Advanced-Level Training Problem Set for Computer Vision, Graphics, and Robotics

(draft, internal)

Su Lab @ UCSD

Contents

Chapter 1

Linear Algebra and Convex Optimization

1.1 SVD, KKT, QCQP

We consider the following constrained least square problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|Ax-b\|_2^2 \\ \text{subject to} \quad & x^T x \le \epsilon, \end{aligned}$$

where $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, \epsilon \in \mathbb{R}, \epsilon > 0$. The variable is $x \in \mathbb{R}^n$.
The solution to this problem will be used as a subroutine in subsequent geometry homeworks and we need an efficient solver. The solution to this optimization problem has to satisfy the Karush–Kuhn–Tucker (KKT) conditions:

$$\begin{cases} \nabla_x \mathscr{L}(x,\lambda) = 0 & (1) \\ x^T x \le \epsilon & (2) \\ \lambda \ge 0 & (3) \\ \lambda(x^T x - \epsilon) = 0 & (4) \end{cases}$$

where $\mathscr{L}(x,\lambda) = \frac{1}{2}\|Ax-b\|_2^2 + \lambda(x^T x - \epsilon)$ is the Lagrangian of the problem.

1. Write down the gradient of the Lagrangian $\nabla_x \mathscr{L}$.

   Next, We will solve this problem by considering 2 cases based on Condition (2).

2. Case 1: $x^T x < \epsilon$. Then, $\lambda = 0$ by Condition (4). To satisfy the KKT condition (1), it is equivalent to solving the unconstrained least square problem with objective $\frac{1}{2}\|Ax-b\|_2^2$. If its solution satisfies (2), we are done. Write down the closed-form solution to the unconstrained least-square problem. You do not need to show intermediate steps. (1pt)

3. Case 2: $x^T x = \epsilon$.

(a) Set $\nabla_x \mathscr{L} = 0$, express x in terms of λ, i.e. $x = h(\lambda)$ (1pt)

(b) Prove $h(\lambda)^T h(\lambda)$ is monotonically decreasing for $\lambda \geq 0$. *Hint:* You might need the fact that $A^T A = U \Lambda U^T$. (2pt)

By the monotone property of $h(\lambda)^T h(\lambda)$, we can solve $x^T x = \epsilon$ by line search over λ (e.g., bisection method or Newton's iterative method). You will implement it in the following programming assignment.

4. (Programming assignment) Solve a provided instance of this problem.

   - You can load the data by

     ```
     npz = np.load('./HW0_P1.npz')
     A = npz['A']
     b = npz['b']
     eps = npz['eps']
     ```

   - You are **not** allowed to use external optimization libraries that solve this problem directly. Linear algebra functions in numpy are allowed, e.g. numpy.linalg.eig, numpy.linalg.svd, numpy.linalg.lstsq, etc. Download the data from Ch1.zip

1.2 Probability

Given a triangle $\triangle ABC$, one common geometric processing question is how to sample n points uniformly inside the triangle.
A straight-forward idea is to interpolate the vertices by barycentric coordinates: we first sample $\alpha, \beta, \gamma \sim U([0,1])$, where $U([0,1])$ is the uniform distribution on $[0,1]$; then, we normalize them to obtain $\alpha' = \frac{\alpha}{\alpha+\beta+\gamma}$, $\beta' = \frac{\beta}{\alpha+\beta+\gamma}$, $\gamma' = \frac{\gamma}{\alpha+\beta+\gamma}$ so that $\alpha'+\beta'+\gamma' = 1$; finally, we obtain a sample $P = \alpha' A + \beta' B + \gamma' C$ inside $\triangle ABC$. However, this straight-forward idea is wrong: it does not assure that P is uniformly sampled in $\triangle ABC$.

1. We seek to rigorously disprove the above algorithm for a lower-dimensional setup, which samples points on a line segment $\overline{AB}$: If we sample $\alpha, \beta \sim U([0,1])$ and normalize by $\alpha' = \frac{\alpha}{\alpha+\beta}$ and $\beta' = \frac{\beta}{\alpha+\beta}$ to obtain $P = \alpha' A + \beta' B$, then P is not uniformly distributed between A and B. For simplicity, we assume that $A = 0$ and $B = 1$.

   - Show the cumulative density function of P, i.e., $\Pr(P \leq t)$. (Hint: $\Pr(A) = \int_x 1_{[x \in A]} f(x) \mathrm{d}x$ where $1_{[x \in A]}$ is the indicator function and $f(x)$ is the density function of random variable x.) (2pt)
   - Compute the value of the density function of P at $P = 0$ and $P = 0.5$. (1pt)

2. A correct algorithm for uniformly sampling points in $\triangle ABC$ is the following:

   (a) Sample $\alpha \sim U([0,1])$ and $\beta \sim U([0,1])$

   (b) $P' = A + \alpha(B - A) + \beta(C - A)$

   (c) If P' inside $\triangle ABC$, accept by letting $P = P'$. Otherwise, let $P = B + C - P'$.

   Question:

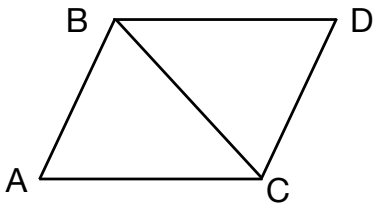


Figure 1.1: Figure for P2

- Prove that P' is uniformly distributed inside the parallelogram $ABDC$. (Hint: Show the density function pdf directly. Do not compute the cdf.) Hint 2: Suppose $\mathbf{y} = H(\mathbf{x})$ and H is bijective and differentiable, J is the Jacobian of $\mathbf{y}$ with respective to $\mathbf{x}$, $\mathbf{x}$ has density f, and $\mathbf{y}$ has density g, then $g(\mathbf{y}) = f(H^{-1}(\mathbf{y}))|\det(J^{-1})|$.) (2pt)
- Prove that P is uniformly distributed in $\triangle ABC$. (1pt)

3. Given a triangle whose vertices are at $A = (0,0)$, $B = (0,1)$, and $C = (1,0)$, write a program to simulate the wrong algorithm at the beginning of this problem and the correct algorithm shown above. Make a comparison of them by sampling 1000 points inside the triangle. Include the code and the plot in your submission. (2pt)

   For your convenience, we provide the following codes:

```
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

pts = np.array([[0,0], [0,1], [1,0]])
def draw_background(index):
    # DRAW THE TRIANGLE AS BACKGROUND
    p = Polygon(pts, closed=True, facecolor=(1,1,1,0), edgecolor=(0, 0, 0))

    plt.subplot(1, 2, index + 1)

    ax = plt.gca()
    ax.set_aspect('equal')
    ax.add_patch(p)
    ax.set_xlim(-0.1,1.1)
    ax.set_ylim(-0.1,1.1)

# YOUR CODE HERE

draw_background(0)
# REPLACE THE FOLLOWING LINE USING YOUR DATA (incorrect method)
plt.scatter(0.4+0.2*np.random.randn(1000),
            0.4+0.2*np.random.randn(1000), s=3)

draw_background(1)
# REPLACE THE FOLLOWING LINE USING YOUR DATA (correct method)
```

```
plt.scatter(0.4+0.2*np.random.randn(1000),
            0.4+0.2*np.random.randn(1000), s=3)

plt.show()
```

Chapter 2

2D Deep Learning

For each problem, we provide dataset, and we give link to relevant papers.
Remember that, for pytorch models, always use nn.Modulelist instead of python lists to contain a list of modules.

2.1 Image Classification, Network Architectures, and Data Augmentation

Image classification is one of the most fundamental computer vision tasks. In this problem, we aim to explore the influence of architecture, data augmentation, and the amount of training data on model performance. **Please put all code to this problem in a single jupyter notebook.**
For the following tasks, use float32 for training, even though the tutorials might say use float16. *It is not advised to use float16 generally as it could often lead to unexpected numerical gradient issues for many other datasets and many other tasks (especially for Reinforcement Learning tasks, you should never use float16). Thus if you want to use float16, only do it after float32 results look reasonable.*
Paper reference: ResNet (`https://arxiv.org/pdf/1512.03385`), Transformer (`https://arxiv.org/pdf/1706.03762.pdf`), vision Transformer (ViT) (`https://arxiv.org/pdf/2010.11929`)

- First, implement **standard ResNet 18** (reference implementation) and train on **CIFAR-100**. Use a recently-proposed high-performance dataloader FFCV (tutorial: link1, link2) for dataset loading and processing. Use a batch size of 128 and train for 120 epochs with the following learning rate (lr) schedule: first 60 epochs, use lr=0.1; next 30 epochs, use lr=0.01; final 30 epochs, use lr=0.001. Use SGD optimizer and weight decay 1e-4 (CIFAR, ImageNet classification are the few places we use SGD for training ResNet; for other cases we use Adam). Use the following standard data augmentation for CIFAR:
  - `(RandomCrop(32, padding=4), RandomHorizontalFlip())` in `torchvision.transforms`
  - or you can use the FFCV (ffcv.transforms) equivalent `(RandomTranslate(padding=4), RandomHorizontalFlip())`

Report training and test accuracy of 3 different models, trained on (1) 5% of training data; (2) 20% of training data, and (3) all of training data, respectively. You should get >71% success rate when trained on the entire training data.

(As a side note, above data augmentation is different from ImageNet training because CIFAR image size is 32x32 while ImageNet image size is 256x256. In ImageNet, the standard way is to take a 224x224 crop of the 256x256 image and apply random horizontal flip)

(As a side note, standard ResNet for image classification uses Batch Normalization. However for other use cases, Batch Normalization should not always be used. For example, in image-based reinforcement learning, Batch Normalization should never be used (since batch statistics for RL varies significantly). In these cases Layer Normalization is a better choice.)

- Next, implement **Vision Transformer (ViT)** and train on CIFAR-100. For this part you only need to train on the entire training data. A reference ViT implementation is here. Make sure that you completely understand multi-head self-attention. Make sure that you can write code using tools like `einops` and operations like torch.einsum, which makes Transformer code readable.

  Use a 6 layer ViT with hidden dimension 512 and MLP dimension 1024. Use a batch size of 128 and standard data augmentation. Use a dropout of 0.1 and embedding dropout of 0.1 (empirically dropout significantly reduces overfitting in Transformers; if you do not know Dropout, Batch Normalization, Layer Normalization, etc yet, make sure that you understand them). Note that Transformer implementation uses Layer Normalization instead of Batch Normalization like in standard ResNet, because empirically for Transformers, the batch statistics vary significantly between batches, which leads to instability.

  Use Adam Optimizer for ViT. You need to tune the learning rate (in small batch-size Transformer training, the lr should be <1e-3, and sth around 3e-4, 1e-4, 5e-5 should work well). Use a linear learning rate decay for training. You also need to tune the patch size (since CIFAR input image size is 32x32) and the number of attention heads (this should be >=8 for a 6-layer Transformer with hidden dim 1024, but should not be too many). Compare the performance of different hyperparameter choices.

  (As a side note, when Google, Facebook etc train Transformers for NLP, they often use a very large batch size and therefore a higher learning rate (often at a scale of 0.01 in the initial stages of training after warmup). However, we don't have much compute compared to industry-level infrastructures, so our batch size should be smaller accordingly)

- Finally, investigate the effect of different data augmentations for both ResNet and ViT. Investigate `ColorJitter` in `torch.transforms` along with one of `GrayScale` or `GaussianBlur` (you need to briefly tune the hyperparameters for these data augmentations). For each data augmentation, compose on top of standard data augmentations in CIFAR, and report training and test performance (also state the hyperparameters used for these data augmentations). A document for these augmentations can be found at this link.

