项目申请书

项目名称：为RustPilot添加MuJoCo/Nvidia Isaac仿真平台的支持
项目主导师：ruiqurm
申请人：程茂霖
日期：2025.6.6
邮箱：2428791869@qq.com
Github：github.com/M3ngL

1. 项目背景

RustPilot 是一款采用 Rust 语言开发、运行于 Linux 系统的开源飞控软件。项目以简洁性与可维护性为核心，基于模块化设计与仿真驱动开发，为开发者、无人机爱好者及研究人员提供灵活可靠的飞控解决方案，同时降低小型无人机的开发门槛。 目前，RustPilot 已支持在 Gazebo 环境中进行仿真调试（SITL）。Gazebo 在飞控控制功能的验证方面具备一定优势，适用于通用仿真场景。然而，由于其物理引擎建模能力相对有限，在接触建模、高自由度系统模拟等方面不及 MuJoCo 与 NVIDIA Isaac。此外，Gazebo 的视觉效果较为粗糙，难以满足合成数据生成和视觉模型训练等现代需求。

因此，为更好地对接强化学习生态，RustPilot 需扩展支持 MuJoCo 和 NVIDIA Isaac 等更先进的仿真平台。

- MuJoCo 平台

MuJoCo 是由 Emo Todorov 开发、目前由 DeepMind 维护的高精度物理仿真引擎，并已于 2021 年开源。其主要特点包括：高效的求解器、多接触动力学建模、低延迟模拟能力，广泛应用于强化学习、机器人控制和优化等领域。MuJoCo 适用于复杂多体系统的高精度仿真，便于嵌入研究环境，并支持大规模训练任务。

- NVIDIA Isaac 平台

NVIDIA Isaac 平台 是 NVIDIA 推出的机器人开发平台，采用 Omniverse + PhysX 引擎，具有出色的图形渲染能力与 GPU 加速仿真性能。该平台支持高保真的传感器模拟（如摄像头、激光雷达等），可与 ROS / ROS 2 系统无缝集成，同样适合用于大规模强化学习训练与系统验证。

项目成果链接： https://github.com/Ncerzzk/RustPilot

2. 技术方案

本项目拟使用rust编写 RustPilot 与 MuJoCo \ NVIDIA Isaac 平台的交互接口模块，实现仿真器与仿真飞控系统之间的数据交互，包括从仿真器获取传感器数据和飞行器状态的模块以及RustPilot 电机控制指令发送到仿真器的模块。

