prove your answer by visually analysing the wrench space! Include annotated drawings in your answer.

Hint: You may either use the method that was demonstrated in the lecture, or use the Graphical Planar Method explained in [1, Sec. 27.2.3].

(iv)  A set of j contact wrenches is in form closure if the matrix $F \in \mathbb{R}^{n \times j}$ (whose columns are made up of the j contact wrenches) is full rank and there exist a set of positive coefficients $k > 0$ such that

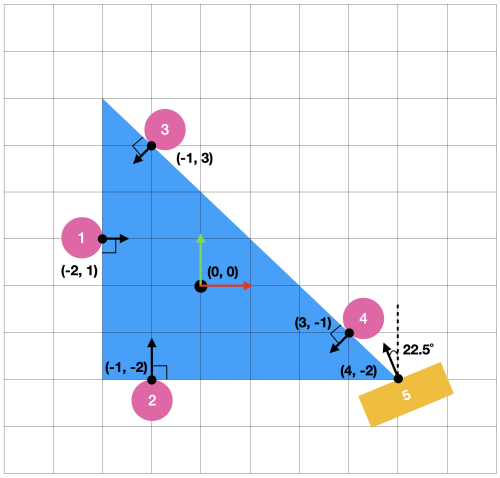

Figure 1: A right-angled triangle in contact with five stationary fingers, yielding form closure and therefore force closure. Analyze the contact when one or more fingers are removed. The triangle's center of mass is at (0, 0). The hypotenuse of the triangle is 45° from the vertical on the page, and contact normal 5 is 22.5° from the vertical.

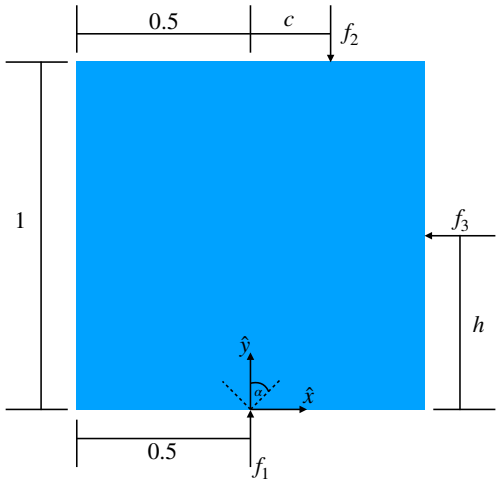

Figure 2: A uniform-density square restrained by three point contacts f_1, f_2 and f_3. $\hat{x}$ and $\hat{y}$ denote the axes of the reference frame.

$Fk = 0$. One way to find if k exists is to set up a linear program (LP):

$$\begin{aligned} \underset{k}{\text{minimize}} \quad & 1^T k \\ \text{subject to} \quad & Fk = 0, \\ & k_i \geq 1 \quad i = 1, \ldots, j \end{aligned} \tag{1}$$

If F is full rank and the LP above has a solution, then a k satisfying the form closure condition exists. If the F is not full rank or the LP is infeasible, then no such k exists.

First, implement `wrench()` in `form_force_closure.py` to construct a wrench from a force f applied at a point p. If τ is the torque applied by f at p, then the wrench is:

$$\omega = \begin{pmatrix} f \\ \tau \end{pmatrix} \tag{2}$$

Next, implement `form_closure_program()` and `is_in_form_closure()` in `form_force_closure.py`. `is_in_form_closure()` should take in a list of forces acting on a planar or spatial body at a given list of contact points and return whether the body is in first-order form closure.

There are a few unit tests included in `test.py`. These tests are not comprehensive, so you should add your own (for example, by adding tests for part (iii)). This script will not be run by the autograder, so feel free to modify it however you wish.

(v) For a planar rigid body, the test for force closure is equivalent to constructing a matrix $F \in \mathbb{R}^{3\times 2j}$ made up of the edges of the j friction cones, and finding a set of positive coefficients $k > 0$ such that $Fk = 0$. Note that this can be solved by the same LP in (1). Also note that if some contact points are frictionless, then there will be fewer than $2j$ columns, since frictionless contact points only contribute one column to the F matrix.