  (The reason for choosing to study these augmentations is that they are used in self-supervised learning approaches like MoCo and BYOL, with implementations in e.g. this link)

2.2 Instance Segmentation

Overview Instance segmentation is a basic and classical computer vision task of detecting instances and predicting a segmentation mask per instance. Object detection is an extension to image classification. From this perspective, instance segmentation can be regarded as a combination of object detection and semantic segmentation (within each instance). Thus, understanding instance segmentation requires knowledge of object detection and semantic segmentation.
The key challenge of instance segmentation (or object detection), different from image classification, is how to predict a varying number of instances. The common practice is to define a set of candidates, and solve classification (foreground/background) and regression problems (size, location, semantic class) for each candidate. The most classical learning-based approach is Mask R-CNN, which extends the classical object detector R-CNN to predict an instance mask for each object. It follows R-CNN to explicitly generate candidates (proposals) in a sliding window fashion. However, the implementation of proposal generation and label assignment is complicated, and there are many relevant hyper-parameters. Besides, it tends to output duplicate detection objects, and requires some post-processing methods, like non-maximum suppression (NMS). In the recent years, DETR implicitly generates candidates and uses a set loss between ground-truth and outputs. It simplifies the implementation, but may converge more slowly. There are several following-up works to accelerate the training of DETR. DETR can also be extended for instance segmentation.
The most widely used dataset is COCO. PASCAL VOC, a smaller dataset, is used in many earlier works. The most common evaluation metric is mean average precision (mAP). Notice that it is also the metric for object detection. "The evaluation metrics for detection with bounding boxes and segmentation masks are identical in all respects except for the IoU computation (which is performed over boxes or masks, respectively)."
We recommend two popular libraries: Detectron2 and MMDetection. Besides, torchvision also provides a simple Mask R-CNN implementation.

Problem In this problem, you are asked to train an instance segmentation network (Mask R-CNN or DETR) on two datasets, Linemod and YCB-Video. The datasets can be downloaded from BOP. For Linemod, please use PBR rendered images as the training set, and all test images as the test set. For YCB-Video, please use the original real training images as the training set, and all test images as the test set.
You need to submit a report to include:

- Hyper-parameters that matter according to your experiments
- Evaluation results (mAP) on the test sets
- Visualization of success and failure cases

In fact, these two datasets only contain one instance per category in each image. However, they still can serve as an instance segmentation dataset.

2.3 Variational Auto-Encoder (VAE)

- Finish problem “VAE” in `vae_gan_release/VAE_GAN.ipynb`, where you will implement a standard VAE (without any label conditioning) on SVHN dataset.

- Derive the conditional VAE (cVAE) objective, where we want to reconstruct the target **y** given source **x** (e.g. given class label, construct images in the class).

2.4 WGAN-GP with SNGAN-like Architecture

Source: UC Berkeley Deep Unsupervised Learning Spring 2020
Paper: https://arxiv.org/pdf/1704.00028.pdf, https://arxiv.org/pdf/1802.05957.pdf
Finish problem "WGAN-GP with SNGAN-like Architecture" in vae_gan_release/VAE_GAN.ipynb, which implements WGAN-GP on CIFAR-10 using architecture similar to SN-GAN. Pay attention to the hyperparameter used for training and the fact that discriminator is optimized 5x more frequently than generator (this is true for image generation; however the schedule is always dependent on the dataset and task being used; for other tasks like GAIL in reinforcement learning, discriminator should be updated much less frequently or otherwise the discriminator reward for policy update will be near zero)
(note: It's important to use `tensor.contiguous()` after the dimension order has been changed, (i.e. after each `permute`) to save memory and increase speed.)

2.5 Cycle GAN

Source: UC Berkeley Deep Unsupervised Learning Spring 2020
Paper: https://arxiv.org/abs/1703.10593
You can refer to CycleGAN implementations here.
Finish problem "CycleGAN" in vae_gan_release/VAE_GAN.ipynb. In this problem, you will implement a CycleGAN that does **unpaired image-to-image translation (domain translation)** from MNIST to Colored-MNIST.

Chapter 3

3D Deep Learning

3.1 Rotation Representations, Conversation (HW1 of ML3D)

Background knowledge: 3D Transformations
Let $p := (1+i)/\sqrt{2}$ and $q := (1+j)/\sqrt{2}$ denote the unit-norm quaternions. Recall that the rotation $M(p)$ is a 90-degree rotation about the X axis, while $M(q)$ is a 90-degree rotation about the Y axis. In the notes, we composed the two rotations $M(p)$ and $M(q)$. Here, we instead investigate the rotation that lies halfway between $M(p)$ and $M(q)$.
The quaternion that lies halfway between p and q is simply

$$\frac{p+q}{2} = \frac{1}{\sqrt{2}} + \frac{i}{2\sqrt{2}} + \frac{j}{2\sqrt{2}}$$

Questions:

1. Calculate the norm $|(p+q)/2|$ of that quaternion, and note that it is not 1. Find a quaternion r that is a scalar multiple of $(p+q)/2$ and that has unit norm, $|r| = 1$, and calculate the rotation matrix $M(r)$. Around what axis does $M(r)$ rotate, and through what angle (say, to the nearest tenth of a degree)?

2. What are the exponential coordinates of p and q?

3. (skew-symmetric representation of rotation) In this problem, we use $[\omega]$ to represent a skew-symmetric matrix constructed from $\omega \in \mathbb{R}^3$ as instructed in class:

   (a) Build the skew-symmetric matrix $[\omega_p]$ of p and $[\omega_q]$ of q, and derive their rotation matrices.

   (b) Using what you have above to verify that the following $\exp([\omega_1] + [\omega_2]) = \exp([\omega_1])\exp([\omega_2])$ relationship does *not* hold for exponential map in general (Note: the condition for this equation to hold is $[\omega_1][\omega_2] = [\omega_2][\omega_1]$). Therefore, composing rotations in skew-symmetric representation should not be done in the above way.

   (c) Given two point clouds $X, Y \in \mathbb{R}^{3\times n}$ sampled from the surfaces of two shapes (two teapots, provided in rotation_conversion.npz), we aim to estimate the rigid transformation to best align them. For simplicity, assume that we have found the correspondence between the two

point clouds, so that the j-th column of X and Y are matched points. We also assume that the two point clouds are already aligned by translation. Then, the point cloud alignment problem can be formulated as the following Stiefel manifold optimization problem:

$$\begin{aligned} \underset{R}{\text{minimize}} \quad & \|RX - Y\|_F^2 \\ \text{subject to} \quad & R^T R = I \\ & \det(R) = 1 \end{aligned}$$

We can solve it using our knowledge of the exponential map. Note that, given a rotation matrix R_1, rotation matrices in its local neighborhood R_2 can be parameterized as $R_2 = R_1 \exp([\Delta\omega]) \approx R_1(I + [\Delta\omega])$ for $\Delta\omega \approx 0$.

i. Use the above knowledge to build an optimization algorithm by filling in the following routine:

Step 1: Initiate $R := I$

Step 2: *Describe your routine to find a* $\Delta\omega$
Hint: Convert the objective to a linear least square problem and solve by your solver in Ch. 1.1:

$$\begin{aligned} \underset{\Delta\omega}{\text{minimize}} \quad & \|A\Delta\omega - B\|_F^2 \\ \text{subject to} \quad & \|\Delta\omega\|^2 \leq \epsilon \end{aligned}$$

Step 3: Update R by $R := R\exp([\Delta\omega])$

Step 4: Go to Step 2

ii. [Programming Assignment] Use your algorithm to solve the point cloud data alignment problem with our provided data.

Note: For the point cloud alignment problem under rigid transformation, there exists a closed form solution to be covered in our next homework. While the closed form solution is more efficient, this iterative solver has better flexibility (e.g, modified derivation may work with different objective and/or constraint set), and is often used in solving non-rigid alignment problems.

4. Double-covering of quaternions

(a) What are the exponential coordinates of $p' = -p$ and $q' = -q$? What do you observe by comparing the exponential coordinates of $(p, -p)$ and $(q, -q)$? Does this relation hold for any quaternion pair $(r, -r)$? If it does, write down the statement and prove it.

(b) Note that the above is called *double-covering* of quaternions. Based on this property of quaternions, answer the following question. When designing a neural network to regress a quaternion output, can you use the L2 distance between the ground truth quaternion and the predicted quaternion? Why and why not?

3.2 Point Cloud Processing (HW1 of ML3D)

In this exercise, we will practice common point cloud processing routines. All problems are programming assignments.

1. Given mesh saddle.obj, sample $100K$ points uniformly on the surface. You may use a library (such as trimesh or open3d) to do it.

2. Use iterative farthest point sampling method to sample $4K$ points from the $100K$ uniform samples. You need to implement this algorithm. You may only use computation libraries such as Numpy and Scipy. *Hint:* To accelerate computation, use Numpy matrix operations whenever possible (they can be 100x times faster than pure python). You may also want to use a progress bar library since the computation may take a minute.

3. At each point of the $4K$ points, estimate the normal vector by Principal Component Analysis using 50 nearest neighbors from the $100K$ uniform points (You can use `sklearn.decomposition.PCA`). Since making the direction of normals consistent is a non-trivial task, for this assignment, you can orient the normals so that they roughly points in the Y direction.

4. Principal curvature estimation for point cloud: Modify the Rusinkiewicz's method (Lecture PDF) and apply it to this point cloud. Describe your algorithm and plot the Gaussian curvature for the shape.

3.3 Shape Deformation (HW2, ML3D)

Background knowledge: Single image to 3D

1. (Laplacian) Given a mesh $M = (V, E, F)$, we assume that the adjacency matrix is $A \in \mathbb{R}^{n\times n}$, $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix where $D[i,i]$ is the degree of the i-th vertex. The **Laplacian** matrix is defined as $L = D - A$.

   Prove that:

   (a) $\sum_{(i,j)\in E} \|x_i - x_j\|^2 = x^T L x$ for $x \in \mathbb{R}^n$.

   (b) $L \in \mathbb{S}^n_+$, i.e., L is a symmetric and positive semi-definite matrix.

   (c) For the data matrix $P \in \mathbb{R}^{n\times 3}$ where each row corresponds to a point in $\mathbb{R}^3$, denote the columns of P as $P = [x, y, z]$ and rows of P as $P = [p_1^T; p_2^T; \ldots; p_n^T]$, show that $\sum_{(i,j)\in E} \|p_i - p_j\|^2 = x^T L x + y^T L y + z^T L z$. (hint: Use the conclusion from 1(a))

2. **Normalized Laplacian** is defined as the normalized version of the Laplacian matrix above:

$$L_{norm} = D^{-1} L \tag{3.1}$$

   (a) Prove that the sum of each row of L_{norm} is 0.

   (b) The difference between a vertex x and the average position of its 1-ring neighborhood is a quantity that provides interesting geometric insight of the shape (see Figure 3.1). It can be shown that,

$$x - \frac{1}{|N(x)|} \sum_{y_i \in N(x)} y_i \approx H\vec{n}\Delta A \tag{3.2}$$

   for a good mesh, where $N(x)$ is the 1-ring neighorhood vertices of x by the mesh topology, $H = \frac{1}{2}(\kappa_{min} + \kappa_{\max})$ is the mean curvature at x (in the sense of the underlying continuous surface being approximated), $\vec{n}$ is the surface normal vector at x, and ΔA is a quantity proportional to the total area of the 1-ring fan (triangles formed by x and vertices along the 1-ring).

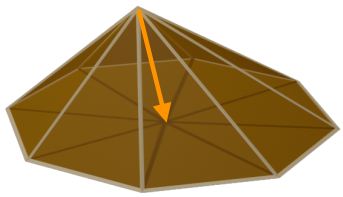


Figure 3.1: Curvature approximation by discrete Laplacian

Define $\Delta p_i := p_i - \frac{1}{|N(p_i)|}\sum_{p_j \in N(p_i)} p_j$. Prove that $\Delta p_i = [L_{norm}P]_i$, where P and p_i are defined as in 1(c), and $[X]_i$ is to access the i-th row of X.

