1.实验名称及目的

1.1实验名称

UE蓝图综合模型（四旋翼）（仅限完整版及以上版本）

1.2实验目的

了解基于UE物理引擎（动力学）+蓝图控制器的综合模型使用方法，该种模型能完全集成在RflySim3D中，从而更好地支持地形和碰撞响应

1.3关键知识点

例程功能API

UE控制接口：python库文件 UE4CtrlAPI.py： RflySimSDK/html/UE4CtrlAPI_8py.html

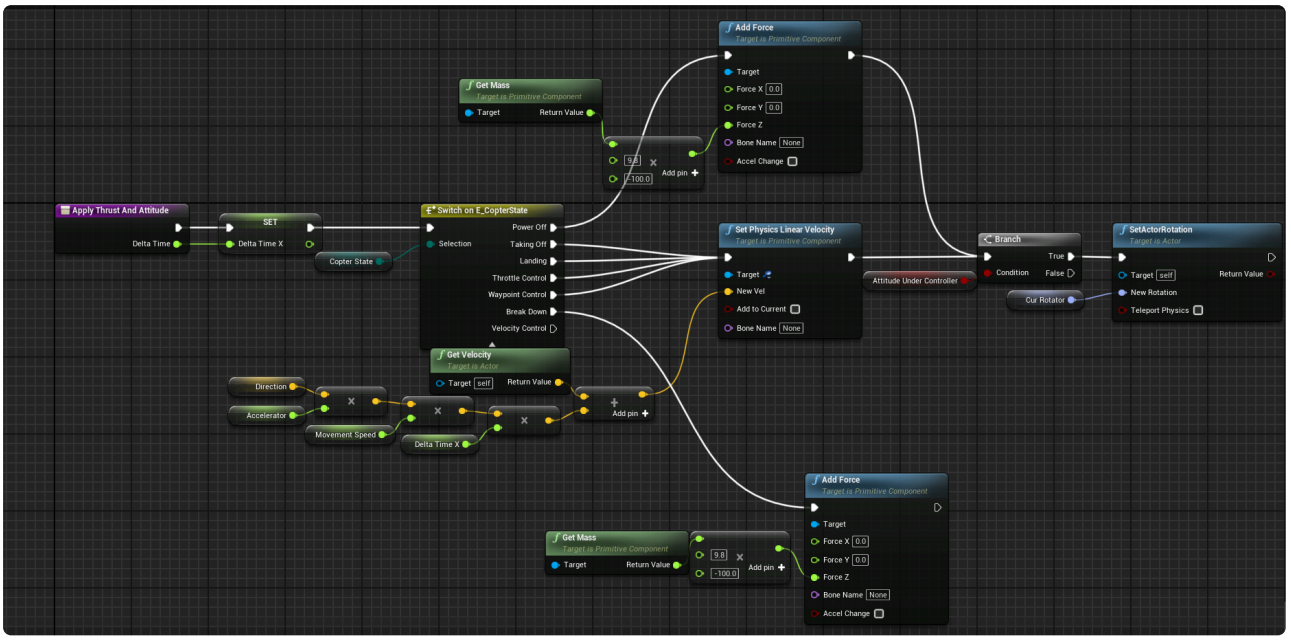
UE三维模型加载接口sendUE4Pos：
classRflySimSDK_1_1ue_1_1UE4CtrlAPI_1_1UE4CtrlAPI.html#sendUE4Cmd

载具模型控制接口：python库文件 DllSimCtrlAPI.py：
RflySimSDK/html/PX4MavCtrlV4_8py.html

外部控制命令发送接口sendSILIntFloat：

关键知识点1：蓝图综合模型的实现原理

蓝图动力学



整体实现依赖于UE物理引擎（PhysX / Chaos），主要通过 Add Force 和 Set Physics Linear Velocity 实时计算无人机速度；通过 SetActorRotation计算无人机姿态。

1. 时间增量输入

- **节点**：Delta Time
  - **原理**：
    - 时间增量（Delta Time）是从上一帧到当前帧之间的时间间隔，由引擎提供，用于确保逻辑的帧率无关性。
    - 通过乘以时间增量，可以将速度和力的计算与帧速率分离，使逻辑更加平滑且具有实时性。
  - **作用**：
    - 时间增量被传递到多个计算节点，用于推力、速度以及其他物理量的更新。

2. 状态切换控制

- **节点**：Switch on E_CopterState
  - **原理**：
    - 状态机（State Machine）用于根据无人机的当前状态选择不同的逻辑路径。
    - 每种状态代表无人机所处的特定操作模式（如起飞、着陆、巡航等）。
    - 这种分支逻辑可以提高代码的组织性，使不同状态下的操作独立。
  - **作用**：

