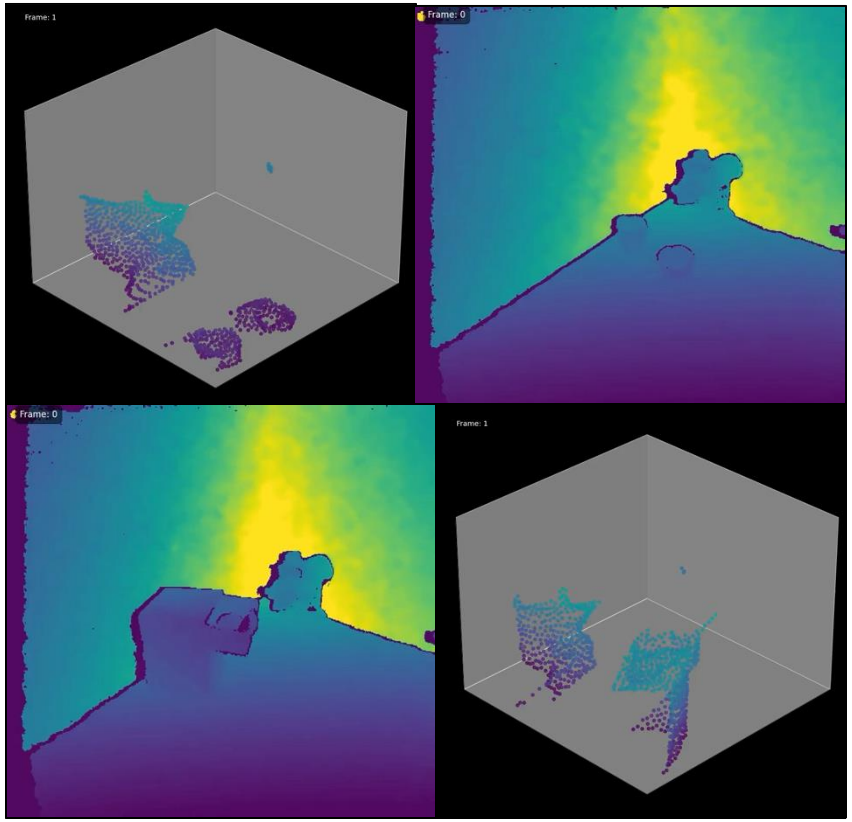
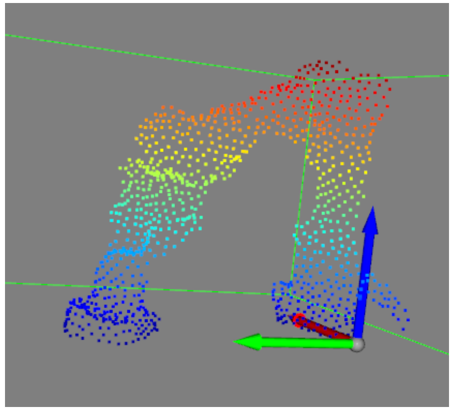
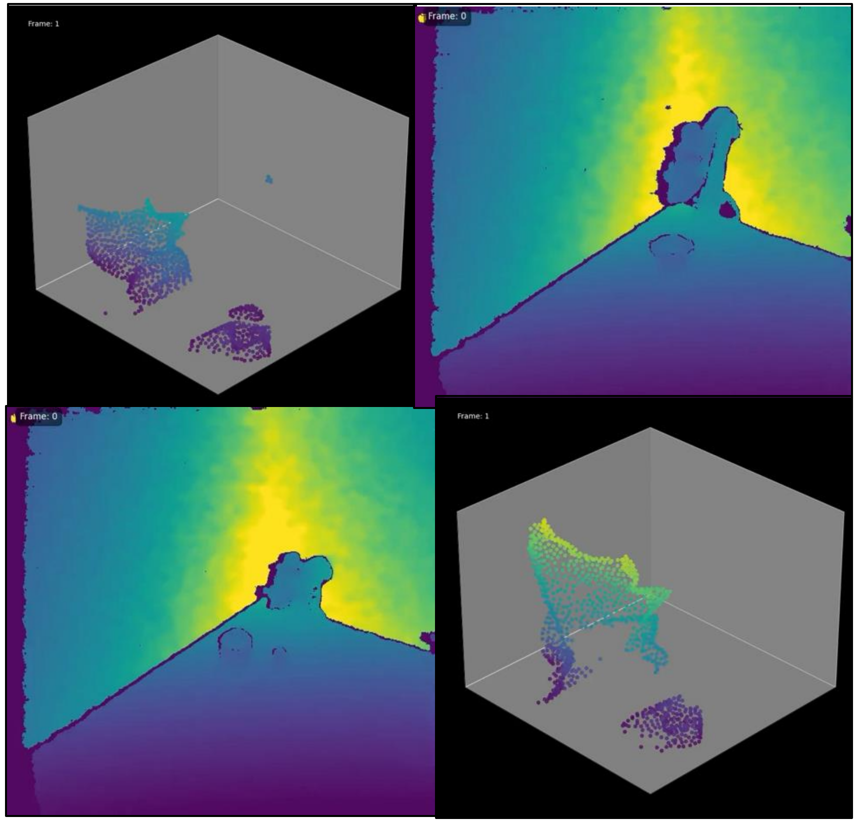
Robotic arm movement via 3D Diffusion policy

Chuyao Fu, Yufeng Wen， Yitao Zeng



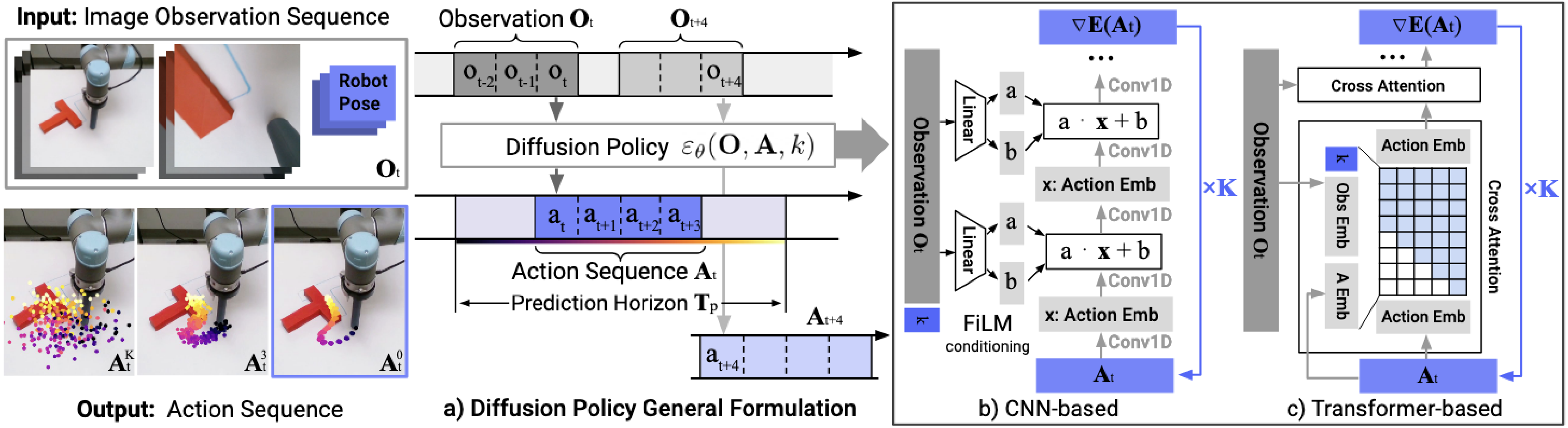
Diffusion Policy





Contents

Introduction of 3D Diffusion Policy



Diffusion policy VS Conventional ways

Conventional ways: requires **explicit environment modeling** and **path planning**, which are complex and error-prone.