For a spatial rigid body, we first approximate the friction cones with friction pyramids and then construct $F \in \mathbb{R}^{6\times 4j}$ from the four edges of the j friction pyramids (or less if some points are frictionless). Implement this pyramid fitting in `cone_edges()` in `form_force_closure.py`. This function should also handle the special case for planar friction cones.

Next, fill out `is_in_force_closure()` to return whether a set of wrenches is in force closure.

Run `test.py` to test your implementation with a few unit tests. Again, these tests are not comprehensive, so you should add your own.

(vi)  Figure 2 shows a square that is restrained by three contact points: f_1 is a point contact with friction coefficient μ, while f_2 and f_3 are frictionless point contacts. The square has uniform density. First, derive the minimum value of μ for this grasp to yield force closure, as a function of h and c! Next, if $c = \frac{1}{4}$ and $h = \frac{1}{2}$, compute the minimum value of μ for the grasp to result in force closure.

You may want to use `is_in_force_closure()` from (v) to test if your derivation is correct!

Problem 2: Grasp Force Optimization.

While the force closure test implemented in Problem 1 tells us whether a grasp is in force closure or not, it does not tell us how to find a set of forces that has to be applied at each contact to resist a given disturbance wrench. In this problem, we explore an optimization method for computing the set of optimal forces that allow the grasp to be maintained under a given disturbance.

Suppose we have a rigid object grasped at M contact points, with positions $p^{(i)} \in \mathbb{R}^3$ for $i = 1, \ldots, M$ in a global coordinate system. Let $f^{(i)} \in \mathbb{R}^3$ be the force applied at contact point $p^{(i)}$, with a local coordinate system defined such that x and y are tangent to the object surface at $p^{(i)}$ and z is normal to the surface pointing inwards. To prevent slip, we need to constrain the contact forces to lie within the friction cone:

$$\sqrt{f_x^{(i)2} + f_y^{(i)2}} \leq \mu_i f_z^{(i)}. \tag{3}$$

To ensure that the grasp is stable, we also need to ensure a force equilibrium with total external force f^{ext} and torque τ^{ext} acting on the object:

$$\sum_{i=1}^{M} T^{(i)} f^{(i)} + f^{\text{ext}} = 0 \tag{4}$$

$$\sum_{i=1}^{M} P_{[\times]}^{(i)} T^{(i)} f^{(i)} + \tau^{\text{ext}} = 0 \tag{5}$$

Here, $T^{(i)}$ is the 3×3 rotation matrix that transforms from the local frame at $p^{(i)}$ to the global frame, and $P_{[\times]}^{(i)}$ is the skew-symmetric cross product matrix:

$$P_{[\times]}^{(i)} = \begin{bmatrix} 0 & -p_z^{(i)} & p_y^{(i)} \\ p_z^{(i)} & 0 & -p_x^{(i)} \\ -p_y^{(i)} & p_x^{(i)} & 0 \end{bmatrix}. \tag{6}$$

Given the friction cone and force equilibrium constraints, we can set up a convex optimization problem to find the optimal set of contact forces for a stable grasp, by some measure of optimality. If we want to minimize the maximum magnitude of the M contact forces, we can express our optimization problem as:

$$\begin{aligned} \underset{f^{(1)}, \ldots, f^{(M)}}{\text{minimize}} \quad & \max\left\{\|f^{(1)}\|, \ldots, \|f^{(M)}\|\right\} \\ \text{subject to} \quad & \sqrt{f_x^{(i)2} + f_y^{(i)2}} \leq \mu_i f_z^{(i)} \quad i = 1, \ldots, M, \\ & \sum_{i=1}^{M} T^{(i)} f^{(i)} + f^{\text{ext}} = 0, \\ & \sum_{i=1}^{M} P_{[\times]}^{(i)} T^{(i)} f^{(i)} + \tau^{\text{ext}} = 0 \end{aligned} \tag{7}$$

