物理引擎学习笔记

1. 概述

物理引擎是用于模拟物理现象的计算机程序，广泛应用于机器人仿真、游戏开发、动画制作等领域。本文将介绍几款主流的物理引擎，包括它们的安装配置和简单使用示例。

---

2. PyBullet

2.1 简介

PyBullet 是一个开源的物理引擎，支持快速原型设计、碰撞检测、动力学模拟等功能，适合机械臂、双足机器人等训练场景。它轻量级且易于使用，具有 Python 接口，非常适合研究和教育用途。

2.2 安装配置

```
# 使用pip安装
pip install pybullet
```

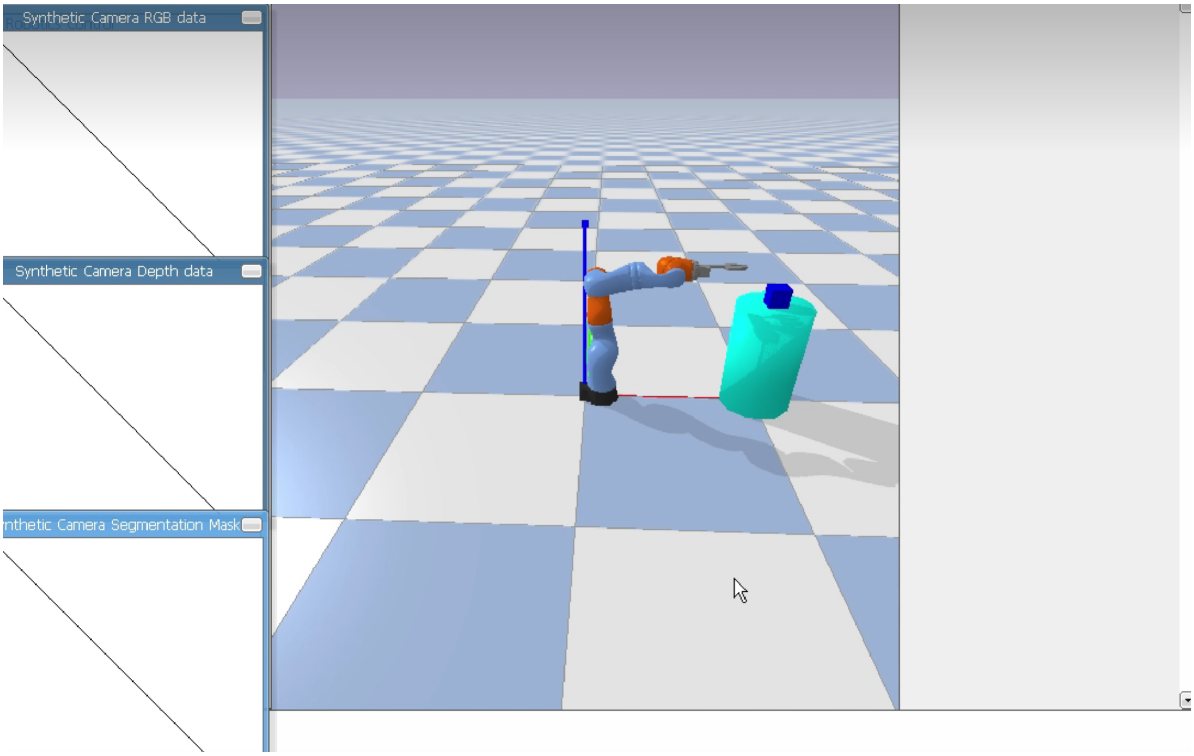
2.3 场景实现

实现了一个 KUKA 机械臂从起始点 [0,0,1.2] 平滑移动到目标点 [0.75,0,0.625] 的仿真过程。通过逆运动学计算，自动求解各关节应处的角度，使机械臂末端按照预定路径运动

```
import pybullet as p
import time
import pybullet_data
import numpy as np

# 连接到PyBullet物理引擎，使用带GUI的模式以便可视化
physicsClient = p.connect(p.GUI)
# p.DIRECT 则是无图形界面模式

# 设置PyBullet的数据路径，用于加载内置模型
p.setAdditionalSearchPath(pybullet_data.getDataPath())

# 设置重力加速度，沿Z轴负方向（向下）
p.setGravity(0, 0, -9.81)

# 加载地面模型
planeId = p.loadURDF("plane.urdf")

# 加载KUKA机械臂模型（带 gripper 夹爪）
robotId = p.loadSDF("/home/xzs/PyBullet_Practice/bullet3-
master/data/kuka_iiwa/kuka_with_gripper.sdf")

# 机械臂初始位置
robotStartPos = [0, 0, 0]
# 圆柱体初始位置（地面上方一点）
```

```
cylinderStartPos = [1, 0, 0.3]
# 立方体初始位置（圆柱体上方）
boxStartPos = [1, 0, 0.6 + 0.05 + 0.01]

# 初始姿态（都设为零姿态，即无旋转）
robotStartOrientation = p.getQuaternionFromEuler([0, 0, 0])
cylinderStartOrientation = p.getQuaternionFromEuler([0, 0, 0])
boxStartOrientation = p.getQuaternionFromEuler([0, 0, 0])

# 重置机械臂的位置和姿态
p.resetBasePositionAndOrientation(robotId[0], robotStartPos,
robotStartOrientation)

# 加载圆柱体和立方体模型
cylinderId = p.loadURDF("/home/xzs/PyBullet_Practice/bullet3-
master/data/cylinder1.urdf",
                        cylinderStartPos, cylinderStartOrientation)
boxId = p.loadURDF("/home/xzs/PyBullet_Practice/bullet3-master/data/cube1.urdf",