Perception → Modeling → Path planning → Execution

Diffusion Policy: an end-to-end approach, directly generates action distribution

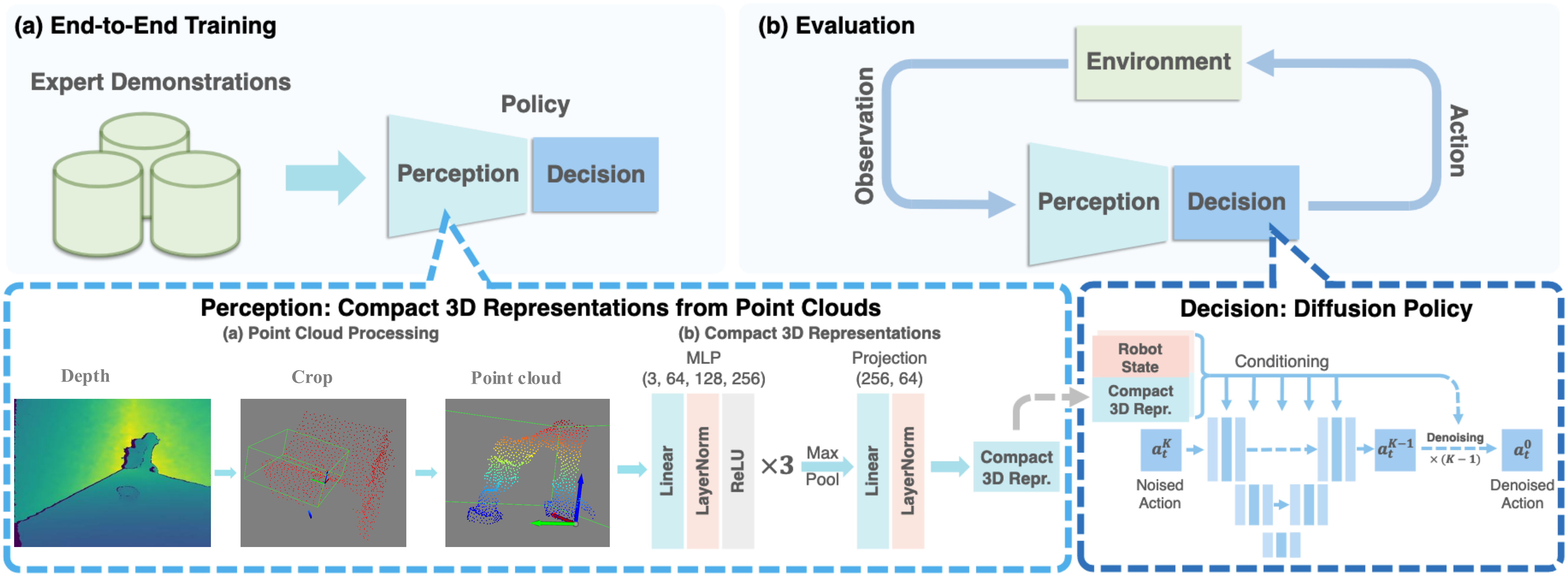
Single view-point cloud

Robot state

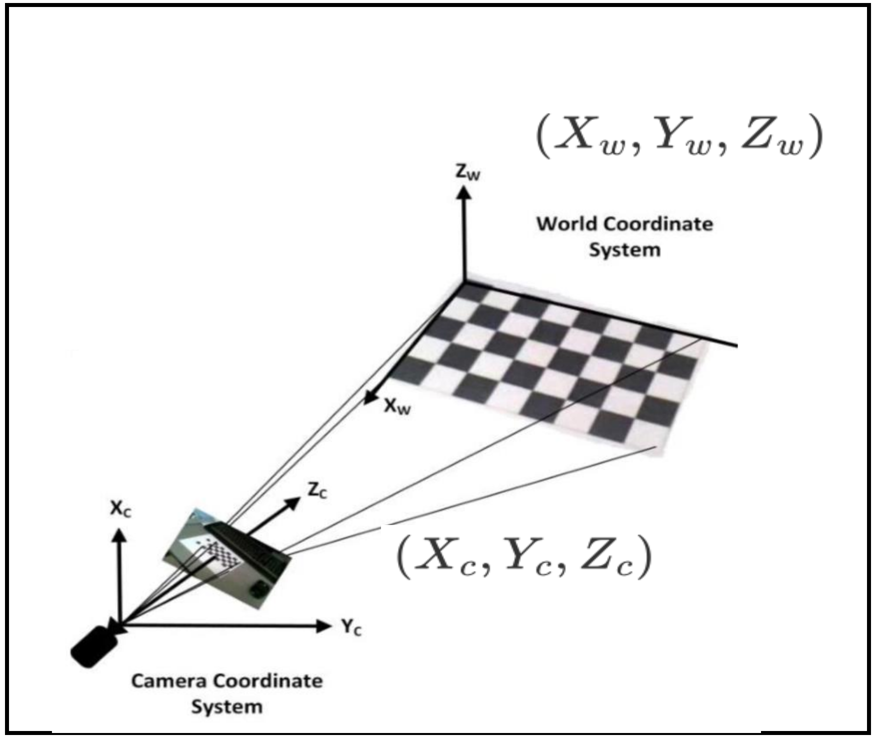
black-box neural network →

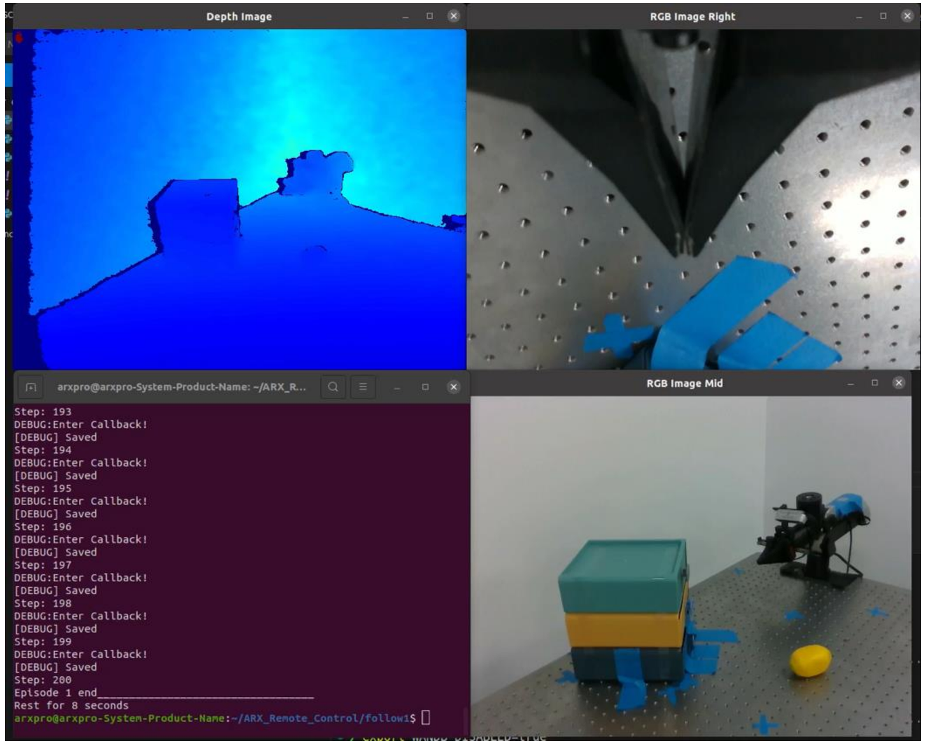
Action sequence

Introduction of DP3



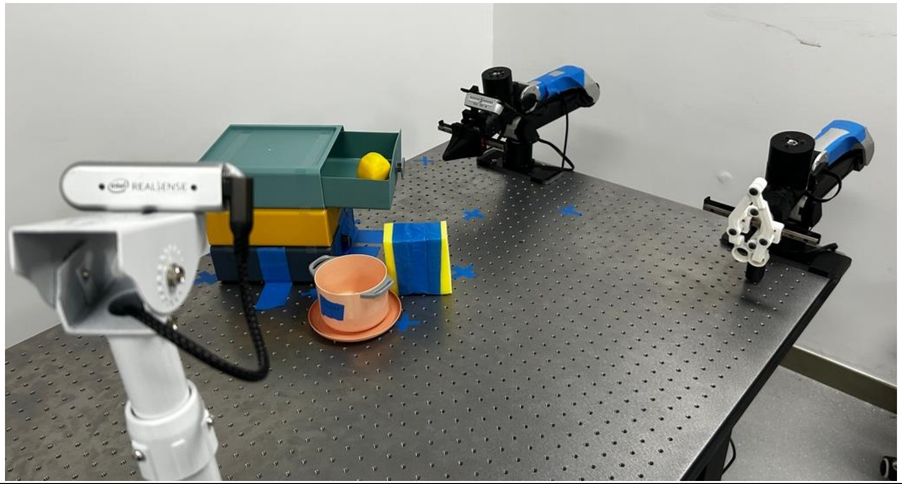
Data collection



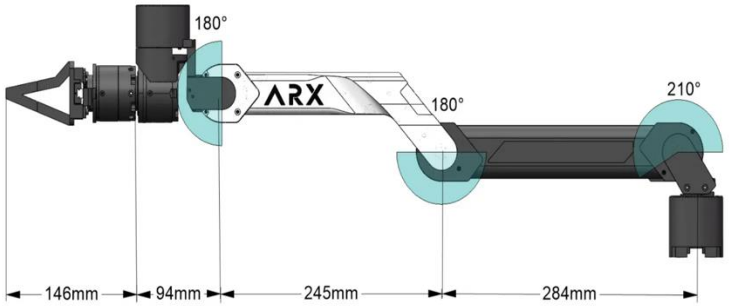


Data collection

Equipment：

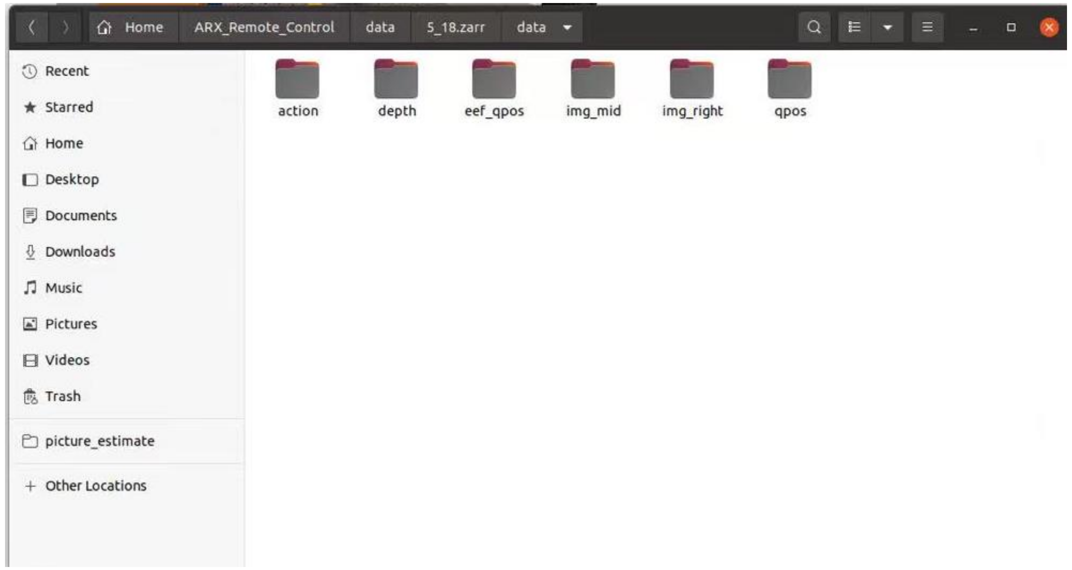


Real sense D435



方舟无限 ARX X5

Collected Data
- Action ☆
- Depth ☆
- eef_qpos
- Img_mid
- Img_right
- qpos ☆



Data collection

Code

Nodes for realsense cameras to publish **image messages**

Start-up procedure for remote operation of robotic arms

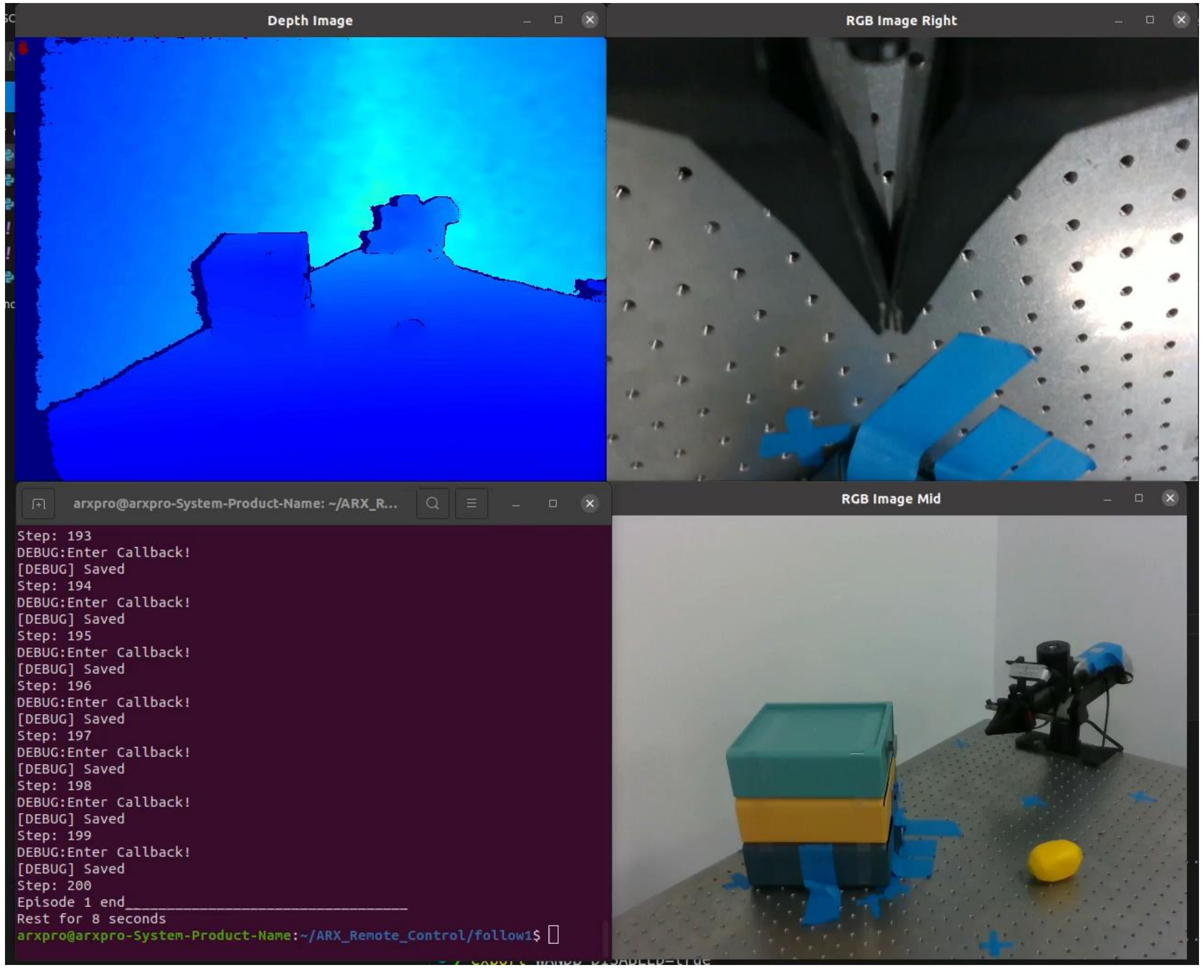
Sensor Data Synchronization and **Zarr** File Storage ★