3. (programming) Please load the deform_source.obj and deform_target.obj files using the trimesh library of Python, and optimize to deform the vertices of the `deform_source.obj` to match `deform_target.obj`. Plot the source object, target object, and deformed object.

   (a) Warm-up by Chamfer-only loss: Use the provided Chamfer distance function as a loss to deform the source towards the target. Optimize the position of the vertices in the source mesh by torch (you can just use the Adam optimizer). Show the result and describe what has happened in the deformation process by language.

   (b) Curvature and normal-based loss: We observe that Chamfer loss alone is not able to deform the source mesh properly, and additional loss needs to be added to regularize the process. Here we introduce a simple idea that would work for the provided instances by matching $\{\Delta p_i\}$:

   - First, we compute the Δp_i for each vertex of the source and the target meshes using 2(b).
   - Then, when we compute Chamfer-loss as in 3(a), we actually know about the correspondences across the source and target mesh vertex sets. For each pair of correspondences found in computing the Chamfer loss, we can use the L_2-norm square difference between the corresponding Δp_is' as the loss. Note that this loss computation can be bidirectional (from source to target and from target to source).

   Implement this loss in pytorch and combine it with the Chamfer loss to deform the source shape. Show your final results and deformation process (*e.g.*, the deformed mesh at every 100 steps).

   Note: `deform_source.obj` and `deform_target.obj` have similar amount of vertices, so the ΔA for vertices in the two meshes do not differ much, and matching Δp_is' corresponds to match mean curvatures and normal direction. If there is significant difference in vertex numbers, we need to compensate for the effect caused by significantly different ΔA in the two meshes.

3.4 Pose Estimation (HW2, ML3D)

In the next problems, you will use 2 methods, ICP and PointNet, to predict object poses in a dataset. The training data and testing data are provided at
https://drive.google.com/drive/folders/11TPw-XOypMLEgZLXUwFdBEHBPaMi0VCs?usp=sharing
You can download the data or directly use it in Google Colab by mounting the folder.

1. The task is to predict 6D poses for all the objects of interest in the scene given the image and the depth map. There are 79 different object classes in total. In each scene, there is only one instance for each object class. Across different scenes, instances of an object class might be scaled differently (in fact, only a discrete range, e.g. 0.5 or 1.0). This scale is provided in both training and testing datasets.

2. Here are some notations you might need to know before reading the rest.

   - NUM_OBJECTS: total number of objects we have. it is 79 in this assignment.
   - object_id: each object class is assigned with a unique id, decided by its order in objects.csv.

3. Data description. In training data, there are 2 folders: v2.2 and splits/v2, and there is a file object_v1.csv. The v2.2 folder contains the training data and splits/v2 is a pre-made train/validation split. You are allowed to create your own train/validate split and ignore splits/v2 completely. In splits/v2/train.txt and splits/v2/val.txt, each line is in the format "{level}-{scene}-{variant}". You can use it to find corresponding data files in the v2.2 data folder. Each variant with the same {level}-{scene} uses the same set of object classes with different poses. Here we give a detailed description of all files.

   - **objects.csv**. It provides metadata for the object meshes used to generate the scenes. The important fields are:
     - location: described where to find the meshes in models.zip.
     - geometric_symmetry: describe what symmetrical properties this object has. For example "z2|x2" means this object has a 2 fold symmetry around z axis and a 2 fold symmetry around x axis (which also implies a 2 fold symmetry around y axis). You can see Wikipedia entry Rotational_symmetry for details. "no" means this object has no symmetry; "zinf" means this object has an infinite-fold symmetry around z (e.g. cylinder). The symmetry properties are considered in our evaluation metric (e.g., for a cube, there are 24 rotations that will result in 0 error in evaluation).
     - visual_symmetry: similar to geometric symmetry, but considers object texture. Our evaluation metric will NOT consider visual symmetry, but it is an interesting research topic.
   - **{level}-{scene}-{variant}_color_kinect.png**: an RGB image captured from a camera containing the target objects.
   - **{level}-{scene}-{variant}_depth_kinect.png**: a depth image captured from a camera containing the target objects. The depth is in the unit of mm. You need to convert it into m.
   - **{level}-{scene}-{variant}_label_kinect.png**: a segmentation image captured from a camera containing the target objects. The segmentation id for objects are from 0 to 78.
   - **{level}-{scene}-{variant}_meta.pkl**: camera parameters, object names, gorund-truth poses, etc.
     - poses_world (list): The length is NUM_OBJECTS; a pose is a 4x4 transformation matrix (rotation and translation) for each object in the world frame, or None for non-existing objects.
     - extents (list): The length is NUM_OBJECTS; an extent is a (3,) array, representing the size of each object in its canonical frame (without scaling), or None for non-existing objects. The order is xyz.

  - scales (list): The length is NUM_OBJECTS; a scale is a (3,) array, representing the scale of each object, or None for non-existing objects.
  - object_ids (list): the object ids of interest.
  - object_names (list): the object names of interest.
  - extrinsic: 4x4 transformation matrix, world $\rightarrow$ viewer(opencv)
  - intrinsic: 3x3 matrix, viewer (opencv) $\rightarrow$ image
- We provide a Jupyter notebook to show how to use the data and also some handy scripts for visualization.
- In testing_data, The only difference from training data is that there are no splits and the ground truth poses are not provided in the metadata.
- There is also the models.zip, which provides the meshes and textures for the objects. You will need to sample canonical-space point cloud from the meshes for ICP.

4. Goal. Your goal is to predict the poses of objects in the testing data. The segmentation masks for objects are given so you do not need to solve the detection problem. You can simply segment out the point cloud for each object and use PointNet or ICP to solve the problem. You need to do it first using ICP and then using neural network.

5. Output format. We provide a sample output file "result.json". You need to read and understand the submission format.

6. Evaluation metrics. You should evaluate pose accuracy using the following criteria: success if rotation error < 5 degree and translation error < 1 cm.

7. Report.

   - A short paragraph describing your method and result. (A short version of "our method" and "experiments" section of academic publications) For now, report the success rate on the training set. Visually show a few pose estimation results on the testing set.

3.4.1 ICP-based Pose Estimation

Solve the problem above with an ICP-based method. You will need to sample canonical-space point cloud from the provided object meshes. You do not need training for this part.

3.4.2 Learning-based Pose Estimation

Solve the problem above with a learning-based method. We recommend using PointNet.

3.4.3 Combine 3.4.1 and 3.4.2

Find a way to combine problem 2 and 3 and improve your score. Describe your method in report and show increase of performance.

3.5 3D Semantic Segmentation (HW0, RoboML)

In this problem, you need to train a neural network to solve the 3D shape part segmentation task. Specifically, we focus on the ShapeNet chair category, and there are only four types of parts: arm, back, leg, and seat. The dataset is stored as a folded named “3DSeg" here.
We provide 1,000 annotated shapes as training data. You can find input point clouds in `3DSeg/train/pts` and ground truth labels in `3DSeg/train/label`. For point cloud files(*.pt), each line denotes the 3D coordinate of a point, and the number of points may vary. For label files(*.txt), each line denotes the part annotation of the corresponding point, where 1-4 indicates arm, back, leg, and seat, respectively. Fig. 3.2 shows some examples of the training data.

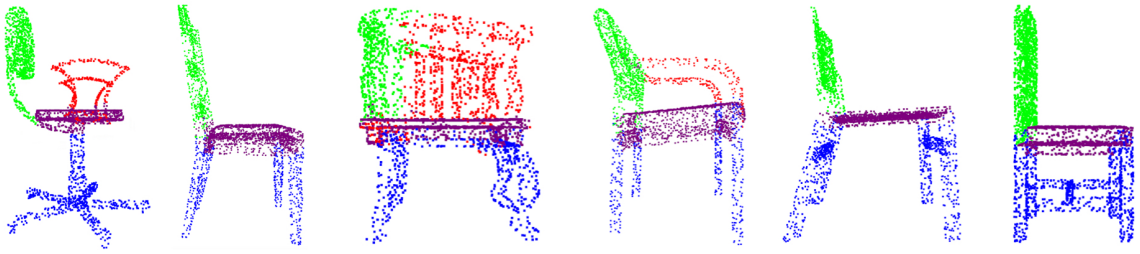


Figure 3.2: Training data: point clouds with ground truth label.

After training, you need to test your model on six testing point clouds provided in 3DSeg/test/ and visualize the results in your report. You can use any 3D library (e.g., Open3D) to visualize the point clouds. **Please color arm parts in red, back parts in green, leg parts in blue, and seat parts in purple**. You can leverage any existing 3D neural network (e.g., PointNet) as your network backbone. Since the segmentation task is relatively easy (only one category and four parts), you don’t need very complex networks, and the network training should converge very fast. You can use quantitative metrics (e.g., `mIoU`) to check your implementation. However, your results will only be evaluated based on visual appearance, and you do not need to achieve very high performance.
Hints:

1 the example implementation of PointNet.

2 the example of using Open3D: building Open3D PointCloud, visualizing pointcloud.

3 building your train-val split to test the generalization performance of your model.

3.6 3D Instance Segmentation (HW3, ML3D)

In the next problems, you will use a network to do object detection/segmentation in a dataset. The training data is the same as before. We additionally added testing for this problem with the name “testing_data_perception.zip”. The dataset link does not change:
https://drive.google.com/drive/folders/11TPw-XOypMLEgZLXUwFdBEHBPaMi0VCs?usp=sharing
You can download the data or directly use it in Google Colab by mounting the folder.

1. The task is to detect/segment objects for all the objects of interest in the scene given the image and the depth map. Essentially, the input is "xxx_color_kinect.png" and "xxx_depth_kinect.png", and you need to output "xxx_label_kinect.png". For dataset description, please refer to the last homework.

2. Model requirements. You are required to use a network with 3D components. If you only use 2D detection/segmentation networks, your maximum score for the report will be 3/5. We recommend implementing Frustum PointNet: https://arxiv.org/abs/1711.08488, as it is probably the simplest and the performance is very strong. You need to describe your method in the report.

3. Output format. The submission to the leaderboard should be a zip file containing 500 images, their names should be "{level}-{scene}-{variant}_label_kinect.png".

4. Evaluation. [5 points]

   Your score is computed as the higher of the absolute and relative score.

   Absolute score

   - 5 points: if you achieve higher than 80% average IOU.
   - 4 points: if you achieve higher than 60% average IOU.
   - 3 points: if you achieve higher than 40% average IOU.
   - 2 points: if you achieve higher than 20% average IOU.
   - 1 points: Make a submission

   Relative score

   - 5 points: rank 1-5.
   - 4 points: rank 6-15.
   - 3 points: rank 16-20.
   - 2 points: rank 21-25.
   - 1 points: the rest.

5. Report. [5 points]

   The report should contain explanation of your network model and experiments. 1-2 pages with pictures should be enough. Things to include: network architecture, experiment setups, training details, validation and test scores, visualizations.

6. Submission. The submission has 3 parts

   - A short report describing your method and result. (A short version of "our method" and "experiments" section of academic publications. 1 page should be enough, but you may write more if you have interesting findings.)
   - A .zip archive containing your code.
   - You need to submit your results on testing data to our internal benchmark.
     https://storage1.ucsd.edu/cse291ibenchmark/benchmark2
     The benchmark will be online shortly after the late due date of HW2.
     A benchmark submission will take about 20s, please be patient and please avoid trying to submit multiple times.