        - **Power Off**：无人机处于关闭状态，不会施加力或更新姿态。
        - **Taking Off**：无人机进入起飞模式，施加额外推力克服重力并加速上升。
        - **Landing**：无人机进入着陆模式，降低推力并逐渐减速接触地面。
        - **Throttle Control**：调整推力输出，控制无人机的悬停或高度变化。
        - **Waypoint Control**：根据预设的航点，调整方向、速度和姿态。
        - **Break Down**：进入停止状态，移除所有推力。
        - **Velocity Control**：直接基于目标速度计算推力和加速度。

3. 获取物体质量

- **节点**：Get Mass
    - **原理**：
        - 每个物理对象都有质量（Mass），质量决定了对象在受力下的加速度大小（根据牛顿第二定律
          F=ma）。
        - 引擎提供的质量信息可以用来计算所需的推力和调整动力学行为。
    - **作用**：
        - 获取无人机的质量值，用于计算重力补偿推力和额外加速所需的力。
        - 传递到 Add Force 节点中。

4. 推力施加

- **节点**：Add Force
    - **原理**：
        - 力（Force）是影响物体运动状态的直接因素，按照牛顿第二定律，力决定物体的加速度。
        - 在引擎中，通过 Add Force 节点可以向物体施加一个指定方向和大小的力。
        - 力可以由重力补偿力、加速度力等多种因素叠加。
    - **作用**：
        - **Force Z**：计算重力补偿力： Fz=Mass×9.8
            - 确保无人机在静止状态下不会因重力下坠。
        - **Accel Change**：启用加速度变化模式（设置为
          True），使施加的力直接影响无人机的加速度。

5. 获取当前速度

- **节点**：Get Velocity
    - **原理**：

        - 速度（Velocity）是物体当前运动状态的重要信息，包含方向和速度大小。
        - 通过获取速度，计算当前速度与目标速度之间的差异，用于确定调整加速度的大小和方向。
    - **作用**：
        - 获取无人机的当前速度，用于计算需要施加的加速度。

6. 基于目标方向和加速度的速度更新

- **节点**：计算新的速度
    - **原理**：
        - 使用目标方向、加速度大小和时间增量，更新无人机的线性速度。
        - 公式描述： New Velocity=Current Velocity+(Direction×Accelerator)×Delta Time
            - **Current Velocity**：当前速度。
            - **Direction**：目标方向，通常为单位向量。
            - **Accelerator**：加速度大小，通常由目标速度和当前速度的差异计算。
            - **Delta Time**：时间增量，确保速度更新与帧率无关。
    - **作用**：
        - 动态调整速度以接近目标速度，确保无人机朝正确方向运动。

7. 应用线性速度

- **节点**：Set Physics Linear Velocity
    - **原理**：
        - 在物理系统中直接设置物体的线性速度，使其符合新的运动状态。
        - 如果 Add to Current 设置为 False，则完全覆盖当前速度。
    - **作用**：
        - 将计算出的新速度应用到无人机上，调整其移动行为。

8. 姿态调整

- **节点**：Set Actor Rotation
    - **原理**：
        - 物体的旋转状态（姿态）用欧拉角（Pitch、Yaw、Roll）或四元数表示，通过旋转器可以直接设置物体的姿态。
        - 在动态控制中，姿态调整确保无人机始终朝向目标方向。
    - **作用**：
        - 根据计算出的旋转角度（Cur Rotator），更新无人机的姿态。