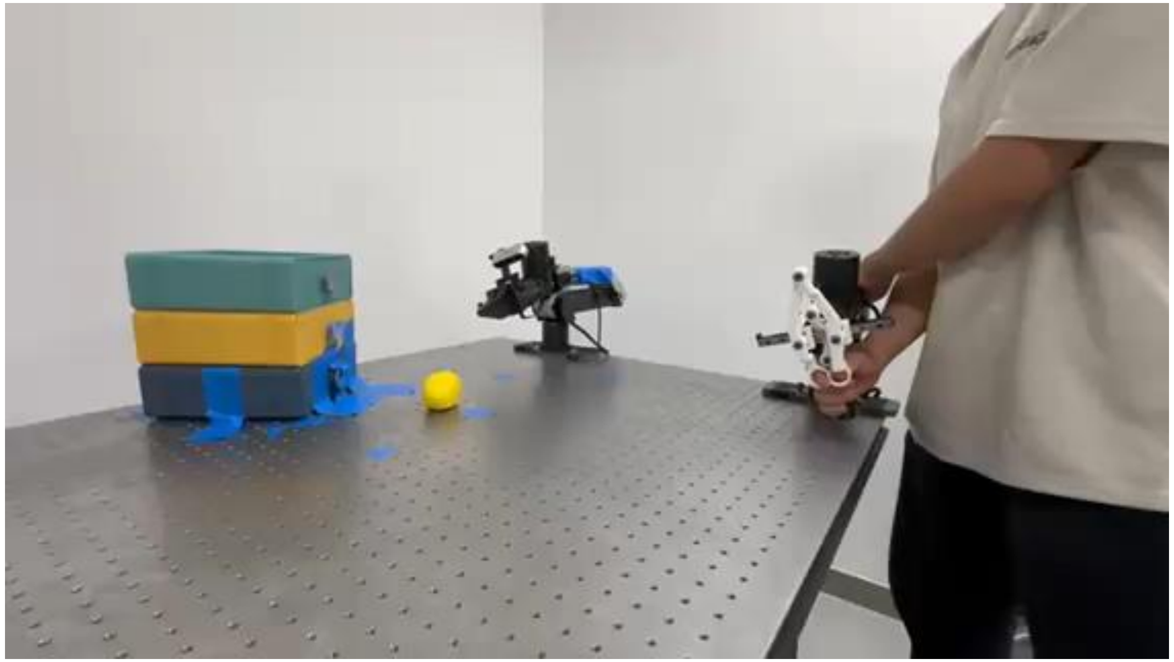
Based on Message Filter (Synchronization Frequency : about 10Hz)

Store collected data as array in the **callback function**

Transform to zarr files when all episodes are completed

Data collection

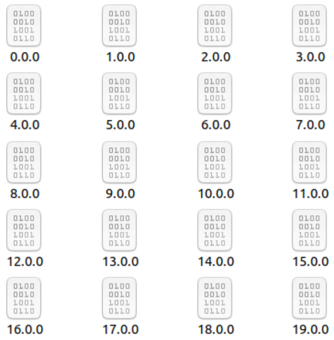




Data collection

Zarr storage

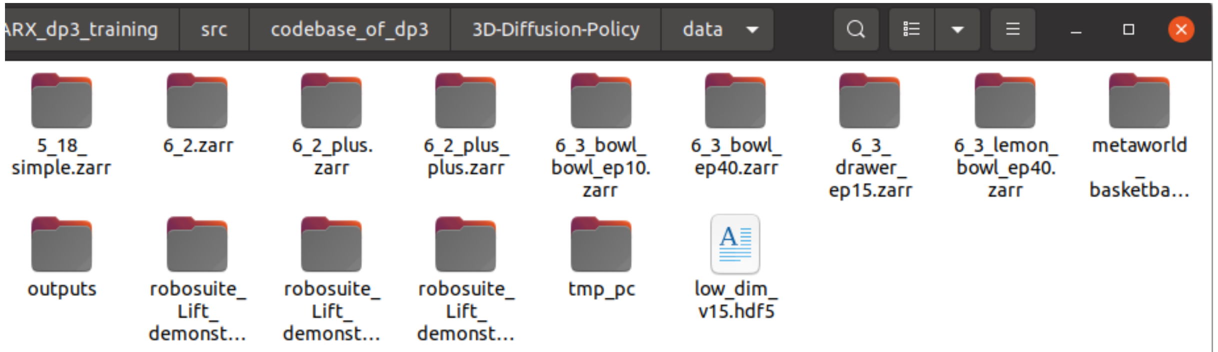
Property : Chuncked base



Advantage:

* Time efficient: Reads/writes only needed chunks (no full-file loading).

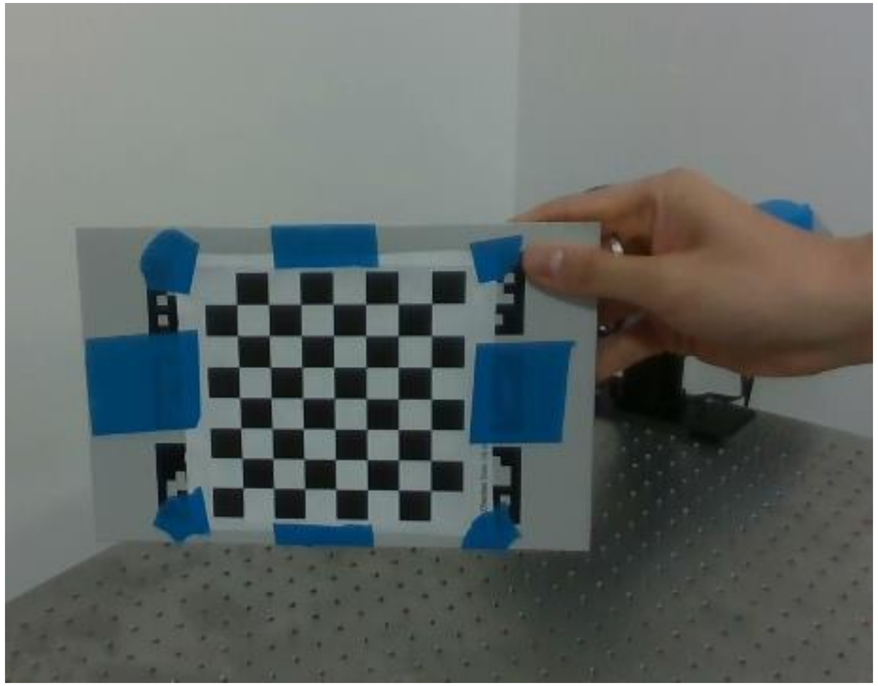
* Usage efficient: could just read a certain amount of frames, convenient for testing

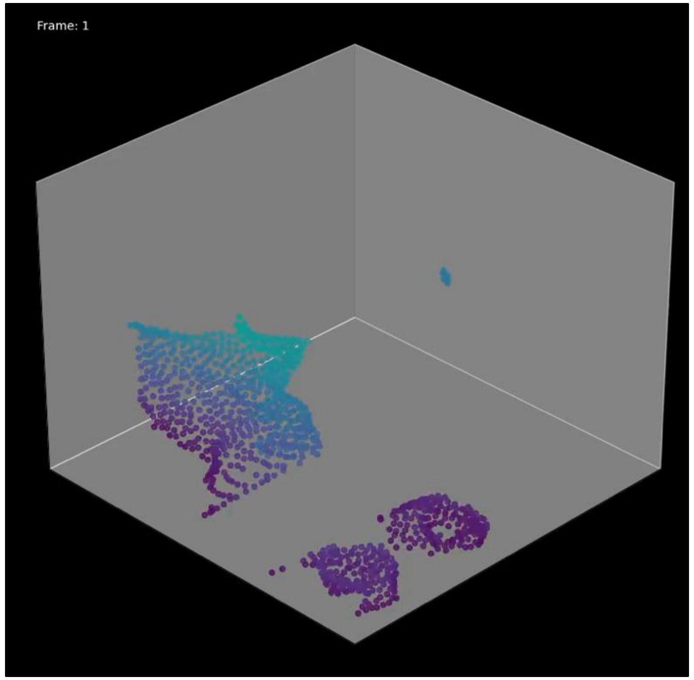


Code

```python
def read_in_depth(zarr_path: str) -> np.ndarray:
    """
    从 Zarr 文件读取深度数据（固定从 "depth" 数据集读取）
    """
    try:
        zarr_group = zarr.open(zarr_path, mode='r')

        # 更安全的检查方式
        if not any(key == "depth" for key in zarr_group.keys()):
            raise KeyError("Zarr 文件中必须包含 'depth' 数据集")

        depth_data = zarr_group["depth"]

        return np.array(depth_data)

    except zarr.errors.PathNotFoundError:
        raise FileNotFoundError(f"Zarr 路径不存在: {zarr_path}")
#storage path
zarr_path = "/home/arxpro/ARX_Remote_Control/data/5_18_simple.zarr/data"
output_zarr_path = "data/5_18_simple.zarr/data/pcd"
#reading from zarr
depth_from_robot = read_in_depth(zarr_path)
```