分为三个步骤实现从仿真器数据到飞控内部的逻辑处理

1. 在 MuJoCo XML \ NVIDIA Isaac URDF模型文件中定义仿真无人机的传感器以及执行器

2. 编写 RustPilot 的仿真接口模块，使其能读取传感器数据

3. 将仿真接口获得的数据适配到 RustPilot 内部逻辑，并将输出数据写入控制执行器

2.1 定义仿真器中无人机的基本部件

代码片段

以 MuJoCo 的xml文件为例

```
<mujoco model="quadrotor">
  <compiler angle="radian" inertiafromgeom="true"/> <option timestep="0.001">
    <gravity refractory="2" mass="1" x="0" y="0" z="-9.81"/> </option>

  <asset>
        ...
  </asset>

  <worldbody>
    <light .../>
    <geom .../>

    <body name="base_link" pos="0 0 0.053302"> <inertial pos="0 0 0"
mass="1.5" diaginertia="0.0347563 0.07 0.0977"/> <geom name="base_geom"
type="box" size="0.235 0.235 0.055" rgba="0 0 1 0.7"/> <site name="imu_site"
pos="0 0 0"/>
    <sensor>
        <accelerometer name="imu_accel" site="imu_site"/>
        <gyro name="imu_gyro" site="imu_site"/>
        <magnetometer name="imu_mag" site="imu_site"/> <framepos
name="gps_pos" objtype="body" objname="base_link"/>
        <framequat name="gps_quat" objtype="body" objname="base_link"/>
        <framelinvel name="gps_linvel" objtype="body" objname="base_link"/>
        <frameangvel name="gps_angvel" objtype="body" objname="base_link"/>
      </sensor>

      <body name="rotor_0_body" pos="0.13 -0.22 0.023">
        <inertial pos="0 0 0" mass="0.005" diaginertia="9.75e-07 4.17041e-05
4.26041e-05"/>
        <geom name="rotor_0_geom" type="cylinder" size="0.1 0.0025"
material="blue_mat" rgba="0 0 1 0.7"/> </body>
      <body name="rotor_1_body" pos="-0.13 0.2 0.023">
        <inertial pos="0 0 0" mass="0.005" diaginertia="9.75e-07 4.17041e-05
4.26041e-05"/>
        <geom name="rotor_1_geom" type="cylinder" size="0.1 0.0025"
material="red_mat" rgba="1 0 0 0.7"/>
        </body>
      <body name="rotor_2_body" pos="0.13 0.22 0.023">
        <inertial pos="0 0 0" mass="0.005" diaginertia="9.75e-07 4.17041e-05
4.26041e-05"/>
        <geom name="rotor_2_geom" type="cylinder" size="0.1 0.0025"
material="blue_mat" rgba="0 0 1 0.7"/>
        </body>
      <body name="rotor_3_body" pos="-0.13 -0.2 0.023">
```

```
        <inertial pos="0 0 0" mass="0.005" diaginertia="9.75e-07 4.17041e-05 4.26041e-05"/>
        <geom name="rotor_3_geom" type="cylinder" size="0.1 0.0025" material="red_mat" rgba="1 0 0 0.7"/>
        </body>
    </body>
  </worldbody>

  <actuator>
    <joint name="rotor_0_joint" type="revolute" pos="0.13 -0.22 0.023" axis="0 0 1" parent="base_link" child="rotor_0_body"/>
    <joint name="rotor_1_joint" type="revolute" pos="-0.13 0.2 0.023" axis="0 0 1" parent="base_link" child="rotor_1_body"/>
    <joint name="rotor_2_joint" type="revolute" pos="0.13 0.22 0.023" axis="0 0 1" parent="base_link" child="rotor_2_body"/>
    <joint name="rotor_3_joint" type="revolute" pos="-0.13 -0.2 0.023" axis="0 0 1" parent="base_link" child="rotor_3_body"/>

    <motor name="motor_0" joint="rotor_0_joint" gear="100"/>
    <motor name="motor_1" joint="rotor_1_joint" gear="100"/>
    <motor name="motor_2" joint="rotor_2_joint" gear="100"/>
    <motor name="motor_3" joint="rotor_3_joint" gear="100"/>
  </actuator>

</mujoco>
```

可行性分析

- MuJoCo 的核心就是通过 XML 文件定义物理模型，包括 `body`、`joint`、`actuator` 和 `sensor`。其 XML 语法清晰，文档完备。各种类型的传感器（加速度计、陀螺仪、磁力计、位置、姿态、力传感器等）和执行器（电机、舵机）都有对应的标签和属性可供配置。
- Isaac Sim 支持多种机器人描述格式，其中 URDF (Unified Robot Description Format) 是 ROS 生态系统中最常用的机器人模型描述语言。URDF 可以定义机器人的连杆、关节、惯性特性以及传感器和执行器（通过 ROS 控制插件）。Isaac Sim 能够导入和解释 URDF 模型。