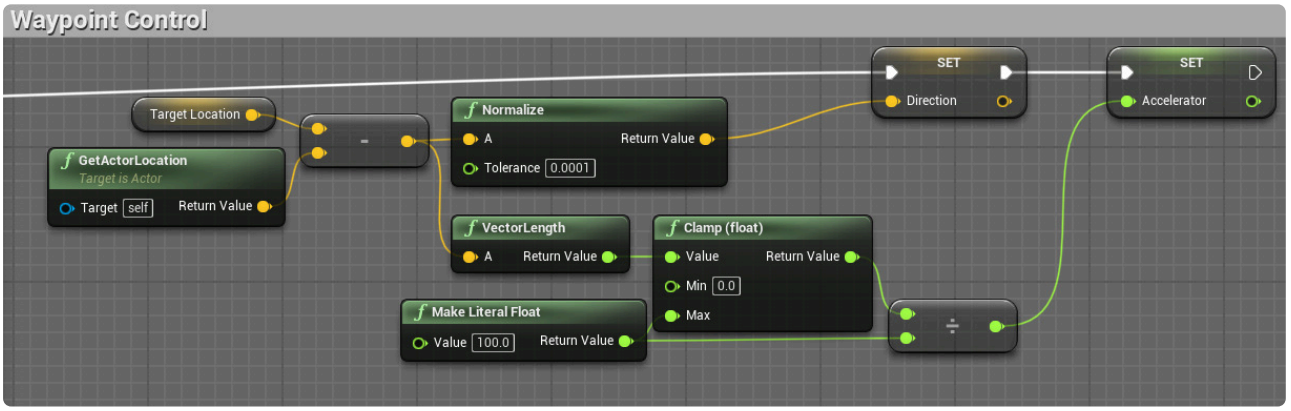
- 确保无人机的俯仰、偏航和滚转符合其运动需求。

9. 条件分支控制

- **节点**：Branch
  - **原理**：
    - 条件分支根据布尔值的结果选择执行路径，用于逻辑上的动态控制。
  - **作用**：
    - 判断是否启用姿态控制：
      - **True**：执行 Set Actor Rotation，调整无人机姿态。
      - **False**：跳过姿态调整，仅更新速度。

位置控制逻辑

更新朝向和加速度



1. 计算目标方向

- **节点：GetActorLocation**
  - **作用**：获取无人机当前的位置。
  - **操作**：从目标位置中减去当前无人机的位置，计算出一个方向向量，这个向量表示从无人机当前位置指向目标位置的方向。
  - **结果**：生成一个差向量，包含了目标位置与当前位置之间的方向信息。

2. 标准化方向向量

- **节点：Normalize**
  - **作用**：对计算出的方向向量进行标准化处理，使其只表示方向，而不包含距离信息。
  - **操作**：将方向向量的长度调整为固定值（通常为1），从而仅保留方向信息。这样可以确保后续逻辑中，所有的方向计算都基于标准化的方向。

- **结果**：输出一个单位向量，表示无人机需要朝向目标的标准化方向。

3. 计算无人机与目标之间的距离

- **节点：VectorLength**
  - **作用**：计算当前无人机与目标位置之间的直线距离。
  - **操作**：通过计算前一步生成的差向量的大小，得出距离值。这一步的结果表示无人机距离目标位置的远近。
  - **结果**：生成一个距离值，用于确定无人机需要多大的加速度去接近目标。

4. 限制加速度范围

- **节点：Clamp (Float)**
  - **作用**：限制计算出的加速度值在一个合理的范围内。
  - **操作**：将无人机与目标位置的距离作为输入，如果距离太大，则加速度被限制在预设的最大值；如果距离太小，则加速度可能会降低到一个更小的值，甚至为零。
  - **结果**：生成一个受限的加速度值，防止无人机过度加速或减速。

5. 设置无人机朝向

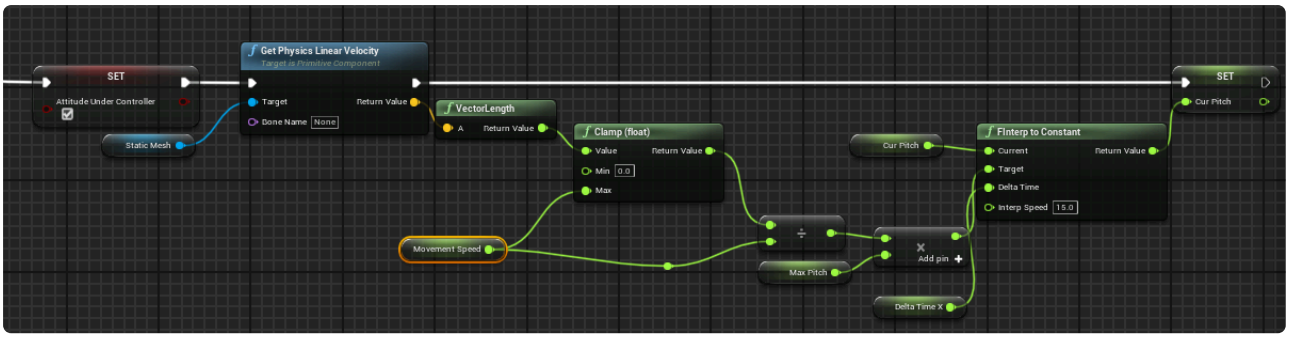
- **节点：Set Direction**
  - **作用**：将标准化后的方向向量存储到无人机的朝向变量中。
  - **操作**：将表示目标方向的单位向量赋值给一个变量，用于后续的运动逻辑。
  - **结果**：确保无人机知道它应该朝哪个方向前进。

6. 设置无人机加速度

- **节点：Set Accelerator**
  - **作用**：将受限制的加速度值存储到无人机的加速度变量中。
  - **操作**：根据计算出的距离和范围限制的结果，将加速度大小赋值到一个变量中，用于后续速度计算。
  - **结果**：确定无人机当前的加速程度，从而控制它的移动速度。