                   boxStartPos, boxStartOrientation)

# 获取机器人的关节总数
p.getNumJoints(robotId[0])

# 获取第7个关节的详细信息
p.getJointInfo(robotId[0], 7)

# 机械臂末端起始位置
robot7StartPos = [0, 0, 1.2]
# 机械臂末端目标位置
robotEndPos = [0.75, 0, 0.625]
# 机械臂末端目标姿态（用四元数表示）
robotEndOrientation = p.getQuaternionFromEuler([1.57, 0, 1.57])  # 转换自欧拉角

# 将位置转换为numpy数组便于计算
startPos_array = np.array(robot7StartPos)
endPos_array = np.array(robotEndPos)

# 运动步数（将路径分为5段）
stepNum = 5
# 计算每步的位移
step_array = (endPos_array - startPos_array) / stepNum

for j in range(stepNum):
    print(j, "step")
    # 计算当前步的目标位置
    robotStepPos = list(step_array + startPos_array)

    # 计算逆运动学：根据末端位置和姿态，计算各关节应处的角度
    targetPositionsJoints = p.calculateInverseKinematics(
        robotId[0],  # 机器人ID
        7,           # 末端关节索引
        robotStepPos,  # 目标位置
        targetOrientation=robotEndOrientation  # 目标姿态
    )

    # 设置所有关节（0-10共11个关节）的位置控制
    p.setJointMotorControlArray(
        robotId[0],
```

```
            range(11),
            p.POSITION_CONTROL,
            targetPositions=targetPositionsJoints
        )

        # 每步执行100次仿真迭代，保持一段时间以便观察
        for i in range(100):
            p.stepSimulation()  # 执行一次仿真步
            time.sleep(1./10.)  # 延迟0.1秒，控制可视化速度
            print("i:", i)

        print("------------------------------------------------------------------------
---------")
        # 更新起始位置为当前步的位置，准备下一步运动
        startPos_array = np.array(robotStepPos)

# 断开与物理引擎的连接
p.disconnect()
```



3. MuJoCo

3.1 简介

MuJoCo（Multi-Joint dynamics with Contact）是一个高级物理引擎，专为机器人学、 biomechanics 和动画设计。它提供了高精度的物理模拟，支持复杂的接触动力学。