Data preprocessing



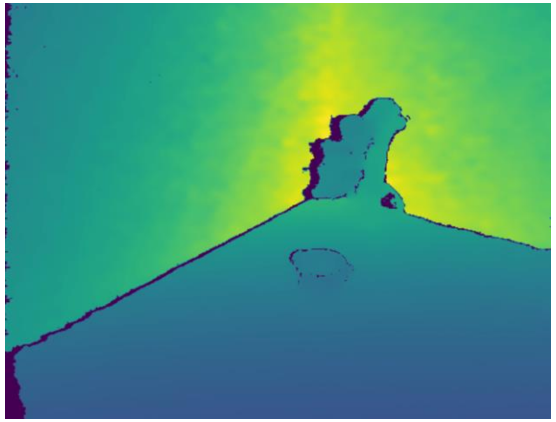


Preprocessing

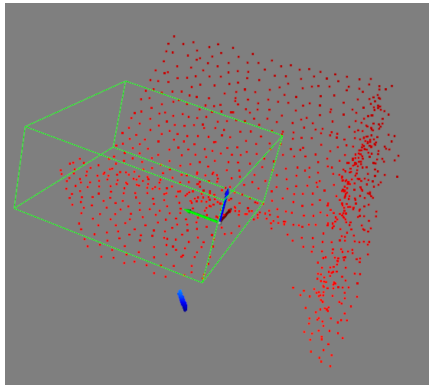
*Depth image is not suitable for training

*Image still contain the walls and desktop, too much irrelevant information

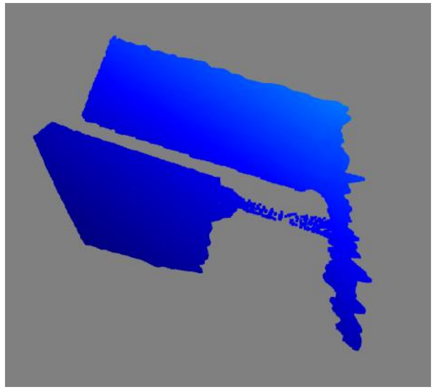
*The amount of points are too large, making train and deployment too slow



still depth

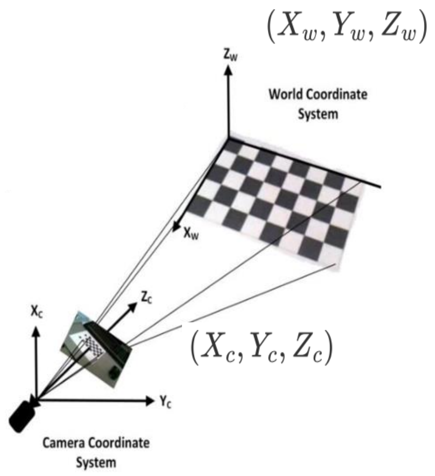


irrelevant information



Over 200,000 points!

Camera Calibrition—Intrinsic Parameter



$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Method :Zhang's calibration method

Planar calibration plates and 2D projection models

Procedure:

1.**Checkerboard Preparation:**

Use an **8×8 grid** with **15mm squares**.

2.**Multi-Angle Image Capture:**

15–20 images of the checkerboard from varying distances/angles.

3.**Feature Detection:**

Detect checkerboard **corners** in 2D

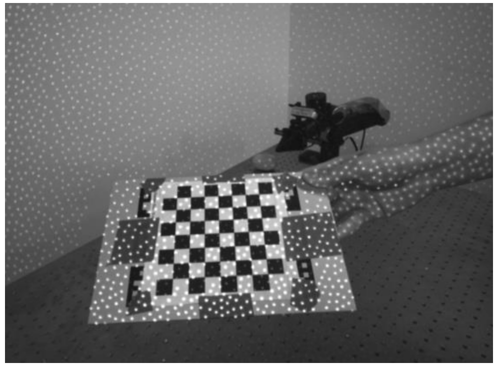
4.**3D-2D Mapping:** (the **XY plane Z=0)**

compute **precise 3D coordinates** for all corners.

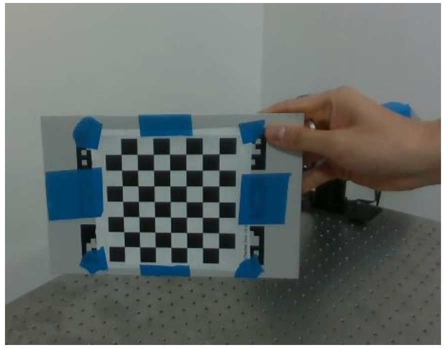
5.**Parameter Estimation:**

Solve for:

1. **Intrinsic matrix** (camera parameters).
2. **Distortion coefficients** 畸变 (radial/tangential).



IR_image



RGB_image

Camera Calibrition—Extrinsic Parameter

purpose:

Obtain the pose relationship from camera to robotic arm base(R and t)

steps:

1: T_c_to_w

Use the PNP method is used to obtain the rotation and translation matrix from camera coordinate system to the calibration plate coordinate system, origin is the upper left corner.(the green)

2. T_w_to_b

Measure the actual rotation and translation process from the origin of the calibration plate to the origin of the robotic arm base in the coordinate system : -118mm along the x-axis, 108mm along the y-axis, -90° clockwise rotation about the x-axis. T_w_to_b=T1*T2*T3

3.Calculate T_c_to_b

T_c_to_w* T_w_to_b=T_c_to_b

T'*相机坐标系下坐标=机械臂下坐标

```
T'=T^{-1} =
[[ 0.75422983  0.23041472 -0.6148548   468.233    ]
 [-0.65643515  0.28624051 -0.69796795 1090.323    ]
 [ 0.01517426  0.93004055  0.36714346 -472.788    ]
 [ 0.          0.          0.            1.       ]]
```

Camera Calibrition—Parameter Validation

Intrinsic Parameter：

Compare the reprojection error with the manufacturer-provided、 the calibration and the theoretical value

$$f_x = \frac{\text{图像宽度}}{2 \times \tan(\text{水平FOV}/2)}$$

$$f_y = \frac{\text{图像高度}}{2 \times \tan(\text{垂直FOV}/2)}$$

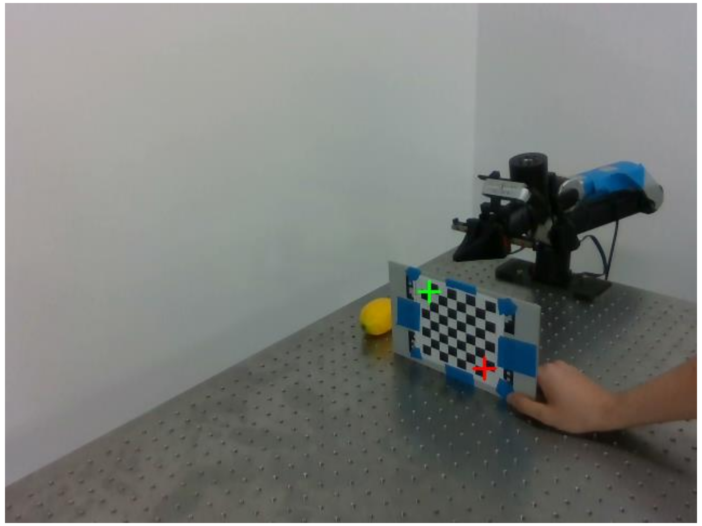
```
======== rgb标定参数对比 ========
| 参数类型 | 重投影误差（像素）|
| 新标定参数 | 0.1699 |
| 厂家提供参数 | 0.1943 |
| 理论参数 | 0.5294 |
```

```
内参1胜率: 2/21 (9.52%) (原厂家)
内参2胜率: 17/21 (80.95%) (标定)
内参3胜率: 2/21 (9.52%) (理论)
综合结果: 内参2 (标定)  最佳
```

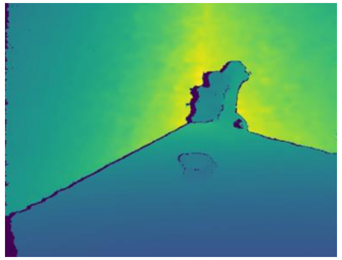
Intrinsic Parameter： ：

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Compare the difference between the calculated and actual coordinates at the point in the upper left corner of the calibrated plate



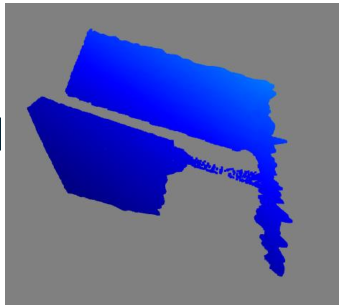
Transforming depth into point cloud:



Using od3 to transform

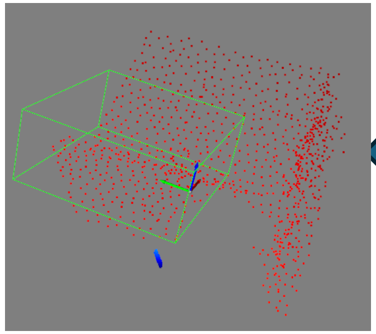
```python
od_cammat = cammat2o3d(self.cam_mat, self.img_width, self.img_height)
od_depth = o3d.geometry.Image(depth_data)
o3d_cloud = o3d.geometry.PointCloud.create_from_depth_image(od_depth, od_cammat)
# 计算相机到世界的变换矩阵
c2w = self.extrinsic_matrix
transformed_cloud = o3d_cloud.transform(c2w)
```

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$



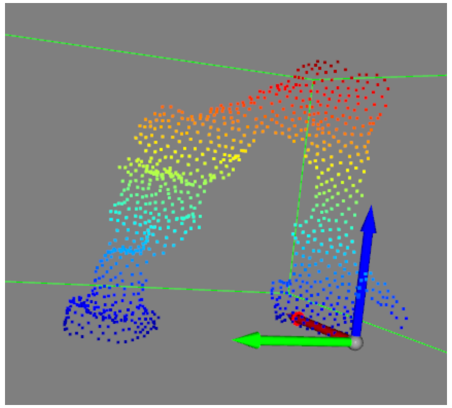
Farthest point sampling

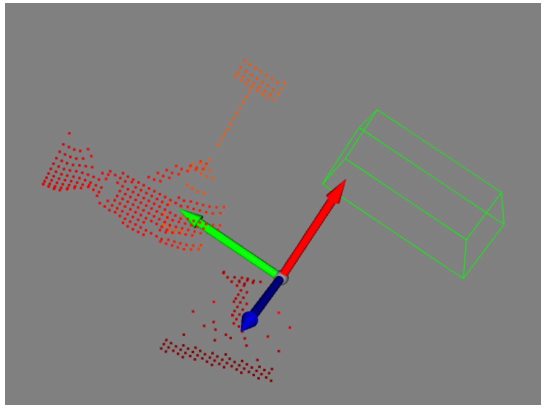
```python
def farthest_point_sampling(points, num_points=1024, use_cuda=True): #For test, increase the number of th
    #points = np.asarray(points, dtype=np.float32)
    if points.size == 0:
        raise ValueError("输入点云为空数组！")
    if points.ndim != 2 or points.shape[1] != 3:
        raise ValueError(f"输入需为 (N,3) 数组，但得到形状 {points.shape}")
    K = [num_points]
    if use_cuda:
        points = torch.from_numpy(points).cuda()
        sampled_points, indices = torch3d_ops.sample_farthest_points(points=points.unsqueeze(0), K=K)
        sampled_points = sampled_points.squeeze(0)
        sampled_points = sampled_points.cpu().numpy()
    else:
        points = torch.from_numpy(points)
        sampled_points, indices = torch3d_ops.sample_farthest_points(points=points.unsqueeze(0), K=K)
        sampled_points = sampled_points.squeeze(0)
        sampled_points = sampled_points.numpy()

    return sampled_points, indices
```