更新飞机姿态（模拟姿态环）

更新俯仰角



基于当前物体的速度，计算并逐步调整俯仰角，使物体的俯仰角随速度变化动态调整，同时通过插值确保角度变化平滑。这在需要通过物理模拟控制物体姿态（如无人机或飞行器）时非常重要。

整体逻辑描述

这段蓝图逻辑的目标是通过动态调整物体的俯仰角（Pitch），使其与当前运动状态相匹配。

整体作用

- 通过计算物体的线性速度，并将其映射到俯仰角，动态调整物体的姿态。
- 使用插值机制保证姿态的调整平滑，避免突然变化导致的不自然运动。

以下是逐步的逻辑描述：

1. 获取物体当前线性速度

- 使用 Get Physics Linear Velocity 节点获取物体的线性速度向量（包含 X、Y 和 Z 方向的速度分量）。
- 这个速度向量表示物体在当前帧中的实际移动速度。

2. 计算速度的大小

- 使用 Vector Length 节点计算线性速度向量的大小，即速度的标量值。
- 速度的大小是通过计算速度向量的欧几里得长度（即平方和的平方根）得出的。
- 输出的结果表示物体当前的实际运动速度。

3. 限制速度范围

- 使用 Clamp (Float) 节点对速度的大小进行限制：
  - 如果速度小于 0，则结果被限制为 0。
  - 如果速度大于预定义的最大速度（Movement Speed），则结果被限制为 Movement

Speed。

- 这一过程确保速度不会超出物理设定的范围，同时避免意外值影响后续计算。

4. 映射速度到目标俯仰角

- 将限制后的速度大小乘以一个比例因子（节点中称为 Max Pitch），以计算目标俯仰角。
- 这一比例因子决定了速度与俯仰角的映射关系。例如：
  - 较高的速度对应较大的俯仰角。
  - 较低的速度对应较小的俯仰角。

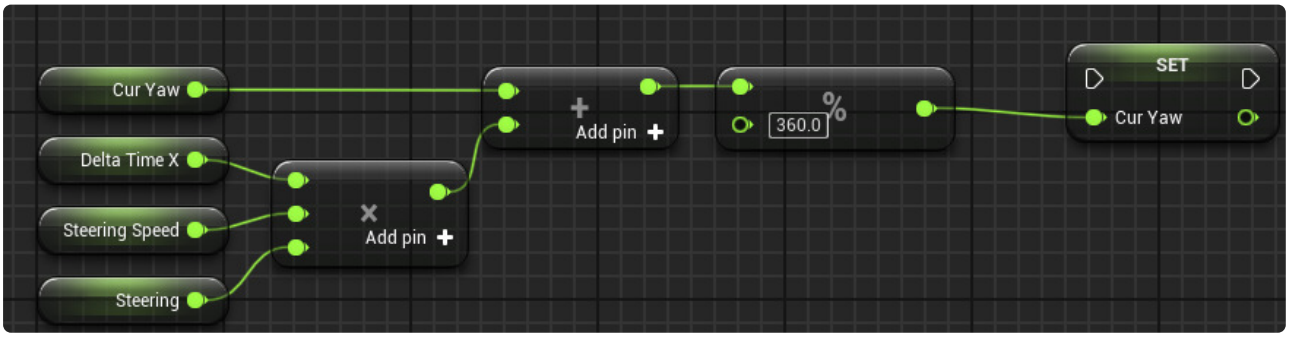
5. 平滑插值俯仰角

- 使用 FInterp To Constant 节点，通过线性插值的方式平滑地将当前俯仰角（Cur Pitch）调整到目标俯仰角：
  - **输入：**
    - 当前俯仰角（Cur Pitch）。
    - 目标俯仰角（由速度映射计算得出）。
    - 每帧的时间增量（Delta Time，自动提供）。
    - 插值速度（Interp Speed，设定为 15.0）。
  - **作用：**
    - 防止俯仰角瞬间跳变，确保调整过程平滑自然。