To solve the optimization problem, we need to rewrite (7) in a standard format that off-the-shelf convex optimization solvers can handle. Specifically, our optimization is a second order cone program (SOCP), where the friction cone constraint is a quadratic (aka second order) cone. The canonical form of an SOCP is:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & h^T x \\ \text{subject to} \quad & \|A_i x + b_i\|_2 \leq c_i^T x + d_i \quad i = 1, \ldots, m, \\ & Fx = g \end{aligned} \tag{8}$$

where the problem parameters are $h \in \mathbb{R}^n$, $A_i \in \mathbb{R}^{n_i \times n}$, $b_i \in \mathbb{R}^{n_i}$, $c_i \in \mathbb{R}^n$, $d_i \in \mathbb{R}$, $F \in \mathbb{R}^{p \times n}$, and $g \in \mathbb{R}^p$. $x \in \mathbb{R}^n$ is the optimization variable.

(i) Let $\omega^{\text{ext}} = (f^{\text{ext}}, \tau^{\text{ext}})$ be the external wrench represented as the concatenation of the external force and torque. Equations (4) and (5) can be written as a single matrix equation:

$$\Phi f + \omega^{\text{ext}} = 0$$

Show how to write Φ (also known as the grasp map) and f in terms of $T^{(i)}$, $P^{(i)}_{[\times]}$, and $f^{(i)}$.

(ii) The objective of the SOCP form is the linear function $h^T x$. While the objective in Eq. (7) is not linear, it can be recast into a linear form by adding second order cone constraints. Show how to do this.

Hint: define a new auxiliary scalar variable s and create M cone constraints involving $f^{(i)}$ and s.

(iii) Now that objective, friction cone constraints, and force equilibrium constraints have been reformulated, we are closer to implementing the SOCP. The last step is to combine all the variables so we can send them to the SOCP solver directly. Define the variable x, along with the SOCP parameters A, b, c, d, g, h, and F.

While your implementation should handle both the planar (2D) and spatial (3D) cases, for this written part, you only need to write out these terms for the spatial case.

(iv) Implement the SOCP above with `cvxpy` in the `grasp_optimization()` function in `grasp_optimization.py`. You can test your implementation with `test.py`, but as usual, you should add your own tests.

(v) This SOCP solves for the optimal grasping force given a single external wrench. To achieve force closure, we need to be able to compute a set of contact forces for any given external wrench. One straightforward way to do this would be to simply solve a new SOCP for every new external wrench, but this could become computationally expensive. To avoid solving the SOCP repeatedly, we can precompute solutions of the SOCP for the 12 signed unit vector wrenches in $\mathbb{R}^6$:

$$\begin{aligned}
\omega_1^+ &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \\
\omega_1^- &= \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \\
&\vdots \\
\omega_6^+ &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\
\omega_6^- &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}
\end{aligned}$$

Let $f_1^+, f_1^-, \ldots, f_6^+, f_6^-$ be the optimal grasp forces for the unit wrenches above. Then, a feasible set of contact forces for any given wrench can be taken as a linear combination of these 12 optimal grasp forces. This solution is suboptimal, but easy to compute.

$$f = (\omega_1^{\text{ext}})_+ f_1^+ + (\omega_1^{\text{ext}})_- f_1^- + \cdots + (\omega_6^{\text{ext}})_+ f_6^+ + (\omega_6^{\text{ext}})_- f_6^- \tag{9}$$

ω_i^{ext} is the ith component of the external wrench, and $(x)_+ = \max\{0, x\}$ and $(x)_- = \max\{0, -x\}$ are functions that take the positive and negative parts of x.

Implement this heuristic in the `precompute_force_closure()` function in `grasp_optimization.py`. Note that this function returns another function `force_closure()` that accepts an external wrench as an argument and returns f using the precomputed solutions.

Problem 3: Learning Intuitive Physics.

Learning physical properties of objects from visual data is an open research problem in robotics. The Physics 101 dataset contains over 17,000 video clips of objects of a wide range of materials, shapes, colors, and sizes that are being perturbed in various ways: dropped onto hard surfaces, dropped into water, attached to springs, collided with other objects, etc. The goal is to learn an intuitive physics model from this data that allows a robot to predict how the world changes when it interacts with it.