2.2 飞控与仿真器的交互接口实现

代码片段

仍然以MuJoCo仿真平台为例

```
struct ImuData {
    accelerometer_x: f32, // 加速度计 X 轴 (m/s^2)
    accelerometer_y: f32, // 加速度计 Y 轴 (m/s^2)
    accelerometer_z: f32, // 加速度计 Z 轴 (m/s^2)
    gyro_x: f32,          // 陀螺仪 X 轴 (rad/s)
    gyro_y: f32,          // 陀螺仪 Y 轴 (rad/s)
    gyro_z: f32,          // 陀螺仪 Z 轴 (rad/s)
}
... // 其他传感器部件以及执行器部件数据结构定义

/// 仿真器接口模块
module sim_interface {
```

```
    extern_ffi_bindings_to_mujoco;

    /// 初始化仿真器
    /// 返回：成功或失败
    fn initialize_simulator(model_path: &str) -> Result<(), SimError> {
        // 调用 MuJoCo 的 mj_activate 或 Isaac 的初始化函数
        // 加载模型文件 (e.g., quadrotor.xml)
        // 创建仿真数据结构 (mjData)
        // 返回成功或错误
    }

    /// 执行仿真器的一个时间步
    fn step_simulation() {
        // 调用 MuJoCo 的 mj_step(model, data)
    }

    /// 从仿真器获取原始 IMU 传感器数据
    fn get_raw_imu_data() -> (f32, f32, f32, f32, f32, f32) {
        return ...;
    }

    /// 从仿真器获取原始磁力计数据
    fn get_raw_magnetometer_data() -> (f32, f32, f32) {
        // return (sim_mag_x, sim_mag_y, sim_mag_z);
    }

    /// 从仿真器获取原始气压计数据
    fn get_raw_barometer_data() -> (f32, f32) {
        // return (sim_pressure_pa, sim_temperature_c);
    }

    /// 从仿真器获取原始激光测距仪数据
    fn get_raw_rangefinder_data() -> f32 {
        // return sim_distance;
    }

    /// 从仿真器获取飞行器状态信息
    fn get_flight_state() -> (f32, f32, f32,   // pos
                              f32, f32, f32, f32, // quat
                              f32, f32, f32,   // vel
                              f32, f32, f32) { // angular_vel
        // return (sim_pos_x, ..., sim_quat_w, ..., sim_vel_x, ...,
sim_angular_vel_x, ...);
    }

    /// 向仿真器发送电机控制指令
    fn set_motor_commands(motor_commands: &[f32]) {
        // for (i, &cmd) in motor_commands.iter().enumerate() {
        //     mujoco_data.ctrl[i] = cmd_to_mujoco_force(cmd);
        // }
    }
}
```

可行性分析

- 性能：Rust 具有接近 C/C++ 的性能，对于实时性要求高的飞控和仿真接口来说是理想选择。

- 内存安全： Rust 的所有权和借用机制能够提供强大的内存安全保证，这在处理复杂的系统（如飞控和仿真数据）时非常重要，可以避免常见的段错误和内存泄漏，提高系统稳定性。
- FFI 能力：Rust 对 FFI 的良好支持使得与 C 库（如 MuJoCo）的集成相对直接。