6. 更新当前俯仰角

- 使用 Set 节点将插值后的俯仰角赋值给 Cur Pitch 变量，作为当前帧的俯仰角值。
- 新的俯仰角将用于物体的姿态更新逻辑，使其匹配当前的运动速度。

更新偏航



1. 当前偏航角的获取

- **节点：Cur Yaw**
  - 当前帧中物体的偏航角（Yaw）表示物体在水平平面上的旋转状态。

- 原理：物体的偏航角是物体当前姿态的一部分，由引擎中的旋转数据存储和管理。
- **作用**：
  - 为当前帧的计算提供基准偏航角。

2. 计算偏航角的增量

- **输入参数**：
  - **Delta Time X**：
    - 当前帧的时间增量，表示从上一帧到这一帧的时间间隔。
    - 原理：在引擎中，帧时间的长短可能不同，为了使旋转调整与帧率无关，必须乘以
      Delta Time 以保持增量的平滑性。
  - **Steering Speed**：
    - 控制偏航角调整的速度（每秒调整的最大角度）。
    - 原理：通过限制角度变化的速率，避免过快或过慢的调整。
  - **Steering**：
    - 偏航调整的方向和强度，通常来源于目标方向与当前方向的夹角。
    - 原理：控制物体应该向左（负值）还是向右（正值）旋转，以及旋转的幅度大小。
- **计算逻辑**：
  - 偏航角增量是帧时间、旋转速度和旋转强度的乘积。
  - **原理**：
    - 将旋转速度（每秒最大调整角度）乘以时间增量，得到当前帧的角度调整量。
    - 乘以旋转强度（Steering）进一步细化调整量，使其与目标方向的误差成正比。

3. 更新偏航角

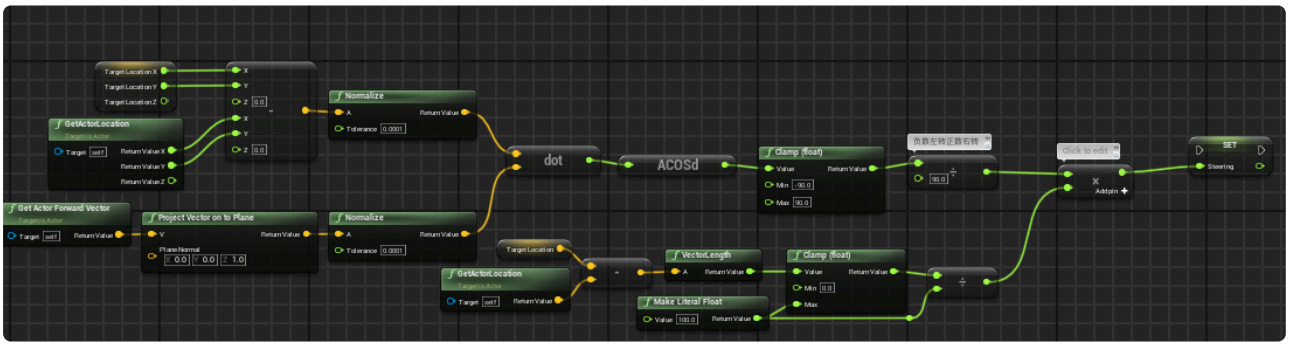
- **节点操作**：
  - 偏航角增量与当前偏航角相加，计算出新的偏航角。
  - **原理**：
    - 通过累积偏航角度，逐帧调整物体的旋转状态。
    - 每一帧的调整量是增量，新的偏航角逐渐接近目标方向。
  - **作用**：
    - 动态更新物体的朝向，确保其在每一帧中逐步向目标方向对齐。

4. 限制偏航角范围

- **取模操作（%）**：
  - 将新的偏航角度限制在 0 到 360 度之间。
  - **原理**：
    - 偏航角本质上是一个周期性的值，当超过 360 度时，应该回归到 0。
    - 模运算是数学中的一种取余操作，可以将任意角度归一化到指定范围。例如：
      - 偏航角 370 度通过模运算变为 10 度。
      - 偏航角 -10 度通过模运算变为 350 度。
  - **作用**：
    - 确保计算出的偏航角值有效且连续，不会因为过大或过小的值导致计算错误。

5. 更新变量

- **节点：Set Cur Yaw**
  - 将新的偏航角存储到 Cur Yaw 中。
  - **原理**：
    - 在每一帧中，新的偏航角将作为下一帧的初始角度，用于连续的计算。
  - **作用**：
    - 保持偏航角的更新状态，使旋转逻辑能够逐帧连续执行。



1. 计算目标方向

- **从目标位置和当前物体位置获取方向向量**：
  - 使用 GetActorLocation
    节点获取当前物体的位置和目标位置，将目标位置减去当前物体位置，得到一个方向向量（从当前物体指向目标）。
- **归一化目标方向向量**：
  - 使用 Normalize 节点将上述方向向量标准化，即将其长度调整为
    1，得到一个单位方向向量。

2. 计算当前方向

- **获取物体的前向向量**：

    - 使用 GetActorForwardVector
      节点获取物体当前的前向向量，表示物体在局部坐标系中的“正前方”方向。
- **投影到水平面**：
    - 使用 Project Vector onto Plane 节点，将当前前向向量投影到水平面（以 Z=0Z = 0 的法向量为标准），忽略垂直方向的影响，仅保留水平向量。
- **归一化前向向量**：
    - 对投影后的向量进行归一化处理，确保其为单位向量，便于与目标方向向量进行比较。