7. The benchmark suite (IOU) is uploaded to piazza resources “benchmark_pose_and_detection.zip”.

3.7 Point Cloud GAN

Overview “GANs are a framework for teaching a DL model to capture the training data’s distribution so we can generate new data from that same distribution. GANs were invented by Ian Goodfellow in 2014 and first described in the paper Generative Adversarial Nets." GANs are widely studied for 2D image generation. PyTorch provides a tutorial for DCGAN, which is a classical CNN architecture for 2D GANs. Wasserstein GAN (WGAN) proposes an alternative to traditional GAN training. WGAN-GP improves training of WGAN. MMgeneration provides implementations for many 2D GANs.
However, there are fewer works about 3D GANs, especially 3D point cloud GANs. Examples are Learning Representations and Generative Models for 3D Point Clouds [1], Point cloud gan [2], and Rethinking Sampling in 3D Point Cloud Generative Adversarial Networks [3].

Problem In this problem, you are asked to train a point cloud GAN on a pre-processed subset of ShapeNet. The data can be downloaded here from this repo. You need to first implement and train a basic (unconditioned) GAN on the data. Then, you need to implement and train a WGAN-GP on the same data.
The training pipeline (like data flow and losses) is almost the same for both image and point cloud GANs, except that the discriminator and the generator need to deal with point clouds now. You can follow [3] to use PointNet as the discriminator, and use a simple MLP as the generator. The output of the generator should be point clouds rather than latent codes in [1]. For WGAN-GP, you need to think over how to interpolate between two point clouds.
Apart from training generative models on point clouds, you need to:

- Visualize some generated point clouds (at least 16 examples, maybe organized in a 4x4 grid), including both success and failure cases (if any). To visualize point clouds, you can try Open3D, trimesh, matplotlib or even Blender.

- For each generated point cloud, find the most similar shape in the "real" data. The metric to measure similarity can be any distance function on point clouds, like Chamfer Distance or Earth Mover Distance. It is suggested to compare and visualize them side by side.

3.8 Learning-based MVS

Overview Reconstructing 3D geometry from images is a long standing problem in computer vision. With its applications range from navigation, autonomous driving to AR / VR. The problem has recently caught significant attention from these relevant communities. Among all 3d reconstruction methods, the multi-view stereo (MVS) aims at solving for the depth map from a set of neighboring images accompanied by camera intrinsic and extrinsic parameters. MVS methods rely on parallax to estimate scene geometry. A special case where only two views are given is typically referred to as stereo matching. We suggest you to go over OpenCV Depth Map from Stereo Images Tutorial, DispNet, PSMNet, and MVSNet before solving this problem. The Multi-View Stereo: A Tutorial could serve as a great resource to look up for some details.

Problem Implement a MVSNet on your own. You only need to implement the basic feature extraction, cost volume building, cost aggregation, and depth generation pipeline. You do not need to implement RefineNet and other techniques in the paper. You implementation should take 3 images at a time, and generate depth maps with the same sizes as input images. Train your model on the DTU dataset. You could consult to public repositories, but not allowed to copy their code directly. You are allowed to use pre-processed datasets available online.

1. Evaluate your method on the bottles dataset. Plot your depth map for image id `1_val_0026`, `1_val_0028` and `1_val_0073` with a proper color map. You can select whichever input views you need (no matter marked `train` or `val`) in this problem. *Hint:* The camera poses (in txt files) are in the opengl coordinate system. Meaning that if your code uses opencv coordinate system, you should flip the sign of the 2nd and 3rd column of the rotation matrices: `R[:, 1:3] *= -1`.

2. Evaluate your method on all 200 images in the dataset. Convert these depth maps into a complete clean point cloud of the object.
Hint: You should find way to filter incorrect points before fusing. Consult to the MVSNet paper for a possible way to achieve this.

3. What are the major issues you observed with MVSNet? Do you observe any failure cases during training or evaluation? Discuss any findings you have here and propose solutions if possible.

3.9 Neural Radience Field

The neural radiance field (NeRF) is a very recent method for scene representation. Unlike most explicit scene representations relying on meshes or voxels, neural radiance fields encode all information explicitly with a multi-layer perceptron (MLP), which yields to smoother scene representation. The NeRF is most commonly used to generate photo-realistic novel views of a scene. Please refer to the original NeRF paper for more details. In this problem, you will fit a NeRF model with given posed images. And then you are required to synthesize novel views of the scene with your model. Your solution to this problem should contain 3 parts: (1) Required test images for evaluation, (2) Answers to questions, and (3) A small report summarizing your method and experiments.

Read the following carefully:

1. The task is to fit a scene using a NeRF model with the bottles dataset. You are then required to synthesize novel views of the scene.

2. Data description. In training data, there are two folders: `rgb` contains training images of the scene, and `pose` contains corresponding 4×4 camera extrinsic matrices. `intrinsics.txt` is the 3×3 camera intrinsic matrix. You are free to use all 200 images to fit the model (`0_train` and `1_val` prefix). `pose` folder also contains camera extrinsic matrices for test novel views (`2_test` prefix).

3. **Your Solution Part 1.** Evaluation. Your submission is graded based on the quality of synthesized images. Specifically, you are required to generate image id `2_test_0000`, `2_test_0016`, `2_test_0055`, `2_test_0093`, and `2_test_0160` with your model. The image size should be 800×800. Your performance is evaluated based on the average PSNR of these images. There are 3 tiers depending on your average PSNR: (1) Acceptable: 20dB+, (2) Good: 29dB+, (3) Exceptional: 33dB+.

4. **Your Solution Part 2.** Questions. Answer these questions in the beginning of your report:
   (a) What is a radiance field? What information is included in a radiance field? How is *neural* radiance field (NeRF) different?
   (b) What is ray marching? Given a radiance field, how is each pixel calculated? (This is called the render equation.) Write down your render equation in a concrete math expression with clarified notations.
   (c) What is positional encoding? What is the purpose of positional encoding in NeRF models? Train your model without positional encoding and compare the results. You need to show at least two pairs of examples.
   (d) Is it possible to extract scene geometry (e.g. depth) information from a trained NeRF model? Describe your method in detail. Implement your method and show two depth maps generated by your method.
   (e) What are the major issues you find when using NeRF? List at least 2 drawbacks. For each of them, propose a possible improvement. You are encouraged to check follow-up papers of NeRF, but you should cite these works if borrowing their ideas.
5. **Your Solution Part 3.** Report. The report should contain explanation of your model and experiments. 1 or 2 pages should be enough. Things to include: models details, network architecture, training and test settings, validation and test scores, visualizations, and references. You are required to show the quality improvement of validation images during training process (e.g. visualize a evaluation image at 10k, 20k, 30k, ... training steps). You are encouraged but not required to find more scenes and visualize your results on these scenes.
6. Submission. You should submit the following:
   - A PDF containing the answers and a short report.
   - A .zip archive containing your code.
   - A .zip archive containing synthesized images with id `2_test_0000`, `2_test_0016`, `2_test_0055`, `2_test_0093`, and `2_test_0160`.
7. You should always assume each neural network experiment will take 1 whole day on a GPU. So you should start solving the problems **as early as possible**. If you are using the free version of Colab, make sure you save and load the network weights often as there is a running time limit.
8. *Hints:*
   (a) You can first train and evaluate your model on down-sampled 200×200 images to validate the entire pipeline before perform experiments on full size images.
   (b) You should split the 200 images into a training set and a validation set to track your progress while prevent over-fitting.
   (c) The camera poses (in txt files) are in the opengl coordinate system. Meaning that if your code uses opencv coordinate system, you should flip the sign of the 2nd and 3rd column of the rotation matrices: `R[:, 1:3] *= -1`.
   (d) Our dataset assumes a white background for all images. We suggest you modifying the original rendering equation to accommodate the dataset. (This is implemented in the original NeRF repository.)

(e) A full model of the original NeRF could give you 29dB+ PSNR. Some further improvements (e.g. using additional voxelized feature to represent the scene) could boost the performance up to around 35dB. You can find many of these useful improvements in follow-up works of NeRF.

(f) You can follow these step-by-step guides to implement your own NeRF model and get a better understanding of NeRF: NeRF Tutotial in Keras and Fit a simple Neural Radiance Field via raymarching. You are allowed to refer to public implementations but not allowed to copy their code directly.

Chapter 4

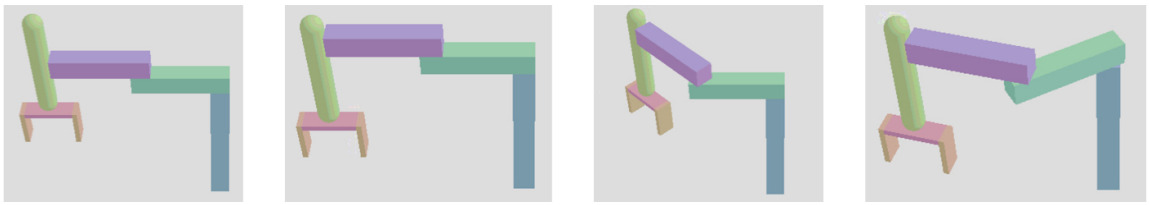
Planning and Control

4.1 Forward Kinematics (HW1 of RobotML).

In this problem, we will build up a toy robot arm in SAPIEN and calculate the forward kinematics for it.

1. **Build up a simple robot.**

   In *robot.py*, we provide an example to create a simple robot arm. As shown in Fig. 4.1, we use some simple primitives (i.e., box and cylinder) to represent links. For simplicity, the robot arm only contains three movable joints (two revolute joints and one prismatic joint). You need to fill in the blanks (marked as 'FILL_ME') in the function *create_robot* to set up the kinematics structure so that outputs of a test function match our expectation.



(a) (b) (c) (d)

Figure 4.1: A simple robot arm with different joint positions.

Specifically, function *test_FK* takes the robot arm and a set of joint positions as input and will print link poses under that configuration. In the *main* function, there are 5 test cases. If you correctly build up the robot arm, the outputs for the first four test cases should match the results shown in Fig. 4.2; and the visualizations should also match Fig. 4.1.

We don't provide the result for the last test case, and please report your output in your PDF submission. We will grade this problem based on your code and your result for the last test case.