2.3 数据适配到飞控内部处理逻辑

代码片段

```
/// RustPilot 飞控核心模块
module rustpilot_core {
    use sim_interface; // 引入仿真器接口模块

    // 飞控内部的传感器处理模块
    struct ImuProcessor;
    impl ImuProcessor {
        fn process_raw_data(raw_accel: (f32, f32, f32), raw_gyro: (f32, f32, f32)) -> ImuData {
            // 对原始数据进行单位转换、校准、滤波等处理并返回标准化的 ImuData
            return ImuData { ... };
        }
    }

    struct MagnetometerProcessor;
    impl MagnetometerProcessor {
        fn process_raw_data(raw_mag: (f32, f32, f32)) -> MagnetometerData {
            // 校准、滤波
            return MagnetometerData { ... };
        }
    }

    struct BarometerProcessor;
    impl BarometerProcessor {
        fn process_raw_data(raw_pressure: f32, raw_temp: f32) -> BarometerData {
            // 滤波并转换为高度
            return BarometerData { ... };
        }
    }

    struct RangefinderProcessor;
    impl RangefinderProcessor {
        fn process_raw_data(raw_distance: f32) -> RangefinderData {
            // 滤波
            return RangefinderData { ... };
        }
    }

    // 状态估计器模块
    struct StateEstimator;
    impl StateEstimator {
        fn update_state(imu_data: &ImuData, mag_data: &MagnetometerData, baro_data: &BarometerData,
```

```
                        rangefinder_data: &RangefinderData, flight_state:
&FlightState) -> FlightState {
            return FlightState { ... }; // 返回更新后的估计状态
        }
    }

    // 飞控控制器模块
    struct FlightController;
    impl FlightController {
        fn compute_motor_commands(estimated_state: &FlightState,
target_attitude: (f32, f32, f32)) -> Vec<f32> {
            // 根据目标姿态和估计姿态计算滚转、俯仰、偏航、油门指令
            // 混控器将指令转换为 4 个电机推力/PWM 值
            return vec![motor1_cmd, motor2_cmd, motor3_cmd, motor4_cmd];
        }
    }

    /// RustPilot 主仿真循环
    fn run_simulation_loop() {
        if
sim_interface.initialize_simulator("path/to/quadrotor.xml").is_err() {
            println!("仿真器初始化失败! ");
            return;
        }

        let mut current_flight_state = FlightState { ... };
        let mut target_attitude = (0.0, 0.0, 0.0); // 目标悬停姿态

        loop {
            // 1. 推进仿真器一步
            sim_interface.step_simulation();

            // 2. 从仿真器获取原始传感器数据
            let (raw_accel_x, raw_accel_y, raw_accel_z, raw_gyro_x,
raw_gyro_y, raw_gyro_z) = sim_interface.get_raw_imu_data();
            let (raw_mag_x, raw_mag_y, raw_mag_z) =
sim_interface.get_raw_magnetometer_data();
            let (raw_pressure, raw_temp) =
sim_interface.get_raw_barometer_data();
            let raw_distance = sim_interface.get_raw_rangefinder_data();

            // 3. 从仿真器获取飞行器真实状态
            let (sim_pos_x, sim_pos_y, sim_pos_z,
                 sim_quat_w, sim_quat_x, sim_quat_y, sim_quat_z,
                 sim_vel_x, sim_vel_y, sim_vel_z,
                 sim_angular_vel_x, sim_angular_vel_y, sim_angular_vel_z) =
sim_interface.get_flight_state();

            // 4. 处理原始传感器数据
            let imu_data = ImuProcessor::process_raw_data(
                (raw_accel_x, raw_accel_y, raw_accel_z),
                (raw_gyro_x, raw_gyro_y, raw_gyro_z)
            );
            let mag_data =
MagnetometerProcessor::process_raw_data((raw_mag_x, raw_mag_y, raw_mag_z));
```

```
            let baro_data = BarometerProcessor::process_raw_data(raw_pressure, raw_temp);
            let rangefinder_data = RangefinderProcessor::process_raw_data(raw_distance);

            // 5. 状态估计
            current_flight_state = StateEstimator::update_state(
                &imu_data, &mag_data, &baro_data, &rangefinder_data,
                &FlightState { ... }
            );

            // 6. 飞控控制计算
            let motor_commands = FlightController::compute_motor_commands(&current_flight_state, target_attitude);

            // 7. 将控制指令发送回仿真器
            sim_interface.set_motor_commands(&motor_commands);

            // 检查退出条件
            if simulation_end_condition_met() {
                break;
            }
        }

        sim_interface.shutdown_simulator();
    }
}
```

可行性分析

- 当前RustPilot项目已有传感器处理逻辑，只需要将处理好的数据传入内部逻辑即可

3. 时间规划

第一阶段：需求分析与设计（2025.7.1 - 2025.7.8）

- 确定详细的仿真器接口需求，包括需要获取的传感器数据和飞行器状态，以及控制接口的定义。
- 设计 RustPilot 与仿真器的交互模块结构（`sim_interface`），以及核心功能模块（数据获取、电机控制、状态估计）。

第二阶段：基础功能开发（2025.7.9 - 2025.7.24）

- 确定交互方式（Rust FFI 或 Python）
- 开发 Rust FFI 或 Python 绑定，实现与 MuJoCo 或 Isaac Sim 的基本交互。

第三阶段：核心功能开发（2025.7.25 - 2025.8.22）

- 开发从仿真器获取传感器数据和飞行器状态的模块。
- 开发将 RustPilot 电机控制指令发送到仿真器的模块。

第四阶段：测试与优化 （2025.8.23 - 2025.9.7）

- 设计并实现基础仿真测试场景（如悬停）。

- 进行功能测试，验证数据获取、电机控制的正确性。
- 修复发现的 Bug，完善代码文档。

第五阶段：测试与优化 （2025.9.8 – 2025.9.30）

- 编写详细的项目技术文档
- 进一步维护发现的项目 bug