3. 计算方向夹角

- **点积计算方向关系**：
    - 使用 Dot Product
      节点计算当前前向向量和目标方向向量之间的点积。点积结果的范围是 [-1, 1]，用来描述两个向量的方向关系：
        - 11：完全同向。
        - 00：完全垂直。
        - −1-1：完全反向。
- **通过反余弦（ACOS）计算夹角**：
    - 使用 ACOS 节点，将点积值转换为夹角，输出结果为 0 至 180 度范围的角度值，表示当前方向与目标方向的角度偏差。

4. 确定调整强度

- **限制夹角范围**：
    - 使用 Clamp Angle 节点，将计算出的夹角值限制在一个合理范围内，例如 -30 度至 30 度。这样可以防止过大角度偏差导致物体旋转过猛。
- **计算比例调整**：
    - 结合夹角值，使用一个比例因子（通常与距离成比例）计算调整强度。这一步的作用是使得较大夹角产生更大的旋转调整。

5. 确定调整方向

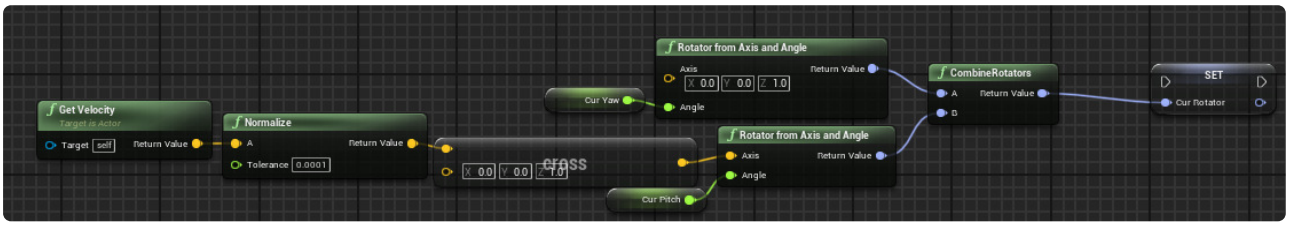
- **计算目标方向的相对位置**：
    - 通过点积或叉积计算目标方向相对于当前方向是在左侧还是右侧。
    - 例如，使用 Z 轴的符号确定方向：
        - 如果结果为正值，物体需要向右调整。
        - 如果结果为负值，物体需要向左调整。
- **结合方向调整夹角强度**：

- 使用调整方向的符号对夹角强度进行修正，使其为正值（右转）或负值（左转）。

6. 输出最终调整强度和方向

- **输出调整值**：
  - 将最终的夹角调整强度与方向输出为一个单一的变量（Steering），表示当前帧中物体需要旋转的强度和方向。
  - Steering 为负值时，表示需要向左旋转；为正值时，表示需要向右旋转。

更新旋转体



1. 获取物体的线性速度

- **节点：Get Velocity**
  - 获取物体当前的线性速度向量 Vactor\mathbf{V}_{\text{actor}}Vactor，表示物体在世界空间中沿 XXX、YYY、ZZZ 方向的速度。
  - 输出值是一个三维向量，用于后续计算。

2. 归一化速度向量

- **节点：Normalize**
  - 对获取的速度向量进行归一化处理，得到一个单位方向向量 Dvelocity\mathbf{D}_{\text{velocity}}Dvelocity，表示速度方向而不包含大小信息。
  - **作用**：
    - 确保后续计算中仅使用方向信息，而速度大小不会干扰旋转角度的计算。

3. 更新偏航角（Yaw）

- **节点：Rotator from Axis and Angle**
  - 通过旋转轴和角度计算偏航旋转器：
    - **旋转轴：(0,0,1)(0, 0, 1)(0,0,1)**：
      - 表示围绕 ZZZ-轴旋转。
    - **角度：Cur Yaw**：
      - 当前帧计算得到的偏航角，表示物体在水平平面上的旋转角度。

- 输出结果是一个仅包含偏航旋转的旋转器。

4. 更新俯仰角（Pitch）

- **节点：Rotator from Axis and Angle**
  - 通过旋转轴和角度计算俯仰旋转器：
    - **旋转轴：(1,0,0)(1, 0, 0)(1,0,0)**：
      - 表示围绕 XXX-轴旋转。
    - **角度：Cur Pitch**：
      - 当前帧计算得到的俯仰角，表示物体在垂直平面上的旋转角度。
  - 输出结果是一个仅包含俯仰旋转的旋转器。

5. 合并旋转器

- **节点：Combine Rotators**
  - 将偏航旋转器和俯仰旋转器合并，生成一个新的旋转器 Our Rotator。
  - **作用**：
    - 同时包含俯仰角和偏航角的旋转信息，用于表示物体的完整姿态。
  - **顺序**：
    - 先应用偏航旋转，再叠加俯仰旋转（旋转顺序会影响最终结果）。