Hints: 1. There are multiple ways to fill in the blanks, and you only need to ensure the resulting robot's behavior matches Fig. 4.1 and Fig. 4.2; 2. Understand the toy car example provided in *tutorial/create_articulation.py*; 3. See the comments in *robot.py*.

```
Test with qpos:  [0, 0, 0] .
Link base's pose is:  Pose([0, 0, 0], [1, 0, 0, 0]) .
Link link1's pose is:  Pose([0, 0.8, 1.7], [1, 0, 0, 0]) .
Link link2's pose is:  Pose([0, 2.4, 2.1], [1, 0, 0, 0]) .
Link link3's pose is:  Pose([0, 3.6, 2.1], [1, 0, 0, 0]) .
Link end_effector's pose is:  Pose([0, 3.6, 0.85], [1, 0, 0, 0]) .
Link left_pad's pose is:  Pose([0, 3.05, 0.5], [1, 0, 0, 0]) .
Link right_pad's pose is:  Pose([0, 4.15, 0.5], [1, 0, 0, 0]) .
----------------------------------------
Test with qpos:  [0, 0, 0.7] .
Link base's pose is:  Pose([0, 0, 0], [1, 0, 0, 0]) .
Link link1's pose is:  Pose([0, 0.8, 1.7], [1, 0, 0, 0]) .
Link link2's pose is:  Pose([0, 2.4, 2.1], [1, 0, 0, 0]) .
Link link3's pose is:  Pose([8.34465e-08, 3.6, 1.4], [1, 0, 0, 0]) .
Link end_effector's pose is:  Pose([8.34465e-08, 3.6, 0.15], [1, 0, 0, 0]) .
Link left_pad's pose is:  Pose([8.34465e-08, 3.05, -0.2], [1, 0, 0, 0]) .
Link right_pad's pose is:  Pose([8.34465e-08, 4.15, -0.2], [1, 0, 0, 0]) .
----------------------------------------
Test with qpos:  [0, 0.7853981633974483, 0.7] .
Link base's pose is:  Pose([0, 0, 0], [1, 0, 0, 0]) .
Link link1's pose is:  Pose([0, 0.8, 1.7], [1, 0, 0, 0]) .
Link link2's pose is:  Pose([0.565685, 2.16569, 2.1], [0.92388, 4.56194e-08, 0, -0.382683]) .
Link link3's pose is:  Pose([1.41421, 3.01421, 1.4], [0.92388, 4.56194e-08, 0, -0.382683]) .
Link end_effector's pose is:  Pose([1.41421, 3.01421, 0.15], [0.92388, 4.56194e-08, 0, -0.382683]) .
Link left_pad's pose is:  Pose([1.0253, 2.6253, -0.2], [0.92388, 4.56194e-08, 0, -0.382683]) .
Link right_pad's pose is:  Pose([1.80312, 3.40312, -0.2], [0.92388, 4.56194e-08, 0, -0.382683]) .
----------------------------------------
Test with qpos:  [-0.5235987755982988, 0.7853981633974483, 0.7] .
Link base's pose is:  Pose([0, 0, 0], [1, 0, 0, 0]) .
Link link1's pose is:  Pose([-0.4, 0.69282, 1.7], [0.965926, -3.08536e-08, 0, 0.258819]) .
Link link2's pose is:  Pose([-0.592945, 2.15838, 2.1], [0.991445, 1.55599e-08, 0, -0.130526]) .
Link link3's pose is:  Pose([-0.282362, 3.31749, 1.4], [0.991445, 1.55599e-08, 0, -0.130526]) .
Link end_effector's pose is:  Pose([-0.282362, 3.31749, 0.15], [0.991445, 1.55599e-08, 0, -0.130526]) .
Link left_pad's pose is:  Pose([-0.424712, 2.78623, -0.2], [0.991445, 1.55599e-08, 0, -0.130526]) .
Link right_pad's pose is:  Pose([-0.140011, 3.84875, -0.2], [0.991445, 1.55599e-08, 0, -0.130526]) .
```

Figure 4.2: Expected output of the first four test cases.

2. **Forward kinematics.** For the robot arm you just created, we want to calculate the forward kinematics. Specifically, given a set of joint angles $\theta = [\theta_1, \theta_2, \theta_3]$, the pose of the link 'end_effector' can be represented as a function $T(\theta)$. Here, we use a $SE(3)$ matrix to represent a pose. Please calculate the expression of $T(\theta)$.

   Hints: 1. $T([0.1\pi, 0.2\pi, -0.3]) = \begin{bmatrix} 0.588 & 0.809 & 0 & 2.112 \\ -0.809 & 0.588 & 0 & 2.697 \\ 0 & 0 & 1 & 1.15 \\ 0 & 0 & 0 & 1 \end{bmatrix}$; 2. You can also check your results with the function *test_FK*.

3. **Reachable workspace.** The set of positions (in $\mathbb{R}^3$) which can be reached by the end-effector with some choice of joint positions is called the *reachable workspace*:

$$W_R = \{p(\theta) : \theta \in Q\} \subset \mathbb{R}^3$$

   where Q is the configuration space and $p(\theta) : Q \to \mathbb{R}^3$ is the position component of the forward kinematics map $T(\theta)$. Please describe the reachable workspace for our robot arm, where $Q = [0, 2\pi) \times [-\frac{\pi}{2}, \frac{\pi}{2}] \times [-1, 1]$.

Hints: 1. You can describe the reachable workspace with figures and expressions; 2. In general, it will be challenging to analyze the reachable workspace if the robot is complex, but it's doable in our cases.

4. **Point velocity. (4pts)** Denote the center of the "left_pad" link as q. The coordinate of q in the spatial frame (i.e., base link frame) is denoted as q^s, while the coordinate in the end effector frame is denoted as q^e. Given the forward kinematics $T_{s\to e}(\theta)$, we can transform the coordinates by $q^s = T_{s\to e}(\theta)q^e$. Please first write down the expressions for both q^e and q^s.

   The velocity of point q in the spatial frame is given by $\mathrm{v}_q^s = \dot{q}^s$, where $\dot{q}^s$ indicates the derivative of q^s with respect to time. It is also possible to specify the velocity with respect to the (instantaneous) end effector frame, which is given by $\mathrm{v}_q^e = T_{s\to e}^{-1}\mathrm{v}_q^s$. For a specific joint position $\theta = (-\frac{\pi}{6}, \frac{\pi}{6}, \frac{1}{2})$ and joint velocity $\dot{\theta} = (1,2,1)$, please calculate v_q^s and v_q^e.

4.2 Motion Planning (HW3, RoboML).

In this problem, we will still play with the robot arm you created in homework 1. As shown in Fig. 4.3, there are 20 cuboids randomly scattered in the reachable workspace of the robot arm. Also, we provide a collision checker *checker*. You need to implement two sampling-based motion planning algorithms in *hw3.py*. To test your implementation, there are 100 test cases. For each test case, you need to find a collision-free path to move the end-effector from a start pose to a goal pose.

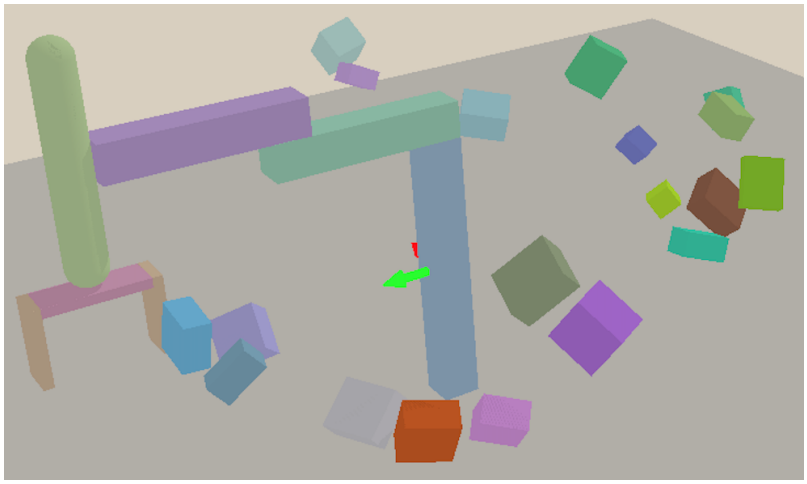


Figure 4.3: The robot arm and 20 obstacles.

1. **RRT (10pts)**

   You need to implement RRT or RRT-Connect in *class RRT* and then call function *testRRT()* to test your implementation. You can implement either RRT or RRT-Connect, but they may have different performance.

   testRRT() will call *rrt.solve(start_pose, goal_pose)* to solve each query. You need to first utilize an IK algorithm to convert the end-effector poses to joint positions, and then find a collision-free path

in the joint space. You can call *checker.check(q)* to test whether there is a collision under a specific configuration (joint position) q. If you find a path, please return a $n \times 3$ *numpy.array*, where n is the length of the found path and each row indicates the joint position of a waypoint on the path. If you cannot find a path, please return *None*.

You have up to 20 seconds for each query. Please note that each query is **independent** and you are **not allowed** to reuse the calculations (e.g., constructed trees) from the previous test cases. In addition, we examine your sample efficiency by counting the number of calls of function *checker.check()*. Specifically, for each test case, if you call function *checker.check()* for more than 20,000 times, you will fail that test case.

If you algorithm return a path within 20 seconds and 20,000 calls of *checker.check()*, *testRRT()* will then check the validity of your returned path. A path is valid if it satisfies:

- Each joint position is valid (within the joint limit & no collision).
- First joint position matches the start pose (forward kinematics).
- Last joint position matches the goal pose (forward kinematics).
- The Euclidean distance between two adjacent joint positions is **no greater than** 0.05.

The test function will output a detailed log, which includes an average success rate r_1%. Please report the **log and your code** in your PDF submission. If there is no other issue after reviewing and running your code, you will get $10 \times r_1$% points for this part.

Hints: 1. We provide an IK function *IK_analytical()*, which is based on analytical solution and is super fast. 2. You can use function *random_sample_qpos()* to randomly sample joint positions. 3. You can use *np.linalg.norm(q1-q2)* to calculate the distance between two joint positions. 4. You can use *visualize_path()* to visualize a planned path. 5. All test cases are guaranteed to be solvable. 6. You can use `NetworkX` for graph processing.

2. **PRM (8pts)** You need to implement the probabilistic roadmap method (PRM) in *class PRM* and then call function *testPRM()* to test your implementation. Unlike RRT, you need to first construct a roadmap before all queries, and then reuse the constructed roadmap to process each query.

   Specifically, function *testPRM()* will call *prm.build()* before all the queries to construct a roadmap in the joint space. You have up to 60 seconds to build the roadmap; and you can call *checker.check()* for up to 50,000 times.

   Then, *testPRM()* will call *prm.solve(start_pose, goal_pose)* to solve each query. You need to first utilize the IK algorithm to convert the end-effector poses to joint positions, and then leverage the constructed roadmap to find a collision-free path in the joint space. At this phase, for each query, you have only 1 seconds and you can only call *checker.check()* 500 times.

   The path validity checking is the same as in the RRT part. The test function will output a detailed log, which includes a average success rate r_2%. Please report the **log and your code** in your PDF submission. If there is no other issue after reviewing and running your code, you will get $8 \times r_2$% points for this part.

3. **Discussion (2pts)** Please compare the two algorithms and discuss how to choose the motion planning algorithm.

4.3 Inverse Kinematics

Overview Inverse Kinematics (IK) is an algorithm used to compute the robot joint position θ in the configuration space Q given the 3D pose T. It is the inverse problem of the forward kinematics (the first problem in Planning and Control). IK has wide application in the field of both robotics and computer graphics, either to control a physical embodied agent or animate different virtual creatures. Meanwhile, it is also the preliminary step for downstream tasks like motion planning or trajectory following control. We will visit these downstream tasks later. In robot kinematics and dynamics, the Jacobian matrix plays a vital role to connect the joint/configuration space and the Cartesian space. $\xi = J(\theta)\dot{\theta}$, where J is the spatial Jacobian matrix, $\dot{\theta}$ is the joint velocity and ξ is the spatial twist. The Jacobian matrix J is provided by us and you do not need to worry about how to compute it on your own. However, you need to understand the physical meaning of the Jacobian matrix and its shape.

Reference Material Read page 11-17 of Several Methods to Solve IK and finish the following problems. You can also refer to CSE291D. Please note that the notation of these two reference materials is different.

Problem Description In the previous forward kinematics problem, you are asked to compute $T(\theta)$ given θ. This time, the $T(\theta)$ is given and you need to compute θ. Different from the previous problem where you are working on a 3-DoF toy robot, you will play with a 7 DoF XArm and Adroit Robot Hand this time. The final goal of this problem is to mimic/follow the provided reference motion to grasp a mug on the table. The reference motion comes from DexYCB Dataset. Read and run the starter code to understand these SAPIEN functionalities. You will need to implement several functions in the starter code to finish this problem. The evaluation function is also provided in the starter code.

File List The files below can be downloaded here.

- *ik_and_control.py*: starter code and the main entry
- *utils.py*: utils used in the starter code
- *assets*: asset folder for robot and object models
- *hand_object_data.pkl*: reference data used in this problem, used in starter code
- *hand_finger_tip_ik.gif*: visualization of a possible solution for sub-problem 2.

Sub-Problem 1: Inverse Kinematics of Kinematics Chain In this sub-problem, you will need to compute the robot arm joint position θ so that the last link of XArm, link7, can be set to the given location. To be more precise, the provided reference motion is a manipulation trajectory from human, which contains multiple steps. You need to solve IK for each step. If your implementation works well, the position of XArm link 7 should be closed to the human palm root position and the distance computed by the provided function *compute_hand_palm_root_distance* should be a small value. At the same time, you will see that the robot arm and human hand will perform the same motion in the SAPIEN visualization window.