3.2 安装配置

```
# 安装mujoco-py
pip install mujoco-py

# 注意：需要先获取MuJoCo的许可证和二进制文件
# 1. 从https://www.roboti.us/license.html获取免费许可证
# 2. 下载MuJoCo二进制文件https://www.roboti.us/index.html
# 3. 将文件解压到~/.mujoco/mujoco200目录
```

3.3 简单场景示例

首先创建一个 XML 模型文件（`simple_model.xml`）：

```
<mujoco model="simple">
    <option timestep="0.01" gravity="0 0 -9.81"/>

    <default>
        <joint armature="0.1" damping="1" limited="true"/>
        <geom conaffinity="0" condim="3" friction="1 0.1 0.1"
              density="500" margin="0.01" rgba="0.8 0.6 0.4 1"/>
    </default>

    <worldbody>
        <light pos="0 0 3" dir="0 0 -1"/>
        <geom name="floor" type="plane" size="5 5 0.1" rgba="0.9 0.9 0.9 1"/>

        <body name="box" pos="0 0 1">
            <freejoint/>
            <geom name="box_geom" type="box" size="0.2 0.2 0.2"/>
        </body>
    </worldbody>

    <actuator>
    </actuator>
</mujoco>
```

然后编写 Python 代码加载并运行仿真：实现了一个基于 Mujoco 物理引擎的 UR5e 机械臂仿真程序

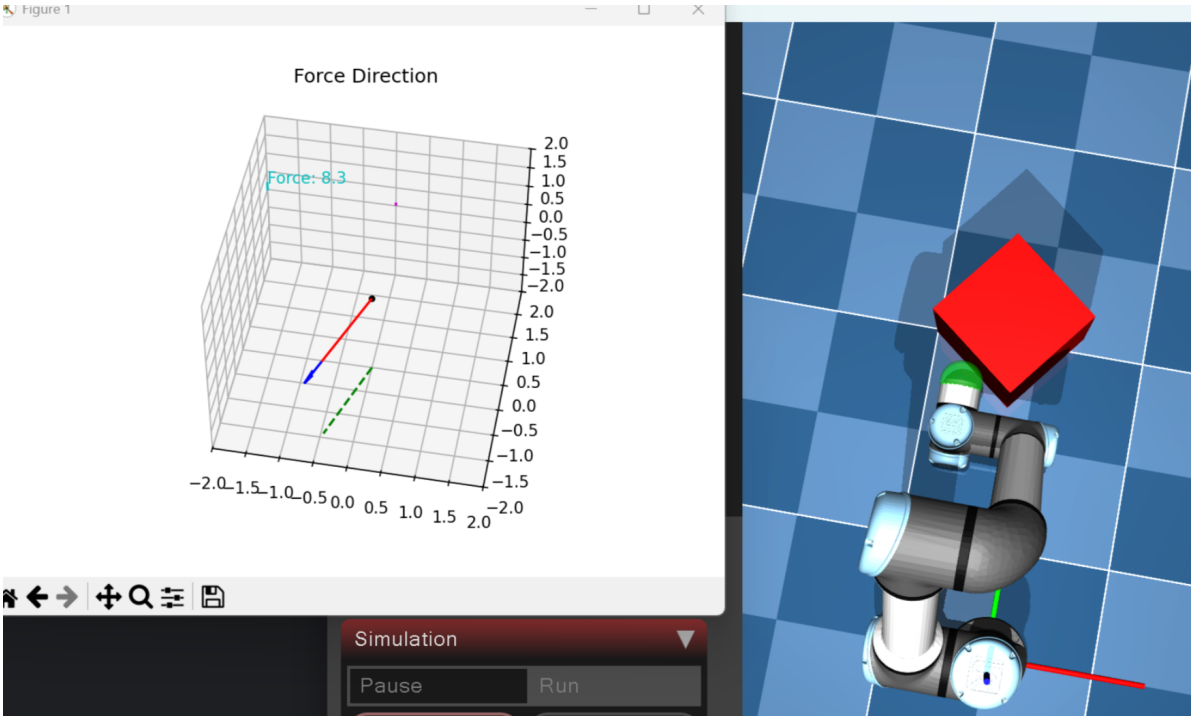
```
# 导入所需库：Mujoco可视化、IKPy机械臂链、坐标变换、数值计算、滑动窗口和绘图
import mujoco.viewer
import ikpy.chain
import transforms3d as tf
import numpy as np
from collections import deque
import matplotlib.pyplot as plt


def viewer_init(viewer):
    """渲染器的摄像头视角初始化"""
    # 设置摄像头类型为自由模式
    viewer.cam.type = mujoco.mjtCamera.mjCAMERA_FREE
    # 设置摄像头焦点位置
    viewer.cam.lookat[:] = [0, 0.5, 0.5]
    # 设置摄像头距离焦点的距离
```

```
    viewer.cam.distance = 2.5
    # 设置摄像头方位角（水平旋转角度）
    viewer.cam.azimuth = 180
    # 设置摄像头仰角（垂直角度）
    viewer.cam.elevation = -30


class ForcePlotter:
    """实时可视化接触力的类"""

    def __init__(self, update_interval=20):
        # 启用matplotlib的交互模式
        plt.ion()
        # 创建图形和3D子图
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111, projection='3d')
        # 设置更新间隔（每多少帧更新一次可视化）
        self.update_interval = update_interval
        # 帧计数器，用于控制更新频率
        self.frame_count = 0

    def plot_force_vector(self, force_vector):
        # 帧计数加1
        self.frame_count += 1
        # 检查是否达到更新间隔，未达到则跳过
        if self.frame_count % self.update_interval != 0:
            return

        # 清除当前子图
        self.ax.clear()

        # 力向量的起点（原点）
        origin = np.array([0, 0, 0])
        # 计算力的大小
        force_magnitude = np.linalg.norm(force_vector)
        # 计算力的方向（单位向量），避免除以零
        force_direction = force_vector / force_magnitude if force_magnitude >
1e-6 else np.zeros(3)

        # 绘制主箭头（红色）表示力的方向和大小