6. 设置物体的旋转

- **节点：Set Our Rotator**
  - 将计算得到的旋转器赋值给物体，更新其姿态。
  - **作用**：
    - 将动态计算的俯仰和偏航角应用到物体的旋转状态。

关键知识点2： ThreadBP.py关键控制接口解析

1. 导入模块

```
import threading

import time

import struct

import socket

import DllSimCtrlAPI as DllSimCtrl
```

- threading：用于实现并发运行多个任务（比如发送和接收数据）。
- time：用于控制延迟，调节程序的运行速度。
- struct：用于将数据转换为特定的字节格式（如unpack用于解析接收到的数据）。
- socket：用于网络通信，这里用于UDP协议的套接字通信。
- DllSimCtrlAPI：自定义的DLL接口，用于与仿真系统进行交互，发送控制指令。

2. ThreadBP 类初始化（init）

class ThreadBP:

def _*init*_(self, CopterID):

DLL

self.DllSim = DllSimCtrl.DllSimCtrlAPI(CopterID)

info

self.CopterID = CopterID

self.Hz = 30 # 控制频率30Hz

self.CurPos = [0, 0, 0] # 初始位置

self.state = 0 # 无人机状态

SIL: 软件在环仿真输入

self.inSILInts = [0] * 8 # 整型输入

self.inSILFloats = [0] * 20 # 浮点型输入

控制命令标志

self.CmdEn = 1 # 控制启用

self.CmdSIL = 1 << 1 # 仿真模式

self.CmdArmed = 1 << 2 # 解锁命令

self.CmdTakeoff = 1 << 8 # 起飞命令

self.CmdPosition = 1 << 9 # 定点命令

self.CmdLand = 1 << 10 # 降落命令

self.CmdReturn = 1 << 11 # 返航命令

self.CmdOffboard_Pos = 1 << 16 # 固定翼的盘旋命令

self.CmdOffboard_Att = 1 << 17 # 定高模式

self.CmdHor = 1 << 14 # 水平位置控制

self.CmdOffboardPos = 1 << 16 # Offboard位置控制

self.CmdOffboardAtt = 1 << 17 # Offboard姿态控制

- self.DllSim：通过调用DllSimCtrlAPI的构造函数，创建与仿真系统的接口对象，使用CopterID来标识不同的无人机。
- self.CopterID：记录当前实例的无人机ID。
- self.Hz：控制系统的频率，设置为30Hz。
- self.CurPos：表示无人机当前位置，初始为[0, 0, 0]。
- self.state：表示无人机的当前状态，初始值为0（未激活）。
- self.inSILInts、self.inSILFloats：这两个列表分别存储要发送的整数和浮点数数据，用于控制信号。
- **命令标志**：各个命令（如起飞、定点、着陆等）通过位移操作定义（例如，CmdArmed是通过1
  << 2定义的，表示一个值的第3位）。这些命令用于控制无人机的不同操作。

3. 初始化MAVLink循环（InitMavLoop）

def InitMavLoop(self):

udp listen address IP+port

port = 30100 + self.CopterID * 2 - 1 # 根据无人机ID计算UDP端口

self.udp_socketTrue = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

self.udp_socketTrue.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1) # 设置广播选项

self.udp_socketTrue.bind(('0.0.0.0', port)) # 绑定端口

启动两个线程：发送控制信号和接收数据

self.thread1 = threading.Thread(target=self.SendSIL, args=())

self.thread1.start()

self.thread2 = threading.Thread(target=self.RecvSIL, args=())

self.thread2.start()

- self.udp_socketTrue：创建一个UDP套接字，用于监听来自仿真系统或控制台的UDP数据。
- self.udp_socketTrue.setsockopt：启用广播选项，允许数据包广播到网络。
- self.udp_socketTrue.bind：绑定UDP端口，监听来自指定端口的数据。
- 启动两个线程：
  - thread1：负责定期发送控制信号（通过SendSIL）。
  - thread2：负责接收来自仿真系统的数据（通过RecvSIL）。

4. 发送SIL数据（SendSIL）

def SendSIL(self):

while True:

self.DllSim.sendSILIntFloat(self.inSILInts, self.inSILFloats)

time.sleep(1 / self.Hz) # 控制发送频率

- 该方法在一个无限循环中工作，不断地调用DLL接口（DllSim.sendSILIntFloat）将整数和浮点数控制信号发送到仿真系统。
- time.sleep(1 / self.Hz)：控制发送频率，使得每秒发送30次数据（30Hz）。