1. Implement the IK using Damped Least Squares method in the reference material page 17 so that your error computed by *compute_hand_palm_root_distance()* can be smaller than $2e-3$

2. Try different λ in the Damped Least Squares method and plot the error. The x-axis is λ and the y-axis is error. Then explain this plot briefly.

3. Choose a different algorithm from the reference material to solve the same problem. Compare the algorithm you choose with Damped Least Squares. Plot the error with respect to the trajectory step.

Hint We do not have constraints for the orientation of the target link (link7 of XArm), you can choose any orientation as you wish, e.g. identity rotation matrix.

Sub-Problem 2: Inverse Kinematics of Kinematics Tree In the previous problem, you solve the IK for a **Kinematics Chain**. In this problem, you will move to a harder setting where you need to solve the IK for a **Kinematics Tree**. You can take a look at **hand_finger_tip_ik.gif** for a better understanding of the goal of this problem.
Similarly, you will follow the same manipulation trajectory using a robot hand. Different from only matching one link in the previous problem, we need to match the position of each fingertip of the robot hand to the fingertip of provided human hand. You will need to read the starter code to get to know which link in the data corresponds to the fingertip, and in which order. Same as before, you do not need to worry about the orientation.

1. Modify the algorithm you are using for the previous problem so that it can be used to solve this problem. Your error computed by *compute_finger_tip_distance()* should be smaller than $1e-2$.

2. Can you choose an arbitrary orientation as before? Make a small DoF analysis to show why orientation needs to be taken care of in this sub-problem.

3. Plot the error from **compute_finger_tip_distance()** for each step.

4. Add more constraint to make the robot joint position temporally consistent over time. The mean distance of the robot joint position should be smaller than $1e-1$ for every two consecutive steps.

Hint The robot hand has a 6-DoF free-motion joint, which enables it to fly freely in the 3D space. You will also need to take care of them to solve this problem.

4.4 PID Controller

Overview PID control, especially PD control, is a simple and commonly used method for a dynamical system such as a robot. It leverages the closed-loop feedback to adjust the input based on the error signal. In robotics, the error signal often refers to the difference between the target position and current position, or that of velocity.

Reference Material Read the control section of CSE 291D Lecture 10 to understand the concept of PID control. A reference implementation in SAPIEN can be also found here, especially the *Write your own PID controller* section.

Problem Description In this problem, we will continue to work on the motion following problem in Problem 4.3. Instead of setting the robot joint position magically without physical process, you will control the robot to move using joint force/torque. To make things simpler, you will revisit the XArm hand palm problem based on the IK solution solved in Sub-Problem 1 of Problem 4.3. Remember that we call *robot.set_qpos()* to move the robot in the previous problem. This time you will need to implement a PID controller to compute the appropriate force/torque so that the robot can move to the joint position solved by your IK algorithm. You are not allowed to control the robot joint position without dynamics, such as *robot.set_qpos()*. Instead, what you are going to do is to compute the force/torque and apply it via *robot.set_qf()*. More details can be found in the *sim_xarm_ik()* function in *ik_and_control.py*. Besides, you are not allowed to use the *robot.set_drive_xxx* function in the SAPIEN tutorial but implement your controller.
To be more specific, you need to do the following steps:

1. Implement a PID controller to compute force/torque to make your robot follow the sequence of the IK solution. Your robot can move at a very slow speed to follow each joint position one by one. Similar as before, tuning the PID so that the error reported by *compute_hand_palm_root_distance()* can be smaller than $5e-3$ in average.

2. Then makes the setting harder by adding constraint on the time to go. Your robot should follow each joint position one by one with a fixed time interval $\delta t = 0.2$. Using the default $4e{-}3$ simulation timestep, your robot should reach the next joint position target every 50 simulation step.

Hint (1) Make sure your IK solution is temporal consistent, otherwise, it is nearly impossible to follow. (2) The self-collision of robot arm is disabled in the starter code, you do not need to worry about the tricky self-collision avoidance here. (3) Implement a PD controller first. PD controller is expected to solve the problem. You can then figure out how the integrator part can help after you make the PD controller work.

4.5 Integration of Modeling, Planning, Kinematics, and Dynamics

Overview System integration is an important skill for solving robotics tasks and for embodied AI research. In the previous problem, you have worked on building up a simple robot (modeling), sample-based motion planning (planning), inverse kinematics (kinematics), and PID control (dynamics). These components are connected to each other in the whole robotics system.

Problem Description In this problem, you will go over all these problems into a unified framework. The task objective is to grasp the same mug as before. You will first build a more realistic robot on your own and use this robot in this problem. Specifically, you are expected to finish the following tasks one by one. You need to combine everything and no starter code is provided for this problem but you can reuse the previous code and data.

Sub-Problem 1: Assembly Existing Part to Build a New Robot Model Understand the **urdf** format to describe a robot model. Based on the provided robot model of XArm and the flying adroit robot hand, build a XArm adroit robot model where the adroit robot hand is mounted on the XArm as the end effector. You can choose the mounting point as long as it is visually reasonable. The new model **should not** contain any global joint, e.g. the 6D joints in the flying hand.

Reference You can check ROS Industrial Tutorial and URDF XML specification.

Sub-Problem 2: Plan to Compute a Trajectory to Grasp the Object In this previous problem, the trajectory solved by IK and followed by your PID controller may also suffer from self-collision. One solution to solve the self-collision is motion planning as what you do in the RRT problem. Different from motion planning using a simple 3-DoF arm, here the robot model is much more complex. It is recommended to use some more efficient and convenient libraries to solve the task. We recommend you use the mplib for convenient motion planning. For the grasp pose, you can reuse the data in the inverse kinematics problem. The trajectory should consist of reaching the mug, grasping the mug, and then lifting it up.

Reference You can check mplib tutorial for how to use mplib for motion planning of a realistic robot model.

Sub-Problem 3: Execute the Planned Trajectory with Your Controller Similar to the previous PID problem, you need to follow the trajectory using the PID controller. This time the trajectory is not given by human but computed from motion planning libraries such as mplib. Visualize the final execution result and submit the video recording of your system. Cheers! You have implemented the most important components in robotics and integrated them into a unified system!

Chapter 5

Reinforcement and Imitation Learning

5.1 DQN (HW5, RoboML)

CartPole: As shown in Fig. 5.1, we use a box to represent a cart whose mass is denoted as M. A pole is attached by an un-actuated joint to the cart, which moves along a frictionless track. Here, we use a sphere and a rod of length l to represent the pole. The mass of the sphere is denoted as m, and the mass of the rod can be ignored. The system is controlled by applying a force f to the cart. Force f is along the x-axis, and the position (x coordinate) of the system is denoted as x. At the very beginning, the pendulum is at its upright position with a small perturbation. The goal is to let the system back to its original state ($x = 0$ and $\theta = 0$).

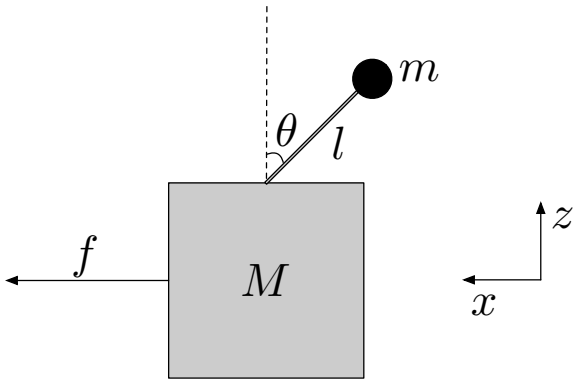


Figure 5.1: CartPole.

In this problem, you need to implement and train a DQN to solve the CartPole-Swingup task. Unlike in problem 1, $\theta = \pi$ at the very beginning. You need to control the system to swing up the pendulum and let it stay at the upright position.
Here is the basic information about the task:

- **observation**: the state vector $[x, \theta, \dot{x}, \dot{\theta}]$.

- **action**: a force of +10 or -10 to the cart instead of a continuous force.
- **reward**: you need to design a reward to solve the task.
- **done**: each episode has a fixed length of 500 steps, done=True at the last step.

We define the environment in `cartpole_swingup.py` and provide a framework for building and training a DQN in `dqn.py`. You need to fill in 17 blanks (marked as "__TODOXX__") in `dqn.py` to set up the hyperparameters, network architecture, optimizer, and some key steps of the algorithm. You also need to fill in 1 blank in `cartpole_swingup.py` to design the reward. You are not allowed or required to modify other codes.
The environment will return a binary label *success* at each step indicating whether $\theta \in [-12°, 12°]$. The task is considered to be solved when the number of successful steps in an episode is greater than 420.

1. Please report your answers for the $17+1$ blanks, your training log, and a video of the final result. (10pts)

   Hint: If you are not familiar with OpenAI Gym, please check their tutorial.

2. Can we directly apply the LQR solution in problem 1 to this CartPole-Swingup task? Why or why not? (ignore the structure difference of the two CartPole instances) (2pts)

5.2 PPO & RND (HW6, RoboML)

In this problem, we will play with Proximal Policy Optimization (PPO).

1. We have attached an implementation of PPO from OpenAI Spinning Up. Please try to read and play with the implementation, and then answer the following questions (14pts):

   - In *core.py*, we have the implementation of both `actor` and `critic`. What are the roles of `actor` and `critic` respectively? What are their input and output? Please describe the function of `MLPActorCritic::step()` (2pts).
   - There are two types of `actor` for both continuous and discrete actions. How are they implemented respectively? What is the difference in network design? (1pt)
   - In `MLPGaussianActor`, we don't use the network to predict the std of a normal distribution. How do we determine the std? Is there anything to take care of when tuning our network? (1pt)
   - What's the role of GAE-Lambda? How do we implement it and tune the parameter? (1pt)
   - When collecting experiences, how do we handle the reward/value of the last step of a trajectory? There are two cases. (1pt)
   - For a collected trajectory, how to calculate the return? How to calculate the advantage? Please note that the implementation uses an advantage normalization trick. What's the potential benefit of this trick? (2pts)
   - How do we update the `critic`? What's the loss function? (1pt)
   - How do we update the `actor`? What's the loss function? (1pt)
   - How to understand clipping ratio for the advantage? What is its motivation? (1pt)

- How to understand the KL-based early stopping trick? What's the potential benefit? (1pt)
- Please note that PPO is an on-policy algorithm, and we only use the latest collected trajectory to update the model. Why can't we use all the history to update like in DQN? (1pt)
- The implementation uses MPI to parallel the program. What's the main part we want to accelerate? Why is it more important in PPO than in DQN? (1pt)

2. In this part, you need to solve the PointMaze task with PPO. As shown in Fig. 5.2, a blue point, with an arrow indicating its direction, is moving in a "CSE291D maze", where the goal is the origin ($x = 0, y = 0$). In each episode, the point starts from a random location and has up to 1,000 steps to reach the goal. If the length of an episode is l, you will get a score of $1000 - l$ for this episode.