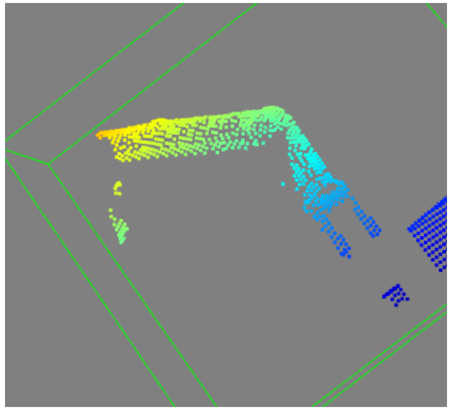


Cropping

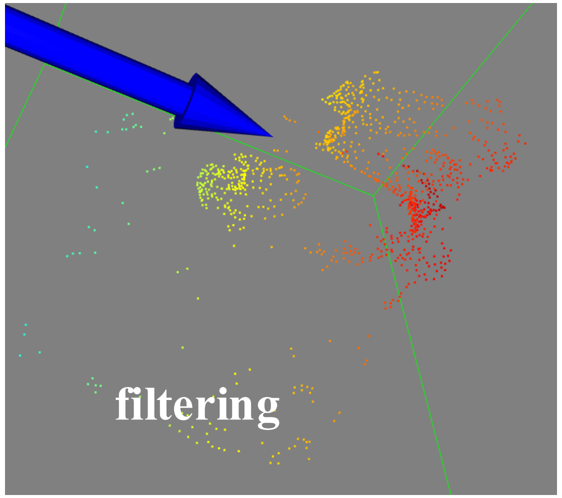
```python
def preprocess_point_cloud(points, use_cuda=True):
    num_points = 1024

    WORK_SPACE = [
    [-0.14, 0.1],
    [-0.03, 0.2],
    [0, 0.1]
]
    # scale
    points = points[..., :3]*0.00025000000118743628
     # crop
    points = points[np.where((points[..., 0] > WORK_SPACE[0][0]) & (points[..., 0] < WORK_SPACE[0][1]) &
                             (points[..., 1] > WORK_SPACE[1][0]) & (points[..., 1] < WORK_SPACE[1][1]) &
                             (points[..., 2] > WORK_SPACE[2][0]) & (points[..., 2] < WORK_SPACE[2][1]))]
    point_xyz = points[..., :3]
    points_xyz, _= farthest_point_sampling(point_xyz, num_points, use_cuda)
    return points_xyz
```



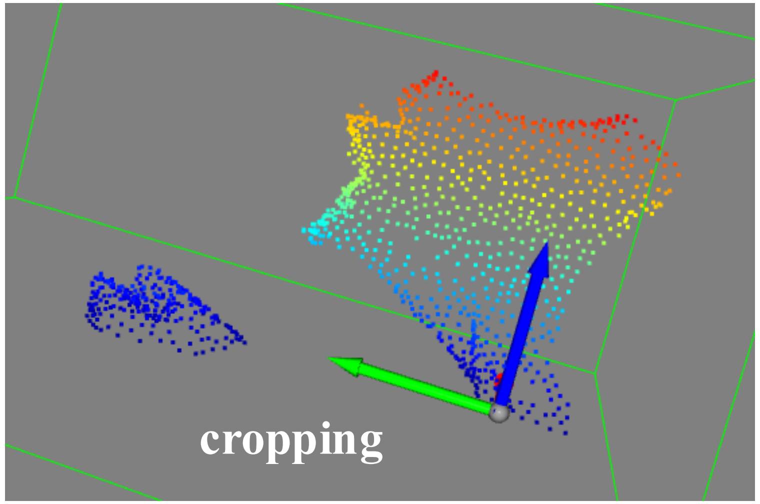


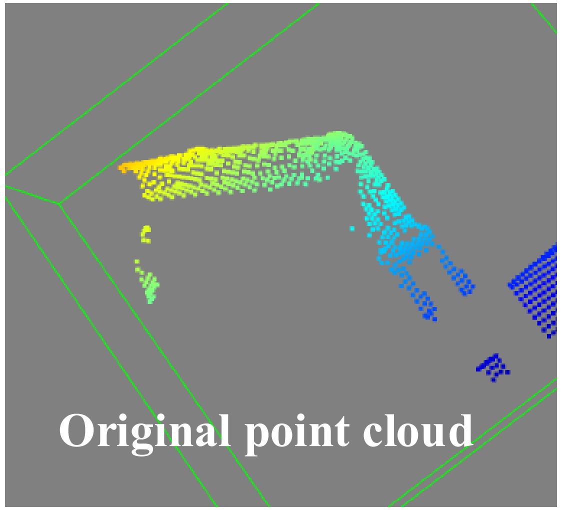


Some other attempts:

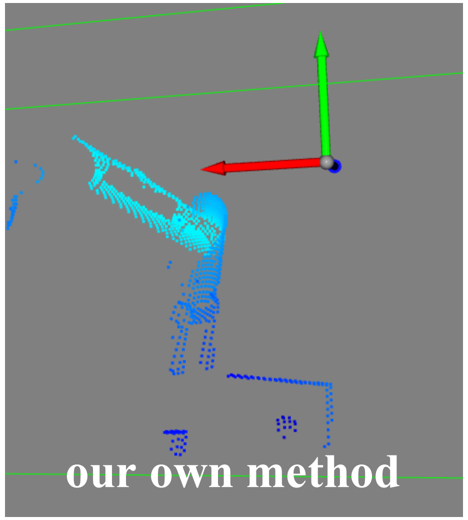


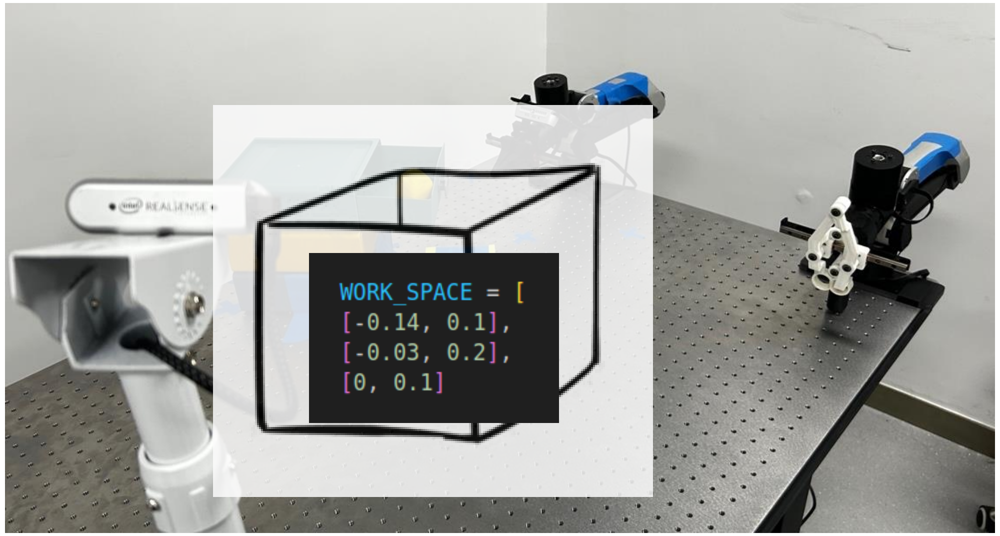
Filter VS Cropping



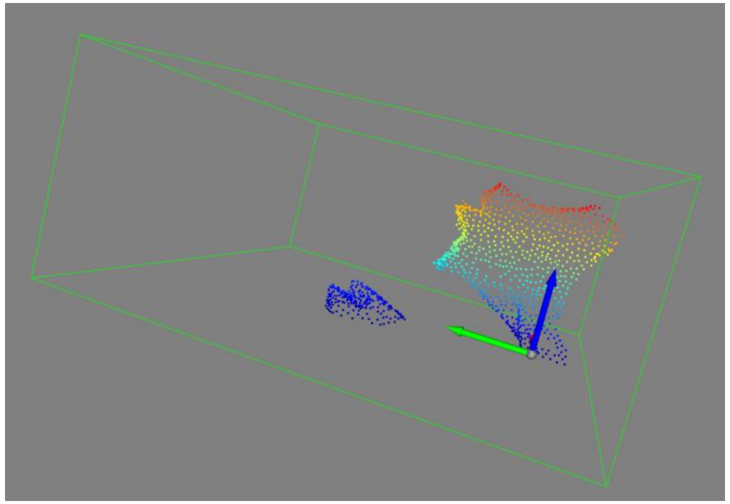


Standard sim dataset





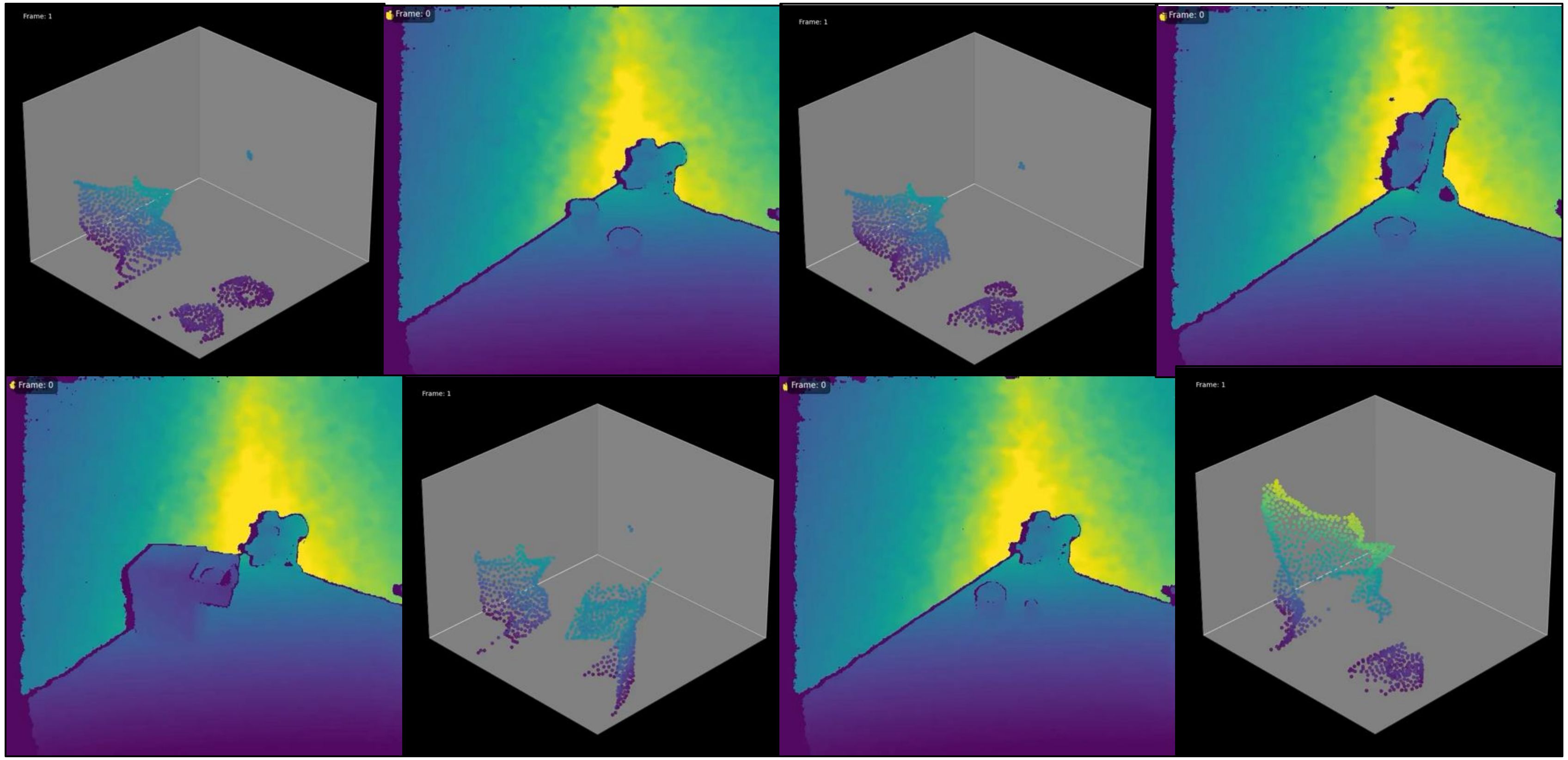
Using a box to find workspace



Results of preprocessing

Two bowls

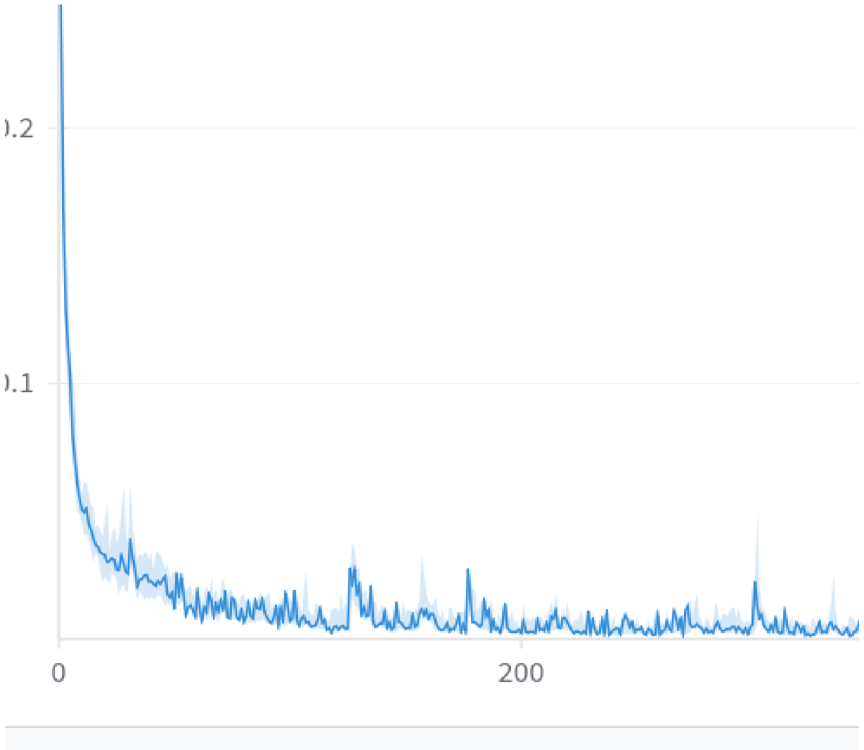
Lemon in bowl

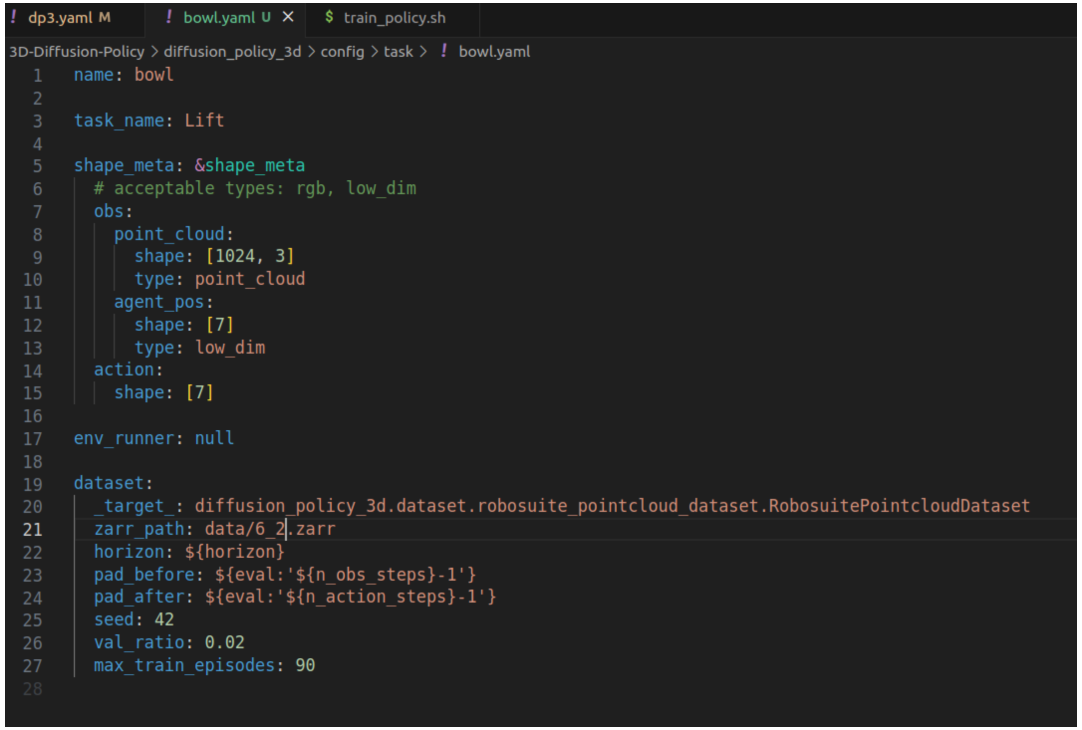


Drawer

Lifting one bowl

Training






```yaml
name: bowl

task_name: Lift

shape_meta: &shape_meta
  # acceptable types: rgb, low_dim
  obs:
    point_cloud:
      shape: [1024, 3]
      type: point_cloud
    agent_pos:
      shape: [7]
      type: low_dim
  action:
    shape: [7]

env_runner: null

dataset:
  _target_: diffusion_policy_3d.dataset.robosuite_pointcloud_dataset.RobosuitePointcloudDataset
  zarr_path: data/6_2.zarr
  horizon: ${horizon}
  pad_before: ${eval:'${n_obs_steps}-1'}
  pad_after: ${eval:'${n_action_steps}-1'}
  seed: 42
  val_ratio: 0.02
  max_train_episodes: 90
```

Training

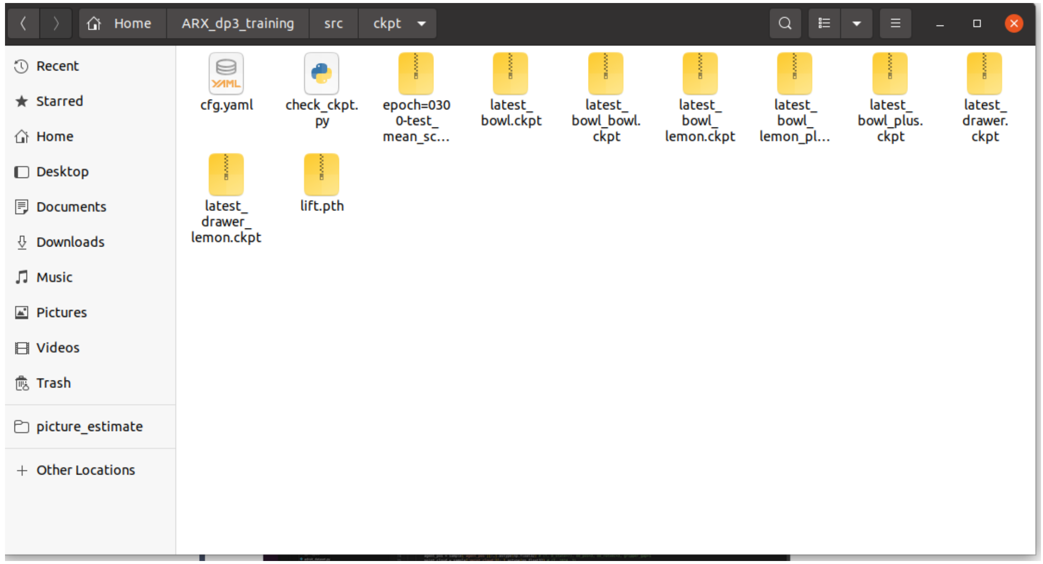
```
dp3.yaml M    bowl.yaml U ×    train_policy.sh
3D-Diffusion-Policy > diffusion_policy_3d > config > task > bowl.yaml
name: bowl

task_name: Lift

shape_meta: &shape_meta
  # acceptable types: rgb, low_dim
  obs:
    point_cloud:
      shape: [1024, 3]
      type: point_cloud
    agent_pos:
      shape: [7]
      type: low_dim
  action:
    shape: [7]

env_runner: null

dataset:
  _target_: diffusion_policy_3d.dataset.robosuite_pointcloud_dataset.RobosuitePointcloudDataset
  zarr_path: data/6_2.zarr
  horizon: ${horizon}
  pad_before: ${eval:'${n_obs_steps}-1'}
  pad_after: ${eval:'${n_action_steps}-1'}
  seed: 42
  val_ratio: 0.02
  max_train_episodes: 90
```

```
obs:
  point_cloud:
    shape: [1024, 3]
    type: point_cloud
  agent_pos:
    shape: [7]
    type: low_dim
action:
  shape: [7]
```

```
defaults:
  - task: metaworld_reach-wall

name: train_dp3

task_name: ${task.name}
shape_meta: ${task.shape_meta}
exp_name: "debug"

horizon: 16
n_obs_steps: 2
n_action_steps: 8
n_latency_steps: 0
dataset_obs_steps: ${n_obs_steps}
keypoint_visible_rate: 1.0
obs_as_global_cond: True

policy:
  _target_: diffusion_policy_3d.policy.dp3.DP3
  use_point_crop: true

  diffusion_step_embed_dim: 128
  kernel_size: 5
  n_groups: 8
  down_dims:
  - 512
  - 1024
  - 2048

  crop_shape:
  - 114
  - 114
  encoder_output_dim: 64
  horizon: ${horizon}
  n_action_steps: ${n_action_steps}
  n_obs_steps: ${n_obs_steps}

  noise_scheduler:
    _target_: diffusers.schedulers.scheduling_ddim.DDIMScheduler
    num_train_timesteps: 100
    beta_start: 0.0001
    beta_end: 0.02
    beta_schedule: squaredcos_cap_v2
    clip_sample: True
    set_alpha_to_one: True
    steps_offset: 0
    prediction_type: sample


  num_inference_steps: 10
  obs_as_global_cond: ${obs_as_global_cond}
  shape_meta: ${shape_meta}
```

```
horizon: 16
n_obs_steps: 2
n_action_steps: 8
n_latency_steps: 0
dataset_obs_steps: ${n_obs_steps}
keypoint_visible_rate: 1.0
obs_as_global_cond: True
policy:
  _target_: diffusion_policy_3d.policy.dp3.DP3
  use_point_crop: true
  diffusion_step_embed_dim: 128
  kernel_size: 5
  n_groups: 8
  down_dims:
  - 512
  - 1024
  - 2048
```