        arrow_tip = force_direction * 1.5
        self.ax.quiver(*origin, *arrow_tip, color='r', arrow_length_ratio=0)

        # 绘制次级箭头（蓝色）增强可视化效果
        self.ax.quiver(*arrow_tip, *(0.5 * force_direction), color='b',
arrow_length_ratio=0.5)

        # 绘制力向量在XY平面上的投影（绿色虚线）
        self.ax.plot([0, arrow_tip[0]], [0, arrow_tip[1]], [-2, -2], 'g--')

        # 绘制力向量在XZ平面上的投影（品红色虚线）
        self.ax.plot([0, 0], [2, 2], [0, arrow_tip[2]], 'm--')

        # 绘制力大小指示条（青色）并显示数值
        scaled_force = min(max(force_magnitude / 50, 0), 2)
        self.ax.plot([-2, -2], [2, 2], [0, scaled_force], 'c-')
        self.ax.text(-2, 2, scaled_force, f'Force: {force_magnitude:.1f}',
color='c')
```

```
        # 绘制原点标记并设置坐标轴范围
        self.ax.scatter(0, 0, 0, color='k', s=10)
        self.ax.set_xlim([-2, 2])
        self.ax.set_ylim([-2, 2])
        self.ax.set_zlim([-2, 2])
        self.ax.set_title(f'Force Direction')

        # 更新图形并短暂暂停以允许渲染
        plt.draw()
        plt.pause(0.001)
        # 重置帧计数器
        self.frame_count = 0


class ForceSensor:
    """处理力传感器数据的类，包含滑动平均滤波"""
    def __init__(self, model, data, window_size=100):
        # 存储Mujoco模型和数据引用
        self.model = model
        self.data = data
        # 滑动窗口大小（用于平均滤波）
        self.window_size = window_size
        # 创建双端队列存储力数据历史
        self.force_history = deque(maxlen=window_size)

    def filter(self):
        """获取并滑动平均滤波力传感器数据(传感器坐标系下)"""
        # 从Mujoco数据中获取原始力传感器数据，取前3个分量并反转方向
        force_local_raw = self.data.sensordata[:3].copy() * -1

        # 将新数据添加到滑动窗口
        self.force_history.append(force_local_raw)

        # 计算滑动窗口内的平均值作为滤波后的力
        filtered_force = np.mean(self.force_history, axis=0)

        return filtered_force


class JointSpaceTrajectory:
    """关节空间坐标系下的线性插值轨迹生成器"""

    def __init__(self, start_joints, end_joints, steps):
        # 存储起始关节角度
        self.start_joints = np.array(start_joints)
        # 存储目标关节角度
        self.end_joints = np.array(end_joints)
        # 轨迹总步数
        self.steps = steps
        # 计算每步的关节角度增量
        self.step = (self.end_joints - self.start_joints) / self.steps
        # 生成完整轨迹的生成器
        self.trajectory = self._generate_trajectory()
        # 当前目标路径点（初始为起始关节角度）
        self.waypoint = self.start_joints

    def _generate_trajectory(self):
```

```
        """生成从起点到终点的线性插值轨迹"""
        for i in range(self.steps + 1):
            # 计算第i步的关节角度