Each experiment has a set of observable outcomes, like object velocity, bounce height, acceleration in water, etc. The Physics 101 paper [2] proposes a model that predicts the outcome of an experiment as a function of latent, i.e. not directly observable, physical properties like mass, density, and coefficient of friction, which are shared across experiments. The proposed model has the physical laws built into the network structure such that by training it to predict experiment outcome, it has to internally estimate the relevant physical properties.

In this problem, we will take a subset of the data to learn just one physical property: the coefficients of friction of objects sliding down a ramp. In this experiment, the objects start at rest on either of two ramps with a 10° and 20° incline. If the ramp is steep enough, then the object will slide down the ramp with a certain acceleration. Starting with Newton's second law of motion, we can write the acceleration as a function of the friction coefficient:

$$F = ma = mg\sin\theta - \mu mg\cos\theta \tag{10}$$

$$a = g(\sin\theta - \mu\cos\theta) \tag{11}$$

Here, m is the mass of the object, a is the acceleration, θ is the incline angle of the ramp, g is acceleration due to gravity, and μ is the coefficient of friction between the object and ramp.

First, download the Physics 101 dataset and extract the contents to the `Problem_3` directory. The data should end up as a folder called `Problem_3/phys101`.

While the dataset contains full videos of the physics experiments, we are only interested in using the first frames of each video to train our neural network. To speed up the training process, we have preprocessed the video frames and exported them as jpg images. Download the following link and extract its contents to the `Problem_3` directory:

`https://drive.google.com/file/d/1BdjezenkQD5ur-cfjS-yFA4oPOv4e68n/view?usp=sharing`

It should end up as a folder called `Problem_3/frames`.

(i) Implement Eqn. (11) as a Tensorflow layer with the `AccelerationLaw` class in `model.py`. This layer takes in the friction coefficient μ and ramp incline θ as input and outputs the predicted acceleration of the object. θ is given to us in the dataset, but μ will be generated by our neural network.

(ii) Implement the `build_model()` function in `model.py`. Pay close attention to the model description in the comments. Specifically, the last two layers before the acceleration prediction layer should output a probability distribution of the object's material type and the friction coefficient of each material type.

Let p_i be the probability that the object is of material type i and μ_i be the friction coefficient of material i. Then, the friction coefficient of the object can be computed as:

$$\mu^{\text{pred}} = \sum_i p_i \mu_i \tag{12}$$

In our network, p_i is implemented as a softmax layer, and μ_i as a linear layer.

The rest of the model is left open for you to design. We use a small 2-layer CNN as a baseline, but you may try experimenting with things like using the pretrained Inception-v3 network from HW1!

(iii) Finally, implement the `loss()` function in `model.py`. For a batch of B samples, this should be the L2 norm of the prediction error:

$$\sqrt{\sum_{i=1}^{B} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}$$

where $y^{(i)}$ is the i-th sample of the batch and $\hat{y}^{(i)}$ is its prediction.

(iv) Train your neural network by running `train.py` and report the training and validation losses. For comparison, our simple CNN achieves a validation loss less than 5, computed over a batch of $B = 32$ samples.

(v) Run `test.py` to visualize your network's predictions on the validation set. How confident is its predictions of the material class (`p_class`)? Why are some μ_i (`mu_class`) negative? Can you think of a way to force the neural network to output positive μ_i? (You don't need to implement this).

(vi) To compare this network with embedded physics to a standard baseline, implement the `build_baseline_model()` function in `model.py`. The structure of this baseline network should match the physics network up until the `p_class` layer. Instead of outputting probabilities p_i, the network should directly output a scalar value through a fully connected layer that represents the predicted acceleration.

(vii) Train this baseline network by running `train.py --baseline` and report the training and validation losses. How does it compare to the physics network? What are some explanations for this result?

Do not worry about finetuning this baseline network to improve its performance. Our purpose here is simply to see how the special structure of the physics network helps it learn better for our problem.

References

[1] I. Kao, K. M. Lynch, and J. W. Burdick, "Contact modeling and manipulation," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer Publishing Company, Incorporated, 2016, ch. 37, pp. 931–954.

[2] J. Wu, J. J. Lim, H. Zhang, J. B. Tenenbaum, and W. T. Freeman, "Physics 101: Learning physical object properties from unlabeled videos," in *British Machine Vision Conference*, 2016.