5. 接收SIL数据（RecvSIL）

def RecvSIL(self):

while True:

buf, addr = self.udp_socketTrue.recvfrom(65500) # 接收数据包

if len(buf) == 24: # 如果数据包长度为24字节

BPData = struct.unpack('3i3f', buf) # 解包，格式为3个整数和3个浮点数

checksum = BPData[0]

if checksum == 1234567897: # 校验和检查

self.state = BPData[2] # 更新无人机状态

self.CurPos = BPData[3:5] # 更新位置

time.sleep(1 / self.Hz)

- recvfrom(65500)：接收UDP数据包。65500是数据包的最大长度，适合处理较大的数据包。
- struct.unpack('3i3f', buf)：解包接收到的字节数据，格式为3个整数和3个浮点数。
- 校验和检查：通过校验和来验证数据的完整性。如果校验和正确（1234567897），则更新状态和位置。

6. 控制命令方法

- **Position**：设置无人机的位置控制命令。

def Position(self, pos):

self.inSILInts[0] = self.CmdEn + self.CmdSIL + self.CmdArmed

self.inSILInts[1] = 1

self.inSILFloats[0:2] = pos

- **TakeOff**：设置无人机起飞命令，并指定目标高度。

def TakeOff(self, Height):

self.inSILInts[0] = self.CmdEn + self.CmdSIL + self.CmdArmed + self.CmdTakeoff

self.inSILFloats[0:2] = [0, 0, Height]

- **Landing**：设置无人机降落命令。

def Landing(self):

self.inSILInts[0] = self.CmdEn + self.CmdSIL + self.CmdArmed + self.CmdLand

- **Return**：设置返航命令。

def Return(self):

self.inSILInts[0] = self.CmdEn + self.CmdSIL + self.CmdArmed + self.CmdReturn

2.实验效果

3.文件目录

例程目录：[安装目录]\RflySimAPIs\4.RflySimModel\3.CustExps\e8_BlueprintCopter

文件夹/文件名称	说明
Python38Run.bat	python环境启动脚本
SITLPosStr.bat	软件在环启动脚本
demo.py	例程功能主程序
ThreadBP.py	

4.运行环境

4.1 软件要求

Windows 10及以上版本；RflySim工具链；VS Code。

> ①：若使用Pixhawk 6X飞控，平台安装时的编译命令为：px4_fmu-v6x_default，推荐PX4固件版本为：1.12.3。其他配套飞控及编译命令请见：https://rflysim.com/doc/zh/1/Hardware.html

4.2 硬件要求

笔记本/台式电脑① 1台。

> ①：推荐配置请见：https://rflysim.com/

5.实验步骤

蓝图飞机操控实验（必做）

Step 1：下载并导入蓝图综合模型

为了保证RflySim平台安装包的大小，本实验中所用到的蓝图控制器和蓝图飞机模型等较大文件均已上传至百度网盘中，请在实验前进行下载，
链接: https://pan.baidu.com/s/1rLliVf0FiXd_N79tYavL_w?pwd=vq93 提取码: vq93

下载完成后，进行解压放入【安装目录】\RflySim3D\RflySim3D\Content文件夹中。注：请勿修改文件夹名称。

Step 2：启动RflySim3D

双击 ue4.bat 启动一个RflySim3D

Step 3：运行python脚本加载并操纵蓝图飞机

```
C:\Windows\system32\cmd.ex
Python3.8 environment has been set with openCV+pymavlink+numpy+pyulog etc.
You can use pip or pip3 command to install other libraries
Put Python38Run.bat into your code folder
Use the command: 'python XXX.py' to run the script with Python

F:\d2\4.RflySimModel\3.CustExps\e7_ExternalSensors\e3_Opticalflow_UE>python Opticalflow.py
```

在文件夹下，双击Python38Run.bat，打开集成好的环境，输入python demo.py，回车运行。

1. **Vscode调试运行实验（选做）**

准备工作：

- 先确保已经按RflySimAPIs\1.RflySimIntro\2.AdvExps\e3_Config\Readme.pdf步骤，正确配置VS Code环境。或者配置了自己的Pycharm等自定义环境。
- 其他步骤与上文相同，在运行文件时，可使用VS Code（或Pycharm等工具）来打开文件文件，并阅读代码，修改代码，调试执行等。

扩展实验：

- 请自行使用VS Code阅读例程中的源码，通过程序跳转，了解每条代码的执行原理；再通过调试工具，验证每条指令的执行效果。

6.参考资料

[1]. [安装目录]\RflySimAPIs\3.RflySim3DUE\API.pdf

[2]. UE4性能调试分析常用方法 - 知乎 : https://zhuanlan.zhihu.com/p/273608458

[3].

7.常见问题

Q1：***

A1：***