            yield self.start_joints + self.step * i
        # 确保最后精确到达目标关节值
        yield self.end_joints

    def get_next_waypoint(self, qpos):
        """根据当前关节位置获取下一个目标路径点"""
        # 检查当前关节位置是否接近当前目标路径点（容差0.02弧度）
        if np.allclose(qpos, self.waypoint, atol=0.02):
            try:
                # 获取下一个路径点
                self.waypoint = next(self.trajectory)
                return self.waypoint
            except StopIteration:
                # 若轨迹已完成则不做改变
                pass
        # 返回当前目标路径点
        return self.waypoint


def main():
    # 加载Mujoco模型（UR5e机械臂场景）
    model =
mujoco.MjModel.from_xml_path('model/universal_robots_ur5e/scene.xml')
    # 初始化Mujoco数据
    data = mujoco.MjData(model)
    # 从URDF文件创建IKPy机械臂链，指定活动关节
    my_chain = ikpy.chain.Chain.from_urdf_file("model/ur5e.urdf",
                                               active_links_mask=[False, False]
+ [True] * 6 + [False])

    # 机械臂初始关节角度（对应末端执行器位姿[-0.13, 0.3, 0.1, 3.14, 0, 1.57]）
    start_joints = np.array([-1.57, -1.34, 2.65, -1.3, 1.55, 0])
    # 设置初始关节角度，确保渲染初始状态正确
    data.qpos[:6] = start_joints

    # 设置末端执行器目标位置
    ee_pos = [-0.13, 0.6, 0.1]
    # 设置末端执行器目标姿态（欧拉角）
    ee_euler = [3.14, 0, 1.57]
    # 逆运动学计算的参考初始位置
    ref_pos = [0, 0, -1.57, -1.34, 2.65, -1.3, 1.55, 0, 0]
    # 将欧拉角转换为旋转矩阵
    ee_orientation = tf.euler.euler2mat(*ee_euler)

    # 计算达到目标位姿的关节角度（逆运动学）
    joint_angles = my_chain.inverse_kinematics(ee_pos, ee_orientation, "all",
initial_position=ref_pos)
    # 提取有效关节角度（去除首尾无关关节）
    end_joints = joint_angles[2:-1]

    # 创建关节空间轨迹生成器（100步完成运动）
    joint_trajectory = JointSpaceTrajectory(start_joints, end_joints, steps=100)

    # 初始化力传感器和力可视化器
    force_sensor = ForceSensor(model, data)
```

```
    force_plotter = ForcePlotter()

    # 启动Mujoco被动 viewer
    with mujoco.viewer.launch_passive(model, data) as viewer:
        # 初始化摄像头视角
        viewer_init(viewer)
        # 仿真循环（直到viewer关闭）
        while viewer.is_running():
            # 获取下一个目标关节角度
            waypoint = joint_trajectory.get_next_waypoint(data.qpos[:6])
            # 设置控制信号（关节位置控制）
            data.ctrl[:6] = waypoint

            # 获取滤波后的力数据并可视化
            filtered_force = force_sensor.filter()
            force_plotter.plot_force_vector(filtered_force)

            # 执行一步仿真
            mujoco.mj_step(model, data)
            # 同步viewer显示
            viewer.sync()


# 程序入口
if __name__ == "__main__":
    main()
```