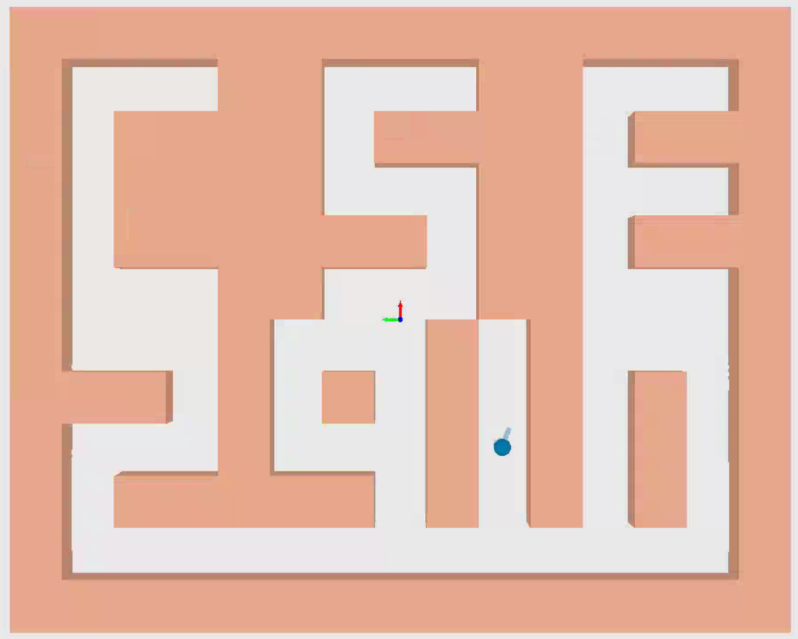


Figure 5.2: PointMaze.

Here are some of the basic information about the task:

- **observation**: The point has 3 DoFs: 2D position (x, y) and rotation θ. The observation is the state vector $[x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]$.
- **action**: The action is a 2D vector, and each element belongs to $[-1, 1]$. The first element controls the movement along the arrow direction, while the second element controls the rotation of the arrow. For the details of the transition, please check the function `PointMazeEnv::step()`.
- **reward**: You need to design a reward to solve the task. You are allowed to use heuristics about the map to design your reward.
- **done**: If $x^2 + y^2 \leq 0.5^2$ or 1000 steps are used up, the episode will terminate.

Based on the provided implementation, you need to design your reward function and tune your network. We have pointed out some parts that you may want to change in `point_maze.py`, `ppo.py`, and `core.py` (marked as "TODO"). You are also allowed to add more techniques that may lead to better performance. **However, you are NOT allowed to:**

- Directly control the point. The action should come from the output of your PPO policy network.
- Load the test cases during training. We may have additional test cases after your submission.
- Call `setqpos()`, `setqvel()`, `setqacc()`, `setqf()`, and etc.
- Modify the evaluate function and the environment except for the reward part.

After training your network, please run the evaluate function `evaluate()` in `ppo.py`. It will test your implementation for 200 cases and return a total score. You may want to use only a single thread to run the evaluation.

The credits for the PointMaze task are divided into three parts:

- (6pts) Your credits for this part is based on your total score:
  - 1pt: above 20,000.
  - 2pts: above 40,000.
  - 3pts: above 60,000.
  - 4pts: above 80,000.
  - 5pts: above 100,000.
  - 6pts: above 120,000.
- (8pts) Please submit your score to our leaderboard (https://forms.gle/QcTdg5C1wk8YYqKK7) before the deadline. Your credit for this part is based on your rank on the leaderboard. Specifically, rank 1 gets 8pts, rank 2 gets 7.7pts, rank 3 gets 7.4pts.....
- (3pts) Please submit a detailed report with your modification, training curve, final score, evaluation screenshot, and how to run your code. You also need to submit your code and the trained model.

Please make sure your code is easy to run and your report is clear. Otherwise, you may lose points.

Hints: 1. You may want to start from some simple cases to check the correctness. 2. You may want to use `env.render()` to visualize.

3. In deep reinforcement learning, we typically add some intrinsic rewards to encourage exploration. Please implement random network distillation (RND) as an exploration bonus in your PointMaze task. Please compare and analyze the results w/ and w/o RND. In your report, please include your implementation and discussion. (3pts)

   Paper reference: https://arxiv.org/pdf/1810.12894.pdf

5.3 Soft Actor-Critic

Overview Soft Actor-Critic is a popular algorithm for continuous control problems in reinforcement learning. Here, you are required to explore two simulated environments in OpenAI gym based on Mujoco, where gym provides a unified interface to define RL environments, and Mujoco is a widely used simulator which supports rigid body simulation.

Problem You are asked to use SAC algorithm to solve `HalfCheetah-v3` and `Swimmer-v3` in MuJoCo continuous control tasks. You can access these environments via OpenAI gym. You are allowed to use open-source codes, but make sure you understand the codes.

Specifically, you need to:

- Training SAC agents on these two environments. The expected return should be higher than 15000 for `HalfCheetah-v3` and 300 for `Swimmer-v3`.
- Run the experiments several times. Plot your training curves with standard deviations.
- Visualize your trained agent and record videos.
- Collect several demonstration trajectories by the trained agents. Please store the demonstrations and they will be used in problem 5.6.

Hints

- Install MuJoCo: `pip install free-mujoco-py`
- Install gym: `pip install gym`
- SAC example code: example-1, example-2
- Possible solution for OpenGL error on Ubuntu with Nvidia GPU: "Offscreen render does not work", "How to run OpenAI Gym .render() over a server".

5.4 Model-based RL, CEM & MPC (HW7, RoboML)

Task Description

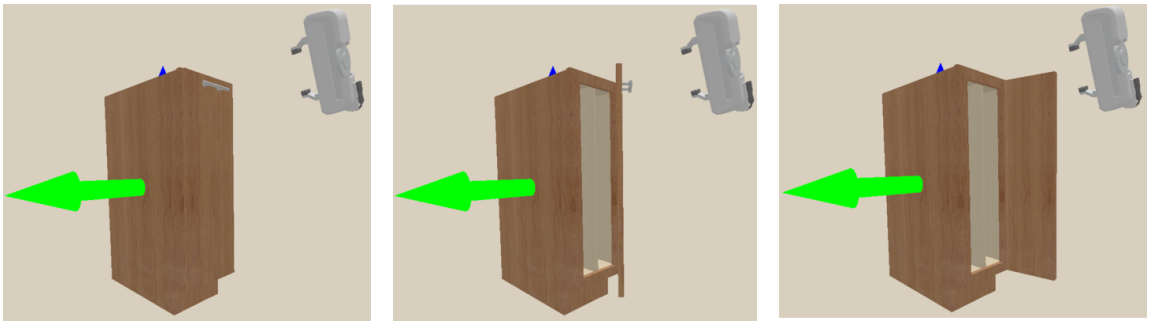


Figure 5.3: Open Cabinet.

As shown in Fig. 5.3, we are using a floating gripper to open a cabinet. The gripper first needs to move to the handle, grasp the handle, and then gradually open the door. In problem 1 and problem 2, the gripper starts from a fixed pose with a small perturbation, while in problem 3 and problem 4, the gripper starts from a random pose.

Action

The agent contains 8 joints, where the first three (dummy) joints are about the position, the next three (dummy) joints are about the rotation, and the last two joints are about the two fingers. We have provided some controllers for the joints (see `controller.py`), and you only need to provide the target signals for these controllers. Specifically, at each step, you need to provide an 8-dim action vector:

- **action[0:3]**: target signals for the velocity controllers of the three position joints;
- **action[3:6]**: target signals for the velocity controllers of the three rotation joints;
- **action[6:8]**: target signals for the position controllers of the two finger joints.

Each element should be in the range of $[-1, 1]$. Please check `controllers.py` and `agent.py` for more details.

Observation

The observation at each step is a 103-dimensional state vector containing the states of both the cabinet and the agent. Here are some of the states you may want to know:

- **obs[13]**: qpos of the cabinet joint;
- **obs[33]**: qvel of the cabinet joint;
- **obs[73:81]**: qpos of the agent joints;

- **obs[81:89]**: qvel of the agent joints;
- **obs[89:95]**: states of the 6 velocity controllers;
- **obs[95:103]**: states of the 2 position controllers.

See `sapien_env.py::get_state()` and `agent.py::get_state()` for more details.

Success Conditions

You have up to 150 steps for each episode. The success conditions include:

- **open_enough**: the qpos of the cabinet is greater than `env.target_qpos`;
- **object_static**: the velocity and angular velocity of the door link is small enough.

An episode is considered "solved" once the above two conditions are satisfied for 20 consecutive steps. See `sapien_env.py::_eval()` and `sapien_env.py::accumulate_eval_results()` for more details.

Reward

You need to design the reward function in `open_cabinet.py::compute_reward()`. It may be harder than previous homework to design a good reward function. You may want to divide the whole process into multiple stages and apply different rewards. Since we only need to open a single cabinet, you can hard-code information about the cabinet (e.g., rotation of the handle). You can find some useful information in `open_cabinet.py::get_custome_observation()`, such as the pose and the velocity of the end effector, and the position of the handle.
In previous homework, we have practiced model-free reinforcement learning (MFRL) methods (e.g., DQN and PPO). In this problem, we will play with model-based reinforcement learning (MBRL) and assume the ground truth dynamics model $p(s_{t+1} \mid s_t, a)$ (given by SAPIEN) is known.
You need to implement the cross-entropy method (CEM) and model predictive control (MPC). The pseudocodes of the two algorithms are shown below.

Algorithm 1: Cross Entropy Method (CEM)

Require: start state s_0, population size M, planning horizon T, # elites e, # iters I, μ, σ
  for $i = 1$ **to** I **do**
    Generate M action sequences according to μ and σ from normal distribution
    for $m = 1$ **to** M **do**
      Generate a trajectory $\tau_m = (s_0, a_1, \ldots, a_T, s_T)_m$ using the dynamics model $p(s_{t+1} \mid s_t, a_t)$ and the action sequence $a_{1:T,m}$.
      Calculate the cost of $a_{1:T,m}$ based on the rewards of τ_m.
    end for
    Update μ and σ using the top e action sequences.
  end for
  return μ, σ

Algorithm 2: Generating an episode using MPC

s = env.reset()
$\mu = \mathbf{0}, \sigma = 0.5\mathbf{I}$
while not done **do**
  μ, σ = CEM(s, μ, σ)
  $a = \mu[0, :]$
  s' =env.step(a)
  $s = s'$
  $\mu = \mu[1 : T]$.append$(\mathbf{0})$
  $\sigma = \sigma[1 : T]$.append$(0.5\mathbf{I})$
end while

CEM can be considered as an evolutionary-style optimizer. Taken a start state s_0 as input, it will return an action sequence of horizon T, which we can use in the future T time steps. Note that when updating the mean and variance of the actions, we assume that the actions across time are independent.
On top of CEM, one thing that can be done to give better planning results is model predictive control (MPC), where we only use the first action in the planned action sequence and re-run CEM for each time step.

1. (10pts) Please first design your reward function in `open_cabinet.py::compute_reward()`, and then implement CEM and MPC in `mpc.py` (marked as "TODO"). You are also allowed to modify the hyper-parameters. If your implementation is correct, you should be able to open the door with a relatively high probability, although it may take about one hour to search for a complete trajectory. Please report the following points in your PDF submission.

   - your reward function;
   - your implementation of CEM and MPC;
   - **complete** log of a successful episode.

2. (2pts) Please note that we have a flag `use_mpc`, which indicates we can have two algorithms MPC+CEM and CEM-only. In MPC+CEM, only the first action in the planned sequence is used. However, in CEM-only, we can strictly follow the whole planned sequence for the next T steps and then re-run CEM for another T steps. Please run and compare the two algorithms MPC+CEM and CEM-only.

3. (2pts) By setting the number of iterations I to 1, population size M to $M \cdot I$ (original I), and the number of elites e to 1, we can get a policy of random shooting with the same number of samples. Please run and compare this random shooting policy (w/o MPC) and CEM-only.

5.5 Reward Design for Manipulation

Overview Reward engineering is very important when you try to solve a task by RL, and usually it is even more important than the RL algorithm itself. In this problem, you are asked to design and optimize the reward function for a manipulation task: peg insertion.

Problem You can find the code of the peg insertion task here.
Specifically, you need to:

- Implement the `get_reward` function. Please iterate several rounds and find a version which is the most friendly to MPC and RL.
- Use MPC and RL to solve the task with the reward you designed. The expected test success rates should be higher than 90%.
- Compare the desired reward functions between MPC and RL.