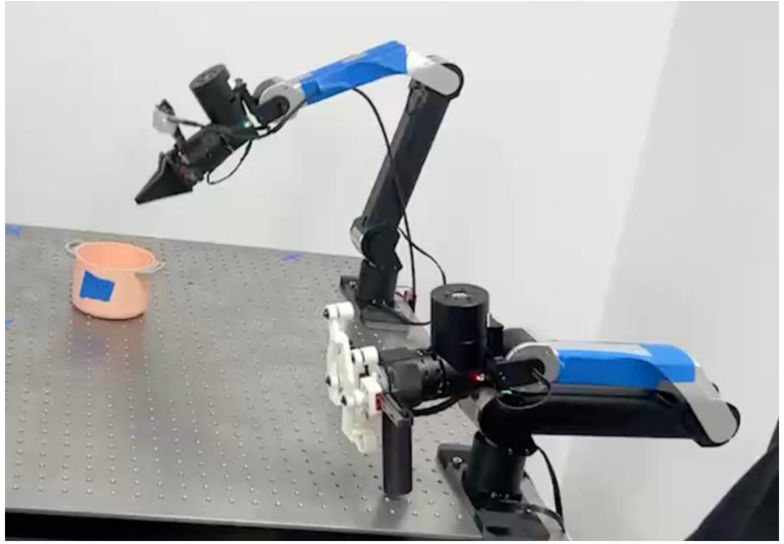
Training

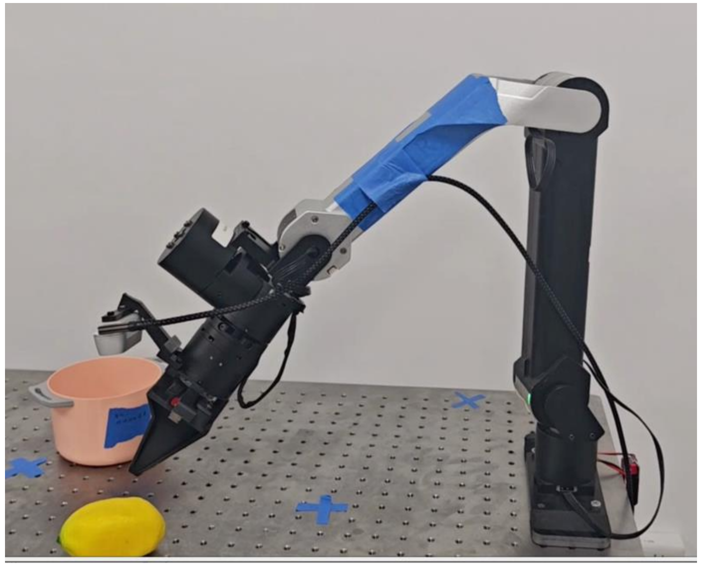


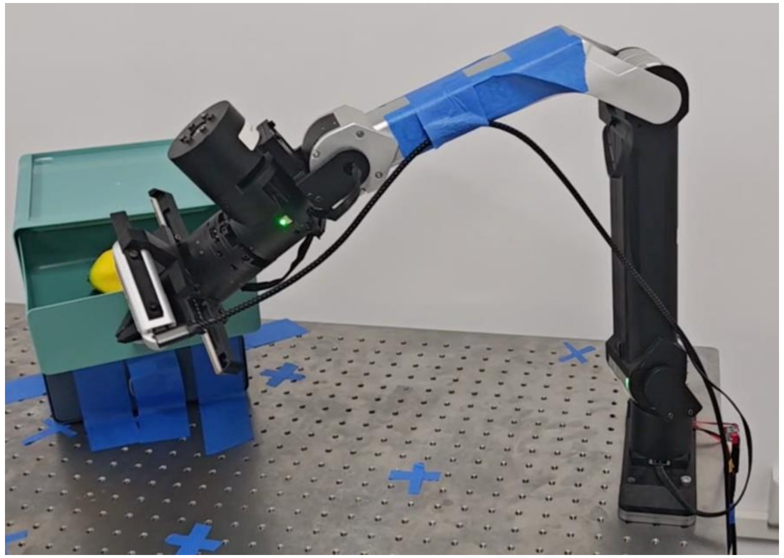

```python
    def __len__(self) -> int:
        return len(self.sampler)

    def _sample_to_data(self, sample):
        agent_pos = sample['agent_pos'][:,].astype(np.float32) # (T, D_state=7), ee_pos=3, ee_rotvec=3, gripper_gap=1
        point_cloud = sample['point_cloud'][:,].astype(np.float32) # (T, 1024, 3)

        data = {
            'obs': {
                'point_cloud': point_cloud, # T, 1024, 3
                'agent_pos': agent_pos, # T, D_pos
            },
            'action': sample['action'].astype(np.float32) # T, D_action=7
        }
        return data

    def __getitem__(self, idx: int) -> Dict[str, torch.Tensor]:
        sample = self.sampler.sample_sequence(idx)
        data = self._sample_to_data(sample)
        torch_data = dict_apply(data, torch.from_numpy)
        return torch_data
```

Deployment

Parameter Initialization → Loading Policy → Obtaining Observations → Policy Inference → Action Execution







Deployment

Parameter Initialization

Control Frequency : 10 Hz (Desired time to perform a step)
Horizon : 16 (Determined in policy training)
Steps Per Inference : 8 to 12 (Number of actions to be executed)
Video_capture_fps : 30 (Camera Data Transfer Frequency)

Loading Policy

```
(dp3) arxpro@arxpro-System-Product-Name:~/ARX_dp3_training/src/scripts$ python e
val_real_ros.py
[DP3Encoder] point cloud shape: [1024, 3]
[DP3Encoder] state shape: [7]
[DP3Encoder] imagination point shape: None
[PointNetEncoderXYZ] use_layernorm: True
[PointNetEncoderXYZ] use_final_norm: layernorm
[DP3Encoder] output dim: 128
[DiffusionUnetHybridPointcloudPolicy] use_pc_color: False
[DiffusionUnetHybridPointcloudPolicy] pointnet_type: pointnet
----------------------------------
Class name: DP3
  Number of parameters: 255.1489M
   _dummy_variable: 0.0000M (0.00%)
   obs_encoder: 0.0637M (0.02%)
   model: 255.0852M (99.98%)
   mask_generator: 0.0000M (0.00%)
----------------------------------
```

```python
# load checkpoint and policy
ckpt_path = input_path
device = torch.device("cuda:0")
payload = torch.load(open(ckpt_path, "rb"), map_location="cuda:0", pickle_module=dill)
cfg = payload["cfg"]
model: DP3 = hydra.utils.instantiate(cfg.policy)
ema_model: DP3 = None
if cfg.training.use_ema:
    try:
        ema_model = copy.deepcopy(model)
    except: # minkowski engine could not be copied. recreate it
        ema_model = hydra.utils.instantiate(cfg.policy)
model.to(device)
if ema_model is not None:
    ema_model.to(device)

if cfg.training.use_ema:
    ema = hydra.utils.instantiate(cfg.ema,model=ema_model)
```

Deployment

Obtaining Observations

Obs_ring_buffer : After entering the callback function of the approximatetimesynchronizer , observation data is deposited into the buffer.

Convert depth images to point clouds in real time

```
arxpro@arxpro-System-Product-Name: ~/ARX_dp3_training/s...
Generate PC: 15.9ms
Started!
enter while2
t_cycle_end:173909.71723556
K is 6
Count is 9
obs_timestamps1:[0. 0.]
Got Observation!
Observation: 0.2ms
Generate PC: 16.7ms
Generate PC: 24.5ms
Generate PC: 19.3ms
Inference: 279.0ms
obs_timestamps:[0. 0.]
timestamps:[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7]
Execute Action!
Action:[[-0.04325804  0.11536891  0.09794503 -0.07716742 -0.05161217  0.0250869
  -0.04166883]
 [-0.04625054  0.09838625  0.07194603 -0.07778218 -0.06624506  0.04819524
  -0.0277885 ]
 [-0.04877763  0.12521763  0.10033099 -0.0825344  -0.07098921  0.06264765
  -0.01435836]
 [-0.05409541  0.17950112  0.1362288  -0.09163591 -0.07338552  0.07480673
  -0.03649284]
```

Deployment

Policy Inference

```
        # 示例输出: {'point_cloud': (1024, 3), 'agent_pos': (7,)}
        action_dict = policy.predict_action(obs_dict)
        print(f"Inference: {(time.perf_counter()-t1)*1000:.1f}ms")

    np_action_dict = dict_apply(action_dict,
                                lambda x: x.detach().to('cpu').numpy())
    action = np_action_dict['action'].squeeze(0)
```