4. Isaac Gym

4.1 简介

Isaac Gym 是 NVIDIA 开发的高性能物理仿真平台，专为强化学习设计。它利用 GPU 加速，可以同时运行数千个并行仿真环境，大幅提高训练效率。

4.2 安装配置

```
# 首先需要安装NVIDIA驱动和CUDA

# 克隆Isaac Gym仓库
git clone https://github.com/NVIDIA-Omniverse/IsaacGymEnvs.git
cd IsaacGymEnvs

# 创建虚拟环境
conda create -n isaacgym python=3.8
conda activate isaacgym

# 安装依赖
pip install -r requirements.txt

# 安装Isaac Gym（需要从NVIDIA官网下载安装包）
# 假设安装包在当前目录
pip install isaacgym-*.whl
```

4.3 简单示例

实现小球落地的效果

```
import isaacgym
from isaacgym import gymapi
from isaacgym import gymutil
import time

# 初始化Gym API
gym = gymapi.acquire_gym()

# 解析命令行参数
args = gymutil.parse_arguments(description="Isaac Gym简单示例")

# 配置仿真
sim_params = gymapi.SimParams()
sim_params.dt = 1.0 / 60.0  # 仿真步长
sim_params.substeps = 2
sim_params.gravity = gymapi.Vec3(0.0, 0.0, -9.81)  # 重力

# 使用GPU物理模拟
sim_params.physx.use_gpu = True
sim_params.physx.solver_type = 1
sim_params.physx.num_position_iterations = 4
sim_params.physx.num_velocity_iterations = 1

# 创建仿真
sim = gym.create_sim(args.compute_device_id, args.graphics_device_id,
args.physics_engine, sim_params)

# 创建地面平面
```

```
plane_params = gymapi.PlaneParams()
plane_params.normal = gymapi.Vec3(0, 0, 1)  # 朝上
plane_params.distance = 0  # 位置
plane_params.static_friction = 0.5
plane_params.dynamic_friction = 0.5
plane_params.restitution = 0.0
gym.add_ground(sim, plane_params)

# 创建环境
env_spacing = 2.0
env_lower = gymapi.Vec3(-env_spacing, -env_spacing, 0.0)
env_upper = gymapi.Vec3(env_spacing, env_spacing, env_spacing)

# 创建16个环境
num_envs = 16
envs = []
actor_handles = []

for i in range(num_envs):
    # 创建环境
    env = gym.create_env(sim, env_lower, env_upper, 4)
    envs.append(env)

    # 创建一个球体
    asset_options = gymapi.AssetOptions()
    asset_options.density = 1000.0
    asset_options.fix_base_link = False
    sphere_asset = gym.create_sphere(sim, 0.5, asset_options)

    # 初始位置
    initial_pose = gymapi.Transform()
    initial_pose.p = gymapi.Vec3(0.0, 0.0, 2.0)

    # 将球体添加到环境
    actor_handle = gym.create_actor(env, sphere_asset, initial_pose, "sphere",
i, 1)
    actor_handles.append(actor_handle)

    # 设置球体颜色
    gym.set_rigid_body_color(env, actor_handle, 0, gymapi.MESH_VISUAL,
gymapi.Vec3(0.5, 0.5, 1.0))

# 创建 viewer
viewer = gym.create_viewer(sim, gymapi.CameraProperties())
if viewer is None:
    print("无法创建viewer")
    quit()

# 仿真循环
while not gym.query_viewer_has_closed(viewer):
    # 步进仿真
    gym.simulate(sim)
    gym.fetch_results(sim, True)

    # 更新viewer
    gym.step_graphics(sim)
    gym.draw_viewer(viewer, sim, True)
```

```
    # 延迟以控制帧率
    time.sleep(0.01)

# 清理
gym.destroy_viewer(viewer)
gym.destroy_sim(sim)
```