Hint

- robosuite implemented several manipulation tasks with dense reward functions. You may check these environments for reference.
- Currently, this environment will not terminate, i.e., each episode is infinite. If you want to make the episode terminate in finite steps, you can try to add a `TimeLimit` wrapper. You can try different episode lengthes to see which one works best.
- You can refer to SAPIEN documentation for the functions you might need to use.

5.6 Behavior Cloning and GAIL

Overview Behavior Cloning(BC) and Generative Adversarial Imitation Learning(GAIL) are two classical imitation learning algorithms. BC train the agents, which input an observation and output an action, via doing supervised learning over demonstrations. While GAIL is an inverse RL algorithm, which converts the demonstrations to a reward function by using adversarial learning. For adversarial learning part, you may check problem 2.4 and 2.5

Problem In problem 5.3, you have stored demonstrations for `HalfCheetah-v3` and `Swimmer-v3`. Run behavior cloning and GAIL with these demonstrations. Specifically, you need to:

- Training both BC agents and GAIL agents on these two environments. The expected return should be similar to the experts(agents who collected the demonstrations).
- Try different number of demonstration trajectories to see the performance change of the imitators.
- Run the above-mentioned experiments several times. Plot your training curves with standard deviations.
- Visualize your trained agent and record videos.

Hints

- GAIL example code: example-1.

Chapter 6

Rendering and Simulation

6.1 Rendering: Rasterization

In this problem, you will implement a simple rasterization pipeline to draw various geometric shapes on screen. You can start with OpenGL or Vulkan. You are also allowed to use library extensions such as GLEW, GLFW and GLUT. We suggest you go with OpenGL if you are not familiar with the rendering pipeline, as more details need to be handled by you manually in Vulkan.
Hint: You can start by reading the following materials:
OpenGL: Learn OpenGL, opengl-tutorial
Vulkan: Vulkan Tutorial, Vulkan Guide

1. Draw a colored cube and display the cube on screen. Achieve colorful effect by assigning colors to vertices. Your resulted window should look like Fig. 6.1.

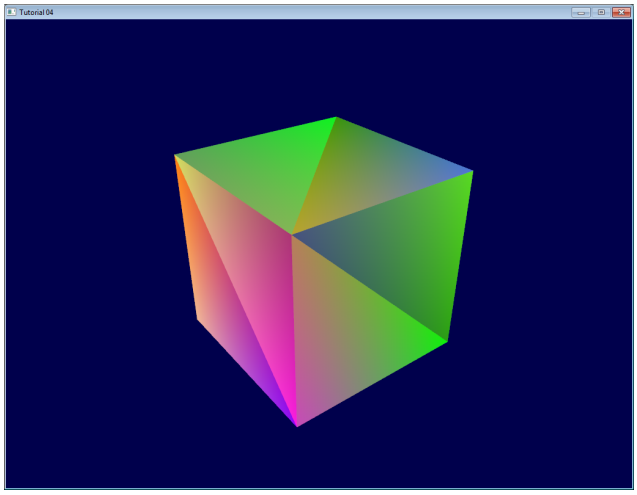


Figure 6.1: Example output of a colored cube.

2. (Optional) Draw a textured cube and display the cube on screen. You can use any cheetah texture you found online. Your resulted window should look like Fig. 6.1.

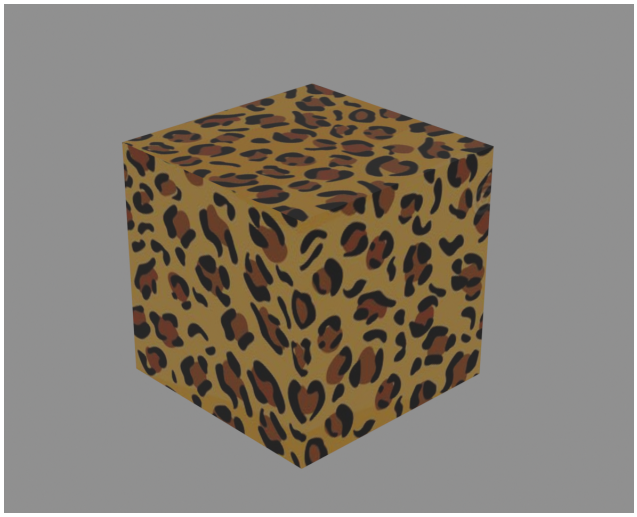


Figure 6.2: Example output of a textured cube.

6.2 Rendering: Ray-tracing

In this problem, you will need to answer several questions about ray tracing based rendering pipelines to show your general understanding of ray tracing. We suggest you go through these materials: Ray Tracing in One Weekend, Introduction to Ray Tracing, Wikipedia Page on Ray Tracing, and Ray Tracing in Vulkan.

1. Briefly describe how ray tracing pipelines generate images.
2. What are the major differences between ray tracing and rasterization pipelines? Briefly describe the benefits and drawbacks of each. Can these effects generated with a vanilla rasterization based pipeline: (1) reflection, (2) refraction, (3) shadows. Explain why.
3. Describe on a high level how ray tracing pipelines are implemented in code with Vulkan. What are the major stages? What kinds of shaders are involved?
4. Fig. 6.3 is an example output of an eye-based ray tracing renderer. Describe the general reason of noise in ray tracing outputs. Can you propose a few solutions to reduce the noise?
5. (Optional) Is there any pattern on noise distribution you can observe in 6.3? Describe the reason behind such pattern. With this information, what additional techniques can you introduce to improve the rendering quality?
6. SAPIEN 2.0 features convenient scene creation APIs and both rasterization and raytracing rendering backends. Design and create a scene and render a set of images or a video in SAPIEN. See

Fig. 6.4 as an example. Your scene does not to be very complex and you could use any object assets from public datasets *e.g.* ShapeNet. You can find the tutorials for SAPIEN 2.0 here.

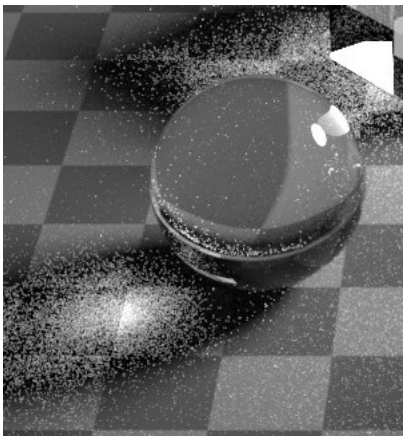


Figure 6.3: An example output of a ray tracing renderer.

Figure 6.4: An example output of a ray tracing renderer.

6.3 Rigid-body

Complete the notebook to implement a simple physical simulator that simulates the motion of rigid bodies. The notebook contains the detailed instructions and the example codes. You need complete the functions with a TODO marker. However, code provided in the repo is only for illustration purpose. You can modify anything in the code, as long as your simulator supports semi-implicit Euler, GJK-like collision detector and LCP. You can write your own test cases to illustrate the correctness of your implementation. We will check the code manually.

The tutorial here is a good starting point to help you understand what happens in the 2D rigid world. You just need to implement a modified version in python. You can also check the nice GDC talk. However in this problem you have to try a different apporach to solve constraints. There are many more advance reading materials for 3D rigid body simulation. Feel free to contact me if you need it.
Here are some hints:
1. we only consider the simple semi-implicit time-stepping approach. One can assume the acceleration a within the $[t, t+h]$ time interval is always constant and treat it the same as an impact ah. Thus we can compute the next state by the following equations

$$\dot{q}_{t+h} = \dot{q}_t + ah$$

$$q_{t+h} = q_t + \dot{q}_{t+h} h$$

2. for collision detection we only consider 2d shapes with limited number of vertices, so you can use $O(n^2 \log n)$ methods to find those contact points (guess why we have a log here?). Of course, you can implement the original GJK/EPA algorithm. However, please make sure that you use Minkowski difference in your code and when two edges overlap, you should return at least two points to span the full contact surface in the LCP problem.
3. Here is a draft/skeleton to derive the LCP problem. Let f_{ext} be the external wrench (force and torque), and f_c be the contact force, $v = \dot{q}$ be the velocity, q be the position. The rigid body dynamics (under Semi-implicit Euler) can be written as $v = v_1 + hf/M, q = q_0 + hv_0 + h^2 f/M \Rightarrow q = Af + b$ where $f = f_{ext} + J^T f_c$. The Jacobian matrix J_o of an object v_o is the matrix that transforms velocity of rigid body v into the velocity of contact point in the world frame. Thus $v_c = (J_a v_a - J_b v_b)^T n_c$ is the velocity that two objects separate. Here n_c is the contact normal. Our goal is to ensure that hv_c is greater than the penetration d_0, e.g. the new penetration depth $d_c = d_0 + hv \geq 0$. The contact force can only have the form of $f_c = \lambda_c n_c$ where λ_c is a scalar. Besides, it should satisfy that $\lambda \geq 0$ and if no contact in the next time step due to other contact forces, λ must be 0 when $d_c > 0$. This is the famous linear complementary constraint that $d_c^T \lambda$ should be 0. In the end, you need solve this LCP to find the contact force. Once you have the contact force, you can apply it to the object and do the integral. Usually you need to transform LCP into a quadratic programming problem, but you can also implement the famous Lemake algorithm. The projected Gauss Siedel is not very reliable in extreme cases.
4.For LCP with multiple objects, you should put everything together into big matrices since the constraints should be satisfied together. Besides, pay attention to the frame you use to represent torque. You can sum two torques only when they are in the same frame.

6.4 Smoothed-Particle Hydrodynamics (SPH)

In this problem, you will implement the Predictive-Corrective Incompressible SPH algorithm based on the paper `https://people.inf.ethz.ch/~sobarbar/papers/Sol09/Sol09.pdf`.
You can follow the following steps.

1. Download the template code here. This file has implemented the framework for the SPH simulator. You just need to fill in several crucial functions. This implementation is based on the taichi language. You can install taichi using pip, and you can test your code by running the python file directly.

2. (Optional) Learn taichi. However, it is possible that you just need to read the template code to understand taichi.

3. Implement all functions marked with a TODO. The important simulation variables are marked with IMPORTANT. You need to read the paper and the template code carefully and finish the implementation.

4. Run the python file, you should see a 2D water cube dropping into a sink and form water waves.

6.5 Finite Element Method (FEM)

Read the classical course on FEM course and the taichi's exmaple code on 2D material simulation. Write code to finish the following tasks:

- Task 1: revise the code so that it can simulate 2D cloth in the 3D space. This means the x and v variable should have an additional $z-$dimension.

- Task 2: revise the code so that it can simulate 3D objects in the 3D space. This means the elements are no-longer triangles but tetrahedron.

For both tasks, you need implement a simple orthogonal projector to help you visualize the edges in 3D space and save the results into videos. You can use this function to write and visualize videos in the jupyter notebook.

6.6 Material Point Method (MPM)

In this problem, you need learn the basic idea of the Material Point Method (MPM). You can read the course notes (It's suggested to read the FEM course in the previous problem, but this is not necessary) and play with the example code.
You then need finish the following tasks:

- augment the example code so that it can simulate particles of different type of materials including plasticine and snow. For plasticine, you can read the PlasticineLab paper. For snow, read the section 5 in this paper.

- In "grid_op" function, there is an example of using attractor to update the gird velocities to interfere the object motion. You need revise the code to add two small rigid spheres as the collider instead. HINT: you should view spheres as time-dependent boundary conditions that you can control the boundary shape at each substep.

- In the end, you should use the two rigid spheres to stack two snow spheres together into a snowman. Record the manipulation process into videos.

HINT: You do not need worry too much about the mathematical details, but you need understand the physical meaning of each variable in the example code and what are transferred from particles to grids and what are transferred back. You also need know where the code determines the material types for particles and how it relates to the formulas in the paper.