Action Execution

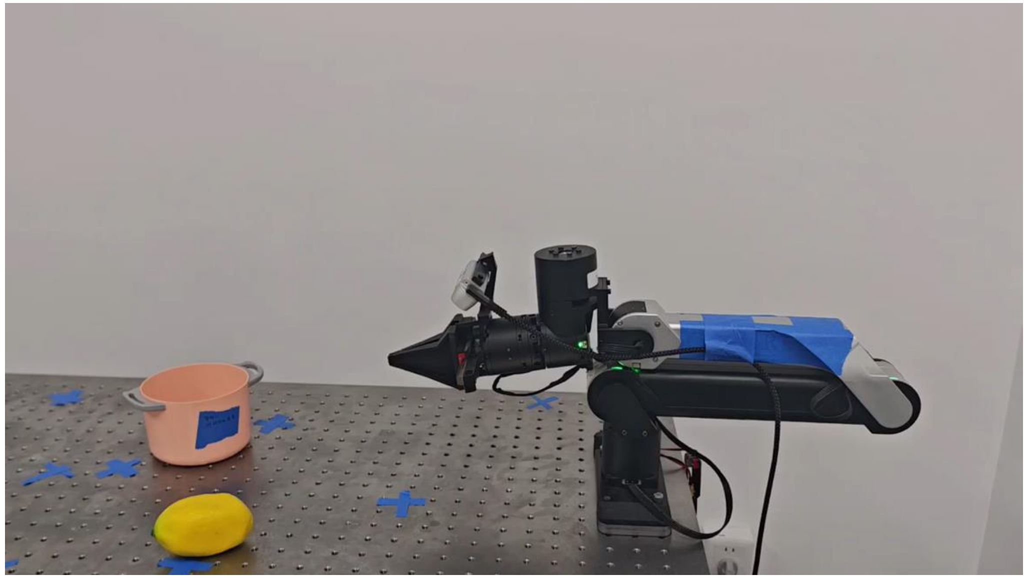
```
print("Execute Action!")
print(f"Action:{action}")
for item in action:
    t3 = time.perf_counter()
    right_control.joint_pos = item
    control_robot2.publish(right_control)
    rate.sleep()
    print(f"Execute: {(time.perf_counter()-t3)*1000:.1f}ms")


precise_wait(t_cycle_end - frame_latency)
iter_idx += steps_per_inference
```

For the policy

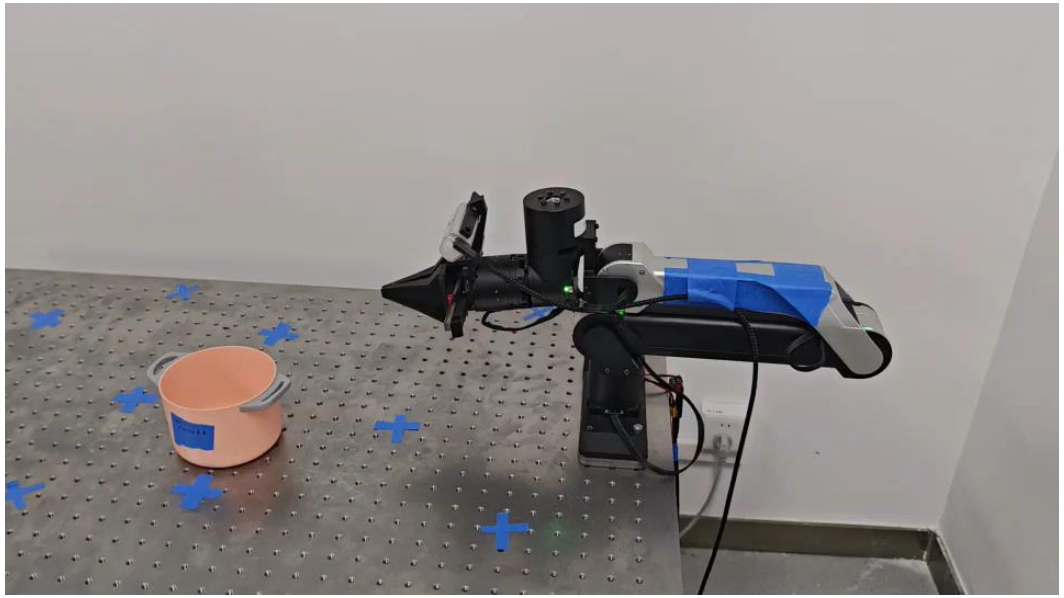
```
Inference: 279.0ms
```

Deployment

Put the Lemon in the Bowl



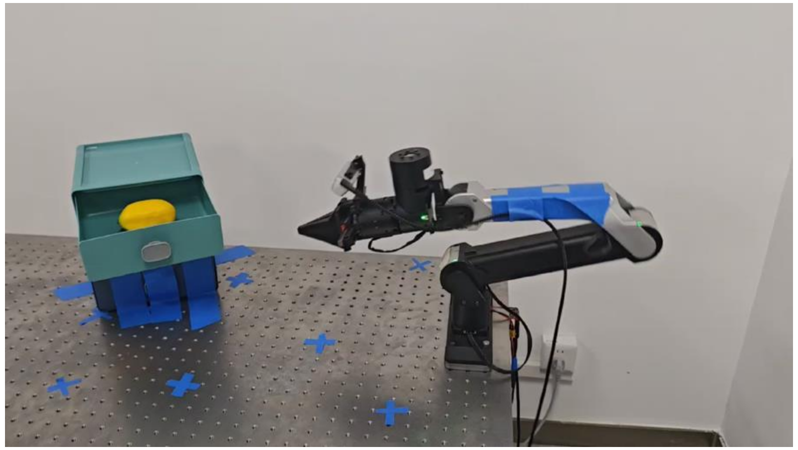
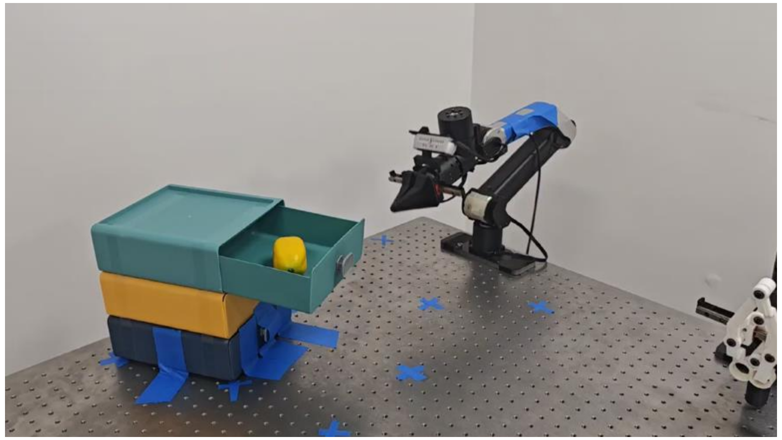
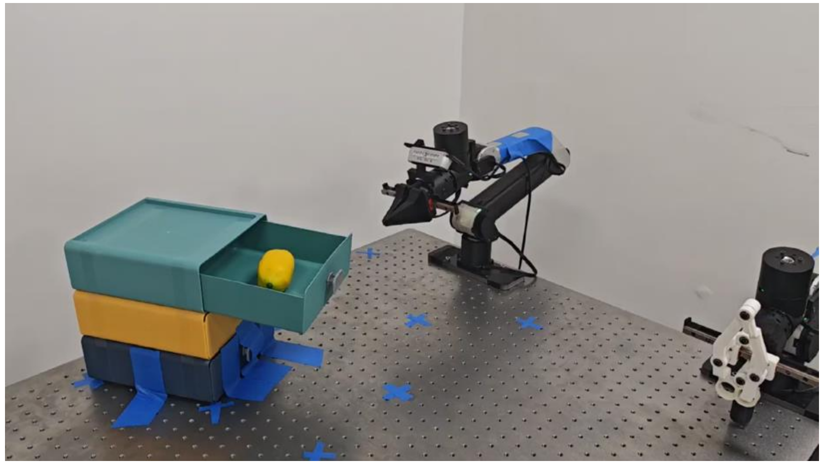
Lift the Bowl



Deployment

Push the Drawer

Steps Per Inference 8、10、12

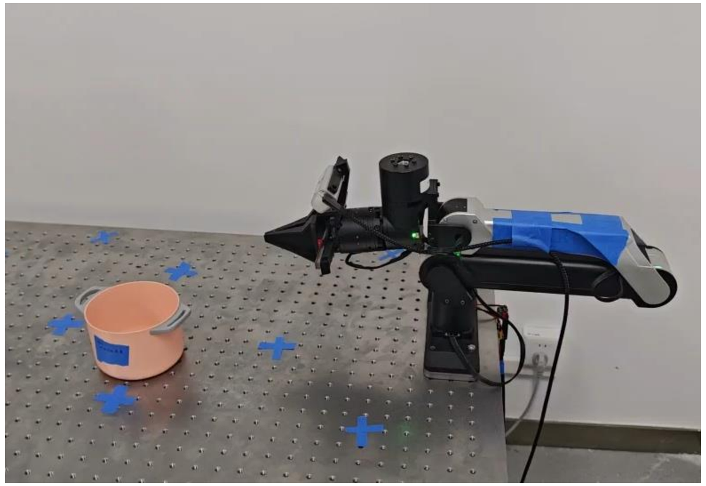
The movement process is gradually stabilized

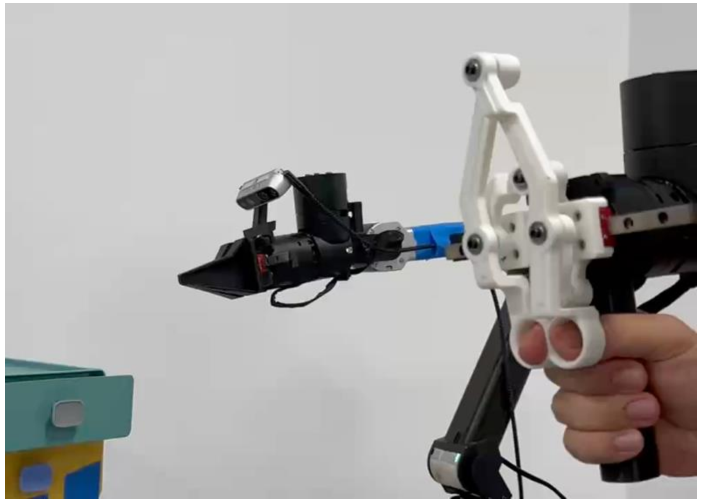
Explanation for the shakiness

Absolute control (via position)

1. Precise
2. No accumulative error

Hight demand for the control algorithm





Conclusion

1. Data-driven Generalizable Manipulation

The performance of DP3-based methods depends very strongly on the quality of the training data. The advantage lies in the generality of the framework: if you want the robotic arm to perform a new task, you only need to record the corresponding data and train it.

2. Engineering Efforts to Deploy DP3 to Reality

3. Questions that deserve further inquiry:

a. How to improve the generalization of learning algorithms? That is, how to make the policy as generalizable as possible with as few sample points as possible.

b. Why increasing the number of motion execution steps improves robot stability?

c. It is difficult to avoid jitter in training data, how to eliminate this jitter from the level of algorithms